

# T81-558: Applications of Deep Neural Networks

## Module 1: Python Preliminaries

- Instructor: [Jeff Heaton](#), McKelvey School of Engineering, [Washington University in St. Louis](#)
- For more information visit the [class website](#).

## Module 1 Material

- **Part 1.1: Course Overview** [\[Video\]](#) [\[Notebook\]](#)
- Part 1.2: Introduction to Python [\[Video\]](#) [\[Notebook\]](#)
- Part 1.3: Python Lists, Dictionaries, Sets and JSON [\[Video\]](#) [\[Notebook\]](#)
- Part 1.4: File Handling [\[Video\]](#) [\[Notebook\]](#)
- Part 1.5: Functions, Lambdas, and Map/Reduce [\[Video\]](#) [\[Notebook\]](#)

Watch one (or more) of these depending on how you want to setup your Python TensorFlow environment:

- [How to Submit a Module Assignment locally](#)
- [How to Use Google CoLab and Submit Assignment](#)
- [Installing TensorFlow, Keras, and Python in Windows CPU or GPU](#)
- [Installing TensorFlow, Keras, and Python for an Intel Mac](#)
- [Installing TensorFlow, Keras, and Python for an M1 Mac](#)

## Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

```
In [1]: try:  
        from google.colab import drive  
        %tensorflow_version 2.x  
        COLAB = True  
        print("Note: using Google CoLab")  
    except:  
        print("Note: not using Google CoLab")  
        COLAB = False
```

Note: not using Google CoLab

## Part 1.1: Overview

Deep learning is a group of exciting new technologies for neural networks.

[Cite:lecun2015deep] By using a combination of advanced training techniques neural network architectural components, it is now possible to train neural networks of much greater complexity. This book introduces the reader to deep neural networks, regularization units (ReLU), convolution neural networks, and recurrent neural networks. High-performance computing (HPC) aspects demonstrate how deep learning can be leveraged both on graphical processing units (GPUs), as well as grids. Deep learning allows a model to learn hierarchies of information in a way that is similar to the function of the human brain. The focus is primarily upon the application of deep learning, with some introduction to the mathematical foundations of deep learning. Readers will make use of the Python programming language to architect a deep learning model for several real-world data sets and interpret the results of these networks.

[Cite:goodfellow2016deep]

## Origins of Deep Learning

Neural networks are one of the earliest examples of a machine learning model. Neural networks were initially introduced in the 1940s and have risen and fallen several times in popularity. The current generation of deep learning began in 2006 with an improved training algorithm by Geoffrey Hinton.

[Cite:hinton2006fast] This technique finally allowed neural networks with many layers (deep neural networks) to be efficiently trained. Four researchers have contributed significantly to the development of neural networks. They have consistently pushed neural network research, both through the ups and downs. These four luminaries are shown in Figure 1.LUM.

**Figure 1.LUM: Neural Network Luminaries**



The current luminaries of artificial neural network (ANN) research and ultimately deep learning, in order as appearing in the figure:

- [Yann LeCun](#), Facebook and New York University - Optical character recognition and computer vision using convolutional neural networks (CNN). The founding father of convolutional nets.
- [Geoffrey Hinton](#), Google and University of Toronto. Extensive work on neural networks. Creator of deep learning and early adapter/creator of backpropagation for neural networks.
- [Yoshua Bengio](#), University of Montreal and Botler AI. Extensive research into deep learning, neural networks, and machine learning.
- [Andrew Ng](#), Badiu and Stanford University. Extensive research into deep learning, neural networks, and application to robotics.

Geoffrey Hinton, Yann LeCun, and Yoshua Bengio won the [Turing Award](#) for their contributions to deep learning.

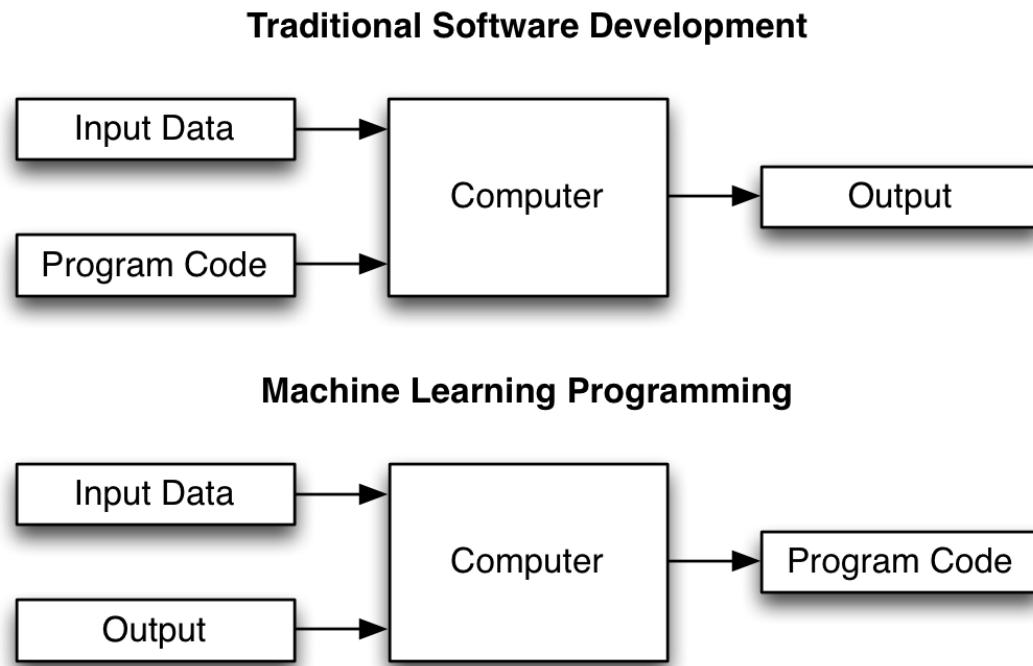
## What is Deep Learning

The focus of this book is deep learning, which is a prevalent type of machine learning that builds upon the original neural networks popularized in the 1980s. There is very little difference between how a deep neural network is calculated compared with the first neural network. We've always been able to create and calculate deep neural networks. A deep neural network is nothing more than a neural network with many layers. While we've always been able to

create/calculate deep neural networks, we've lacked an effective means of training them. Deep learning provides an efficient means to train deep neural networks.

If deep learning is a type of machine learning, this begs the question, "What is machine learning?" Figure 1.ML-DEV illustrates how machine learning differs from traditional software development.

**Figure 1.ML-DEV: ML vs Traditional Software Development**



- **Traditional Software Development** - Programmers create programs that specify how to transform input into the desired output.
- **Machine Learning** - Programmers create models that can learn to produce the desired output for given input. This learning fills the traditional role of the computer program.

Researchers have applied machine learning to many different areas. This class explores three specific domains for the application of deep neural networks, as illustrated in Figure 1.ML-DOM.

**Figure 1.ML-DOM: Application of Machine Learning**



- **Computer Vision** - The use of machine learning to detect patterns in visual data. For example, is an image a picture of a cat or a dog.
- **Tabular Data** - Several named input values allow the neural network to predict another named value that becomes the output. For example, we are using four measurements of iris flowers to predict the species. This type of data is often called tabular data.
- **Natural Language Processing (NLP)** - Deep learning transformers have revolutionized NLP, allowing text sequences to generate more text, images, or classifications.
- **Reinforcement Learning** - Reinforcement learning trains a neural network to choose ongoing actions so that the algorithm rewards the neural network for optimally completing a task.
- **Time Series** - The use of machine learning to detect patterns in time. Typical time series applications are financial applications, speech recognition, and even natural language processing (NLP).
- **Generative Models** - Neural networks can learn to produce new original synthetic data from input. We will examine StyleGAN, which learns to create new images similar to those it saw during training.

## Regression, Classification and Beyond

Machine learning research looks at problems in broad terms of supervised and unsupervised learning. Supervised learning occurs when you know the correct outcome for each item in the training set. On the other hand, unsupervised learning utilizes training sets where no correct outcome is known. Deep learning supports both supervised and unsupervised learning; however, it also adds reinforcement and adversarial learning. Reinforcement learning teaches the neural network to carry out actions based on an environment. Adversarial learning pits two neural networks against each other to learn when the data provides no correct outcomes. Researchers continue to add new deep learning training techniques.

Machine learning practitioners usually divide supervised learning into classification and regression. Classification networks might accept financial data and classify the investment risk as risk or safe. Similarly, a regression neural network outputs a number and might take the same data and return a risk score. Additionally, neural networks can output multiple regression and classification scores simultaneously.

One of the most powerful aspects of neural networks is that the input and output of a neural network can be of many different types, such as:

- An image
- A series of numbers that could represent text, audio, or another time series
- A regression number
- A classification class

## Why Deep Learning?

For tabular data, neural networks often do not perform significantly better than other models, such as:

- [Support Vector Machines](#)
- [Random Forests](#)
- [Gradient Boosted Machines](#)

Like these other models, neural networks can perform both **classification** and **regression**. When applied to relatively low-dimensional tabular data tasks, deep neural networks do not necessarily add significant accuracy over other model types. However, most state-of-the-art solutions depend on deep neural networks for images, video, text, and audio data.

## Python for Deep Learning

We will utilize the Python 3.x programming language for this book. Python has some of the widest support for deep learning as a programming language. The two most popular frameworks for deep learning in Python are:

- [TensorFlow/Keras](#) (Google)
- [PyTorch](#) (Facebook)

This book focuses primarily upon Keras, with some applications in PyTorch. For many tasks, we will utilize Keras directly. We will utilize third-party libraries for higher-level tasks, such as reinforcement learning, generative adversarial neural networks, and others. These third-party libraries may internally make use of

either PyTorch or Keras. I chose these libraries based on popularity and application, not whether they used PyTorch or Keras.

To successfully use this book, you must be able to compile and execute Python code that makes use of TensorFlow for deep learning. There are two options for you to accomplish this:

- Install Python, TensorFlow and some IDE (Jupyter, TensorFlow, and others).
- Use [Google CoLab](#) in the cloud, with free GPU access.

If you look at this notebook on Github, near the top of the document, there are links to videos that describe how to use Google CoLab. There are also videos explaining how to install Python on your local computer. The following sections take you through the process of installing Python on your local computer. This process is essentially the same on Windows, Linux, or Mac. For specific OS instructions, refer to one of the tutorial YouTube videos earlier in this document.

To install Python on your computer, complete the following instructions:

- [Installing Python and TensorFlow - Windows/Linux](#)
- [Installing Python and TensorFlow - Mac Intel](#)
- [Installing Python and TensorFlow - Mac M1](#)

## Check your Python Installation

Once you've installed Python, you can utilize the following code to check your Python and library versions. If you have a GPU, you can also check to see that Keras recognize it.

```
In [3]: # What version of Python do you have?
import sys

import tensorflow.keras
import pandas as pd
import sklearn as sk
import tensorflow as tf

check_gpu = len(tf.config.list_physical_devices('GPU')) > 0

print(f"Tensor Flow Version: {tf.__version__}")
print(f"Keras Version: {tensorflow.keras.__version__}")
print()
print(f"Python {sys.version}")
print(f"Pandas {pd.__version__}")
print(f"Scikit-Learn {sk.__version__}")
print("GPU is", "available" if check_gpu \
      else "NOT AVAILABLE")
```

Tensor Flow Version: 2.1.0

Keras Version: 2.2.4-tf

Python 3.7.16 (default, Jan 17 2023, 22:20:44)

[GCC 11.2.0]

Pandas 1.3.5

Scikit-Learn 1.0.2

GPU is NOT AVAILABLE

# Module 1 Assignment

You can find the first assignment here: [assignment 1](#)

# T81-558: Applications of Deep Neural Networks

## Module 1: Python Preliminaries

- Instructor: [Jeff Heaton](#), McKelvey School of Engineering, [Washington University in St. Louis](#)
- For more information visit the [class website](#).

## Module 1 Material

- Part 1.1: Course Overview [\[Video\]](#) [\[Notebook\]](#)
- **Part 1.2: Introduction to Python** [\[Video\]](#) [\[Notebook\]](#)
- Part 1.3: Python Lists, Dictionaries, Sets and JSON [\[Video\]](#) [\[Notebook\]](#)
- Part 1.4: File Handling [\[Video\]](#) [\[Notebook\]](#)
- Part 1.5: Functions, Lambdas, and Map/Reduce [\[Video\]](#) [\[Notebook\]](#)

## Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

```
In [1]: try:  
    from google.colab import drive  
    %tensorflow_version 2.x  
    COLAB = True  
    print("Note: using Google CoLab")  
except:  
    print("Note: not using Google CoLab")  
    COLAB = False
```

Note: not using Google CoLab

## Part 1.2: Introduction to Python

Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python's design philosophy emphasizes code readability with its notable use of significant whitespace. Its language constructs and object-oriented approach aim to help

programmers write clear, logical code for small and large-scale projects. Python has become a common language for machine learning research and is the primary language for TensorFlow.

Python 3.0, released in 2008, was a significant revision of the language that is not entirely backward-compatible, and much Python 2 code does not run unmodified on Python 3. This course makes use of Python 3. Furthermore, TensorFlow is not compatible with versions of Python earlier than 3. A non-profit organization, the Python Software Foundation (PSF), manages and directs resources for Python development. On January 1, 2020, the PSF discontinued the Python 2 language and no longer provides security patches and other improvements. Python interpreters are available for many operating systems.

The first two modules of this course provide an introduction to some aspects of the Python programming language. However, entire books focus on Python. Two modules will not cover every detail of this language. The reader is encouraged to consult additional sources on the Python language.

Like most tutorials, we will begin by printing Hello World.

```
In [2]: print("Hello World")
```

```
Hello World
```

The above code passes a constant string, containing the text "hello world" to a function that is named print.

You can also leave comments in your code to explain what you are doing. Comments can begin anywhere in a line.

```
In [3]: # Single line comment (this has no effect on your program)
print("Hello World") # Say hello
```

```
Hello World
```

Strings are very versatile and allow your program to process textual information. Constant string, enclosed in quotes, define literal string values inside your program. Sometimes you may wish to define a larger amount of literal text inside of your program. This text might consist of multiple lines. The triple quote allows for multiple lines of text.

```
In [4]: print("""Print
Multiple
Lines
""")
```

## Print Multiple Lines

Like many languages Python uses single (' ) and double (" ) quotes interchangeably to denote literal string constants. The general convention is that double quotes should enclose actual text, such as words or sentences. Single quotes should enclose symbolic text, such as error codes. An example of an error code might be 'HTTP404'.

However, there is no difference between single and double quotes in Python, and you may use whichever you like. The following code makes use of a single quote.

```
In [5]: print('Hello World')
```

```
Hello World
```

In addition to strings, Python allows numbers as literal constants in programs. Python includes support for floating-point, integer, complex, and other types of numbers. This course will not make use of complex numbers. Unlike strings, quotes do not enclose numbers.

The presence of a decimal point differentiates floating-point and integer numbers. For example, the value 42 is an integer. Similarly, 42.5 is a floating-point number. If you wish to have a floating-point number, without a fraction part, you should specify a zero fraction. The value 42.0 is a floating-point number, although it has no fractional part. As an example, the following code prints two numbers.

```
In [6]: print(42)  
print(42.5)
```

```
42  
42.5
```

So far, we have only seen how to define literal numeric and string values. These literal values are constant and do not change as your program runs. Variables allow your program to hold values that can change as the program runs.

Variables have names that allow you to reference their values. The following code assigns an integer value to a variable named "a" and a string value to a variable named "b."

```
In [7]: a = 10  
b = "ten"  
print(a)  
print(b)
```

```
10  
ten
```

The key feature of variables is that they can change. The following code demonstrates how to change the values held by variables.

```
In [8]: a = 10  
print(a)  
a = a + 1  
print(a)
```

10

11

You can mix strings and variables for printing. This technique is called a formatted or interpolated string. The variables must be inside of the curly braces. In Python, this type of string is generally called an f-string. The f-string is denoted by placing an "f" just in front of the opening single or double quote that begins the string. The following code demonstrates the use of an f-string to mix several variables with a literal string.

```
In [4]: a = 9  
print(f'The value of a is {a}')
```

The value of a is 9

You can also use f-strings with math (called an expression). Curly braces can enclose any valid Python expression for printing. The following code demonstrates the use of an expression inside of the curly braces of an f-string.

```
In [10]: a = 10  
print(f'The value of a plus 5 is {a+5}')
```

The value of a plus 5 is 15

Python has many ways to print numbers; these are all correct. However, for this course, we will use f-strings. The following code demonstrates some of the varied methods of printing numbers in Python.

```
In [5]: a = 5  
  
print(f'a is {a}') # Preferred method for this course.  
print('a is {}'.format(a))  
print('a is ' + str(a))  
print('a is %d' % (a))
```

a is 5  
a is 5  
a is 5  
a is 5

You can use if-statements to perform logic. Notice the indents? These if-statements are how Python defines blocks of code to execute together. A block usually begins after a colon and includes any lines at the same level of indent. Unlike many other programming languages, Python uses whitespace to define

blocks of code. The fact that whitespace is significant to the meaning of program code is a frequent source of annoyance for new programmers of Python. Tabs and spaces are both used to define the scope in a Python program. Mixing both spaces and tabs in the same program is not recommended.

```
In [12]: a = 5
if a>5:
    print('The variable a is greater than 5.')
else:
    print('The variable a is not greater than 5')
```

The variable a is not greater than 5

The following if-statement has multiple levels. It can be easy to indent these levels improperly, so be careful. This code contains a nested if-statement under the first "a==5" if-statement. Only if a is equal to 5 will the nested "b==6" if-statement be executed. Also, note that the "elif" command means "else if."

```
In [13]: a = 5
b = 6

if a==5:
    print('The variable a is 5')
    if b==6:
        print('The variable b is also 6')
elif a==6:
    print('The variable a is 6')
```

The variable a is 5  
The variable b is also 6

It is also important to note that the double equal ("==") operator is used to test the equality of two expressions. The single equal ("=") operator is only used to assign values to variables in Python. The greater than (">"), less than ("<"), greater than or equal (">="), less than or equal ("<=") all perform as would generally be accepted. Testing for inequality is performed with the not equal ("!=") operator.

It is common in programming languages to loop over a range of numbers. Python accomplishes this through the use of the **range** operation. Here you can see a **for** loop and a **range** operation that causes the program to loop between 1 and 3.

```
In [14]: for x in range(1, 3): # If you ever see xrange, you are in Python 2
          print(x)
# If you ever see print x (no parenthesis), you are in Python 2
```

1  
2

This code illustrates some incompatibilities between Python 2 and Python 3. Before Python 3, it was acceptable to leave the parentheses off of a *print* function call. This method of invoking the *print* command is no longer allowed in Python 3. Similarly, it used to be a performance improvement to use the *xrange* command in place of range command at times. Python 3 incorporated all of the functionality of the *xrange* Python 2 command into the normal *range* command. As a result, the programmer should not use the *xrange* command in Python 3. If you see either of these constructs used in example code, then you are looking at an older Python 2 era example.

The *range* command is used in conjunction with loops to pass over a specific range of numbers. Cases, where you must loop over specific number ranges, are somewhat uncommon. Generally, programmers use loops on collections of items, rather than hard-coding numeric values into your code. Collections, as well as the operations that loops can perform on them, is covered later in this module.

The following is a further example of a looped printing of strings and numbers.

```
In [15]: acc = 0
for x in range(1, 3):
    acc += x
    print(f"Adding {x}, sum so far is {acc}")

print(f"Final sum: {acc}")

Adding 1, sum so far is 1
Adding 2, sum so far is 3
Final sum: 3
```



# T81-558: Applications of Deep Neural Networks

## Module 1: Python Preliminaries

- Instructor: [Jeff Heaton](#), McKelvey School of Engineering, [Washington University in St. Louis](#)
- For more information visit the [class website](#).

## Module 1 Material

- Part 1.1: Course Overview [\[Video\]](#) [\[Notebook\]](#)
- Part 1.2: Introduction to Python [\[Video\]](#) [\[Notebook\]](#)
- **Part 1.3: Python Lists, Dictionaries, Sets and JSON** [\[Video\]](#) [\[Notebook\]](#)
- Part 1.4: File Handling [\[Video\]](#) [\[Notebook\]](#)
- Part 1.5: Functions, Lambdas, and Map/Reduce [\[Video\]](#) [\[Notebook\]](#)

## Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

```
In [1]: try:  
    from google.colab import drive  
    %tensorflow_version 2.x  
    COLAB = True  
    print("Note: using Google CoLab")  
  
except:  
    print("Note: not using Google CoLab")  
    COLAB = False
```

Note: not using Google CoLab

## Part 1.3: Python Lists, Dictionaries, Sets, and JSON

Like most modern programming languages, Python includes Lists, Sets, Dictionaries, and other data structures as built-in types. The syntax appearance

of both of these is similar to JSON. Python and JSON compatibility is discussed later in this module. This course will focus primarily on Lists, Sets, and Dictionaries. It is essential to understand the differences between these three fundamental collection types.

- **Dictionary** - A dictionary is a mutable unordered collection that Python indexes with name and value pairs.
- **List** - A list is a mutable ordered collection that allows duplicate elements.
- **Set** - A set is a mutable unordered collection with no duplicate elements.
- **Tuple** - A tuple is an immutable ordered collection that allows duplicate elements.

Most Python collections are mutable, meaning the program can add and remove elements after definition. An immutable collection cannot add or remove items after definition. It is also essential to understand that an ordered collection means that items maintain their order as the program adds them to a collection. This order might not be any specific ordering, such as alphabetic or numeric.

Lists and tuples are very similar in Python and are often confused. The significant difference is that a list is mutable, but a tuple isn't. So, we include a list when we want to contain similar items and a tuple when we know what information goes into it ahead of time.

Many programming languages contain a data collection called an array. The array type is noticeably absent in Python. Generally, the programmer will use a list in place of an array in Python. Arrays in most programming languages were fixed-length, requiring the program to know the maximum number of elements needed ahead of time. This restriction leads to the infamous array-overrun bugs and security issues. The Python list is much more flexible in that the program can dynamically change the size of a list.

The next sections will look at each collection type in more detail.

## Lists and Tuples

For a Python program, lists and tuples are very similar. Both lists and tuples hold an ordered collection of items. It is possible to get by as a programmer using only lists and ignoring tuples.

The primary difference that you will see syntactically is that a list is enclosed by square braces [], and a tuple is enclosed by parenthesis (). The following code defines both list and tuple.

```
In [1]: l = ['a', 'b', 'c', 'd']
t = ('a', 'b', 'c', 'd')
```

```
print(l)
print(t)

['a', 'b', 'c', 'd']
('a', 'b', 'c', 'd')
```

The primary difference you will see programmatically is that a list is mutable, which means the program can change it. A tuple is immutable, which means the program cannot change it. The following code demonstrates that the program can change a list. This code also illustrates that Python indexes lists starting at element 0. Accessing element one modifies the second element in the collection. One advantage of tuples over lists is that tuples are generally slightly faster to iterate over than lists.

```
In [2]: l[1] = 'chaged'
#t[1] = 'changed' # This would result in an error

print(l)

['a', 'chaged', 'c', 'd']
```

Like many languages, Python has a for-each statement. This statement allows you to loop over every element in a collection, such as a list or a tuple.

```
In [4]: # Iterate over a collection.
for s in l:
    print(s)
```

a  
changed  
c  
d

The **enumerate** function is useful for enumerating over a collection and having access to the index of the element that we are currently on.

```
In [5]: # Iterate over a collection, and know where your index. (Python is zero-based)
for i,l in enumerate(l):
    print(f"{i}:{l}")

0:a
1:changed
2:c
3:d
```

A **list** can have multiple objects added, such as strings. Duplicate values are allowed. **Tuples** do not allow the program to add additional objects after definition.

```
In [6]: # Manually add items, lists allow duplicates
c = []
c.append('a')
c.append('b')
```

```
c.append('c')
c.append('c')
print(c)

['a', 'b', 'c', 'c']
```

Ordered collections, such as lists and tuples, allow you to access an element by its index number, as done in the following code. Unordered collections, such as dictionaries and sets, do not allow the program to access them in this way.

```
In [7]: print(c[1])
```

```
b
```

A **list** can have multiple objects added, such as strings. Duplicate values are allowed. Tuples do not allow the program to add additional objects after definition. The programmer must specify an index for the insert function, an index. These operations are not allowed for tuples because they would result in a change.

```
In [8]: # Insert
c = ['a', 'b', 'c']
c.insert(0, 'a0')
print(c)
# Remove
c.remove('b')
print(c)
# Remove at index
del c[0]
print(c)
```

```
['a0', 'a', 'b', 'c']
['a0', 'a', 'c']
['a', 'c']
```

## Sets

A Python **set** holds an unordered collection of objects, but sets do *not* allow duplicates. If a program adds a duplicate item to a set, only one copy of each item remains in the collection. Adding a duplicate item to a set does not result in an error. Any of the following techniques will define a set.

```
In [9]: s = set()
s = { 'a', 'b', 'c'}
s = set(['a', 'b', 'c'])
print(s)
```

```
{'c', 'a', 'b'}
```

A **list** is always enclosed in square braces [], a **tuple** in parenthesis (), and similarly a **set** is enclosed in curly braces. Programs can add items to a **set** as they run. Programs can dynamically add items to a **set** with the **add** function. It

is important to note that the **append** function adds items to lists, whereas the **add** function adds items to a **set**.

```
In [5]: # Manually add items, sets do not allow duplicates
# Sets add, lists append. I find this annoying.
c = set()
c.add('a')
c.add('b')
c.add('c')
c.add('c')
c.add('gc')
print(c)

{'a', 'b', 'gc', 'c'}
```

## Maps/Dictionaries/Hash Tables

Many programming languages include the concept of a map, dictionary, or hash table. These are all very related concepts. Python provides a dictionary that is essentially a collection of name-value pairs. Programs define dictionaries using curly braces, as seen here.

```
In [8]: d = {'name': "Jeff", 'address': "123 Main"}
print(d)
print(d['name'])

if 'name' in d:
    print("Name is defined")
    print(d['name'])
    print(d['address'])

if 'age' in d:
    print("age defined")
else:
    print("age undefined")

{'name': 'Jeff', 'address': '123 Main'}
Jeff
Name is defined
Jeff
123 Main
age undefined
```

Be careful that you do not attempt to access an undefined key, as this will result in an error. You can check to see if a key is defined, as demonstrated above. You can also access the dictionary and provide a default value, as the following code demonstrates.

```
In [12]: d.get('unknown_key', 'default')

Out[12]: 'default'
```

You can also access the individual keys and values of a dictionary.

```
In [13]: d = {'name': "Jeff", 'address': "123 Main"}  
# All of the keys  
print(f"Key: {d.keys()}")  
  
# All of the values  
print(f"Values: {d.values()}")
```

```
Key: dict_keys(['name', 'address'])  
Values: dict_values(['Jeff', '123 Main'])
```

Dictionaries and lists can be combined. This syntax is closely related to [JSON](#).

Dictionaries and lists together are a good way to build very complex data structures. While Python allows quotes ("") and apostrophe ('') for strings, JSON only allows double-quotes (""). We will cover JSON in much greater detail later in this module.

The following code shows a hybrid usage of dictionaries and lists.

```
In [14]: # Python list & map structures  
customers = [  
    {"name": "Jeff & Tracy Heaton", "pets": ["Wynton", "Cricket",  
        "Hickory"]},  
    {"name": "John Smith", "pets": ["rover"]},  
    {"name": "Jane Doe"}  
]  
  
print(customers)  
  
for customer in customers:  
    print(f"{customer['name']}: {customer.get('pets', 'no pets')}")
```

```
[{'name': 'Jeff & Tracy Heaton', 'pets': ['Wynton', 'Cricket', 'Hickory']},  
 {'name': 'John Smith', 'pets': ['rover']}, {'name': 'Jane Doe'}]  
Jeff & Tracy Heaton: ['Wynton', 'Cricket', 'Hickory']  
John Smith: ['rover']  
Jane Doe: no pets
```

The variable **customers** is a list that holds three dictionaries that represent customers. You can think of these dictionaries as records in a table. The fields in these individual records are the keys of the dictionary. Here the keys **name** and **pets** are fields. However, the field **pets** holds a list of pet names. There is no limit to how deep you might choose to nest lists and maps. It is also possible to nest a map inside of a map or a list inside of another list.

## More Advanced Lists

Several advanced features are available for lists that this section introduces. One such function is **zip**. Two lists can be combined into a single list by the **zip**

command. The following code demonstrates the **zip** command.

```
In [9]: a = [1,2,3,4,5]
b = [5,4,3,2,1]

print(zip(a,b))
```

```
<zip object at 0x7fc5d8419e60>
```

To see the results of the **zip** function, we convert the returned zip object into a list. As you can see, the **zip** function returns a list of tuples. Each tuple represents a pair of items that the function zipped together. The order in the two lists was maintained.

```
In [12]: a = [1,2,3,4,5,9]
b = [5,4,3,2,1,3]

print(list(zip(a,b)))
```

```
[(1, 5), (2, 4), (3, 3), (4, 2), (5, 1), (9, 3)]
```

The usual method for using the **zip** command is inside of a for-loop. The following code shows how a for-loop can assign a variable to each collection that the program is iterating.

```
In [17]: a = [1,2,3,4,5]
b = [5,4,3,2,1]

for x,y in zip(a,b):
    print(f'{x} - {y}')
```

```
1 - 5
2 - 4
3 - 3
4 - 2
5 - 1
```

Usually, both collections will be of the same length when passed to the **zip** command. It is not an error to have collections of different lengths. As the following code illustrates, the **zip** command will only process elements up to the length of the smaller collection.

```
In [18]: a = [1,2,3,4,5]
b = [5,4,3]

print(list(zip(a,b)))
```

```
[(1, 5), (2, 4), (3, 3)]
```

Sometimes you may wish to know the current numeric index when a for-loop is iterating through an ordered collection. Use the **enumerate** command to track the index location for a collection element. Because the **enumerate** command

deals with numeric indexes of the collection, the zip command will assign arbitrary indexes to elements from unordered collections.

Consider how you might construct a Python program to change every element greater than 5 to the value of 5. The following program performs this transformation. The enumerate command allows the loop to know which element index it is currently on, thus allowing the program to be able to change the value of the current element of the collection.

```
In [19]: a = [2, 10, 3, 11, 10, 3, 2, 1]
for i, x in enumerate(a):
    if x>5:
        a[i] = 5
print(a)
```

[2, 5, 3, 5, 5, 3, 2, 1]

The comprehension command can dynamically build up a list. The comprehension below counts from 0 to 9 and adds each value (multiplied by 10) to a list.

```
In [20]: lst = [x*10 for x in range(10)]
print(lst)
```

[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]

A dictionary can also be a comprehension. The general format for this is:

```
dict_variable = {key:value for (key,value) in
                 dictionary.items()}
```

A common use for this is to build up an index to symbolic column names.

```
In [21]: text = ['col-zero','col-one', 'col-two', 'col-three']
lookup = {key:value for (value,key) in enumerate(text)}
print(lookup)
```

{'col-zero': 0, 'col-one': 1, 'col-two': 2, 'col-three': 3}

This can be used to easily find the index of a column by name.

```
In [22]: print(f'The index of "col-two" is {lookup["col-two"]}')
```

The index of "col-two" is 2

## An Introduction to JSON

Data stored in a CSV file must be flat; it must fit into rows and columns. Most people refer to this type of data as structured or tabular. This data is tabular because the number of columns is the same for every row. Individual rows may

be missing a value for a column; however, these rows still have the same columns.

This data is convenient for machine learning because most models, such as neural networks, also expect incoming data to be of fixed dimensions. Real-world information is not always so tabular. Consider if the rows represent customers. These people might have multiple phone numbers and addresses. How would you describe such data using a fixed number of columns? It would be useful to have a list of these courses in each row that can be variable length for each row or student.

JavaScript Object Notation (JSON) is a standard file format that stores data in a hierarchical format similar to eXtensible Markup Language (XML). JSON is nothing more than a hierarchy of lists and dictionaries. Programmers refer to this sort of data as semi-structured data or hierarchical data. The following is a sample JSON file.

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    },
    {
      "type": "mobile",
      "number": "123 456-7890"
    }
  ],
  "children": [],
  "spouse": null
}
```

The above file may look somewhat like Python code. You can see curly braces that define dictionaries and square brackets that define lists. JSON does require

there to be a single root element. A list or dictionary can fulfill this role. JSON requires double-quotes to enclose strings and names. Single quotes are not allowed in JSON.

JSON files are always legal JavaScript syntax. JSON is also generally valid as Python code, as demonstrated by the following Python program.

```
In [23]: jsonHardCoded = {  
    "firstName": "John",  
    "lastName": "Smith",  
    "isAlive": True,  
    "age": 27,  
    "address": {  
        "streetAddress": "21 2nd Street",  
        "city": "New York",  
        "state": "NY",  
        "postalCode": "10021-3100"  
    },  
    "phoneNumbers": [  
        {  
            "type": "home",  
            "number": "212 555-1234"  
        },  
        {  
            "type": "office",  
            "number": "646 555-4567"  
        },  
        {  
            "type": "mobile",  
            "number": "123 456-7890"  
        }  
    ],  
    "children": [],  
    "spouse": None  
}
```

Generally, it is better to read JSON from files, strings, or the Internet than hard coding, as demonstrated here. However, for internal data structures, sometimes such hard-coding can be useful.

Python contains support for JSON. When a Python program loads a JSON the root list or dictionary is returned, as demonstrated by the following code.

```
In [24]: import json  
  
json_string = '{"first":"Jeff","last":"Heaton"}'  
obj = json.loads(json_string)  
print(f"First name: {obj['first']}")  
print(f"Last name: {obj['last']}")
```

```
First name: Jeff  
Last name: Heaton
```

Python programs can also load JSON from a file or URL.

