



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.

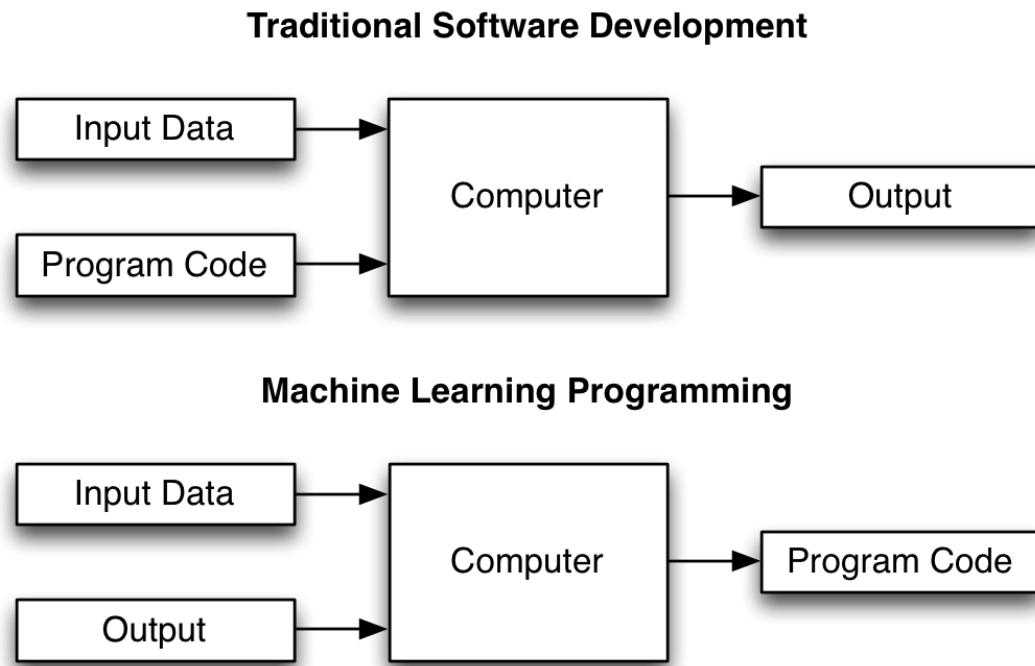
What is Deep Learning

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)
```

	name	mean	var	sdev
0	mpg	23.514573	61.089611	7.815984
1	cylinders	5.454774	2.893415	1.701004
2	displacement	193.425879	10872.199152	104.269838
3	weight	2970.424623	717140.990526	846.841774
4	acceleration	15.568090	7.604848	2.757689
5	year	76.010050	13.672443	3.697627
6	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
```

	mpg	cylinders	displacement	horsepower	weight	acceleration	year	or
0	18.0	8	307.0	130.0	3504	12.0	70	
1	15.0	8	350.0	165.0	3693	11.5	70	
...
396	28.0	4	120.0	79.0	2625	18.6	82	
397	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.

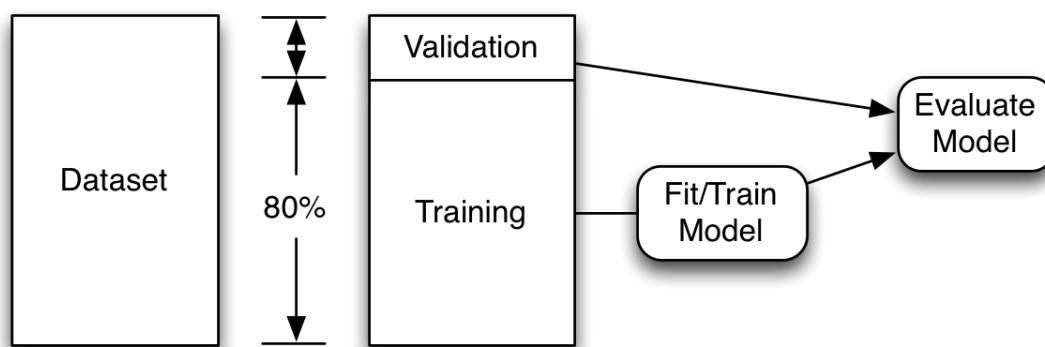
- **Training Data - In Sample Data** - The data that the neural network used to train.
- **Validation Data - Out of Sample Data** - The data that the machine learning model is evaluated upon after it is fit to the training data.

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
387	38.0	6	262.0	...	82	1	oldsmobile	cutlass ciera (diesel)
361	25.4	6	168.0	...	81	3	toyota	cressida
...
358	31.6	4	120.0	...	81	3	mazda	626
237	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(μ or \bar{x}) and the standard deviation (σ). 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
0	-0.706439	8	307.0	...	70	1	chevrolet chevelle malibu
1	-1.090751	8	350.0	...	70	1	buick skylark 320
...
396	0.574601	4	120.0	...	82	1	ford ranger
397	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)
```

	id	job	area	...	retail_dense	crime	product
0	1	vv	c	...	0.492126	0.071100	b
1	2	kd	c	...	0.342520	0.400809	c
...
1998	1999	qp	c	...	0.598425	0.117803	c
1999	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']])
```

	id	job	area	income	area_a	area_b	area_c	area_d
0	1	vv	c	50876.0	0	0	1	0
1	2	kd	c	60369.0	0	0	1	0
2	3	pe	c	55126.0	0	0	1	0
3	4	11	c	51690.0	0	0	1	0
4	5	kl	d	28347.0	0	0	0	1
...
1995	1996	vv	c	51017.0	0	0	1	0
1996	1997	kl	d	26576.0	0	0	0	1
1997	1998	kl	d	28595.0	0	0	0	1
1998	1999	qp	c	67949.0	0	0	1	0
1999	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']])
```

	id	job	income	area_a	area_b	area_c	area_d
0	1	vv	50876.0	0	0	1	0
1	2	kd	60369.0	0	0	1	0
...
1998	1999	qp	67949.0	0	0	1	0
1999	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)
```

	area_b	area_c	area_d
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']])
```

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

	cont_9	cat_0	cat_1	y	cat_0_enc	cat_1_enc
0	11.505457	dog	wolf	1	0.65	0.535714
1	60.906654	dog	wolf	0	0.65	0.535714
2	13.339096	dog	wolf	1	0.65	0.535714
3	24.058962	dog	wolf	1	0.65	0.535714
4	32.713906	dog	wolf	1	0.65	0.535714
5	85.913749	cat	wolf	1	0.35	0.535714
6	66.609021	cat	wolf	0	0.35	0.535714
7	54.116221	cat	wolf	0	0.35	0.535714
8	2.901382	cat	wolf	0	0.35	0.535714
9	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
96	13.0	8	360.0	...	73	1	amc ambassador brougham
9	15.0	8	390.0	...	70	1	amc ambassador dpl
...
325	44.3	4	90.0	...	80	2	vw rabbit c (diesel)
293	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)
```

	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

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
6	AL	35004	1	1510
7	AL	35004	2	1410
8	AL	35004	3	950
9	AL	35004	4	650
10	AL	35004	5	630
...
179785	WY	83414	2	40
179786	WY	83414	3	40
179787	WY	83414	4	0
179788	WY	83414	5	40
179789	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)
```

	STATE	zipcode	agi_stub	N1
6	AL	35004	12500	1510
7	AL	35004	37500	1410
8	AL	35004	62500	950
9	AL	35004	87500	650
10	AL	35004	112500	630
...
179785	WY	83414	37500	40
179786	WY	83414	62500	40
179787	WY	83414	87500	0
179788	WY	83414	112500	40
179789	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
...
29867	99921	48042.168675
29868	99922	32954.545455
29869	99925	45639.534884
29870	99926	41136.363636
29871	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
...
29867	99921	48042.168675
29868	99922	32954.545455
29869	99925	45639.534884
29870	99926	41136.363636
29871	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))

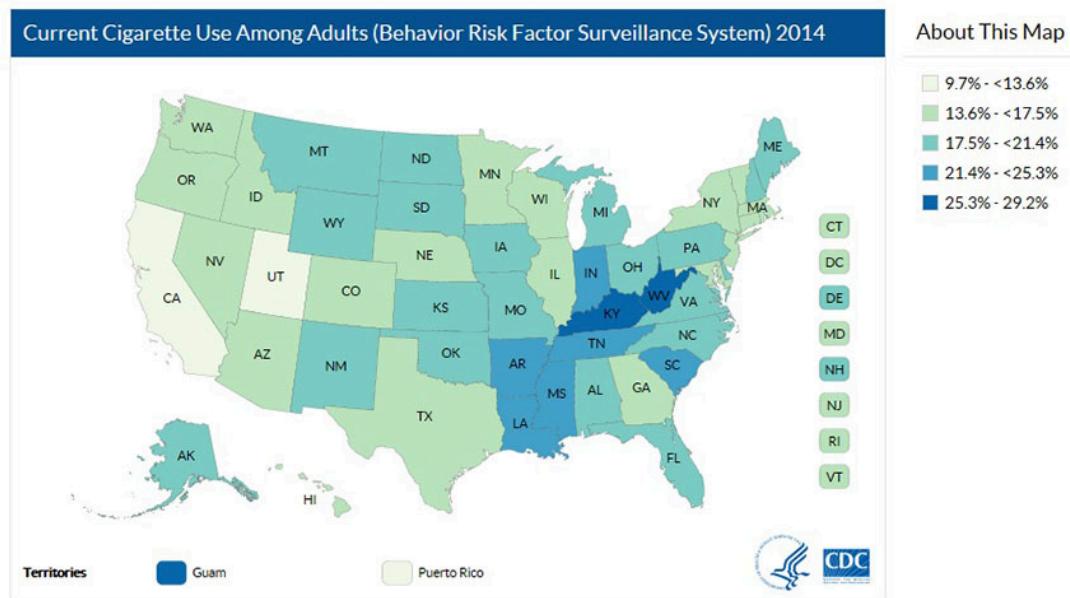
resp_json_payload = response.json()

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.

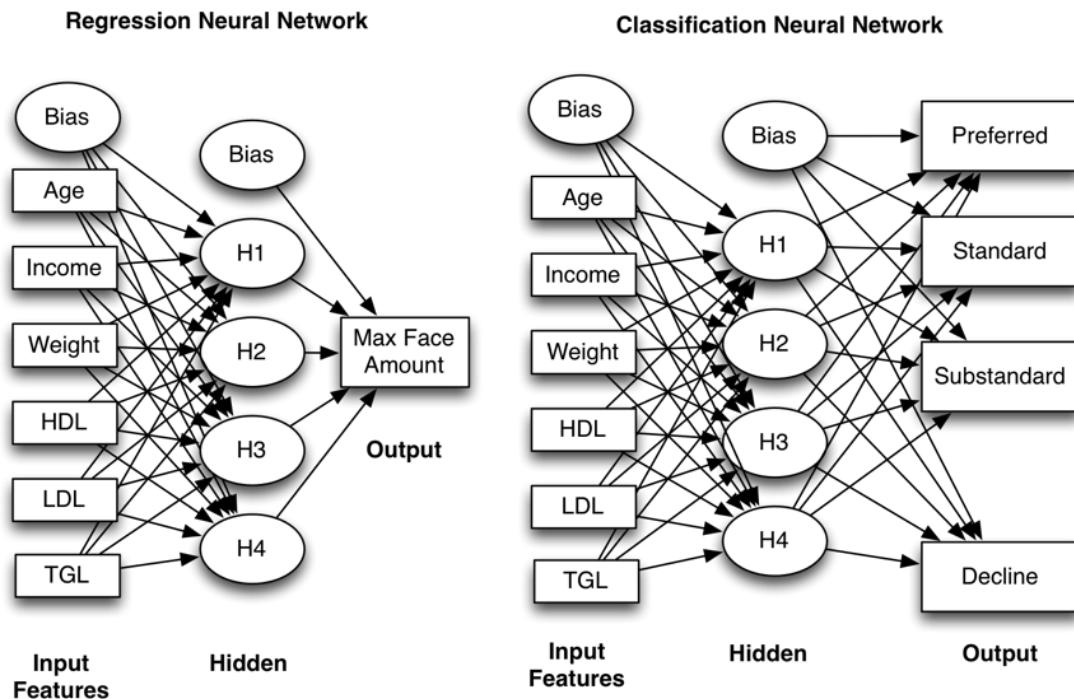
Classification or Regression

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



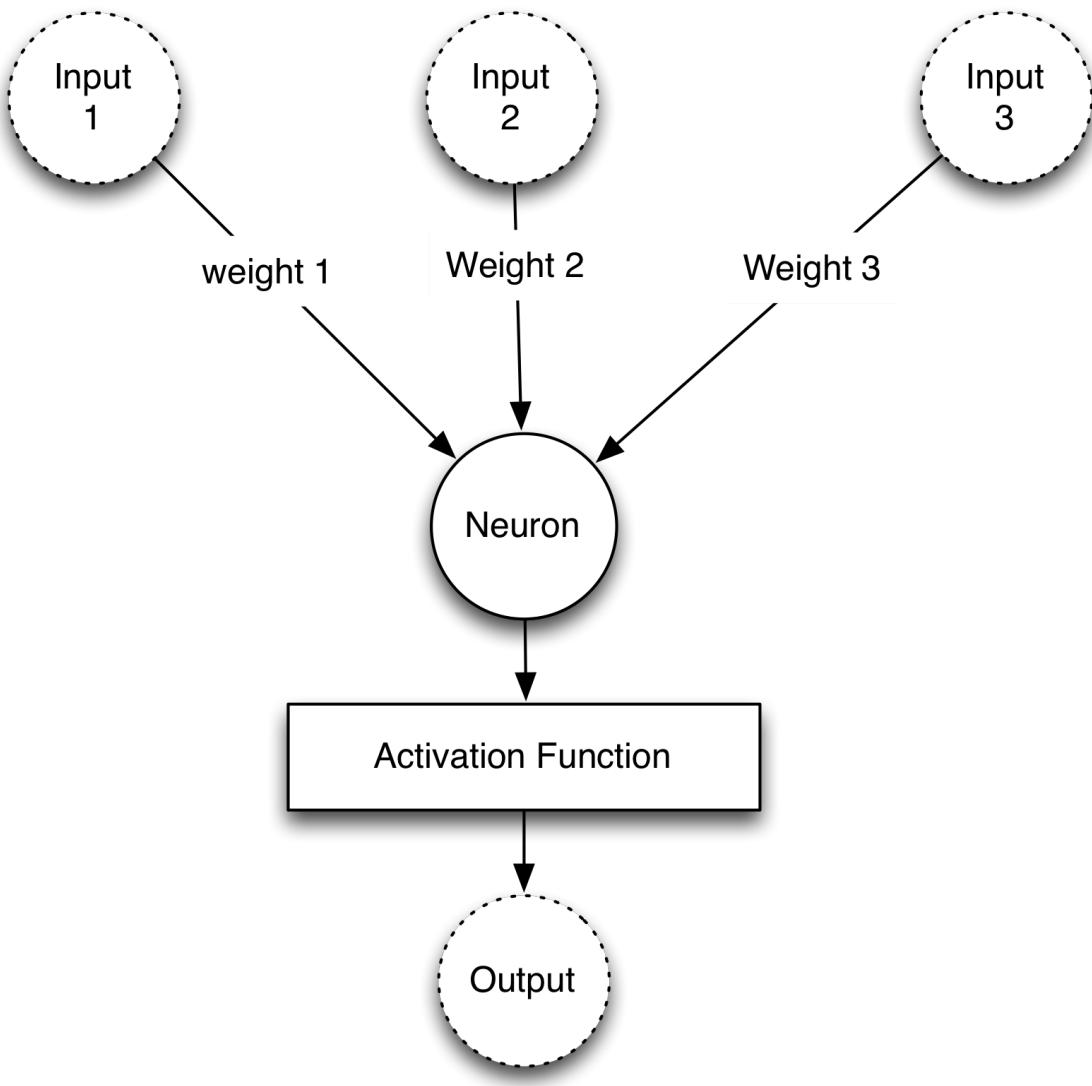
Notice that the output of the regression neural network is numeric, and the classification output is a class. Regression, or two-class classification, networks always have a single output. Classification neural networks have an output neuron for each category.

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 θ 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 ϕ (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:

$$[0.1, 0.2, 0.3]$$

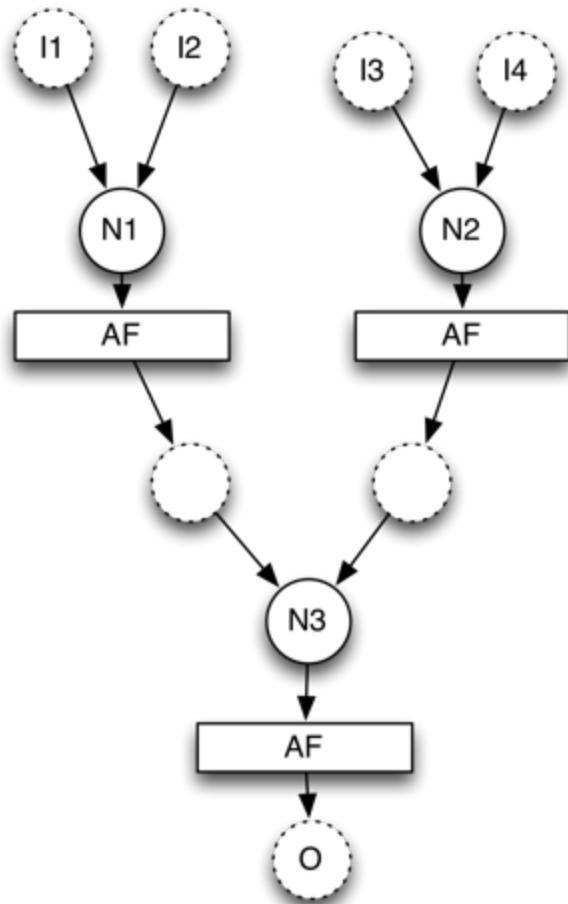
To calculate the summation, perform the following:

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

The program passes a value of 0.8 to the ϕ (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:

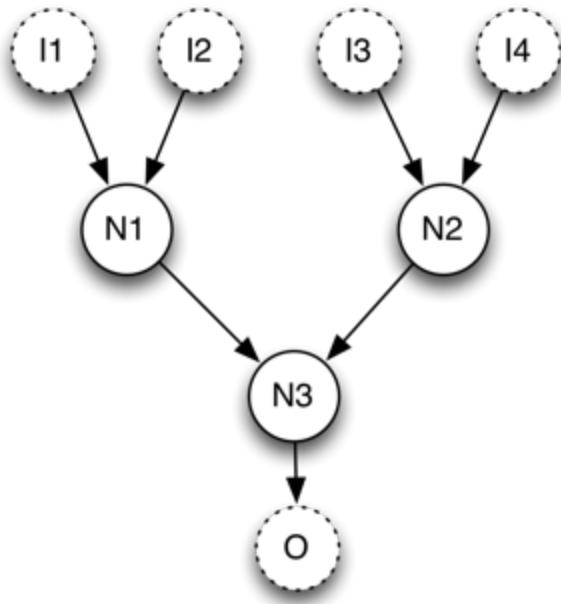
Figure 3.ANN-3: Three Neuron Neural Network



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.

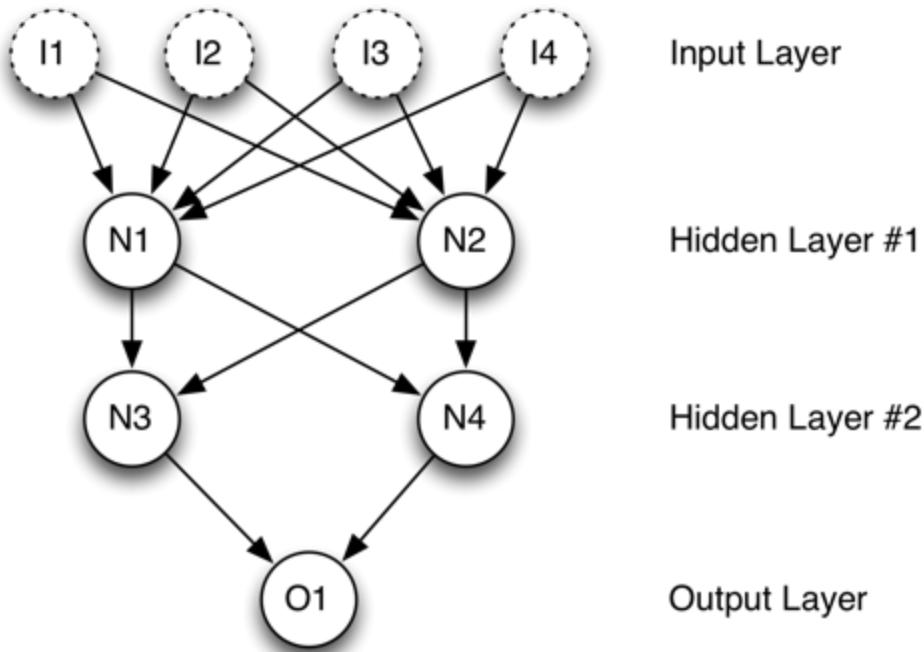
Figure 3.SANN-3: Three Neuron Neural Network



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

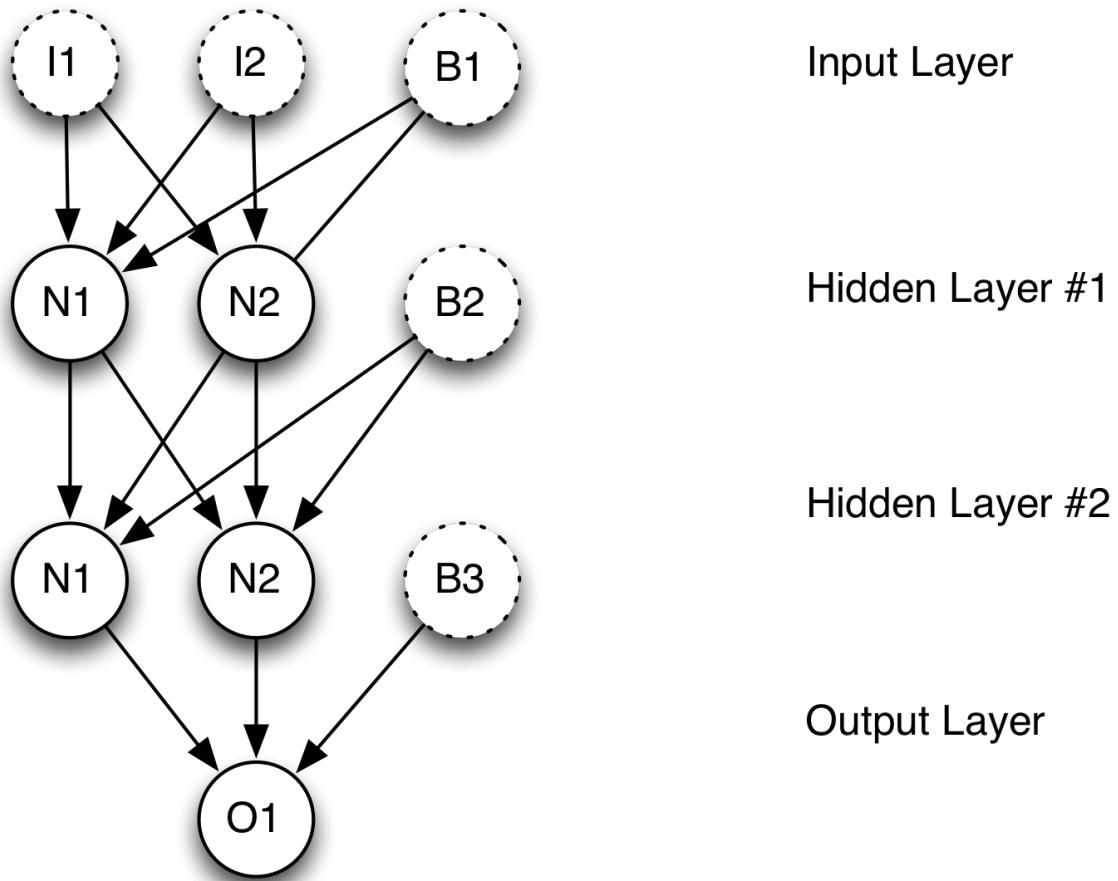
a single hidden layer is excessive. [Cite:hornik1989multilayer] Researchers have proven that a single-hidden-layer neural network can function as a universal approximator. In other words, this network should be able to learn to produce (or approximate) any output from any input as long as it has enough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

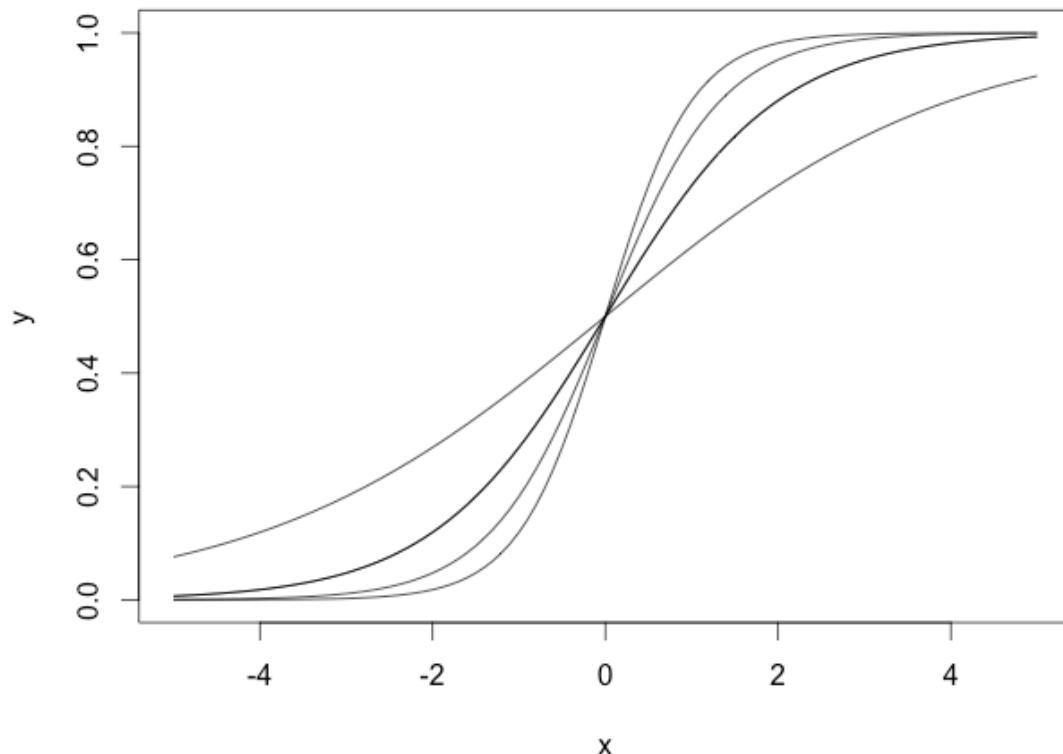
The activation functions from the previous section specify the output of a single neuron. Together, the weight and bias of a neuron shape the output of the activation to produce the desired output. To see how this process occurs, consider the following equation. It represents a single-input sigmoid activation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

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

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

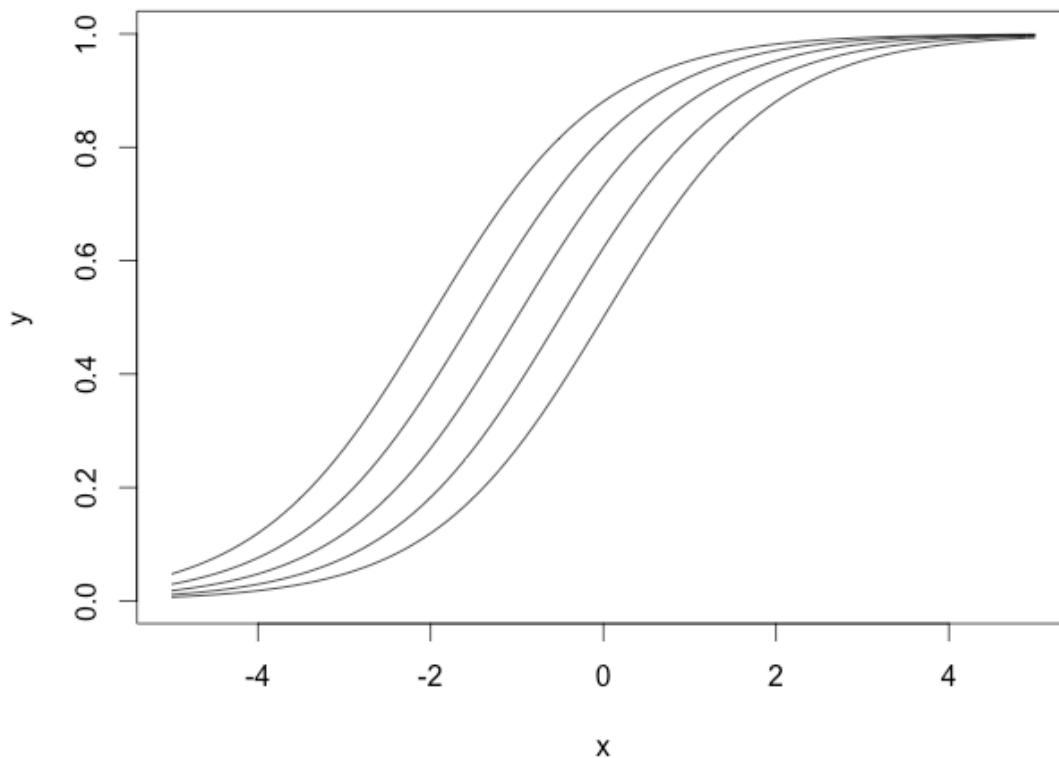
$$f(x, 1.5, 0.0)$$

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

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).

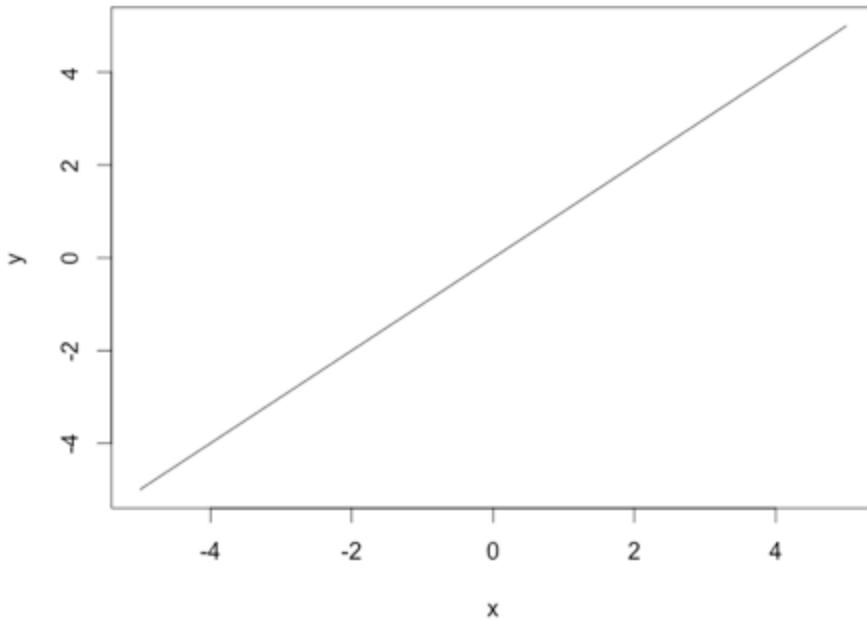
Linear Activation Function

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.

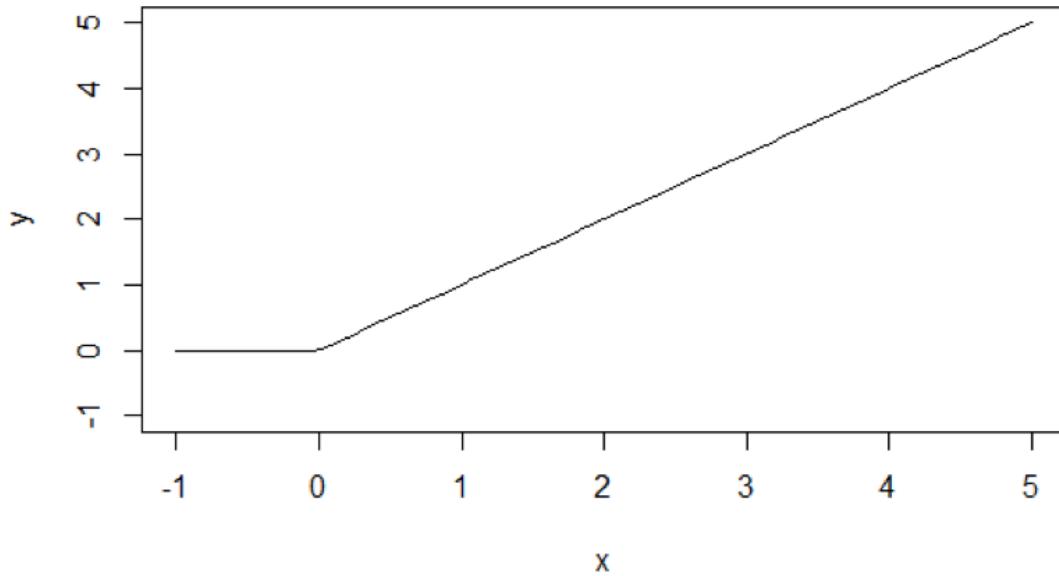
Rectified Linear Units (ReLU)

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 (ϕ) 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:

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

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

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

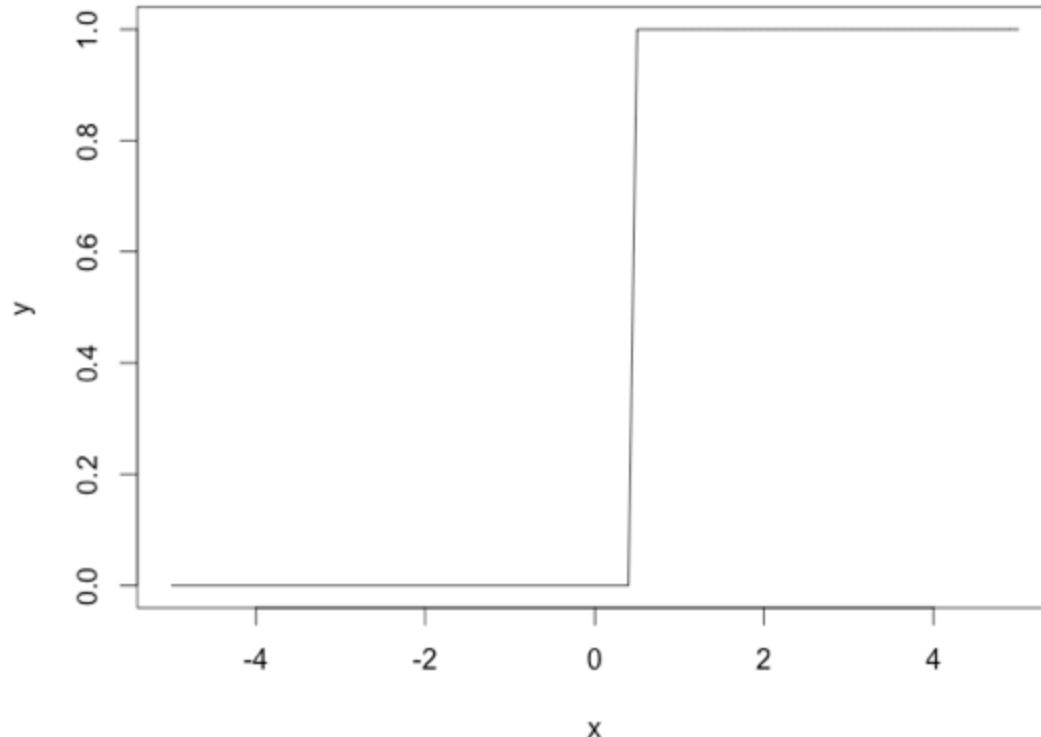
$$j_2 = \exp(0.4)/sum = 0.28839620365112$$

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



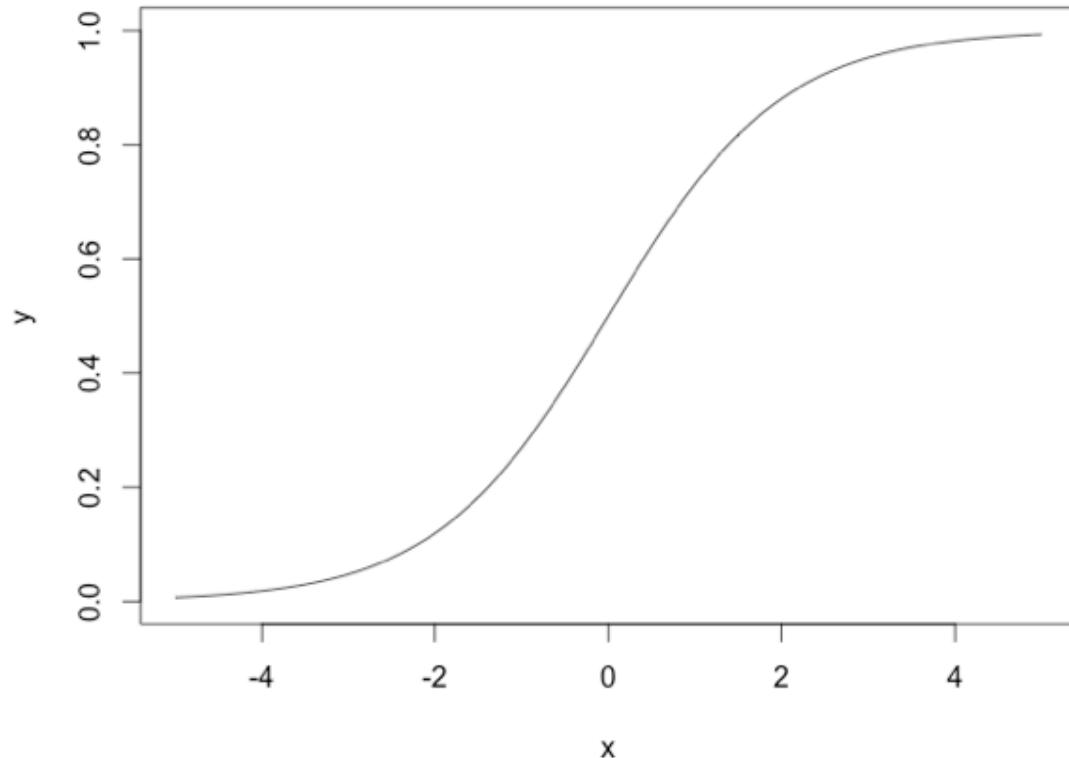
Sigmoid Activation Function

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.

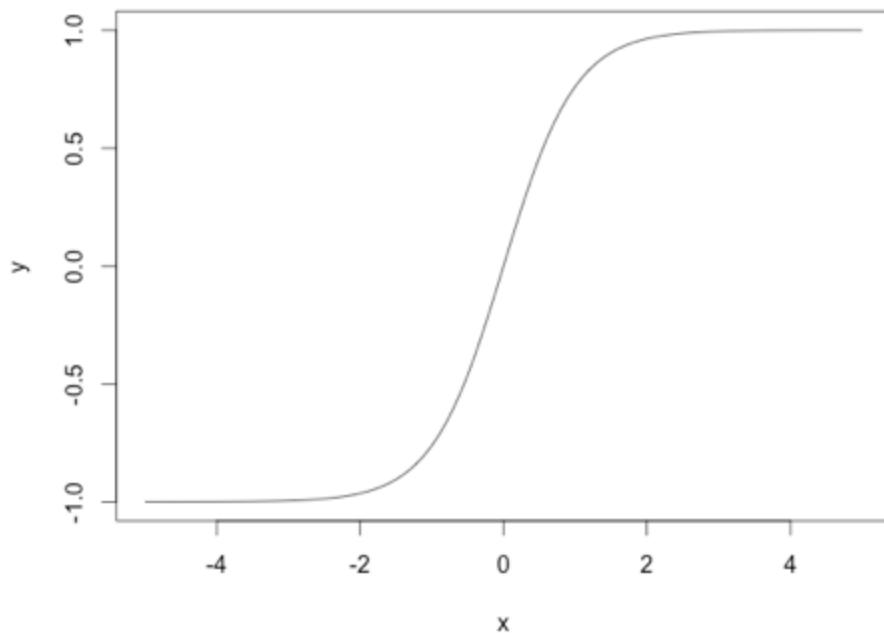
Hyperbolic Tangent Activation Function

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

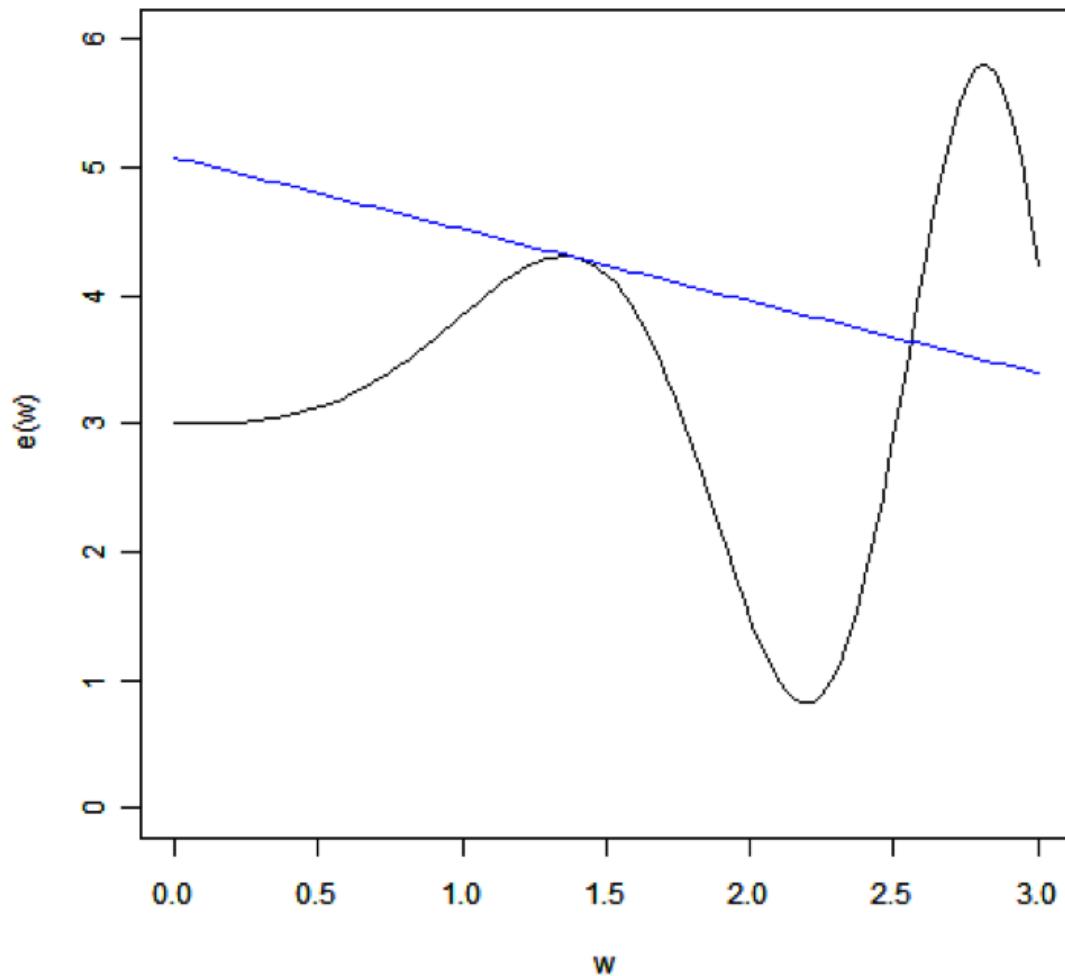


The hyperbolic tangent function has several advantages over the sigmoid activation function.

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



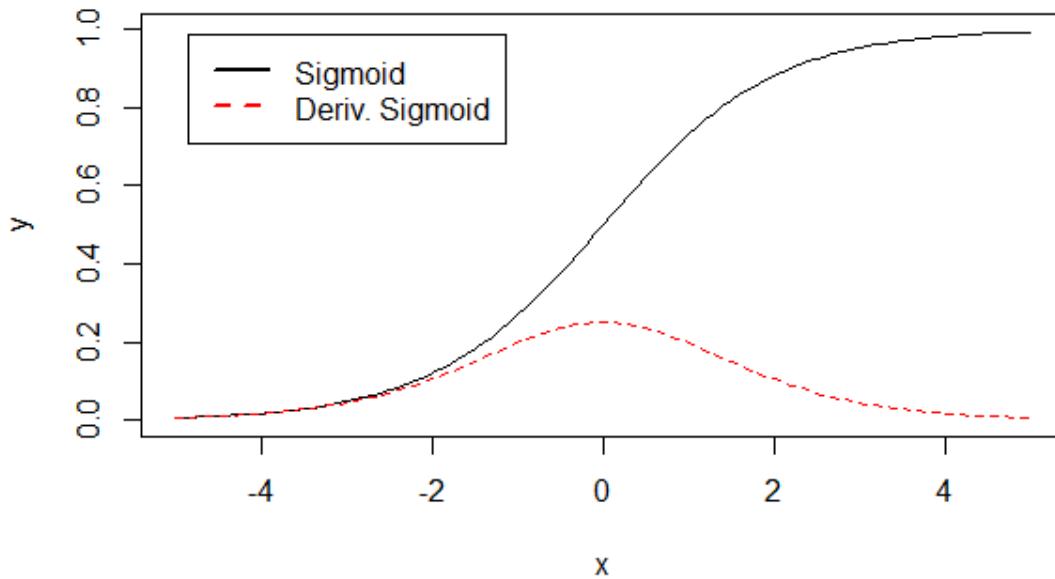
The derivative of the sigmoid function is given here:

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

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))

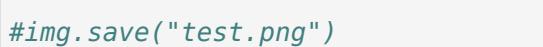
    mandelbrot_helper(grid_c, current_values, counts, cycles)
    return counts.numpy()