```
In [25]: import requests  
  
r = requests.get("https://raw.githubusercontent.com/jeffheaton/"  
                  +"t81_558_deep_learning/master/person.json")  
print(r.json())
```

```
{'firstName': 'John', 'lastName': 'Smith', 'isAlive': True, 'age': 27, 'address': {'streetAddress': '21 2nd Street', 'city': 'New York', 'state': 'NY', 'postalCode': '10021-3100'}, 'phoneNumbers': [{'type': 'home', 'number': '212 555-1234'}, {'type': 'office', 'number': '646 555-4567'}, {'type': 'mobile', 'number': '123 456-7890'}], 'children': [], 'spouse': None}
```

Python programs can easily generate JSON strings from Python objects of dictionaries and lists.

```
In [26]: python_obj = {"first": "Jeff", "last": "Heaton"}  
print(json.dumps(python_obj))  
  
{"first": "Jeff", "last": "Heaton"}
```

A data scientist will generally encounter JSON when they access web services to get their data. A data scientist might use the techniques presented in this section to convert the semi-structured JSON data into tabular data for the program to use with a model such as a neural network.

```
In [ ]:
```

# T81-558: Applications of Deep Neural Networks

## Module 1: Python Preliminaries

- Instructor: [Jeff Heaton](#), McKelvey School of Engineering, [Washington University in St. Louis](#)
- For more information visit the [class website](#).

## Module 1 Material

- Part 1.1: Course Overview [\[Video\]](#) [\[Notebook\]](#)
- Part 1.2: Introduction to Python [\[Video\]](#) [\[Notebook\]](#)
- Part 1.3: Python Lists, Dictionaries, Sets and JSON [\[Video\]](#) [\[Notebook\]](#)
- **Part 1.4: File Handling** [\[Video\]](#) [\[Notebook\]](#)
- Part 1.5: Functions, Lambdas, and Map/Reduce [\[Video\]](#) [\[Notebook\]](#)

## Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow. Running the following code will map your GDrive to `/content/drive`.

```
In [ ]: try:  
    from google.colab import drive  
    drive.mount('/content/drive', force_remount=True)  
    COLAB = True  
    print("Note: using Google CoLab")  
    %tensorflow_version 2.x  
except:  
    print("Note: not using Google CoLab")  
    COLAB = False
```

## Part 1.4: File Handling

Files often contain the data that you use to train your AI programs. Once trained, your models may use real-time data to form predictions. These predictions might

be made on files too. Regardless of predicting or training, file processing is a vital skill for the AI practitioner.

There are many different types of files that you must process as an AI practitioner. Some of these file types are listed here:

- **CSV files** (generally have the .csv extension) hold tabular data that resembles spreadsheet data.
- **Image files** (generally with the .png or .jpg extension) hold images for computer vision.
- **Text files** (often have the .txt extension) hold unstructured text and are essential for natural language processing.
- **JSON** (often have the .json extension) contain semi-structured textual data in a human-readable text-based format.
- **H5** (can have a wide array of extensions) contain semi-structured textual data in a human-readable text-based format. Keras and TensorFlow store neural networks as H5 files.
- **Audio Files** (often have an extension such as .au or .wav) contain recorded sound.

Data can come from a variety of sources. In this class, we obtain data from three primary locations:

- **Your Hard Drive** - This type of data is stored locally, and Python accesses it from a path that looks something like: **c:\data\myfile.csv or /Users/jheaton/data/myfile.csv**.
- **The Internet** - This type of data resides in the cloud, and Python accesses it from a URL that looks something like:

<https://data.heatonresearch.com/data/t81-558/iris.csv>.

- **Google Drive (cloud)** - If your code in Google CoLab, you use GoogleDrive to save and load some data files. CoLab mounts your GoogleDrive into a path similar to the following: **/content/drive/My Drive/myfile.csv**.

## Read a CSV File

Python programs can read CSV files with Pandas. We will see more about Pandas in the next section, but for now, its general format is:

```
In [1]: import pandas as pd  
df = pd.read_csv("https://data.heatonresearch.com/data/t81-558/iris.csv")
```

The above command loads Fisher's Iris data set from the Internet. It might take a few seconds to load, so it is good to keep the loading code in a separate Jupyter notebook cell so that you do not have to reload it as you test your program. You can load Internet data, local hard drive, and Google Drive data this way.

Now that the data is loaded, you can display the first five rows with this command.

In [2]:

```
display(df[0:5])
```

	sepal_l	sepal_w	petal_l	petal_w	species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

## Read (stream) a Large CSV File

Pandas will read the entire CSV file into memory. Usually, this is fine. However, at times you may wish to "stream" a huge file. Streaming allows you to process this file one record at a time. Because the program does not load all of the data into memory, you can handle huge files. The following code loads the Iris dataset and calculates averages, one row at a time. This technique would work for large files.

In [3]:

```
import csv
import urllib.request
import codecs
import numpy as np

url = "https://data/heatonresearch.com/data/t81-558/iris.csv"
urlstream = urllib.request.urlopen(url)
csvfile = csv.reader(codecs.iterdecode(urlstream, 'utf-8'))
next(csvfile) # Skip header row
sum = np.zeros(4)
count = 0

for line in csvfile:
    # Convert each row to Numpy array
    line2 = np.array(line)[0:4].astype(float)

    # If the line is of the right length (skip empty lines), then add
    if len(line2) == 4:
        sum += line2
        count += 1
```

```
# Calculate the average, and print the average of the 4 iris
# measurements (features)
print(sum/count)
```

```
[5.84333333 3.05733333 3.758      1.19933333]
```

## Read a Text File

The following code reads the [Sonnet 18](#) by William Shakespeare as a text file. This code streams the document and reads it line-by-line. This code could handle a huge file.

```
In [4]: import urllib.request
import codecs

url = "https://data.heatonresearch.com/data/t81-558/datasets/sonnet_18.txt"
with urllib.request.urlopen(url) as urlstream:
    for line in codecs.iterdecode(urlstream, 'utf-8'):
        print(line.rstrip())
```

Sonnet 18 original text  
William Shakespeare

```
Shall I compare thee to a summer's day?
Thou art more lovely and more temperate:
Rough winds do shake the darling buds of May,
And summer's lease hath all too short a date:
Sometime too hot the eye of heaven shines,
And often is his gold complexion dimm'd;
And every fair from fair sometime declines,
By chance or nature's changing course untrimm'd;
But thy eternal summer shall not fade
Nor lose possession of that fair thou owest;
Nor shall Death brag thou wander'st in his shade,
When in eternal lines to time thou growest:
So long as men can breathe or eyes can see,
So long lives this and this gives life to thee.
```

## Read an Image

Computer vision is one of the areas that neural networks outshine other models. To support computer vision, the Python programmer needs to understand how to process images. For this course, we will use the Python PIL package for image processing. The following code demonstrates how to load an image from a URL and display it.

```
In [5]: %matplotlib inline
from PIL import Image
import requests
from io import BytesIO
```

```
url = "https://data.heatonresearch.com/images/jupyter/brookings.jpeg"

response = requests.get(url)
img = Image.open(BytesIO(response.content))

img
```

Out[5]:



In [ ]:

# T81-558: Applications of Deep Neural Networks

## Module 1: Python Preliminaries

- Instructor: [Jeff Heaton](#), McKelvey School of Engineering, [Washington University in St. Louis](#)
- For more information visit the [class website](#).

## Module 1 Material

- Part 1.1: Course Overview [\[Video\]](#) [\[Notebook\]](#)
- Part 1.2: Introduction to Python [\[Video\]](#) [\[Notebook\]](#)
- Part 1.3: Python Lists, Dictionaries, Sets and JSON [\[Video\]](#) [\[Notebook\]](#)
- Part 1.4: File Handling [\[Video\]](#) [\[Notebook\]](#)
- **Part 1.5: Functions, Lambdas, and Map/Reduce** [\[Video\]](#) [\[Notebook\]](#)

## Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

```
In [1]: try:  
    from google.colab import drive  
    %tensorflow_version 2.x  
    COLAB = True  
    print("Note: using Google CoLab")  
except:  
    print("Note: not using Google CoLab")  
    COLAB = False
```

## Part 1.5: Functions, Lambdas, and Map/Reduce

Functions, **lambdas**, and **map/reduce** can allow you to process your data in advanced ways. We will introduce these techniques here and expand on them in the next module, which will discuss Pandas.

Function parameters can be named or unnamed in Python. Default values can also be used. Consider the following function.

```
In [2]: def say_hello(speaker, person_to_greet, greeting = "Hello"):
    print(f'{greeting} {person_to_greet}, this is {speaker}.')
say_hello('Jeff', "John")
say_hello('Jeff', "John", "Goodbye")
say_hello(speaker='Jeff', person_to_greet="John", greeting = "Goodbye")
```

Hello John, this is Jeff.  
Goodbye John, this is Jeff.  
Goodbye John, this is Jeff.

A function is a way to capture code that is commonly executed. Consider the following function that can be used to trim white space from a string capitalize the first letter.

```
In [3]: def process_string(str):
    t = str.strip()
    return t[0].upper()+t[1:]
```

This function can now be called quite easily.

```
In [4]: str = process_string(" hello ")
print(f'{str}')
```

"Hello"

Python's **map** is a very useful function that is provided in many different programming languages. The **map** function takes a **list** and applies a function to each member of the **list** and returns a second **list** that is the same size as the first.

```
In [5]: l = [' apple ', 'pear ', 'orange', 'pine apple ']
list(map(process_string, l))
```

Out[5]: ['Apple', 'Pear', 'Orange', 'Pine apple']

## Map

The **map** function is very similar to the Python **comprehension** that we previously explored. The following **comprehension** accomplishes the same task as the previous call to **map**.

```
In [6]: l = [' apple ', 'pear ', 'orange', 'pine apple ']
l2 = [process_string(x) for x in l]
print(l2)
```

['Apple', 'Pear', 'Orange', 'Pine apple']

The choice of using a **map** function or **comprehension** is up to the programmer. I tend to prefer **map** since it is so common in other programming languages.

## Filter

While a **map function** always creates a new **list** of the same size as the original, the **filter** function creates a potentially smaller **list**.

```
In [7]: def greater_than_five(x):
    return x>5

l = [ 1, 10, 20, 3, -2, 0]
l2 = list(filter(greater_than_five, l))
print(l2)

[10, 20]
```

## Lambda

It might seem somewhat tedious to have to create an entire function just to check to see if a value is greater than 5. A **lambda** saves you this effort. A lambda is essentially an unnamed function.

```
In [8]: l = [ 1, 10, 20, 3, -2, 0]
l2 = list(filter(lambda x: x>5, l))
print(l2)

[10, 20]
```

## Reduce

Finally, we will make use of **reduce**. Like **filter** and **map** the **reduce** function also works on a **list**. However, the result of the **reduce** is a single value. Consider if you wanted to sum the **values** of a **list**. The sum is implemented by a **lambda**.

```
In [9]: from functools import reduce

l = [ 1, 10, 20, 3, -2, 0]
result = reduce(lambda x,y: x+y, l)
print(result)
```



# T81-558: Applications of Deep Neural Networks

## Module 2: Python for Machine Learning

- Instructor: [Jeff Heaton](#), McKelvey School of Engineering, [Washington University in St. Louis](#)
- For more information visit the [class website](#).

## Module 2 Material

Main video lecture:

- **Part 2.1: Introduction to Pandas** [\[Video\]](#) [\[Notebook\]](#)
- Part 2.2: Categorical Values [\[Video\]](#) [\[Notebook\]](#)
- Part 2.3: Grouping, Sorting, and Shuffling in Python Pandas [\[Video\]](#) [\[Notebook\]](#)
- Part 2.4: Using Apply and Map in Pandas for Keras [\[Video\]](#) [\[Notebook\]](#)
- Part 2.5: Feature Engineering in Pandas for Deep Learning in Keras [\[Video\]](#) [\[Notebook\]](#)

## Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

```
In [1]: try:  
    from google.colab import drive  
    %tensorflow_version 2.x  
    COLAB = True  
    print("Note: using Google CoLab")  
except:  
    print("Note: not using Google CoLab")  
    COLAB = False
```

Note: not using Google CoLab

## Part 2.1: Introduction to Pandas

Pandas is an open-source library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language. It is based on the [dataframe](#) concept found in the [R programming language](#). For this class, Pandas will be the primary means by which we manipulate data to be processed by neural networks.

The data frame is a crucial component of Pandas. We will use it to access the [auto-mpg dataset](#). You can find this dataset on the UCI machine learning repository. For this class, we will use a version of the Auto MPG dataset, where I added column headers. You can find my [version](#) at <https://data.heatonresearch.com/>.

UCI took this dataset from the StatLib library, which Carnegie Mellon University maintains. The dataset was used in the 1983 American Statistical Association Exposition. It contains data for 398 cars, including [mpg](#), [cylinders](#), [displacement](#), [horsepower](#), weight, acceleration, model year, origin and the car's name.

The following code loads the MPG dataset into a data frame:

```
In [2]: # Simple dataframe
import os
import pandas as pd

pd.set_option('display.max_columns', 7)
df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv")
display(df[0:5])
```

	mpg	cylinders	displacement	...	year	origin		name
0	18.0	8	307.0	...	70	1	chevrolet chevelle malibu	
1	15.0	8	350.0	...	70	1	buick skylark 320	
2	18.0	8	318.0	...	70	1	plymouth satellite	
3	16.0	8	304.0	...	70	1	amc rebel sst	
4	17.0	8	302.0	...	70	1	ford torino	

5 rows × 9 columns

The [display](#) function provides a cleaner display than merely printing the data frame. Specifying the maximum rows and columns allows you to achieve greater control over the display.

```
In [3]: pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)
display(df)
```

	mpg	cylinders	displacement	...	year	origin	name
0	18.0	8	307.0	...	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	...	70	1	buick skylark 320
...	...	...	...	...	...	...	...
396	28.0	4	120.0	...	82	1	ford ranger
397	31.0	4	119.0	...	82	1	chevy s-10

398 rows × 9 columns

It is possible to generate a second data frame to display statistical information about the first data frame.

```
In [4]: # Strip non-numerics
df = df.select_dtypes(include=['int', 'float'])

headers = list(df.columns.values)
fields = []

for field in headers:
    fields.append({
        'name' : field,
        'mean': df[field].mean(),
        'var': df[field].var(),
        'sdev': df[field].std()
    })

for field in fields:
    print(field)

{'name': 'mpg', 'mean': 23.514572864321615, 'var': 61.089610774274405, 'sde v': 7.815984312565782}
{'name': 'cylinders', 'mean': 5.454773869346734, 'var': 2.8934154399199943, 'sdev': 1.7010042445332094}
{'name': 'displacement', 'mean': 193.42587939698493, 'var': 10872.199152247364, 'sdev': 104.26983817119581}
{'name': 'weight', 'mean': 2970.424623115578, 'var': 717140.9905256768, 'sde v': 846.8417741973271}
{'name': 'acceleration', 'mean': 15.568090452261291, 'var': 7.604848233611381, 'sdev': 2.7576889298126757}
{'name': 'year', 'mean': 76.01005025125629, 'var': 13.672442818627143, 'sde v': 3.697626646732623}
{'name': 'origin', 'mean': 1.5728643216080402, 'var': 0.6432920268850575, 's dev': 0.8020548777266163}
```

This code outputs a list of dictionaries that hold this statistical information. This information looks similar to the JSON code seen in Module 1. If proper JSON is needed, the program should add these records to a list and call the Python JSON library's **dumps** command.

The Python program can convert this JSON-like information to a data frame for better display.

```
In [5]: pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 0)
df2 = pd.DataFrame(fields)
display(df2)
```

	<b>name</b>	<b>mean</b>	<b>var</b>	<b>sdev</b>
<b>0</b>	mpg	23.514573	61.089611	7.815984
<b>1</b>	cylinders	5.454774	2.893415	1.701004
<b>2</b>	displacement	193.425879	10872.199152	104.269838
<b>3</b>	weight	2970.424623	717140.990526	846.841774
<b>4</b>	acceleration	15.568090	7.604848	2.757689
<b>5</b>	year	76.010050	13.672443	3.697627
<b>6</b>	origin	1.572864	0.643292	0.802055

## Missing Values

Missing values are a reality of machine learning. Ideally, every row of data will have values for all columns. However, this is rarely the case. Most of the values are present in the MPG database. However, there are missing values in the horsepower column. A common practice is to replace missing values with the median value for that column. The program calculates the [median](#). The following code replaces any NA values in horsepower with the median:

```
In [6]: import os
import pandas as pd

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])
print(f"horsepower has na? {pd.isnull(df['horsepower']).values.any()}")

print("Filling missing values...")
med = df['horsepower'].median()
df['horsepower'] = df['horsepower'].fillna(med)
# df = df.dropna() # you can also simply drop NA values

print(f"horsepower has na? {pd.isnull(df['horsepower']).values.any()}")

horsepower has na? True
Filling missing values...
horsepower has na? False
```

## Dealing with Outliers

Outliers are values that are unusually high or low. We typically consider outliers to be a value that is several standard deviations from the mean. Sometimes outliers are simply errors; this is a result of [observation error](#). Outliers can also be truly large or small values that may be difficult to address. The following function can remove such values.

```
In [7]: # Remove all rows where the specified column is +/- sd standard deviations
def remove_outliers(df, name, sd):
    drop_rows = df.index[(np.abs(df[name] - df[name].mean()) >= (sd * df[name].std()))]
    df.drop(drop_rows, axis=0, inplace=True)
```

The code below will drop every row from the Auto MPG dataset where the horsepower is two standard deviations or more above or below the mean.

```
In [8]: import pandas as pd
import os
import numpy as np
from sklearn import metrics
from scipy.stats import zscore

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

# create feature vector
med = df['horsepower'].median()
df['horsepower'] = df['horsepower'].fillna(med)

# Drop the name column
df.drop('name', 1, inplace=True)

# Drop outliers in horsepower
print("Length before MPG outliers dropped: {}".format(len(df)))
remove_outliers(df, 'mpg', 2)
print("Length after MPG outliers dropped: {}".format(len(df)))

pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 5)
display(df)
```

```
Length before MPG outliers dropped: 398
Length after MPG outliers dropped: 388
```

	<b>mpg</b>	<b>cylinders</b>	<b>displacement</b>	<b>horsepower</b>	<b>weight</b>	<b>acceleration</b>	<b>year</b>	<b>or</b>
<b>0</b>	18.0	8	307.0	130.0	3504	12.0	70	
<b>1</b>	15.0	8	350.0	165.0	3693	11.5	70	
...	...	...	...	...	...	...	...	...
<b>396</b>	28.0	4	120.0	79.0	2625	18.6	82	
<b>397</b>	31.0	4	119.0	82.0	2720	19.4	82	

388 rows × 8 columns

## Dropping Fields

You must drop fields that are of no value to the neural network. The following code removes the name column from the MPG dataset.

In [9]:

```
import os
import pandas as pd

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

print(f"Before drop: {list(df.columns)}")
df.drop('name', 1, inplace=True)
print(f"After drop: {list(df.columns)}")
```

Before drop: ['mpg', 'cylinders', 'displacement', 'horsepower', 'weight', 'acceleration', 'year', 'origin', 'name']  
After drop: ['mpg', 'cylinders', 'displacement', 'horsepower', 'weight', 'acceleration', 'year', 'origin']

## Concatenating Rows and Columns

Python can concatenate rows and columns together to form new data frames. The code below creates a new data frame from the **name** and **horsepower** columns from the Auto MPG dataset. The program does this by concatenating two columns together.

In [10]:

```
# Create a new dataframe from name and horsepower

import os
import pandas as pd

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

col_horsepower = df['horsepower']
```

```

col_name = df['name']
result = pd.concat([col_name, col_horsepower], axis=1)

pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 5)
display(result)

```

	name	horsepower
0	chevrolet chevelle malibu	130.0
1	buick skylark 320	165.0
...	...	...
396	ford ranger	79.0
397	chevy s-10	82.0

398 rows × 2 columns

The **concat** function can also concatenate rows together. This code concatenates the first two rows and the last two rows of the Auto MPG dataset.

In [11]: # Create a new dataframe from first 2 rows and last 2 rows

```

import os
import pandas as pd

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

result = pd.concat([df[0:2], df[-2:]], axis=0)

pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 0)
display(result)

```

	mpg	cylinders	displacement	...	year	origin	name
0	18.0	8	307.0	...	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	...	70	1	buick skylark 320
396	28.0	4	120.0	...	82	1	ford ranger
397	31.0	4	119.0	...	82	1	chevy s-10

4 rows × 9 columns

## Training and Validation

We must evaluate a machine learning model based on its ability to predict values that it has never seen before. Because of this, we often divide the training data into a validation and training set. The machine learning model will learn from the training data but ultimately be evaluated based on the validation data.

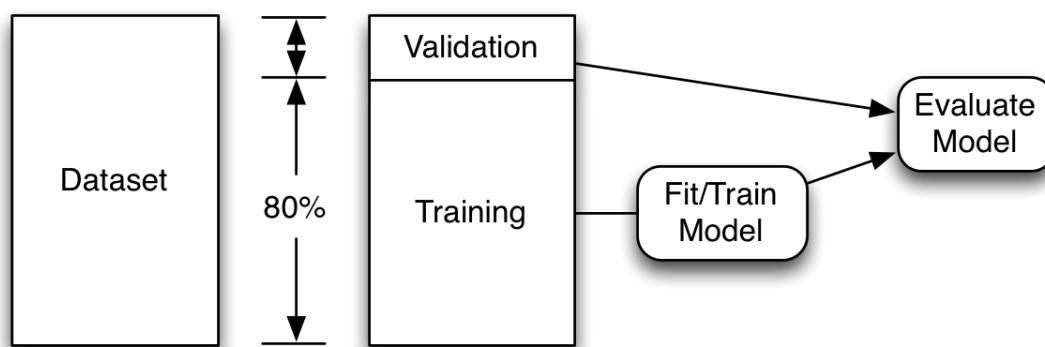
- **Training Data - In Sample Data** - The data that the neural network used to train.
- **Validation Data - Out of Sample Data** - The data that the machine learning model is evaluated upon after it is fit to the training data.

There are two effective means of dealing with training and validation data:

- **Training/Validation Split** - The program splits the data according to some ratio between a training and validation (hold-out) set. Typical rates are 80% training and 20% validation.
- **K-Fold Cross Validation** - The program splits the data into several folds and models. Because the program creates the same number of models as folds, the program can generate out-of-sample predictions for the entire dataset.

The code below splits the MPG data into a training and validation set. The training set uses 80% of the data, and the validation set uses 20%. Figure 2.TRN-VAL shows how we train a model on 80% of the data and then validated against the remaining 20%.

**Figure 2.TRN-VAL: Training and Validation**



```
In [12]: import os
import pandas as pd
import numpy as np

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

# Usually a good idea to shuffle
df = df.reindex(np.random.permutation(df.index))
```

```
mask = np.random.rand(len(df)) < 0.8
trainDF = pd.DataFrame(df[mask])
validationDF = pd.DataFrame(df[~mask])

print(f"Training DF: {len(trainDF)}")
print(f"Validation DF: {len(validationDF)}")
```

```
Training DF: 333
Validation DF: 65
```

## Converting a Dataframe to a Matrix

Neural networks do not directly operate on Python data frames. A neural network requires a numeric matrix. The program uses a data frame's **values** property to convert the data to a matrix.

```
In [13]: df.values
```

```
Out[13]: array([[20.2, 6, 232.0, ..., 79, 1, 'amc concord dl 6'],
                 [14.0, 8, 304.0, ..., 74, 1, 'amc matador (sw)'],
                 [14.0, 8, 351.0, ..., 71, 1, 'ford galaxie 500'],
                 ...,
                 [20.2, 6, 200.0, ..., 78, 1, 'ford fairmont (auto)'],
                 [26.0, 4, 97.0, ..., 70, 2, 'volkswagen 1131 deluxe sedan'],
                 [19.4, 6, 232.0, ..., 78, 1, 'amc concord']], dtype=object)
```

You might wish only to convert some of the columns, to leave out the name column, use the following code.

```
In [14]: df[['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
           'acceleration', 'year', 'origin']].values
```

```
Out[14]: array([[ 20.2,    6.,   232., ...,   18.2,    79.,    1. ],
                 [ 14.,     8.,   304., ...,   15.5,    74.,    1. ],
                 [ 14.,     8.,   351., ...,   13.5,    71.,    1. ],
                 ...,
                 [ 20.2,    6.,   200., ...,   15.8,    78.,    1. ],
                 [ 26.,     4.,   97., ...,   20.5,    70.,    2. ],
                 [ 19.4,    6.,   232., ...,   17.2,    78.,    1. ]])
```

## Saving a Dataframe to CSV

Many of the assignments in this course will require that you save a data frame to submit to the instructor. The following code performs a shuffle and then saves a new copy.

```
In [15]: import os
import pandas as pd
import numpy as np
```

```

path = "."
df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

filename_write = os.path.join(path, "auto-mpg-shuffle.csv")
df = df.reindex(np.random.permutation(df.index))
# Specify index = false to not write row numbers
df.to_csv(filename_write, index=False)

```

Done

## Saving a Dataframe to Pickle

A variety of software programs can use text files stored as CSV. However, they take longer to generate and can sometimes lose small amounts of precision in the conversion. Generally, you will output to CSV because it is very compatible, even outside of Python. Another format is **Pickle**. The code below stores the Dataframe to Pickle. Pickle stores data in the exact binary representation used by Python. The benefit is that there is no loss of data going to CSV format. The disadvantage is that generally, only Python programs can read Pickle files.

```

In [16]: import os
import pandas as pd
import numpy as np
import pickle

path = "."

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

filename_write = os.path.join(path, "auto-mpg-shuffle.pkl")
df = df.reindex(np.random.permutation(df.index))

with open(filename_write, "wb") as fp:
    pickle.dump(df, fp)

```

Loading the pickle file back into memory is accomplished by the following lines of code. Notice that the index numbers are still jumbled from the previous shuffle? Loading the CSV rebuilt (in the last step) did not preserve these values.

```

In [17]: import os
import pandas as pd
import numpy as np
import pickle

path = "."

df = pd.read_csv(

```

```

    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

filename_read = os.path.join(path, "auto-mpg-shuffle.pkl")

with open(filename_write, "rb") as fp:
    df = pickle.load(fp)

pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)
display(df)

```

	mpg	cylinders	displacement	...	year	origin		name
<b>387</b>	38.0	6	262.0	...	82	1	oldsmobile	cutlass ciera (diesel)
<b>361</b>	25.4	6	168.0	...	81	3	toyota	cressida
...	...	...	...	...	...	...	...	...
<b>358</b>	31.6	4	120.0	...	81	3	mazda	626
<b>237</b>	30.5	4	98.0	...	77	1	chevrolet	chevette

398 rows × 9 columns

## Module 2 Assignment

You can find the first assignment here: [assignment 2](#)

In [ ]:



# T81-558: Applications of Deep Neural Networks

## Module 2: Python for Machine Learning

- Instructor: [Jeff Heaton](#), McKelvey School of Engineering, [Washington University in St. Louis](#)
- For more information visit the [class website](#).

## Module 2 Material

Main video lecture:

- Part 2.1: Introduction to Pandas [\[Video\]](#) [\[Notebook\]](#)
- **Part 2.2: Categorical Values** [\[Video\]](#) [\[Notebook\]](#)
- Part 2.3: Grouping, Sorting, and Shuffling in Python Pandas [\[Video\]](#) [\[Notebook\]](#)
- Part 2.4: Using Apply and Map in Pandas for Keras [\[Video\]](#) [\[Notebook\]](#)
- Part 2.5: Feature Engineering in Pandas for Deep Learning in Keras [\[Video\]](#) [\[Notebook\]](#)

## Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

```
In [1]: try:  
    %tensorflow_version 2.x  
    COLAB = True  
    print("Note: using Google CoLab")  
except:  
    print("Note: not using Google CoLab")  
    COLAB = False
```

Note: not using Google CoLab

## Part 2.2: Categorical and Continuous Values

Neural networks require their input to be a fixed number of columns. This input format is very similar to spreadsheet data; it must be entirely numeric. It is essential to represent the data so that the neural network can train from it. Before we look at specific ways to preprocess data, it is important to consider four basic types of data, as defined by [Cite:stevens1946theory]. Statisticians commonly refer to as the [levels of measure](#):

- Character Data (strings)
  - **Nominal** - Individual discrete items, no order. For example, color, zip code, and shape.
  - **Ordinal** - Individual distinct items have an implied order. For example, grade level, job title, Starbucks(tm) coffee size (tall, vente, grande)
- Numeric Data
  - **Interval** - Numeric values, no defined start. For example, temperature. You would never say, "yesterday was twice as hot as today."
  - **Ratio** - Numeric values, clearly defined start. For example, speed. You could say, "The first car is going twice as fast as the second."

## Encoding Continuous Values

One common transformation is to normalize the inputs. It is sometimes valuable to normalize numeric inputs in a standard form so that the program can easily compare these two values. Consider if a friend told you that he received a 10-dollar discount. Is this a good deal? Maybe. But the cost is not normalized. If your friend purchased a car, the discount is not that good. If your friend bought lunch, this is an excellent discount!

Percentages are a prevalent form of normalization. If your friend tells you they got 10% off, we know that this is a better discount than 5%. It does not matter how much the purchase price was. One widespread machine learning normalization is the Z-Score:

$$z = \frac{x - \mu}{\sigma}$$

To calculate the Z-Score, you also need to calculate the mean( $\mu$  or  $\bar{x}$ ) and the standard deviation ( $\sigma$ ). You can calculate the mean with this equation:

$$\mu = \bar{x} = \frac{x_1 + x_2 + \dots + x_n}{n}$$

The standard deviation is calculated as follows:

$$\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2}$$

The following Python code replaces the mpg with a z-score. Cars with average MPG will be near zero, above zero is above average, and below zero is below average. Z-Scores more than 3 above or below are very rare; these are outliers.

In [2]:

```
import os
import pandas as pd
from scipy.stats import zscore

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)

df['mpg'] = zscore(df['mpg'])
display(df)
```

	mpg	cylinders	displacement	...	year	origin	name
<b>0</b>	-0.706439	8	307.0	...	70	1	chevrolet chevelle malibu
<b>1</b>	-1.090751	8	350.0	...	70	1	buick skylark 320
...	...	...	...	...	...	...	...
<b>396</b>	0.574601	4	120.0	...	82	1	ford ranger
<b>397</b>	0.958913	4	119.0	...	82	1	chevy s-10

398 rows × 9 columns

## Encoding Categorical Values as Dummies

The traditional means of encoding categorical values is to make them dummy variables. This technique is also called one-hot-encoding. Consider the following data set.

In [3]:

```
import pandas as pd

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/jh-simple-dataset.csv",
    na_values=['NA', '?'])

pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)
```

```
display(df)
```

	<b>id</b>	<b>job</b>	<b>area</b>	...	<b>retail_dense</b>	<b>crime</b>	<b>product</b>
<b>0</b>	1	vv	c	...	0.492126	0.071100	b
<b>1</b>	2	kd	c	...	0.342520	0.400809	c
...	...	...	...	...	...	...	...
<b>1998</b>	1999	qp	c	...	0.598425	0.117803	c
<b>1999</b>	2000	pe	c	...	0.539370	0.451973	c

2000 rows × 14 columns

The *area* column is not numeric, so you must encode it with one-hot encoding. We display the number of areas and individual values. There are just four values in the *area* categorical variable in this case.

```
In [4]: areas = list(df['area'].unique())
print(f'Number of areas: {len(areas)}')
print(f'Areas: {areas}'')
```

```
Number of areas: 4
Areas: ['c', 'd', 'a', 'b']
```

There are four unique values in the *area* column. To encode these dummy variables, we would use four columns, each representing one of the areas. For each row, one column would have a value of one, the rest zeros. For this reason, this type of encoding is sometimes called one-hot encoding. The following code shows how you might encode the values "a" through "d." The value A becomes [1,0,0,0] and the value B becomes [0,1,0,0].

```
In [5]: dummies = pd.get_dummies(['a', 'b', 'c', 'd'], prefix='area')
print(dummies)
```

```
    area_a  area_b  area_c  area_d
0      1      0      0      0
1      0      1      0      0
2      0      0      1      0
3      0      0      0      1
```

We can now encode the actual column.

```
In [6]: dummies = pd.get_dummies(df['area'], prefix='area')
print(dummies[0:10]) # Just show the first 10
```

```

area_a  area_b  area_c  area_d
0        0        0        1        0
1        0        0        1        0
...
8        ...      ...      ...
9        1        0        0        0

```

[10 rows x 4 columns]

For the new dummy/one hot encoded values to be of any use, they must be merged back into the data set.

```
In [7]: df = pd.concat([df,dummies],axis=1)
```

To encode the *area* column, we use the following code. Note that it is necessary to merge these dummies back into the data frame.

```
In [8]: pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 10)
```

```
display(df[['id','job','area','income','area_a',
           'area_b','area_c','area_d']])
```

	<b>id</b>	<b>job</b>	<b>area</b>	<b>income</b>	<b>area_a</b>	<b>area_b</b>	<b>area_c</b>	<b>area_d</b>
<b>0</b>	1	vv	c	50876.0	0	0	1	0
<b>1</b>	2	kd	c	60369.0	0	0	1	0
<b>2</b>	3	pe	c	55126.0	0	0	1	0
<b>3</b>	4	11	c	51690.0	0	0	1	0
<b>4</b>	5	kl	d	28347.0	0	0	0	1
...	...	...	...	...	...	...	...	...
<b>1995</b>	1996	vv	c	51017.0	0	0	1	0
<b>1996</b>	1997	kl	d	26576.0	0	0	0	1
<b>1997</b>	1998	kl	d	28595.0	0	0	0	1
<b>1998</b>	1999	qp	c	67949.0	0	0	1	0
<b>1999</b>	2000	pe	c	61467.0	0	0	1	0

2000 rows x 8 columns

Usually, you will remove the original column *area* because the goal is to get the data frame to be entirely numeric for the neural network.