```

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

```

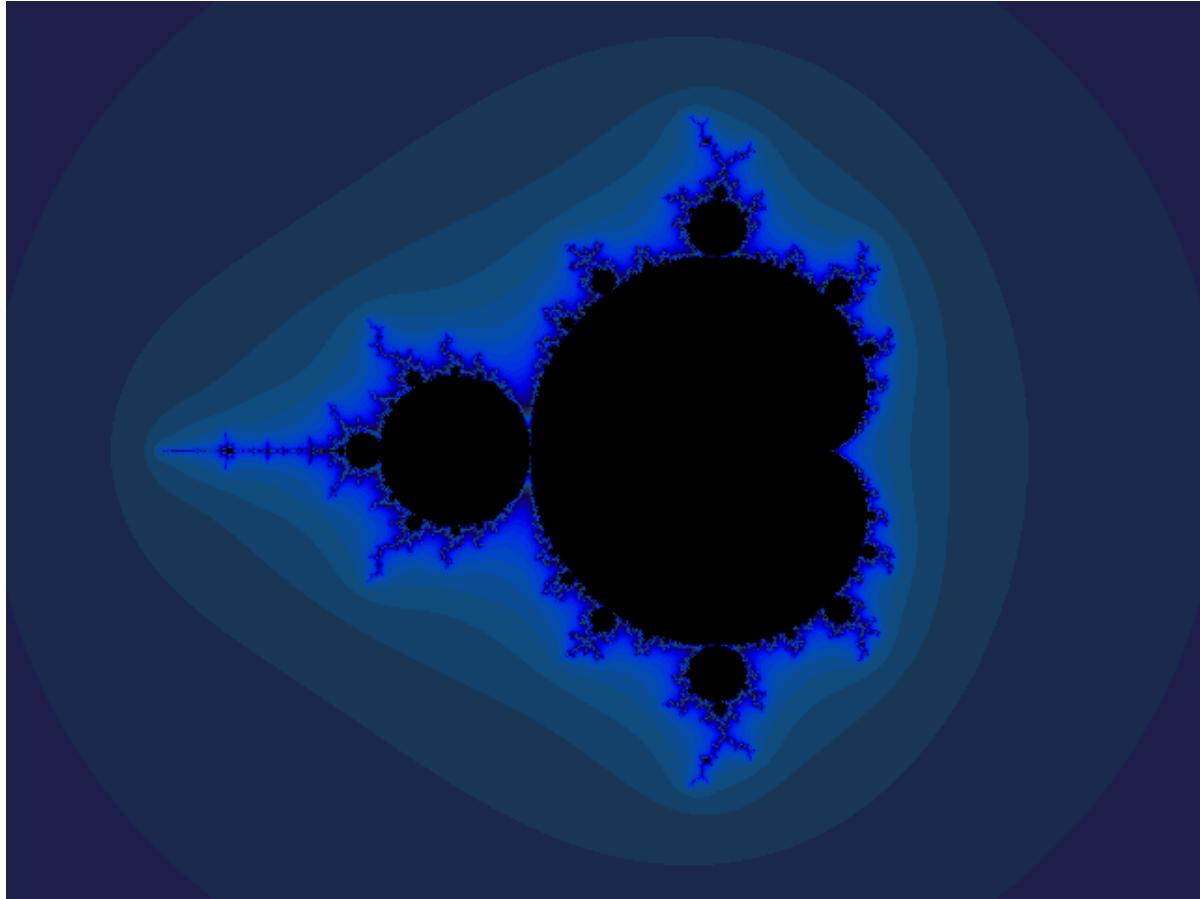
In [34]: counts = mandelbrot(
    #render_size=(3840,2160), # 4K
    #render_size=(1920,1080), # HD
    render_size=(640,480),
    center=(-0.5,0),
    zoom=4,
    cycles=200
)
img = render(counts)
print(img.size)
img



```

(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
```

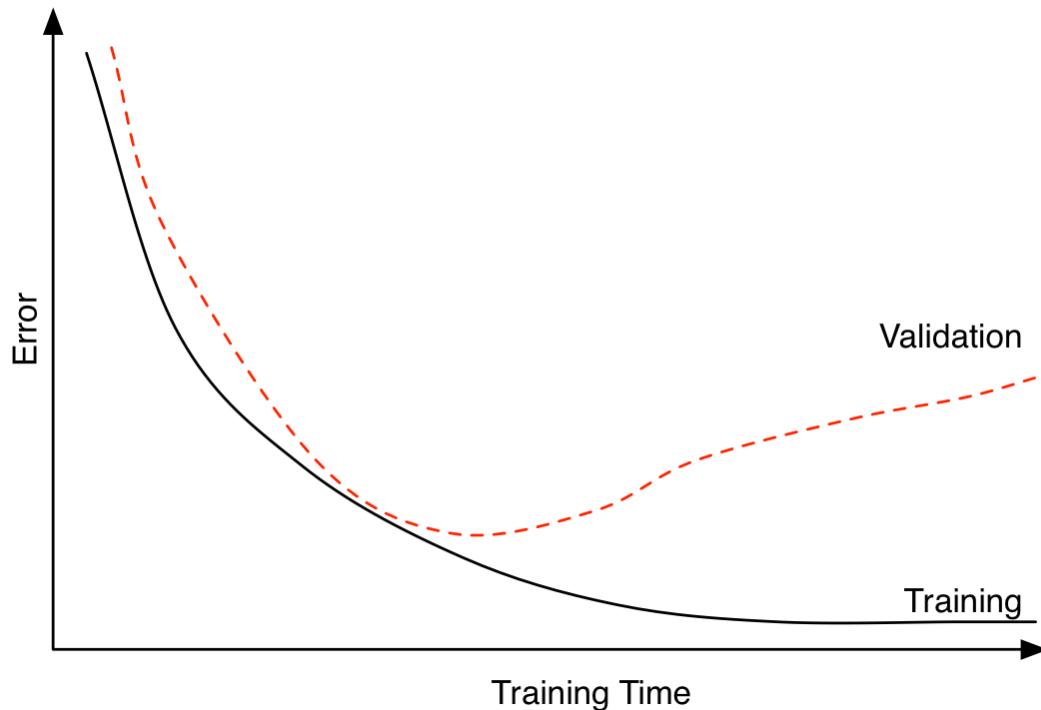
Note: not using Google CoLab

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



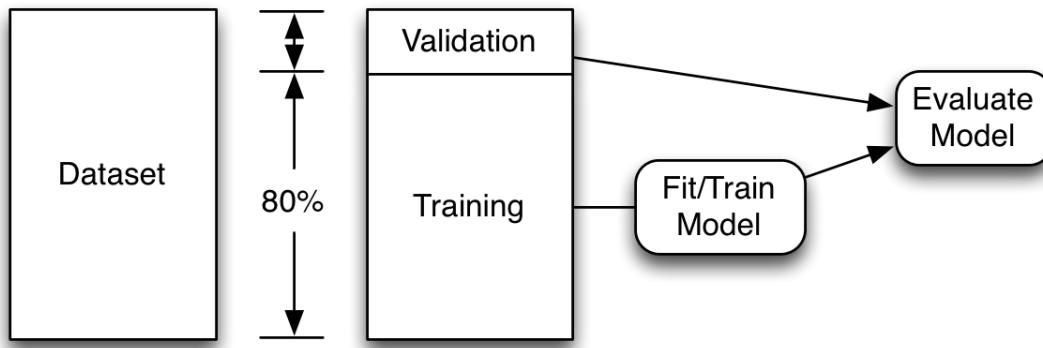
It is important to segment the original dataset into several datasets:

- **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 (μ , 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 (σ , 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 []:



T81-558: Applications of Deep Neural Networks

Module 4: Training for Tabular Data

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

Module 4 Material

- **Part 4.1: Encoding a Feature Vector for Keras Deep Learning** [\[Video\]](#) [\[Notebook\]](#)
- Part 4.2: Keras Multiclass Classification for Deep Neural Networks with ROC and AUC [\[Video\]](#) [\[Notebook\]](#)
- Part 4.3: Keras Regression for Deep Neural Networks with RMSE [\[Video\]](#) [\[Notebook\]](#)
- Part 4.4: Backpropagation, Nesterov Momentum, and ADAM Neural Network Training [\[Video\]](#) [\[Notebook\]](#)
- Part 4.5: Neural Network RMSE and Log Loss Error Calculation from Scratch [\[Video\]](#) [\[Notebook\]](#)

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 4.1: Encoding a Feature Vector for Keras Deep Learning

Neural networks can accept many types of data. We will begin with tabular data, where there are well-defined rows and columns. This data is what you would typically see in Microsoft Excel. Neural networks require numeric input. This numeric form is called a feature vector. Each input neurons receive one feature (or column) from this vector. Each row of training data typically becomes one vector. This section will see how to encode the following tabular data into a feature vector. You can see an example of tabular data below.

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

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

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

pd.set_option('display.max_columns', 9)
pd.set_option('display.max_rows', 5)

display(df)
```

	id	job	area	income	...	pop_dense	retail_dense	crime	product
0	1	vv	c	50876.0	...	0.885827	0.492126	0.071100	b
1	2	kd	c	60369.0	...	0.874016	0.342520	0.400809	c
...
1998	1999	qp	c	67949.0	...	0.909449	0.598425	0.117803	c
1999	2000	pe	c	61467.0	...	0.925197	0.539370	0.451973	c

2000 rows × 14 columns

You can make the following observations from the above data:

- The target column is the column that you seek to predict. There are several candidates here. However, we will initially use the column "product". This field specifies what product someone bought.
- There is an ID column. You should exclude this column because it contains no information useful for prediction.
- Many of these fields are numeric and might not require further processing.
- The income column does have some missing values.
- There are categorical values: job, area, and product.

To begin with, we will convert the job code into dummy variables.

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

dummies = pd.get_dummies(df['job'], prefix="job")
print(dummies.shape)

pd.set_option('display.max_columns', 9)
pd.set_option('display.max_rows', 10)

display(dummies)
```

(2000, 33)

	job_11	job_al	job_am	job_ax	...	job_rn	job_sa	job_vv	job_zz
0	0	0	0	0	...	0	0	1	0
1	0	0	0	0	...	0	0	0	0
2	0	0	0	0	...	0	0	0	0
3	1	0	0	0	...	0	0	0	0
4	0	0	0	0	...	0	0	0	0
...
1995	0	0	0	0	...	0	0	1	0
1996	0	0	0	0	...	0	0	0	0
1997	0	0	0	0	...	0	0	0	0
1998	0	0	0	0	...	0	0	0	0
1999	0	0	0	0	...	0	0	0	0

2000 rows × 33 columns

Because there are 33 different job codes, there are 33 dummy variables. We also specified a prefix because the job codes (such as "ax") are not that meaningful by themselves. Something such as "job_ax" also tells us the origin of this field.

Next, we must merge these dummies back into the main data frame. We also drop the original "job" field, as the dummies now represent it.

```
In [4]: pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)

df = pd.concat([df,dummies],axis=1)
df.drop('job', axis=1, inplace=True)

pd.set_option('display.max_columns', 9)
pd.set_option('display.max_rows', 10)
```

```
display(df)
```

	id	area	income	aspect	...	job_rn	job_sa	job_vv	job_zz
0	1	c	50876.0	13.100000	...	0	0	1	0
1	2	c	60369.0	18.625000	...	0	0	0	0
2	3	c	55126.0	34.766667	...	0	0	0	0
3	4	c	51690.0	15.808333	...	0	0	0	0
4	5	d	28347.0	40.941667	...	0	0	0	0
...
1995	1996	c	51017.0	38.233333	...	0	0	1	0
1996	1997	d	26576.0	33.358333	...	0	0	0	0
1997	1998	d	28595.0	39.425000	...	0	0	0	0
1998	1999	c	67949.0	5.733333	...	0	0	0	0
1999	2000	c	61467.0	16.891667	...	0	0	0	0

2000 rows × 46 columns

We also introduce dummy variables for the area column.

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

df = pd.concat([df,pd.get_dummies(df['area'],prefix="area")],axis=1)
df.drop('area', axis=1, inplace=True)

pd.set_option('display.max_columns', 9)
pd.set_option('display.max_rows', 10)
display(df)
```

	id	income	aspect	subscriptions	...	area_a	area_b	area_c	area_d
0	1	50876.0	13.100000		1	...	0	0	1
1	2	60369.0	18.625000		2	...	0	0	1
2	3	55126.0	34.766667		1	...	0	0	1
3	4	51690.0	15.808333		1	...	0	0	1
4	5	28347.0	40.941667		3	...	0	0	0
...
1995	1996	51017.0	38.233333		1	...	0	0	1
1996	1997	26576.0	33.358333		2	...	0	0	0
1997	1998	28595.0	39.425000		3	...	0	0	0
1998	1999	67949.0	5.733333		0	...	0	0	1
1999	2000	61467.0	16.891667		0	...	0	0	1

2000 rows × 49 columns

The last remaining transformation is to fill in missing income values.

```
In [6]: med = df['income'].median()
df['income'] = df['income'].fillna(med)
```

There are more advanced ways of filling in missing values, but they require more analysis. The idea would be to see if another field might hint at what the income was. For example, it might be beneficial to calculate a median income for each area or job category. This technique is something to keep in mind for the class Kaggle competition.

At this point, the Pandas dataframe is ready to be converted to Numpy for neural network training. We need to know a list of the columns that will make up x (the predictors or inputs) and y (the target).

The complete list of columns is:

```
In [7]: print(list(df.columns))
```

```
['id', 'income', 'aspect', 'subscriptions', 'dist_healthy', 'save_rate', 'dist_unhealthy', 'age', 'pop_dense', 'retail_dense', 'crime', 'product', 'job_11', 'job_al', 'job_am', 'job_ax', 'job_bf', 'job_by', 'job_cv', 'job_de', 'job_dz', 'job_e2', 'job_f8', 'job_gj', 'job_gv', 'job_kd', 'job_ke', 'job_kl', 'job_kp', 'job_ks', 'job_kw', 'job_mm', 'job_nb', 'job_nn', 'job_ob', 'job_pe', 'job_po', 'job_pq', 'job_pz', 'job_qp', 'job_qw', 'job_rn', 'job_sa', 'job_vv', 'job_zz', 'area_a', 'area_b', 'area_c', 'area_d']
```

This data includes both the target and predictors. We need a list with the target removed. We also remove **id** because it is not useful for prediction.

```
In [8]: x_columns = df.columns.drop('product').drop('id')
print(list(x_columns))

['income', 'aspect', 'subscriptions', 'dist_healthy', 'save_rate', 'dist_unhealthy', 'age', 'pop_dense', 'retail_dense', 'crime', 'job_11', 'job_al', 'job_am', 'job_ax', 'job_bf', 'job_by', 'job_cv', 'job_de', 'job_dz', 'job_e2', 'job_f8', 'job_gj', 'job_gv', 'job_kd', 'job_ke', 'job_kl', 'job_kp', 'job_ks', 'job_kw', 'job_mm', 'job_nb', 'job_nn', 'job_ob', 'job_pe', 'job_po', 'job_pq', 'job_pz', 'job_qp', 'job_qw', 'job_rn', 'job_sa', 'job_vv', 'job_zz', 'area_a', 'area_b', 'area_c', 'area_d']
```

Generate X and Y for a Classification Neural Network

We can now generate x and y . Note that this is how we generate y for a classification problem. Regression would not use dummies and would encode the numeric value of the target.

```
In [9]: # Convert to numpy - Classification
x_columns = df.columns.drop('product').drop('id')
x = df[x_columns].values
dummies = pd.get_dummies(df['product']) # Classification
products = dummies.columns
y = dummies.values
```

We can display the x and y matrices.

```
In [10]: print(x)
print(y)

[[5.08760000e+04 1.31000000e+01 1.00000000e+00 ... 0.00000000e+00
 1.00000000e+00 0.00000000e+00]
 [6.03690000e+04 1.86250000e+01 2.00000000e+00 ... 0.00000000e+00
 1.00000000e+00 0.00000000e+00]
 [5.51260000e+04 3.47666667e+01 1.00000000e+00 ... 0.00000000e+00
 1.00000000e+00 0.00000000e+00]
 ...
 [2.85950000e+04 3.94250000e+01 3.00000000e+00 ... 0.00000000e+00
 0.00000000e+00 1.00000000e+00]
 [6.79490000e+04 5.73333333e+00 0.00000000e+00 ... 0.00000000e+00
 1.00000000e+00 0.00000000e+00]
 [6.14670000e+04 1.68916667e+01 0.00000000e+00 ... 0.00000000e+00
 1.00000000e+00 0.00000000e+00]]
 [[0 1 0 ... 0 0 0]
 [0 0 1 ... 0 0 0]
 [0 1 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 1 0]
 [0 0 1 ... 0 0 0]
 [0 0 1 ... 0 0 0]]
```

The x and y values are now ready for a neural network. Make sure that you construct the neural network for a classification problem. Specifically,

- Classification neural networks have an output neuron count equal to the number of classes.
- Classification neural networks should use **categorical_crossentropy** and a **softmax** activation function on the output layer.

Generate X and Y for a Regression Neural Network

The program generates the x values the say way for a regression neural network. However, y does not use dummies. Make sure to replace **income** with your actual target.

```
In [11]: y = df['income'].values
```

Module 4 Assignment

You can find the first assignment here: [assignment 4](#)

```
In [ ]:
```



T81-558: Applications of Deep Neural Networks

Module 4: Training for Tabular Data

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

Module 4 Material

- Part 4.1: Encoding a Feature Vector for Keras Deep Learning [[Video](#)] [[Notebook](#)]
- **Part 4.2: Keras Multiclass Classification for Deep Neural Networks with ROC and AUC** [[Video](#)] [[Notebook](#)]
- Part 4.3: Keras Regression for Deep Neural Networks with RMSE [[Video](#)] [[Notebook](#)]
- Part 4.4: Backpropagation, Nesterov Momentum, and ADAM Neural Network Training [[Video](#)] [[Notebook](#)]
- Part 4.5: Neural Network RMSE and Log Loss Error Calculation from Scratch [[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: using Google CoLab

Part 4.2: Multiclass Classification with ROC and AUC

The output of modern neural networks can be of many different forms. However, classically, neural network output has typically been one of the following:

- **Binary Classification** - Classification between two possibilities (positive and negative). Common in medical testing, does the person has the disease (positive) or not (negative).
- **Classification** - Classification between more than 2. The iris dataset (3-way classification).
- **Regression** - Numeric prediction. How many MPG does a car get? (covered in next video)

We will look at some visualizations for all three in this section.

It is important to evaluate the false positives and negatives in the results produced by a neural network. We will now look at assessing error for both classification and regression neural networks.

Binary Classification and ROC Charts

Binary classification occurs when a neural network must choose between two options: true/false, yes/no, correct/incorrect, or buy/sell. To see how to use binary classification, we will consider a classification system for a credit card company. This system will either "issue a credit card" or "decline a credit card." This classification system must decide how to respond to a new potential customer.

When you have only two classes that you can consider, the objective function's score is the number of false-positive predictions versus the number of false negatives. False negatives and false positives are both types of errors, and it is essential to understand the difference. For the previous example, issuing a credit card would be positive. A false positive occurs when a model decides to issue a credit card to someone who will not make payments as agreed. A false negative happens when a model denies a credit card to someone who would have made payments as agreed.

Because only two options exist, we can choose the mistake that is the more serious type of error, a false positive or a false negative. For most banks issuing credit cards, a false positive is worse than a false negative. Declining a potentially good credit card holder is better than accepting a credit card holder who would cause the bank to undertake expensive collection activities.

Consider the following program that uses the [wcbreast_wdbc dataset](#) to classify if a breast tumor is cancerous (malignant) or not (benign).

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

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

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

display(df)
```

	id	diagnosis	...	worst_symmetry	worst_fractal_dimension
0	842302	M	...	0.4601	0.11890
1	842517	M	...	0.2750	0.08902
...
567	927241	M	...	0.4087	0.12400
568	92751	B	...	0.2871	0.07039

569 rows × 32 columns

ROC curves can be a bit confusing. However, they are prevalent in analytics. It is essential to know how to read them. Even their name is confusing. Do not worry about their name; the receiver operating characteristic curve (ROC) comes from electrical engineering (EE).

Binary classification is common in medical testing. Often you want to diagnose if someone has a disease. This diagnosis can lead to two types of errors, known as false positives and false negatives:

- **False Positive** - Your test (neural network) indicated that the patient had the disease; however, the patient did not.
- **False Negative** - Your test (neural network) indicated that the patient did not have the disease; however, the patient did have the disease.
- **True Positive** - Your test (neural network) correctly identified that the patient had the disease.
- **True Negative** - Your test (neural network) correctly identified that the patient did not have the disease.

Figure 4.ETYP shows you these types of errors.

Figure 4.ETYP: Type of Error

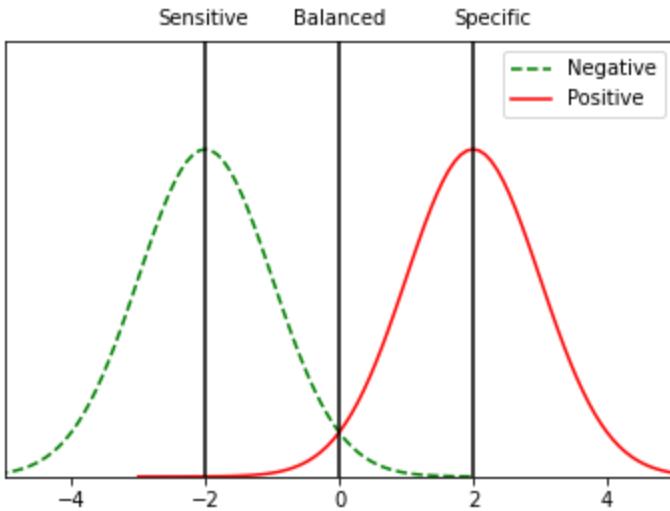
True vs False Positives	Type-1 Error	Sensitivity of Test
True vs False Negatives	Type-2 Error	Specificity of Test

Neural networks classify in terms of the probability of it being positive. However, at what possibility do you give a positive result? Is the cutoff 50%? 90%? Where you set, this cutoff is called the threshold. Anything above the cutoff is positive; anything below is negative. Setting this cutoff allows the model to be more sensitive or specific:

More info on Sensitivity vs. Specificity: [Khan Academy](#)

```
In [3]: %matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import scipy.stats as stats
import math

mu1 = -2
mu2 = 2
variance = 1
sigma = math.sqrt(variance)
x1 = np.linspace(mu1 - 5*sigma, mu1 + 4*sigma, 100)
x2 = np.linspace(mu2 - 5*sigma, mu2 + 4*sigma, 100)
plt.plot(x1, stats.norm.pdf(x1, mu1, sigma)/1,color="green",
          linestyle='dashed')
plt.plot(x2, stats.norm.pdf(x2, mu2, sigma)/1,color="red")
plt.axvline(x=-2,color="black")
plt.axvline(x=0,color="black")
plt.axvline(x=+2,color="black")
plt.text(-2.7,0.55,"Sensitive")
plt.text(-0.7,0.55,"Balanced")
plt.text(1.7,0.55,"Specific")
plt.ylim([0,0.53])
plt.xlim([-5,5])
plt.legend(['Negative','Positive'])
plt.yticks([])
plt.show()
```



We will now train a neural network for the Wisconsin breast cancer dataset. We begin by preprocessing the data. Because we have all numeric data, we compute a z-score for each column.

```
In [4]: from scipy.stats import zscore

x_columns = df.columns.drop('diagnosis').drop('id')
for col in x_columns:
    df[col] = zscore(df[col])

# Convert to numpy - Regression
x = df[x_columns].values
y = df['diagnosis'].map({'M':1, "B":0}).values # Binary classification,
                                                # M is 1 and B is 0
```

We can now define two functions. The first function plots a confusion matrix. The second function plots a ROC chart.

```
In [5]: %matplotlib inline
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc

# Plot a confusion matrix.
# cm is the confusion matrix, names are the names of the classes.
def plot_confusion_matrix(cm, names, title='Confusion matrix',
                          cmap=plt.cm.Blues):
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(names))
    plt.xticks(tick_marks, names, rotation=45)
    plt.yticks(tick_marks, names)
    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

```
# Plot an ROC. pred - the predictions, y - the expected output.
def plot_roc(pred,y):
    fpr, tpr, _ = roc_curve(y, pred)
    roc_auc = auc(fpr, tpr)

    plt.figure()
    plt.plot(fpr, tpr, label='ROC curve (area = %0.2f)' % roc_auc)
    plt.plot([0, 1], [0, 1], 'k--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristic (ROC)')
    plt.legend(loc="lower right")
    plt.show()
```

ROC Chart Example

The following code demonstrates how to implement a ROC chart in Python.

```
In [6]: # Classification neural network
import numpy as np
import tensorflow.keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.model_selection import train_test_split

# Split into train/test
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)

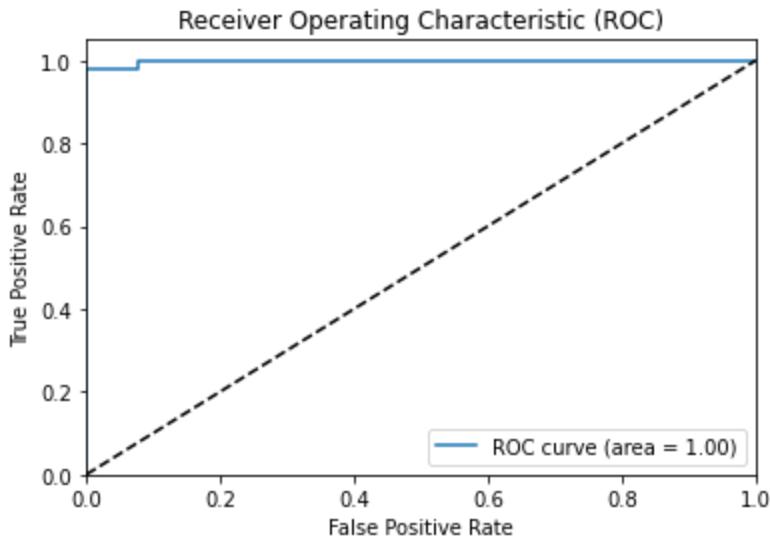
model = Sequential()
model.add(Dense(100, input_dim=x.shape[1], activation='relu',
               kernel_initializer='random_normal'))
model.add(Dense(50, activation='relu', kernel_initializer='random_normal'))
model.add(Dense(25, activation='relu', kernel_initializer='random_normal'))
model.add(Dense(1, activation='sigmoid', kernel_initializer='random_normal'))
model.compile(loss='binary_crossentropy',
              optimizer=tensorflow.keras.optimizers.Adam(),
              metrics=['accuracy'])
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)
```

```
Epoch 1/1000
14/14 - 2s - loss: 0.6850 - accuracy: 0.8427 - val_loss: 0.6673 - val_accuracy: 0.9441 - 2s/epoch - 164ms/step
Epoch 2/1000
14/14 - 0s - loss: 0.6261 - accuracy: 0.9319 - val_loss: 0.5393 - val_accuracy: 0.9720 - 130ms/epoch - 9ms/step
Epoch 3/1000
14/14 - 0s - loss: 0.4265 - accuracy: 0.9413 - val_loss: 0.2536 - val_accuracy: 0.9720 - 148ms/epoch - 11ms/step
Epoch 4/1000
14/14 - 0s - loss: 0.2112 - accuracy: 0.9437 - val_loss: 0.1067 - val_accuracy: 0.9720 - 145ms/epoch - 10ms/step
Epoch 5/1000
14/14 - 0s - loss: 0.1171 - accuracy: 0.9624 - val_loss: 0.0644 - val_accuracy: 0.9790 - 110ms/epoch - 8ms/step
Epoch 6/1000
14/14 - 0s - loss: 0.0852 - accuracy: 0.9789 - val_loss: 0.0552 - val_accuracy: 0.9860 - 114ms/epoch - 8ms/step
Epoch 7/1000
14/14 - 0s - loss: 0.0744 - accuracy: 0.9789 - val_loss: 0.0541 - val_accuracy: 0.9860 - 115ms/epoch - 8ms/step
Epoch 8/1000
14/14 - 0s - loss: 0.0662 - accuracy: 0.9812 - val_loss: 0.0473 - val_accuracy: 0.9930 - 154ms/epoch - 11ms/step
Epoch 9/1000
14/14 - 0s - loss: 0.0602 - accuracy: 0.9812 - val_loss: 0.0493 - val_accuracy: 0.9860 - 90ms/epoch - 6ms/step
Epoch 10/1000
14/14 - 0s - loss: 0.0548 - accuracy: 0.9859 - val_loss: 0.0468 - val_accuracy: 0.9860 - 225ms/epoch - 16ms/step
Epoch 11/1000
14/14 - 0s - loss: 0.0491 - accuracy: 0.9836 - val_loss: 0.0484 - val_accuracy: 0.9860 - 133ms/epoch - 10ms/step
Epoch 12/1000
14/14 - 0s - loss: 0.0458 - accuracy: 0.9836 - val_loss: 0.0486 - val_accuracy: 0.9860 - 119ms/epoch - 8ms/step
Epoch 13/1000
Restoring model weights from the end of the best epoch: 8.
14/14 - 0s - loss: 0.0417 - accuracy: 0.9883 - val_loss: 0.0477 - val_accuracy: 0.9860 - 124ms/epoch - 9ms/step
Epoch 13: early stopping
```

Out[6]: <keras.callbacks.History at 0x7f6a8aee46d0>

In [7]: pred = model.predict(x_test)
plot_roc(pred,y_test)



Multiclass Classification Error Metrics

If you want to predict more than one outcome, you will need more than one output neuron. Because a single neuron can predict two results, a neural network with two output neurons is somewhat rare. If there are three or more outcomes, there will be three or more output neurons. The following sections will examine several metrics for evaluating classification error. We will assess the following classification neural network.

```
In [8]: import pandas as pd
from scipy.stats import zscore

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

# Generate dummies for job
df = pd.concat([df,pd.get_dummies(df['job'],prefix="job")],axis=1)
df.drop('job', axis=1, inplace=True)

# Generate dummies for area
df = pd.concat([df,pd.get_dummies(df['area'],prefix="area")],axis=1)
df.drop('area', axis=1, inplace=True)

# Missing values for income
med = df['income'].median()
df['income'] = df['income'].fillna(med)

# Standardize ranges
df['income'] = zscore(df['income'])
df['aspect'] = zscore(df['aspect'])
df['save_rate'] = zscore(df['save_rate'])
df['age'] = zscore(df['age'])
df['subscriptions'] = zscore(df['subscriptions'])
```

```
# Convert to numpy - Classification
x_columns = df.columns.drop('product').drop('id')
x = df[x_columns].values
dummies = pd.get_dummies(df['product']) # Classification
products = dummies.columns
y = dummies.values
```

```
In [9]: # Classification neural network
import numpy as np
import tensorflow.keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.model_selection import train_test_split

# Split into train/test
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)

model = Sequential()
model.add(Dense(100, input_dim=x.shape[1], activation='relu',
               kernel_initializer='random_normal'))
model.add(Dense(50, activation='relu', kernel_initializer='random_normal'))
model.add(Dense(25, activation='relu', kernel_initializer='random_normal'))
model.add(Dense(y.shape[1], activation='softmax',
               kernel_initializer='random_normal'))
model.compile(loss='categorical_crossentropy',
              optimizer=tensorflow.keras.optimizers.Adam(),
              metrics=['accuracy'])
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)
```

Epoch 1/1000
47/47 - 2s - loss: 1.4456 - accuracy: 0.4753 - val_loss: 1.1348 - val_accuracy: 0.4980 - 2s/epoch - 45ms/step
Epoch 2/1000
47/47 - 0s - loss: 1.1477 - accuracy: 0.4720 - val_loss: 1.0890 - val_accuracy: 0.4980 - 249ms/epoch - 5ms/step
Epoch 3/1000
47/47 - 0s - loss: 1.0847 - accuracy: 0.5040 - val_loss: 1.0205 - val_accuracy: 0.5380 - 482ms/epoch - 10ms/step
Epoch 4/1000
47/47 - 0s - loss: 0.9608 - accuracy: 0.5920 - val_loss: 0.9546 - val_accuracy: 0.5740 - 309ms/epoch - 7ms/step
Epoch 5/1000
47/47 - 0s - loss: 0.8508 - accuracy: 0.6480 - val_loss: 0.8616 - val_accuracy: 0.6600 - 290ms/epoch - 6ms/step
Epoch 6/1000
47/47 - 0s - loss: 0.7942 - accuracy: 0.6660 - val_loss: 0.8018 - val_accuracy: 0.6900 - 298ms/epoch - 6ms/step
Epoch 7/1000
47/47 - 0s - loss: 0.7581 - accuracy: 0.6927 - val_loss: 0.8044 - val_accuracy: 0.6740 - 271ms/epoch - 6ms/step
Epoch 8/1000
47/47 - 0s - loss: 0.7434 - accuracy: 0.6893 - val_loss: 0.7885 - val_accuracy: 0.6660 - 246ms/epoch - 5ms/step
Epoch 9/1000
47/47 - 0s - loss: 0.7522 - accuracy: 0.6867 - val_loss: 0.7835 - val_accuracy: 0.6720 - 281ms/epoch - 6ms/step
Epoch 10/1000
47/47 - 0s - loss: 0.7158 - accuracy: 0.6987 - val_loss: 0.7727 - val_accuracy: 0.6840 - 327ms/epoch - 7ms/step
Epoch 11/1000
47/47 - 0s - loss: 0.7129 - accuracy: 0.6887 - val_loss: 0.7966 - val_accuracy: 0.6820 - 231ms/epoch - 5ms/step
Epoch 12/1000
47/47 - 0s - loss: 0.7105 - accuracy: 0.6947 - val_loss: 0.7700 - val_accuracy: 0.6620 - 239ms/epoch - 5ms/step
Epoch 13/1000
47/47 - 0s - loss: 0.7119 - accuracy: 0.6940 - val_loss: 0.7680 - val_accuracy: 0.6700 - 254ms/epoch - 5ms/step
Epoch 14/1000
47/47 - 0s - loss: 0.6934 - accuracy: 0.7047 - val_loss: 0.7743 - val_accuracy: 0.6600 - 289ms/epoch - 6ms/step
Epoch 15/1000
47/47 - 0s - loss: 0.6904 - accuracy: 0.7093 - val_loss: 0.7564 - val_accuracy: 0.6860 - 266ms/epoch - 6ms/step
Epoch 16/1000
47/47 - 0s - loss: 0.6837 - accuracy: 0.7007 - val_loss: 0.7423 - val_accuracy: 0.7000 - 297ms/epoch - 6ms/step
Epoch 17/1000
47/47 - 0s - loss: 0.6783 - accuracy: 0.7120 - val_loss: 0.7519 - val_accuracy: 0.6840 - 258ms/epoch - 5ms/step
Epoch 18/1000
47/47 - 0s - loss: 0.6665 - accuracy: 0.7153 - val_loss: 0.7582 - val_accuracy: 0.6660 - 259ms/epoch - 6ms/step
Epoch 19/1000
47/47 - 0s - loss: 0.6702 - accuracy: 0.7000 - val_loss: 0.7504 - val_accuracy:

```
cy: 0.6880 - 271ms/epoch - 6ms/step
Epoch 20/1000
47/47 - 0s - loss: 0.6624 - accuracy: 0.7147 - val_loss: 0.7527 - val_accuracy: 0.6800 - 328ms/epoch - 7ms/step
Epoch 21/1000
Restoring model weights from the end of the best epoch: 16.
47/47 - 1s - loss: 0.6558 - accuracy: 0.7160 - val_loss: 0.7653 - val_accuracy: 0.6720 - 527ms/epoch - 11ms/step
Epoch 21: early stopping
Out[9]: <keras.callbacks.History at 0x7f6a8ad5db50>
```

Calculate Classification Accuracy

Accuracy is the number of rows where the neural network correctly predicted the target class. Accuracy is only used for classification, not regression.

$$\text{accuracy} = \frac{c}{N}$$

Where c is the number correct and N is the size of the evaluated set (training or validation). Higher accuracy numbers are desired.

As we just saw, by default, Keras will return the percent probability for each class. We can change these prediction probabilities into the actual iris predicted with **argmax**.

```
In [10]: pred = model.predict(x_test)
pred = np.argmax(pred, axis=1)
# raw probabilities to chosen class (highest probability)
```

Now that we have the actual iris flower predicted, we can calculate the percent accuracy (how many were correctly classified).

```
In [11]: from sklearn import metrics

y_compare = np.argmax(y_test, axis=1)
score = metrics.accuracy_score(y_compare, pred)
print("Accuracy score: {}".format(score))
```

Accuracy score: 0.7

Calculate Classification Log Loss

Accuracy is like a final exam with no partial credit. However, neural networks can predict a probability of each of the target classes. Neural networks will give high probabilities to predictions that are more likely. Log loss is an error metric that penalizes confidence in wrong answers. Lower log loss values are desired.

The following code shows the output of predict_proba:

```
In [12]: from IPython.display import display

# Don't display numpy in scientific notation
np.set_printoptions(precision=4)
np.set_printoptions(suppress=True)

# Generate predictions
pred = model.predict(x_test)

print("Numpy array of predictions")
display(pred[0:5])

print("As percent probability")
print(pred[0]*100)

score = metrics.log_loss(y_test, pred)
print("Log loss score: {}".format(score))

# raw probabilities to chosen class (highest probability)
pred = np.argmax(pred, axis=1)
```

Numpy array of predictions
array([[0. , 0.1201, 0.7286, 0.1494, 0.0018, 0. , 0. ,
 [0. , 0.6962, 0.3016, 0.0001, 0.0022, 0. , 0. , 0.],
 [0. , 0.7234, 0.2708, 0.0003, 0.0053, 0.0001, 0. , 0.],
 [0. , 0.3836, 0.6039, 0.0086, 0.0039, 0. , 0. , 0.],
 [0. , 0.0609, 0.6303, 0.3079, 0.001 , 0. , 0. , 0.]],
 dtype=float32)
As percent probability
[0.0001 12.0143 72.8578 14.9446 0.1823 0.0009 0.0001]
Log loss score: 0.7423401429280638

Log loss is calculated as follows:

$$\text{log loss} = -\frac{1}{N} \sum_{i=1}^N (y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i))$$

You should use this equation only as an objective function for classifications that have two outcomes. The variable \hat{y} is the neural network's prediction, and the variable y is the known correct answer. In this case, y will always be 0 or 1. The training data have no probabilities. The neural network classifies it either into one class (1) or the other (0).

The variable N represents the number of elements in the training set the number of questions in the test. We divide by N because this process is customary for an average. We also begin the equation with a negative because the log function is always negative over the domain 0 to 1. This negation allows a positive score for the training to minimize.

You will notice two terms are separated by the addition (+). Each contains a log function. Because y will be either 0 or 1, then one of these two terms will cancel

out to 0. If y is 0, then the first term will reduce to 0. If y is 1, then the second term will be 0.

If your prediction for the first class of a two-class prediction is $y\text{-hat}$, then your prediction for the second class is 1 minus $y\text{-hat}$. Essentially, if your prediction for class A is 70% (0.7), then your prediction for class B is 30% (0.3). Your score will increase by the log of your prediction for the correct class. If the neural network had predicted 1.0 for class A, and the correct answer was A, your score would increase by $\log(1)$, which is 0. For log loss, we seek a low score, so a correct answer results in 0. Some of these log values for a neural network's probability estimate for the correct class:

- $-\log(1.0) = 0$
- $-\log(0.95) = 0.02$
- $-\log(0.9) = 0.05$
- $-\log(0.8) = 0.1$
- $-\log(0.5) = 0.3$
- $-\log(0.1) = 1$
- $-\log(0.01) = 2$
- $-\log(1.0e-12) = 12$
- $-\log(0.0) = \text{negative infinity}$

As you can see, giving a low confidence to the correct answer affects the score the most. Because $\log(0)$ is negative infinity, we typically impose a minimum value. Of course, the above log values are for a single training set element. We will average the log values for the entire training set.

The log function is useful to penalizing wrong answers. The following code demonstrates the utility of the log function:

```
In [13]: %matplotlib inline
from matplotlib.pyplot import figure, show
from numpy import arange, sin, pi

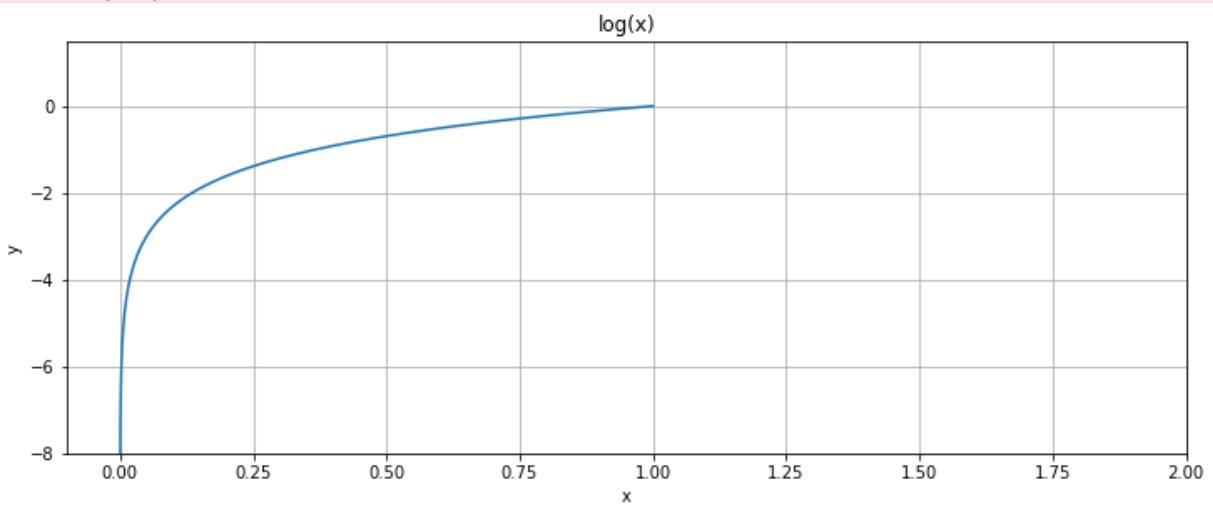
#t = arange(1e-5, 5.0, 0.00001)
#t = arange(1.0, 5.0, 0.00001) # computer scientists
t = arange(0.0, 1.0, 0.00001) # data scientists

fig = figure(1, figsize=(12, 10))

ax1 = fig.add_subplot(211)
ax1.plot(t, np.log(t))
ax1.grid(True)
ax1.set_xlim((-8, 1.5))
ax1.set_xlabel('x')
ax1.set_ylabel('y')
ax1.set_title('log(x)')
```

```
show()
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:12: RuntimeWarning: divide by zero encountered in log
  if sys.path[0] == '':
```



Confusion Matrix

A confusion matrix shows which predicted classes are often confused for the other classes. The vertical axis (y) represents the true labels and the horizontal axis (x) represents the predicted labels. When the true label and predicted label are the same, the highest values occur down the diagonal extending from the upper left to the lower right. The other values, outside the diagonal, represent incorrect predictions. For example, in the confusion matrix below, the value in row 2, column 1 shows how often the predicted value A occurred when it should have been B.

In [14]:

```
import numpy as np
from sklearn import svm, datasets
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix

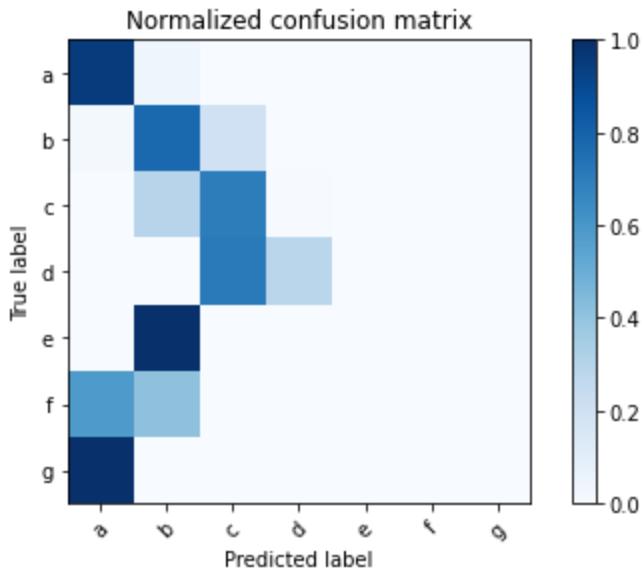
# Compute confusion matrix
cm = confusion_matrix(y_compare, pred)
np.set_printoptions(precision=2)

# Normalize the confusion matrix by row (i.e. by the number of samples
# in each class)
cm_normalized = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
print('Normalized confusion matrix')
print(cm_normalized)
plt.figure()
plot_confusion_matrix(cm_normalized, products,
                      title='Normalized confusion matrix')

plt.show()
```

Normalized confusion matrix

```
[[0.95 0.05 0.  0.  0.  0.  0. ]
 [0.02 0.78 0.2 0.  0.  0.  0. ]
 [0.  0.29 0.7 0.01 0.  0.  0. ]
 [0.  0.  0.71 0.29 0.  0.  0. ]
 [0.  1.  0.  0.  0.  0.  0. ]
 [0.59 0.41 0.  0.  0.  0.  0. ]
 [1.  0.  0.  0.  0.  0.  0. ]]
```



In [14]:



T81-558: Applications of Deep Neural Networks

Module 4: Training for Tabular Data

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

Module 4 Material

- Part 4.1: Encoding a Feature Vector for Keras Deep Learning [[Video](#)] [[Notebook](#)]
- Part 4.2: Keras Multiclass Classification for Deep Neural Networks with ROC and AUC [[Video](#)] [[Notebook](#)]
- **Part 4.3: Keras Regression for Deep Neural Networks with RMSE** [[Video](#)] [[Notebook](#)]
- Part 4.4: Backpropagation, Nesterov Momentum, and ADAM Neural Network Training [[Video](#)] [[Notebook](#)]
- Part 4.5: Neural Network RMSE and Log Loss Error Calculation from Scratch [[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 4.3: Keras Regression for Deep Neural Networks with RMSE

We evaluate regression results differently than classification. Consider the following code that trains a neural network for regression on the data set **jh-simple-dataset.csv**. We begin by preparing the data set.

In [2]:

```
import pandas as pd
from scipy.stats import zscore
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

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

# Generate dummies for job
df = pd.concat([df,pd.get_dummies(df['job'],prefix="job")],axis=1)
df.drop('job', axis=1, inplace=True)

# Generate dummies for area
df = pd.concat([df,pd.get_dummies(df['area'],prefix="area")],axis=1)
df.drop('area', axis=1, inplace=True)

# Generate dummies for product
df = pd.concat([df,pd.get_dummies(df['product'],prefix="product")],axis=1)
df.drop('product', axis=1, inplace=True)

# Missing values for income
med = df['income'].median()
df['income'] = df['income'].fillna(med)

# Standardize ranges
df['income'] = zscore(df['income'])
df['aspect'] = zscore(df['aspect'])
df['save_rate'] = zscore(df['save_rate'])
df['subscriptions'] = zscore(df['subscriptions'])

# Convert to numpy - Classification
x_columns = df.columns.drop('age').drop('id')
x = df[x_columns].values
y = df['age'].values

# Create train/test
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)
```

Next, we create a neural network to fit the data we just loaded.

```
In [3]: from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping

# 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 1500 samples, validate on 500 samples
Epoch 1/1000
1500/1500 - 1s - loss: 1905.4454 - val_loss: 1628.1341
Epoch 2/1000
1500/1500 - 0s - loss: 1331.4213 - val_loss: 889.0575
Epoch 3/1000
1500/1500 - 0s - loss: 554.8426 - val_loss: 303.7261
Epoch 4/1000
1500/1500 - 0s - loss: 276.2087 - val_loss: 241.2495
Epoch 5/1000
1500/1500 - 0s - loss: 232.2832 - val_loss: 208.2143
Epoch 6/1000
1500/1500 - 0s - loss: 198.5331 - val_loss: 179.5262
Epoch 7/1000
1500/1500 - 0s - loss: 169.0791 - val_loss: 154.5270
Epoch 8/1000
1500/1500 - 0s - loss: 144.1286 - val_loss: 132.8691
Epoch 9/1000
1500/1500 - 0s - loss: 122.9873 - val_loss: 115.0928
Epoch 10/1000
1500/1500 - 0s - loss: 104.7249 - val_loss: 98.7375
Epoch 11/1000
1500/1500 - 0s - loss: 89.8292 - val_loss: 86.2749
Epoch 12/1000
1500/1500 - 0s - loss: 77.3071 - val_loss: 75.0022
Epoch 13/1000
1500/1500 - 0s - loss: 67.0604 - val_loss: 66.1396
Epoch 14/1000
1500/1500 - 0s - loss: 58.9584 - val_loss: 58.4367
Epoch 15/1000
1500/1500 - 0s - loss: 51.2491 - val_loss: 52.7136
Epoch 16/1000
1500/1500 - 0s - loss: 45.1765 - val_loss: 46.5179
Epoch 17/1000
1500/1500 - 0s - loss: 39.8843 - val_loss: 41.3721
Epoch 18/1000
1500/1500 - 0s - loss: 35.1468 - val_loss: 37.2132
Epoch 19/1000
1500/1500 - 0s - loss: 31.1755 - val_loss: 33.0697
Epoch 20/1000
1500/1500 - 0s - loss: 27.6307 - val_loss: 30.3131
Epoch 21/1000
1500/1500 - 0s - loss: 24.8457 - val_loss: 26.9474
Epoch 22/1000
1500/1500 - 0s - loss: 22.4056 - val_loss: 24.3656
Epoch 23/1000
1500/1500 - 0s - loss: 20.3071 - val_loss: 22.1642
Epoch 24/1000
1500/1500 - 0s - loss: 18.5446 - val_loss: 20.4782
Epoch 25/1000
1500/1500 - 0s - loss: 17.1571 - val_loss: 18.8670
Epoch 26/1000
1500/1500 - 0s - loss: 15.9407 - val_loss: 17.6862
Epoch 27/1000
1500/1500 - 0s - loss: 14.9866 - val_loss: 16.5275
Epoch 28/1000

1500/1500 - 0s - loss: 14.1251 - val_loss: 15.6342
Epoch 29/1000
1500/1500 - 0s - loss: 13.4655 - val_loss: 14.8625
Epoch 30/1000
1500/1500 - 0s - loss: 12.8994 - val_loss: 14.2826
Epoch 31/1000
1500/1500 - 0s - loss: 12.5566 - val_loss: 13.6121
Epoch 32/1000
1500/1500 - 0s - loss: 12.0077 - val_loss: 13.3087
Epoch 33/1000
1500/1500 - 0s - loss: 11.5357 - val_loss: 12.6593
Epoch 34/1000
1500/1500 - 0s - loss: 11.2365 - val_loss: 12.1849
Epoch 35/1000
1500/1500 - 0s - loss: 10.8074 - val_loss: 11.9388
Epoch 36/1000
1500/1500 - 0s - loss: 10.5593 - val_loss: 11.4006
Epoch 37/1000
1500/1500 - 0s - loss: 10.2093 - val_loss: 10.9751
Epoch 38/1000
1500/1500 - 0s - loss: 9.8386 - val_loss: 10.8651
Epoch 39/1000
1500/1500 - 0s - loss: 9.5938 - val_loss: 10.5728
Epoch 40/1000
1500/1500 - 0s - loss: 9.1488 - val_loss: 9.8661
Epoch 41/1000
1500/1500 - 0s - loss: 8.8920 - val_loss: 9.5228
Epoch 42/1000
1500/1500 - 0s - loss: 8.5156 - val_loss: 9.1506
Epoch 43/1000
1500/1500 - 0s - loss: 8.2628 - val_loss: 8.9486
Epoch 44/1000
1500/1500 - 0s - loss: 7.9219 - val_loss: 8.5034
Epoch 45/1000
1500/1500 - 0s - loss: 7.7077 - val_loss: 8.0760
Epoch 46/1000
1500/1500 - 0s - loss: 7.3165 - val_loss: 7.6620
Epoch 47/1000
1500/1500 - 0s - loss: 7.0259 - val_loss: 7.4933
Epoch 48/1000
1500/1500 - 0s - loss: 6.7422 - val_loss: 7.0583
Epoch 49/1000
1500/1500 - 0s - loss: 6.5163 - val_loss: 6.8024
Epoch 50/1000
1500/1500 - 0s - loss: 6.2633 - val_loss: 7.3045
Epoch 51/1000
1500/1500 - 0s - loss: 6.0029 - val_loss: 6.2712
Epoch 52/1000
1500/1500 - 0s - loss: 5.6791 - val_loss: 5.9342
Epoch 53/1000
1500/1500 - 0s - loss: 5.4798 - val_loss: 6.0110
Epoch 54/1000
1500/1500 - 0s - loss: 5.2115 - val_loss: 5.3928
Epoch 55/1000
1500/1500 - 0s - loss: 4.9592 - val_loss: 5.2215
Epoch 56/1000

1500/1500 - 0s - loss: 4.7189 - val_loss: 5.0103
Epoch 57/1000
1500/1500 - 0s - loss: 4.4683 - val_loss: 4.7098
Epoch 58/1000
1500/1500 - 0s - loss: 4.2650 - val_loss: 4.5259
Epoch 59/1000
1500/1500 - 0s - loss: 4.0953 - val_loss: 4.4263
Epoch 60/1000
1500/1500 - 0s - loss: 3.8027 - val_loss: 4.1103
Epoch 61/1000
1500/1500 - 0s - loss: 3.5759 - val_loss: 3.7770
Epoch 62/1000
1500/1500 - 0s - loss: 3.3755 - val_loss: 3.5737
Epoch 63/1000
1500/1500 - 0s - loss: 3.1781 - val_loss: 3.4833
Epoch 64/1000
1500/1500 - 0s - loss: 3.0001 - val_loss: 3.2246
Epoch 65/1000
1500/1500 - 0s - loss: 2.7691 - val_loss: 3.1021
Epoch 66/1000
1500/1500 - 0s - loss: 2.6227 - val_loss: 2.8215
Epoch 67/1000
1500/1500 - 0s - loss: 2.4682 - val_loss: 2.7528
Epoch 68/1000
1500/1500 - 0s - loss: 2.3243 - val_loss: 2.5394
Epoch 69/1000
1500/1500 - 0s - loss: 2.1664 - val_loss: 2.3886
Epoch 70/1000
1500/1500 - 0s - loss: 2.0377 - val_loss: 2.2536
Epoch 71/1000
1500/1500 - 0s - loss: 1.8845 - val_loss: 2.2354
Epoch 72/1000
1500/1500 - 0s - loss: 1.7931 - val_loss: 2.0831
Epoch 73/1000
1500/1500 - 0s - loss: 1.6889 - val_loss: 1.8866
Epoch 74/1000
1500/1500 - 0s - loss: 1.5820 - val_loss: 1.7964
Epoch 75/1000
1500/1500 - 0s - loss: 1.5085 - val_loss: 1.7138
Epoch 76/1000
1500/1500 - 0s - loss: 1.4159 - val_loss: 1.6468
Epoch 77/1000
1500/1500 - 0s - loss: 1.3606 - val_loss: 1.5906
Epoch 78/1000
1500/1500 - 0s - loss: 1.2652 - val_loss: 1.5063
Epoch 79/1000
1500/1500 - 0s - loss: 1.1937 - val_loss: 1.4506
Epoch 80/1000
1500/1500 - 0s - loss: 1.1180 - val_loss: 1.4817
Epoch 81/1000
1500/1500 - 0s - loss: 1.1412 - val_loss: 1.2800
Epoch 82/1000
1500/1500 - 0s - loss: 1.0385 - val_loss: 1.2412
Epoch 83/1000
1500/1500 - 0s - loss: 0.9846 - val_loss: 1.1891
Epoch 84/1000

1500/1500 - 0s - loss: 0.9937 - val_loss: 1.1322
Epoch 85/1000
1500/1500 - 0s - loss: 0.8915 - val_loss: 1.0847
Epoch 86/1000
1500/1500 - 0s - loss: 0.8562 - val_loss: 1.1110
Epoch 87/1000
1500/1500 - 0s - loss: 0.8468 - val_loss: 1.0686
Epoch 88/1000
1500/1500 - 0s - loss: 0.7947 - val_loss: 0.9805
Epoch 89/1000
1500/1500 - 0s - loss: 0.7807 - val_loss: 0.9463
Epoch 90/1000
1500/1500 - 0s - loss: 0.7502 - val_loss: 0.9965
Epoch 91/1000
1500/1500 - 0s - loss: 0.7529 - val_loss: 0.9532
Epoch 92/1000
1500/1500 - 0s - loss: 0.6857 - val_loss: 0.8712
Epoch 93/1000
1500/1500 - 0s - loss: 0.6717 - val_loss: 0.8498
Epoch 94/1000
1500/1500 - 0s - loss: 0.6869 - val_loss: 0.8518
Epoch 95/1000
1500/1500 - 0s - loss: 0.6626 - val_loss: 0.8275
Epoch 96/1000
1500/1500 - 0s - loss: 0.6308 - val_loss: 0.7850
Epoch 97/1000
1500/1500 - 0s - loss: 0.6056 - val_loss: 0.7708
Epoch 98/1000
1500/1500 - 0s - loss: 0.5991 - val_loss: 0.7643
Epoch 99/1000
1500/1500 - 0s - loss: 0.6102 - val_loss: 0.8104
Epoch 100/1000
1500/1500 - 0s - loss: 0.5647 - val_loss: 0.7227
Epoch 101/1000
1500/1500 - 0s - loss: 0.5474 - val_loss: 0.7107
Epoch 102/1000
1500/1500 - 0s - loss: 0.5395 - val_loss: 0.6847
Epoch 103/1000
1500/1500 - 0s - loss: 0.5350 - val_loss: 0.7383
Epoch 104/1000
1500/1500 - 0s - loss: 0.5551 - val_loss: 0.6698
Epoch 105/1000
1500/1500 - 0s - loss: 0.5032 - val_loss: 0.6520
Epoch 106/1000
1500/1500 - 0s - loss: 0.5418 - val_loss: 0.7518
Epoch 107/1000
1500/1500 - 0s - loss: 0.4949 - val_loss: 0.6307
Epoch 108/1000
1500/1500 - 0s - loss: 0.5166 - val_loss: 0.6741
Epoch 109/1000
1500/1500 - 0s - loss: 0.4992 - val_loss: 0.6195
Epoch 110/1000
1500/1500 - 0s - loss: 0.4610 - val_loss: 0.6268
Epoch 111/1000
1500/1500 - 0s - loss: 0.4554 - val_loss: 0.5956
Epoch 112/1000

```
1500/1500 - 0s - loss: 0.4704 - val_loss: 0.5977
Epoch 113/1000
1500/1500 - 0s - loss: 0.4687 - val_loss: 0.5736
Epoch 114/1000
1500/1500 - 0s - loss: 0.4497 - val_loss: 0.5817
Epoch 115/1000
1500/1500 - 0s - loss: 0.4326 - val_loss: 0.5833
Epoch 116/1000
1500/1500 - 0s - loss: 0.4181 - val_loss: 0.5738
Epoch 117/1000
1500/1500 - 0s - loss: 0.4252 - val_loss: 0.5688
Epoch 118/1000
1500/1500 - 0s - loss: 0.4675 - val_loss: 0.5680
Epoch 119/1000
1500/1500 - 0s - loss: 0.4328 - val_loss: 0.5463
Epoch 120/1000
1500/1500 - 0s - loss: 0.4091 - val_loss: 0.5912
Epoch 121/1000
1500/1500 - 0s - loss: 0.4047 - val_loss: 0.5459
Epoch 122/1000
1500/1500 - 0s - loss: 0.4456 - val_loss: 0.5509
Epoch 123/1000
1500/1500 - 0s - loss: 0.4081 - val_loss: 0.5540
Epoch 124/1000
Restoring model weights from the end of the best epoch.
1500/1500 - 0s - loss: 0.4353 - val_loss: 0.5538
Epoch 00124: early stopping
```

Out[3]: <tensorflow.python.keras.callbacks.History at 0x1a40e6b0d0>

Mean Square Error

The mean square error (MSE) is the sum of the squared differences between the prediction (\hat{y}) and the expected (y). MSE values are not of a particular unit. If an MSE value has decreased for a model, that is good. However, beyond this, there is not much more you can determine. We seek to achieve low MSE values. The following equation demonstrates how to calculate MSE.

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2$$

The following code calculates the MSE on the predictions from the neural network.

```
In [4]: from sklearn import metrics

# Predict
pred = model.predict(x_test)

# Measure MSE error.
score = metrics.mean_squared_error(pred,y_test)
print("Final score (MSE): {}".format(score))
```

```
Final score (MSE): 0.5463447829677607
```

Root Mean Square Error

The root mean square (RMSE) is essentially the square root of the MSE. Because of this, the RMSE error is in the same units as the training data outcome. We desire Low RMSE values. The following equation calculates RMSE.

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2}$$

```
In [5]: import numpy as np
```

```
# Measure RMSE error. RMSE is common for regression.
score = np.sqrt(metrics.mean_squared_error(pred,y_test))
print("Final score (RMSE): {}".format(score))
```

```
Final score (RMSE): 0.7391513938076291
```

Lift Chart

We often visualize the results of regression with a lift chart. To generate a lift chart, perform the following activities:

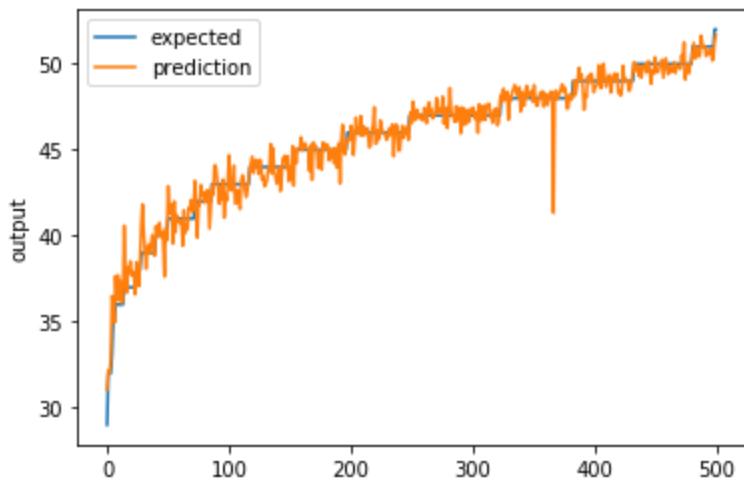
- Sort the data by expected output and plot these values.
- For every point on the x-axis, plot that same data point's predicted value in another color.
- The x-axis is just 0 to 100% of the dataset. The expected always starts low and ends high.
- The y-axis is ranged according to the values predicted.

You can interpret the lift chart as follows:

- The expected and predict lines should be close. Notice where one is above the other.
- The below chart is the most accurate for lower ages.