```
In [9]: pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 5)

df.drop('area', axis=1, inplace=True)
```

```
display(df[['id','job','income','area_a',
           'area_b','area_c','area_d']])
```

	<b>id</b>	<b>job</b>	<b>income</b>	<b>area_a</b>	<b>area_b</b>	<b>area_c</b>	<b>area_d</b>
<b>0</b>	1	vv	50876.0	0	0	1	0
<b>1</b>	2	kd	60369.0	0	0	1	0
...	...	...	...	...	...	...	...
<b>1998</b>	1999	qp	67949.0	0	0	1	0
<b>1999</b>	2000	pe	61467.0	0	0	1	0

2000 rows × 7 columns

## Removing the First Level

The **pd.concat** function also includes a parameter named *drop\_first*, which specifies whether to get k-1 dummies out of k categorical levels by removing the first level. Why would you want to remove the first level, in this case, *area\_a*? This technique provides a more efficient encoding by using the ordinarily unused encoding of [0,0,0]. We encode the *area* to just three columns and map the categorical value of *a* to [0,0,0]. The following code demonstrates this technique.

```
In [10]: import pandas as pd

dummies = pd.get_dummies(['a','b','c','d'],prefix='area', drop_first=True)
print(dummies)
```

	<b>area_b</b>	<b>area_c</b>	<b>area_d</b>
0	0	0	0
1	1	0	0
2	0	1	0
3	0	0	1

As you can see from the above data, the *area\_a* column is missing, as it **get\_dummies** replaced it by the encoding of [0,0,0]. The following code shows how to apply this technique to a dataframe.

```
In [11]: import pandas as pd

# Read the dataset
df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/jh-simple-dataset.csv",
    na_values=['NA','?'])

# encode the area column as dummy variables
dummies = pd.get_dummies(df['area'], drop_first=True, prefix='area')
df = pd.concat([df,dummies],axis=1)
df.drop('area', axis=1, inplace=True)
```

```
# display the encoded dataframe
pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 10)

display(df[['id','job','income',
           'area_b','area_c','area_d']])
```

	<b>id</b>	<b>job</b>	<b>income</b>	<b>area_b</b>	<b>area_c</b>	<b>area_d</b>
<b>0</b>	1	vv	50876.0	0	1	0
<b>1</b>	2	kd	60369.0	0	1	0
<b>2</b>	3	pe	55126.0	0	1	0
<b>3</b>	4	11	51690.0	0	1	0
<b>4</b>	5	kl	28347.0	0	0	1
...	...	...	...	...	...	...
<b>1995</b>	1996	vv	51017.0	0	1	0
<b>1996</b>	1997	kl	26576.0	0	0	1
<b>1997</b>	1998	kl	28595.0	0	0	1
<b>1998</b>	1999	qp	67949.0	0	1	0
<b>1999</b>	2000	pe	61467.0	0	1	0

2000 rows × 6 columns

## Target Encoding for Categoricals

Target encoding is a popular technique for Kaggle competitions. Target encoding can sometimes increase the predictive power of a machine learning model. However, it also dramatically increases the risk of overfitting. Because of this risk, you must take care of using this method.

Generally, target encoding can only be used on a categorical feature when the output of the machine learning model is numeric (regression).

The concept of target encoding is straightforward. For each category, we calculate the average target value for that category. Then to encode, we substitute the percent corresponding to the category that the categorical value has. Unlike dummy variables, where you have a column for each category with target encoding, the program only needs a single column. In this way, target coding is more efficient than dummy variables.

```
In [13]: # Create a small sample dataset
import pandas as pd
import numpy as np
```

```

np.random.seed(43)
df = pd.DataFrame({
    'cont_9': np.random.rand(10)*100,
    'cat_0': ['dog'] * 5 + ['cat'] * 5,
    'cat_1': ['wolf'] * 9 + ['tiger'] * 1,
    'y': [1, 0, 1, 1, 1, 1, 0, 0, 0, 0]
})

pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 0)
display(df)

```

	cont_9	cat_0	cat_1	y
0	11.505457	dog	wolf	1
1	60.906654	dog	wolf	0
2	13.339096	dog	wolf	1
3	24.058962	dog	wolf	1
4	32.713906	dog	wolf	1
5	85.913749	cat	wolf	1
6	66.609021	cat	wolf	0
7	54.116221	cat	wolf	0
8	2.901382	cat	wolf	0
9	73.374830	cat	tiger	0

We want to change them to a number rather than creating dummy variables for "dog" and "cat," we would like to change them to a number. We could use 0 for a cat and 1 for a dog. However, we can encode more information than just that. The simple 0 or 1 would also only work for one animal. Consider what the mean target value is for cat and dog.

```
In [14]: means0 = df.groupby('cat_0')['y'].mean().to_dict()
means0
```

```
Out[14]: {'cat': 0.2, 'dog': 0.8}
```

The danger is that we are now using the target value ( $y$ ) for training. This technique will potentially lead to overfitting. The possibility of overfitting is even greater if a small number of a particular category. To prevent this from happening, we use a weighting factor. The stronger the weight, the more categories with fewer values will tend towards the overall average of  $y$ . You can perform this calculation as follows.

```
In [15]: df['y'].mean()
```

```
Out[15]: 0.5
```

You can implement target encoding as follows. For more information on Target Encoding, refer to the article "["Target Encoding Done the Right Way"](#)", that I based this code upon.

```
In [16]: def calc_smooth_mean(df1, df2, cat_name, target, weight):
    # Compute the global mean
    mean = df[target].mean()

    # Compute the number of values and the mean of each group
    agg = df.groupby(cat_name)[target].agg(['count', 'mean'])
    counts = agg['count']
    means = agg['mean']

    # Compute the "smoothed" means
    smooth = (counts * means + weight * mean) / (counts + weight)

    # Replace each value by the according smoothed mean
    if df2 is None:
        return df1[cat_name].map(smooth)
    else:
        return df1[cat_name].map(smooth), df2[cat_name].map(smooth.to_dict())
```

The following code encodes these two categories.

```
In [17]: WEIGHT = 5
df['cat_0_enc'] = calc_smooth_mean(df1=df, df2=None,
                                     cat_name='cat_0', target='y', weight=WEIGHT)
df['cat_1_enc'] = calc_smooth_mean(df1=df, df2=None,
                                     cat_name='cat_1', target='y', weight=WEIGHT)

pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 0)

display(df)
```

	<b>cont_9</b>	<b>cat_0</b>	<b>cat_1</b>	<b>y</b>	<b>cat_0_enc</b>	<b>cat_1_enc</b>
<b>0</b>	11.505457	dog	wolf	1	0.65	0.535714
<b>1</b>	60.906654	dog	wolf	0	0.65	0.535714
<b>2</b>	13.339096	dog	wolf	1	0.65	0.535714
<b>3</b>	24.058962	dog	wolf	1	0.65	0.535714
<b>4</b>	32.713906	dog	wolf	1	0.65	0.535714
<b>5</b>	85.913749	cat	wolf	1	0.35	0.535714
<b>6</b>	66.609021	cat	wolf	0	0.35	0.535714
<b>7</b>	54.116221	cat	wolf	0	0.35	0.535714
<b>8</b>	2.901382	cat	wolf	0	0.35	0.535714
<b>9</b>	73.374830	cat	tiger	0	0.35	0.416667

## Encoding Categorical Values as Ordinal

Typically categoricals will be encoded as dummy variables. However, there might be other techniques to convert categoricals to numeric. Any time there is an order to the categoricals, a number should be used. Consider if you had a categorical that described the current education level of an individual.

- Kindergarten (0)
- First Grade (1)
- Second Grade (2)
- Third Grade (3)
- Fourth Grade (4)
- Fifth Grade (5)
- Sixth Grade (6)
- Seventh Grade (7)
- Eighth Grade (8)
- High School Freshman (9)
- High School Sophomore (10)
- High School Junior (11)
- High School Senior (12)
- College Freshman (13)
- College Sophomore (14)
- College Junior (15)
- College Senior (16)
- Graduate Student (17)
- PhD Candidate (18)
- Doctorate (19)

- Post Doctorate (20)

The above list has 21 levels and would take 21 dummy variables to encode. However, simply encoding this to dummies would lose the order information. Perhaps the most straightforward approach would be to simply number them and assign the category a single number equal to the value in the parenthesis above. However, we might be able to do even better. A graduate student is likely more than a year so you might increase one value.

## High Cardinality Categorical

If there were many, perhaps thousands or tens of thousands, then one-hot encoding is no longer a good choice. We call these cases high cardinality categorical. We generally encode such values with an embedding layer, which we will discuss later when introducing natural language processing (NLP).



# T81-558: Applications of Deep Neural Networks

## Module 2: Python for Machine Learning

- Instructor: [Jeff Heaton](#), McKelvey School of Engineering, [Washington University in St. Louis](#)
- For more information visit the [class website](#).

## Module 2 Material

Main video lecture:

- Part 2.1: Introduction to Pandas [\[Video\]](#) [\[Notebook\]](#)
- Part 2.2: Categorical Values [\[Video\]](#) [\[Notebook\]](#)
- **Part 2.3: Grouping, Sorting, and Shuffling in Python Pandas** [\[Video\]](#) [\[Notebook\]](#)
- Part 2.4: Using Apply and Map in Pandas for Keras [\[Video\]](#) [\[Notebook\]](#)
- Part 2.5: Feature Engineering in Pandas for Deep Learning in Keras [\[Video\]](#) [\[Notebook\]](#)

## Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

```
In [1]: try:  
    %tensorflow_version 2.x  
    COLAB = True  
    print("Note: using Google CoLab")  
except:  
    print("Note: not using Google CoLab")  
    COLAB = False
```

Note: not using Google CoLab

## Part 2.3: Grouping, Sorting, and Shuffling

We will take a look at a few ways to affect an entire Pandas data frame. These techniques will allow us to group, sort, and shuffle data sets. These are all essential operations for both data preprocessing and evaluation.

## Shuffling a Dataset

There may be information lurking in the order of the rows of your dataset. Unless you are dealing with time-series data, the order of the rows should not be significant. Consider if your training set included employees in a company. Perhaps this dataset is ordered by the number of years the employees were with the company. It is okay to have an individual column that specifies years of service. However, having the data in this order might be problematic.

Consider if you were to split the data into training and validation. You could end up with your validation set having only the newer employees and the training set longer-term employees. Separating the data into a k-fold cross validation could have similar problems. Because of these issues, it is important to shuffle the data set.

Often shuffling and reindexing are both performed together. Shuffling randomizes the order of the data set. However, it does not change the Pandas row numbers. The following code demonstrates a reshuffle. Notice that the program has not reset the row indexes' first column. Generally, this will not cause any issues and allows tracing back to the original order of the data. However, I usually prefer to reset this index. I reason that I typically do not care about the initial position, and there are a few instances where this unordered index can cause issues.

```
In [2]: import os
import pandas as pd
import numpy as np

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

#np.random.seed(42) # Uncomment this line to get the same shuffle each time
df = df.reindex(np.random.permutation(df.index))

pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)
display(df)
```

	mpg	cylinders	displacement	...	year	origin	name
117	29.0	4	68.0	...	73	2	fiat 128
245	36.1	4	98.0	...	78	1	ford fiesta
...	...	...	...	...	...	...	...
88	14.0	8	302.0	...	73	1	ford gran torino
26	10.0	8	307.0	...	70	1	chevy c20

398 rows × 9 columns

The following code demonstrates a reindex. Notice how the reindex orders the row indexes.

```
In [3]: pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)

df.reset_index(inplace=True, drop=True)
display(df)
```

	mpg	cylinders	displacement	...	year	origin	name
0	29.0	4	68.0	...	73	2	fiat 128
1	36.1	4	98.0	...	78	1	ford fiesta
...	...	...	...	...	...	...	...
396	14.0	8	302.0	...	73	1	ford gran torino
397	10.0	8	307.0	...	70	1	chevy c20

398 rows × 9 columns

## Sorting a Data Set

While it is always good to shuffle a data set before training, during training and preprocessing, you may also wish to sort the data set. Sorting the data set allows you to order the rows in either ascending or descending order for one or more columns. The following code sorts the MPG dataset by name and displays the first car.

```
In [4]: import os
import pandas as pd

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

df = df.sort_values(by='name', ascending=True)
print(f"The first car is: {df['name'].iloc[0]}")
```

```
pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)
display(df)
```

The first car is: amc ambassador brougham

	mpg	cylinders	displacement	...	year	origin	name
<b>96</b>	13.0	8	360.0	...	73	1	amc ambassador brougham
<b>9</b>	15.0	8	390.0	...	70	1	amc ambassador dpl
...	...	...	...	...	...	...	...
<b>325</b>	44.3	4	90.0	...	80	2	vw rabbit c (diesel)
<b>293</b>	31.9	4	89.0	...	79	2	vw rabbit custom

398 rows × 9 columns

## Grouping a Data Set

Grouping is a typical operation on data sets. Structured Query Language (SQL) calls this operation a "GROUP BY." Programmers use grouping to summarize data. Because of this, the summarization row count will usually shrink, and you cannot undo the grouping. Because of this loss of information, it is essential to keep your original data before the grouping.

We use the Auto MPG dataset to demonstrate grouping.

```
In [5]: import os
import pandas as pd

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)
display(df)
```

	<b>mpg</b>	<b>cylinders</b>	<b>displacement</b>	...	<b>year</b>	<b>origin</b>	<b>name</b>
<b>0</b>	18.0	8	307.0	...	70	1	chevrolet chevelle malibu
<b>1</b>	15.0	8	350.0	...	70	1	buick skylark 320
...	...	...	...	...	...	...	...
<b>396</b>	28.0	4	120.0	...	82	1	ford ranger
<b>397</b>	31.0	4	119.0	...	82	1	chevy s-10

398 rows × 9 columns

You can use the above data set with the groupby to perform summaries. For example, the following code will group cylinders by the average (mean). This code will provide the grouping. In addition to **mean**, you can use other aggregating functions, such as **sum** or **count**.

```
In [6]: g = df.groupby('cylinders')['mpg'].mean()
g
```

```
Out[6]: cylinders
3    20.550000
4    29.286765
5    27.366667
6    19.985714
8    14.963107
Name: mpg, dtype: float64
```

It might be useful to have these **mean** values as a dictionary.

```
In [7]: d = g.to_dict()
d
```

```
Out[7]: {3: 20.55,
4: 29.28676470588236,
5: 27.366666666666664,
6: 19.985714285714284,
8: 14.963106796116508}
```

A dictionary allows you to access an individual element quickly. For example, you could quickly look up the mean for six-cylinder cars. You will see that target encoding, introduced later in this module, uses this technique.

```
In [8]: d[6]
```

```
Out[8]: 19.985714285714284
```

The code below shows how to count the number of rows that match each cylinder count.

```
In [9]: df.groupby('cylinders')['mpg'].count().to_dict()
```

```
Out[9]: {3: 4, 4: 204, 5: 3, 6: 84, 8: 103}
```



# T81-558: Applications of Deep Neural Networks

## Module 2: Python for Machine Learning

- Instructor: [Jeff Heaton](#), McKelvey School of Engineering, [Washington University in St. Louis](#)
- For more information visit the [class website](#).

## Module 2 Material

Main video lecture:

- Part 2.1: Introduction to Pandas [\[Video\]](#) [\[Notebook\]](#)
- Part 2.2: Categorical Values [\[Video\]](#) [\[Notebook\]](#)
- Part 2.3: Grouping, Sorting, and Shuffling in Python Pandas [\[Video\]](#) [\[Notebook\]](#)
- **Part 2.4: Using Apply and Map in Pandas for Keras** [\[Video\]](#) [\[Notebook\]](#)
- Part 2.5: Feature Engineering in Pandas for Deep Learning in Keras [\[Video\]](#) [\[Notebook\]](#)

## Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

```
In [1]: try:  
    %tensorflow_version 2.x  
    COLAB = True  
    print("Note: using Google CoLab")  
except:  
    print("Note: not using Google CoLab")  
    COLAB = False
```

Note: not using Google CoLab

## Part 2.4: Apply and Map

If you've ever worked with Big Data or functional programming languages before, you've likely heard of map/reduce. Map and reduce are two functions that apply a task you create to a data frame. Pandas supports functional programming techniques that allow you to use functions across an entire data frame. In addition to functions that you write, Pandas also provides several standard functions for use with data frames.

## Using Map with Dataframes

The map function allows you to transform a column by mapping certain values in that column to other values. Consider the Auto MPG data set that contains a field **origin\_name** that holds a value between one and three that indicates the geographic origin of each car. We can see how to use the map function to transform this numeric origin into the textual name of each origin.

We will begin by loading the Auto MPG data set.

```
In [2]: import os
import pandas as pd
import numpy as np

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)

display(df)
```

	mpg	cylinders	displacement	...	year	origin	name
0	18.0	8	307.0	...	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	...	70	1	buick skylark 320
...	...	...	...	...	...	...	...
396	28.0	4	120.0	...	82	1	ford ranger
397	31.0	4	119.0	...	82	1	chevy s-10

398 rows × 9 columns

The **map** method in Pandas operates on a single column. You provide **map** with a dictionary of values to transform the target column. The map keys specify what values in the target column should be turned into values specified by those keys. The following code shows how the map function can transform the numeric values of 1, 2, and 3 into the string values of North America, Europe, and Asia.

In [3]:

```
# Apply the map
df['origin_name'] = df['origin'].map(
    {1: 'North America', 2: 'Europe', 3: 'Asia'})

# Shuffle the data, so that we hopefully see
# more regions.
df = df.reindex(np.random.permutation(df.index))

# Display
pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 10)
display(df)
```

	mpg	cylinders	displacement	...	origin		name	origin_name
45	18.0	6	258.0	...	1	amc hornet sportabout (sw)		North America
290	15.5	8	351.0	...	1	ford country squire (sw)		North America
313	28.0	4	151.0	...	1	chevrolet citation		North America
82	23.0	4	120.0	...	3	toyouta corona mark ii (sw)		Asia
33	19.0	6	232.0	...	1	amc gremlin		North America
...	...	...	...	...	...	...	...	...
329	44.6	4	91.0	...	3	honda civic 1500 gl		Asia
326	43.4	4	90.0	...	2	vw dasher (diesel)		Europe
34	16.0	6	225.0	...	1	plymouth satellite custom		North America
118	24.0	4	116.0	...	2	opel manta		Europe
15	22.0	6	198.0	...	1	plymouth duster		North America

398 rows × 10 columns

## Using Apply with Dataframes

The **apply** function of the data frame can run a function over the entire data frame. You can use either a traditional named function or a lambda function. Python will execute the provided function against each of the rows or columns in the data frame. The **axis** parameter specifies that the function is run across rows or columns. For axis = 1, rows are used. The following code calculates a series called **efficiency** that is the **displacement** divided by **horsepower**.

```
In [4]: efficiency = df.apply(lambda x: x['displacement']/x['horsepower'], axis=1)
display(efficiency[0:10])
```

```
45      2.345455
290     2.471831
313     1.677778
82      1.237113
33      2.320000
249     2.363636
27      1.514286
7       2.046512
302     1.500000
179     1.234694
dtype: float64
```

You can now insert this series into the data frame, either as a new column or to replace an existing column. The following code inserts this new series into the data frame.

```
In [5]: df['efficiency'] = efficiency
```

## Feature Engineering with Apply and Map

In this section, we will see how to calculate a complex feature using map, apply, and grouping. The data set is the following CSV:

- <https://www.irs.gov/pub/irs-soi/16zpallagi.csv>

This URL contains US Government public data for "SOI Tax Stats - Individual Income Tax Statistics." The entry point to the website is here:

- <https://www.irs.gov/statistics/soi-tax-stats-individual-income-tax-statistics-2016-zip-code-data-soi>

Documentation describing this data is at the above link.

For this feature, we will attempt to estimate the adjusted gross income (AGI) for each of the zip codes. The data file contains many columns; however, you will only use the following:

- **STATE** - The state (e.g., MO)
- **zipcode** - The zipcode (e.g. 63017)
- **agi\_stub** - Six different brackets of annual income (1 through 6)
- **N1** - The number of tax returns for each of the agi\_stubs

Note, that the file will have six rows for each zip code for each of the agi\_stub brackets. You can skip zip codes with 0 or 99999.

We will create an output CSV with these columns; however, only one row per zip code. Calculate a weighted average of the income brackets. For example, the following six rows are present for 63017:

zipcode	agi_stub	N1
63017	1	4710
63017	2	2780
63017	3	2130
63017	4	2010
63017	5	5240
63017	6	3510

We must combine these six rows into one. For privacy reasons, AGI's are broken out into 6 buckets. We need to combine the buckets and estimate the actual AGI of a zipcode. To do this, consider the values for N1:

- 1 = 1 to 25,000
- 2 = 25,000 to 50,000
- 3 = 50,000 to 75,000
- 4 = 75,000 to 100,000
- 5 = 100,000 to 200,000
- 6 = 200,000 or more

The median of each of these ranges is approximately:

- 1 = 12,500
- 2 = 37,500
- 3 = 62,500
- 4 = 87,500
- 5 = 112,500
- 6 = 212,500

Using this, you can estimate 63017's average AGI as:

```
>>> totalCount = 4710 + 2780 + 2130 + 2010 + 5240 + 3510
>>> totalAGI = 4710 * 12500 + 2780 * 37500 + 2130 * 62500
      + 2010 * 87500 + 5240 * 112500 + 3510 * 212500
>>> print(totalAGI / totalCount)

88689.89205103042
```

We begin by reading the government data.

```
In [6]: import pandas as pd
```

```
df=pd.read_csv('https://www.irs.gov/pub/irs-soi/16zpallagi.csv')
```

First, we trim all zip codes that are either 0 or 99999. We also select the three fields that we need.

```
In [7]: df=df.loc[(df['zipcode']!=0) & (df['zipcode']!=99999),  
                 ['STATE','zipcode','agi_stub','N1']]  
  
pd.set_option('display.max_columns', 0)  
pd.set_option('display.max_rows', 10)  
  
display(df)
```

	STATE	zipcode	agi_stub	N1
<b>6</b>	AL	35004	1	1510
<b>7</b>	AL	35004	2	1410
<b>8</b>	AL	35004	3	950
<b>9</b>	AL	35004	4	650
<b>10</b>	AL	35004	5	630
...	...	...	...	...
<b>179785</b>	WY	83414	2	40
<b>179786</b>	WY	83414	3	40
<b>179787</b>	WY	83414	4	0
<b>179788</b>	WY	83414	5	40
<b>179789</b>	WY	83414	6	30

179184 rows × 4 columns

We replace all of the **agi\_stub** values with the correct median values with the **map** function.

```
In [8]: medians = {1:12500,2:37500,3:62500,4:87500,5:112500,6:212500}  
df['agi_stub']=df.agi_stub.map(medians)  
  
pd.set_option('display.max_columns', 0)  
pd.set_option('display.max_rows', 10)  
display(df)
```

	<b>STATE</b>	<b>zipcode</b>	<b>agi_stub</b>	<b>N1</b>
<b>6</b>	AL	35004	12500	1510
<b>7</b>	AL	35004	37500	1410
<b>8</b>	AL	35004	62500	950
<b>9</b>	AL	35004	87500	650
<b>10</b>	AL	35004	112500	630
...	...	...	...	...
<b>179785</b>	WY	83414	37500	40
<b>179786</b>	WY	83414	62500	40
<b>179787</b>	WY	83414	87500	0
<b>179788</b>	WY	83414	112500	40
<b>179789</b>	WY	83414	212500	30

179184 rows × 4 columns

Next, we group the data frame by zip code.

```
In [9]: groups = df.groupby(by='zipcode')
```

The program applies a lambda across the groups and calculates the AGI estimate.

```
In [11]: df = pd.DataFrame(groups.apply(
    lambda x:sum(x['N1']*x['agi_stub'])/sum(x['N1']))) \
    .reset_index()

pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 10)

display(df)
```

	zipcode	0
0	1001	52895.322940
1	1002	64528.451001
2	1003	15441.176471
3	1005	54694.092827
4	1007	63654.353562
...	...	...
<b>29867</b>	99921	48042.168675
<b>29868</b>	99922	32954.545455
<b>29869</b>	99925	45639.534884
<b>29870</b>	99926	41136.363636
<b>29871</b>	99929	45911.214953

29872 rows × 2 columns

We can now rename the new **agi\_estimate** column.

```
In [13]: df.columns = ['zipcode', 'agi_estimate']

pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 10)

display(df)
```

	zipcode	agi_estimate
0	1001	52895.322940
1	1002	64528.451001
2	1003	15441.176471
3	1005	54694.092827
4	1007	63654.353562
...	...	...
<b>29867</b>	99921	48042.168675
<b>29868</b>	99922	32954.545455
<b>29869</b>	99925	45639.534884
<b>29870</b>	99926	41136.363636
<b>29871</b>	99929	45911.214953

29872 rows × 2 columns

Finally, we check to see that our zip code of 63017 got the correct value.

```
In [14]: df[ df['zipcode']==63017 ]
```

```
Out[14]:      zipcode  agi_estimate
19909      63017  88689.892051
```



# T81-558: Applications of Deep Neural Networks

## Module 2: Python for Machine Learning

- Instructor: [Jeff Heaton](#), McKelvey School of Engineering, [Washington University in St. Louis](#)
- For more information visit the [class website](#).

## Module 2 Material

Main video lecture:

- Part 2.1: Introduction to Pandas [\[Video\]](#) [\[Notebook\]](#)
- Part 2.2: Categorical Values [\[Video\]](#) [\[Notebook\]](#)
- Part 2.3: Grouping, Sorting, and Shuffling in Python Pandas [\[Video\]](#) [\[Notebook\]](#)
- Part 2.4: Using Apply and Map in Pandas for Keras [\[Video\]](#) [\[Notebook\]](#)
- **Part 2.5: Feature Engineering in Pandas for Deep Learning in Keras** [\[Video\]](#) [\[Notebook\]](#)

## Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

```
In [1]: try:  
    %tensorflow_version 2.x  
    COLAB = True  
    print("Note: using Google CoLab")  
except:  
    print("Note: not using Google CoLab")  
    COLAB = False
```

Note: not using Google CoLab

## Part 2.5: Feature Engineering

Feature engineering is an essential part of machine learning. For now, we will manually engineer features. However, later in this course, we will see some techniques for automatic feature engineering.

## Calculated Fields

It is possible to add new fields to the data frame that your program calculates from the other fields. We can create a new column that gives the weight in kilograms. The equation to calculate a metric weight, given weight in pounds, is:

$$m_{(kg)} = m_{(lb)} \times 0.45359237$$

The following Python code performs this transformation:

```
In [2]: import os
import pandas as pd

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

df.insert(1, 'weight_kg', (df['weight'] * 0.45359237).astype(int))
pd.set_option('display.max_columns', 6)
pd.set_option('display.max_rows', 5)
df
```

```
Out[2]:      mpg  weight_kg  cylinders  ...  year  origin          name
0   18.0       1589          8  ...    70     1  chevrolet chevelle malibu
1   15.0       1675          8  ...    70     1        buick skylark 320
...
396  28.0       1190          4  ...    82     1        ford ranger
397  31.0       1233          4  ...    82     1      chevy s-10
```

398 rows × 10 columns

## Google API Keys

Sometimes you will use external APIs to obtain data. The following examples show how to use the Google API keys to encode addresses for use with neural networks. To use these, you will need your own Google API key. The key I have below is not a real key; you need to put your own there. Google will ask for a credit card, but there will be no actual cost unless you use a massive number of lookups. YOU ARE NOT required to get a Google API key for this class; this only

shows you how. If you want to get a Google API key, visit this site and obtain one for **geocode**.

You can obtain your key from this link: [Google API Keys](#).

```
In [3]: if 'GOOGLE_API_KEY' in os.environ:
    # If the API key is defined in an environmental variable,
    # the use the env variable.
    GOOGLE_KEY = os.environ['GOOGLE_API_KEY']
else:
    # If you have a Google API key of your own, you can also just
    # put it here:
    GOOGLE_KEY = 'REPLACE WITH YOUR GOOGLE API KEY'
```

## Other Examples: Dealing with Addresses

Addresses can be difficult to encode into a neural network. There are many different approaches, and you must consider how you can transform the address into something more meaningful. Map coordinates can be a good approach. [latitude and longitude](#) can be a useful encoding. Thanks to the power of the Internet, it is relatively easy to transform an address into its latitude and longitude values. The following code determines the coordinates of [Washington University](#):

```
In [4]: import requests

address = "1 Brookings Dr, St. Louis, MO 63130"

response = requests.get(
    'https://maps.googleapis.com/maps/api/geocode/json?key={}&address={}' \
    .format(GOOGLE_KEY, address))

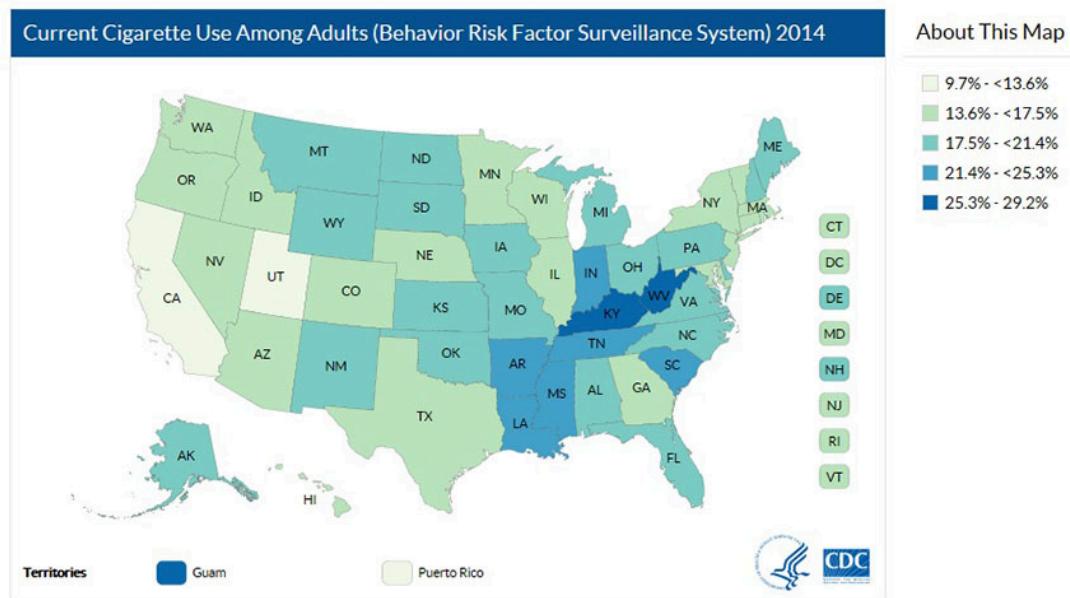
resp_json_payload = response.json()

if 'error_message' in resp_json_payload:
    print(resp_json_payload['error_message'])
else:
    print(resp_json_payload['results'][0]['geometry']['location'])

{'lat': 38.6481653, 'lng': -90.3049506}
```

They might not be overly helpful if you feed latitude and longitude into the neural network as two features. These two values would allow your neural network to cluster locations on a map. Sometimes cluster locations on a map can be useful. Figure 2.SMK shows the percentage of the population that smokes in the USA by state.

**Figure 2.SMK: Smokers by State**



The above map shows that certain behaviors, like smoking, can be clustered by the global region.

However, often you will want to transform the coordinates into distances. It is reasonably easy to estimate the distance between any two points on Earth by using the [great circle distance](#) between any two points on a sphere:

The following code implements this formula:

$$\Delta\sigma = \arccos(\sin \phi_1 \cdot \sin \phi_2 + \cos \phi_1 \cdot \cos \phi_2 \cdot \cos(\Delta\lambda))$$

$$d = r \Delta\sigma$$

```
In [5]: from math import sin, cos, sqrt, atan2, radians

URL='https://maps.googleapis.com' + \
    '/maps/api/geocode/json?key={}&address={}'"

# Distance function
def distance_lat_lng(lat1,lng1,lat2,lng2):
    # approximate radius of earth in km
    R = 6373.0

    # degrees to radians (lat/lon are in degrees)
    lat1 = radians(lat1)
    lng1 = radians(lng1)
    lat2 = radians(lat2)
    lng2 = radians(lng2)

    dlng = lng2 - lng1
    dlat = lat2 - lat1

    a = sin(dlat / 2)**2 + cos(lat1) * cos(lat2) * sin(dlng / 2)**2
```

```

    c = 2 * atan2(sqrt(a), sqrt(1 - a))

    return R * c

# Find lat lon for address
def lookup_lat_lng(address):
    response = requests.get( \
        URL.format(GOOGLE_KEY,address))
    json = response.json()
    if len(json['results']) == 0:
        raise ValueError("Google API error on: {}".format(address))
    map = json['results'][0]['geometry']['location']
    return map['lat'],map['lng']

# Distance between two locations

import requests

address1 = "1 Brookings Dr, St. Louis, MO 63130"
address2 = "3301 College Ave, Fort Lauderdale, FL 33314"

lat1, lng1 = lookup_lat_lng(address1)
lat2, lng2 = lookup_lat_lng(address2)

print("Distance, St. Louis, MO to Ft. Lauderdale, FL: {} km".format(
    distance_lat_lng(lat1,lng1,lat2,lng2)))

```

Distance, St. Louis, MO to Ft. Lauderdale, FL: 1685.3019808607426 km

Distances can be a useful means to encode addresses. It would help if you considered what distance might be helpful for your dataset. Consider:

- Distance to a major metropolitan area
- Distance to a competitor
- Distance to a distribution center
- Distance to a retail outlet

The following code calculates the distance between 10 universities and Washington University in St. Louis:

In [6]: # Encoding other universities by their distance to Washington University

```

schools = [
    ["Princeton University, Princeton, NJ 08544", 'Princeton'],
    ["Massachusetts Hall, Cambridge, MA 02138", 'Harvard'],
    ["5801 S Ellis Ave, Chicago, IL 60637", 'University of Chicago'],
    ["Yale, New Haven, CT 06520", 'Yale'],
    ["116th St & Broadway, New York, NY 10027", 'Columbia University'],
    ["450 Serra Mall, Stanford, CA 94305", 'Stanford'],
    ["77 Massachusetts Ave, Cambridge, MA 02139", 'MIT'],
    ["Duke University, Durham, NC 27708", 'Duke University'],
    ["University of Pennsylvania, Philadelphia, PA 19104",
     'University of Pennsylvania'],

```