```
In [7]: # Regression chart.
def chart_regression(pred, y, sort=True):
    t = pd.DataFrame({'pred': pred, 'y': y.flatten()})
    if sort:
        t.sort_values(by=['y'], inplace=True)
    plt.plot(t['y'].tolist(), label='expected')
    plt.plot(t['pred'].tolist(), label='prediction')
    plt.ylabel('output')
    plt.legend()
    plt.show()
```

```
# Plot the chart  
chart_regression(pred.flatten(),y_test)
```



In []:

T81-558: Applications of Deep Neural Networks

Module 4: Training for Tabular Data

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

Module 4 Material

- Part 4.1: Encoding a Feature Vector for Keras Deep Learning [[Video](#)] [[Notebook](#)]
- Part 4.2: Keras Multiclass Classification for Deep Neural Networks with ROC and AUC [[Video](#)] [[Notebook](#)]
- Part 4.3: Keras Regression for Deep Neural Networks with RMSE [[Video](#)] [[Notebook](#)]
- **Part 4.4: Backpropagation, Nesterov Momentum, and ADAM Neural Network Training** [[Video](#)] [[Notebook](#)]
- Part 4.5: Neural Network RMSE and Log Loss Error Calculation from Scratch [[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 4.4: Training Neural Networks

Backpropagation [Cite:rumelhart1986learning] is one of the most common methods for training a neural network. Rumelhart, Hinton, & Williams introduced backpropagation, and it remains popular today. Programmers frequently train deep neural networks with backpropagation because it scales really well when run on graphical processing units (GPUs). To understand this algorithm for neural networks, we must examine how to train it as well as how it processes a pattern.

Researchers have extended classic backpropagation and modified to give rise to many different training algorithms. This section will discuss the most commonly used training algorithms for neural networks. We begin with classic backpropagation and end the chapter with stochastic gradient descent (SGD).

Backpropagation is the primary means of determining a neural network's weights during training. Backpropagation works by calculating a weight change amount (v_t) for every weight(θ , theta) in the neural network. This value is subtracted from every weight by the following equation:

$$\theta_t = \theta_{t-1} - v_t$$

We repeat this process for every iteration(t). The training algorithm determines how we calculate the weight change. Classic backpropagation calculates a gradient (∇ , nabla) for every weight in the neural network for the neural network's error function (J). We scale the gradient by a learning rate (η , eta).

$$v_t = \eta \nabla_{\theta_{t-1}} J(\theta_{t-1})$$

The learning rate is an important concept for backpropagation training. Setting the learning rate can be complex:

- Too low a learning rate will usually converge to a reasonable solution; however, the process will be prolonged.
- Too high of a learning rate will either fail outright or converge to a higher error than a better learning rate.

Common values for learning rate are: 0.1, 0.01, 0.001, etc.

Backpropagation is a gradient descent type, and many texts will use these two terms interchangeably. Gradient descent refers to calculating a gradient on each weight in the neural network for each training element. Because the neural network will not output the expected value for a training element, the gradient of each weight will indicate how to modify each weight to achieve the expected output. If the neural network did output exactly what was expected, the gradient for each weight would be 0, indicating that no change to the weight is necessary.

The gradient is the derivative of the error function at the weight's current value. The error function measures the distance of the neural network's output from the

expected output. We can use gradient descent, a process in which each weight's gradient value can reach even lower values of the error function.

The gradient is the partial derivative of each weight in the neural network concerning the error function. Each weight has a gradient that is the slope of the error function. Weight is a connection between two neurons. Calculating the gradient of the error function allows the training method to determine whether it should increase or decrease the weight. In turn, this determination will decrease the error of the neural network. The error is the difference between the expected output and actual output of the neural network. Many different training methods called propagation-training algorithms utilize gradients. In all of them, the sign of the gradient tells the neural network the following information:

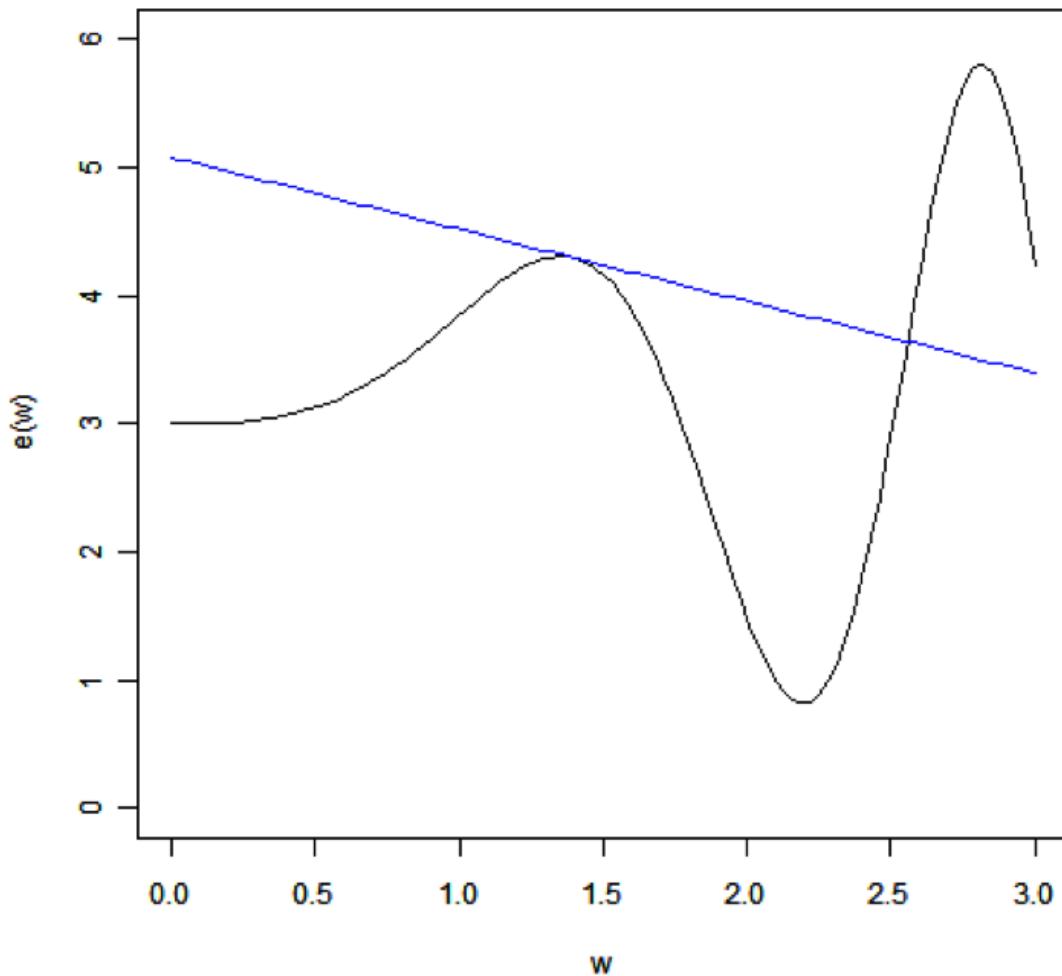
- Zero gradient - The weight does not contribute to the neural network's error.
- Negative gradient - The algorithm should increase the weight to lower error.
- Positive gradient - The algorithm should decrease the weight to lower error.

Because many algorithms depend on gradient calculation, we will begin with an analysis of this process. First of all, let's examine the gradient. Essentially, training is a search for the set of weights that will cause the neural network to have the lowest error for a training set. If we had infinite computation resources, we would try every possible combination of weights to determine the one that provided the lowest error during the training.

Because we do not have unlimited computing resources, we have to use some shortcuts to prevent the need to examine every possible weight combination. These training methods utilize clever techniques to avoid performing a brute-force search of all weight values. This type of exhaustive search would be impossible because even small networks have an infinite number of weight combinations.

Consider a chart that shows the error of a neural network for each possible weight. Figure 4.DRV is a graph that demonstrates the error for a single weight:

Figure 4.DRV: Derivative



Looking at this chart, you can easily see that the optimal weight is where the line has the lowest y-value. The problem is that we see only the error for the current value of the weight; we do not see the entire graph because that process would require an exhaustive search. However, we can determine the slope of the error curve at a particular weight. In the above chart, we see the slope of the error curve at 1.5. The straight line barely touches the error curve at 1.5 gives the slope. In this case, the slope, or gradient, is -0.5622. The negative slope indicates that an increase in the weight will lower the error. The gradient is the instantaneous slope of the error function at the specified weight. The derivative of the error curve at that point gives the gradient. This line tells us the steepness of the error function at the given weight.

Derivatives are one of the most fundamental concepts in calculus. For this book, you need to understand that a derivative provides the slope of a function at a specific point. A training technique and this slope can give you the information to

adjust the weight for a lower error. Using our working definition of the gradient, we will show how to calculate it.

Momentum Backpropagation

Momentum adds another term to the calculation of v_t :

$$v_t = \eta \nabla_{\theta_{t-1}} J(\theta_{t-1}) + \lambda v_{t-1}$$

Like the learning rate, momentum adds another training parameter that scales the effect of momentum. Momentum backpropagation has two training parameters: learning rate (η , eta) and momentum (λ , lambda). Momentum adds the scaled value of the previous weight change amount (v_{t-1}) to the current weight change amount(v_t).

This technique has the effect of adding additional force behind the direction a weight is moving. Figure 4.MTM shows how this might allow the weight to escape local minima.

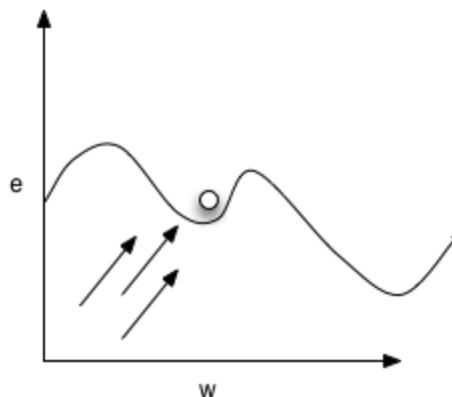


Figure 4.MTM: Momentum

A typical value for momentum is 0.9.

Batch and Online Backpropagation

How often should the weights of a neural network be updated? We can calculate gradients for a training set element. These gradients can also be summed together into batches, and the weights updated once per batch.

- **Online Training** - Update the weights based on gradients calculated from a single training set element.
- **Batch Training** - Update the weights based on the sum of the gradients over all training set elements.
- **Batch Size** - Update the weights based on the sum of some batch size of training set elements.

- **Mini-Batch Training** - The same as batch size, but with minimal batch size. Mini-batches are very popular, often in the 32-64 element range.

Because the batch size is smaller than the full training set size, it may take several batches to make it completely through the training set.

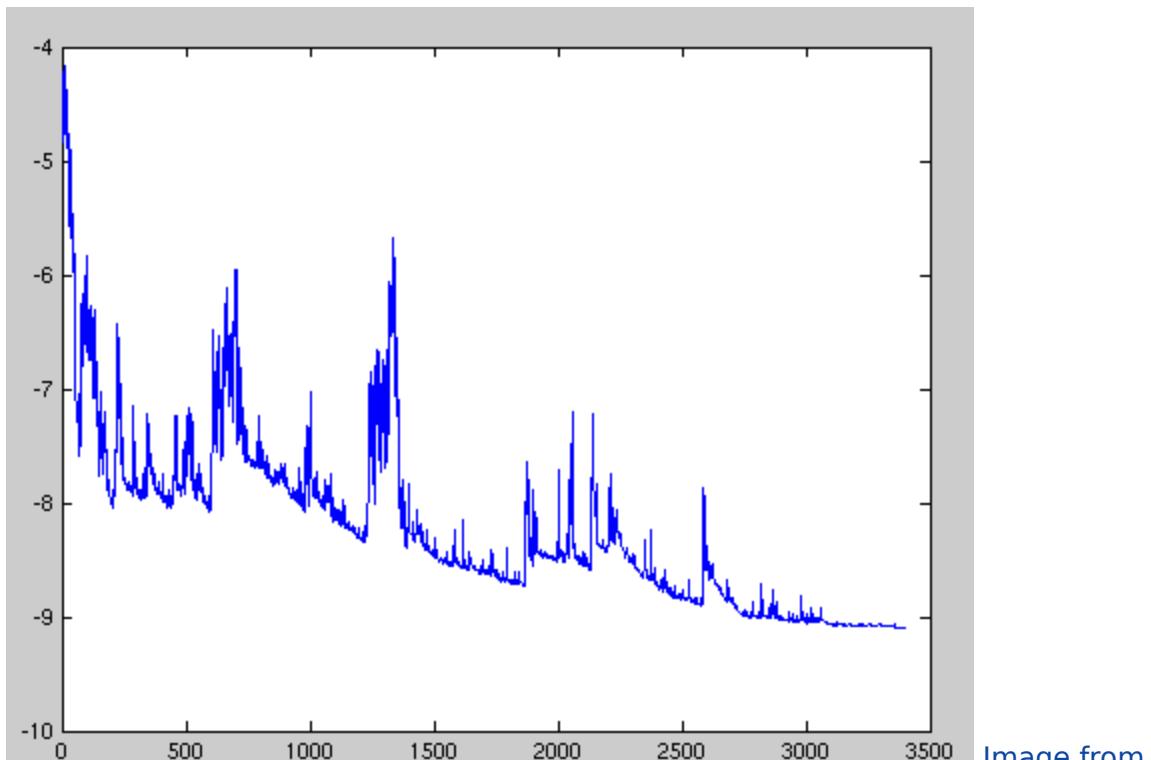
- **Step/Iteration** - The number of processed batches.
- **Epoch** - The number of times the algorithm processed the complete training set.

Stochastic Gradient Descent

Stochastic gradient descent (SGD) is currently one of the most popular neural network training algorithms. It works very similarly to Batch/Mini-Batch training, except that the batches are made up of a random set of training elements.

This technique leads to a very irregular convergence in error during training, as shown in Figure 4.SGD.

Figure 4.SGD: SGD Error



[Wikipedia](#)

Image from

Because the neural network is trained on a random sample of the complete training set each time, the error does not make a smooth transition downward. However, the error usually does go down.

Advantages to SGD include:

- Computationally efficient. Each training step can be relatively fast, even with a huge training set.
- Decreases overfitting by focusing on only a portion of the training set each step.

Other Techniques

One problem with simple backpropagation training algorithms is that they are susceptible to learning rate and momentum. This technique is difficult because:

- Learning rate must be adjusted to a small enough level to train an accurate neural network.
- Momentum must be large enough to overcome local minima yet small enough not to destabilize the training.
- A single learning rate/momentum is often not good enough for the entire training process. It is often helpful to automatically decrease the learning rate as the training progresses.
- All weights share a single learning rate/momentum.

Other training techniques:

- **Resilient Propagation** - Use only the magnitude of the gradient and allow each neuron to learn at its rate. There is no need for learning rate/momentum; however, it only works in full batch mode.
- **Nesterov accelerated gradient** - Helps mitigate the risk of choosing a bad mini-batch.
- **Adagrad** - Allows an automatically decaying per-weight learning rate and momentum concept.
- **Adadelta** - Extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate.
- **Non-Gradient Methods** - Non-gradient methods can *sometimes* be useful, though rarely outperform gradient-based backpropagation methods. These include: [simulated annealing](#), [genetic algorithms](#), [particle swarm optimization](#), [Nelder Mead](#), and [many more](#).

ADAM Update

ADAM is the first training algorithm you should try. It is very effective. Kingma and Ba (2014) introduced the Adam update rule that derives its name from the adaptive moment estimates. [\[Cite:kingma2014adam\]](#) Adam estimates the first (mean) and second (variance) moments to determine the weight corrections. Adam begins with an exponentially decaying average of past gradients (m):

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

This average accomplishes a similar goal as classic momentum update; however, its value is calculated automatically based on the current gradient (g_t). The update rule then calculates the second moment (v_t):

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

The values m_t and v_t are estimates of the gradients' first moment (the mean) and the second moment (the uncentered variance). However, they will be strongly biased towards zero in the initial training cycles. The first moment's bias is corrected as follows.

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

Similarly, the second moment is also corrected:

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

These bias-corrected first and second moment estimates are applied to the ultimate Adam update rule, as follows:

$$\theta_t = \theta_{t-1} - \frac{\alpha \cdot \hat{m}_t}{\sqrt{\hat{v}_t} + \eta} \hat{m}_t$$

Adam is very tolerant to initial learning rate (α) and other training parameters. Kingma and Ba (2014) propose default values of 0.9 for β_1 , 0.999 for β_2 , and 10-8 for η .

Methods Compared

The following image shows how each of these algorithms train. It is animated, so it is not displayed in the printed book, but can be accessed from here: <https://bit.ly/3kykkbn>.

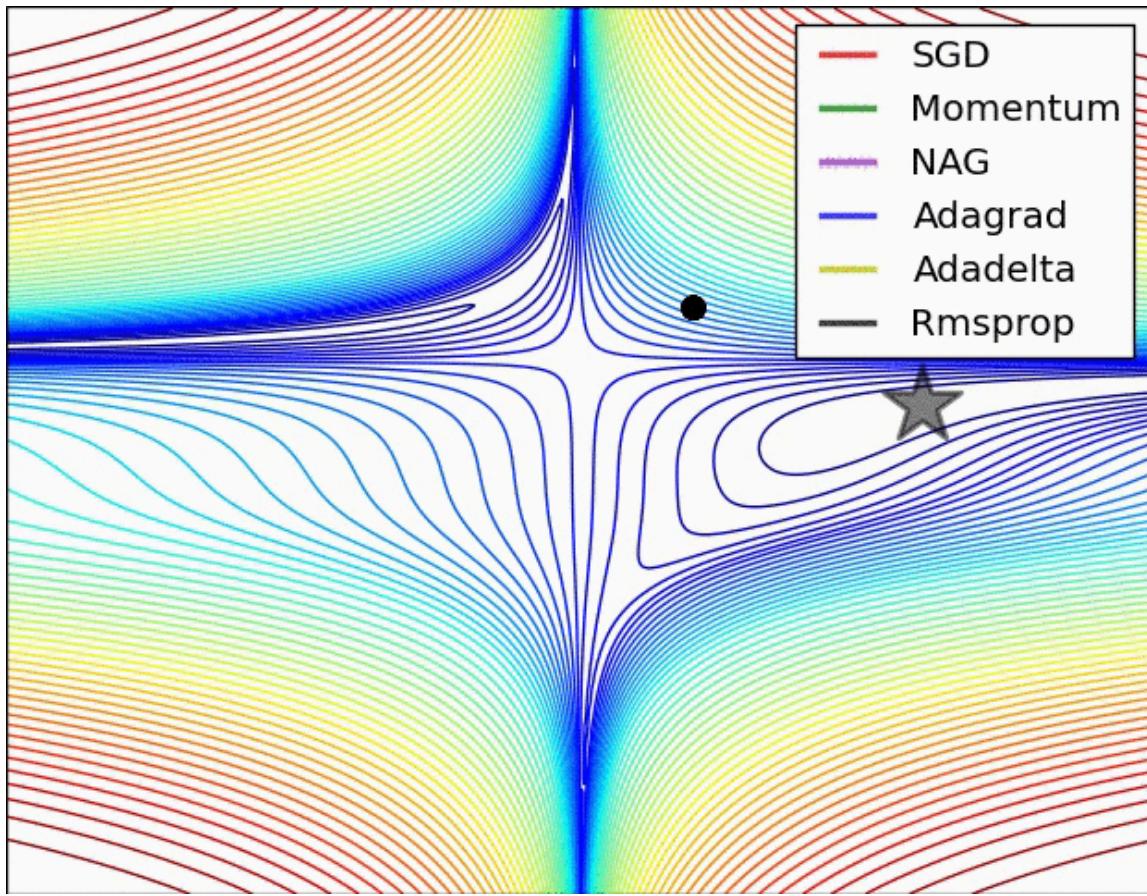


Image credits: [Alec Radford](#)

Specifying the Update Rule in Keras

TensorFlow allows the update rule to be set to one of:

- Adagrad
- **Adam**
- Ftrl
- Momentum
- RMSProp
- **SGD**

```
In [2]: %matplotlib inline

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping
from scipy.stats import zscore
from sklearn.model_selection import train_test_split
import pandas as pd
import matplotlib.pyplot as plt

# Regression chart.
def chart_regression(pred, y, sort=True):
```

```

t = pd.DataFrame({'pred': pred, 'y': y.flatten()})
if sort:
    t.sort_values(by=['y'], inplace=True)
plt.plot(t['y'].tolist(), label='expected')
plt.plot(t['pred'].tolist(), label='prediction')
plt.ylabel('output')
plt.legend()
plt.show()

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

# Generate dummies for job
df = pd.concat([df,pd.get_dummies(df['job'],prefix="job")],axis=1)
df.drop('job', axis=1, inplace=True)

# Generate dummies for area
df = pd.concat([df,pd.get_dummies(df['area'],prefix="area")],axis=1)
df.drop('area', axis=1, inplace=True)

# Generate dummies for product
df = pd.concat([df,pd.get_dummies(df['product'],prefix="product")],axis=1)
df.drop('product', axis=1, inplace=True)

# Missing values for income
med = df['income'].median()
df['income'] = df['income'].fillna(med)

# Standardize ranges
df['income'] = zscore(df['income'])
df['aspect'] = zscore(df['aspect'])
df['save_rate'] = zscore(df['save_rate'])
df['subscriptions'] = zscore(df['subscriptions'])

# Convert to numpy - Classification
x_columns = df.columns.drop('age').drop('id')
x = df[x_columns].values
y = df['age'].values

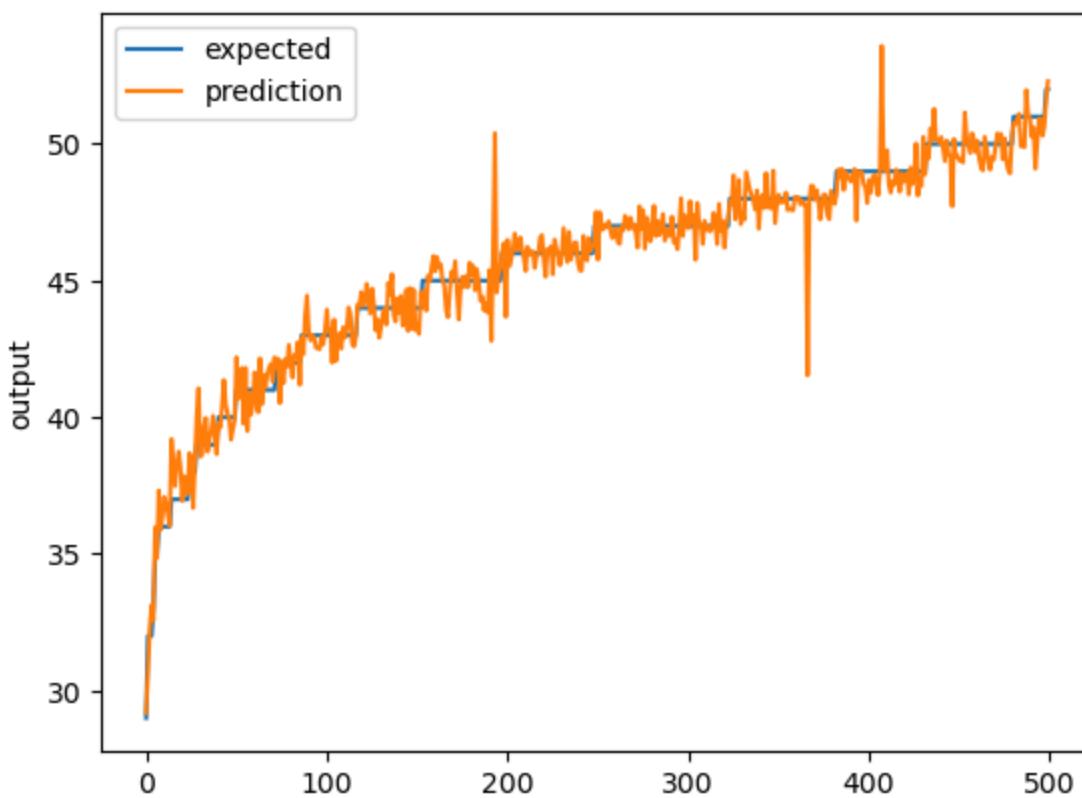
# Create train/test
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)

# 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') # Modify here
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=0,epochs=1000)

```

```
# Plot the chart
pred = model.predict(x_test)
chart_regression(pred.flatten(),y_test)
```

```
2024-02-14 00:07:14.691344: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations: SSE4.1 SSE4.2 AVX AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2024-02-14 00:07:16.219137: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations: SSE4.1 SSE4.2 AVX AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2024-02-14 00:07:16.221467: I tensorflow/core/common_runtime/process_util.c:146] Creating new thread pool with default inter op setting: 2. Tune using inter_op_parallelism_threads for best performance.
Restoring model weights from the end of the best epoch: 155.
Epoch 160: early stopping
16/16 [=====] - 0s 2ms/step
```



In []:



T81-558: Applications of Deep Neural Networks

Module 4: Training for Tabular Data

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

Module 4 Material

- Part 4.1: Encoding a Feature Vector for Keras Deep Learning [[Video](#)] [[Notebook](#)]
- Part 4.2: Keras Multiclass Classification for Deep Neural Networks with ROC and AUC [[Video](#)] [[Notebook](#)]
- Part 4.3: Keras Regression for Deep Neural Networks with RMSE [[Video](#)] [[Notebook](#)]
- Part 4.4: Backpropagation, Nesterov Momentum, and ADAM Neural Network Training [[Video](#)] [[Notebook](#)]
- **Part 4.5: Neural Network RMSE and Log Loss Error Calculation from Scratch** [[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 4.5: Error Calculation from Scratch

We will now look at how to calculate RMSE and logloss by hand. RMSE is typically used for regression. We begin by calculating RMSE with libraries.

```
In [2]: from sklearn import metrics
import numpy as np

predicted = [1.1,1.9,3.4,4.2,4.3]
expected = [1,2,3,4,5]

score_mse = metrics.mean_squared_error(predicted,expected)
score_rmse = np.sqrt(score_mse)
print("Score (MSE): {}".format(score_mse))
print("Score (RMSE): {}".format(score_rmse))
```

```
Score (MSE): 0.14200000000000007
Score (RMSE): 0.37682887362833556
```

We can also calculate without libraries.

```
In [3]: score_mse = ((predicted[0]-expected[0])**2 + (predicted[1]-expected[1])**2
+ (predicted[2]-expected[2])**2 + (predicted[3]-expected[3])**2
+ (predicted[4]-expected[4])**2)/len(predicted)
score_rmse = np.sqrt(score_mse)

print("Score (MSE): {}".format(score_mse))
print("Score (RMSE): {}".format(score_rmse))
```

```
Score (MSE): 0.14200000000000007
Score (RMSE): 0.37682887362833556
```

Classification

We will now look at how to calculate a logloss by hand. For this, we look at a binary prediction. The predicted is some number between 0-1 that indicates the probability true (1). The expected is always 0 or 1. Therefore, a prediction of 1.0 is completely correct if the expected is 1 and completely wrong if the expected is 0.

```
In [4]: from sklearn import metrics

expected = [1,1,0,0,0]
predicted = [0.9,0.99,0.1,0.05,0.06]

print(metrics.log_loss(expected,predicted))
```

```
0.06678801305495843
```

Now we attempt to calculate the same logloss manually.

```
In [5]: import numpy as np

score_logloss = (np.log(1.0-np.abs(expected[0]-predicted[0]))+\
np.log(1.0-np.abs(expected[1]-predicted[1]))+\\
```

```
np.log(1.0-np.abs(expected[2]-predicted[2]))+\nnp.log(1.0-np.abs(expected[3]-predicted[3]))+\nnp.log(1.0-np.abs(expected[4]-predicted[4])))\n*(-1/len(predicted))\n\nprint(f'Score Logloss {score_logloss}')
```

Score Logloss 0.06678801305495843

In []:



T81-558: Applications of Deep Neural Networks

Module 5: Regularization and Dropout

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

Module 5 Material

- **Part 5.1: Introduction to Regularization: Ridge and Lasso**
[\[Video\]](#) [\[Notebook\]](#)
- Part 5.2: Using K-Fold Cross Validation with Keras [\[Video\]](#) [\[Notebook\]](#)
- Part 5.3: Using L1 and L2 Regularization with Keras to Decrease Overfitting
[\[Video\]](#) [\[Notebook\]](#)
- Part 5.4: Drop Out for Keras to Decrease Overfitting [\[Video\]](#) [\[Notebook\]](#)
- Part 5.5: Benchmarking Keras Deep Learning Regularization Techniques
[\[Video\]](#) [\[Notebook\]](#)

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 5.1: Introduction to Regularization: Ridge and Lasso

Regularization is a technique that reduces overfitting, which occurs when neural networks attempt to memorize training data rather than learn from it. Humans are capable of overfitting as well. Before examining how a machine accidentally overfits, we will first explore how humans can suffer from it.

Human programmers often take certification exams to show their competence in a given programming language. To help prepare for these exams, the test makers often make practice exams available. Consider a programmer who enters a loop of taking the practice exam, studying more, and then retaking the practice exam. The programmer has memorized much of the practice exam at some point rather than learning the techniques necessary to figure out the individual questions. The programmer has now overfitted for the practice exam. When this programmer takes the real exam, his actual score will likely be lower than what he earned on the practice exam.

Although a neural network received a high score on its training data, this result does not mean that the same neural network will score high on data that was not inside the training set. A computer can overfit as well. Regularization is one of the techniques that can prevent overfitting. Several different regularization techniques exist. Most work by analyzing and potentially modifying the weights of a neural network as it trains.

L1 and L2 Regularization

L1 and L2 regularization are two standard regularization techniques that can reduce the effects of overfitting. These algorithms can either work with an objective function or as part of the backpropagation algorithm. The regularization algorithm is attached to the training algorithm by adding an objective in both cases.

These algorithms work by adding a weight penalty to the neural network training. This penalty encourages the neural network to keep the weights to small values. Both L1 and L2 calculate this penalty differently. You can add this penalty calculation to the calculated gradients for gradient-descent-based algorithms, such as backpropagation. The penalty is negatively combined with the objective score for objective-function-based training, such as simulated annealing.

We will look at linear regression to see how L1 and L2 regularization work. The following code sets up the auto-mpg data for this purpose.

```
In [2]: from sklearn.linear_model import LassoCV
import pandas as pd
import os
import numpy as np
```

```

from sklearn import metrics
from scipy.stats import zscore
from sklearn.model_selection import train_test_split

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

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

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

# Split into train/test
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=45)

```

We will use the data just loaded for several examples. The first examples in this part use several forms of linear regression. For linear regression, it is helpful to examine the model's coefficients. The following function is utilized to display these coefficients.

```

In [3]: # Simple function to evaluate the coefficients of a regression
%matplotlib inline
from IPython.display import display, HTML

def report_coef(names, coef, intercept):
    r = pd.DataFrame( { 'coef': coef, 'positive': coef>=0 }, index = names
    r = r.sort_values(by=['coef'])
    display(r)
    print(f"Intercept: {intercept}")
    r['coef'].plot(kind='barh', color=r['positive'].map(
        {True: 'b', False: 'r'}))

```

Linear Regression

Before jumping into L1/L2 regularization, we begin with linear regression. Researchers first introduced the L1/L2 form of regularization for [linear regression](#). We can also make use of L1/L2 for neural networks. To fully understand L1/L2 we will begin with how we can use them with linear regression.

The following code uses linear regression to fit the auto-mpg data set. The RMSE reported will not be as good as a neural network.

```

In [4]: import sklearn

# Create linear regression

```

```

regressor = sklearn.linear_model.LinearRegression()

# Fit/train linear regression
regressor.fit(x_train,y_train)
# Predict
pred = regressor.predict(x_test)

# Measure RMSE error. RMSE is common for regression.
score = np.sqrt(metrics.mean_squared_error(pred,y_test))
print(f"Final score (RMSE): {score}")

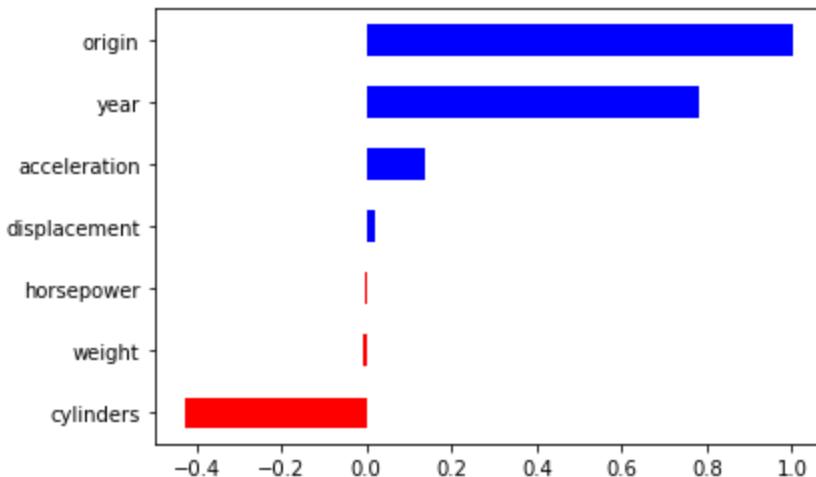
report_coef(
    names,
    regressor.coef_,
    regressor.intercept_)

```

Final score (RMSE): 3.0019345985860784

	coef	positive
cylinders	-0.427721	False
weight	-0.007255	False
horsepower	-0.005491	False
displacement	0.020166	True
acceleration	0.138575	True
year	0.783047	True
origin	1.003762	True

Intercept: -19.101231042200112



L1 (Lasso) Regularization

L1 regularization, also called LASSO (Least Absolute Shrinkage and Selection Operator) should be used to create sparsity in the neural network. In other words, the L1 algorithm will push many weight connections to near 0. When the

weight is near 0, the program drops it from the network. Dropping weighted connections will create a sparse neural network.

Feature selection is a useful byproduct of sparse neural networks. Features are the values that the training set provides to the input neurons. Once all the weights of an input neuron reach 0, the neural network training determines that the feature is unnecessary. If your data set has many unnecessary input features, L1 regularization can help the neural network detect and ignore unnecessary features.

L1 is implemented by adding the following error to the objective to minimize:

$$E_1 = \alpha \sum_w |w|$$

You should use L1 regularization to create sparsity in the neural network. In other words, the L1 algorithm will push many weight connections to near 0. When the weight is near 0, the program drops it from the network. Dropping weighted connections will create a sparse neural network.

The following code demonstrates lasso regression. Notice the effect of the coefficients compared to the previous section that used linear regression.

```
In [5]: import sklearn
from sklearn.linear_model import Lasso

# Create linear regression
regressor = Lasso(random_state=0, alpha=0.1)

# Fit/train LASSO
regressor.fit(x_train,y_train)
# Predict
pred = regressor.predict(x_test)

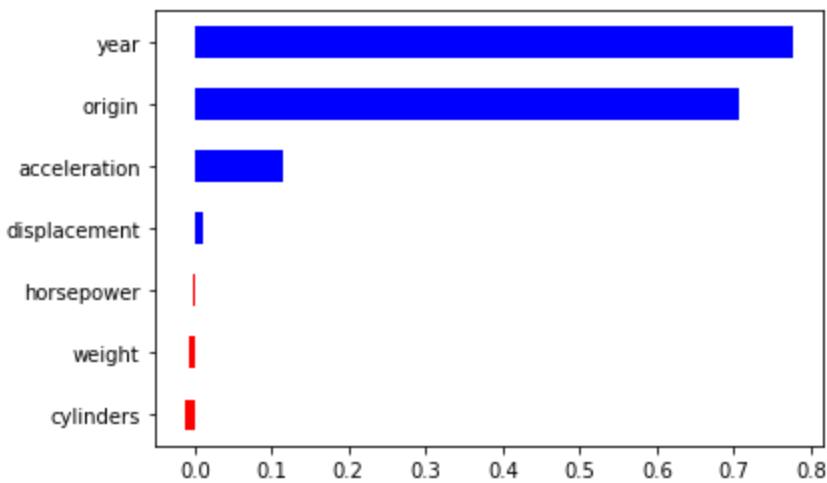
# Measure RMSE error. RMSE is common for regression.
score = np.sqrt(metrics.mean_squared_error(pred,y_test))
print(f"Final score (RMSE): {score}")

report_coef(
    names,
    regressor.coef_,
    regressor.intercept_)
```

Final score (RMSE): 3.0604021904033303

	coef	positive
cylinders	-0.012995	False
weight	-0.007328	False
horsepower	-0.002715	False
displacement	0.011601	True
acceleration	0.114391	True
origin	0.708222	True
year	0.777480	True

Intercept: -18.506677982383252



L2 (Ridge) Regularization

You should use Tikhonov/Ridge/L2 regularization when you are less concerned about creating a sparse network and are more concerned about low weight values. The lower weight values will typically lead to less overfitting.

$$E_2 = \alpha \sum_w w^2$$

Like the L1 algorithm, the α value determines how important the L2 objective is compared to the neural network's error. Typical L2 values are below 0.1 (10%). The main calculation performed by L2 is the summing of the squares of all of the weights. The algorithm will not sum bias values.

You should use L2 regularization when you are less concerned about creating a sparse network and are more concerned about low weight values. The lower weight values will typically lead to less overfitting. Generally, L2 regularization will produce better overall performance than L1. However, L1 might be useful in situations with many inputs, and you can prune some of the weaker inputs.

The following code uses L2 with linear regression (Ridge regression):

```
In [7]: import sklearn
from sklearn.linear_model import Ridge

# Create linear regression
regressor = Ridge(alpha=1)

# Fit/train Ridge
regressor.fit(x_train,y_train)
# Predict
pred = regressor.predict(x_test)

# Measure RMSE error. RMSE is common for regression.
score = np.sqrt(metrics.mean_squared_error(pred,y_test))
print("Final score (RMSE): {score}")

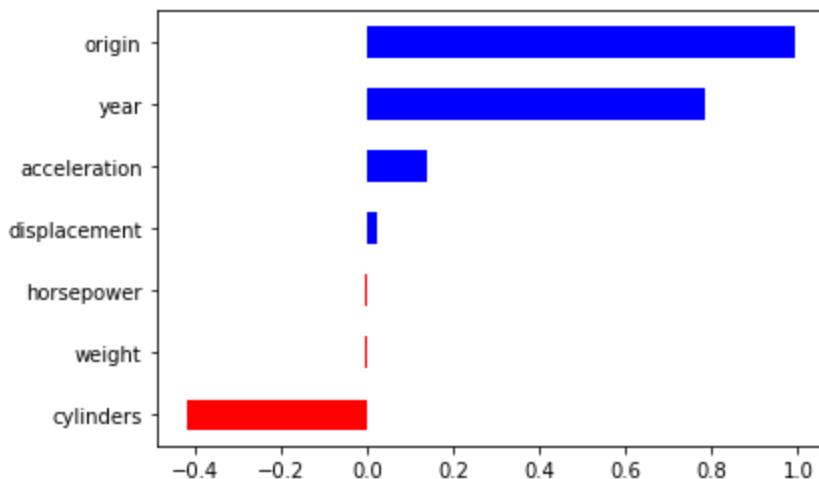
report_coef(
    names,
    regressor.coef_,
    regressor.intercept_)
```

Final score (RMSE): {score}

coef positive

cylinders	-0.421393	False
weight	-0.007257	False
horsepower	-0.005385	False
displacement	0.020006	True
acceleration	0.138470	True
year	0.782889	True
origin	0.994621	True

Intercept: -19.07980074425469



ElasticNet Regularization

The ElasticNet regression combines both L1 and L2. Both penalties are applied. The amount of L1 and L2 are governed by the parameters alpha and beta.

$$a * \text{L1} + b * \text{L2}$$

```
In [8]: import sklearn
from sklearn.linear_model import ElasticNet

# Create linear regression
regressor = ElasticNet(alpha=0.1, l1_ratio=0.1)

# Fit/train LASSO
regressor.fit(x_train,y_train)
# Predict
pred = regressor.predict(x_test)

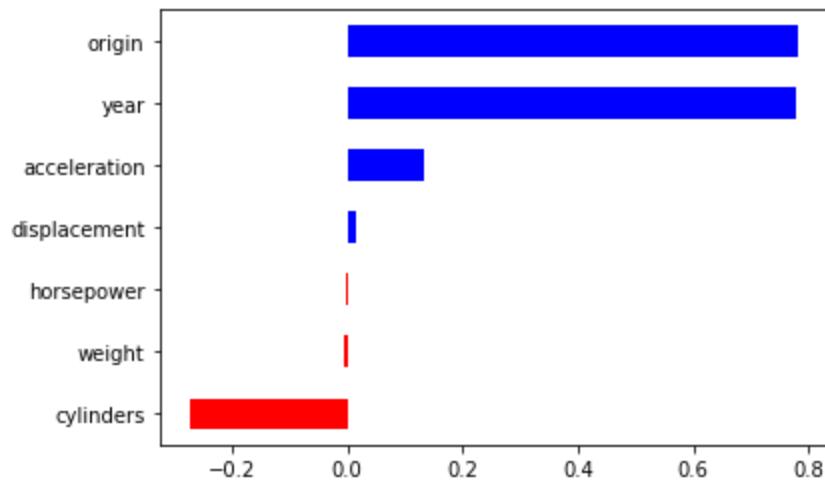
# Measure RMSE error. RMSE is common for regression.
score = np.sqrt(metrics.mean_squared_error(pred,y_test))
print(f"Final score (RMSE): {score}")

report_coef(
    names,
    regressor.coef_,
    regressor.intercept_)
```

Final score (RMSE): 3.0450899960775013

	coef	positive
cylinders	-0.274010	False
weight	-0.007303	False
horsepower	-0.003231	False
displacement	0.016194	True
acceleration	0.132348	True
year	0.777482	True
origin	0.782781	True

Intercept: -18.389355690429767



T81-558: Applications of Deep Neural Networks

Module 5: Regularization and Dropout

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

Module 5 Material

- Part 5.1: Part 5.1: Introduction to Regularization: Ridge and Lasso [\[Video\]](#) [\[Notebook\]](#)
- **Part 5.2: Using K-Fold Cross Validation with Keras** [\[Video\]](#) [\[Notebook\]](#)
- Part 5.3: Using L1 and L2 Regularization with Keras to Decrease Overfitting [\[Video\]](#) [\[Notebook\]](#)
- Part 5.4: Drop Out for Keras to Decrease Overfitting [\[Video\]](#) [\[Notebook\]](#)
- Part 5.5: Benchmarking Keras Deep Learning Regularization Techniques [\[Video\]](#) [\[Notebook\]](#)

Google CoLab Instructions

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

```
In [1]: import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

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

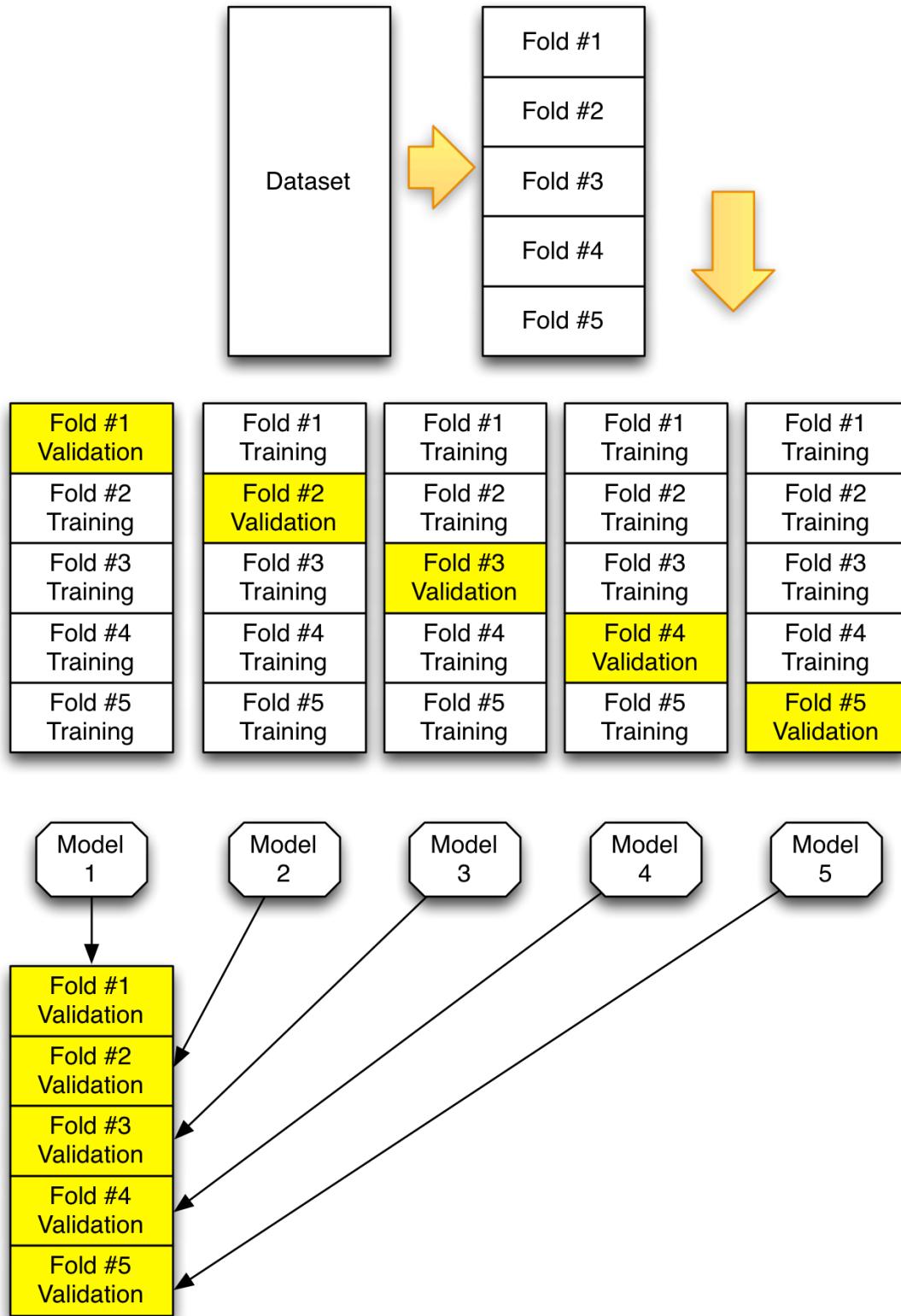
Part 5.2: Using K-Fold Cross-validation with Keras

You can use cross-validation for a variety of purposes in predictive modeling:

- Generating out-of-sample predictions from a neural network
- Estimate a good number of epochs to train a neural network for (early stopping)
- Evaluate the effectiveness of certain hyperparameters, such as activation functions, neuron counts, and layer counts

Cross-validation uses several folds and multiple models to provide each data segment a chance to serve as both the validation and training set. Figure 5.CROSS shows cross-validation.

Figure 5.CROSS: K-Fold Crossvalidation



It is important to note that each fold will have one model (neural network). To generate predictions for new data (not present in the training set), predictions from the fold models can be handled in several ways:

- Choose the model with the highest validation score as the final model.

- Preset new data to the five models (one for each fold) and average the result (this is an [ensemble](#)).
- Retrain a new model (using the same settings as the cross-validation) on the entire dataset. Train for as many epochs and with the same hidden layer structure.

Generally, I prefer the last approach and will retrain a model on the entire data set once I have selected hyper-parameters. Of course, I will always set aside a final holdout set for model validation that I do not use in any aspect of the training process.

Regression vs Classification K-Fold Cross-Validation

Regression and classification are handled somewhat differently concerning cross-validation. Regression is the simpler case where you can break up the data set into K folds with little regard for where each item lands. For regression, the data items should fall into the folds as randomly as possible. It is also important to remember that not every fold will necessarily have the same number of data items. It is not always possible for the data set to be evenly divided into K folds. For regression cross-validation, we will use the Scikit-Learn class **KFold**.

Cross-validation for classification could also use the **KFold** object; however, this technique would not ensure that the class balance remains the same in each fold as in the original. The balance of classes that a model was trained on must remain the same (or similar) to the training set. Drift in this distribution is one of the most important things to monitor after a trained model has been placed into actual use. Because of this, we want to make sure that the cross-validation itself does not introduce an unintended shift. This technique is called stratified sampling and is accomplished by using the Scikit-Learn object **StratifiedKFold** in place of **KFold** whenever you use classification. In summary, you should use the following two objects in Scikit-Learn:

- **KFold** When dealing with a regression problem.
- **StratifiedKFold** When dealing with a classification problem.

The following two sections demonstrate cross-validation with classification and regression.

Out-of-Sample Regression Predictions with K-Fold Cross-Validation

The following code trains the simple dataset using a 5-fold cross-validation. The expected performance of a neural network of the type trained here would be the score for the generated out-of-sample predictions. We begin by preparing a feature vector using the **jh-simple-dataset** to predict age. This model is set up as a regression problem.

In [2]:

```
import pandas as pd
from scipy.stats import zscore
from sklearn.model_selection import train_test_split

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

# Generate dummies for job
df = pd.concat([df,pd.get_dummies(df['job'],prefix="job")],axis=1)
df.drop('job', axis=1, inplace=True)

# Generate dummies for area
df = pd.concat([df,pd.get_dummies(df['area'],prefix="area")],axis=1)
df.drop('area', axis=1, inplace=True)

# Generate dummies for product
df = pd.concat([df,pd.get_dummies(df['product'],prefix="product")],axis=1)
df.drop('product', axis=1, inplace=True)

# Missing values for income
med = df['income'].median()
df['income'] = df['income'].fillna(med)

# Standardize ranges
df['income'] = zscore(df['income'])
df['aspect'] = zscore(df['aspect'])
df['save_rate'] = zscore(df['save_rate'])
df['subscriptions'] = zscore(df['subscriptions'])

# Convert to numpy - Classification
x_columns = df.columns.drop('age').drop('id')
x = df[x_columns].values
y = df['age'].values
```

Now that the feature vector is created a 5-fold cross-validation can be performed to generate out-of-sample predictions. We will assume 500 epochs and not use early stopping. Later we will see how we can estimate a more optimal epoch count.

In [4]:

EPOCHS=500

```
import pandas as pd
import os
import numpy as np
```

```

from sklearn import metrics
from scipy.stats import zscore
from sklearn.model_selection import KFold
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation

# Cross-Validate
kf = KFold(5, shuffle=True, random_state=42) # Use for KFold classification
oos_y = []
oos_pred = []

fold = 0
for train, test in kf.split(x):
    fold+=1
    print(f"Fold #{fold}")

    x_train = x[train]
    y_train = y[train]
    x_test = x[test]
    y_test = y[test]

    model = Sequential()
    model.add(Dense(20, input_dim=x.shape[1], activation='relu'))
    model.add(Dense(10, activation='relu'))
    model.add(Dense(1))
    model.compile(loss='mean_squared_error', optimizer='adam')

    model.fit(x_train,y_train,validation_data=(x_test,y_test),verbose=0,
              epochs=EP0CHS)

    pred = model.predict(x_test)

    oos_y.append(y_test)
    oos_pred.append(pred)

    # Measure this fold's RMSE
    score = np.sqrt(metrics.mean_squared_error(pred,y_test))
    print(f"Fold score (RMSE): {score}")

# Build the oos prediction list and calculate the error.
oos_y = np.concatenate(oos_y)
oos_pred = np.concatenate(oos_pred)
score = np.sqrt(metrics.mean_squared_error(oos_pred,oos_y))
print(f"Final, out of sample score (RMSE): {score}")

# Write the cross-validated prediction
oos_y = pd.DataFrame(oos_y)
oos_pred = pd.DataFrame(oos_pred)
oosDF = pd.concat([df, oos_y, oos_pred],axis=1)
#oosDF.to_csv(filename_write,index=False)

```

```
Fold #1
Fold score (RMSE): 0.6814299426511208
Fold #2
Fold score (RMSE): 0.45486513719487165
Fold #3
Fold score (RMSE): 0.571615041876392
Fold #4
Fold score (RMSE): 0.46416356081116916
Fold #5
Fold score (RMSE): 1.0426518491685475
Final, out of sample score (RMSE): 0.678316077597408
```

As you can see, the above code also reports the average number of epochs needed. A common technique is to then train on the entire dataset for the average number of epochs required.

Classification with Stratified K-Fold Cross-Validation

The following code trains and fits the **jh-simple-dataset** dataset with cross-validation to generate out-of-sample. It also writes the out-of-sample (predictions on the test set) results.

It is good to perform stratified k-fold cross-validation with classification data. This technique ensures that the percentages of each class remain the same across all folds. Use the **StratifiedKFold** object instead of the **KFold** object used in the regression.

```
In [5]: import pandas as pd
from scipy.stats import zscore

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

# Generate dummies for job
df = pd.concat([df,pd.get_dummies(df['job'],prefix="job")],axis=1)
df.drop('job', axis=1, inplace=True)

# Generate dummies for area
df = pd.concat([df,pd.get_dummies(df['area'],prefix="area")],axis=1)
df.drop('area', axis=1, inplace=True)

# Missing values for income
med = df['income'].median()
df['income'] = df['income'].fillna(med)

# Standardize ranges
df['income'] = zscore(df['income'])
df['aspect'] = zscore(df['aspect'])
```

```

df['save_rate'] = zscore(df['save_rate'])
df['age'] = zscore(df['age'])
df['subscriptions'] = zscore(df['subscriptions'])

# Convert to numpy - Classification
x_columns = df.columns.drop('product').drop('id')
x = df[x_columns].values
dummies = pd.get_dummies(df['product']) # Classification
products = dummies.columns
y = dummies.values

```

We will assume 500 epochs and not use early stopping. Later we will see how we can estimate a more optimal epoch count.

In [6]:

```

import pandas as pd
import os
import numpy as np
from sklearn import metrics
from sklearn.model_selection import StratifiedKFold
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation

# np.argmax(pred, axis=1)
# Cross-validate
# Use for StratifiedKFold classification
kf = StratifiedKFold(5, shuffle=True, random_state=42)

oos_y = []
oos_pred = []
fold = 0

# Must specify y StratifiedKFold for
for train, test in kf.split(x, df['product']):
    fold+=1
    print(f"Fold #{fold}")

    x_train = x[train]
    y_train = y[train]
    x_test = x[test]
    y_test = y[test]

    model = Sequential()
    # Hidden 1
    model.add(Dense(50, input_dim=x.shape[1], activation='relu'))
    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_train,y_train,validation_data=(x_test,y_test),
              verbose=0, epochs=EPOCHS)

    pred = model.predict(x_test)

    oos_y.append(y_test)
    # raw probabilities to chosen class (highest probability)

```

```

pred = np.argmax(pred, axis=1)
oos_pred.append(pred)

# Measure this fold's accuracy
y_compare = np.argmax(y_test, axis=1) # For accuracy calculation
score = metrics.accuracy_score(y_compare, pred)
print(f"Fold score (accuracy): {score}")

# Build the oos prediction list and calculate the error.
oos_y = np.concatenate(oos_y)
oos_pred = np.concatenate(oos_pred)
oos_y_compare = np.argmax(oos_y, axis=1) # For accuracy calculation

score = metrics.accuracy_score(oos_y_compare, oos_pred)
print(f"Final score (accuracy): {score}")

# Write the cross-validated prediction
oos_y = pd.DataFrame(oos_y)
oos_pred = pd.DataFrame(oos_pred)
oosDF = pd.concat([df, oos_y, oos_pred], axis=1)
#oosDF.to_csv(filename_write, index=False)

```

```

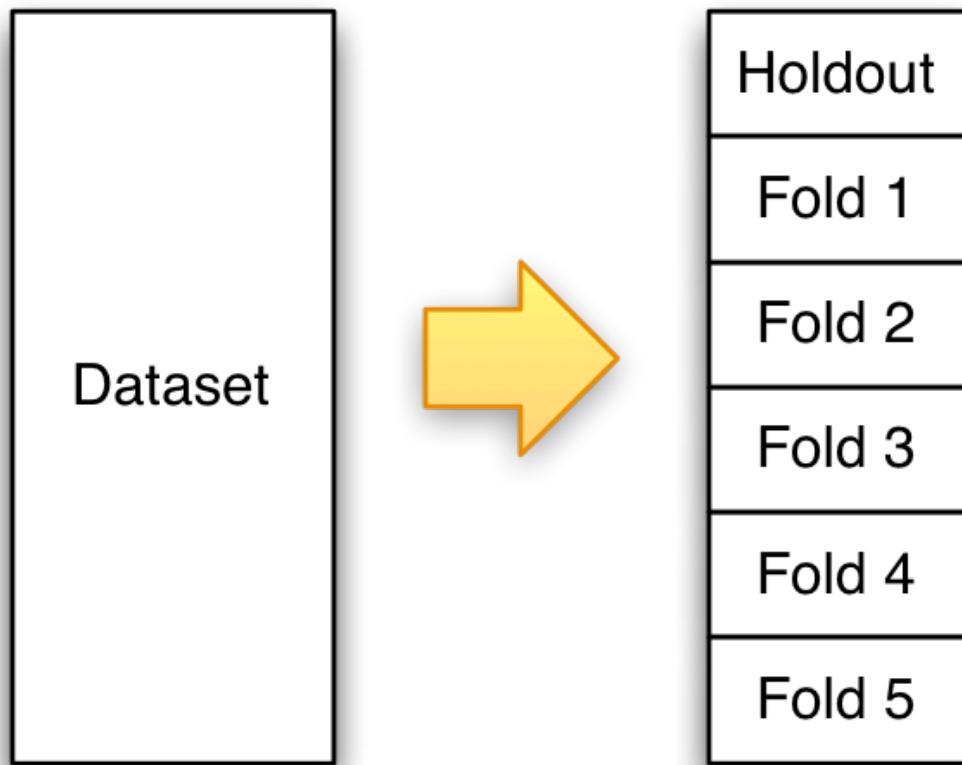
Fold #1
Fold score (accuracy): 0.6325
Fold #2
Fold score (accuracy): 0.6725
Fold #3
Fold score (accuracy): 0.6975
Fold #4
Fold score (accuracy): 0.6575
Fold #5
Fold score (accuracy): 0.675
Final score (accuracy): 0.667

```

Training with both a Cross-Validation and a Holdout Set

If you have a considerable amount of data, it is always valuable to set aside a holdout set before you cross-validate. This holdout set will be the final evaluation before using your model for its real-world use. Figure 5. HOLDOUT shows this division.

Figure 5. HOLDOUT: Cross-Validation and a Holdout Set



The following program uses a holdout set and then still cross-validates.

```
In [7]: import pandas as pd
from scipy.stats import zscore
from sklearn.model_selection import train_test_split

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

# Generate dummies for job
df = pd.concat([df,pd.get_dummies(df['job'],prefix="job")],axis=1)
df.drop('job', axis=1, inplace=True)

# Generate dummies for area
df = pd.concat([df,pd.get_dummies(df['area'],prefix="area")],axis=1)
df.drop('area', axis=1, inplace=True)

# Generate dummies for product
df = pd.concat([df,pd.get_dummies(df['product'],prefix="product")],axis=1)
df.drop('product', axis=1, inplace=True)

# Missing values for income
med = df['income'].median()
```

```

df['income'] = df['income'].fillna(med)

# Standardize ranges
df['income'] = zscore(df['income'])
df['aspect'] = zscore(df['aspect'])
df['save_rate'] = zscore(df['save_rate'])
df['subscriptions'] = zscore(df['subscriptions'])

# Convert to numpy - Classification
x_columns = df.columns.drop('age').drop('id')
x = df[x_columns].values
y = df['age'].values

```

Now that the data has been preprocessed, we are ready to build the neural network.

```

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

# Keep a 10% holdout
x_main, x_holdout, y_main, y_holdout = train_test_split(
    x, y, test_size=0.10)

# Cross-validate
kf = KFold(5)

oos_y = []
oos_pred = []
fold = 0
for train, test in kf.split(x_main):
    fold+=1
    print(f"Fold #{fold}")

    x_train = x_main[train]
    y_train = y_main[train]
    x_test = x_main[test]
    y_test = y_main[test]

    model = Sequential()
    model.add(Dense(20, input_dim=x.shape[1], activation='relu'))
    model.add(Dense(5, activation='relu'))
    model.add(Dense(1))
    model.compile(loss='mean_squared_error', optimizer='adam')

    model.fit(x_train,y_train,validation_data=(x_test,y_test),
               verbose=0,epochs=EPOCHS)

    pred = model.predict(x_test)

```

```
oos_y.append(y_test)
oos_pred.append(pred)

# Measure accuracy
score = np.sqrt(metrics.mean_squared_error(pred,y_test))
print(f"Fold score (RMSE): {score}")

# Build the oos prediction list and calculate the error.
oos_y = np.concatenate(oos_y)
oos_pred = np.concatenate(oos_pred)
score = np.sqrt(metrics.mean_squared_error(oos_pred,oos_y))
print()
print(f"Cross-validated score (RMSE): {score}")

# Write the cross-validated prediction (from the last neural network)
holdout_pred = model.predict(x_holdout)

score = np.sqrt(metrics.mean_squared_error(holdout_pred,y_holdout))
print(f"Holdout score (RMSE): {score}")
```

```
Fold #1
Fold score (RMSE): 0.544195299216696
Fold #2
Fold score (RMSE): 0.48070599342910353
Fold #3
Fold score (RMSE): 0.7034584765928998
Fold #4
Fold score (RMSE): 0.5397141785190473
Fold #5
Fold score (RMSE): 24.126205213080077
```

```
Cross-validated score (RMSE): 10.801732731207947
Holdout score (RMSE): 24.097657947297677
```

In []:



T81-558: Applications of Deep Neural Networks

Module 5: Regularization and Dropout

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

Module 5 Material

- Part 5.1: Introduction to Regularization: Ridge and Lasso [\[Video\]](#) [\[Notebook\]](#)
- Part 5.2: Using K-Fold Cross Validation with Keras [\[Video\]](#) [\[Notebook\]](#)
- **Part 5.3: Using L1 and L2 Regularization with Keras to Decrease Overfitting** [\[Video\]](#) [\[Notebook\]](#)
- Part 5.4: Drop Out for Keras to Decrease Overfitting [\[Video\]](#) [\[Notebook\]](#)
- Part 5.5: Benchmarking Keras Deep Learning Regularization Techniques [\[Video\]](#) [\[Notebook\]](#)

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

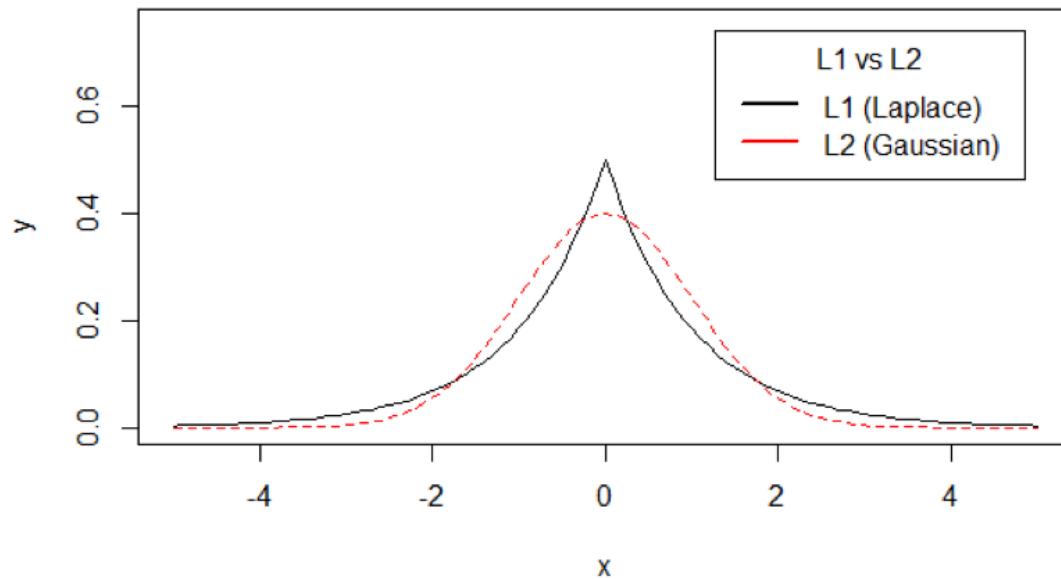
Part 5.3: L1 and L2 Regularization to Decrease Overfitting

L1 and L2 regularization are two common regularization techniques that can reduce the effects of overfitting [Cite:ng2004feature]. These algorithms can either work with an objective function or as a part of the backpropagation algorithm. In both cases, the regularization algorithm is attached to the training algorithm by adding an objective.

These algorithms work by adding a weight penalty to the neural network training. This penalty encourages the neural network to keep the weights to small values. Both L1 and L2 calculate this penalty differently. You can add this penalty calculation to the calculated gradients for gradient-descent-based algorithms, such as backpropagation. The penalty is negatively combined with the objective score for objective-function-based training, such as simulated annealing.

Both L1 and L2 work differently in that they penalize the size of the weight. L2 will force the weights into a pattern similar to a Gaussian distribution; the L1 will force the weights into a pattern similar to a Laplace distribution, as demonstrated in Figure 5.L1L2.

Figure 5.L1L2: L1 vs L2



As you can see, L1 algorithm is more tolerant of weights further from 0, whereas the L2 algorithm is less tolerant. We will highlight other important differences between L1 and L2 in the following sections. You also need to note that both L1 and L2 count their penalties based only on weights; they do not count penalties on bias values. Keras allows L1/L2 to be directly added to your network.

```
In [2]: import pandas as pd  
from scipy.stats import zscore
```

```

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

# Generate dummies for job
df = pd.concat([df,pd.get_dummies(df['job'],prefix="job")],axis=1)
df.drop('job', axis=1, inplace=True)

# Generate dummies for area
df = pd.concat([df,pd.get_dummies(df['area'],prefix="area")],axis=1)
df.drop('area', axis=1, inplace=True)

# Missing values for income
med = df['income'].median()
df['income'] = df['income'].fillna(med)

# Standardize ranges
df['income'] = zscore(df['income'])
df['aspect'] = zscore(df['aspect'])
df['save_rate'] = zscore(df['save_rate'])
df['age'] = zscore(df['age'])
df['subscriptions'] = zscore(df['subscriptions'])

# Convert to numpy - Classification
x_columns = df.columns.drop('product').drop('id')
x = df[x_columns].values
dummies = pd.get_dummies(df['product']) # Classification
products = dummies.columns
y = dummies.values

```

We now create a Keras network with L1 regression.

In [3]:

```

import pandas as pd
import os
import numpy as np
from sklearn import metrics
from sklearn.model_selection import KFold
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras import regularizers

# Cross-validate
kf = KFold(5, shuffle=True, random_state=42)

oos_y = []
oos_pred = []
fold = 0

for train, test in kf.split(x):
    fold+=1
    print(f"Fold #{fold}")

    x_train = x[train]

```

```

y_train = y[train]
x_test = x[test]
y_test = y[test]

#kernel_regularizer=regularizers.l2(0.01),

model = Sequential()
# Hidden 1
model.add(Dense(50, input_dim=x.shape[1],
                activation='relu',
                activity_regularizer=regularizers.l1(1e-4)))
# Hidden 2
model.add(Dense(25, activation='relu',
                activity_regularizer=regularizers.l1(1e-4)))
# Output
model.add(Dense(y.shape[1],activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam')

model.fit(x_train,y_train,validation_data=(x_test,y_test),
           verbose=0,epochs=500)

pred = model.predict(x_test)

oos_y.append(y_test)
# raw probabilities to chosen class (highest probability)
pred = np.argmax(pred, axis=1)
oos_pred.append(pred)

# Measure this fold's accuracy
y_compare = np.argmax(y_test, axis=1) # For accuracy calculation
score = metrics.accuracy_score(y_compare, pred)
print(f"Fold score (accuracy): {score}")

# Build the oos prediction list and calculate the error.
oos_y = np.concatenate(oos_y)
oos_pred = np.concatenate(oos_pred)
oos_y_compare = np.argmax(oos_y, axis=1) # For accuracy calculation

score = metrics.accuracy_score(oos_y_compare, oos_pred)
print(f"Final score (accuracy): {score}")

# Write the cross-validated prediction
oos_y = pd.DataFrame(oos_y)
oos_pred = pd.DataFrame(oos_pred)
oosDF = pd.concat([df, oos_y, oos_pred], axis=1 )
#oosDF.to_csv(filename_write, index=False)

```

```
Fold #1
Fold score (accuracy): 0.64
Fold #2
Fold score (accuracy): 0.6775
Fold #3
Fold score (accuracy): 0.6825
Fold #4
Fold score (accuracy): 0.6675
Fold #5
Fold score (accuracy): 0.645
Final score (accuracy): 0.6625
```

In []:



T81-558: Applications of Deep Neural Networks

Module 5: Regularization and Dropout

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

Module 5 Material

- Part 5.1: [Introduction to Regularization: Ridge and Lasso \[Video\]](#) [\[Notebook\]](#)
- Part 5.2: [Using K-Fold Cross Validation with Keras \[Video\]](#) [\[Notebook\]](#)
- Part 5.3: [Using L1 and L2 Regularization with Keras to Decrease Overfitting \[Video\]](#) [\[Notebook\]](#)
- **Part 5.4: Drop Out for Keras to Decrease Overfitting** [\[Video\]](#) [\[Notebook\]](#)
- Part 5.5: [Benchmarking Keras Deep Learning Regularization Techniques \[Video\]](#) [\[Notebook\]](#)

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

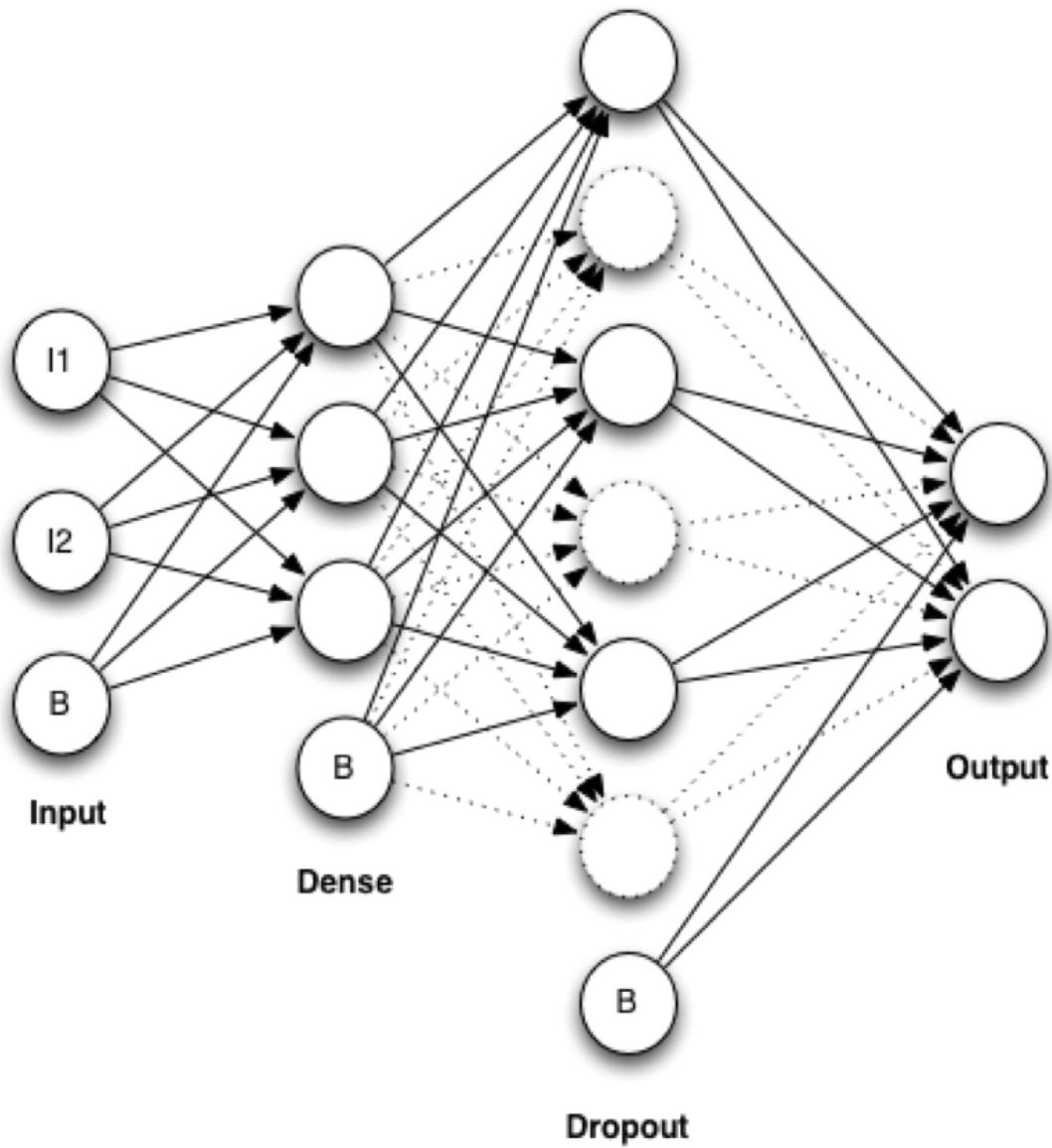
Part 5.4: Drop Out for Keras to Decrease Overfitting

Hinton, Srivastava, Krizhevsky, Sutskever, & Salakhutdinov (2012) introduced the dropout regularization algorithm. [Cite:srivastava2014dropout] Although dropout works differently than L1 and L2, it accomplishes the same goal—the prevention of overfitting. However, the algorithm does the task by actually removing neurons and connections—at least temporarily. Unlike L1 and L2, no weight penalty is added. Dropout does not directly seek to train small weights. Dropout works by causing hidden neurons of the neural network to be unavailable during part of the training. Dropping part of the neural network causes the remaining portion to be trained to still achieve a good score even without the dropped neurons. This technique decreases co-adaptation between neurons, which results in less overfitting.

Most neural network frameworks implement dropout as a separate layer. Dropout layers function like a regular, densely connected neural network layer. The only difference is that the dropout layers will periodically drop some of their neurons during training. You can use dropout layers on regular feedforward neural networks.

The program implements a dropout layer as a dense layer that can eliminate some of its neurons. Contrary to popular belief about the dropout layer, the program does not permanently remove these discarded neurons. A dropout layer does not lose any of its neurons during the training process, and it will still have the same number of neurons after training. In this way, the program only temporarily masks the neurons rather than dropping them. Figure 5.DROPOUT shows how a dropout layer might be situated with other layers.

Figure 5.DROPOUT: Dropout Regularization



The discarded neurons and their connections are shown as dashed lines. The input layer has two input neurons as well as a bias neuron. The second layer is a dense layer with three neurons and a bias neuron. The third layer is a dropout layer with six regular neurons even though the program has dropped 50% of them. While the program drops these neurons, it neither calculates nor trains them. However, the final neural network will use all of these neurons for the output. As previously mentioned, the program only temporarily discards the neurons.