```
[ "Johns Hopkins University, Baltimore, MD 21218", 'Johns Hopkins' ]  
  
lat1, lng1 = lookup_lat_lng("1 Brookings Dr, St. Louis, MO 63130")  
  
for address, name in schools:  
    lat2,lng2 = lookup_lat_lng(address)  
    dist = distance_lat_lng(lat1,lng1,lat2,lng2)  
    print("School '{}', distance to wustl is: {}".format(name,dist))
```

```
School 'Princeton', distance to wustl is: 1354.4830895052746  
School 'Harvard', distance to wustl is: 1670.6297027161022  
School 'University of Chicago', distance to wustl is: 418.0815972177934  
School 'Yale', distance to wustl is: 1508.217831712127  
School 'Columbia University', distance to wustl is: 1418.2264083295695  
School 'Stanford', distance to wustl is: 2780.6829398114114  
School 'MIT', distance to wustl is: 1672.4444489665696  
School 'Duke University', distance to wustl is: 1046.7970984423719  
School 'University of Pennsylvania', distance to wustl is: 1307.19541200423  
School 'Johns Hopkins', distance to wustl is: 1184.3831076555425
```

In [ ]:

# T81-558: Applications of Deep Neural Networks

## Module 3: Introduction to TensorFlow

- Instructor: [Jeff Heaton](#), McKelvey School of Engineering, [Washington University in St. Louis](#)
- For more information visit the [class website](#).

## Module 3 Material

- **Part 3.1: Deep Learning and Neural Network Introduction** [\[Video\]](#) [\[Notebook\]](#)
- Part 3.2: Introduction to Tensorflow and Keras [\[Video\]](#) [\[Notebook\]](#)
- Part 3.3: Saving and Loading a Keras Neural Network [\[Video\]](#) [\[Notebook\]](#)
- Part 3.4: Early Stopping in Keras to Prevent Overfitting [\[Video\]](#) [\[Notebook\]](#)
- Part 3.5: Extracting Weights and Manual Calculation [\[Video\]](#) [\[Notebook\]](#)

## Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

```
In [1]: try:  
    %tensorflow_version 2.x  
    COLAB = True  
    print("Note: using Google CoLab")  
except:  
    print("Note: not using Google CoLab")  
    COLAB = False
```

## Part 3.1: Deep Learning and Neural Network Introduction

Neural networks were one of the first machine learning models. Their popularity has fallen twice and is now on its third rise. Deep learning implies the use of neural networks. The "deep" in deep learning refers to a neural network with

many hidden layers. Because neural networks have been around for so long, they have quite a bit of baggage. Researchers have created many different training algorithms, activation/transfer functions, and structures. This course is only concerned with the latest, most current state-of-the-art techniques for deep neural networks. I will not spend much time discussing the history of neural networks.

Neural networks accept input and produce output. The input to a neural network is called the feature vector. The size of this vector is always a fixed length. Changing the size of the feature vector usually means recreating the entire neural network. Though the feature vector is called a "vector," this is not always the case. A vector implies a 1D array. Later we will learn about convolutional neural networks (CNNs), which can allow the input size to change without retraining the neural network. Historically the input to a neural network was always 1D. However, with modern neural networks, you might see input data, such as:

- **1D vector** - Classic input to a neural network, similar to rows in a spreadsheet. Common in predictive modeling.
- **2D Matrix** - Grayscale image input to a CNN.
- **3D Matrix** - Color image input to a CNN.
- **nD Matrix** - Higher-order input to a CNN.

Before CNNs, programs either encoded images to an intermediate form or sent the image input to a neural network by merely squashing the image matrix into a long array by placing the image's rows side-by-side. CNNs are different as the matrix passes through the neural network layers.

Initially, this book will focus on 1D input to neural networks. However, later modules will focus more heavily on higher dimension input.

The term dimension can be confusing in neural networks. In the sense of a 1D input vector, dimension refers to how many elements are in that 1D array. For example, a neural network with ten input neurons has ten dimensions. However, now that we have CNNs, the input has dimensions. The input to the neural network will *usually* have 1, 2, or 3 dimensions. Four or more dimensions are unusual. You might have a 2D input to a neural network with 64x64 pixels. This configuration would result in 4,096 input neurons. This network is either 2D or 4,096D, depending on which dimensions you reference.

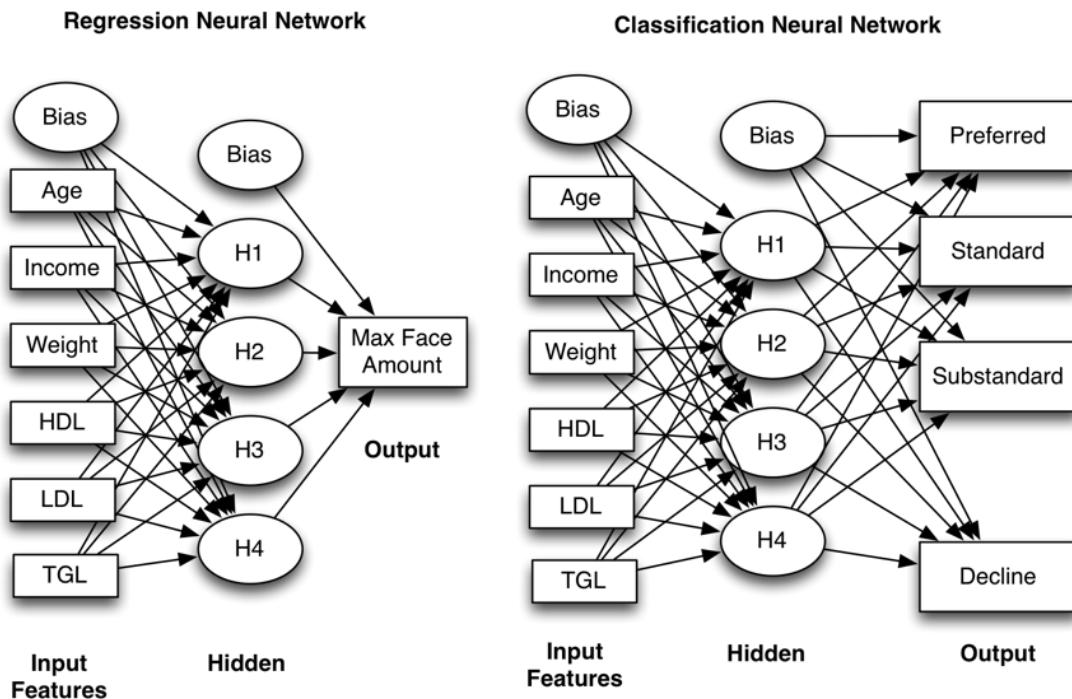
## Classification or Regression

Like many models, neural networks can function in classification or regression:

- **Regression** - You expect a number as your neural network's prediction.
- **Classification** - You expect a class/category as your neural network's prediction.

A classification and regression neural network is shown by Figure 3.CLS-REG.

**Figure 3.CLS-REG: Neural Network Classification and Regression**



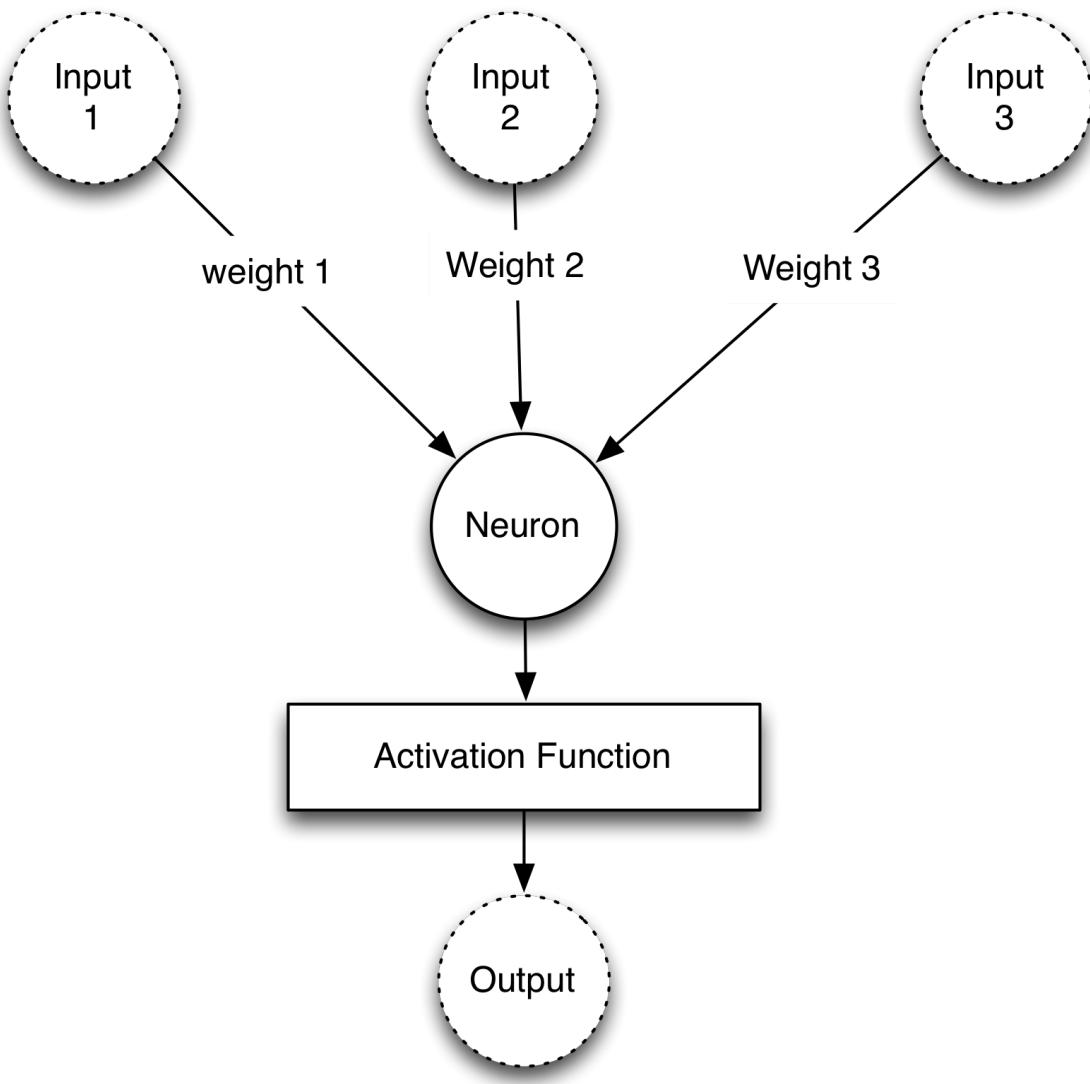
Notice that the output of the regression neural network is numeric, and the classification output is a class. Regression, or two-class classification, networks always have a single output. Classification neural networks have an output neuron for each category.

## Neurons and Layers

Most neural network structures use some type of neuron. Many different neural networks exist, and programmers introduce experimental neural network structures. Consequently, it is not possible to cover every neural network architecture. However, there are some commonalities among neural network implementations. A neural network algorithm would typically be composed of individual, interconnected units, even though these units may or may not be called neurons. The name for a neural network processing unit varies among the literature sources. It could be called a node, neuron, or unit.

A diagram shows the abstract structure of a single artificial neuron in Figure 3.ANN.

**Figure 3.ANN: An Artificial Neuron**



The artificial neuron receives input from one or more sources that may be other neurons or data fed into the network from a computer program. This input is usually floating-point or binary. Often binary input is encoded to floating-point by representing true or false as 1 or 0. Sometimes the program also depicts the binary information using a bipolar system with true as one and false as -1.

An artificial neuron multiplies each of these inputs by a weight. Then it adds these multiplications and passes this sum to an activation function. Some neural networks do not use an activation function. The following equation summarizes the calculated output of a neuron:

$$f(x, w) = \phi\left(\sum_i (\theta_i \cdot x_i)\right)$$

In the above equation, the variables  $x$  and  $\theta$  represent the input and weights of the neuron. The variable  $i$  corresponds to the number of weights and inputs. You must always have the same number of weights as inputs. The neural network

multiplies each weight by its respective input and feeds the products of these multiplications into an activation function, denoted by the Greek letter  $\phi$  (phi). This process results in a single output from the neuron.

The above neuron has two inputs plus the bias as a third. This neuron might accept the following input feature vector:

$$[1, 2]$$

Because a bias neuron is present, the program should append the value of one as follows:

$$[1, 2, 1]$$

The weights for a 3-input layer (2 real inputs + bias) will always have additional weight for the bias. A weight vector might be:

$$[0.1, 0.2, 0.3]$$

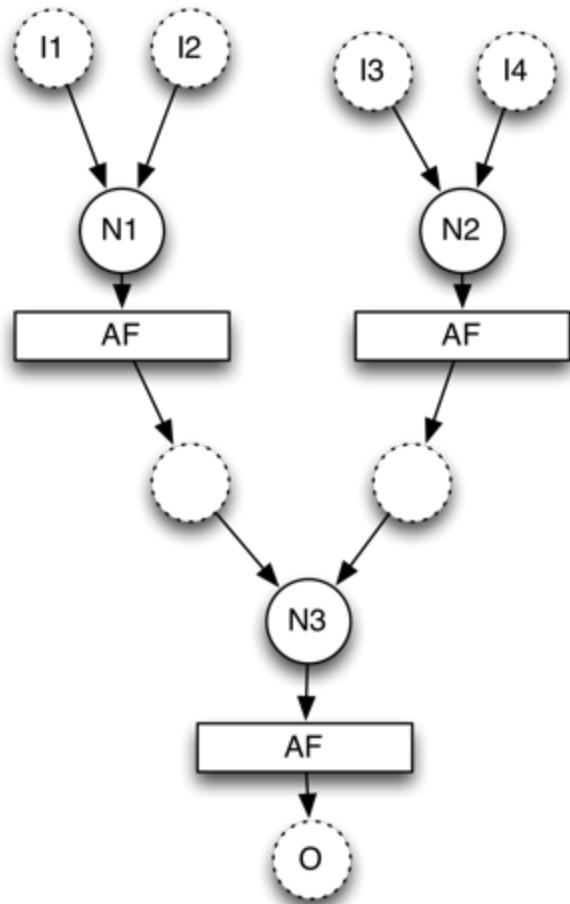
To calculate the summation, perform the following:

$$0.1 * 1 + 0.2 * 2 + 0.3 * 1 = 0.8$$

The program passes a value of 0.8 to the  $\phi$  (phi) function, representing the activation function.

The above figure shows the structure with just one building block. You can chain together many artificial neurons to build an artificial neural network (ANN). Think of the artificial neurons as building blocks for which the input and output circles are the connectors. Figure 3.ANN-3 shows an artificial neural network composed of three neurons:

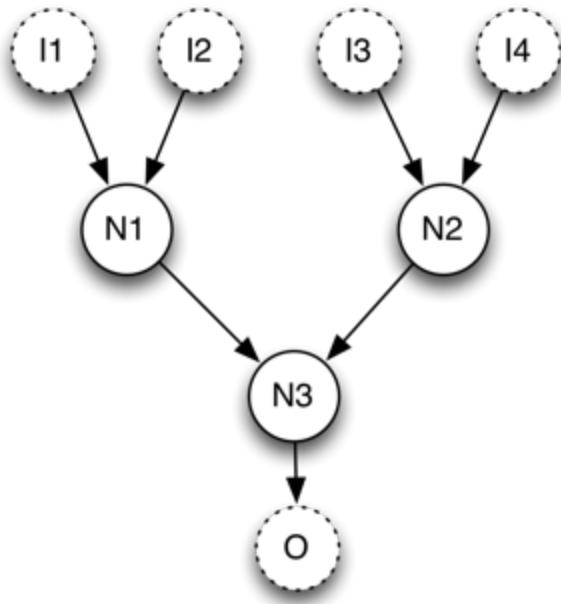
**Figure 3.ANN-3: Three Neuron Neural Network**



The above diagram shows three interconnected neurons. This representation is essentially this figure, minus a few inputs, repeated three times and then connected. It also has a total of four inputs and a single output. The output of neurons **N1** and **N2** feed **N3** to produce the output **O**. To calculate the output for this network, we perform the previous equation three times. The first two times calculate **N1** and **N2**, and the third calculation uses the output of **N1** and **N2** to calculate **N3**.

Neural network diagrams do not typically show the detail seen in the previous figure. We can omit the activation functions and intermediate outputs to simplify the chart, resulting in Figure 3.SANN-3.

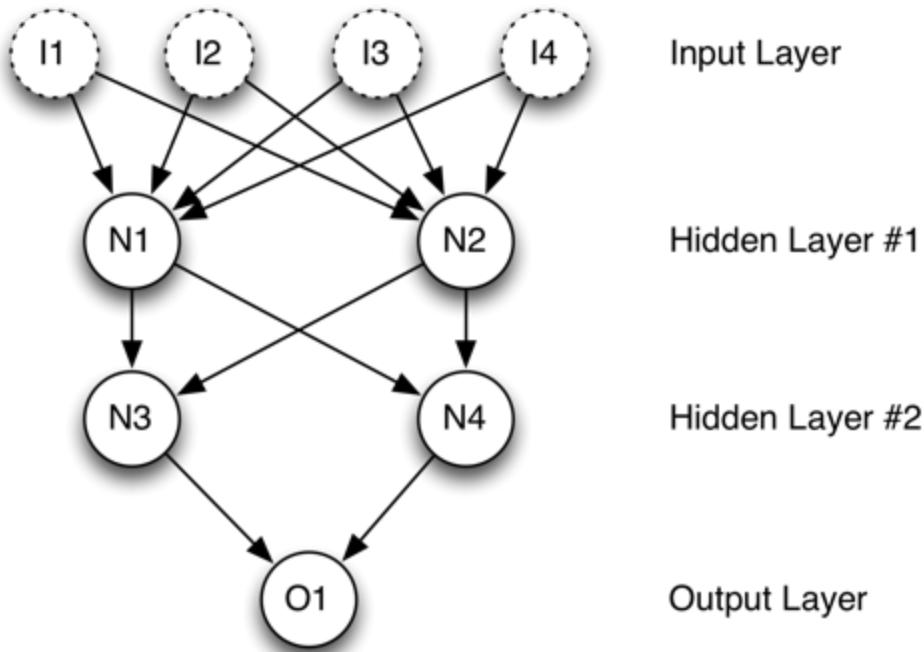
**Figure 3.SANN-3: Three Neuron Neural Network**



Looking at the previous figure, you can see two additional components of neural networks. First, consider the graph represents the inputs and outputs as abstract dotted line circles. The input and output could be parts of a more extensive neural network. However, the input and output are often a particular type of neuron that accepts data from the computer program using the neural network. The output neurons return a result to the program. This type of neuron is called an input neuron. We will discuss these neurons in the next section. This figure shows the neurons arranged in layers. The input neurons are the first layer, the **N1** and **N2** neurons create the second layer, the third layer contains **N3**, and the fourth layer has **O**. Most neural networks arrange neurons into layers.

The neurons that form a layer share several characteristics. First, every neuron in a layer has the same activation function. However, the activation functions employed by each layer may be different. Each of the layers fully connects to the next layer. In other words, every neuron in one layer has a connection to neurons in the previous layer. The former figure is not fully connected. Several layers are missing connections. For example, **I1** and **N2** do not connect. The next neural network in Figure 3.F-ANN is fully connected and has an additional layer.

**Figure 3.F-ANN: Fully Connected Neural Network Diagram**



In this figure, you see a fully connected, multilayered neural network. Networks such as this one will always have an input and output layer. The hidden layer structure determines the name of the network architecture. The network in this figure is a two-hidden-layer network. Most networks will have between zero and two hidden layers. Without implementing deep learning strategies, networks with more than two hidden layers are rare.

You might also notice that the arrows always point downward or forward from the input to the output. Later in this course, we will see recurrent neural networks that form inverted loops among the neurons. This type of neural network is called a feedforward neural network.

## Types of Neurons

In the last section, we briefly introduced the idea that different types of neurons exist. Not every neural network will use every kind of neuron. It is also possible for a single neuron to fill the role of several different neuron types. Now we will explain all the neuron types described in the course.

There are usually four types of neurons in a neural network:

- **Input Neurons** - We map each input neuron to one element in the feature vector.
- **Hidden Neurons** - Hidden neurons allow the neural network to be abstract and process the input into the output.
- **Output Neurons** - Each output neuron calculates one part of the output.

- **Bias Neurons** - Work similar to the y-intercept of a linear equation.

We place each neuron into a layer:

- **Input Layer** - The input layer accepts feature vectors from the dataset. Input layers usually have a bias neuron.
- **Output Layer** - The output from the neural network. The output layer does not have a bias neuron.
- **Hidden Layers** - Layers between the input and output layers. Each hidden layer will usually have a bias neuron.

## Input and Output Neurons

Nearly every neural network has input and output neurons. The input neurons accept data from the program for the network. The output neuron provides processed data from the network back to the program. The program will group these input and output neurons into separate layers called the input and output layers. The program normally represents the input to a neural network as an array or vector. The number of elements contained in the vector must equal the number of input neurons. For example, a neural network with three input neurons might accept the following input vector:

$$[0.5, 0.75, 0.2]$$

Neural networks typically accept floating-point vectors as their input. To be consistent, we will represent the output of a single output neuron network as a single-element vector. Likewise, neural networks will output a vector with a length equal to the number of output neurons. The output will often be a single value from a single output neuron.

## Hidden Neurons

Hidden neurons have two essential characteristics. First, hidden neurons only receive input from other neurons, such as input or other hidden neurons. Second, hidden neurons only output to other neurons, such as output or other hidden neurons. Hidden neurons help the neural network understand the input and form the output. Programmers often group hidden neurons into fully connected hidden layers. However, these hidden layers do not directly process the incoming data or the eventual output.

A common question for programmers concerns the number of hidden neurons in a network. Since the answer to this question is complex, more than one section of the course will include a relevant discussion of the number of hidden neurons. Before deep learning, researchers generally suggested that anything more than

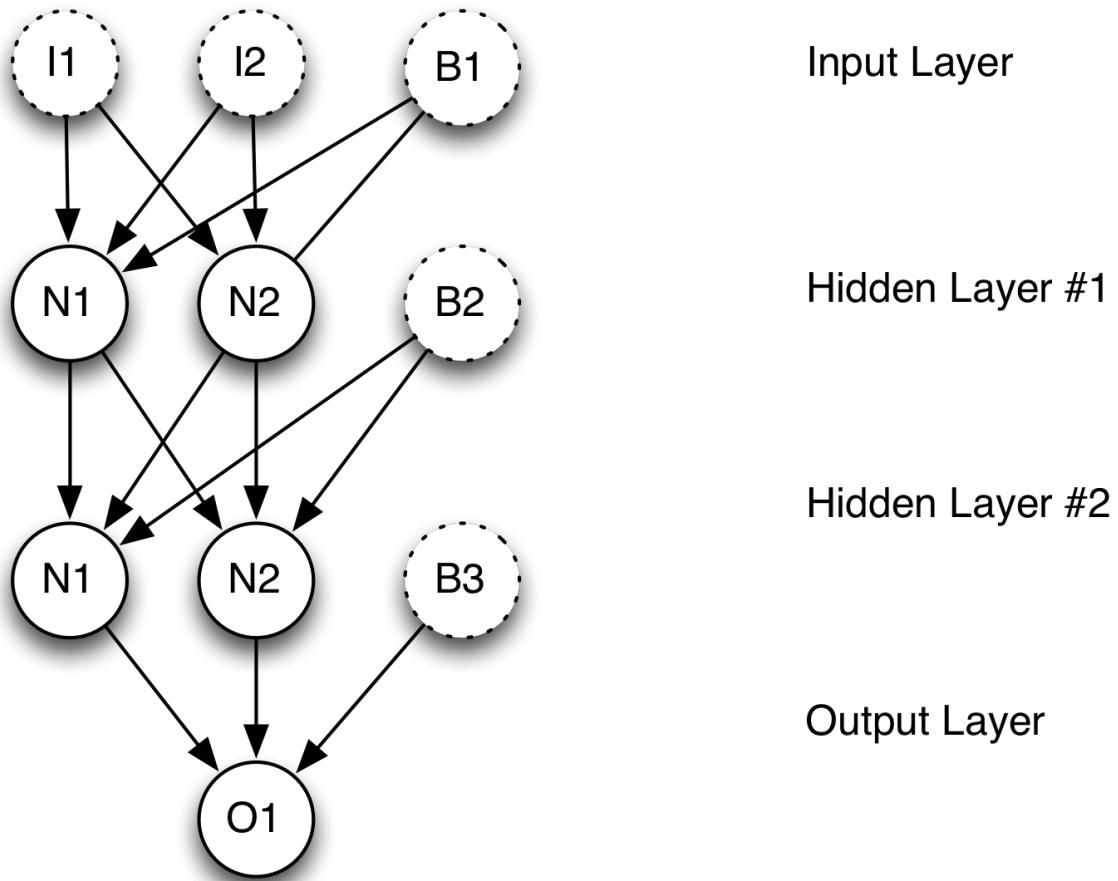
a single hidden layer is excessive. [Cite:hornik1989multilayer] Researchers have proven that a single-hidden-layer neural network can function as a universal approximator. In other words, this network should be able to learn to produce (or approximate) any output from any input as long as it has enough hidden neurons in a single layer.

Training refers to the process that determines good weight values. Before the advent of deep learning, researchers feared additional layers would lengthen training time or encourage overfitting. Both concerns are true; however, increased hardware speeds and clever techniques can mitigate these concerns. Before researchers introduced deep learning techniques, we did not have an efficient way to train a deep network, which is a neural network with many hidden layers. Although a single-hidden-layer neural network can theoretically learn anything, deep learning facilitates a more complex representation of patterns in the data.

## Bias Neurons

Programmers add bias neurons to neural networks to help them learn patterns. Bias neurons function like an input neuron that always produces a value of 1. Because the bias neurons have a constant output of 1, they are not connected to the previous layer. The value of 1, called the bias activation, can be set to values other than 1. However, 1 is the most common bias activation. Not all neural networks have bias neurons. Figure 3.BIAS shows a single-hidden-layer neural network with bias neurons:

**Figure 3.BIAS: Neural Network with Bias Neurons**



The above network contains three bias neurons. Except for the output layer, every level includes a single bias neuron. Bias neurons allow the program to shift the output of an activation function. We will see precisely how this shifting occurs later in the module when discussing activation functions.

## Other Neuron Types

The individual units that comprise a neural network are not always called neurons. Researchers will sometimes refer to these neurons as nodes, units, or summations. You will almost always construct neural networks of weighted connections between these units.

## Why are Bias Neurons Needed?

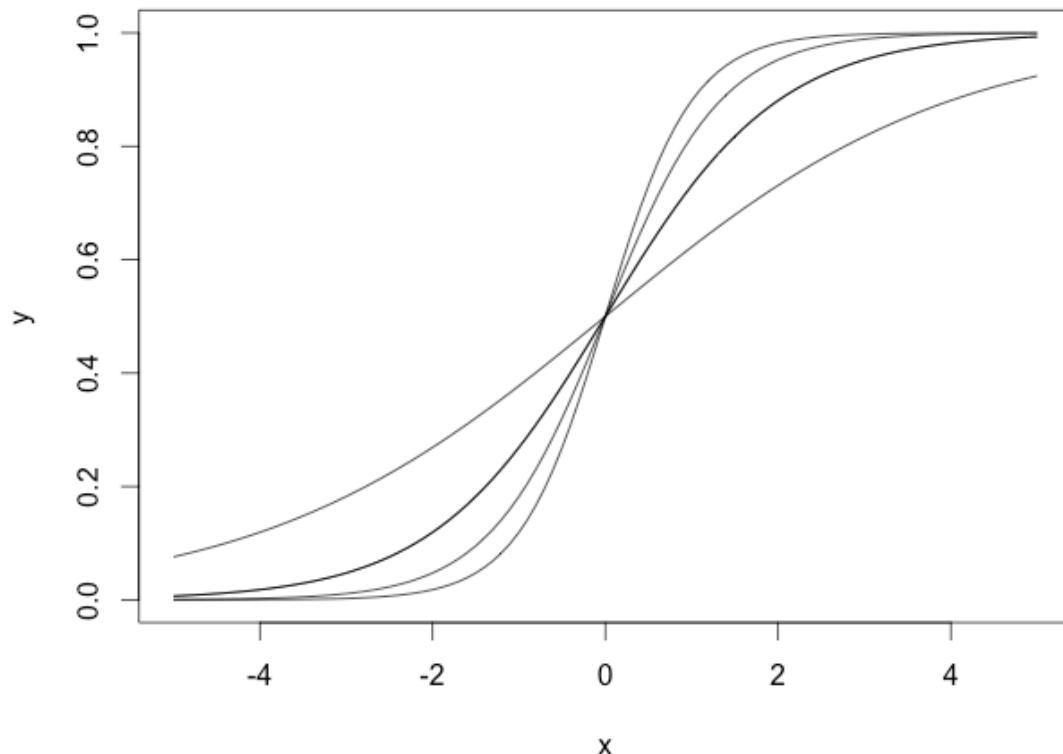
The activation functions from the previous section specify the output of a single neuron. Together, the weight and bias of a neuron shape the output of the activation to produce the desired output. To see how this process occurs, consider the following equation. It represents a single-input sigmoid activation neural network.

$$f(x, w, b) = \frac{1}{1 + e^{-(wx+b)}}$$

The  $x$  variable represents the single input to the neural network. The  $w$  and  $b$  variables specify the weight and bias of the neural network. The above equation combines the weighted sum of the inputs and the sigmoid activation function. For this section, we will consider the sigmoid function because it demonstrates a bias neuron's effect.

The weights of the neuron allow you to adjust the slope or shape of the activation function. Figure 3.A-WEIGHT shows the effect on the output of the sigmoid activation function if the weight is varied:

**Figure 3.A-WEIGHT: Neuron Weight Shifting**



The above diagram shows several sigmoid curves using the following parameters:

$$f(x, 0.5, 0.0)$$

$$f(x, 1.0, 0.0)$$

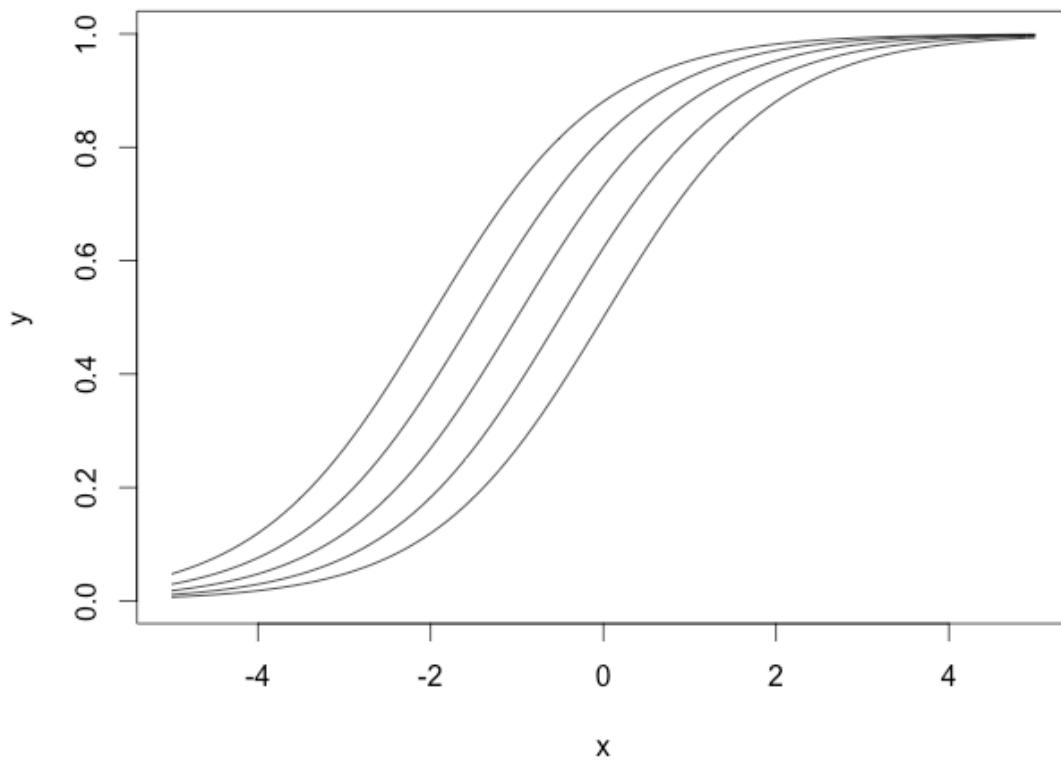
$$f(x, 1.5, 0.0)$$

$$f(x, 2.0, 0.0)$$

We did not use bias to produce the curves, which is evident in the third parameter of 0 in each case. Using four weight values yields four different sigmoid curves in the above figure. No matter the weight, we always get the same value of 0.5 when  $x$  is 0 because all curves hit the same point when  $x$  is 0. We might need the neural network to produce other values when the input is near 0.5.

Bias does shift the sigmoid curve, which allows values other than 0.5 when  $x$  is near 0. Figure 3.A-BIAS shows the effect of using a weight of 1.0 with several different biases:

**Figure 3.A-BIAS: Neuron Bias Shifting**



The above diagram shows several sigmoid curves with the following parameters:

$$f(x, 1.0, 1.0)$$

$$f(x, 1.0, 0.5)$$

$$f(x, 1.0, 1.5)$$

$$f(x, 1.0, 2.0)$$

We used a weight of 1.0 for these curves in all cases. When we utilized several different biases, sigmoid curves shifted to the left or right. Because all the curves merge at the top right or bottom left, it is not a complete shift.

When we put bias and weights together, they produced a curve that created the necessary output. The above curves are the output from only one neuron. In a complete network, the output from many different neurons will combine to produce intricate output patterns.

## Modern Activation Functions

Activation functions, also known as transfer functions, are used to calculate the output of each layer of a neural network. Historically neural networks have used a hyperbolic tangent, sigmoid/logistic, or linear activation function. However, modern deep neural networks primarily make use of the following activation functions:

- **Rectified Linear Unit (ReLU)** - Used for the output of hidden layers.  
[\[Cite:glorot2011deep\]](#)
- **Softmax** - Used for the output of classification neural networks.
- **Linear** - Used for the output of regression neural networks (or 2-class classification).

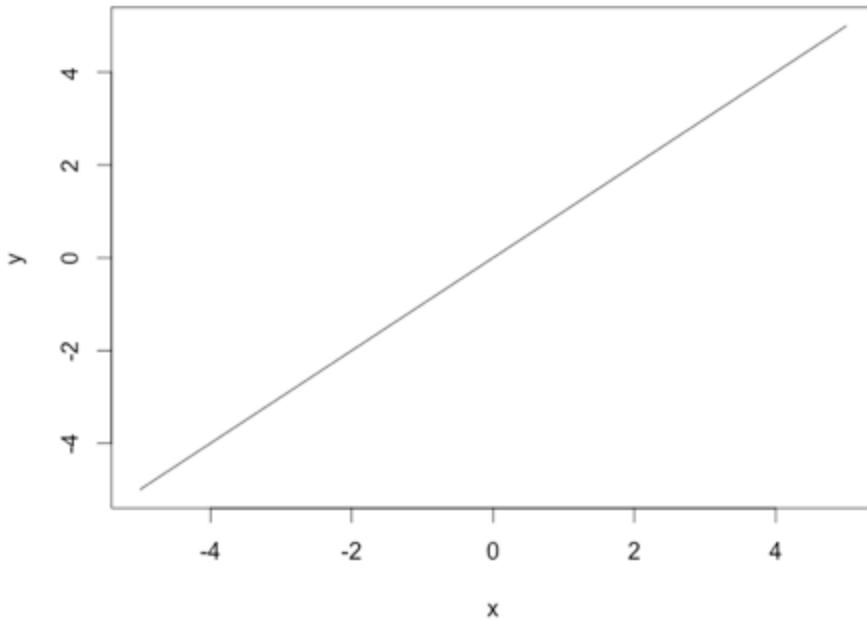
## Linear Activation Function

The most basic activation function is the linear function because it does not change the neuron output. The following equation 1.2 shows how the program typically implements a linear activation function:

$$\phi(x) = x$$

As you can observe, this activation function simply returns the value that the neuron inputs passed to it. Figure 3.LIN shows the graph for a linear activation function:

**Figure 3.LIN: Linear Activation Function**



Regression neural networks, which learn to provide numeric values, will usually use a linear activation function on their output layer. Classification neural networks, which determine an appropriate class for their input, will often utilize a softmax activation function for their output layer.

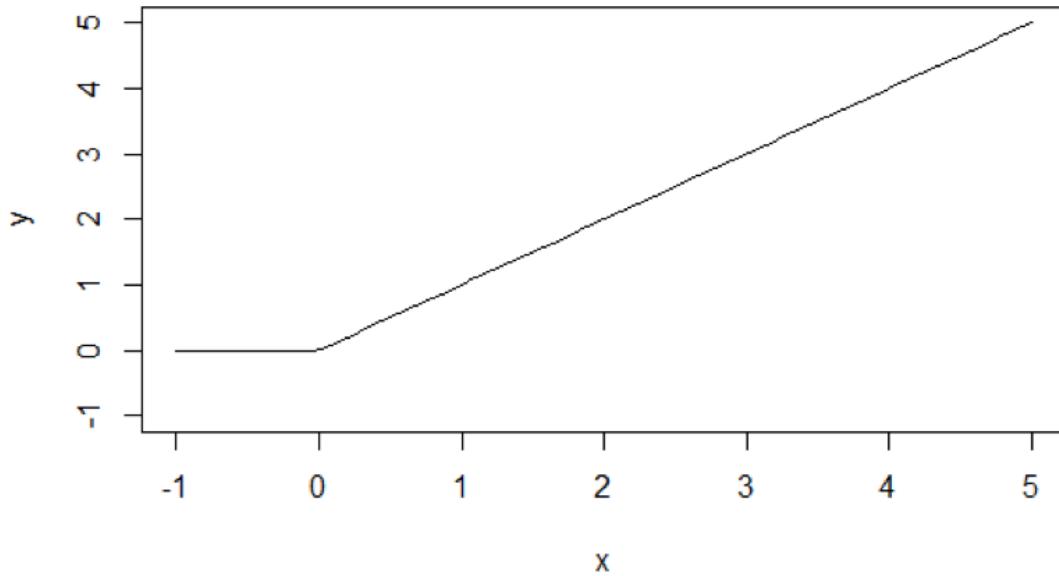
## Rectified Linear Units (ReLU)

Since its introduction, researchers have rapidly adopted the rectified linear unit (ReLU). [\[Cite:nair2010rectified\]](#) Before the ReLU activation function, the programmers generally regarded the hyperbolic tangent as the activation function of choice. Most current research now recommends the ReLU due to superior training results. As a result, most neural networks should utilize the ReLU on hidden layers and either softmax or linear on the output layer. The following equation shows the straightforward ReLU function:

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

Figure 3.RELU shows the graph of the ReLU activation function:

**Figure 3.RELU: Rectified Linear Units (ReLU)**



Most current research states that the hidden layers of your neural network should use the ReLU activation.

## Softmax Activation Function

The final activation function that we will examine is the softmax activation function. Along with the linear activation function, you can usually find the softmax function in the output layer of a neural network. Classification neural networks typically employ the softmax function. The neuron with the highest value claims the input as a member of its class. Because it is a preferable method, the softmax activation function forces the neural network's output to represent the probability that the input falls into each of the classes. The neuron's outputs are numeric values without the softmax, with the highest indicating the winning class.

To see how the program uses the softmax activation function, we will look at a typical neural network classification problem. The iris data set contains four measurements for 150 different iris flowers. Each of these flowers belongs to one of three species of iris. When you provide the measurements of a flower, the softmax function allows the neural network to give you the probability that these measurements belong to each of the three species. For example, the neural network might tell you that there is an 80% chance that the iris is setosa, a 15% probability that it is virginica, and only a 5% probability of versicolor. Because these are probabilities, they must add up to 100%. There could not be an 80%

probability of setosa, a 75% probability of virginica, and a 20% probability of versicolor—this type of result would be nonsensical.

To classify input data into one of three iris species, you will need one output neuron for each species. The output neurons do not inherently specify the probability of each of the three species. Therefore, it is desirable to provide probabilities that sum to 100%. The neural network will tell you the likelihood of a flower being each of the three species. To get the probability, use the softmax function in the following equation:

$$\phi_i(x) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

In the above equation,  $i$  represents the index of the output neuron ( $\phi$ ) that the program is calculating, and  $j$  represents the indexes of all neurons in the group/level. The variable  $x$  designates the array of output neurons. It's important to note that the program calculates the softmax activation differently than the other activation functions in this module. When softmax is the activation function, the output of a single neuron is dependent on the other output neurons.

To see the softmax function in operation, refer to this [Softmax example website](#).

Consider a trained neural network that classifies data into three categories: the three iris species. In this case, you would use one output neuron for each of the target classes. Consider if the neural network were to output the following:

- **Neuron 1:** setosa: 0.9
- **Neuron 2:** versicolour: 0.2
- **Neuron 3:** virginica: 0.4

The above output shows that the neural network considers the data to represent a setosa iris. However, these numbers are not probabilities. The 0.9 value does not represent a 90% likelihood of the data representing a setosa. These values sum to 1.5. For the program to treat them as probabilities, they must sum to 1.0. The output vector for this neural network is the following:

$$[0.9, 0.2, 0.4]$$

If you provide this vector to the softmax function it will return the following vector:

$$[0.47548495534876745, 0.2361188410001125, 0.28839620365112]$$

The above three values do sum to 1.0 and can be treated as probabilities. The likelihood of the data representing a setosa iris is 48% because the first value in

the vector rounds to 0.48 (48%). You can calculate this value in the following manner:

$$sum = \exp(0.9) + \exp(0.2) + \exp(0.4) = 5.17283056695839$$

$$j_0 = \exp(0.9)/sum = 0.47548495534876745$$

$$j_1 = \exp(0.2)/sum = 0.2361188410001125$$

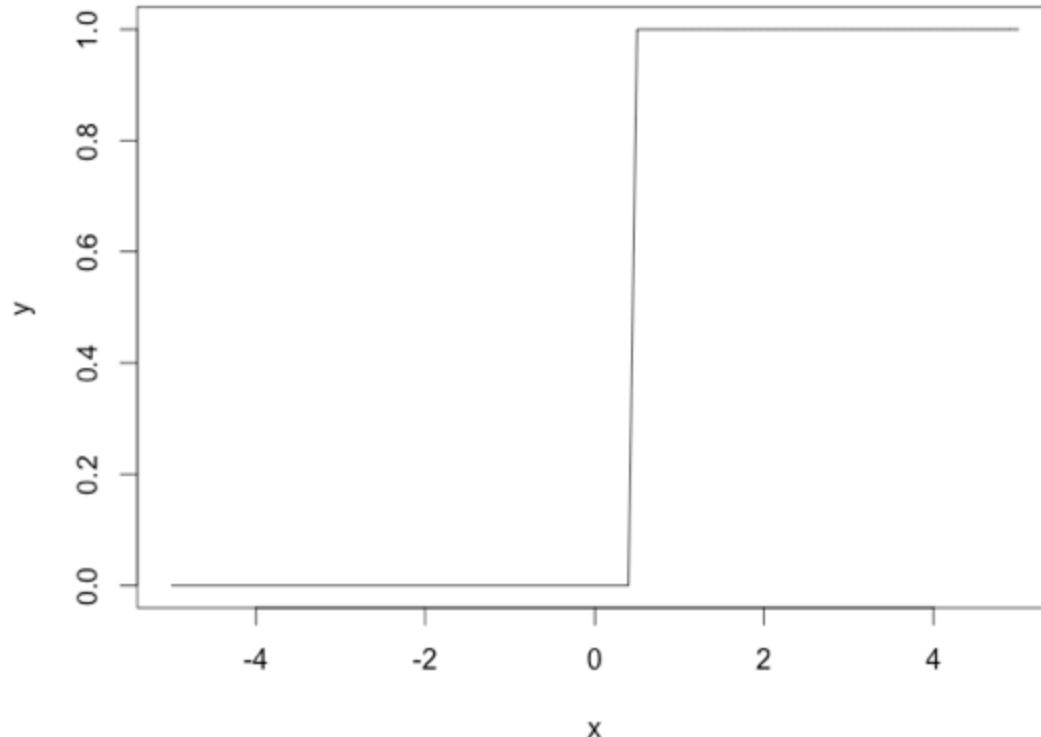
$$j_2 = \exp(0.4)/sum = 0.28839620365112$$

## Step Activation Function

The step or threshold activation function is another simple activation function. Neural networks were initially called perceptrons. McCulloch & Pitts (1943) introduced the original perceptron and used a step activation function like the following equation:[[Cite:mcculloch1943logical](#)] The step activation is 1 if  $x \geq 0.5$ , and 0 otherwise.

This equation outputs a value of 1.0 for incoming values of 0.5 or higher and 0 for all other values. Step functions, also known as threshold functions, only return 1 (true) for values above the specified threshold, as seen in Figure 3.STEP.

**Figure 3.STEP: Step Activation Function**



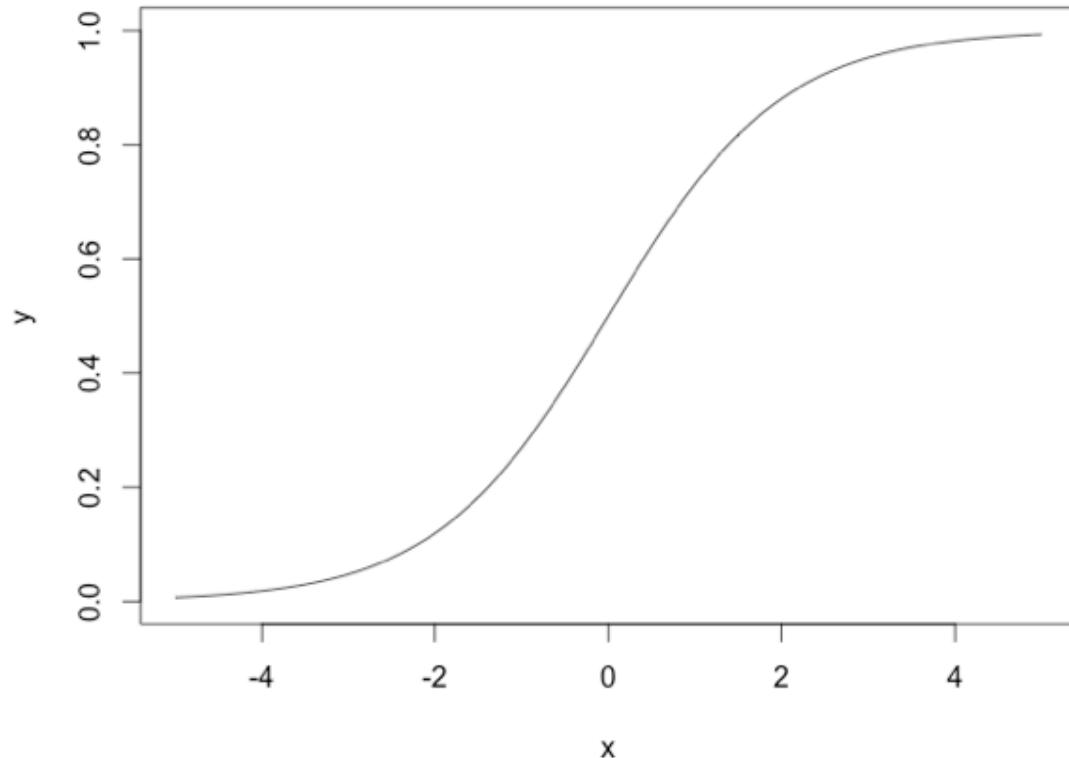
# Sigmoid Activation Function

The sigmoid or logistic activation function is a common choice for feedforward neural networks that need to output only positive numbers. Despite its widespread use, the hyperbolic tangent or the rectified linear unit (ReLU) activation function is usually a more suitable choice. We introduce the ReLU activation function later in this module. The following equation shows the sigmoid activation function:

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

Use the sigmoid function to ensure that values stay within a relatively small range, as seen in Figure 3.SIGMOID:

**Figure 3.SIGMOID: Sigmoid Activation Function**



As you can see from the above graph, we can force values to a range. Here, the function compressed values above or below 0 to the approximate range between 0 and 1.

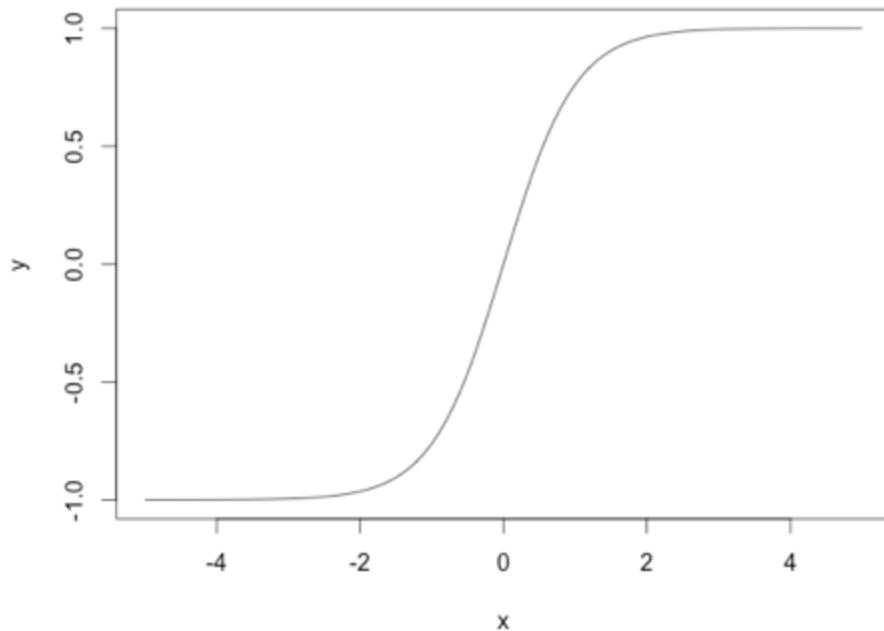
# Hyperbolic Tangent Activation Function

The hyperbolic tangent function is also a prevalent activation function for neural networks that must output values between -1 and 1. This activation function is simply the hyperbolic tangent (tanh) function, as shown in the following equation:

$$\phi(x) = \tanh(x)$$

The graph of the hyperbolic tangent function has a similar shape to the sigmoid activation function, as seen in Figure 3.HTAN.

**Figure 3.HTAN: Hyperbolic Tangent Activation Function**

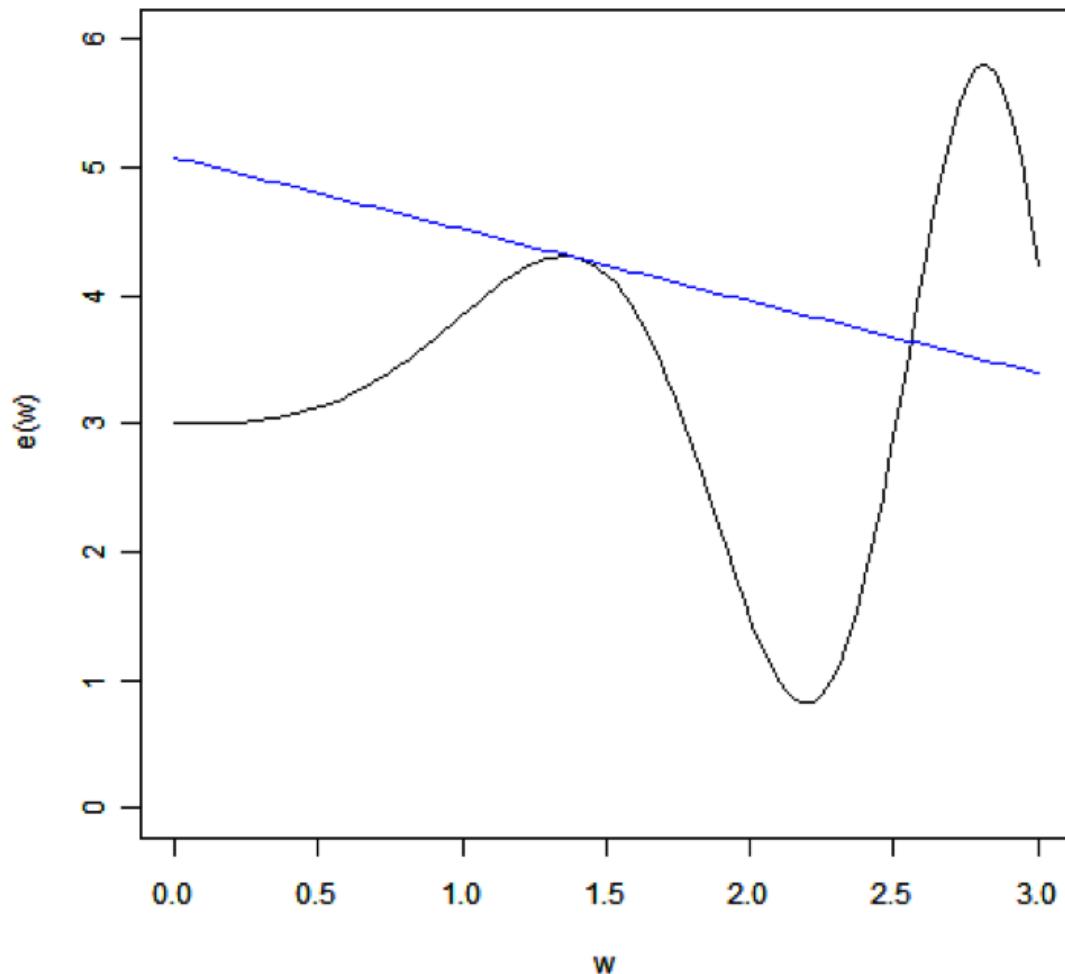


The hyperbolic tangent function has several advantages over the sigmoid activation function.

## Why ReLU?

Why is the ReLU activation function so popular? One of the critical improvements to neural networks makes deep learning work. [\[Cite:nair2010rectified\]](#) Before deep learning, the sigmoid activation function was prevalent. We covered the sigmoid activation function earlier in this module. Frameworks like Keras often train neural networks with gradient descent. For the neural network to use gradient descent, it is necessary to take the derivative of the activation function. The program must derive partial derivatives of each of the weights for the error function. Figure 3.DERV shows a derivative, the instantaneous rate of change.

**Figure 3.DERV: Derivative**



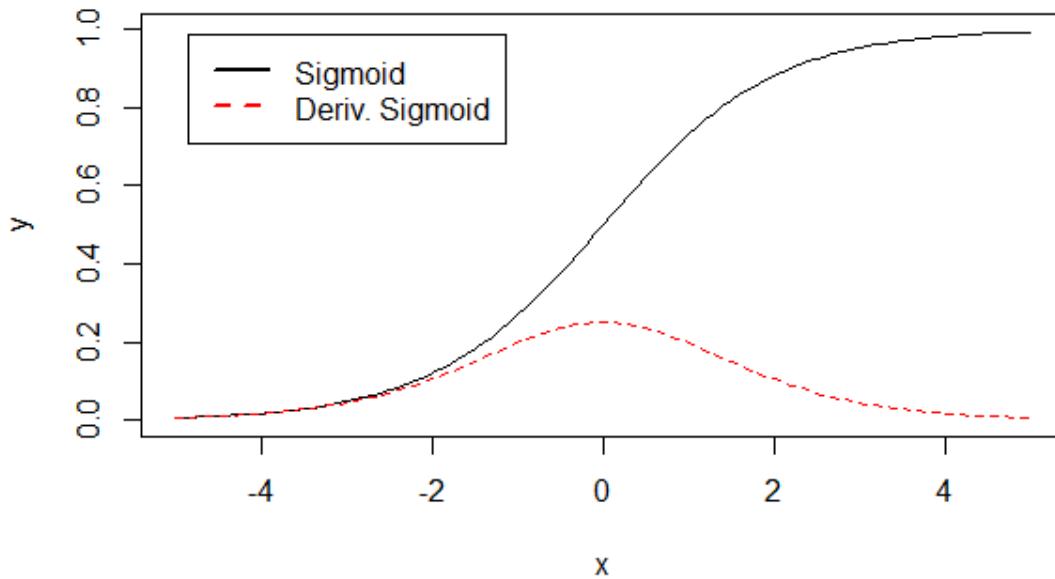
The derivative of the sigmoid function is given here:

$$\phi'(x) = \phi(x)(1 - \phi(x))$$

Textbooks often give this derivative in other forms. We use the above form for computational efficiency. To see how we determined this derivative, [refer to the following article](#).

We present the graph of the sigmoid derivative in Figure 3.SDERV.

**Figure 3.SDERV: Sigmoid Derivative**



The derivative quickly saturates to zero as  $x$  moves from zero. This is not a problem for the derivative of the ReLU, which is given here:

$$\phi'(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$$

## Module 3 Assignment

You can find the first assignment here: [assignment 3](#)

In [ ]:

# T81-558: Applications of Deep Neural Networks

## Module 3: Introduction to TensorFlow

- Instructor: [Jeff Heaton](#), McKelvey School of Engineering, [Washington University in St. Louis](#)
- For more information visit the [class website](#).

## Module 3 Material

- Part 3.1: Deep Learning and Neural Network Introduction [\[Video\]](#) [\[Notebook\]](#)
- **Part 3.2: Introduction to Tensorflow and Keras** [\[Video\]](#) [\[Notebook\]](#)
- Part 3.3: Saving and Loading a Keras Neural Network [\[Video\]](#) [\[Notebook\]](#)
- Part 3.4: Early Stopping in Keras to Prevent Overfitting [\[Video\]](#) [\[Notebook\]](#)
- Part 3.5: Extracting Weights and Manual Calculation [\[Video\]](#) [\[Notebook\]](#)

## Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