The program chooses different sets of neurons from the dropout layer during subsequent training iterations. Although we chose a probability of 50% for dropout, the computer will not necessarily drop three neurons. It is as if we flipped a coin for each of the dropout candidate neurons to choose if that neuron was dropped out. You must know that the program should never drop the bias

neuron. Only the regular neurons on a dropout layer are candidates. The implementation of the training algorithm influences the process of discarding neurons. The dropout set frequently changes once per training iteration or batch. The program can also provide intervals where all neurons are present. Some neural network frameworks give additional hyper-parameters to allow you to specify exactly the rate of this interval.

Why dropout is capable of decreasing overfitting is a common question. The answer is that dropout can reduce the chance of codependency developing between two neurons. Two neurons that develop codependency will not be able to operate effectively when one is dropped out. As a result, the neural network can no longer rely on the presence of every neuron, and it trains accordingly. This characteristic decreases its ability to memorize the information presented, thereby forcing generalization.

Dropout also decreases overfitting by forcing a bootstrapping process upon the neural network. Bootstrapping is a prevalent ensemble technique. Ensembling is a technique of machine learning that combines multiple models to produce a better result than those achieved by individual models. The ensemble is a term that originates from the musical ensembles in which the final music product that the audience hears is the combination of many instruments.

Bootstrapping is one of the most simple ensemble techniques. The bootstrapping programmer simply trains several neural networks to perform precisely the same task. However, each neural network will perform differently because of some training techniques and the random numbers used in the neural network weight initialization. The difference in weights causes the performance variance. The output from this ensemble of neural networks becomes the average output of the members taken together. This process decreases overfitting through the consensus of differently trained neural networks.

Dropout works somewhat like bootstrapping. You might think of each neural network that results from a different set of neurons being dropped out as an individual member in an ensemble. As training progresses, the program creates more neural networks in this way. However, dropout does not require the same amount of processing as bootstrapping. The new neural networks created are temporary; they exist only for a training iteration. The final result is also a single neural network rather than an ensemble of neural networks to be averaged together.

The following animation shows how dropout works: [animation link](#)

```
In [2]: import pandas as pd  
from scipy.stats import zscore
```

```

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

# Generate dummies for job
df = pd.concat([df,pd.get_dummies(df['job'],prefix="job")],axis=1)
df.drop('job', axis=1, inplace=True)

# Generate dummies for area
df = pd.concat([df,pd.get_dummies(df['area'],prefix="area")],axis=1)
df.drop('area', axis=1, inplace=True)

# Missing values for income
med = df['income'].median()
df['income'] = df['income'].fillna(med)

# Standardize ranges
df['income'] = zscore(df['income'])
df['aspect'] = zscore(df['aspect'])
df['save_rate'] = zscore(df['save_rate'])
df['age'] = zscore(df['age'])
df['subscriptions'] = zscore(df['subscriptions'])

# Convert to numpy - Classification
x_columns = df.columns.drop('product').drop('id')
x = df[x_columns].values
dummies = pd.get_dummies(df['product']) # Classification
products = dummies.columns
y = dummies.values

```

Now we will see how to apply dropout to classification.

In [3]:

```

#####
# Keras with dropout for Classification
#####

import pandas as pd
import os
import numpy as np
from sklearn import metrics
from sklearn.model_selection import KFold
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation, Dropout
from tensorflow.keras import regularizers

# Cross-validate
kf = KFold(5, shuffle=True, random_state=42)

oos_y = []
oos_pred = []
fold = 0

for train, test in kf.split(x):
    fold+=1

```

```

print(f"Fold #{fold}")

x_train = x[train]
y_train = y[train]
x_test = x[test]
y_test = y[test]

#kernel_regularizer=regularizers.l2(0.01),

model = Sequential()
model.add(Dense(50, input_dim=x.shape[1], activation='relu')) # Hidden 1
model.add(Dropout(0.5))
model.add(Dense(25, activation='relu', \
               activity_regularizer=regularizers.l1(1e-4))) # Hidden 2
# Usually do not add dropout after final hidden layer
#model.add(Dropout(0.5))
model.add(Dense(y.shape[1],activation='softmax')) # Output
model.compile(loss='categorical_crossentropy', optimizer='adam')

model.fit(x_train,y_train,validation_data=(x_test,y_test),\
           verbose=0,epochs=500)

pred = model.predict(x_test)

oos_y.append(y_test)
# raw probabilities to chosen class (highest probability)
pred = np.argmax(pred, axis=1)
oos_pred.append(pred)

# Measure this fold's accuracy
y_compare = np.argmax(y_test, axis=1) # For accuracy calculation
score = metrics.accuracy_score(y_compare, pred)
print(f"Fold score (accuracy): {score}")

# Build the oos prediction list and calculate the error.
oos_y = np.concatenate(oos_y)
oos_pred = np.concatenate(oos_pred)
oos_y_compare = np.argmax(oos_y, axis=1) # For accuracy calculation

score = metrics.accuracy_score(oos_y_compare, oos_pred)
print(f"Final score (accuracy): {score}")

# Write the cross-validated prediction
oos_y = pd.DataFrame(oos_y)
oos_pred = pd.DataFrame(oos_pred)
oosDF = pd.concat([df, oos_y, oos_pred], axis=1 )
#oosDF.to_csv(filename_write, index=False)

```

```
Fold #1
Fold score (accuracy): 0.68
Fold #2
Fold score (accuracy): 0.695
Fold #3
Fold score (accuracy): 0.7425
Fold #4
Fold score (accuracy): 0.71
Fold #5
Fold score (accuracy): 0.6625
Final score (accuracy): 0.698
```

In []:

T81-558: Applications of Deep Neural Networks

Module 5: Regularization and Dropout

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

Module 5 Material

- Part 5.1: [Introduction to Regularization: Ridge and Lasso \[Video\]](#) [\[Notebook\]](#)
- Part 5.2: [Using K-Fold Cross Validation with Keras \[Video\]](#) [\[Notebook\]](#)
- Part 5.3: [Using L1 and L2 Regularization with Keras to Decrease Overfitting \[Video\]](#) [\[Notebook\]](#)
- Part 5.4: [Drop Out for Keras to Decrease Overfitting \[Video\]](#) [\[Notebook\]](#)
- **Part 5.5: Benchmarking Keras Deep Learning Regularization Techniques** [\[Video\]](#) [\[Notebook\]](#)

Google CoLab Instructions

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

```
In [5]: try:  
    %tensorflow_version 2.x  
    COLAB = True  
    print("Note: using Google CoLab")  
except:  
    print("Note: not using Google CoLab")  
    COLAB = False
```

Note: using Google CoLab

Part 5.5: Benchmarking Regularization Techniques

Quite a few hyperparameters have been introduced so far. Tweaking each of these values can have an effect on the score obtained by your neural networks. Some of the hyperparameters seen so far include:

- Number of layers in the neural network
- How many neurons in each layer
- What activation functions to use on each layer
- Dropout percent on each layer
- L1 and L2 values on each layer

To try out each of these hyperparameters you will need to run train neural networks with multiple settings for each hyperparameter. However, you may have noticed that neural networks often produce somewhat different results when trained multiple times. This is because the neural networks start with random weights. Because of this it is necessary to fit and evaluate a neural network times to ensure that one set of hyperparameters are actually better than another. Bootstrapping can be an effective means of benchmarking (comparing) two sets of hyperparameters.

Bootstrapping is similar to cross-validation. Both go through a number of cycles/folds providing validation and training sets. However, bootstrapping can have an unlimited number of cycles. Bootstrapping chooses a new train and validation split each cycle, with replacement. The fact that each cycle is chosen with replacement means that, unlike cross validation, there will often be repeated rows selected between cycles. If you run the bootstrap for enough cycles, there will be duplicate cycles.

In this part we will use bootstrapping for hyperparameter benchmarking. We will train a neural network for a specified number of splits (denoted by the SPLITS constant). For these examples we use 100. We will compare the average score at the end of the 100. By the end of the cycles the mean score will have converged somewhat. This ending score will be a much better basis of comparison than a single cross-validation. Additionally, the average number of epochs will be tracked to give an idea of a possible optimal value. Because the early stopping validation set is also used to evaluate the the neural network as well, it might be slightly inflated. This is because we are both stopping and evaluating on the same sample. However, we are using the scores only as relative measures to determine the superiority of one set of hyperparameters to another, so this slight inflation should not present too much of a problem.

Because we are benchmarking, we will display the amount of time taken for each cycle. The following function can be used to nicely format a time span.

```
In [6]: # Nicely formatted time string
def hms_string(sec_elapsed):
```

```

h = int(sec_elapsed / (60 * 60))
m = int((sec_elapsed % (60 * 60)) / 60)
s = sec_elapsed % 60
return "{}:{}{:>02}{}{:>05.2f}{}".format(h, m, s)

```

Bootstrapping for Regression

Regression bootstrapping uses the **ShuffleSplit** object to perform the splits. This technique is similar to **KFold** for cross-validation; no balancing occurs. We will attempt to predict the age column for the **jh-simple-dataset**; the following code loads this data.

```
In [7]: import pandas as pd
from scipy.stats import zscore
from sklearn.model_selection import train_test_split

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

# Generate dummies for job
df = pd.concat([df,pd.get_dummies(df['job'],prefix="job")],axis=1)
df.drop('job', axis=1, inplace=True)

# Generate dummies for area
df = pd.concat([df,pd.get_dummies(df['area'],prefix="area")],axis=1)
df.drop('area', axis=1, inplace=True)

# Generate dummies for product
df = pd.concat([df,pd.get_dummies(df['product'],prefix="product")],axis=1)
df.drop('product', axis=1, inplace=True)

# Missing values for income
med = df['income'].median()
df['income'] = df['income'].fillna(med)

# Standardize ranges
df['income'] = zscore(df['income'])
df['aspect'] = zscore(df['aspect'])
df['save_rate'] = zscore(df['save_rate'])
df['subscriptions'] = zscore(df['subscriptions'])

# Convert to numpy - Classification
x_columns = df.columns.drop('age').drop('id')
x = df[x_columns].values
y = df['age'].values
```

The following code performs the bootstrap. The architecture of the neural network can be adjusted to compare many different configurations.

```
In [8]: import pandas as pd
import os
import numpy as np
import time
import statistics
from sklearn import metrics
from sklearn.model_selection import StratifiedKFold
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras import regularizers
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.model_selection import ShuffleSplit

SPLITS = 50

# Bootstrap
boot = ShuffleSplit(n_splits=SPLITS, test_size=0.1, random_state=42)

# Track progress
mean_benchmark = []
epochs_needed = []
num = 0

# Loop through samples
for train, test in boot.split(x):
    start_time = time.time()
    num+=1

    # Split train and test
    x_train = x[train]
    y_train = y[train]
    x_test = x[test]
    y_test = y[test]

    # Construct neural network
    model = Sequential()
    model.add(Dense(20, input_dim=x_train.shape[1], activation='relu'))
    model.add(Dense(10, activation='relu'))
    model.add(Dense(1))
    model.compile(loss='mean_squared_error', optimizer='adam')

    monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3,
                           patience=5, verbose=0, mode='auto', restore_best_weights=True)

    # Train on the bootstrap sample
    model.fit(x_train,y_train,validation_data=(x_test,y_test),
               callbacks=[monitor],verbose=0,epochs=1000)
    epochs = monitor.stopped_epoch
    epochs_needed.append(epochs)

    # Predict on the out of boot (validation)
    pred = model.predict(x_test)

    # Measure this bootstrap's log loss
    score = np.sqrt(metrics.mean_squared_error(pred,y_test))
```

```
mean_benchmark.append(score)
m1 = statistics.mean(mean_benchmark)
m2 = statistics.mean(epochs_needed)
mdev = statistics.pstdev(mean_benchmark)

# Record this iteration
time_took = time.time() - start_time
print(f"# {num}: score={score:.6f}, mean score={m1:.6f},"
      f" stdev={mdev:.6f}",
      f" epochs={epochs}, mean epochs={int(m2)}",
      f" time={hms_string(time_took)}")
```

```
#1: score=0.630750, mean score=0.630750, stdev=0.000000 epochs=147, mean epochs=147 time=0:00:12.56
#2: score=1.020895, mean score=0.825823, stdev=0.195072 epochs=101, mean epochs=124 time=0:00:08.70
#3: score=0.803801, mean score=0.818482, stdev=0.159614 epochs=155, mean epochs=134 time=0:00:20.85
#4: score=0.540871, mean score=0.749079, stdev=0.183188 epochs=122, mean epochs=131 time=0:00:10.64
#5: score=0.802589, mean score=0.759781, stdev=0.165240 epochs=116, mean epochs=128 time=0:00:10.84
#6: score=0.862807, mean score=0.776952, stdev=0.155653 epochs=108, mean epochs=124 time=0:00:10.65
#7: score=0.550373, mean score=0.744584, stdev=0.164478 epochs=131, mean epochs=125 time=0:00:10.85
#8: score=0.659148, mean score=0.733904, stdev=0.156428 epochs=118, mean epochs=124 time=0:00:10.10
#9: score=0.606425, mean score=0.719740, stdev=0.152826 epochs=99, mean epochs=121 time=0:00:10.64
#10: score=1.169816, mean score=0.764748, stdev=0.198120 epochs=101, mean epochs=119 time=0:00:10.65
#11: score=0.985013, mean score=0.784772, stdev=0.199231 epochs=106, mean epochs=118 time=0:00:09.02
#12: score=0.857432, mean score=0.790827, stdev=0.191803 epochs=113, mean epochs=118 time=0:00:09.46
#13: score=0.495272, mean score=0.768092, stdev=0.200402 epochs=151, mean epochs=120 time=0:00:20.88
#14: score=1.079376, mean score=0.790326, stdev=0.209092 epochs=104, mean epochs=119 time=0:00:10.65
#15: score=0.616606, mean score=0.778745, stdev=0.206597 epochs=130, mean epochs=120 time=0:00:10.70
#16: score=0.781853, mean score=0.778939, stdev=0.200038 epochs=123, mean epochs=120 time=0:00:10.69
#17: score=0.781730, mean score=0.779103, stdev=0.194067 epochs=116, mean epochs=120 time=0:00:10.64
#18: score=0.845470, mean score=0.782790, stdev=0.189211 epochs=143, mean epochs=121 time=0:00:20.89
#19: score=0.643181, mean score=0.775442, stdev=0.186784 epochs=124, mean epochs=121 time=0:00:10.63
#20: score=1.026157, mean score=0.787978, stdev=0.190078 epochs=91, mean epochs=119 time=0:00:10.63
#21: score=0.587819, mean score=0.778447, stdev=0.190332 epochs=106, mean epochs=119 time=0:00:10.62
#22: score=0.600830, mean score=0.770373, stdev=0.189600 epochs=117, mean epochs=119 time=0:00:10.32
#23: score=0.662913, mean score=0.765701, stdev=0.186723 epochs=126, mean epochs=119 time=0:00:20.89
#24: score=0.671352, mean score=0.761770, stdev=0.183762 epochs=130, mean epochs=119 time=0:00:20.87
#25: score=0.647940, mean score=0.757217, stdev=0.181425 epochs=143, mean epochs=120 time=0:00:12.29
#26: score=0.684534, mean score=0.754421, stdev=0.178450 epochs=94, mean epochs=119 time=0:00:08.29
#27: score=0.534195, mean score=0.746265, stdev=0.179986 epochs=149, mean epochs=120 time=0:00:20.91
#28: score=0.901485, mean score=0.751808, stdev=0.179074 epochs=110, mean epochs=120 time=0:00:10.66
```

```

#29: score=0.696614, mean score=0.749905, stdev=0.176248 epochs=117, mean e
pochs=120 time=0:00:10.46
#30: score=0.656065, mean score=0.746777, stdev=0.174102 epochs=109, mean e
pochs=120 time=0:00:10.63
#31: score=0.749652, mean score=0.746870, stdev=0.171272 epochs=118, mean e
pochs=119 time=0:00:10.66
#32: score=0.508090, mean score=0.739408, stdev=0.173619 epochs=106, mean e
pochs=119 time=0:00:10.66
#33: score=0.732891, mean score=0.739210, stdev=0.170971 epochs=124, mean e
pochs=119 time=0:00:10.76
#34: score=1.089590, mean score=0.749516, stdev=0.178539 epochs=95, mean ep
ochs=118 time=0:00:08.24
#35: score=0.568665, mean score=0.744349, stdev=0.178530 epochs=115, mean e
pochs=118 time=0:00:10.64
#36: score=0.523255, mean score=0.738207, stdev=0.179744 epochs=108, mean e
pochs=118 time=0:00:09.23
#37: score=1.082163, mean score=0.747503, stdev=0.185865 epochs=87, mean ep
ochs=117 time=0:00:10.62
#38: score=0.752920, mean score=0.747646, stdev=0.183405 epochs=125, mean e
pochs=117 time=0:00:10.66
#39: score=0.587106, mean score=0.743529, stdev=0.182808 epochs=118, mean e
pochs=117 time=0:00:10.18
#40: score=0.781335, mean score=0.744474, stdev=0.180605 epochs=103, mean e
pochs=117 time=0:00:10.64
#41: score=1.209243, mean score=0.755810, stdev=0.192257 epochs=82, mean ep
ochs=116 time=0:00:07.39
#42: score=0.650733, mean score=0.753308, stdev=0.190628 epochs=141, mean e
pochs=117 time=0:00:21.19
#43: score=0.622103, mean score=0.750257, stdev=0.189434 epochs=116, mean e
pochs=117 time=0:00:09.85
#44: score=0.519172, mean score=0.745005, stdev=0.190409 epochs=135, mean e
pochs=117 time=0:00:11.94
#45: score=0.926205, mean score=0.749032, stdev=0.190167 epochs=87, mean ep
ochs=116 time=0:00:07.78
#46: score=0.604350, mean score=0.745887, stdev=0.189268 epochs=78, mean ep
ochs=116 time=0:00:10.64
#47: score=0.690874, mean score=0.744716, stdev=0.187412 epochs=136, mean e
pochs=116 time=0:00:20.86
#48: score=0.719645, mean score=0.744194, stdev=0.185484 epochs=112, mean e
pochs=116 time=0:00:09.33
#49: score=0.911419, mean score=0.747607, stdev=0.185098 epochs=124, mean e
pochs=116 time=0:00:10.66
#50: score=0.599252, mean score=0.744639, stdev=0.184411 epochs=132, mean e
pochs=116 time=0:00:20.91

```

The bootstrapping process for classification is similar, and I present it in the next section.

Bootstrapping for Classification

Regression bootstrapping uses the **StratifiedShuffleSplit** class to perform the splits. This class is similar to **StratifiedKFold** for cross-validation, as the classes are balanced so that the sampling does not affect proportions. To demonstrate

this technique, we will attempt to predict the product column for the **jh-simple-dataset**; the following code loads this data.

```
In [9]: import pandas as pd
from scipy.stats import zscore

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

# Generate dummies for job
df = pd.concat([df,pd.get_dummies(df['job'],prefix="job")],axis=1)
df.drop('job', axis=1, inplace=True)

# Generate dummies for area
df = pd.concat([df,pd.get_dummies(df['area'],prefix="area")],axis=1)
df.drop('area', axis=1, inplace=True)

# Missing values for income
med = df['income'].median()
df['income'] = df['income'].fillna(med)

# Standardize ranges
df['income'] = zscore(df['income'])
df['aspect'] = zscore(df['aspect'])
df['save_rate'] = zscore(df['save_rate'])
df['age'] = zscore(df['age'])
df['subscriptions'] = zscore(df['subscriptions'])

# Convert to numpy - Classification
x_columns = df.columns.drop('product').drop('id')
x = df[x_columns].values
dummies = pd.get_dummies(df['product']) # Classification
products = dummies.columns
y = dummies.values
```

We now run this data through a number of splits specified by the SPLITS variable. We track the average error through each of these splits.

```
In [10]: import pandas as pd
import os
import numpy as np
import time
import statistics
from sklearn import metrics
from sklearn.model_selection import StratifiedKFold
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras import regularizers
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.model_selection import StratifiedShuffleSplit

SPLITS = 50
```

```

# Bootstrap
boot = StratifiedShuffleSplit(n_splits=SPLITS, test_size=0.1,
                               random_state=42)

# Track progress
mean_benchmark = []
epochs_needed = []
num = 0

# Loop through samples
for train, test in boot.split(x, df['product']):
    start_time = time.time()
    num+=1

    # Split train and test
    x_train = x[train]
    y_train = y[train]
    x_test = x[test]
    y_test = y[test]

    # Construct 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=25, verbose=0, mode='auto', restore_best_weights=True)

    # Train on the bootstrap sample
    model.fit(x_train,y_train,validation_data=(x_test,y_test),
              callbacks=[monitor],verbose=0,epochs=1000)
    epochs = monitor.stopped_epoch
    epochs_needed.append(epochs)

    # Predict on the out of boot (validation)
    pred = model.predict(x_test)

    # Measure this bootstrap's log loss
    y_compare = np.argmax(y_test, axis=1) # For log loss calculation
    score = metrics.log_loss(y_compare, pred)
    mean_benchmark.append(score)
    m1 = statistics.mean(mean_benchmark)
    m2 = statistics.mean(epochs_needed)
    mdev = statistics.pstdev(mean_benchmark)

    # Record this iteration
    time_took = time.time() - start_time
    print(f"#{num}: score={score:.6f}, mean score={m1:.6f}, " +\
          f"stdev={mdev:.6f}, epochs={epochs}, mean epochs={int(m2)}, " +\
          f" time={hms_string(time_took)}")

```

```
#1: score=0.666342, mean score=0.666342,stdev=0.000000, epochs=66, mean epoch  
hs=66, time=0:00:06.31  
#2: score=0.645598, mean score=0.655970,stdev=0.010372, epochs=59, mean epoch  
hs=62, time=0:00:10.63  
#3: score=0.676924, mean score=0.662955,stdev=0.013011, epochs=66, mean epoch  
hs=63, time=0:00:10.64  
#4: score=0.672602, mean score=0.665366,stdev=0.012017, epochs=84, mean epoch  
hs=68, time=0:00:08.20  
#5: score=0.667274, mean score=0.665748,stdev=0.010776, epochs=73, mean epoch  
hs=69, time=0:00:10.65  
#6: score=0.706372, mean score=0.672518,stdev=0.018055, epochs=50, mean epoch  
hs=66, time=0:00:04.81  
#7: score=0.687937, mean score=0.674721,stdev=0.017565, epochs=71, mean epoch  
hs=67, time=0:00:06.89  
#8: score=0.734794, mean score=0.682230,stdev=0.025781, epochs=43, mean epoch  
hs=64, time=0:00:05.51  
#9: score=0.623972, mean score=0.675757,stdev=0.030431, epochs=65, mean epoch  
hs=64, time=0:00:10.66  
#10: score=0.650303, mean score=0.673212,stdev=0.029862, epochs=109, mean epoch  
hs=68, time=0:00:10.63  
#11: score=0.679500, mean score=0.673783,stdev=0.028529, epochs=83, mean epoch  
hs=69, time=0:00:10.63  
#12: score=0.736851, mean score=0.679039,stdev=0.032403, epochs=51, mean epoch  
hs=68, time=0:00:05.51  
#13: score=0.703048, mean score=0.680886,stdev=0.031782, epochs=92, mean epoch  
hs=70, time=0:00:08.48  
#14: score=0.733015, mean score=0.684609,stdev=0.033439, epochs=52, mean epoch  
hs=68, time=0:00:05.13  
#15: score=0.664863, mean score=0.683293,stdev=0.032679, epochs=77, mean epoch  
hs=69, time=0:00:10.62  
#16: score=0.740248, mean score=0.686853,stdev=0.034514, epochs=79, mean epoch  
hs=70, time=0:00:10.94  
#17: score=0.639677, mean score=0.684078,stdev=0.035276, epochs=82, mean epoch  
hs=70, time=0:00:10.64  
#18: score=0.648893, mean score=0.682123,stdev=0.035216, epochs=64, mean epoch  
hs=70, time=0:00:06.14  
#19: score=0.603215, mean score=0.677970,stdev=0.038541, epochs=60, mean epoch  
hs=69, time=0:00:10.72  
#20: score=0.691074, mean score=0.678625,stdev=0.037673, epochs=49, mean epoch  
hs=68, time=0:00:05.07  
#21: score=0.649008, mean score=0.677215,stdev=0.037302, epochs=54, mean epoch  
hs=68, time=0:00:05.51  
#22: score=0.745487, mean score=0.680318,stdev=0.039121, epochs=39, mean epoch  
hs=66, time=0:00:05.54  
#23: score=0.588884, mean score=0.676343,stdev=0.042563, epochs=74, mean epoch  
hs=67, time=0:00:07.11  
#24: score=0.697504, mean score=0.677224,stdev=0.041881, epochs=61, mean epoch  
hs=66, time=0:00:05.86  
#25: score=0.569334, mean score=0.672909,stdev=0.046161, epochs=64, mean epoch  
hs=66, time=0:00:10.67  
#26: score=0.632199, mean score=0.671343,stdev=0.045936, epochs=65, mean epoch  
hs=66, time=0:00:06.16  
#27: score=0.707666, mean score=0.672688,stdev=0.045597, epochs=74, mean epoch  
hs=66, time=0:00:07.34  
#28: score=0.747781, mean score=0.675370,stdev=0.046894, epochs=48, mean epoch  
hs=66, time=0:00:05.56
```

```

#29: score=0.648160, mean score=0.674432,stdev=0.046345, epochs=61, mean epo
chs=66, time=0:00:05.80
#30: score=0.695912, mean score=0.675148,stdev=0.045729, epochs=70, mean epo
chs=66, time=0:00:06.72
#31: score=0.692880, mean score=0.675720,stdev=0.045094, epochs=61, mean epo
chs=66, time=0:00:10.65
#32: score=0.675613, mean score=0.675717,stdev=0.044384, epochs=73, mean epo
chs=66, time=0:00:10.66
#33: score=0.625625, mean score=0.674199,stdev=0.044542, epochs=57, mean epo
chs=65, time=0:00:05.34
#34: score=0.571148, mean score=0.671168,stdev=0.047210, epochs=130, mean ep
ochs=67, time=0:00:20.88
#35: score=0.542365, mean score=0.667488,stdev=0.051240, epochs=75, mean epo
chs=68, time=0:00:10.67
#36: score=0.645099, mean score=0.666866,stdev=0.050657, epochs=59, mean epo
chs=67, time=0:00:05.59
#37: score=0.639249, mean score=0.666119,stdev=0.050168, epochs=78, mean epo
chs=68, time=0:00:10.68
#38: score=0.684326, mean score=0.666598,stdev=0.049589, epochs=75, mean epo
chs=68, time=0:00:10.63
#39: score=0.728835, mean score=0.668194,stdev=0.049928, epochs=79, mean epo
chs=68, time=0:00:07.78
#40: score=0.706089, mean score=0.669142,stdev=0.049654, epochs=46, mean epo
chs=67, time=0:00:04.59
#41: score=0.727177, mean score=0.670557,stdev=0.049855, epochs=68, mean epo
chs=67, time=0:00:10.69
#42: score=0.653240, mean score=0.670145,stdev=0.049329, epochs=53, mean epo
chs=67, time=0:00:05.17
#43: score=0.692113, mean score=0.670656,stdev=0.048864, epochs=51, mean epo
chs=67, time=0:00:05.56
#44: score=0.745355, mean score=0.672353,stdev=0.049572, epochs=66, mean epo
chs=67, time=0:00:10.66
#45: score=0.631125, mean score=0.671437,stdev=0.049393, epochs=75, mean epo
chs=67, time=0:00:07.16
#46: score=0.664004, mean score=0.671276,stdev=0.048865, epochs=57, mean epo
chs=67, time=0:00:05.69
#47: score=0.686937, mean score=0.671609,stdev=0.048395, epochs=52, mean epo
chs=66, time=0:00:05.07
#48: score=0.760827, mean score=0.673468,stdev=0.049555, epochs=40, mean epo
chs=66, time=0:00:04.14
#49: score=0.665493, mean score=0.673305,stdev=0.049060, epochs=60, mean epo
chs=66, time=0:00:10.65
#50: score=0.692625, mean score=0.673691,stdev=0.048642, epochs=55, mean epo
chs=65, time=0:00:05.22

```

Benchmarking

Now that we've seen how to bootstrap with both classification and regression, we can start to try to optimize the hyperparameters for the **jh-simple-dataset** data. For this example, we will encode for classification of the product column. Evaluation will be in log loss.

```
In [11]: import pandas as pd
from scipy.stats import zscore

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

# Generate dummies for job
df = pd.concat([df,pd.get_dummies(df['job'],prefix="job")],axis=1)
df.drop('job', axis=1, inplace=True)

# Generate dummies for area
df = pd.concat([df,pd.get_dummies(df['area'],prefix="area")],
               axis=1)
df.drop('area', axis=1, inplace=True)

# Missing values for income
med = df['income'].median()
df['income'] = df['income'].fillna(med)

# Standardize ranges
df['income'] = zscore(df['income'])
df['aspect'] = zscore(df['aspect'])
df['save_rate'] = zscore(df['save_rate'])
df['age'] = zscore(df['age'])
df['subscriptions'] = zscore(df['subscriptions'])

# Convert to numpy - Classification
x_columns = df.columns.drop('product').drop('id')
x = df[x_columns].values
dummies = pd.get_dummies(df['product']) # Classification
products = dummies.columns
y = dummies.values
```

I performed some optimization, and the code has the best settings that I could determine. Later in this book, we will see how we can use an automatic process to optimize the hyperparameters.

```
In [13]: import pandas as pd
import os
import numpy as np
import time
import tensorflow.keras.initializers
import statistics
from sklearn import metrics
from sklearn.model_selection import StratifiedKFold
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation, Dropout
from tensorflow.keras import regularizers
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.model_selection import StratifiedShuffleSplit
from tensorflow.keras.layers import LeakyReLU, PReLU
```

```

SPLITS = 100

# Bootstrap
boot = StratifiedShuffleSplit(n_splits=SPLITS, test_size=0.1)

# Track progress
mean_benchmark = []
epochs_needed = []
num = 0

# Loop through samples
for train, test in boot.split(x, df['product']):
    start_time = time.time()
    num+=1

    # Split train and test
    x_train = x[train]
    y_train = y[train]
    x_test = x[test]
    y_test = y[test]

    # Construct neural network
    model = Sequential()
    model.add(Dense(100, input_dim=x.shape[1], activation=PReLU(), \
                  kernel_regularizer=regularizers.l2(1e-4))) # Hidden 1
    model.add(Dropout(0.5))
    model.add(Dense(100, activation=PReLU(), \
                  activity_regularizer=regularizers.l2(1e-4))) # Hidden 2
    model.add(Dropout(0.5))
    model.add(Dense(100, activation=PReLU(), \
                  activity_regularizer=regularizers.l2(1e-4)))
    )) # Hidden 3
    # model.add(Dropout(0.5)) - Usually better performance
    # without dropout on final layer
    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=100, verbose=0, mode='auto', restore_best_weights=True)

    # Train on the bootstrap sample
    model.fit(x_train,y_train,validation_data=(x_test,y_test), \
              callbacks=[monitor],verbose=0,epochs=1000)
    epochs = monitor.stopped_epoch
    epochs_needed.append(epochs)

    # Predict on the out of boot (validation)
    pred = model.predict(x_test)

    # Measure this bootstrap's log loss
    y_compare = np.argmax(y_test, axis=1) # For log loss calculation
    score = metrics.log_loss(y_compare, pred)
    mean_benchmark.append(score)
    m1 = statistics.mean(mean_benchmark)
    m2 = statistics.mean(epochs_needed)
    mdev = statistics.pstdev(mean_benchmark)