```
In [28]: try:  
    %tensorflow_version 2.x  
    COLAB = True  
    print("Note: using Google CoLab")  
except:  
    print("Note: not using Google CoLab")  
    COLAB = False
```

Note: not using Google CoLab

## Part 3.2: Introduction to Tensorflow and Keras

TensorFlow [\[Cite:GoogleTensorFlow\]](#) is an open-source software library for machine learning in various kinds of perceptual and language understanding tasks. It is currently used for research and production by different teams in many

commercial Google products, such as speech recognition, Gmail, Google Photos, and search, many of which had previously used its predecessor DistBelief. TensorFlow was originally developed by the Google Brain team for Google's research and production purposes and later released under the Apache 2.0 open source license on November 9, 2015.

- [TensorFlow Homepage](#)
- [TensorFlow GitHub](#)
- [TensorFlow Google Groups Support](#)
- [TensorFlow Google Groups Developer Discussion](#)
- [TensorFlow FAQ](#)

## Why TensorFlow

- Supported by Google
- Works well on Windows, Linux, and Mac
- Excellent GPU support
- Python is an easy to learn programming language
- Python is extremely popular in the data science community

## Deep Learning Tools

TensorFlow is not the only game in town. The biggest competitor to TensorFlow/Keras is PyTorch. Listed below are some of the deep learning toolkits actively being supported:

- **TensorFlow** - Google's deep learning API. The focus of this class, along with Keras.
- **Keras** - Acts as a higher-level to Tensorflow.
- **PyTorch** - PyTorch is an open-source machine learning library based on the Torch library, used for computer vision and natural language applications processing. Facebook's AI Research lab primarily develops PyTorch.

Other deep learning tools:

- **Deeplearning4J** - Java-based. Supports all major platforms. GPU support in Java!
- **H2O** - Java-based.

In my opinion, the two primary Python libraries for deep learning are PyTorch and Keras. Generally, PyTorch requires more lines of code to perform the deep learning applications presented in this course. This trait of PyTorch gives Keras an easier learning curve than PyTorch. However, if you are creating entirely new

neural network structures in a research setting, PyTorch can make for easier access to some of the low-level internals of deep learning.

## Using TensorFlow Directly

Most of the time in the course, we will communicate with TensorFlow using Keras [Cite:francois2017deep], which allows you to specify the number of hidden layers and create the neural network. TensorFlow is a low-level mathematics API, similar to [Numpy](#). However, unlike Numpy, TensorFlow is built for deep learning. TensorFlow compiles these compute graphs into highly efficient C++/[CUDA](#) code.

## TensorFlow Linear Algebra Examples

TensorFlow is a library for linear algebra. Keras is a higher-level abstraction for neural networks that you build upon TensorFlow. In this section, I will demonstrate some basic linear algebra that directly employs TensorFlow and does not use Keras. First, we will see how to multiply a row and column matrix.

```
In [29]: import tensorflow as tf

# Create a Constant op that produces a 1x2 matrix.  The op is
# added as a node to the default graph.
#
# The value returned by the constructor represents the output
# of the Constant op.
matrix1 = tf.constant([[3., 3.]])

# Create another Constant that produces a 2x1 matrix.
matrix2 = tf.constant([[2.],[2.]])  

# Create a Matmul op that takes 'matrix1' and 'matrix2' as inputs.
# The returned value, 'product', represents the result of the matrix
# multiplication.
product = tf.matmul(matrix1, matrix2)

print(product)
print(float(product))

tf.Tensor([12.], shape=(1, 1), dtype=float32)
12.0
```

This example multiplied two TensorFlow constant tensors. Next, we will see how to subtract a constant from a variable.

```
In [30]: import tensorflow as tf

x = tf.Variable([1.0, 2.0])
a = tf.constant([3.0, 3.0])
```

```
# Add an op to subtract 'a' from 'x'. Run it and print the result
sub = tf.subtract(x, a)
print(sub)
print(sub.numpy())
# ==> [-2. -1.]
```

```
tf.Tensor([-2. -1.], shape=(2,), dtype=float32)
[-2. -1.]
```

Of course, variables are only useful if their values can be changed. The program can accomplish this change in value by calling the `assign` function.

```
In [31]: x.assign([4.0, 6.0])
```

```
Out[31]: <tf.Variable 'UnreadVariable' shape=(2,) dtype=float32, numpy=array([4., 6.], dtype=float32)>
```

The program can now perform the subtraction with this new value.

```
In [32]: sub = tf.subtract(x, a)
print(sub)
print(sub.numpy())
```

```
tf.Tensor([1. 3.], shape=(2,), dtype=float32)
[1. 3.]
```

In the next section, we will see a TensorFlow example that has nothing to do with neural networks.

## TensorFlow Mandelbrot Set Example

Next, we examine another example where we use TensorFlow directly. To demonstrate that TensorFlow is mathematical and does not only provide neural networks, we will also first use it for a non-machine learning rendering task. The code presented here can render a [Mandelbrot set](#).

```
In [33]: import tensorflow as tf
import numpy as np

import PIL.Image
from io import BytesIO
from IPython.display import Image, display

def render(a):
    a_cyclic = (a*0.3).reshape(list(a.shape)+[1])
    img = np.concatenate([10+20*np.cos(a_cyclic),
                         30+50*np.sin(a_cyclic),
                         155-80*np.cos(a_cyclic)], 2)
    img[a==a.max()] = 0
    a = img
    a = np.uint8(np.clip(a, 0, 255))
    f = BytesIO()
    return PIL.Image.fromarray(a)
```

```

#@tf.function
def mandelbrot_helper(grid_c, current_values, counts, cycles):

    for i in range(cycles):
        temp = current_values*current_values + grid_c
        not_diverged = tf.abs(temp) < 4
        current_values.assign(temp),
        counts.assign_add(tf.cast(not_diverged, tf.float32))

def mandelbrot(render_size, center, zoom, cycles):
    f = zoom/render_size[0]
    real_start = center[0]-(render_size[0]/2)*f
    real_end = real_start + render_size[0]*f
    imag_start = center[1]-(render_size[1]/2)*f
    imag_end = imag_start + render_size[1]*f

    real_range = tf.range(real_start, real_end, f, dtype=tf.float64)
    imag_range = tf.range(imag_start, imag_end, f, dtype=tf.float64)
    real, imag = tf.meshgrid(real_range, imag_range)
    grid_c = tf.constant(tf.complex(real, imag))
    current_values = tf.Variable(grid_c)
    counts = tf.Variable(tf.zeros_like(grid_c, tf.float32))

    mandelbrot_helper(grid_c, current_values, counts, cycles)
    return counts.numpy()

```

With the above code defined, we can now calculate and render a Mandlebrot plot.

```

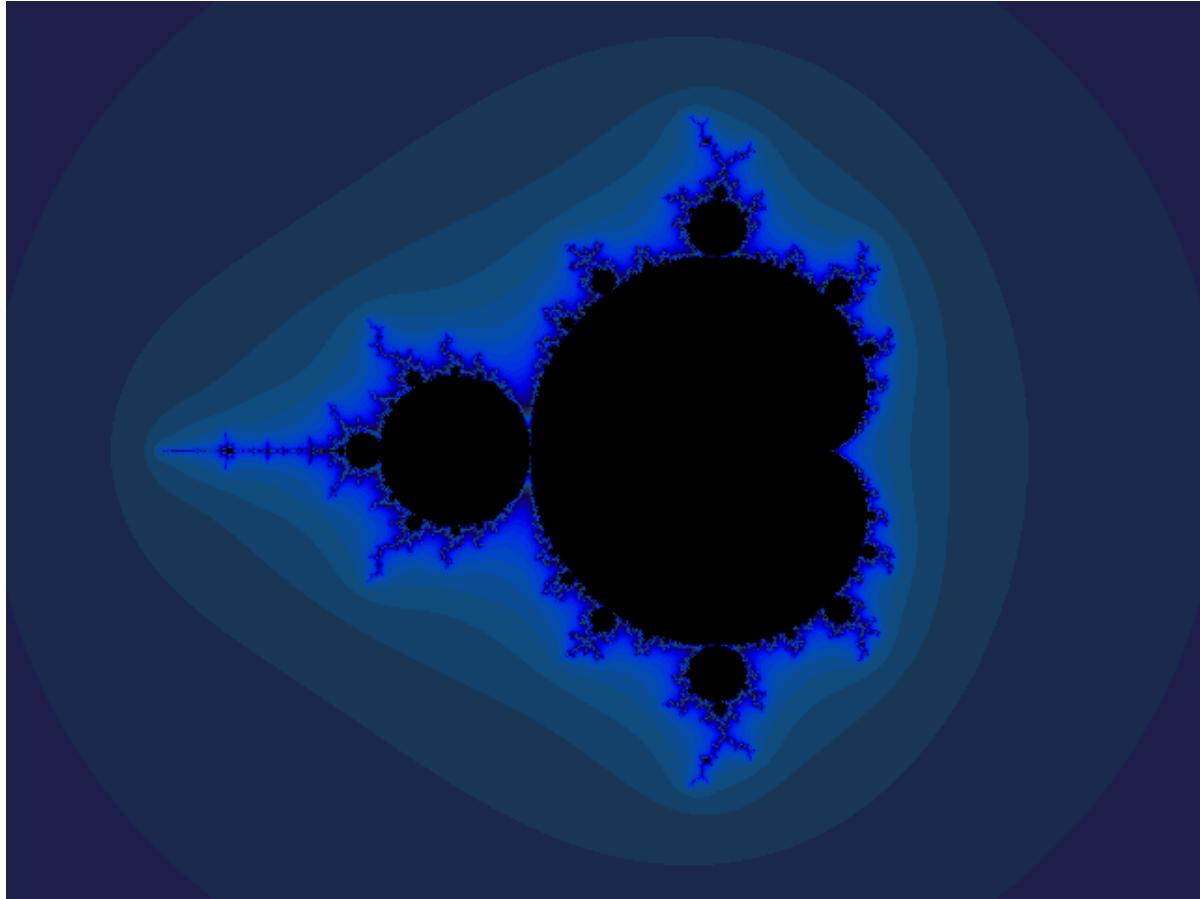
In [34]: counts = mandelbrot(
    #render_size=(3840,2160), # 4K
    #render_size=(1920,1080), # HD
    render_size=(640,480),
    center=(-0.5,0),
    zoom=4,
    cycles=200
)
img = render(counts)
print(img.size)
img



```

(640, 480)

Out[34]:



Mandlebrot rendering programs are both simple and infinitely complex at the same time. This view shows the entire Mandlebrot universe simultaneously, as a view completely zoomed out. However, if you zoom in on any non-black portion of the plot, you will find infinite hidden complexity.

## Introduction to Keras

[Keras](#) is a layer on top of Tensorflow that makes it much easier to create neural networks. Rather than define the graphs, as you see above, you set the individual layers of the network with a much more high-level API. Unless you are researching entirely new structures of deep neural networks, it is unlikely that you need to program TensorFlow directly.

**For this class, we will usually use TensorFlow through Keras, rather than direct TensorFlow**

## Simple TensorFlow Regression: MPG

This example shows how to encode the MPG dataset for regression and predict values. We will see if we can predict the miles per gallon (MPG) for a car based on the car's weight, cylinders, engine size, and other features.

```
In [35]: from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
import pandas as pd
import io
import os
import requests
import numpy as np
from sklearn import metrics

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

cars = df['name']

# Handle missing value
df['horsepower'] = df['horsepower'].fillna(df['horsepower'].median())

# Pandas to Numpy
x = df[['cylinders', 'displacement', 'horsepower', 'weight',
         'acceleration', 'year', 'origin']].values
y = df['mpg'].values # regression

# Build the neural network
model = Sequential()
model.add(Dense(25, input_dim=x.shape[1], activation='relu')) # Hidden 1
model.add(Dense(10, activation='relu')) # Hidden 2
model.add(Dense(1)) # Output
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(x,y,verbose=2,epochs=100)
```

Train on 398 samples  
Epoch 1/100  
398/398 - 0s - loss: 359076.9421  
Epoch 2/100  
398/398 - 0s - loss: 74176.6153  
Epoch 3/100  
398/398 - 0s - loss: 9767.8802  
Epoch 4/100  
398/398 - 0s - loss: 1427.8838  
Epoch 5/100  
398/398 - 0s - loss: 2057.3394  
Epoch 6/100  
398/398 - 0s - loss: 1513.2297  
Epoch 7/100  
398/398 - 0s - loss: 1295.3801  
Epoch 8/100  
398/398 - 0s - loss: 1272.4230  
Epoch 9/100  
398/398 - 0s - loss: 1239.1273  
Epoch 10/100  
398/398 - 0s - loss: 1207.5261  
Epoch 11/100  
398/398 - 0s - loss: 1183.4229  
Epoch 12/100  
398/398 - 0s - loss: 1154.0081  
Epoch 13/100  
398/398 - 0s - loss: 1124.9607  
Epoch 14/100  
398/398 - 0s - loss: 1099.9529  
Epoch 15/100  
398/398 - 0s - loss: 1076.2990  
Epoch 16/100  
398/398 - 0s - loss: 1060.9813  
Epoch 17/100  
398/398 - 0s - loss: 1047.1127  
Epoch 18/100  
398/398 - 0s - loss: 1035.2276  
Epoch 19/100  
398/398 - 0s - loss: 1023.7794  
Epoch 20/100  
398/398 - 0s - loss: 1012.4624  
Epoch 21/100  
398/398 - 0s - loss: 1001.8168  
Epoch 22/100  
398/398 - 0s - loss: 992.2781  
Epoch 23/100  
398/398 - 0s - loss: 979.8584  
Epoch 24/100  
398/398 - 0s - loss: 969.5121  
Epoch 25/100  
398/398 - 0s - loss: 958.6515  
Epoch 26/100  
398/398 - 0s - loss: 950.3542  
Epoch 27/100  
398/398 - 0s - loss: 941.1478  
Epoch 28/100

398/398 - 0s - loss: 926.8202  
Epoch 29/100  
398/398 - 0s - loss: 917.2789  
Epoch 30/100  
398/398 - 0s - loss: 905.2999  
Epoch 31/100  
398/398 - 0s - loss: 892.8540  
Epoch 32/100  
398/398 - 0s - loss: 884.2463  
Epoch 33/100  
398/398 - 0s - loss: 871.6899  
Epoch 34/100  
398/398 - 0s - loss: 864.9260  
Epoch 35/100  
398/398 - 0s - loss: 849.8424  
Epoch 36/100  
398/398 - 0s - loss: 840.7186  
Epoch 37/100  
398/398 - 0s - loss: 836.8501  
Epoch 38/100  
398/398 - 0s - loss: 824.8931  
Epoch 39/100  
398/398 - 0s - loss: 810.3899  
Epoch 40/100  
398/398 - 0s - loss: 800.1687  
Epoch 41/100  
398/398 - 0s - loss: 784.4148  
Epoch 42/100  
398/398 - 0s - loss: 774.2326  
Epoch 43/100  
398/398 - 0s - loss: 762.1020  
Epoch 44/100  
398/398 - 0s - loss: 751.2068  
Epoch 45/100  
398/398 - 0s - loss: 740.0900  
Epoch 46/100  
398/398 - 0s - loss: 728.2087  
Epoch 47/100  
398/398 - 0s - loss: 719.4002  
Epoch 48/100  
398/398 - 0s - loss: 710.2463  
Epoch 49/100  
398/398 - 0s - loss: 695.5551  
Epoch 50/100  
398/398 - 0s - loss: 686.5956  
Epoch 51/100  
398/398 - 0s - loss: 676.1623  
Epoch 52/100  
398/398 - 0s - loss: 661.9155  
Epoch 53/100  
398/398 - 0s - loss: 652.7001  
Epoch 54/100  
398/398 - 0s - loss: 649.1527  
Epoch 55/100  
398/398 - 0s - loss: 626.8102  
Epoch 56/100

398/398 - 0s - loss: 617.8689  
Epoch 57/100  
398/398 - 0s - loss: 610.6475  
Epoch 58/100  
398/398 - 0s - loss: 592.5715  
Epoch 59/100  
398/398 - 0s - loss: 582.9929  
Epoch 60/100  
398/398 - 0s - loss: 571.0975  
Epoch 61/100  
398/398 - 0s - loss: 560.9386  
Epoch 62/100  
398/398 - 0s - loss: 549.7743  
Epoch 63/100  
398/398 - 0s - loss: 542.2976  
Epoch 64/100  
398/398 - 0s - loss: 524.7271  
Epoch 65/100  
398/398 - 0s - loss: 512.6835  
Epoch 66/100  
398/398 - 0s - loss: 501.3199  
Epoch 67/100  
398/398 - 0s - loss: 489.1912  
Epoch 68/100  
398/398 - 0s - loss: 476.9358  
Epoch 69/100  
398/398 - 0s - loss: 465.7011  
Epoch 70/100  
398/398 - 0s - loss: 454.4821  
Epoch 71/100  
398/398 - 0s - loss: 444.3988  
Epoch 72/100  
398/398 - 0s - loss: 432.2656  
Epoch 73/100  
398/398 - 0s - loss: 420.4270  
Epoch 74/100  
398/398 - 0s - loss: 408.0120  
Epoch 75/100  
398/398 - 0s - loss: 400.3801  
Epoch 76/100  
398/398 - 0s - loss: 384.8745  
Epoch 77/100  
398/398 - 0s - loss: 373.1397  
Epoch 78/100  
398/398 - 0s - loss: 364.7359  
Epoch 79/100  
398/398 - 0s - loss: 349.1083  
Epoch 80/100  
398/398 - 0s - loss: 336.7873  
Epoch 81/100  
398/398 - 0s - loss: 329.1152  
Epoch 82/100  
398/398 - 0s - loss: 315.6647  
Epoch 83/100  
398/398 - 0s - loss: 302.7605  
Epoch 84/100

```
398/398 - 0s - loss: 292.5489
Epoch 85/100
398/398 - 0s - loss: 280.1445
Epoch 86/100
398/398 - 0s - loss: 268.6837
Epoch 87/100
398/398 - 0s - loss: 260.4212
Epoch 88/100
398/398 - 0s - loss: 250.5896
Epoch 89/100
398/398 - 0s - loss: 247.6127
Epoch 90/100
398/398 - 0s - loss: 230.2208
Epoch 91/100
398/398 - 0s - loss: 218.4122
Epoch 92/100
398/398 - 0s - loss: 208.7282
Epoch 93/100
398/398 - 0s - loss: 200.0094
Epoch 94/100
398/398 - 0s - loss: 189.9866
Epoch 95/100
398/398 - 0s - loss: 182.8754
Epoch 96/100
398/398 - 0s - loss: 175.6958
Epoch 97/100
398/398 - 0s - loss: 172.0255
Epoch 98/100
398/398 - 0s - loss: 156.4483
Epoch 99/100
398/398 - 0s - loss: 149.1817
Epoch 100/100
398/398 - 0s - loss: 142.9415
```

Out[35]: <tensorflow.python.keras.callbacks.History at 0x7f3ee44468d0>

## Introduction to Neural Network Hyperparameters

If you look at the above code, you will see that the neural network contains four layers. The first layer is the input layer because it contains the **input\_dim** parameter that the programmer sets to be the number of inputs the dataset has. The network needs one input neuron for every column in the data set (including dummy variables).

There are also several hidden layers, with 25 and 10 neurons each. You might be wondering how the programmer chose these numbers. Selecting a hidden neuron structure is one of the most common questions about neural networks. Unfortunately, there is no right answer. These are hyperparameters. They are settings that can affect neural network performance, yet there are no clearly defined means of setting them.

In general, more hidden neurons mean more capability to fit complex problems. However, too many neurons can lead to overfitting and lengthy training times. Too few can lead to underfitting the problem and will sacrifice accuracy. Also, how many layers you have is another hyperparameter. In general, more layers allow the neural network to perform more of its feature engineering and data preprocessing. But this also comes at the expense of training times and the risk of overfitting. In general, you will see that neuron counts start larger near the input layer and tend to shrink towards the output layer in a triangular fashion.

Some techniques use machine learning to optimize these values. These will be discussed in [Module 8.3](#).

## Controlling the Amount of Output

The program produces one line of output for each training epoch. You can eliminate this output by setting the verbose setting of the fit command:

- **verbose=0** - No progress output (use with Jupyter if you do not want output).
- **verbose=1** - Display progress bar, does not work well with Jupyter.
- **verbose=2** - Summary progress output (use with Jupyter if you want to know the loss at each epoch).

## Regression Prediction

Next, we will perform actual predictions. The program assigns these predictions to the **pred** variable. These are all MPG predictions from the neural network.

Notice that this is a 2D array? You can always see the dimensions of what Keras returns by printing out **pred.shape**. Neural networks can return multiple values, so the result is always an array. Here the neural network only returns one value per prediction (there are 398 cars, so 398 predictions). However, a 2D range is needed because the neural network has the potential of returning more than one value.

```
In [36]: pred = model.predict(x)
print(f"Shape: {pred.shape}")
print(pred[0:10])
```

```
Shape: (398, 1)
[[22.639828]
 [20.882801]
 [19.801853]
 [20.337807]
 [21.1946 ]
 [23.72337 ]
 [21.285397]
 [21.545208]
 [21.873882]
 [19.303974]]
```

We would like to see how good these predictions are. We know the correct MPG for each car so we can measure how close the neural network was.

```
In [37]: # Measure RMSE error. RMSE is common for regression.
score = np.sqrt(metrics.mean_squared_error(pred,y))
print(f"Final score (RMSE): {score}")
```

```
Final score (RMSE): 11.862759295790802
```

The number printed above is the average number of predictions above or below the expected output. We can also print out the first ten cars with predictions and actual MPG.

```
In [38]: # Sample predictions
for i in range(10):
    print(f"{i+1}. Car name: {cars[i]}, MPG: {y[i]}, "
          + f"predicted MPG: {pred[i]}")
```

1. Car name: chevrolet chevelle malibu, MPG: 18.0, predicted MPG: [22.639828]
2. Car name: buick skylark 320, MPG: 15.0, predicted MPG: [20.882801]
3. Car name: plymouth satellite, MPG: 18.0, predicted MPG: [19.801853]
4. Car name: amc rebel sst, MPG: 16.0, predicted MPG: [20.337807]
5. Car name: ford torino, MPG: 17.0, predicted MPG: [21.1946]
6. Car name: ford galaxie 500, MPG: 15.0, predicted MPG: [23.72337]
7. Car name: chevrolet impala, MPG: 14.0, predicted MPG: [21.285397]
8. Car name: plymouth fury iii, MPG: 14.0, predicted MPG: [21.545208]
9. Car name: pontiac catalina, MPG: 14.0, predicted MPG: [21.873882]
10. Car name: amc ambassador dpl, MPG: 15.0, predicted MPG: [19.303974]

## Simple TensorFlow Classification: Iris

Classification is how a neural network attempts to classify the input into one or more classes. The simplest way of evaluating a classification network is to track the percentage of training set items classified incorrectly. We typically score human results in this manner. For example, you might have taken multiple-choice exams in school in which you had to shade in a bubble for choices A, B, C, or D. If you chose the wrong letter on a 10-question exam, you would earn a 90%. In the same way, we can grade computers; however, most classification algorithms do not merely choose A, B, C, or D. Computers typically report a

classification as their percent confidence in each class. Figure 3.EXAM shows how a computer and a human might respond to question number 1 on an exam.

### Figure 3.EXAM: Classification Neural Network Output



As you can see, the human test taker marked the first question as "B." However, the computer test taker had an 80% (0.8) confidence in "B" and was also somewhat sure with 10% (0.1) on "A." The computer then distributed the remaining points to the other two. In the simplest sense, the machine would get 80% of the score for this question if the correct answer were "B." The computer would get only 5% (0.05) of the points if the correct answer were "D."

We previously saw how to train a neural network to predict the MPG of a car. Based on four measurements, we will now see how to predict a class, such as the type of iris flower. The code to classify iris flowers is similar to MPG; however, there are several important differences:

- The output neuron count matches the number of classes (in the case of Iris, 3).
- The Softmax transfer function is utilized by the output layer.\* The loss function is cross entropy.

```
In [39]: import pandas as pd
import io
import requests
import numpy as np
from sklearn import metrics
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/iris.csv",
    na_values=['NA', '?'])

# Convert to numpy - Classification
x = df[['sepal_l', 'sepal_w', 'petal_l', 'petal_w']].values
dummies = pd.get_dummies(df['species']) # Classification
species = dummies.columns
y = dummies.values

# Build neural network
model = Sequential()
model.add(Dense(50, input_dim=x.shape[1], activation='relu')) # Hidden 1
model.add(Dense(25, activation='relu')) # Hidden 2
model.add(Dense(y.shape[1], activation='softmax')) # Output
```

```
model.compile(loss='categorical_crossentropy', optimizer='adam')
model.fit(x,y,verbose=2,epochs=100)
```

Train on 150 samples  
Epoch 1/100  
150/150 - 0s - loss: 1.4756  
Epoch 2/100  
150/150 - 0s - loss: 1.3098  
Epoch 3/100  
150/150 - 0s - loss: 1.1715  
Epoch 4/100  
150/150 - 0s - loss: 1.0646  
Epoch 5/100  
150/150 - 0s - loss: 0.9959  
Epoch 6/100  
150/150 - 0s - loss: 0.9439  
Epoch 7/100  
150/150 - 0s - loss: 0.8898  
Epoch 8/100  
150/150 - 0s - loss: 0.8446  
Epoch 9/100  
150/150 - 0s - loss: 0.8176  
Epoch 10/100  
150/150 - 0s - loss: 0.7961  
Epoch 11/100  
150/150 - 0s - loss: 0.7736  
Epoch 12/100  
150/150 - 0s - loss: 0.7527  
Epoch 13/100  
150/150 - 0s - loss: 0.7327  
Epoch 14/100  
150/150 - 0s - loss: 0.7109  
Epoch 15/100  
150/150 - 0s - loss: 0.6913  
Epoch 16/100  
150/150 - 0s - loss: 0.6720  
Epoch 17/100  
150/150 - 0s - loss: 0.6527  
Epoch 18/100  
150/150 - 0s - loss: 0.6311  
Epoch 19/100  
150/150 - 0s - loss: 0.6117  
Epoch 20/100  
150/150 - 0s - loss: 0.5933  
Epoch 21/100  
150/150 - 0s - loss: 0.5761  
Epoch 22/100  
150/150 - 0s - loss: 0.5568  
Epoch 23/100  
150/150 - 0s - loss: 0.5384  
Epoch 24/100  
150/150 - 0s - loss: 0.5222  
Epoch 25/100  
150/150 - 0s - loss: 0.5063  
Epoch 26/100  
150/150 - 0s - loss: 0.4945  
Epoch 27/100  
150/150 - 0s - loss: 0.4753  
Epoch 28/100

150/150 - 0s - loss: 0.4608  
Epoch 29/100  
150/150 - 0s - loss: 0.4478  
Epoch 30/100  
150/150 - 0s - loss: 0.4333  
Epoch 31/100  
150/150 - 0s - loss: 0.4209  
Epoch 32/100  
150/150 - 0s - loss: 0.4114  
Epoch 33/100  
150/150 - 0s - loss: 0.3964  
Epoch 34/100  
150/150 - 0s - loss: 0.3886  
Epoch 35/100  
150/150 - 0s - loss: 0.3799  
Epoch 36/100  
150/150 - 0s - loss: 0.3674  
Epoch 37/100  
150/150 - 0s - loss: 0.3569  
Epoch 38/100  
150/150 - 0s - loss: 0.3477  
Epoch 39/100  
150/150 - 0s - loss: 0.3386  
Epoch 40/100  
150/150 - 0s - loss: 0.3297  
Epoch 41/100  
150/150 - 0s - loss: 0.3243  
Epoch 42/100  
150/150 - 0s - loss: 0.3121  
Epoch 43/100  
150/150 - 0s - loss: 0.3051  
Epoch 44/100  
150/150 - 0s - loss: 0.2995  
Epoch 45/100  
150/150 - 0s - loss: 0.2886  
Epoch 46/100  
150/150 - 0s - loss: 0.2836  
Epoch 47/100  
150/150 - 0s - loss: 0.2746  
Epoch 48/100  
150/150 - 0s - loss: 0.2683  
Epoch 49/100  
150/150 - 0s - loss: 0.2608  
Epoch 50/100  
150/150 - 0s - loss: 0.2545  
Epoch 51/100  
150/150 - 0s - loss: 0.2483  
Epoch 52/100  
150/150 - 0s - loss: 0.2426  
Epoch 53/100  
150/150 - 0s - loss: 0.2349  
Epoch 54/100  
150/150 - 0s - loss: 0.2310  
Epoch 55/100  
150/150 - 0s - loss: 0.2238  
Epoch 56/100

150/150 - 0s - loss: 0.2193  
Epoch 57/100  
150/150 - 0s - loss: 0.2144  
Epoch 58/100  
150/150 - 0s - loss: 0.2101  
Epoch 59/100  
150/150 - 0s - loss: 0.2043  
Epoch 60/100  
150/150 - 0s - loss: 0.1996  
Epoch 61/100  
150/150 - 0s - loss: 0.1954  
Epoch 62/100  
150/150 - 0s - loss: 0.1909  
Epoch 63/100  
150/150 - 0s - loss: 0.1858  
Epoch 64/100  
150/150 - 0s - loss: 0.1823  
Epoch 65/100  
150/150 - 0s - loss: 0.1789  
Epoch 66/100  
150/150 - 0s - loss: 0.1747  
Epoch 67/100  
150/150 - 0s - loss: 0.1722  
Epoch 68/100  
150/150 - 0s - loss: 0.1684  
Epoch 69/100  
150/150 - 0s - loss: 0.1633  
Epoch 70/100  
150/150 - 0s - loss: 0.1623  
Epoch 71/100  
150/150 - 0s - loss: 0.1579  
Epoch 72/100  
150/150 - 0s - loss: 0.1563  
Epoch 73/100  
150/150 - 0s - loss: 0.1551  
Epoch 74/100  
150/150 - 0s - loss: 0.1513  
Epoch 75/100  
150/150 - 0s - loss: 0.1484  
Epoch 76/100  
150/150 - 0s - loss: 0.1435  
Epoch 77/100  
150/150 - 0s - loss: 0.1425  
Epoch 78/100  
150/150 - 0s - loss: 0.1396  
Epoch 79/100  
150/150 - 0s - loss: 0.1378  
Epoch 80/100  
150/150 - 0s - loss: 0.1351  
Epoch 81/100  
150/150 - 0s - loss: 0.1357  
Epoch 82/100  
150/150 - 0s - loss: 0.1308  
Epoch 83/100  
150/150 - 0s - loss: 0.1284  
Epoch 84/100

```
150/150 - 0s - loss: 0.1278
Epoch 85/100
150/150 - 0s - loss: 0.1248
Epoch 86/100
150/150 - 0s - loss: 0.1237
Epoch 87/100
150/150 - 0s - loss: 0.1212
Epoch 88/100
150/150 - 0s - loss: 0.1214
Epoch 89/100
150/150 - 0s - loss: 0.1183
Epoch 90/100
150/150 - 0s - loss: 0.1199
Epoch 91/100
150/150 - 0s - loss: 0.1155
Epoch 92/100
150/150 - 0s - loss: 0.1159
Epoch 93/100
150/150 - 0s - loss: 0.1112
Epoch 94/100
150/150 - 0s - loss: 0.1141
Epoch 95/100
150/150 - 0s - loss: 0.1104
Epoch 96/100
150/150 - 0s - loss: 0.1089
Epoch 97/100
150/150 - 0s - loss: 0.1089
Epoch 98/100
150/150 - 0s - loss: 0.1089
Epoch 99/100
150/150 - 0s - loss: 0.1197
Epoch 100/100
150/150 - 0s - loss: 0.1076
```

```
Out[39]: <tensorflow.python.keras.callbacks.History at 0x7f3ee4332410>
```

```
In [40]: # Print out number of species found:
print(species)
```

```
Index(['Iris-setosa', 'Iris-versicolor', 'Iris-virginica'], dtype='object')
```

Now that you have a neural network trained, we would like to be able to use it. The following code makes use of our neural network. Exactly like before, we will generate predictions. Notice that three values come back for each of the 150 iris flowers. There were three types of iris (Iris-setosa, Iris-versicolor, and Iris-virginica).

```
In [41]: pred = model.predict(x)
print(f"Shape: {pred.shape}")
print(pred[0:10])
```

```
Shape: (150, 3)
[[0.99278367 0.00704507 0.00017127]
 [0.98646706 0.01317458 0.0003583 ]
 [0.98927665 0.01040124 0.00032212]
 [0.98444796 0.01508906 0.00046296]
 [0.99318063 0.00664989 0.0001695 ]
 [0.9944395 0.00544237 0.00011823]
 [0.99035525 0.00932539 0.0003193 ]
 [0.99134773 0.00843632 0.00021587]
 [0.980791 0.01855918 0.00064985]
 [0.98711026 0.01257468 0.00031499]]
```

If you would like to turn off scientific notation, the following line can be used:

```
In [42]: np.set_printoptions(suppress=True)
```

Now we see these values rounded up.

```
In [43]: print(y[0:10])
```

Usually, the program considers the column with the highest prediction to be the prediction of the neural network. It is easy to convert the predictions to the expected iris species. The `argmax` function finds the index of the maximum prediction for each row.

```
In [44]: predict_classes = np.argmax(pred, axis=1)
expected_classes = np.argmax(y, axis=1)
print(f"Predictions: {predict_classes}")
print(f"Expected: {expected_classes}")
```

Of course, it is straightforward to turn these indexes back into iris species. We use the species list that we created earlier.

```
In [45]: print(species[predict_classes[1:10]])
```

```
Index(['Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
       'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
       'Iris-setosa'],
     dtype='object')
```

Accuracy might be a more easily understood error metric. It is essentially a test score. For all of the iris predictions, what percent were correct? The downside is it does not consider how confident the neural network was in each prediction.

```
In [46]: from sklearn.metrics import accuracy_score
```

```
correct = accuracy_score(expected_classes,predict_classes)
print(f"Accuracy: {correct}")
```

```
Accuracy: 0.973333333333334
```

The code below performs two ad hoc predictions. The first prediction is a single iris flower, and the second predicts two iris flowers. Notice that the **argmax** in the second prediction requires **axis=1**? Since we have a 2D array now, we must specify which axis to take the **argmax** over. The value **axis=1** specifies we want the max column index for each row.

```
In [47]: sample_flower = np.array( [[5.0,3.0,4.0,2.0]], dtype=float)
pred = model.predict(sample_flower)
print(pred)
pred = np.argmax(pred)
print(f"Predict that {sample_flower} is: {species[pred]}")
```

```
[0.00402835 0.25205988 0.74391174]
Predict that [[5. 3. 4. 2.]] is: Iris-virginica
```

You can also predict two sample flowers.

```
In [48]: sample_flower = np.array( [[5.0,3.0,4.0,2.0],[5.2,3.5,1.5,0.8]],\
                               dtype=float)
pred = model.predict(sample_flower)
print(pred)
pred = np.argmax(pred, axis=1)
print(f"Predict that these two flowers {sample_flower} ")
print(f"are: {species[pred]}")
```

```
[0.00402835 0.25205988 0.74391174]
[0.98642194 0.01315794 0.0004201 ]
Predict that these two flowers [[5. 3. 4. 2. ]
 [5.2 3.5 1.5 0.8]]
are: Index(['Iris-virginica', 'Iris-setosa'], dtype='object')
```



# T81-558: Applications of Deep Neural Networks

## Module 3: Introduction to TensorFlow

- Instructor: [Jeff Heaton](#), McKelvey School of Engineering, [Washington University in St. Louis](#)
- For more information visit the [class website](#).

## Module 3 Material

- Part 3.1: Deep Learning and Neural Network Introduction [\[Video\]](#) [\[Notebook\]](#)
- Part 3.2: Introduction to Tensorflow and Keras [\[Video\]](#) [\[Notebook\]](#)
- **Part 3.3: Saving and Loading a Keras Neural Network** [\[Video\]](#) [\[Notebook\]](#)
- Part 3.4: Early Stopping in Keras to Prevent Overfitting [\[Video\]](#) [\[Notebook\]](#)
- Part 3.5: Extracting Weights and Manual Calculation [\[Video\]](#) [\[Notebook\]](#)

## Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow. Running the following code will map your GDrive to `/content/drive`.

```
In [1]: try:  
    from google.colab import drive  
    drive.mount('/content/drive', force_remount=True)  
    COLAB = True  
    print("Note: using Google CoLab")  
    %tensorflow_version 2.x  
except:  
    print("Note: not using Google CoLab")  
    COLAB = False
```

Mounted at /content/drive  
Note: using Google CoLab

## Part 3.3: Saving and Loading a Keras Neural Network

Complex neural networks will take a long time to fit/train. It is helpful to be able to save these neural networks so that you can reload them later. A reloaded neural network will not require retraining. Keras provides three formats for neural network saving.

- **JSON** - Stores the neural network structure (no weights) in the [JSON file format](#).
- **HDF5** - Stores the complete neural network (with weights) in the [HDF5 file format](#). Do not confuse HDF5 with [HDFS](#). They are different. We do not use HDFS in this class.

Usually, you will want to save in HDF5.

```
In [2]: from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
import pandas as pd
import io
import os
import requests
import numpy as np
from sklearn import metrics

save_path = "."

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

cars = df['name']

# Handle missing value
df['horsepower'] = df['horsepower'].fillna(df['horsepower'].median())

# Pandas to Numpy
x = df[['cylinders', 'displacement', 'horsepower', 'weight',
         'acceleration', 'year', 'origin']].values
y = df['mpg'].values # regression

# Build the neural network
model = Sequential()
model.add(Dense(25, input_dim=x.shape[1], activation='relu')) # Hidden 1
model.add(Dense(10, activation='relu')) # Hidden 2
model.add(Dense(1)) # Output
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(x,y,verbose=2,epochs=100)

# Predict
pred = model.predict(x)

# Measure RMSE error. RMSE is common for regression.
score = np.sqrt(metrics.mean_squared_error(pred,y))
print(f"Before save score (RMSE): {score}")
```

```
# save neural network structure to JSON (no weights)
model_json = model.to_json()
with open(os.path.join(save_path,"network.json"), "w") as json_file:
    json_file.write(model_json)

# save entire network to HDF5 (save everything, suggested)
model.save(os.path.join(save_path,"network.h5"))
```

Epoch 1/100  
13/13 - 1s - loss: 223035.4531 - 729ms/epoch - 56ms/step  
Epoch 2/100  
13/13 - 0s - loss: 94454.9609 - 26ms/epoch - 2ms/step  
Epoch 3/100  
13/13 - 0s - loss: 31163.9199 - 26ms/epoch - 2ms/step  
Epoch 4/100  
13/13 - 0s - loss: 6590.2344 - 24ms/epoch - 2ms/step  
Epoch 5/100  
13/13 - 0s - loss: 692.0208 - 23ms/epoch - 2ms/step  
Epoch 6/100  
13/13 - 0s - loss: 110.7149 - 23ms/epoch - 2ms/step  
Epoch 7/100  
13/13 - 0s - loss: 154.5042 - 22ms/epoch - 2ms/step  
Epoch 8/100  
13/13 - 0s - loss: 118.5529 - 29ms/epoch - 2ms/step  
Epoch 9/100  
13/13 - 0s - loss: 91.5691 - 27ms/epoch - 2ms/step  
Epoch 10/100  
13/13 - 0s - loss: 85.7397 - 24ms/epoch - 2ms/step  
Epoch 11/100  
13/13 - 0s - loss: 85.6981 - 23ms/epoch - 2ms/step  
Epoch 12/100  
13/13 - 0s - loss: 85.2837 - 24ms/epoch - 2ms/step  
Epoch 13/100  
13/13 - 0s - loss: 84.9037 - 23ms/epoch - 2ms/step  
Epoch 14/100  
13/13 - 0s - loss: 84.6506 - 30ms/epoch - 2ms/step  
Epoch 15/100  
13/13 - 0s - loss: 84.4048 - 26ms/epoch - 2ms/step  
Epoch 16/100  
13/13 - 0s - loss: 84.1072 - 24ms/epoch - 2ms/step  
Epoch 17/100  
13/13 - 0s - loss: 83.9168 - 23ms/epoch - 2ms/step  
Epoch 18/100  
13/13 - 0s - loss: 83.7391 - 24ms/epoch - 2ms/step  
Epoch 19/100  
13/13 - 0s - loss: 83.1922 - 21ms/epoch - 2ms/step  
Epoch 20/100  
13/13 - 0s - loss: 82.9178 - 27ms/epoch - 2ms/step  
Epoch 21/100  
13/13 - 0s - loss: 82.5835 - 28ms/epoch - 2ms/step  
Epoch 22/100  
13/13 - 0s - loss: 82.2728 - 24ms/epoch - 2ms/step  
Epoch 23/100  
13/13 - 0s - loss: 81.9899 - 24ms/epoch - 2ms/step  
Epoch 24/100  
13/13 - 0s - loss: 81.7262 - 23ms/epoch - 2ms/step  
Epoch 25/100  
13/13 - 0s - loss: 81.2958 - 26ms/epoch - 2ms/step  
Epoch 26/100  
13/13 - 0s - loss: 80.9488 - 30ms/epoch - 2ms/step  
Epoch 27/100  
13/13 - 0s - loss: 80.5811 - 33ms/epoch - 3ms/step  
Epoch 28/100  
13/13 - 0s - loss: 80.3213 - 25ms/epoch - 2ms/step

Epoch 29/100  
13/13 - 0s - loss: 79.8659 - 27ms/epoch - 2ms/step  
Epoch 30/100  
13/13 - 0s - loss: 79.5628 - 24ms/epoch - 2ms/step  
Epoch 31/100  
13/13 - 0s - loss: 79.2613 - 24ms/epoch - 2ms/step  
Epoch 32/100  
13/13 - 0s - loss: 78.8549 - 23ms/epoch - 2ms/step  
Epoch 33/100  
13/13 - 0s - loss: 78.3649 - 23ms/epoch - 2ms/step  
Epoch 34/100  
13/13 - 0s - loss: 78.0478 - 23ms/epoch - 2ms/step  
Epoch 35/100  
13/13 - 0s - loss: 77.6581 - 30ms/epoch - 2ms/step  
Epoch 36/100  
13/13 - 0s - loss: 77.1970 - 24ms/epoch - 2ms/step  
Epoch 37/100  
13/13 - 0s - loss: 76.8659 - 23ms/epoch - 2ms/step  
Epoch 38/100  
13/13 - 0s - loss: 76.6319 - 23ms/epoch - 2ms/step  
Epoch 39/100  
13/13 - 0s - loss: 76.0007 - 24ms/epoch - 2ms/step  
Epoch 40/100  
13/13 - 0s - loss: 75.5929 - 25ms/epoch - 2ms/step  
Epoch 41/100  
13/13 - 0s - loss: 75.2667 - 26ms/epoch - 2ms/step  
Epoch 42/100  
13/13 - 0s - loss: 75.3607 - 24ms/epoch - 2ms/step  
Epoch 43/100  
13/13 - 0s - loss: 74.5779 - 27ms/epoch - 2ms/step  
Epoch 44/100  
13/13 - 0s - loss: 73.9867 - 21ms/epoch - 2ms/step  
Epoch 45/100  
13/13 - 0s - loss: 73.7650 - 25ms/epoch - 2ms/step  
Epoch 46/100  
13/13 - 0s - loss: 73.0263 - 24ms/epoch - 2ms/step  
Epoch 47/100  
13/13 - 0s - loss: 72.7102 - 23ms/epoch - 2ms/step  
Epoch 48/100  
13/13 - 0s - loss: 72.2177 - 28ms/epoch - 2ms/step  
Epoch 49/100  
13/13 - 0s - loss: 71.8469 - 22ms/epoch - 2ms/step  
Epoch 50/100  
13/13 - 0s - loss: 71.4904 - 28ms/epoch - 2ms/step  
Epoch 51/100  
13/13 - 0s - loss: 71.1223 - 25ms/epoch - 2ms/step  
Epoch 52/100  
13/13 - 0s - loss: 70.5943 - 25ms/epoch - 2ms/step  
Epoch 53/100  
13/13 - 0s - loss: 70.1748 - 21ms/epoch - 2ms/step  
Epoch 54/100  
13/13 - 0s - loss: 69.8101 - 25ms/epoch - 2ms/step  
Epoch 55/100  
13/13 - 0s - loss: 69.3219 - 23ms/epoch - 2ms/step  
Epoch 56/100  
13/13 - 0s - loss: 68.7525 - 22ms/epoch - 2ms/step

Epoch 57/100  
13/13 - 0s - loss: 68.4256 - 22ms/epoch - 2ms/step  
Epoch 58/100  
13/13 - 0s - loss: 67.8394 - 23ms/epoch - 2ms/step  
Epoch 59/100  
13/13 - 0s - loss: 67.4138 - 22ms/epoch - 2ms/step  
Epoch 60/100  
13/13 - 0s - loss: 66.9941 - 33ms/epoch - 3ms/step  
Epoch 61/100  
13/13 - 0s - loss: 66.6573 - 29ms/epoch - 2ms/step  
Epoch 62/100  
13/13 - 0s - loss: 66.1712 - 22ms/epoch - 2ms/step  
Epoch 63/100  
13/13 - 0s - loss: 65.8375 - 29ms/epoch - 2ms/step  
Epoch 64/100  
13/13 - 0s - loss: 65.3441 - 23ms/epoch - 2ms/step  
Epoch 65/100  
13/13 - 0s - loss: 64.9143 - 22ms/epoch - 2ms/step  
Epoch 66/100  
13/13 - 0s - loss: 64.7354 - 24ms/epoch - 2ms/step  
Epoch 67/100  
13/13 - 0s - loss: 63.8731 - 30ms/epoch - 2ms/step  
Epoch 68/100  
13/13 - 0s - loss: 63.5211 - 26ms/epoch - 2ms/step  
Epoch 69/100  
13/13 - 0s - loss: 62.9679 - 22ms/epoch - 2ms/step  
Epoch 70/100  
13/13 - 0s - loss: 62.6917 - 21ms/epoch - 2ms/step  
Epoch 71/100  
13/13 - 0s - loss: 62.1212 - 22ms/epoch - 2ms/step  
Epoch 72/100  
13/13 - 0s - loss: 62.1577 - 32ms/epoch - 2ms/step  
Epoch 73/100  
13/13 - 0s - loss: 61.1758 - 22ms/epoch - 2ms/step  
Epoch 74/100  
13/13 - 0s - loss: 61.0303 - 24ms/epoch - 2ms/step  
Epoch 75/100  
13/13 - 0s - loss: 60.5673 - 23ms/epoch - 2ms/step  
Epoch 76/100  
13/13 - 0s - loss: 60.0197 - 24ms/epoch - 2ms/step  
Epoch 77/100  
13/13 - 0s - loss: 59.7046 - 24ms/epoch - 2ms/step  
Epoch 78/100  
13/13 - 0s - loss: 59.0460 - 25ms/epoch - 2ms/step  
Epoch 79/100  
13/13 - 0s - loss: 58.6879 - 27ms/epoch - 2ms/step  
Epoch 80/100  
13/13 - 0s - loss: 58.2086 - 28ms/epoch - 2ms/step  
Epoch 81/100  
13/13 - 0s - loss: 58.1870 - 40ms/epoch - 3ms/step  
Epoch 82/100  
13/13 - 0s - loss: 57.3580 - 35ms/epoch - 3ms/step  
Epoch 83/100  
13/13 - 0s - loss: 57.0140 - 24ms/epoch - 2ms/step  
Epoch 84/100  
13/13 - 0s - loss: 56.5466 - 36ms/epoch - 3ms/step