```

```
# Record this iteration
time_took = time.time() - start_time
print(f"#{{num}}: score={{score:.6f}}, mean score={{m1:.6f}}, "
      f"stdev={{mdev:.6f}}, epochs={{epochs}}, "
      f"mean epochs={{int(m2)}}, time={{hms_string(time_took)}}")
```

```
#1: score=0.642887, mean score=0.642887,stdev=0.000000, epochs=325,mean epoch  
hs=325, time=0:00:42.10  
#2: score=0.555518, mean score=0.599202,stdev=0.043684, epochs=208,mean epoch  
hs=266, time=0:00:41.74  
#3: score=0.605537, mean score=0.601314,stdev=0.035793, epochs=187,mean epoch  
hs=240, time=0:00:24.22  
#4: score=0.609415, mean score=0.603339,stdev=0.031195, epochs=250,mean epoch  
hs=242, time=0:00:41.72  
#5: score=0.619657, mean score=0.606603,stdev=0.028655, epochs=201,mean epoch  
hs=234, time=0:00:26.10  
#6: score=0.638641, mean score=0.611943,stdev=0.028755, epochs=172,mean epoch  
hs=223, time=0:00:41.73  
#7: score=0.671137, mean score=0.620399,stdev=0.033731, epochs=203,mean epoch  
hs=220, time=0:00:26.58  
#8: score=0.635294, mean score=0.622261,stdev=0.031935, epochs=209,mean epoch  
hs=219, time=0:00:41.74  
#9: score=0.633694, mean score=0.623531,stdev=0.030322, epochs=162,mean epoch  
hs=213, time=0:00:41.78  
#10: score=0.596081, mean score=0.620786,stdev=0.029921, epochs=197,mean epoch  
chs=211, time=0:00:41.74  
#11: score=0.583717, mean score=0.617416,stdev=0.030454, epochs=232,mean epoch  
chs=213, time=0:00:41.77  
#12: score=0.686736, mean score=0.623193,stdev=0.034889, epochs=216,mean epoch  
chs=213, time=0:00:28.25  
#13: score=0.684454, mean score=0.627905,stdev=0.037284, epochs=134,mean epoch  
chs=207, time=0:00:21.58  
#14: score=0.573696, mean score=0.624033,stdev=0.038545, epochs=184,mean epoch  
chs=205, time=0:00:23.81  
#15: score=0.723944, mean score=0.630694,stdev=0.044808, epochs=170,mean epoch  
chs=203, time=0:00:41.80  
#16: score=0.659891, mean score=0.632519,stdev=0.043957, epochs=203,mean epoch  
chs=203, time=0:00:41.80  
#17: score=0.569637, mean score=0.628820,stdev=0.045139, epochs=204,mean epoch  
chs=203, time=0:00:41.77  
#18: score=0.608905, mean score=0.627713,stdev=0.044103, epochs=233,mean epoch  
chs=205, time=0:00:41.76  
#19: score=0.734381, mean score=0.633328,stdev=0.049092, epochs=193,mean epoch  
chs=204, time=0:00:25.25  
#20: score=0.587099, mean score=0.631016,stdev=0.048899, epochs=252,mean epoch  
chs=206, time=0:00:42.07  
#21: score=0.661902, mean score=0.632487,stdev=0.048171, epochs=211,mean epoch  
chs=206, time=0:00:41.79  
#22: score=0.656783, mean score=0.633591,stdev=0.047335, epochs=145,mean epoch  
chs=204, time=0:00:19.19  
#23: score=0.611230, mean score=0.632619,stdev=0.046519, epochs=201,mean epoch  
chs=204, time=0:00:41.77  
#24: score=0.638759, mean score=0.632875,stdev=0.045556, epochs=223,mean epoch  
chs=204, time=0:00:28.67  
#25: score=0.635676, mean score=0.632987,stdev=0.044639, epochs=240,mean epoch  
chs=206, time=0:00:41.77  
#26: score=0.599321, mean score=0.631692,stdev=0.044248, epochs=199,mean epoch  
chs=205, time=0:00:42.10  
#27: score=0.696892, mean score=0.634107,stdev=0.045133, epochs=146,mean epoch  
chs=203, time=0:00:21.28  
#28: score=0.637397, mean score=0.634224,stdev=0.044324, epochs=179,mean epoch  
chs=202, time=0:00:23.53
```

```
#29: score=0.645323, mean score=0.634607,stdev=0.043600, epochs=256,mean epo
chs=204, time=0:00:32.44
#30: score=0.588104, mean score=0.633057,stdev=0.043672, epochs=199,mean epo
chs=204, time=0:00:25.96
#31: score=0.676097, mean score=0.634445,stdev=0.043630, epochs=229,mean epo
chs=205, time=0:00:41.74
#32: score=0.667709, mean score=0.635485,stdev=0.043331, epochs=155,mean epo
chs=203, time=0:00:20.61
#33: score=0.616544, mean score=0.634911,stdev=0.042793, epochs=283,mean epo
chs=206, time=0:00:36.55
#34: score=0.622340, mean score=0.634541,stdev=0.042212, epochs=174,mean epo
chs=205, time=0:00:22.82
#35: score=0.665123, mean score=0.635415,stdev=0.041916, epochs=205,mean epo
chs=205, time=0:00:27.05
#36: score=0.573597, mean score=0.633698,stdev=0.042560, epochs=205,mean epo
chs=205, time=0:00:41.81
#37: score=0.617111, mean score=0.633249,stdev=0.042067, epochs=253,mean epo
chs=206, time=0:00:31.92
#38: score=0.627494, mean score=0.633098,stdev=0.041520, epochs=205,mean epo
chs=206, time=0:00:41.76
#39: score=0.669212, mean score=0.634024,stdev=0.041380, epochs=193,mean epo
chs=206, time=0:00:42.14
#40: score=0.684894, mean score=0.635296,stdev=0.041624, epochs=171,mean epo
chs=205, time=0:00:21.86
#41: score=0.648313, mean score=0.635613,stdev=0.041162, epochs=205,mean epo
chs=205, time=0:00:41.74
#42: score=0.679919, mean score=0.636668,stdev=0.041226, epochs=251,mean epo
chs=206, time=0:00:41.74
#43: score=0.701787, mean score=0.638183,stdev=0.041909, epochs=146,mean epo
chs=204, time=0:00:21.29
#44: score=0.660646, mean score=0.638693,stdev=0.041566, epochs=168,mean epo
chs=204, time=0:00:41.80
#45: score=0.660335, mean score=0.639174,stdev=0.041225, epochs=136,mean epo
chs=202, time=0:00:21.67
#46: score=0.656875, mean score=0.639559,stdev=0.040856, epochs=154,mean epo
chs=201, time=0:00:21.33
#47: score=0.679169, mean score=0.640402,stdev=0.040821, epochs=286,mean epo
chs=203, time=0:00:41.78
#48: score=0.608082, mean score=0.639728,stdev=0.040656, epochs=173,mean epo
chs=202, time=0:00:22.20
#49: score=0.590421, mean score=0.638722,stdev=0.040839, epochs=185,mean epo
chs=202, time=0:00:24.23
#50: score=0.616646, mean score=0.638281,stdev=0.040546, epochs=273,mean epo
chs=203, time=0:00:41.76
#51: score=0.683312, mean score=0.639163,stdev=0.040629, epochs=163,mean epo
chs=202, time=0:00:41.76
#52: score=0.686289, mean score=0.640070,stdev=0.040754, epochs=166,mean epo
chs=202, time=0:00:22.03
#53: score=0.701892, mean score=0.641236,stdev=0.041235, epochs=185,mean epo
chs=201, time=0:00:24.02
#54: score=0.647809, mean score=0.641358,stdev=0.040861, epochs=171,mean epo
chs=201, time=0:00:22.38
#55: score=0.678673, mean score=0.642036,stdev=0.040793, epochs=160,mean epo
chs=200, time=0:00:41.77
#56: score=0.594752, mean score=0.641192,stdev=0.040910, epochs=185,mean epo
chs=200, time=0:00:25.42
```

```
#57: score=0.719842, mean score=0.642572,stdev=0.041843, epochs=124,mean epo
chs=198, time=0:00:17.03
#58: score=0.689348, mean score=0.643378,stdev=0.041926, epochs=223,mean epo
chs=199, time=0:00:29.47
#59: score=0.657452, mean score=0.643617,stdev=0.041608, epochs=220,mean epo
chs=199, time=0:00:28.31
#60: score=0.611100, mean score=0.643075,stdev=0.041470, epochs=226,mean epo
chs=200, time=0:00:29.49
#61: score=0.660965, mean score=0.643368,stdev=0.041191, epochs=162,mean epo
chs=199, time=0:00:21.28
#62: score=0.669189, mean score=0.643785,stdev=0.040987, epochs=147,mean epo
chs=198, time=0:00:21.31
#63: score=0.652563, mean score=0.643924,stdev=0.040675, epochs=187,mean epo
chs=198, time=0:00:41.74
#64: score=0.590525, mean score=0.643090,stdev=0.040896, epochs=275,mean epo
chs=199, time=0:00:35.78
#65: score=0.699827, mean score=0.643963,stdev=0.041176, epochs=182,mean epo
chs=199, time=0:00:23.86
#66: score=0.665028, mean score=0.644282,stdev=0.040944, epochs=214,mean epo
chs=199, time=0:00:28.54
#67: score=0.729557, mean score=0.645554,stdev=0.041932, epochs=225,mean epo
chs=199, time=0:00:41.79
#68: score=0.586906, mean score=0.644692,stdev=0.042217, epochs=219,mean epo
chs=200, time=0:00:28.43
#69: score=0.717007, mean score=0.645740,stdev=0.042792, epochs=124,mean epo
chs=199, time=0:00:21.30
#70: score=0.670428, mean score=0.646093,stdev=0.042586, epochs=198,mean epo
chs=199, time=0:00:41.92
#71: score=0.717004, mean score=0.647091,stdev=0.043103, epochs=203,mean epo
chs=199, time=0:00:42.09
#72: score=0.582071, mean score=0.646188,stdev=0.043474, epochs=174,mean epo
chs=198, time=0:00:41.77
#73: score=0.723909, mean score=0.647253,stdev=0.044110, epochs=199,mean epo
chs=198, time=0:00:41.78
#74: score=0.685384, mean score=0.647768,stdev=0.044032, epochs=145,mean epo
chs=198, time=0:00:19.13
#75: score=0.584444, mean score=0.646924,stdev=0.044336, epochs=205,mean epo
chs=198, time=0:00:41.77
#76: score=0.681646, mean score=0.647381,stdev=0.044221, epochs=160,mean epo
chs=197, time=0:00:21.30
#77: score=0.585961, mean score=0.646583,stdev=0.044480, epochs=195,mean epo
chs=197, time=0:00:42.19
#78: score=0.626380, mean score=0.646324,stdev=0.044252, epochs=231,mean epo
chs=198, time=0:00:41.76
#79: score=0.700790, mean score=0.647014,stdev=0.044391, epochs=188,mean epo
chs=197, time=0:00:24.41
#80: score=0.664455, mean score=0.647232,stdev=0.044155, epochs=164,mean epo
chs=197, time=0:00:21.95
#81: score=0.601657, mean score=0.646669,stdev=0.044169, epochs=205,mean epo
chs=197, time=0:00:41.80
#82: score=0.661004, mean score=0.646844,stdev=0.043927, epochs=151,mean epo
chs=197, time=0:00:19.90
#83: score=0.693299, mean score=0.647404,stdev=0.043955, epochs=161,mean epo
chs=196, time=0:00:21.32
#84: score=0.732184, mean score=0.648413,stdev=0.044649, epochs=147,mean epo
chs=196, time=0:00:20.04
```

```
#85: score=0.628028, mean score=0.648173,stdev=0.044440, epochs=197,mean epo  
chs=196, time=0:00:25.61  
#86: score=0.626073, mean score=0.647916,stdev=0.044245, epochs=176,mean epo  
chs=195, time=0:00:23.27  
#87: score=0.632806, mean score=0.647742,stdev=0.044019, epochs=261,mean epo  
chs=196, time=0:00:41.76  
#88: score=0.694768, mean score=0.648277,stdev=0.044051, epochs=204,mean epo  
chs=196, time=0:00:41.80  
#89: score=0.699703, mean score=0.648855,stdev=0.044137, epochs=183,mean epo  
chs=196, time=0:00:23.10  
#90: score=0.611230, mean score=0.648437,stdev=0.044068, epochs=270,mean epo  
chs=197, time=0:00:34.62  
#91: score=0.637264, mean score=0.648314,stdev=0.043841, epochs=257,mean epo  
chs=197, time=0:00:41.78  
#92: score=0.678976, mean score=0.648647,stdev=0.043718, epochs=158,mean epo  
chs=197, time=0:00:21.24  
#93: score=0.627937, mean score=0.648424,stdev=0.043534, epochs=218,mean epo  
chs=197, time=0:00:41.74  
#94: score=0.644387, mean score=0.648381,stdev=0.043304, epochs=197,mean epo  
chs=197, time=0:00:41.76  
#95: score=0.660005, mean score=0.648504,stdev=0.043092, epochs=167,mean epo  
chs=197, time=0:00:21.69  
#96: score=0.674187, mean score=0.648771,stdev=0.042946, epochs=174,mean epo  
chs=197, time=0:00:22.79  
#97: score=0.654942, mean score=0.648835,stdev=0.042729, epochs=162,mean epo  
chs=196, time=0:00:21.30  
#98: score=0.644139, mean score=0.648787,stdev=0.042513, epochs=173,mean epo  
chs=196, time=0:00:22.70  
#99: score=0.697473, mean score=0.649279,stdev=0.042577, epochs=172,mean epo  
chs=196, time=0:00:41.79  
#100: score=0.678298, mean score=0.649569,stdev=0.042462, epochs=169,mean epo  
chs=196, time=0:00:21.90
```

[Open in Colab](#)

T81-558: Applications of Deep Neural Networks

Module 6: Convolutional Neural Networks (CNN) for Computer Vision

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

Module 6 Material

- **Part 6.1: Image Processing in Python** [\[Video\]](#) [\[Notebook\]](#)
- Part 6.2: Using Convolutional Neural Networks [\[Video\]](#) [\[Notebook\]](#)
- Part 6.3: Using Pretrained Neural Networks with Keras [\[Video\]](#) [\[Notebook\]](#)
- Part 6.4: Looking at Keras Generators and Image Augmentation [\[Video\]](#) [\[Notebook\]](#)
- Part 6.5: Recognizing Multiple Images with YOLOv5 [\[Video\]](#) [\[Notebook\]](#)

Google CoLab Instructions

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

In []:

```
try:  
    %tensorflow_version 2.x  
    COLAB = True  
    print("Note: using Google CoLab")  
except:  
    print("Note: not using Google CoLab")  
    COLAB = False  
  
# Nicely formatted time string  
def hms_string(sec_elapsed):  
    h = int(sec_elapsed / (60 * 60))  
    m = int((sec_elapsed % (60 * 60)) / 60)  
    s = sec_elapsed % 60  
    return f"{h}:{m:02}:{s:05.2f}"
```

Note: using Google CoLab

Part 6.1: Image Processing in Python

Computer vision requires processing images. These images might come from a

video stream, a camera, or files on a storage drive. We begin this chapter by looking at how to process images with Python. To use images in Python, we will make use of the Pillow package. The following program uses Pillow to load and display an image.

In []:

```
from PIL import Image, ImageFile
from matplotlib.pyplot import imshow
import requests
from io import BytesIO
import numpy as np

%matplotlib inline

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

response = requests.get(url,headers={'User-Agent': 'Mozilla/5.0'})
img = Image.open(BytesIO(response.content))
img.load()

print(np.asarray(img))

img
```

```
[[[199 213 240]
 [200 214 240]
 [200 214 240]

 ...
 [ 86   34   96]
 [ 48    4   57]
 [ 57   21   65]]]

[[199 213 239]
 [200 214 240]
 [200 214 240]

 ...
 [215 215 251]
 [252 242 255]
 [237 218 250]]]

[[200 214 240]
 [200 214 240]
 [201 215 241]

 ...
 [227 238 255]
 [167 180 197]
 [ 61   79   91]]]

...
[[136 112 108]
 [137 113 109]
 [140 116 112]

 ...
 [ 85   84   63]
 [ 91   90   69]
 [ 93   92   72]]]

[[119  90  84]
 [118  89  83]
 [119  90  84]

 ...
 [ 86   84   61]
 [ 89   87   64]
 [ 90   88   65]]]

[[129  96  89]
 [129  96  89]
 [131  98  91]

 ...
 [ 86   82   57]
 [ 89   85   60]
 [ 89   85   60]]]
```

Out[]:



Creating Images from Pixels in Python

You can use Pillow to create an image from a 3D NumPy cube-shaped array. The rows and columns specify the pixels. The third dimension (size 3) defines red, green, and blue color values. The following code demonstrates creating a simple image from a NumPy array.

In []:

```
from PIL import Image
import numpy as np

w, h = 64, 64
data = np.zeros((h, w, 3), dtype=np.uint8)

# Yellow
for row in range(32):
    for col in range(32):
        data[row,col] = [255,255,0]

# Red
for row in range(32):
    for col in range(32):
        data[row+32,col] = [255,0,0]

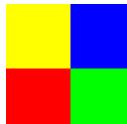
# Green
for row in range(32):
    for col in range(32):
        data[row+32,col+32] = [0,255,0]

# Blue
for row in range(32):
    for col in range(32):
```

```
data[row,col+32] = [0,0,255]

img = Image.fromarray(data, 'RGB')
img
```

Out[]:



Transform Images in Python (at the pixel level)

We can combine the last two programs and modify images. Here we take the mean color of each pixel and form a grayscale image.

```
In [ ]:
from PIL import Image, ImageFile
from matplotlib.pyplot import imshow
import requests
from io import BytesIO

%matplotlib inline

url = "https://data.heatonresearch.com/images/jupyter/brookings.jpeg"
response = requests.get(url,headers={'User-Agent': 'Mozilla/5.0'})

img = Image.open(BytesIO(response.content))
img.load()

img_array = np.asarray(img)
rows = img_array.shape[0]
cols = img_array.shape[1]

print("Rows: {}, Cols: {}".format(rows,cols))

# Create new image
img2_array = np.zeros((rows, cols, 3), dtype=np.uint8)
for row in range(rows):
    for col in range(cols):
        t = np.mean(img_array[row,col])
        img2_array[row,col] = [t,t,t]

img2 = Image.fromarray(img2_array, 'RGB')
img2
```

Rows: 768, Cols: 1024

Out[]:



Standardize Images

When processing several images together, it is sometimes essential to standardize them. The following code reads a sequence of images and causes them to all be of the same size and perfectly square. If the input images are not square, cropping will occur.

In []:

```
%matplotlib inline
from PIL import Image, ImageFile
from matplotlib.pyplot import imshow
import requests
import numpy as np
from io import BytesIO
from IPython.display import display, HTML

images = [
    "https://data.heatonresearch.com/images/jupyter/brookings.jpeg",
    "https://data.heatonresearch.com/images/jupyter/SeigleHall.jpeg",
    "https://data.heatonresearch.com/images/jupyter/WUSTLKnight.jpeg"
]

def crop_square(image):
    width, height = image.size

    # Crop the image, centered
    new_width = min(width, height)
    new_height = new_width
    left = (width - new_width)/2
    top = (height - new_height)/2
    right = (width + new_width)/2
    bottom = (height + new_height)/2
```

```
    return image.crop((left, top, right, bottom))

x = []

for url in images:
    ImageFile.LOAD_TRUNCATED_IMAGES = False
    response = requests.get(url, headers={'User-Agent': 'Mozilla/5.0'})
    img = Image.open(BytesIO(response.content))
    img.load()
    img = crop_square(img)
    img = img.resize((128, 128), Image.ANTIALIAS)
    print(url)
    display(img)
    img_array = np.asarray(img)
    img_array = img_array.flatten()
    img_array = img_array.astype(np.float32)
    img_array = (img_array - 128) / 128
    x.append(img_array)

x = np.array(x)

print(x.shape)
```

<https://data.heatonresearch.com/images/jupyter/brookings.jpeg>



<https://data.heatonresearch.com/images/jupyter/SeigleHall.jpeg>



<https://data.heatonresearch.com/images/jupyter/WUSTLKnight.jpeg>



(3, 49152)

Adding Noise to an Image

Sometimes it is beneficial to add noise to images. We might use noise to augment images to generate more training data or modify images to test the recognition capabilities of neural networks. It is essential to see how to add noise to an image. There are many ways to add such noise. The following code adds random black

squares to the image to produce noise.

In []:

```
from PIL import Image, ImageFile
from matplotlib.pyplot import imshow
import requests
from io import BytesIO

%matplotlib inline

def add_noise(a):
    a2 = a.copy()
    rows = a2.shape[0]
    cols = a2.shape[1]
    s = int(min(rows,cols)/20) # size of spot is 1/20 of smallest dimens.

    for i in range(100):
        x = np.random.randint(cols-s)
        y = np.random.randint(rows-s)
        a2[y:(y+s),x:(x+s)] = 0

    return a2

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

response = requests.get(url,headers={'User-Agent': 'Mozilla/5.0'})
img = Image.open(BytesIO(response.content))
img.load()

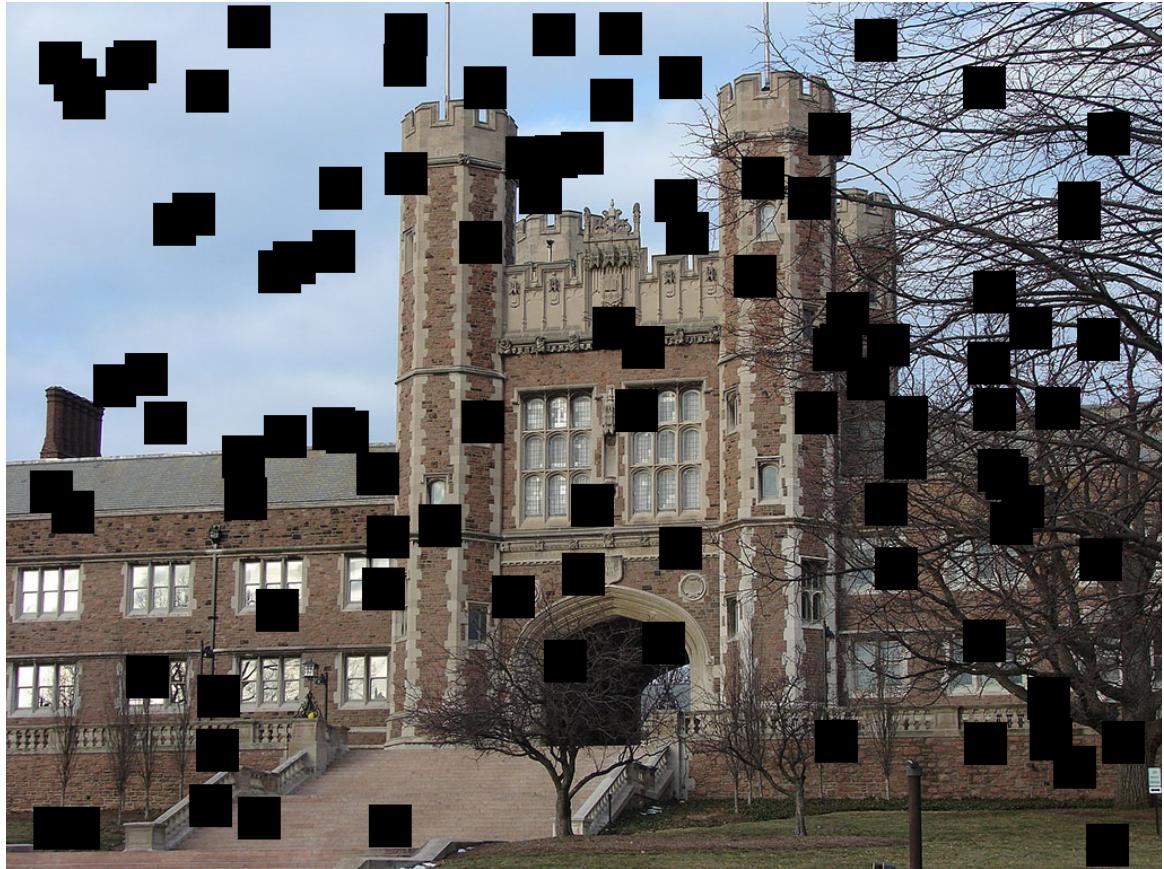
img_array = np.asarray(img)
rows = img_array.shape[0]
cols = img_array.shape[1]

print("Rows: {}, Cols: {}".format(rows,cols))

# Create new image
img2_array = img_array.astype(np.uint8)
print(img2_array.shape)
img2_array = add_noise(img2_array)
img2 = Image.fromarray(img2_array, 'RGB')
img2
```

Rows: 768, Cols: 1024
(768, 1024, 3)

Out[]:



Preprocessing Many Images

To download images, we define several paths. We will download sample images of paperclips from the URL specified by **DOWNLOAD_SOURCE**. Once downloaded, we will unzip and perform the preprocessing on these paper clips. I mean for this code as a starting point for other image preprocessing.

In []:

```
import os

URL = "https://github.com/jeffheaton/data-mirror/releases/"
#DOWNLOAD_SOURCE = URL+"download/v1/iris-image.zip"
DOWNLOAD_SOURCE = URL+"download/v1/paperclips.zip"
DOWNLOAD_NAME = DOWNLOAD_SOURCE[DOWNLOAD_SOURCE.rfind('/')+1:]

if COLAB:
    PATH = "/content"
    EXTRACT_TARGET = os.path.join(PATH, "clips")
    SOURCE = os.path.join(PATH, "/content/clips/paperclips")
    TARGET = os.path.join(PATH, "/content/clips-processed")
else:
    # I used this locally on my machine, you may need different
    PATH = "/Users/jeff/temp"
    EXTRACT_TARGET = os.path.join(PATH, "clips")
    SOURCE = os.path.join(PATH, "clips/paperclips")
    TARGET = os.path.join(PATH, "clips-processed")
```

Next, we download the images. This part depends on the origin of your images. The following code downloads images from a URL, where a ZIP file contains the images. The code unzips the ZIP file.

In []:

```
# HIDE OUTPUT
!wget -O {os.path.join(PATH,DOWNLOAD_NAME)} {DOWNLOAD_SOURCE}
!mkdir -p {SOURCE}
!mkdir -p {TARGET}
!mkdir -p {EXTRACT_TARGET}
!unzip -o -j -d {SOURCE} {os.path.join(PATH, DOWNLOAD_NAME)} >/dev/null

--2021-11-26 19:11:35-- https://github.com/jeffheaton/data-mirror/releases/download/v1/paperclips.zip
Resolving github.com (github.com)... 140.82.114.4
Connecting to github.com (github.com)|140.82.114.4|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://objects.githubusercontent.com/github-production-release-asset-2e65be/408419764/25830812-b9e6-4ddf-93b6-7932d9ef5982?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAIWNJYAX4CSVEH53A%2F20211126%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Date=20211126T191135Z&X-Amz-Expires=300&X-Amz-Signature=37ac15e40f8fcdc15ad13d36ef58562e1fdab5e74d71e8e6adedd5cdf5808c60&X-Amz-SignedHeaders=host&actor_id=0&key_id=0&repo_id=408419764&response-content-disposition=attachment%3B%20filename%3Dpaperclips.zip&response-content-type=application%2Foctet-stream [following]
--2021-11-26 19:11:35-- https://objects.githubusercontent.com/github-production-release-asset-2e65be/408419764/25830812-b9e6-4ddf-93b6-7932d9ef5982?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAIWNJYAX4CSVEH53A%2F20211126%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Date=20211126T191135Z&X-Amz-Expires=300&X-Amz-Signature=37ac15e40f8fcdc15ad13d36ef58562e1fdab5e74d71e8e6adedd5cdf5808c60&X-Amz-SignedHeaders=host&actor_id=0&key_id=0&repo_id=408419764&response-content-disposition=attachment%3B%20filename%3Dpaperclips.zip&response-content-type=application%2Foctet-stream
Resolving objects.githubusercontent.com (objects.githubusercontent.com)... 185.199.108.133, 185.199.109.133, 185.199.110.133, ...
Connecting to objects.githubusercontent.com (objects.githubusercontent.com)|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 163590691 (156M) [application/octet-stream]
Saving to: '/content/paperclips.zip'

/content/paperclips 100%[=====] 156.01M 29.9MB/s in 5.4s

2021-11-26 19:11:41 (29.1 MB/s) - '/content/paperclips.zip' saved [163590691]
```

The following code contains functions that we use to preprocess the images. The **crop_square** function converts images to a square by cropping extra data. The **scale** function increases or decreases the size of an image. The **standardize** function ensures an image is full color; a mix of color and grayscale images can be problematic.

In []:

```
import imageio
import glob
from tqdm import tqdm
from PIL import Image
import os

def scale(img, scale_width, scale_height):
    # Scale the image
    img = img.resize((
        scale_width,
```

```
    scale_height),
    Image.ANTIALIAS)

    return img

def standardize(image):
    rgbimg = Image.new("RGB", image.size)
    rgbimg.paste(image)
    return rgbimg

def fail_below(image, check_width, check_height):
    width, height = image.size
    assert width == check_width
    assert height == check_height
```

Next, we loop through each image. The images are loaded, and you can apply any desired transformations. Ultimately, the script saves the images as JPG.

```
In [ ]: files = glob.glob(os.path.join(SOURCE, "*.jpg"))

for file in tqdm(files):
    try:
        target = ""
        name = os.path.basename(file)
        filename, _ = os.path.splitext(name)
        img = Image.open(file)
        img = standardize(img)
        img = crop_square(img)
        img = scale(img, 128, 128)
        #fail_below(img, 128, 128)

        target = os.path.join(TARGET, filename + ".jpg")
        img.save(target, quality=25)
    except KeyboardInterrupt:
        print("Keyboard interrupt")
        break
    except AssertionError:
        print("Assertion")
        break
    except:
        print("Unexpected exception while processing image source: " \
              f"{file}, target: {target}", exc_info=True)
```

100% |██████████| 25000/25000 [01:32<00:00, 268.82it/s]

Now we can zip the preprocessed files and store them somewhere.

Module 6 Assignment

You can find the first assignment here: [assignment 6](#)

```
In [ ]:
```

[Open in Colab](#)

T81-558: Applications of Deep Neural Networks

Module 6: Convolutional Neural Networks (CNN) for Computer Vision

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

Module 6 Material

- Part 6.1: Image Processing in Python [\[Video\]](#) [\[Notebook\]](#)
- **Part 6.2: Using Convolutional Neural Networks** [\[Video\]](#) [\[Notebook\]](#)
- Part 6.3: Using Pretrained Neural Networks with Keras [\[Video\]](#) [\[Notebook\]](#)
- Part 6.4: Looking at Keras Generators and Image Augmentation [\[Video\]](#) [\[Notebook\]](#)
- Part 6.5: Recognizing Multiple Images with YOLOv5 [\[Video\]](#) [\[Notebook\]](#)

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  
  
# Nicely formatted time string  
def hms_string(sec_elapsed):  
    h = int(sec_elapsed / (60 * 60))  
    m = int((sec_elapsed % (60 * 60)) / 60)  
    s = sec_elapsed % 60  
    return f"{h}:{m:02}:{s:05.2f}"
```

Note: using Google CoLab

Part 6.2: Keras Neural Networks for Digits and Fashion MNIST

This module will focus on computer vision. There are some important differences and similarities with previous neural networks.

- We will usually use classification, though regression is still an option.
- The input to the neural network is now 3D (height, width, color)
- Data are not transformed; no z-scores or dummy variables.
- Processing time is much longer.
- We now have different layer types: dense layers (just like before), convolution layers, and max-pooling layers.
- Data will no longer arrive as CSV files. TensorFlow provides some utilities for going directly from the image to the input for a neural network.

Common Computer Vision Data Sets

There are many data sets for computer vision. Two of the most popular classic datasets are the MNIST digits data set and the CIFAR image data sets. We will not use either of these datasets in this course, but it is important to be familiar with them since neural network texts often refer to them.

The [MNIST Digits Data Set](#) is very popular in the neural network research community. You can see a sample of it in Figure 6.MNIST.

Figure 6.MNIST: MNIST Data Set

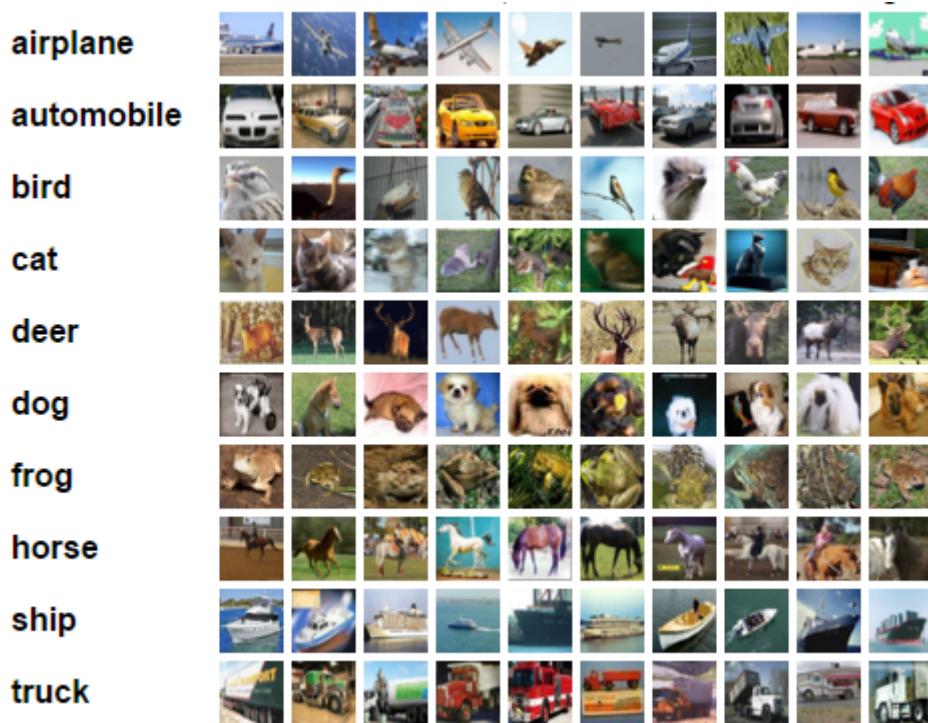


[Fashion-MNIST](#) is a dataset of [Zalando](#)'s article images—consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image associated with a label from 10 classes. Fashion-MNIST is a direct drop-in replacement for the original [MNIST dataset](#) for benchmarking machine learning algorithms. It shares the same image size and structure of training and testing splits. You can see this data in Figure 6.MNIST-FASHION.

Figure 6.MNIST-FASHION: MNIST Fashion Data Set

The [CIFAR-10](#) and [CIFAR-100](#) datasets are also frequently used by the neural network research community.

Figure 6.CIFAR: CIFAR Data Set



The CIFAR-10 data set contains low-rez images that are divided into 10 classes. The CIFAR-100 data set contains 100 classes in a hierarchy.

Convolutional Neural Networks (CNNs)

The convolutional neural network (CNN) is a neural network technology that has profoundly impacted the area of computer vision (CV). Fukushima (1980) [Cite:fukushima1980neocognitron] introduced the original concept of a convolutional neural network, and LeCun, Bottou, Bengio & Haffner (1998) [Cite:lecun1995convolutional] greatly improved this work. From this research, Yan LeCun introduced the famous LeNet-5 neural network architecture. This chapter follows the LeNet-5 style of convolutional neural network.

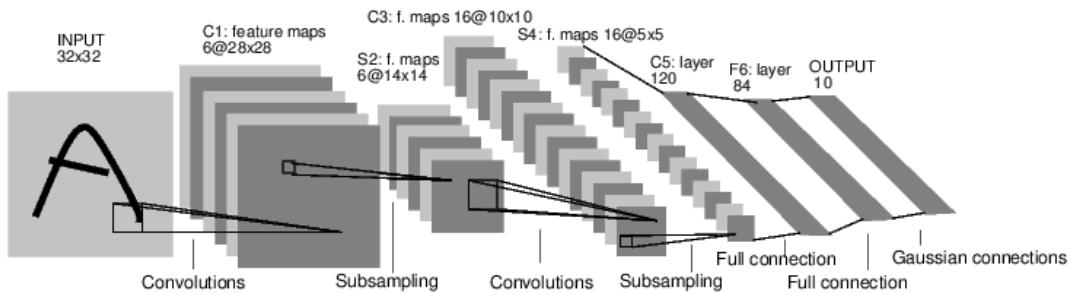
Although computer vision primarily uses CNNs, this technology has some applications outside of the field. You need to realize that if you want to utilize CNNs on non-visual data, you must find a way to encode your data to mimic the properties of visual data.

The order of the input array elements is crucial to the training. In contrast, most neural networks that are not CNNs treat their input data as a long vector of values, and the order in which you arrange the incoming features in this vector is irrelevant. You cannot change the order for these types of neural networks after you have trained the network.

The CNN network arranges the inputs into a grid. This arrangement worked well with images because the pixels in closer proximity to each other are important to each other. The order of pixels in an image is significant. The human body is a relevant example of this type of order. For the design of the face, we are accustomed to eyes being near to each other.

This advance in CNNs is due to years of research on biological eyes. In other words, CNNs utilize overlapping fields of input to simulate features of biological eyes. Until this breakthrough, AI had been unable to reproduce the capabilities of biological vision. Scale, rotation, and noise have presented challenges for AI computer vision research. You can observe the complexity of biological eyes in the example that follows. A friend raises a sheet of paper with a large number written on it. As your friend moves nearer to you, the number is still identifiable. In the same way, you can still identify the number when your friend rotates the paper. Lastly, your friend creates noise by drawing lines on the page, but you can still identify the number. As you can see, these examples demonstrate the high function of the biological eye and allow you to understand better the research breakthrough of CNNs. That is, this neural network can process scale, rotation, and noise in the field of computer vision. You can see this network structure in Figure 6.LENET.

Figure 6.LENET: A LeNET-5 Network (LeCun, 1998)



So far, we have only seen one layer type (dense layers). By the end of this book we will have seen:

- **Dense Layers** - Fully connected layers.
- **Convolution Layers** - Used to scan across images.
- **Max Pooling Layers** - Used to downsample images.
- **Dropout Layers** - Used to add regularization.
- **LSTM and Transformer Layers** - Used for time series data.

Convolution Layers

The first layer that we will examine is the convolutional layer. We will begin by looking at the hyper-parameters that you must specify for a convolutional layer in most neural network frameworks that support the CNN:

- Number of filters
- Filter Size
- Stride
- Padding
- Activation Function/Non-Linearity

The primary purpose of a convolutional layer is to detect features such as edges, lines, blobs of color, and other visual elements. The filters can detect these features. The more filters we give to a convolutional layer, the more features it can see.

A filter is a square-shaped object that scans over the image. A grid can represent the individual pixels of a grid. You can think of the convolutional layer as a smaller grid that sweeps left to right over each image row. There is also a hyperparameter that specifies both the width and height of the square-shaped filter. The following figure shows this configuration in which you see the six convolutional filters sweeping over the image grid:

A convolutional layer has weights between it and the previous layer or image grid. Each pixel on each convolutional layer is a weight. Therefore, the number of weights between a convolutional layer and its predecessor layer or image field is the following:

$$[\text{FilterSize}] * [\text{FilterSize}] * [\# \text{ of Filters}]$$

For example, if the filter size were 5 (5x5) for 10 filters, there would be 250 weights.

You need to understand how the convolutional filters sweep across the previous layer's output or image grid. Figure 6.CNN illustrates the sweep:

Figure 6.CNN: Convolutional Neural Network

0	0	0	0	0	0	0	0	0	0
0	1	3	2	0	8	4	2	1	3
0	0	5	4	0	8	7	3	2	1
0	8	1	8	0	4	1	3	6	2
0	18	4	8	1	23	2	4	17	0
0	19	8	24	14	22	10	11	12	0
0	20	62	23	9	21	6	7	4	0
0	3	13	17	5	13	16	2	8	0
0	0	0	0	0	0	0	0	0	0

The above figure shows a convolutional filter with 4 and a padding size of 1. The padding size is responsible for the border of zeros in the area that the filter sweeps. Even though the image is 8x7, the extra padding provides a virtual image size of 9x8 for the filter to sweep across. The stride specifies the number of positions the convolutional filters will stop. The convolutional filters move to the right, advancing by the number of cells specified in the stride. Once you reach the far right, the convolutional filter moves back to the far left; then, it moves down by the stride amount and continues to the right again.

Some constraints exist concerning the size of the stride. The stride cannot be 0. The convolutional filter would never move if you set the stride. Furthermore, neither the stride nor the convolutional filter size can be larger than the previous grid. There

are additional constraints on the stride (s), padding (p), and the filter width (f) for an image of width (w). Specifically, the convolutional filter must be able to start at the far left or top border, move a certain number of strides, and land on the far right or bottom border. The following equation shows the number of steps a convolutional operator must take to cross the image:

$$steps = \frac{w - f + 2p}{s} + 1$$

The number of steps must be an integer. In other words, it cannot have decimal places. The purpose of the padding (p) is to be adjusted to make this equation become an integer value.

Max Pooling Layers

Max-pool layers downsample a 3D box to a new one with smaller dimensions. Typically, you can always place a max-pool layer immediately following the convolutional layer. The LENET shows the max-pool layer immediately after layers C1 and C3. These max-pool layers progressively decrease the size of the dimensions of the 3D boxes passing through them. This technique can avoid overfitting (Krizhevsky, Sutskever & Hinton, 2012).

A pooling layer has the following hyper-parameters:

- Spatial Extent (f)
- Stride (s)

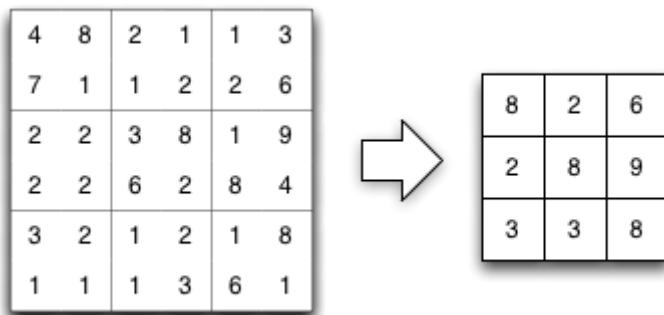
Unlike convolutional layers, max-pool layers do not use padding. Additionally, max-pool layers have no weights, so training does not affect them. These layers downsample their 3D box input. The 3D box output by a max-pool layer will have a width equal to this equation:

$$w_2 = \frac{w_1 - f}{s} + 1$$

The height of the 3D box produced by the max-pool layer is calculated similarly with this equation:

$$h_2 = \frac{h_1 - f}{s} + 1$$

The depth of the 3D box produced by the max-pool layer is equal to the depth the 3D box received as input. The most common setting for the hyper-parameters of a max-pool layer is $f=2$ and $s=2$. The spatial extent (f) specifies that boxes of 2×2 will be scaled down to single pixels. Of these four pixels, the pixel with the maximum value will represent the 2×2 pixel in the new grid. Because squares of size 4 are replaced with size 1, 75% of the pixel information is lost. The following figure shows this transformation as a 6×6 grid becomes a 3×3 :

Figure 6.MAXPOOL: Max Pooling Layer

Of course, the above diagram shows each pixel as a single number. A grayscale image would have this characteristic. We usually take the average of the three numbers for an RGB image to determine which pixel has the maximum value.

Regression Convolutional Neural Networks

We will now look at two examples, one for regression and another for classification. For supervised computer vision, your dataset will need some labels. For classification, this label usually specifies what the image is a picture of. For regression, this "label" is some numeric quantity the image should produce, such as a count. We will look at two different means of providing this label.

The first example will show how to handle regression with convolution neural networks. We will provide an image and expect the neural network to count items in that image. We will use a [dataset](#) that I created that contains a random number of paperclips. The following code will download this dataset for you.

In [2]:

```
import os

URL = "https://github.com/jeffheaton/data-mirror/releases/"
DOWNLOAD_SOURCE = URL+"download/v1/paperclips.zip"
DOWNLOAD_NAME = DOWNLOAD_SOURCE[DOWNLOAD_SOURCE.rfind('/')+1:]

if COLAB:
    PATH = "/content"
else:
    # I used this locally on my machine, you may need different
    PATH = "/Users/jeff/temp"

EXTRACT_TARGET = os.path.join(PATH, "clips")
SOURCE = os.path.join(EXTRACT_TARGET, "paperclips")
```

Next, we download the images. This part depends on the origin of your images. The following code downloads images from a URL, where a ZIP file contains the images. The code unzips the ZIP file.

In [3]:

```
# HIDE OUTPUT
!wget -O {os.path.join(PATH,DOWNLOAD_NAME)} {DOWNLOAD_SOURCE}
!mkdir -p {SOURCE}
```

```
!mkdir -p {TARGET}
!mkdir -p {EXTRACT_TARGET}
!unzip -o -j -d {SOURCE} {os.path.join(PATH, DOWNLOAD_NAME)} >/dev/null

--2022-03-01 22:45:29-- https://github.com/jeffheaton/data-mirror/releases/download/v1/paperclips.zip
Resolving github.com (github.com)... 140.82.121.4
Connecting to github.com (github.com)|140.82.121.4|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://objects.githubusercontent.com/github-production-release-asset-2e65be/408419764/25830812-b9e6-4ddf-93b6-7932d9ef598?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAIWNJYAX4CSVEH53A%2F20220301%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Date=20220301T224529Z&X-Amz-Expires=300&X-Amz-Signature=92d63a15fadf8b244e13610e65457b6628f86d8465cb450a763a00f4e70fde8f&X-Amz-SignedHeaders=host&actor_id=0&key_id=0&repo_id=408419764&response-content-disposition=attachment%3B%20filename%3Dpaperclips.zip&response-content-type=application%2Foctet-stream [following]
--2022-03-01 22:45:29-- https://objects.githubusercontent.com/github-production-release-asset-2e65be/408419764/25830812-b9e6-4ddf-93b6-7932d9ef598?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAIWNJYAX4CSVEH53A%2F20220301%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Date=20220301T224529Z&X-Amz-Expires=300&X-Amz-Signature=92d63a15fadf8b244e13610e65457b6628f86d8465cb450a763a00f4e70fde8f&X-Amz-SignedHeaders=host&actor_id=0&key_id=0&repo_id=408419764&response-content-disposition=attachment%3B%20filename%3Dpaperclips.zip&response-content-type=application%2Foctet-stream
Resolving objects.githubusercontent.com (objects.githubusercontent.com)... 185.199.108.133, 185.199.109.133, 185.199.110.133, ...
Connecting to objects.githubusercontent.com (objects.githubusercontent.com)|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 163590691 (156M) [application/octet-stream]
Saving to: '/content/paperclips.zip'

/content/paperclips 100%[=====] 156.01M 22.9MB/s in 6.0s

2022-03-01 22:45:35 (26.1 MB/s) - '/content/paperclips.zip' saved [163590691/163590691]
```

The labels are contained in a CSV file named **train.csv** for regression. This file has just two labels, **id** and **clip_count**. The ID specifies the filename; for example, row id 1 corresponds to the file **clips-1.jpg**. The following code loads the labels for the training set and creates a new column, named **filename**, that contains the filename of each image, based on the **id** column.

In [4]:

```
import pandas as pd

df = pd.read_csv(
    os.path.join(SOURCE, "train.csv"),
    na_values=['NA', '?'])

df['filename']="clips-"+df["id"].astype(str)+".jpg"
```

This results in the following dataframe.

In [5]:

```
df
```

Out[5]:

	id	clip_count	filename
0	30001	11	clips-30001.jpg
1	30002	2	clips-30002.jpg
2	30003	26	clips-30003.jpg
3	30004	41	clips-30004.jpg
4	30005	49	clips-30005.jpg
...
19995	49996	35	clips-49996.jpg
19996	49997	54	clips-49997.jpg
19997	49998	72	clips-49998.jpg
19998	49999	24	clips-49999.jpg
19999	50000	35	clips-50000.jpg

20000 rows × 3 columns

Separate into a training and validation (for early stopping)

In [6]:

```
TRAIN_PCT = 0.9
TRAIN_CUT = int(len(df) * TRAIN_PCT)

df_train = df[0:TRAIN_CUT]
df_validate = df[TRAIN_CUT:]

print(f"Training size: {len(df_train)}")
print(f"Validate size: {len(df_validate)}")
```

```
Training size: 18000
Validate size: 2000
```

We are now ready to create two `ImageDataGenerator` objects. We currently use a generator, which creates additional training data by manipulating the source material. This technique can produce considerably stronger neural networks. The generator below flips the images both vertically and horizontally. Keras will train the neuron network both on the original images and the flipped images. This augmentation increases the size of the training data considerably. Module 6.4 goes deeper into the transformations you can perform. You can also specify a target size to resize the images automatically.

The function `flow_from_dataframe` loads the labels from a Pandas dataframe connected to our `train.csv` file. When we demonstrate classification, we will use the `flow_from_directory`; which loads the labels from the directory structure rather than a CSV.

In [7]:

```
import tensorflow as tf
import keras.preprocessing
```

```
from keras.preprocessing import image
from keras.preprocessing.image import ImageDataGenerator

training_datagen = ImageDataGenerator(
    rescale = 1./255,
    horizontal_flip=True,
    vertical_flip=True,
    fill_mode='nearest')

train_generator = training_datagen.flow_from_dataframe(
    dataframe=df_train,
    directory=SOURCE,
    x_col="filename",
    y_col="clip_count",
    target_size=(256, 256),
    batch_size=32,
    class_mode='other')

validation_datagen = ImageDataGenerator(rescale = 1./255)

val_generator = validation_datagen.flow_from_dataframe(
    dataframe=df_validate,
    directory=SOURCE,
    x_col="filename",
    y_col="clip_count",
    target_size=(256, 256),
    class_mode='other')
```

Found 18000 validated image filenames.

Found 2000 validated image filenames.

We can now train the neural network. The code to build and train the neural network is not that different than in the previous modules. We will use the Keras Sequential class to provide layers to the neural network. We now have several new layer types that we did not previously see.

- **Conv2D** - The convolution layers.
- **MaxPooling2D** - The max-pooling layers.
- **Flatten** - Flatten the 2D (and higher) tensors to allow a Dense layer to process.
- **Dense** - Dense layers, the same as demonstrated previously. Dense layers often form the final output layers of the neural network.

The training code is very similar to previously. This code is for regression, so a final linear activation is used, along with mean_squared_error for the loss function. The generator provides both the x and y matrixes we previously supplied.

In [8]:

```
from tensorflow.keras.callbacks import EarlyStopping
import time

model = tf.keras.models.Sequential([
    # Note the input shape is the desired size of the image 150x150
    # with 3 bytes color.
    # This is the first convolution
    tf.keras.layers.Conv2D(64, (3,3), activation='relu',
        input_shape=(256, 256, 3)),
    tf.keras.layers.MaxPooling2D(2, 2),
    # The second convolution
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
```

```
tf.keras.layers.MaxPooling2D(2,2),  
tf.keras.layers.Flatten(),  
# 512 neuron hidden layer  
tf.keras.layers.Dense(512, activation='relu'),  
tf.keras.layers.Dense(1, activation='linear')  
])  
  
model.summary()  
epoch_steps = 250 # needed for 2.2  
validation_steps = len(df_validate)  
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)  
  
start_time = time.time()  
history = model.fit(train_generator,  
    verbose = 1,  
    validation_data=val_generator, callbacks=[monitor], epochs=25)  
  
elapsed_time = time.time() - start_time  
print("Elapsed time: {}".format(hms_string(elapsed_time)))
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 254, 254, 64)	1792
max_pooling2d (MaxPooling2D)	(None, 127, 127, 64)	0
conv2d_1 (Conv2D)	(None, 125, 125, 64)	36928
max_pooling2d_1 (MaxPooling 2D)	(None, 62, 62, 64)	0
flatten (Flatten)	(None, 246016)	0
dense (Dense)	(None, 512)	125960704
dense_1 (Dense)	(None, 1)	513
<hr/>		
Total params: 125,999,937		
Trainable params: 125,999,937		
Non-trainable params: 0		

Epoch 1/25
563/563 [=====] - 64s 96ms/step - loss: 193.3214 - val_loss: 25.4486
Epoch 2/25
563/563 [=====] - 52s 92ms/step - loss: 25.5836 - val_loss: 13.8235
Epoch 3/25
563/563 [=====] - 53s 93ms/step - loss: 18.8956 - val_loss: 12.4469
Epoch 4/25
563/563 [=====] - 52s 92ms/step - loss: 18.8489 - val_loss: 20.0634
Epoch 5/25
563/563 [=====] - 52s 92ms/step - loss: 16.5164 - val_loss: 17.8989
Epoch 6/25
563/563 [=====] - 52s 91ms/step - loss: 15.5483 - val_loss: 16.3132
Epoch 7/25
563/563 [=====] - 52s 92ms/step - loss: 15.6795 - val_loss: 16.9717
Epoch 8/25
563/563 [=====] - 52s 92ms/step - loss: 11.5606 - val_loss: 12.1518
Epoch 9/25
563/563 [=====] - 52s 92ms/step - loss: 26.0139 - val_loss: 34.7480
Epoch 10/25
563/563 [=====] - 52s 93ms/step - loss: 13.2884 - val_loss: 12.1712
Epoch 11/25
563/563 [=====] - 52s 93ms/step - loss: 10.7682 - val_loss: 12.7467
Epoch 12/25
563/563 [=====] - 53s 94ms/step - loss: 9.8051 - val_loss: 9.8815
Epoch 13/25
563/563 [=====] - 62s 109ms/step - loss: 8.2021 -

```
val_loss: 8.1277
Epoch 14/25
563/563 [=====] - 52s 93ms/step - loss: 7.0807 -
val_loss: 6.8239
Epoch 15/25
563/563 [=====] - 52s 93ms/step - loss: 6.6521 -
val_loss: 7.0292
Epoch 16/25
563/563 [=====] - 52s 92ms/step - loss: 5.2696 -
val_loss: 6.5994
Epoch 17/25
563/563 [=====] - 52s 93ms/step - loss: 5.1749 -
val_loss: 12.0623
Epoch 18/25
563/563 [=====] - 52s 92ms/step - loss: 27.0990 -
val_loss: 15.1000
Epoch 19/25
563/563 [=====] - 52s 92ms/step - loss: 10.3702 -
val_loss: 6.4094
Epoch 20/25
563/563 [=====] - 54s 96ms/step - loss: 5.1338 -
val_loss: 5.1066
Epoch 21/25
563/563 [=====] - 60s 107ms/step - loss: 4.1413 -
val_loss: 6.3927
Epoch 22/25
563/563 [=====] - 55s 97ms/step - loss: 3.8659 -
val_loss: 4.6390
Epoch 23/25
563/563 [=====] - 55s 98ms/step - loss: 3.4385 -
val_loss: 4.1845
Epoch 24/25
563/563 [=====] - 54s 95ms/step - loss: 3.2399 -
val_loss: 4.0449
Epoch 25/25
563/563 [=====] - 53s 94ms/step - loss: 3.2823 -
val_loss: 4.4899
Elapsed time: 0:22:22.78
```

This code will run very slowly if you do not use a GPU. The above code takes approximately 13 minutes with a GPU.

Score Regression Image Data

Scoring/predicting from a generator is a bit different than training. We do not want augmented images, and we do not wish to have the dataset shuffled. For scoring, we want a prediction for each input. We construct the generator as follows:

- shuffle=False
- batch_size=1
- class_mode=None

We use a **batch_size** of 1 to guarantee that we do not run out of GPU memory if our prediction set is large. You can increase this value for better performance. The **class_mode** is None because there is no y, or label. After all, we are predicting.

In [9]: df_test = pd.read_csv(

```
os.path.join(SOURCE,"test.csv"),
na_values=['NA', '?'])

df_test['filename']="clips-"+df_test["id"].astype(str)+".jpg"

test_datagen = ImageDataGenerator(rescale = 1./255)

test_generator = validation_datagen.flow_from_dataframe(
    dataframe=df_test,
    directory=SOURCE,
    x_col="filename",
    batch_size=1,
    shuffle=False,
    target_size=(256, 256),
    class_mode=None)
```

Found 5000 validated image filenames.

We need to reset the generator to ensure we are always at the beginning.

In [10]:

```
test_generator.reset()
pred = model.predict(test_generator,steps=len(df_test))
```

We can now generate a CSV file to hold the predictions.

In [11]:

```
df_submit = pd.DataFrame({'id':df_test['id'],'clip_count':pred.flatten()})
df_submit.to_csv(os.path.join(PATH,"submit.csv"),index=False)
```

Classification Neural Networks

Just like earlier in this module, we will load data. However, this time we will use a dataset of images of three different types of the iris flower. This zip file contains three different directories that specify each image's label. The directories are named the same as the labels:

- iris-setosa
- iris-versicolour
- iris-virginica

In [12]:

```
import os

URL = "https://github.com/jeffheaton/data-mirror/releases"
DOWNLOAD_SOURCE = URL + "/download/v1/iris-image.zip"
DOWNLOAD_NAME = DOWNLOAD_SOURCE[DOWNLOAD_SOURCE.rfind('/')+1:]

if COLAB:
    PATH = "/content"
    EXTRACT_TARGET = os.path.join(PATH, "iris")
    SOURCE = EXTRACT_TARGET # In this case its the same, no subfolder
else:
    # I used this locally on my machine, you may need different
    PATH = "/Users/jeff/temp"
    EXTRACT_TARGET = os.path.join(PATH, "iris")
    SOURCE = EXTRACT_TARGET # In this case its the same, no subfolder
```

Just as before, we unzip the images.

In [13]:

```
# HIDE OUTPUT
!wget -O {os.path.join(PATH,DOWNLOAD_NAME)} {DOWNLOAD_SOURCE}
!mkdir -p {SOURCE}
!mkdir -p {TARGET}
!mkdir -p {EXTRACT_TARGET}
!unzip -o -d {EXTRACT_TARGET} {os.path.join(PATH, DOWNLOAD_NAME)} >/dev/null

--2022-03-01 23:08:29-- https://github.com/jeffheaton/data-mirror/releases/download/v1/iris-image.zip
Resolving github.com (github.com)... 140.82.121.4
Connecting to github.com (github.com)|140.82.121.4|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://objects.githubusercontent.com/github-production-release-asset-2e65be/408419764/d548babd-36c3-414e-add2-a5d9ab941e6e?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAIWNJYAX4CSVEH53A%2F20220301%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Date=20220301T230829Z&X-Amz-Expires=300&X-Amz-Signature=f9315f39373d1b8e6ae60a618db5da58714c8bab017e0f55b38c7c0a55d2cb20&X-Amz-SignedHeaders=host&actor_id=0&key_id=0&repo_id=408419764&response-content-disposition=attachment%3B%20filename%3Diris-image.zip&response-content-type=application%2Foctet-stream [following]
--2022-03-01 23:08:30-- https://objects.githubusercontent.com/github-production-release-asset-2e65be/408419764/d548babd-36c3-414e-add2-a5d9ab941e6e?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAIWNJYAX4CSVEH53A%2F20220301%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Date=20220301T230829Z&X-Amz-Expires=300&X-Amz-Signature=f9315f39373d1b8e6ae60a618db5da58714c8bab017e0f55b38c7c0a55d2cb20&X-Amz-SignedHeaders=host&actor_id=0&key_id=0&repo_id=408419764&response-content-disposition=attachment%3B%20filename%3Diris-image.zip&response-content-type=application%2Foctet-stream
Resolving objects.githubusercontent.com (objects.githubusercontent.com)... 185.199.109.133, 185.199.110.133, 185.199.108.133, ...
Connecting to objects.githubusercontent.com (objects.githubusercontent.com)|185.199.109.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 5587253 (5.3M) [application/octet-stream]
Saving to: '/content/iris-image.zip'

/content/iris-image 100%[=====] 5.33M 6.65MB/s in 0.8s

2022-03-01 23:08:31 (6.65 MB/s) - '/content/iris-image.zip' saved [5587253/5587253]
```

You can see these folders with the following command.

In [14]:

```
!ls /content/iris

iris-setosa  iris-versicolour  iris-virginica
```

We set up the generator, similar to before. This time we use `flow_from_directory` to get the labels from the directory structure.

In [15]:

```
import tensorflow as tf
import keras.preprocessing
from keras.preprocessing import image
from keras.preprocessing.image import ImageDataGenerator

training_datagen = ImageDataGenerator(
    rescale = 1./255,
```

```
horizontal_flip=True,
vertical_flip=True,
width_shift_range=[-200,200],
rotation_range=360,

fill_mode='nearest')

train_generator = training_datagen.flow_from_directory(
    directory=SOURCE, target_size=(256, 256),
    class_mode='categorical', batch_size=32, shuffle=True)

validation_datagen = ImageDataGenerator(rescale = 1./255)

validation_generator = validation_datagen.flow_from_directory(
    directory=SOURCE, target_size=(256, 256),
    class_mode='categorical', batch_size=32, shuffle=True)
```

Found 421 images belonging to 3 classes.

Found 421 images belonging to 3 classes.

Training the neural network with classification is similar to regression.

In [16]:

```
from tensorflow.keras.callbacks import EarlyStopping

class_count = len(train_generator.class_indices)

model = tf.keras.models.Sequential([
    # Note the input shape is the desired size of the image
    # 300x300 with 3 bytes color
    # This is the first convolution
    tf.keras.layers.Conv2D(16, (3,3), activation='relu',
        input_shape=(256, 256, 3)),
    tf.keras.layers.MaxPooling2D(2, 2),
    # The second convolution
    tf.keras.layers.Conv2D(32, (3,3), activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.MaxPooling2D(2,2),
    # The third convolution
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.MaxPooling2D(2,2),
    # The fourth convolution
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    # The fifth convolution
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    # Flatten the results to feed into a DNN

    tf.keras.layers.Flatten(),
    tf.keras.layers.Dropout(0.5),
    # 512 neuron hidden layer
    tf.keras.layers.Dense(512, activation='relu'),
    # Only 1 output neuron. It will contain a value from 0-1
    tf.keras.layers.Dense(class_count, activation='softmax')
])

model.summary()

model.compile(loss = 'categorical_crossentropy', optimizer='adam')

model.fit(train_generator, epochs=50, steps_per_epoch=10,
```

```
verbose = 1)
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 254, 254, 16)	448
max_pooling2d_2 (MaxPooling 2D)	(None, 127, 127, 16)	0
conv2d_3 (Conv2D)	(None, 125, 125, 32)	4640
dropout (Dropout)	(None, 125, 125, 32)	0
max_pooling2d_3 (MaxPooling 2D)	(None, 62, 62, 32)	0
conv2d_4 (Conv2D)	(None, 60, 60, 64)	18496
dropout_1 (Dropout)	(None, 60, 60, 64)	0
max_pooling2d_4 (MaxPooling 2D)	(None, 30, 30, 64)	0
conv2d_5 (Conv2D)	(None, 28, 28, 64)	36928
max_pooling2d_5 (MaxPooling 2D)	(None, 14, 14, 64)	0
conv2d_6 (Conv2D)	(None, 12, 12, 64)	36928
max_pooling2d_6 (MaxPooling 2D)	(None, 6, 6, 64)	0
flatten_1 (Flatten)	(None, 2304)	0
dropout_2 (Dropout)	(None, 2304)	0
dense_2 (Dense)	(None, 512)	1180160
dense_3 (Dense)	(None, 3)	1539

Total params: 1,279,139

Trainable params: 1,279,139

Non-trainable params: 0

Epoch 1/50
 10/10 [=====] - 6s 486ms/step - loss: 1.0254
 Epoch 2/50
 10/10 [=====] - 5s 472ms/step - loss: 0.9060
 Epoch 3/50
 10/10 [=====] - 5s 474ms/step - loss: 0.9712
 Epoch 4/50
 10/10 [=====] - 5s 520ms/step - loss: 0.9099
 Epoch 5/50
 10/10 [=====] - 5s 517ms/step - loss: 0.9061
 Epoch 6/50
 10/10 [=====] - 5s 510ms/step - loss: 0.8965
 Epoch 7/50
 10/10 [=====] - 5s 458ms/step - loss: 0.8909
 Epoch 8/50
 10/10 [=====] - 5s 514ms/step - loss: 0.8941
 Epoch 9/50

```
10/10 [=====] - 5s 493ms/step - loss: 0.9248
Epoch 10/50
10/10 [=====] - 5s 500ms/step - loss: 0.8780
Epoch 11/50
10/10 [=====] - 5s 453ms/step - loss: 0.8724
Epoch 12/50
10/10 [=====] - 5s 448ms/step - loss: 0.8901
Epoch 13/50
10/10 [=====] - 5s 456ms/step - loss: 0.8817
Epoch 14/50
10/10 [=====] - 5s 465ms/step - loss: 0.9040
Epoch 15/50
10/10 [=====] - 5s 449ms/step - loss: 0.8779
Epoch 16/50
10/10 [=====] - 4s 441ms/step - loss: 0.8479
Epoch 17/50
10/10 [=====] - 5s 499ms/step - loss: 0.8713
Epoch 18/50
10/10 [=====] - 5s 456ms/step - loss: 0.8432
Epoch 19/50
10/10 [=====] - 4s 444ms/step - loss: 0.8816
Epoch 20/50
10/10 [=====] - 5s 508ms/step - loss: 0.8791
Epoch 21/50
10/10 [=====] - 5s 497ms/step - loss: 0.8553
Epoch 22/50
10/10 [=====] - 5s 448ms/step - loss: 0.8275
Epoch 23/50
10/10 [=====] - 5s 502ms/step - loss: 0.8216
Epoch 24/50
10/10 [=====] - 5s 456ms/step - loss: 0.8739
Epoch 25/50
10/10 [=====] - 5s 510ms/step - loss: 0.8650
Epoch 26/50
10/10 [=====] - 5s 456ms/step - loss: 0.8405
Epoch 27/50
10/10 [=====] - 5s 456ms/step - loss: 0.8729
Epoch 28/50
10/10 [=====] - 5s 499ms/step - loss: 0.8618
Epoch 29/50
10/10 [=====] - 5s 500ms/step - loss: 0.8125
Epoch 30/50
10/10 [=====] - 5s 504ms/step - loss: 0.8813
Epoch 31/50
10/10 [=====] - 5s 508ms/step - loss: 0.8392
Epoch 32/50
10/10 [=====] - 5s 449ms/step - loss: 0.8377
Epoch 33/50
10/10 [=====] - 5s 499ms/step - loss: 0.8509
Epoch 34/50
10/10 [=====] - 5s 454ms/step - loss: 0.8647
Epoch 35/50
10/10 [=====] - 5s 466ms/step - loss: 0.8874
Epoch 36/50
10/10 [=====] - 5s 502ms/step - loss: 0.9221
Epoch 37/50
10/10 [=====] - 5s 511ms/step - loss: 0.9186
Epoch 38/50
10/10 [=====] - 5s 496ms/step - loss: 0.8549
Epoch 39/50
10/10 [=====] - 5s 493ms/step - loss: 0.9194
Epoch 40/50
10/10 [=====] - 5s 496ms/step - loss: 0.8528
```

```
Epoch 41/50
10/10 [=====] - 5s 453ms/step - loss: 0.9105
Epoch 42/50
10/10 [=====] - 5s 454ms/step - loss: 0.8462
Epoch 43/50
10/10 [=====] - 5s 459ms/step - loss: 0.8858
Epoch 44/50
10/10 [=====] - 5s 497ms/step - loss: 0.9119
Epoch 45/50
10/10 [=====] - 5s 458ms/step - loss: 0.8799
Epoch 46/50
10/10 [=====] - 5s 499ms/step - loss: 0.8582
Epoch 47/50
10/10 [=====] - 5s 493ms/step - loss: 0.8536
Epoch 48/50
10/10 [=====] - 5s 490ms/step - loss: 0.8669
Epoch 49/50
10/10 [=====] - 5s 458ms/step - loss: 0.7957
Epoch 50/50
10/10 [=====] - 5s 501ms/step - loss: 0.8670
```

The iris image dataset is not easy to predict; it turns out that a tabular dataset of measurements is more manageable. However, we can achieve a 63%.

In [22]:

```
from sklearn.metrics import accuracy_score
import numpy as np

validation_generator.reset()
pred = model.predict(validation_generator)

predict_classes = np.argmax(pred, axis=1)
expected_classes = validation_generator.classes

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

Accuracy: 0.6389548693586699

Other Resources

- [Imagenet:Large Scale Visual Recognition Challenge 2014](#)
- [Andrej Karpathy](#) - PhD student/instructor at Stanford.
- [CS231n Convolutional Neural Networks for Visual Recognition](#) - Stanford course on computer vision/CNN's.
- [CS231n - GitHub](#)
- [ConvNetJS](#) - JavaScript library for deep learning.

[!\[\]\(efc1a7ccd6ed48da13de67779d2998d5_img.jpg\) Open in Colab](#)

T81-558: Applications of Deep Neural Networks

Module 6: Convolutional Neural Networks (CNN) for Computer Vision

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

Module 6 Material

- Part 6.1: Image Processing in Python [\[Video\]](#) [\[Notebook\]](#)
- Part 6.2: Using Convolutional Neural Networks [\[Video\]](#) [\[Notebook\]](#)
- **Part 6.3: Using Pretrained Neural Networks with Keras** [\[Video\]](#) [\[Notebook\]](#)
- Part 6.4: Looking at Keras Generators and Image Augmentation [\[Video\]](#) [\[Notebook\]](#)
- Part 6.5: Recognizing Multiple Images with YOLOv5 [\[Video\]](#) [\[Notebook\]](#)

Google CoLab Instructions

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

In [1]:

```
# Detect Colab if present
try:
    from google.colab import drive
    COLAB = True
    print("Note: using Google CoLab")
    %tensorflow_version 2.x
except:
    print("Note: not using Google CoLab")
    COLAB = False

# Nicely formatted time string
def hms_string(sec_elapsed):
    h = int(sec_elapsed / (60 * 60))
    m = int((sec_elapsed % (60 * 60)) / 60)
    s = sec_elapsed % 60
    return f"{h}:{m:>02}:{s:>05.2f}"
```

Note: using Google CoLab

Part 6.3: Transfer Learning for

Computer Vision

Many advanced prebuilt neural networks are available for computer vision, and Keras provides direct access to many networks. Transfer learning is the technique where you use these prebuilt neural networks. Module 9 takes a deeper look at transfer learning.

There are several different levels of transfer learning.

- Use a prebuilt neural network in its entirety
- Use a prebuilt neural network's structure
- Use a prebuilt neural network's weights

We will begin by using the MobileNet prebuilt neural network in its entirety. MobileNet will be loaded and allowed to classify simple images. We can already classify 1,000 images through this technique without ever having trained the network.

In [2]:

```
import pandas as pd
import numpy as np
import os
import tensorflow.keras
import matplotlib.pyplot as plt
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D
from tensorflow.keras.applications import MobileNet
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.mobilenet import preprocess_input
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
```

We begin by downloading weights for a MobileNet trained for the imagenet dataset, which will take some time to download the first time you train the network.

In [3]:

```
# HIDE OUTPUT
model = MobileNet(weights='imagenet', include_top=True)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/mobilenet/mobilenet_1_0_224_tf.h5
17227776/17225924 [=====] - 0s 0us/step
17235968/17225924 [=====] - 0s 0us/step
```

The loaded network is a Keras neural network. However, this is a neural network that a third party engineered on advanced hardware. Merely looking at the structure of an advanced state-of-the-art neural network can be educational.

In [4]:

```
model.summary()
```

Model: "mobilenet_1.00_224"

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 224, 224, 3)]	0
conv1 (Conv2D)	(None, 112, 112, 32)	864
conv1_bn (BatchNormalizatio n)	(None, 112, 112, 32)	128
conv1_relu (ReLU)	(None, 112, 112, 32)	0
conv_dw_1 (DepthwiseConv2D)	(None, 112, 112, 32)	288
conv_dw_1_bn (BatchNormaliz ation)	(None, 112, 112, 32)	128
conv_dw_1_relu (ReLU)	(None, 112, 112, 32)	0
conv_pw_1 (Conv2D)	(None, 112, 112, 64)	2048
conv_pw_1_bn (BatchNormaliz ation)	(None, 112, 112, 64)	256
conv_pw_1_relu (ReLU)	(None, 112, 112, 64)	0
conv_pad_2 (ZeroPadding2D)	(None, 113, 113, 64)	0
conv_dw_2 (DepthwiseConv2D)	(None, 56, 56, 64)	576
conv_dw_2_bn (BatchNormaliz ation)	(None, 56, 56, 64)	256
conv_dw_2_relu (ReLU)	(None, 56, 56, 64)	0
conv_pw_2 (Conv2D)	(None, 56, 56, 128)	8192
conv_pw_2_bn (BatchNormaliz ation)	(None, 56, 56, 128)	512
conv_pw_2_relu (ReLU)	(None, 56, 56, 128)	0
conv_dw_3 (DepthwiseConv2D)	(None, 56, 56, 128)	1152
conv_dw_3_bn (BatchNormaliz ation)	(None, 56, 56, 128)	512
conv_dw_3_relu (ReLU)	(None, 56, 56, 128)	0
conv_pw_3 (Conv2D)	(None, 56, 56, 128)	16384
conv_pw_3_bn (BatchNormaliz ation)	(None, 56, 56, 128)	512
conv_pw_3_relu (ReLU)	(None, 56, 56, 128)	0
conv_pad_4 (ZeroPadding2D)	(None, 57, 57, 128)	0
conv_dw_4 (DepthwiseConv2D)	(None, 28, 28, 128)	1152
conv_dw_4_bn (BatchNormaliz ation)	(None, 28, 28, 128)	512

conv_dw_4_relu (ReLU)	(None, 28, 28, 128)	0
conv_pw_4 (Conv2D)	(None, 28, 28, 256)	32768
conv_pw_4_bn (BatchNormalization)	(None, 28, 28, 256)	1024
conv_pw_4_relu (ReLU)	(None, 28, 28, 256)	0
conv_dw_5 (DepthwiseConv2D)	(None, 28, 28, 256)	2304
conv_dw_5_bn (BatchNormalization)	(None, 28, 28, 256)	1024
conv_dw_5_relu (ReLU)	(None, 28, 28, 256)	0
conv_pw_5 (Conv2D)	(None, 28, 28, 256)	65536
conv_pw_5_bn (BatchNormalization)	(None, 28, 28, 256)	1024
conv_pw_5_relu (ReLU)	(None, 28, 28, 256)	0
conv_pad_6 (ZeroPadding2D)	(None, 29, 29, 256)	0
conv_dw_6 (DepthwiseConv2D)	(None, 14, 14, 256)	2304
conv_dw_6_bn (BatchNormalization)	(None, 14, 14, 256)	1024
conv_dw_6_relu (ReLU)	(None, 14, 14, 256)	0
conv_pw_6 (Conv2D)	(None, 14, 14, 512)	131072
conv_pw_6_bn (BatchNormalization)	(None, 14, 14, 512)	2048
conv_pw_6_relu (ReLU)	(None, 14, 14, 512)	0
conv_dw_7 (DepthwiseConv2D)	(None, 14, 14, 512)	4608
conv_dw_7_bn (BatchNormalization)	(None, 14, 14, 512)	2048
conv_dw_7_relu (ReLU)	(None, 14, 14, 512)	0
conv_pw_7 (Conv2D)	(None, 14, 14, 512)	262144
conv_pw_7_bn (BatchNormalization)	(None, 14, 14, 512)	2048
conv_pw_7_relu (ReLU)	(None, 14, 14, 512)	0
conv_dw_8 (DepthwiseConv2D)	(None, 14, 14, 512)	4608
conv_dw_8_bn (BatchNormalization)	(None, 14, 14, 512)	2048
conv_dw_8_relu (ReLU)	(None, 14, 14, 512)	0
conv_pw_8 (Conv2D)	(None, 14, 14, 512)	262144

conv_pw_8_bn (BatchNormalization)	(None, 14, 14, 512)	2048
conv_pw_8_relu (ReLU)	(None, 14, 14, 512)	0
conv_dw_9 (DepthwiseConv2D)	(None, 14, 14, 512)	4608
conv_dw_9_bn (BatchNormalization)	(None, 14, 14, 512)	2048
conv_dw_9_relu (ReLU)	(None, 14, 14, 512)	0
conv_pw_9 (Conv2D)	(None, 14, 14, 512)	262144
conv_pw_9_bn (BatchNormalization)	(None, 14, 14, 512)	2048
conv_pw_9_relu (ReLU)	(None, 14, 14, 512)	0
conv_dw_10 (DepthwiseConv2D)	(None, 14, 14, 512)	4608
conv_dw_10_bn (BatchNormalization)	(None, 14, 14, 512)	2048
conv_dw_10_relu (ReLU)	(None, 14, 14, 512)	0
conv_pw_10 (Conv2D)	(None, 14, 14, 512)	262144
conv_pw_10_bn (BatchNormalization)	(None, 14, 14, 512)	2048
conv_pw_10_relu (ReLU)	(None, 14, 14, 512)	0
conv_dw_11 (DepthwiseConv2D)	(None, 14, 14, 512)	4608
conv_dw_11_bn (BatchNormalization)	(None, 14, 14, 512)	2048
conv_dw_11_relu (ReLU)	(None, 14, 14, 512)	0
conv_pw_11 (Conv2D)	(None, 14, 14, 512)	262144
conv_pw_11_bn (BatchNormalization)	(None, 14, 14, 512)	2048
conv_pw_11_relu (ReLU)	(None, 14, 14, 512)	0
conv_pad_12 (ZeroPadding2D)	(None, 15, 15, 512)	0
conv_dw_12 (DepthwiseConv2D)	(None, 7, 7, 512)	4608
conv_dw_12_bn (BatchNormalization)	(None, 7, 7, 512)	2048
conv_dw_12_relu (ReLU)	(None, 7, 7, 512)	0
conv_pw_12 (Conv2D)	(None, 7, 7, 1024)	524288
conv_pw_12_bn (BatchNormalization)	(None, 7, 7, 1024)	4096

conv_pw_12_relu (ReLU)	(None, 7, 7, 1024)	0
conv_dw_13 (DepthwiseConv2D)	(None, 7, 7, 1024)	9216
conv_dw_13_bn (BatchNormali zation)	(None, 7, 7, 1024)	4096
conv_dw_13_relu (ReLU)	(None, 7, 7, 1024)	0
conv_pw_13 (Conv2D)	(None, 7, 7, 1024)	1048576
conv_pw_13_bn (BatchNormali zation)	(None, 7, 7, 1024)	4096
conv_pw_13_relu (ReLU)	(None, 7, 7, 1024)	0
global_average_pooling2d (G lobalAveragePooling2D)	(None, 1, 1, 1024)	0
dropout (Dropout)	(None, 1, 1, 1024)	0
conv_preds (Conv2D)	(None, 1, 1, 1000)	1025000
reshape_2 (Reshape)	(None, 1000)	0
predictions (Activation)	(None, 1000)	0

Total params: 4,253,864
Trainable params: 4,231,976
Non-trainable params: 21