```
Epoch 85/100
13/13 - 0s - loss: 56.2083 - 30ms/epoch - 2ms/step
Epoch 86/100
13/13 - 0s - loss: 55.7131 - 24ms/epoch - 2ms/step
Epoch 87/100
13/13 - 0s - loss: 55.2924 - 28ms/epoch - 2ms/step
Epoch 88/100
13/13 - 0s - loss: 54.9157 - 26ms/epoch - 2ms/step
Epoch 89/100
13/13 - 0s - loss: 54.8022 - 27ms/epoch - 2ms/step
Epoch 90/100
13/13 - 0s - loss: 53.9416 - 25ms/epoch - 2ms/step
Epoch 91/100
13/13 - 0s - loss: 53.6013 - 30ms/epoch - 2ms/step
Epoch 92/100
13/13 - 0s - loss: 53.3547 - 25ms/epoch - 2ms/step
Epoch 93/100
13/13 - 0s - loss: 52.7261 - 39ms/epoch - 3ms/step
Epoch 94/100
13/13 - 0s - loss: 52.3562 - 25ms/epoch - 2ms/step
Epoch 95/100
13/13 - 0s - loss: 51.9567 - 23ms/epoch - 2ms/step
Epoch 96/100
13/13 - 0s - loss: 51.4552 - 29ms/epoch - 2ms/step
Epoch 97/100
13/13 - 0s - loss: 51.4597 - 23ms/epoch - 2ms/step
Epoch 98/100
13/13 - 0s - loss: 50.6219 - 24ms/epoch - 2ms/step
Epoch 99/100
13/13 - 0s - loss: 50.2118 - 25ms/epoch - 2ms/step
Epoch 100/100
13/13 - 0s - loss: 49.8828 - 25ms/epoch - 2ms/step
Before save score (RMSE): 7.044431690300903
```

The code below sets up a neural network and reads the data (for predictions), but it does not clear the model directory or fit the neural network. The code loads the weights from the previous fit. Now we reload the network and perform another prediction. The RMSE should match the previous one exactly if we saved and reloaded the neural network correctly.

```
In [3]: from tensorflow.keras.models import load_model
model2 = load_model(os.path.join(save_path,"network.h5"))
pred = model2.predict(x)
# Measure RMSE error. RMSE is common for regression.
score = np.sqrt(metrics.mean_squared_error(pred,y))
print(f"After load score (RMSE): {score}")
```

```
After load score (RMSE): 7.044431690300903
```

# T81-558: Applications of Deep Neural Networks

## Module 3: Introduction to TensorFlow

- Instructor: [Jeff Heaton](#), McKelvey School of Engineering, [Washington University in St. Louis](#)
- For more information visit the [class website](#).

## Module 3 Material

- Part 3.1: Deep Learning and Neural Network Introduction [\[Video\]](#) [\[Notebook\]](#)
- Part 3.2: Introduction to Tensorflow and Keras [\[Video\]](#) [\[Notebook\]](#)
- Part 3.3: Saving and Loading a Keras Neural Network [\[Video\]](#) [\[Notebook\]](#)
- **Part 3.4: Early Stopping in Keras to Prevent Overfitting** [\[Video\]](#) [\[Notebook\]](#)
- Part 3.5: Extracting Weights and Manual Calculation [\[Video\]](#) [\[Notebook\]](#)

## Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

```
In [1]: try:  
    %tensorflow_version 2.x  
    COLAB = True  
    print("Note: using Google CoLab")  
except:  
    print("Note: not using Google CoLab")  
    COLAB = False
```

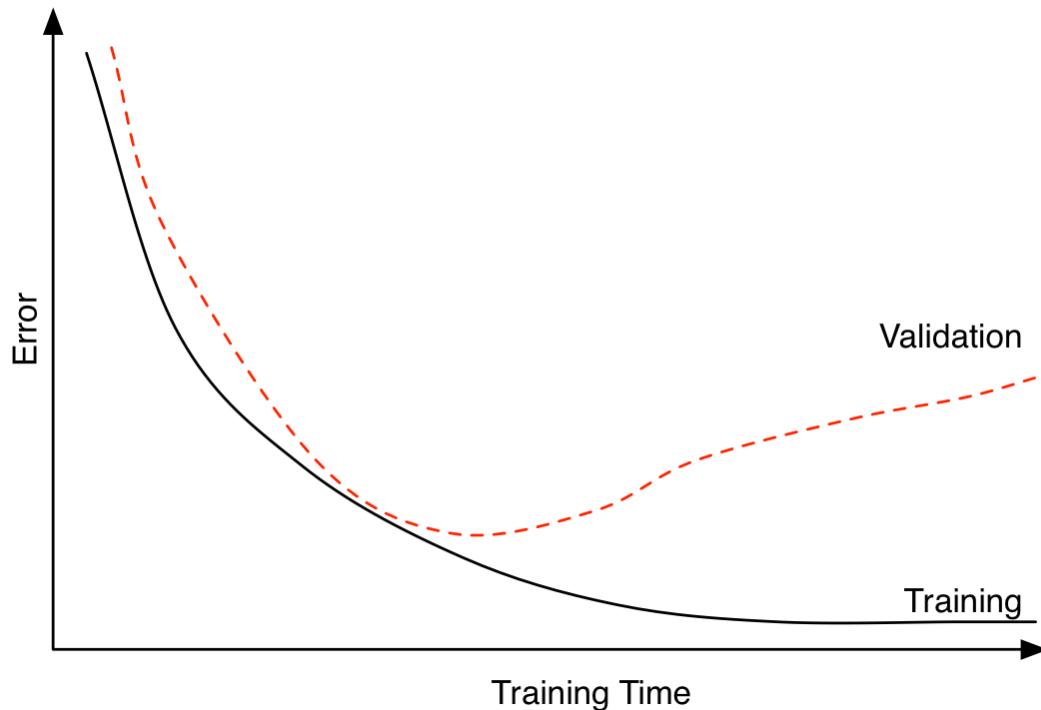
Note: not using Google CoLab

## Part 3.4: Early Stopping in Keras to Prevent Overfitting

It can be difficult to determine how many epochs to cycle through to train a neural network. Overfitting will occur if you train the neural network for too

many epochs, and the neural network will not perform well on new data, despite attaining a good accuracy on the training set. Overfitting occurs when a neural network is trained to the point that it begins to memorize rather than generalize, as demonstrated in Figure 3.OVER.

**Figure 3.OVER: Training vs. Validation Error for Overfitting**



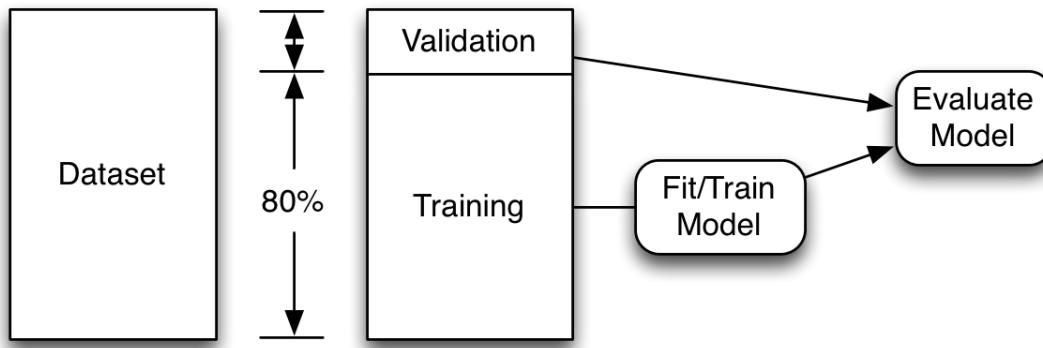
It is important to segment the original dataset into several datasets:

- **Training Set**
- **Validation Set**
- **Holdout Set**

You can construct these sets in several different ways. The following programs demonstrate some of these.

The first method is a training and validation set. We use the training data to train the neural network until the validation set no longer improves. This attempts to stop at a near-optimal training point. This method will only give accurate "out of sample" predictions for the validation set; this is usually 20% of the data. The predictions for the training data will be overly optimistic, as these were the data that we used to train the neural network. Figure 3.VAL demonstrates how we divide the dataset.

**Figure 3.VAL: Training with a Validation Set**



## Early Stopping with Classification

We will now see an example of classification training with early stopping. We will train the neural network until the error no longer improves on the validation set.

```
In [2]: import pandas as pd
import io
import requests
import numpy as np
from sklearn import metrics
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/iris.csv",
    na_values=['NA', '?'])

# Convert to numpy - Classification
x = df[['sepal_l', 'sepal_w', 'petal_l', 'petal_w']].values
dummies = pd.get_dummies(df['species']) # Classification
species = dummies.columns
y = dummies.values

# Split into validation and training sets
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)

# Build neural network
model = Sequential()
model.add(Dense(50, input_dim=x.shape[1], activation='relu')) # Hidden 1
model.add(Dense(25, activation='relu')) # Hidden 2
model.add(Dense(y.shape[1],activation='softmax')) # Output
model.compile(loss='categorical_crossentropy', optimizer='adam')

monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3, patience=5,
                        verbose=1, mode='auto', restore_best_weights=True)
```

```
model.fit(x_train,y_train,validation_data=(x_test,y_test),  
          callbacks=[monitor],verbose=2,epochs=1000)
```

Train on 112 samples, validate on 38 samples  
Epoch 1/1000  
112/112 - 0s - loss: 1.1940 - val\_loss: 1.1126  
Epoch 2/1000  
112/112 - 0s - loss: 1.0545 - val\_loss: 0.9984  
Epoch 3/1000  
112/112 - 0s - loss: 0.9533 - val\_loss: 0.9130  
Epoch 4/1000  
112/112 - 0s - loss: 0.8823 - val\_loss: 0.8365  
Epoch 5/1000  
112/112 - 0s - loss: 0.8243 - val\_loss: 0.7619  
Epoch 6/1000  
112/112 - 0s - loss: 0.7592 - val\_loss: 0.7059  
Epoch 7/1000  
112/112 - 0s - loss: 0.7142 - val\_loss: 0.6644  
Epoch 8/1000  
112/112 - 0s - loss: 0.6788 - val\_loss: 0.6302  
Epoch 9/1000  
112/112 - 0s - loss: 0.6481 - val\_loss: 0.5979  
Epoch 10/1000  
112/112 - 0s - loss: 0.6198 - val\_loss: 0.5698  
Epoch 11/1000  
112/112 - 0s - loss: 0.5957 - val\_loss: 0.5434  
Epoch 12/1000  
112/112 - 0s - loss: 0.5738 - val\_loss: 0.5189  
Epoch 13/1000  
112/112 - 0s - loss: 0.5539 - val\_loss: 0.4964  
Epoch 14/1000  
112/112 - 0s - loss: 0.5344 - val\_loss: 0.4771  
Epoch 15/1000  
112/112 - 0s - loss: 0.5177 - val\_loss: 0.4601  
Epoch 16/1000  
112/112 - 0s - loss: 0.5022 - val\_loss: 0.4455  
Epoch 17/1000  
112/112 - 0s - loss: 0.4869 - val\_loss: 0.4334  
Epoch 18/1000  
112/112 - 0s - loss: 0.4786 - val\_loss: 0.4236  
Epoch 19/1000  
112/112 - 0s - loss: 0.4634 - val\_loss: 0.4096  
Epoch 20/1000  
112/112 - 0s - loss: 0.4521 - val\_loss: 0.3980  
Epoch 21/1000  
112/112 - 0s - loss: 0.4409 - val\_loss: 0.3872  
Epoch 22/1000  
112/112 - 0s - loss: 0.4296 - val\_loss: 0.3776  
Epoch 23/1000  
112/112 - 0s - loss: 0.4204 - val\_loss: 0.3688  
Epoch 24/1000  
112/112 - 0s - loss: 0.4113 - val\_loss: 0.3598  
Epoch 25/1000  
112/112 - 0s - loss: 0.4025 - val\_loss: 0.3519  
Epoch 26/1000  
112/112 - 0s - loss: 0.3970 - val\_loss: 0.3478  
Epoch 27/1000  
112/112 - 0s - loss: 0.3860 - val\_loss: 0.3382  
Epoch 28/1000

112/112 - 0s - loss: 0.3763 - val\_loss: 0.3297  
Epoch 29/1000  
112/112 - 0s - loss: 0.3678 - val\_loss: 0.3213  
Epoch 30/1000  
112/112 - 0s - loss: 0.3600 - val\_loss: 0.3137  
Epoch 31/1000  
112/112 - 0s - loss: 0.3535 - val\_loss: 0.3062  
Epoch 32/1000  
112/112 - 0s - loss: 0.3451 - val\_loss: 0.2995  
Epoch 33/1000  
112/112 - 0s - loss: 0.3380 - val\_loss: 0.2940  
Epoch 34/1000  
112/112 - 0s - loss: 0.3301 - val\_loss: 0.2860  
Epoch 35/1000  
112/112 - 0s - loss: 0.3228 - val\_loss: 0.2791  
Epoch 36/1000  
112/112 - 0s - loss: 0.3152 - val\_loss: 0.2726  
Epoch 37/1000  
112/112 - 0s - loss: 0.3084 - val\_loss: 0.2668  
Epoch 38/1000  
112/112 - 0s - loss: 0.3009 - val\_loss: 0.2608  
Epoch 39/1000  
112/112 - 0s - loss: 0.2945 - val\_loss: 0.2558  
Epoch 40/1000  
112/112 - 0s - loss: 0.2874 - val\_loss: 0.2516  
Epoch 41/1000  
112/112 - 0s - loss: 0.2818 - val\_loss: 0.2437  
Epoch 42/1000  
112/112 - 0s - loss: 0.2744 - val\_loss: 0.2364  
Epoch 43/1000  
112/112 - 0s - loss: 0.2689 - val\_loss: 0.2313  
Epoch 44/1000  
112/112 - 0s - loss: 0.2612 - val\_loss: 0.2268  
Epoch 45/1000  
112/112 - 0s - loss: 0.2556 - val\_loss: 0.2219  
Epoch 46/1000  
112/112 - 0s - loss: 0.2498 - val\_loss: 0.2179  
Epoch 47/1000  
112/112 - 0s - loss: 0.2443 - val\_loss: 0.2111  
Epoch 48/1000  
112/112 - 0s - loss: 0.2381 - val\_loss: 0.2053  
Epoch 49/1000  
112/112 - 0s - loss: 0.2331 - val\_loss: 0.2008  
Epoch 50/1000  
112/112 - 0s - loss: 0.2273 - val\_loss: 0.1956  
Epoch 51/1000  
112/112 - 0s - loss: 0.2249 - val\_loss: 0.1906  
Epoch 52/1000  
112/112 - 0s - loss: 0.2172 - val\_loss: 0.1909  
Epoch 53/1000  
112/112 - 0s - loss: 0.2170 - val\_loss: 0.1943  
Epoch 54/1000  
112/112 - 0s - loss: 0.2099 - val\_loss: 0.1791  
Epoch 55/1000  
112/112 - 0s - loss: 0.2073 - val\_loss: 0.1758  
Epoch 56/1000

112/112 - 0s - loss: 0.2031 - val\_loss: 0.1712  
Epoch 57/1000  
112/112 - 0s - loss: 0.1970 - val\_loss: 0.1717  
Epoch 58/1000  
112/112 - 0s - loss: 0.1907 - val\_loss: 0.1648  
Epoch 59/1000  
112/112 - 0s - loss: 0.1862 - val\_loss: 0.1606  
Epoch 60/1000  
112/112 - 0s - loss: 0.1831 - val\_loss: 0.1572  
Epoch 61/1000  
112/112 - 0s - loss: 0.1840 - val\_loss: 0.1590  
Epoch 62/1000  
112/112 - 0s - loss: 0.1753 - val\_loss: 0.1518  
Epoch 63/1000  
112/112 - 0s - loss: 0.1721 - val\_loss: 0.1470  
Epoch 64/1000  
112/112 - 0s - loss: 0.1706 - val\_loss: 0.1443  
Epoch 65/1000  
112/112 - 0s - loss: 0.1660 - val\_loss: 0.1488  
Epoch 66/1000  
112/112 - 0s - loss: 0.1643 - val\_loss: 0.1441  
Epoch 67/1000  
112/112 - 0s - loss: 0.1598 - val\_loss: 0.1390  
Epoch 68/1000  
112/112 - 0s - loss: 0.1566 - val\_loss: 0.1334  
Epoch 69/1000  
112/112 - 0s - loss: 0.1554 - val\_loss: 0.1316  
Epoch 70/1000  
112/112 - 0s - loss: 0.1519 - val\_loss: 0.1315  
Epoch 71/1000  
112/112 - 0s - loss: 0.1483 - val\_loss: 0.1396  
Epoch 72/1000  
112/112 - 0s - loss: 0.1502 - val\_loss: 0.1327  
Epoch 73/1000  
112/112 - 0s - loss: 0.1441 - val\_loss: 0.1229  
Epoch 74/1000  
112/112 - 0s - loss: 0.1417 - val\_loss: 0.1198  
Epoch 75/1000  
112/112 - 0s - loss: 0.1411 - val\_loss: 0.1189  
Epoch 76/1000  
112/112 - 0s - loss: 0.1365 - val\_loss: 0.1207  
Epoch 77/1000  
112/112 - 0s - loss: 0.1350 - val\_loss: 0.1229  
Epoch 78/1000  
112/112 - 0s - loss: 0.1355 - val\_loss: 0.1182  
Epoch 79/1000  
112/112 - 0s - loss: 0.1320 - val\_loss: 0.1152  
Epoch 80/1000  
112/112 - 0s - loss: 0.1300 - val\_loss: 0.1092  
Epoch 81/1000  
112/112 - 0s - loss: 0.1285 - val\_loss: 0.1091  
Epoch 82/1000  
112/112 - 0s - loss: 0.1258 - val\_loss: 0.1140  
Epoch 83/1000  
112/112 - 0s - loss: 0.1308 - val\_loss: 0.1144  
Epoch 84/1000

```
112/112 - 0s - loss: 0.1259 - val_loss: 0.1027
Epoch 85/1000
112/112 - 0s - loss: 0.1237 - val_loss: 0.1022
Epoch 86/1000
112/112 - 0s - loss: 0.1202 - val_loss: 0.1022
Epoch 87/1000
112/112 - 0s - loss: 0.1180 - val_loss: 0.1049
Epoch 88/1000
112/112 - 0s - loss: 0.1174 - val_loss: 0.1028
Epoch 89/1000
112/112 - 0s - loss: 0.1153 - val_loss: 0.0974
Epoch 90/1000
112/112 - 0s - loss: 0.1167 - val_loss: 0.0946
Epoch 91/1000
112/112 - 0s - loss: 0.1149 - val_loss: 0.0966
Epoch 92/1000
112/112 - 0s - loss: 0.1157 - val_loss: 0.1050
Epoch 93/1000
112/112 - 0s - loss: 0.1122 - val_loss: 0.0930
Epoch 94/1000
112/112 - 0s - loss: 0.1136 - val_loss: 0.0905
Epoch 95/1000
112/112 - 0s - loss: 0.1086 - val_loss: 0.1000
Epoch 96/1000
112/112 - 0s - loss: 0.1118 - val_loss: 0.1087
Epoch 97/1000
112/112 - 0s - loss: 0.1095 - val_loss: 0.0923
Epoch 98/1000
112/112 - 0s - loss: 0.1096 - val_loss: 0.0864
Epoch 99/1000
112/112 - 0s - loss: 0.1138 - val_loss: 0.0856
Epoch 100/1000
112/112 - 0s - loss: 0.1096 - val_loss: 0.1144
Epoch 101/1000
112/112 - 0s - loss: 0.1197 - val_loss: 0.1026
Epoch 102/1000
112/112 - 0s - loss: 0.1064 - val_loss: 0.0827
Epoch 103/1000
112/112 - 0s - loss: 0.1069 - val_loss: 0.0823
Epoch 104/1000
112/112 - 0s - loss: 0.1022 - val_loss: 0.0863
Epoch 105/1000
112/112 - 0s - loss: 0.0992 - val_loss: 0.0933
Epoch 106/1000
112/112 - 0s - loss: 0.1017 - val_loss: 0.0926
Epoch 107/1000
Restoring model weights from the end of the best epoch.
112/112 - 0s - loss: 0.1001 - val_loss: 0.0869
Epoch 00107: early stopping
```

Out[2]: <tensorflow.python.keras.callbacks.History at 0x22a9ad34708>

There are a number of parameters that are specified to the **EarlyStopping** object.

- **min\_delta** This value should be kept small. It simply means the minimum change in error to be registered as an improvement. Setting it even smaller will not likely have a great deal of impact.
- **patience** How long should the training wait for the validation error to improve?
- **verbose** How much progress information do you want?
- **mode** In general, always set this to "auto". This allows you to specify if the error should be minimized or maximized. Consider accuracy, where higher numbers are desired vs log-loss/RMSE where lower numbers are desired.
- **restore\_best\_weights** This should always be set to true. This restores the weights to the values they were at when the validation set is the highest. Unless you are manually tracking the weights yourself (we do not use this technique in this course), you should have Keras perform this step for you.

As you can see from above, the entire number of requested epochs were not used. The neural network training stopped once the validation set no longer improved.

```
In [3]: from sklearn.metrics import accuracy_score

pred = model.predict(x_test)
predict_classes = np.argmax(pred, axis=1)
expected_classes = np.argmax(y_test, axis=1)
correct = accuracy_score(expected_classes, predict_classes)
print(f"Accuracy: {correct}")
```

Accuracy: 1.0

## Early Stopping with Regression

The following code demonstrates how we can apply early stopping to a regression problem. The technique is similar to the early stopping for classification code that we just saw.

```
In [4]: from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
import pandas as pd
import io
import os
import requests
import numpy as np
from sklearn import metrics

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

cars = df['name']
```

```
# Handle missing value
df['horsepower'] = df['horsepower'].fillna(df['horsepower'].median())

# Pandas to Numpy
x = df[['cylinders', 'displacement', 'horsepower', 'weight',
         'acceleration', 'year', 'origin']].values
y = df['mpg'].values # regression

# Split into validation and training sets
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)

# Build the neural network
model = Sequential()
model.add(Dense(25, input_dim=x.shape[1], activation='relu')) # Hidden 1
model.add(Dense(10, activation='relu')) # Hidden 2
model.add(Dense(1)) # Output
model.compile(loss='mean_squared_error', optimizer='adam')

monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3,
                        patience=5, verbose=1, mode='auto',
                        restore_best_weights=True)
model.fit(x_train,y_train,validation_data=(x_test,y_test),
          callbacks=[monitor], verbose=2,epochs=1000)
```

Train on 298 samples, validate on 100 samples  
Epoch 1/1000  
298/298 - 0s - loss: 254618.1117 - val\_loss: 104859.9187  
Epoch 2/1000  
298/298 - 0s - loss: 53735.2417 - val\_loss: 10033.3467  
Epoch 3/1000  
298/298 - 0s - loss: 3456.0443 - val\_loss: 2832.0205  
Epoch 4/1000  
298/298 - 0s - loss: 4912.1159 - val\_loss: 5504.1926  
Epoch 5/1000  
298/298 - 0s - loss: 4154.7669 - val\_loss: 2042.1780  
Epoch 6/1000  
298/298 - 0s - loss: 1411.5907 - val\_loss: 1259.3724  
Epoch 7/1000  
298/298 - 0s - loss: 1189.8836 - val\_loss: 1435.5145  
Epoch 8/1000  
298/298 - 0s - loss: 1207.4120 - val\_loss: 1259.7002  
Epoch 9/1000  
298/298 - 0s - loss: 1069.7891 - val\_loss: 1189.8975  
Epoch 10/1000  
298/298 - 0s - loss: 1068.2267 - val\_loss: 1188.1633  
Epoch 11/1000  
298/298 - 0s - loss: 1068.9461 - val\_loss: 1175.8650  
Epoch 12/1000  
298/298 - 0s - loss: 1044.6897 - val\_loss: 1185.7492  
Epoch 13/1000  
298/298 - 0s - loss: 1056.0984 - val\_loss: 1178.5605  
Epoch 14/1000  
298/298 - 0s - loss: 1041.7714 - val\_loss: 1157.2365  
Epoch 15/1000  
298/298 - 0s - loss: 1031.7727 - val\_loss: 1146.1638  
Epoch 16/1000  
298/298 - 0s - loss: 1026.6840 - val\_loss: 1140.5295  
Epoch 17/1000  
298/298 - 0s - loss: 1019.7115 - val\_loss: 1131.8495  
Epoch 18/1000  
298/298 - 0s - loss: 1010.8711 - val\_loss: 1122.4224  
Epoch 19/1000  
298/298 - 0s - loss: 1013.6087 - val\_loss: 1111.3609  
Epoch 20/1000  
298/298 - 0s - loss: 995.9503 - val\_loss: 1105.5188  
Epoch 21/1000  
298/298 - 0s - loss: 987.5903 - val\_loss: 1094.2863  
Epoch 22/1000  
298/298 - 0s - loss: 990.0723 - val\_loss: 1089.7853  
Epoch 23/1000  
298/298 - 0s - loss: 968.7077 - val\_loss: 1074.3502  
Epoch 24/1000  
298/298 - 0s - loss: 968.9280 - val\_loss: 1065.4332  
Epoch 25/1000  
298/298 - 0s - loss: 955.3398 - val\_loss: 1055.7287  
Epoch 26/1000  
298/298 - 0s - loss: 955.5287 - val\_loss: 1052.8219  
Epoch 27/1000  
298/298 - 0s - loss: 935.7177 - val\_loss: 1035.1746  
Epoch 28/1000

298/298 - 0s - loss: 938.9435 - val\_loss: 1026.7096  
Epoch 29/1000  
298/298 - 0s - loss: 921.2798 - val\_loss: 1021.8623  
Epoch 30/1000  
298/298 - 0s - loss: 918.7541 - val\_loss: 1021.8645  
Epoch 31/1000  
298/298 - 0s - loss: 903.5642 - val\_loss: 994.2775  
Epoch 32/1000  
298/298 - 0s - loss: 896.2183 - val\_loss: 984.4263  
Epoch 33/1000  
298/298 - 0s - loss: 886.1336 - val\_loss: 978.4129  
Epoch 34/1000  
298/298 - 0s - loss: 877.7422 - val\_loss: 964.1715  
Epoch 35/1000  
298/298 - 0s - loss: 871.3048 - val\_loss: 956.3459  
Epoch 36/1000  
298/298 - 0s - loss: 861.6707 - val\_loss: 948.6097  
Epoch 37/1000  
298/298 - 0s - loss: 850.0068 - val\_loss: 932.7441  
Epoch 38/1000  
298/298 - 0s - loss: 846.9615 - val\_loss: 921.6213  
Epoch 39/1000  
298/298 - 0s - loss: 830.3624 - val\_loss: 913.5166  
Epoch 40/1000  
298/298 - 0s - loss: 831.6781 - val\_loss: 907.8736  
Epoch 41/1000  
298/298 - 0s - loss: 814.4517 - val\_loss: 889.8433  
Epoch 42/1000  
298/298 - 0s - loss: 804.2001 - val\_loss: 879.6267  
Epoch 43/1000  
298/298 - 0s - loss: 793.5329 - val\_loss: 869.0650  
Epoch 44/1000  
298/298 - 0s - loss: 786.6698 - val\_loss: 857.7609  
Epoch 45/1000  
298/298 - 0s - loss: 775.7591 - val\_loss: 847.1539  
Epoch 46/1000  
298/298 - 0s - loss: 767.7103 - val\_loss: 836.7088  
Epoch 47/1000  
298/298 - 0s - loss: 756.9816 - val\_loss: 825.8035  
Epoch 48/1000  
298/298 - 0s - loss: 747.9103 - val\_loss: 819.3103  
Epoch 49/1000  
298/298 - 0s - loss: 739.1126 - val\_loss: 805.0508  
Epoch 50/1000  
298/298 - 0s - loss: 734.6592 - val\_loss: 795.2228  
Epoch 51/1000  
298/298 - 0s - loss: 724.3488 - val\_loss: 783.2872  
Epoch 52/1000  
298/298 - 0s - loss: 710.7389 - val\_loss: 779.2385  
Epoch 53/1000  
298/298 - 0s - loss: 702.9931 - val\_loss: 762.7323  
Epoch 54/1000  
298/298 - 0s - loss: 694.2653 - val\_loss: 751.5614  
Epoch 55/1000  
298/298 - 0s - loss: 682.2225 - val\_loss: 744.4663  
Epoch 56/1000

298/298 - 0s - loss: 683.8359 - val\_loss: 738.8125  
Epoch 57/1000  
298/298 - 0s - loss: 673.9678 - val\_loss: 723.7866  
Epoch 58/1000  
298/298 - 0s - loss: 655.8523 - val\_loss: 715.6897  
Epoch 59/1000  
298/298 - 0s - loss: 649.6330 - val\_loss: 704.0192  
Epoch 60/1000  
298/298 - 0s - loss: 643.9476 - val\_loss: 691.1572  
Epoch 61/1000  
298/298 - 0s - loss: 627.9205 - val\_loss: 685.6211  
Epoch 62/1000  
298/298 - 0s - loss: 630.9766 - val\_loss: 675.3809  
Epoch 63/1000  
298/298 - 0s - loss: 620.4021 - val\_loss: 664.9146  
Epoch 64/1000  
298/298 - 0s - loss: 601.4826 - val\_loss: 655.5067  
Epoch 65/1000  
298/298 - 0s - loss: 602.5151 - val\_loss: 644.4906  
Epoch 66/1000  
298/298 - 0s - loss: 584.9831 - val\_loss: 631.7765  
Epoch 67/1000  
298/298 - 0s - loss: 582.3892 - val\_loss: 620.7529  
Epoch 68/1000  
298/298 - 0s - loss: 580.3517 - val\_loss: 617.2255  
Epoch 69/1000  
298/298 - 0s - loss: 575.3606 - val\_loss: 603.6507  
Epoch 70/1000  
298/298 - 0s - loss: 551.4546 - val\_loss: 598.6873  
Epoch 71/1000  
298/298 - 0s - loss: 552.0443 - val\_loss: 583.6519  
Epoch 72/1000  
298/298 - 0s - loss: 536.8391 - val\_loss: 576.5555  
Epoch 73/1000  
298/298 - 0s - loss: 529.9672 - val\_loss: 564.6031  
Epoch 74/1000  
298/298 - 0s - loss: 522.4439 - val\_loss: 556.2015  
Epoch 75/1000  
298/298 - 0s - loss: 513.4194 - val\_loss: 548.1135  
Epoch 76/1000  
298/298 - 0s - loss: 505.7009 - val\_loss: 537.8890  
Epoch 77/1000  
298/298 - 0s - loss: 496.8726 - val\_loss: 530.3638  
Epoch 78/1000  
298/298 - 0s - loss: 488.8692 - val\_loss: 520.3936  
Epoch 79/1000  
298/298 - 0s - loss: 481.2276 - val\_loss: 512.5432  
Epoch 80/1000  
298/298 - 0s - loss: 477.8306 - val\_loss: 503.1329  
Epoch 81/1000  
298/298 - 0s - loss: 473.3998 - val\_loss: 494.9358  
Epoch 82/1000  
298/298 - 0s - loss: 465.8867 - val\_loss: 490.6273  
Epoch 83/1000  
298/298 - 0s - loss: 453.1066 - val\_loss: 479.8850  
Epoch 84/1000

298/298 - 0s - loss: 445.6094 - val\_loss: 471.7849  
Epoch 85/1000  
298/298 - 0s - loss: 444.9835 - val\_loss: 462.8412  
Epoch 86/1000  
298/298 - 0s - loss: 443.5763 - val\_loss: 456.5965  
Epoch 87/1000  
298/298 - 0s - loss: 436.6940 - val\_loss: 453.5159  
Epoch 88/1000  
298/298 - 0s - loss: 414.3947 - val\_loss: 447.0089  
Epoch 89/1000  
298/298 - 0s - loss: 416.7841 - val\_loss: 433.9080  
Epoch 90/1000  
298/298 - 0s - loss: 403.5432 - val\_loss: 423.4334  
Epoch 91/1000  
298/298 - 0s - loss: 403.1473 - val\_loss: 415.1188  
Epoch 92/1000  
298/298 - 0s - loss: 390.5989 - val\_loss: 408.5711  
Epoch 93/1000  
298/298 - 0s - loss: 385.0042 - val\_loss: 400.7886  
Epoch 94/1000  
298/298 - 0s - loss: 380.2837 - val\_loss: 394.4561  
Epoch 95/1000  
298/298 - 0s - loss: 382.1260 - val\_loss: 388.9179  
Epoch 96/1000  
298/298 - 0s - loss: 371.1698 - val\_loss: 380.5425  
Epoch 97/1000  
298/298 - 0s - loss: 359.9534 - val\_loss: 373.7457  
Epoch 98/1000  
298/298 - 0s - loss: 358.0036 - val\_loss: 366.5114  
Epoch 99/1000  
298/298 - 0s - loss: 348.6594 - val\_loss: 359.0750  
Epoch 100/1000  
298/298 - 0s - loss: 344.6860 - val\_loss: 352.5845  
Epoch 101/1000  
298/298 - 0s - loss: 338.0005 - val\_loss: 345.9701  
Epoch 102/1000  
298/298 - 0s - loss: 331.2779 - val\_loss: 340.2206  
Epoch 103/1000  
298/298 - 0s - loss: 325.3663 - val\_loss: 334.1550  
Epoch 104/1000  
298/298 - 0s - loss: 319.3072 - val\_loss: 327.7170  
Epoch 105/1000  
298/298 - 0s - loss: 313.7492 - val\_loss: 322.3784  
Epoch 106/1000  
298/298 - 0s - loss: 313.7471 - val\_loss: 315.4883  
Epoch 107/1000  
298/298 - 0s - loss: 304.8789 - val\_loss: 309.6435  
Epoch 108/1000  
298/298 - 0s - loss: 301.6150 - val\_loss: 304.9265  
Epoch 109/1000  
298/298 - 0s - loss: 300.2148 - val\_loss: 299.9399  
Epoch 110/1000  
298/298 - 0s - loss: 289.3050 - val\_loss: 292.6603  
Epoch 111/1000  
298/298 - 0s - loss: 282.8135 - val\_loss: 286.8729  
Epoch 112/1000

298/298 - 0s - loss: 283.6183 - val\_loss: 281.0534  
Epoch 113/1000  
298/298 - 0s - loss: 274.6550 - val\_loss: 275.6063  
Epoch 114/1000  
298/298 - 0s - loss: 269.9542 - val\_loss: 271.9059  
Epoch 115/1000  
298/298 - 0s - loss: 265.6656 - val\_loss: 265.1887  
Epoch 116/1000  
298/298 - 0s - loss: 262.1005 - val\_loss: 260.1739  
Epoch 117/1000  
298/298 - 0s - loss: 256.3500 - val\_loss: 255.2909  
Epoch 118/1000  
298/298 - 0s - loss: 251.3900 - val\_loss: 252.0265  
Epoch 119/1000  
298/298 - 0s - loss: 247.2246 - val\_loss: 245.5129  
Epoch 120/1000  
298/298 - 0s - loss: 241.7555 - val\_loss: 240.8349  
Epoch 121/1000  
298/298 - 0s - loss: 237.9977 - val\_loss: 236.3335  
Epoch 122/1000  
298/298 - 0s - loss: 233.5239 - val\_loss: 231.7200  
Epoch 123/1000  
298/298 - 0s - loss: 229.3251 - val\_loss: 227.2675  
Epoch 124/1000  
298/298 - 0s - loss: 225.5864 - val\_loss: 222.6441  
Epoch 125/1000  
298/298 - 0s - loss: 221.2191 - val\_loss: 218.1110  
Epoch 126/1000  
298/298 - 0s - loss: 217.8098 - val\_loss: 213.9518  
Epoch 127/1000  
298/298 - 0s - loss: 214.3937 - val\_loss: 210.5598  
Epoch 128/1000  
298/298 - 0s - loss: 210.2760 - val\_loss: 205.6227  
Epoch 129/1000  
298/298 - 0s - loss: 206.5413 - val\_loss: 202.4728  
Epoch 130/1000  
298/298 - 0s - loss: 202.3109 - val\_loss: 197.9401  
Epoch 131/1000  
298/298 - 0s - loss: 199.8272 - val\_loss: 196.1144  
Epoch 132/1000  
298/298 - 0s - loss: 197.1229 - val\_loss: 190.0905  
Epoch 133/1000  
298/298 - 0s - loss: 192.5514 - val\_loss: 186.7910  
Epoch 134/1000  
298/298 - 0s - loss: 189.2665 - val\_loss: 184.1961  
Epoch 135/1000  
298/298 - 0s - loss: 185.1848 - val\_loss: 179.9203  
Epoch 136/1000  
298/298 - 0s - loss: 186.1516 - val\_loss: 176.2954  
Epoch 137/1000  
298/298 - 0s - loss: 182.4030 - val\_loss: 173.4539  
Epoch 138/1000  
298/298 - 0s - loss: 177.4716 - val\_loss: 169.6453  
Epoch 139/1000  
298/298 - 0s - loss: 173.9908 - val\_loss: 166.0001  
Epoch 140/1000

298/298 - 0s - loss: 173.2805 - val\_loss: 162.8689  
Epoch 141/1000  
298/298 - 0s - loss: 172.0611 - val\_loss: 159.8967  
Epoch 142/1000  
298/298 - 0s - loss: 165.5859 - val\_loss: 157.3186  
Epoch 143/1000  
298/298 - 0s - loss: 162.3572 - val\_loss: 153.6901  
Epoch 144/1000  
298/298 - 0s - loss: 161.0297 - val\_loss: 151.6905  
Epoch 145/1000  
298/298 - 0s - loss: 164.1645 - val\_loss: 148.8484  
Epoch 146/1000  
298/298 - 0s - loss: 156.3238 - val\_loss: 145.0771  
Epoch 147/1000  
298/298 - 0s - loss: 152.3051 - val\_loss: 142.4923  
Epoch 148/1000  
298/298 - 0s - loss: 149.8716 - val\_loss: 140.4517  
Epoch 149/1000  
298/298 - 0s - loss: 147.7921 - val\_loss: 137.0884  
Epoch 150/1000  
298/298 - 0s - loss: 144.5433 - val\_loss: 134.4882  
Epoch 151/1000  
298/298 - 0s - loss: 144.0840 - val\_loss: 134.6734  
Epoch 152/1000  
298/298 - 0s - loss: 142.7512 - val\_loss: 129.2658  
Epoch 153/1000  
298/298 - 0s - loss: 138.6744 - val\_loss: 126.8691  
Epoch 154/1000  
298/298 - 0s - loss: 136.3120 - val\_loss: 125.7347  
Epoch 155/1000  
298/298 - 0s - loss: 134.8607 - val\_loss: 122.1199  
Epoch 156/1000  
298/298 - 0s - loss: 132.3261 - val\_loss: 120.8815  
Epoch 157/1000  
298/298 - 0s - loss: 130.2538 - val\_loss: 117.9441  
Epoch 158/1000  
298/298 - 0s - loss: 127.5774 - val\_loss: 116.9117  
Epoch 159/1000  
298/298 - 0s - loss: 128.5830 - val\_loss: 114.8769  
Epoch 160/1000  
298/298 - 0s - loss: 123.8368 - val\_loss: 112.2658  
Epoch 161/1000  
298/298 - 0s - loss: 121.8774 - val\_loss: 110.3176  
Epoch 162/1000  
298/298 - 0s - loss: 121.1990 - val\_loss: 108.8108  
Epoch 163/1000  
298/298 - 0s - loss: 119.1470 - val\_loss: 106.4554  
Epoch 164/1000  
298/298 - 0s - loss: 117.1019 - val\_loss: 104.7673  
Epoch 165/1000  
298/298 - 0s - loss: 114.4462 - val\_loss: 102.9108  
Epoch 166/1000  
298/298 - 0s - loss: 113.8899 - val\_loss: 100.6241  
Epoch 167/1000  
298/298 - 0s - loss: 113.7473 - val\_loss: 99.1480  
Epoch 168/1000

298/298 - 0s - loss: 109.9129 - val\_loss: 98.5171  
Epoch 169/1000  
298/298 - 0s - loss: 111.6148 - val\_loss: 95.8686  
Epoch 170/1000  
298/298 - 0s - loss: 109.5533 - val\_loss: 97.8955  
Epoch 171/1000  
298/298 - 0s - loss: 110.5111 - val\_loss: 92.7941  
Epoch 172/1000  
298/298 - 0s - loss: 110.6292 - val\_loss: 96.9406  
Epoch 173/1000  
298/298 - 0s - loss: 108.2353 - val\_loss: 90.7488  
Epoch 174/1000  
298/298 - 0s - loss: 103.7881 - val\_loss: 88.2208  
Epoch 175/1000  
298/298 - 0s - loss: 100.4373 - val\_loss: 89.0537  
Epoch 176/1000  
298/298 - 0s - loss: 100.0941 - val\_loss: 85.4782  
Epoch 177/1000  
298/298 - 0s - loss: 97.8368 - val\_loss: 85.8181  
Epoch 178/1000  
298/298 - 0s - loss: 95.8849 - val\_loss: 83.0792  
Epoch 179/1000  
298/298 - 0s - loss: 94.7138 - val\_loss: 84.0111  
Epoch 180/1000  
298/298 - 0s - loss: 93.9980 - val\_loss: 80.8398  
Epoch 181/1000  
298/298 - 0s - loss: 92.4562 - val\_loss: 81.9521  
Epoch 182/1000  
298/298 - 0s - loss: 91.9720 - val\_loss: 81.2425  
Epoch 183/1000  
298/298 - 0s - loss: 93.9076 - val\_loss: 77.1700  
Epoch 184/1000  
298/298 - 0s - loss: 92.0447 - val\_loss: 76.0691  
Epoch 185/1000  
298/298 - 0s - loss: 92.4003 - val\_loss: 77.9899  
Epoch 186/1000  
298/298 - 0s - loss: 87.6844 - val\_loss: 73.9357  
Epoch 187/1000  
298/298 - 0s - loss: 86.4119 - val\_loss: 74.5456  
Epoch 188/1000  
298/298 - 0s - loss: 85.1260 - val\_loss: 73.0177  
Epoch 189/1000  
298/298 - 0s - loss: 85.2527 - val\_loss: 71.2634  
Epoch 190/1000  
298/298 - 0s - loss: 84.7504 - val\_loss: 73.4859  
Epoch 191/1000  
298/298 - 0s - loss: 83.9971 - val\_loss: 70.3122  
Epoch 192/1000  
298/298 - 0s - loss: 82.2615 - val\_loss: 68.4355  
Epoch 193/1000  
298/298 - 0s - loss: 86.2356 - val\_loss: 76.1497  
Epoch 194/1000  
298/298 - 0s - loss: 82.0077 - val\_loss: 70.1432  
Epoch 195/1000  
298/298 - 0s - loss: 84.0382 - val\_loss: 74.0556  
Epoch 196/1000

298/298 - 0s - loss: 79.0808 - val\_loss: 65.3704  
Epoch 197/1000  
298/298 - 0s - loss: 77.5371 - val\_loss: 65.6799  
Epoch 198/1000  
298/298 - 0s - loss: 76.7543 - val\_loss: 63.9797  
Epoch 199/1000  
298/298 - 0s - loss: 75.4548 - val\_loss: 65.2337  
Epoch 200/1000  
298/298 - 0s - loss: 75.2814 - val\_loss: 62.7816  
Epoch 201/1000  
298/298 - 0s - loss: 78.7884 - val\_loss: 66.5500  
Epoch 202/1000  
298/298 - 0s - loss: 74.7617 - val\_loss: 62.7047  
Epoch 203/1000  
298/298 - 0s - loss: 73.3059 - val\_loss: 63.5815  
Epoch 204/1000  
298/298 - 0s - loss: 73.3379 - val\_loss: 60.1637  
Epoch 205/1000  
298/298 - 0s - loss: 72.8527 - val\_loss: 59.5174  
Epoch 206/1000  
298/298 - 0s - loss: 71.3816 - val\_loss: 58.9280  
Epoch 207/1000  
298/298 - 0s - loss: 71.0684 - val\_loss: 58.5003  
Epoch 208/1000  
298/298 - 0s - loss: 69.5999 - val\_loss: 58.6842  
Epoch 209/1000  
298/298 - 0s - loss: 69.6249 - val\_loss: 63.3405  
Epoch 210/1000  
298/298 - 0s - loss: 70.2080 - val\_loss: 56.3041  
Epoch 211/1000  
298/298 - 0s - loss: 68.7671 - val\_loss: 56.1137  
Epoch 212/1000  
298/298 - 0s - loss: 67.4164 - val\_loss: 56.0807  
Epoch 213/1000  
298/298 - 0s - loss: 67.4641 - val\_loss: 60.8322  
Epoch 214/1000  
298/298 - 0s - loss: 70.2280 - val\_loss: 55.4504  
Epoch 215/1000  
298/298 - 0s - loss: 70.7004 - val\_loss: 56.4594  
Epoch 216/1000  
298/298 - 0s - loss: 69.3142 - val\_loss: 66.7034  
Epoch 217/1000  
298/298 - 0s - loss: 70.9057 - val\_loss: 52.7473  
Epoch 218/1000  
298/298 - 0s - loss: 63.8462 - val\_loss: 53.7675  
Epoch 219/1000  
298/298 - 0s - loss: 65.2959 - val\_loss: 56.9050  
Epoch 220/1000  
298/298 - 0s - loss: 63.8828 - val\_loss: 52.9221  
Epoch 221/1000  
298/298 - 0s - loss: 66.2621 - val\_loss: 61.4800  
Epoch 222/1000  
298/298 - 0s - loss: 66.0702 - val\_loss: 51.9835  
Epoch 223/1000  
298/298 - 0s - loss: 62.1414 - val\_loss: 50.4767  
Epoch 224/1000

298/298 - 0s - loss: 60.9776 - val\_loss: 51.0747  
Epoch 225/1000  
298/298 - 0s - loss: 61.1262 - val\_loss: 49.3356  
Epoch 226/1000  
298/298 - 0s - loss: 59.9358 - val\_loss: 56.2200  
Epoch 227/1000  
298/298 - 0s - loss: 61.8749 - val\_loss: 48.5184  
Epoch 228/1000  
298/298 - 0s - loss: 59.3500 - val\_loss: 49.2315  
Epoch 229/1000  
298/298 - 0s - loss: 58.7732 - val\_loss: 49.6212  
Epoch 230/1000  
298/298 - 0s - loss: 59.0191 - val\_loss: 47.4893  
Epoch 231/1000  
298/298 - 0s - loss: 58.5962 - val\_loss: 52.0647  
Epoch 232/1000  
298/298 - 0s - loss: 57.5451 - val\_loss: 47.0744  
Epoch 233/1000  
298/298 - 0s - loss: 57.4292 - val\_loss: 48.5805  
Epoch 234/1000  
298/298 - 0s - loss: 57.2974 - val\_loss: 46.4830  
Epoch 235/1000  
298/298 - 0s - loss: 59.5053 - val\_loss: 48.0127  
Epoch 236/1000  
298/298 - 0s - loss: 57.6045 - val\_loss: 48.8987  
Epoch 237/1000  
298/298 - 0s - loss: 55.6797 - val\_loss: 45.2071  
Epoch 238/1000  
298/298 - 0s - loss: 54.9872 - val\_loss: 46.9131  
Epoch 239/1000  
298/298 - 0s - loss: 55.2195 - val\_loss: 44.9971  
Epoch 240/1000  
298/298 - 0s - loss: 54.4574 - val\_loss: 47.3636  
Epoch 241/1000  
298/298 - 0s - loss: 55.7777 - val\_loss: 43.9272  
Epoch 242/1000  
298/298 - 0s - loss: 56.6080 - val\_loss: 43.6550  
Epoch 243/1000  
298/298 - 0s - loss: 53.3914 - val\_loss: 44.0960  
Epoch 244/1000  
298/298 - 0s - loss: 53.3937 - val\_loss: 46.4250  
Epoch 245/1000  
298/298 - 0s - loss: 52.5582 - val\_loss: 43.4441  
Epoch 246/1000  
298/298 - 0s - loss: 52.2242 - val\_loss: 42.5886  
Epoch 247/1000  
298/298 - 0s - loss: 53.1087 - val\_loss: 45.4969  
Epoch 248/1000  
298/298 - 0s - loss: 51.2835 - val\_loss: 42.2982  
Epoch 249/1000  
298/298 - 0s - loss: 51.9679 - val\_loss: 42.0797  
Epoch 250/1000  
298/298 - 0s - loss: 50.6096 - val\_loss: 41.9481  
Epoch 251/1000  
298/298 - 0s - loss: 52.3675 - val\_loss: 42.1443  
Epoch 252/1000

298/298 - 0s - loss: 52.0081 - val\_loss: 41.5254  
Epoch 253/1000  
298/298 - 0s - loss: 52.4647 - val\_loss: 46.1836  
Epoch 254/1000  
298/298 - 0s - loss: 49.0224 - val\_loss: 40.2575  
Epoch 255/1000  
298/298 - 0s - loss: 50.8724 - val\_loss: 40.5554  
Epoch 256/1000  
298/298 - 0s - loss: 48.6178 - val\_loss: 40.2881  
Epoch 257/1000  
298/298 - 0s - loss: 48.1621 - val\_loss: 40.1415  
Epoch 258/1000  
298/298 - 0s - loss: 47.9184 - val\_loss: 39.6353  
Epoch 259/1000  
298/298 - 0s - loss: 47.7817 - val\_loss: 44.1131  
Epoch 260/1000  
298/298 - 0s - loss: 48.0547 - val\_loss: 38.6934  
Epoch 261/1000  
298/298 - 0s - loss: 49.1476 - val\_loss: 38.5595  
Epoch 262/1000  
298/298 - 0s - loss: 48.3410 - val\_loss: 38.4703  
Epoch 263/1000  
298/298 - 0s - loss: 47.1575 - val\_loss: 43.8495  
Epoch 264/1000  
298/298 - 0s - loss: 47.5766 - val\_loss: 37.7489  
Epoch 265/1000  
298/298 - 0s - loss: 45.9611 - val\_loss: 37.8400  
Epoch 266/1000  
298/298 - 0s - loss: 45.3411 - val\_loss: 37.4187  
Epoch 267/1000  
298/298 - 0s - loss: 44.8844 - val\_loss: 40.0926  
Epoch 268/1000  
298/298 - 0s - loss: 45.0760 - val\_loss: 36.9468  
Epoch 269/1000  
298/298 - 0s - loss: 45.1810 - val\_loss: 40.3046  
Epoch 270/1000  
298/298 - 0s - loss: 44.8097 - val\_loss: 37.5340  
Epoch 271/1000  
298/298 - 0s - loss: 44.2911 - val\_loss: 38.6985  
Epoch 272/1000  
298/298 - 0s - loss: 43.8413 - val\_loss: 37.3905  
Epoch 273/1000  
298/298 - 0s - loss: 43.3722 - val\_loss: 36.7338  
Epoch 274/1000  
298/298 - 0s - loss: 43.0023 - val\_loss: 35.9522  
Epoch 275/1000  
298/298 - 0s - loss: 43.3070 - val\_loss: 42.2387  
Epoch 276/1000  
298/298 - 0s - loss: 43.3620 - val\_loss: 35.6415  
Epoch 277/1000  
298/298 - 0s - loss: 44.2254 - val\_loss: 35.0081  
Epoch 278/1000  
298/298 - 0s - loss: 43.6141 - val\_loss: 35.4647  
Epoch 279/1000  
298/298 - 0s - loss: 42.5499 - val\_loss: 37.3217  
Epoch 280/1000

298/298 - 0s - loss: 42.4206 - val\_loss: 36.6365  
Epoch 281/1000  
298/298 - 0s - loss: 41.8326 - val\_loss: 33.9366  
Epoch 282/1000  
298/298 - 0s - loss: 40.7090 - val\_loss: 35.2874  
Epoch 283/1000  
298/298 - 0s - loss: 41.1847 - val\_loss: 35.7322  
Epoch 284/1000  
298/298 - 0s - loss: 40.2632 - val\_loss: 33.2830  
Epoch 285/1000  
298/298 - 0s - loss: 40.4647 - val\_loss: 33.4544  
Epoch 286/1000  
298/298 - 0s - loss: 41.8345 - val\_loss: 33.3342  
Epoch 287/1000  
298/298 - 0s - loss: 40.1833 - val\_loss: 33.5219  
Epoch 288/1000  
298/298 - 0s - loss: 42.5633 - val\_loss: 45.5246  
Epoch 289/1000  
298/298 - 0s - loss: 43.4740 - val\_loss: 32.2915  
Epoch 290/1000  
298/298 - 0s - loss: 40.7724 - val\_loss: 35.0065  
Epoch 291/1000  
298/298 - 0s - loss: 40.1270 - val\_loss: 41.1526  
Epoch 292/1000  
298/298 - 0s - loss: 41.5003 - val\_loss: 35.0315  
Epoch 293/1000  
298/298 - 0s - loss: 39.4004 - val\_loss: 33.8747  
Epoch 294/1000  
298/298 - 0s - loss: 41.5784 - val\_loss: 31.3118  
Epoch 295/1000  
298/298 - 0s - loss: 38.1686 - val\_loss: 31.1514  
Epoch 296/1000  
298/298 - 0s - loss: 38.6330 - val\_loss: 37.5739  
Epoch 297/1000  
298/298 - 0s - loss: 38.8436 - val\_loss: 30.6906  
Epoch 298/1000  
298/298 - 0s - loss: 37.6227 - val\_loss: 32.6170  
Epoch 299/1000  
298/298 - 0s - loss: 36.6737 - val\_loss: 30.3784  
Epoch 300/1000  
298/298 - 0s - loss: 36.7113 - val\_loss: 30.9689  
Epoch 301/1000  
298/298 - 0s - loss: 36.3901 - val\_loss: 31.8580  
Epoch 302/1000  
298/298 - 0s - loss: 36.4300 - val\_loss: 29.8985  
Epoch 303/1000  
298/298 - 0s - loss: 36.6609 - val\_loss: 31.8773  
Epoch 304/1000  
298/298 - 0s - loss: 39.6073 - val\_loss: 29.4928  
Epoch 305/1000  
298/298 - 0s - loss: 37.2211 - val\_loss: 29.9193  
Epoch 306/1000  
298/298 - 0s - loss: 38.2181 - val\_loss: 42.4494  
Epoch 307/1000  
298/298 - 0s - loss: 41.9627 - val\_loss: 36.3420  
Epoch 308/1000

```
298/298 - 0s - loss: 35.2754 - val_loss: 29.0452
Epoch 309/1000
298/298 - 0s - loss: 34.5570 - val_loss: 28.5060
Epoch 310/1000
298/298 - 0s - loss: 35.0860 - val_loss: 28.4189
Epoch 311/1000
298/298 - 0s - loss: 34.4839 - val_loss: 29.8177
Epoch 312/1000
298/298 - 0s - loss: 37.1565 - val_loss: 27.9970
Epoch 313/1000
298/298 - 0s - loss: 38.1949 - val_loss: 34.7456
Epoch 314/1000
298/298 - 0s - loss: 35.7598 - val_loss: 29.5360
Epoch 315/1000
298/298 - 0s - loss: 34.7382 - val_loss: 30.6052
Epoch 316/1000
298/298 - 0s - loss: 34.0591 - val_loss: 29.3044
Epoch 317/1000
Restoring model weights from the end of the best epoch.
298/298 - 0s - loss: 32.9764 - val_loss: 29.1071
Epoch 00317: early stopping
```

Out[4]: <tensorflow.python.keras.callbacks.History at 0x22a9acc8608>

Finally, we evaluate the error.

```
In [5]: # Measure RMSE error. RMSE is common for regression.
pred = model.predict(x_test)
score = np.sqrt(metrics.mean_squared_error(pred,y_test))
print(f"Final score (RMSE): {score}")
```

Final score (RMSE): 5.291219300799398

In [ ]:

# T81-558: Applications of Deep Neural Networks

## Module 3: Introduction to TensorFlow

- Instructor: [Jeff Heaton](#), McKelvey School of Engineering, [Washington University in St. Louis](#)
- For more information visit the [class website](#).

## Module 3 Material

- Part 3.1: Deep Learning and Neural Network Introduction [\[Video\]](#) [\[Notebook\]](#)
- Part 3.2: Introduction to Tensorflow and Keras [\[Video\]](#) [\[Notebook\]](#)
- Part 3.3: Saving and Loading a Keras Neural Network [\[Video\]](#) [\[Notebook\]](#)
- Part 3.4: Early Stopping in Keras to Prevent Overfitting [\[Video\]](#) [\[Notebook\]](#)
- **Part 3.5: Extracting Weights and Manual Calculation** [\[Video\]](#) [\[Notebook\]](#)

## Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

```
In [1]: try:  
    %tensorflow_version 2.x  
    COLAB = True  
    print("Note: using Google CoLab")  
except:  
    print("Note: not using Google CoLab")  
    COLAB = False
```

## Part 3.5: Extracting Weights and Manual Network Calculation

### Weight Initialization

The weights of a neural network determine the output for the neural network. The training process can adjust these weights, so the neural network produces

useful output. Most neural network training algorithms begin by initializing the weights to a random state. Training then progresses through iterations that continuously improve the weights to produce better output.

The random weights of a neural network impact how well that neural network can be trained. If a neural network fails to train, you can remedy the problem by simply restarting with a new set of random weights. However, this solution can be frustrating when you are experimenting with the architecture of a neural network and trying different combinations of hidden layers and neurons. If you add a new layer, and the network's performance improves, you must ask yourself if this improvement resulted from the new layer or from a new set of weights. Because of this uncertainty, we look for two key attributes in a weight initialization algorithm:

- How consistently does this algorithm provide good weights?
- How much of an advantage do the weights of the algorithm provide?

One of the most common yet least practical approaches to weight initialization is to set the weights to random values within a specific range. Numbers between -1 and +1 or -5 and +5 are often the choice. If you want to ensure that you get the same set of random weights each time, you should use a seed. The seed specifies a set of predefined random weights to use. For example, a seed of 1000 might produce random weights of 0.5, 0.75, and 0.2. These values are still random; you cannot predict them, yet you will always get these values when you choose a seed of 1000. Not all seeds are created equal. One problem with random weight initialization is that the random weights created by some seeds are much more difficult to train than others. The weights can be so bad that training is impossible. If you cannot train a neural network with a particular weight set, you should generate a new set of weights using a different seed.

Because weight initialization is a problem, considerable research has been around it. By default, Keras uses the Xavier weight initialization algorithm, introduced in 2006 by Glorot & Bengio [[Cite:glorot2010understanding](#)], produces good weights with reasonable consistency. This relatively simple algorithm uses normally distributed random numbers.

To use the Xavier weight initialization, it is necessary to understand that normally distributed random numbers are not the typical random numbers between 0 and 1 that most programming languages generate. Normally distributed random numbers are centered on a mean ( $\mu$ , mu) that is typically 0. If 0 is the center (mean), then you will get an equal number of random numbers above and below 0. The next question is how far these random numbers will venture from 0. In theory, you could end up with both positive and negative numbers close to the maximum positive and negative ranges supported by your computer. However,

the reality is that you will more likely see random numbers that are between 0 and three standard deviations from the center.

The standard deviation ( $\sigma$ , sigma) parameter specifies the size of this standard deviation. For example, if you specified a standard deviation of 10, you would mainly see random numbers between -30 and +30, and the numbers nearer to 0 have a much higher probability of being selected.

The above figure illustrates that the center, which in this case is 0, will be generated with a 0.4 (40%) probability. Additionally, the probability decreases very quickly beyond -2 or +2 standard deviations. By defining the center and how large the standard deviations are, you can control the range of random numbers that you will receive.

The Xavier weight initialization sets all weights to normally distributed random numbers. These weights are always centered at 0; however, their standard deviation varies depending on how many connections are present for the current layer of weights. Specifically, Equation 4.2 can determine the standard deviation:

$$Var(W) = \frac{2}{n_{in} + n_{out}}$$

The above equation shows how to obtain the variance for all weights. The square root of the variance is the standard deviation. Most random number generators accept a standard deviation rather than a variance. As a result, you usually need to take the square root of the above equation. Figure 3.XAVIER shows how this algorithm might initialize one layer.

**Figure 3.XAVIER: Xavier Weight Initialization**  Xavier Weight Initialization

We complete this process for each layer in the neural network.

## Manual Neural Network Calculation

This section will build a neural network and analyze it down the individual weights. We will train a simple neural network that learns the XOR function. It is not hard to hand-code the neurons to provide an [XOR function](#); however, we will allow Keras for simplicity to train this network for us. The neural network is small, with two inputs, two hidden neurons, and a single output. We will use 100K epochs on the ADAM optimizer. This approach is overkill, but it gets the result, and our focus here is not on tuning.

```
In [2]: from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
import numpy as np

# Create a dataset for the XOR function
```

```

x = np.array([
    [0,0],
    [1,0],
    [0,1],
    [1,1]
])

y = np.array([
    0,
    1,
    1,
    0
])

# Build the network
# sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)

done = False
cycle = 1

while not done:
    print("Cycle #{}".format(cycle))
    cycle+=1
    model = Sequential()
    model.add(Dense(2, input_dim=2, activation='relu'))
    model.add(Dense(1))
    model.compile(loss='mean_squared_error', optimizer='adam')
    model.fit(x,y,verbose=0,epochs=10000)

    # Predict
    pred = model.predict(x)

    # Check if successful. It takes several runs with this
    # small of a network
    done = pred[0]<0.01 and pred[3]<0.01 and pred[1] > 0.9 \
          and pred[2] > 0.9
    print(pred)

```

```
Cycle #1
[[0.49999997]
 [0.49999997]
 [0.49999997]
 [0.49999997]]
Cycle #2
[[0.33333334]
 [1.]
 [0.33333334]
 [0.33333334]]
Cycle #3
[[0.33333334]
 [1.]
 [0.33333334]
 [0.33333334]]
Cycle #4
[[0.]
 [1.]
 [1.]
 [0.]]
```

```
In [3]: pred[3]
```

```
Out[3]: array([0.], dtype=float32)
```

The output above should have two numbers near 0.0 for the first and fourth spots (input [0,0] and [1,1]). The middle two numbers should be near 1.0 (input [1,0] and [0,1]). These numbers are in scientific notation. Due to random starting weights, it is sometimes necessary to run the above through several cycles to get a good result.

Now that we've trained the neural network, we can dump the weights.

```
In [4]: # Dump weights
for layerNum, layer in enumerate(model.layers):
    weights = layer.get_weights()[0]
    biases = layer.get_weights()[1]

    for toNeuronNum, bias in enumerate(biases):
        print(f'{layerNum}B -> L{layerNum+1}N{toNeuronNum}: {bias}')

    for fromNeuronNum, wgt in enumerate(weights):
        for toNeuronNum, wgt2 in enumerate(wgt):
            print(f'L{layerNum}N{fromNeuronNum} \
                  -> L{layerNum+1}N{toNeuronNum} = {wgt2}')
```

```

0B -> L1N0: 1.3025760914331386e-08
0B -> L1N1: -1.4192625741316078e-08
L0N0 -> L1N0 = 0.659289538860321
L0N0 -> L1N1 = -0.9533336758613586
L0N1 -> L1N0 = -0.659289538860321
L0N1 -> L1N1 = 0.9533336758613586
1B -> L2N0: -1.9757269598130733e-08
L1N0 -> L2N0 = 1.5167843103408813
L1N1 -> L2N0 = 1.0489506721496582

```

If you rerun this, you probably get different weights. There are many ways to solve the XOR function.

In the next section, we copy/paste the weights from above and recreate the calculations done by the neural network. Because weights can change with each training, the weights used for the below code came from this:

```

0B -> L1N0: -1.2913415431976318
0B -> L1N1: -3.021530048386012e-08
L0N0 -> L1N0 = 1.2913416624069214
L0N0 -> L1N1 = 1.1912699937820435
L0N1 -> L1N0 = 1.2913411855697632
L0N1 -> L1N1 = 1.1912697553634644
1B -> L2N0: 7.626241297587034e-36
L1N0 -> L2N0 = -1.548777461051941
L1N1 -> L2N0 = 0.8394404649734497

```

```

In [5]: input0 = 0
input1 = 1

hidden0Sum = (input0*1.3)+(input1*1.3)+(-1.3)
hidden1Sum = (input0*1.2)+(input1*1.2)+(0)

print(hidden0Sum) # 0
print(hidden1Sum) # 1.2

hidden0 = max(0,hidden0Sum)
hidden1 = max(0,hidden1Sum)

print(hidden0) # 0
print(hidden1) # 1.2

outputSum = (hidden0*-1.6)+(hidden1*0.8)+(0)
print(outputSum) # 0.96

output = max(0,outputSum)

print(output) # 0.96

```

```
0.0  
1.2  
0  
1.2  
0.96  
0.96
```

In [ ]: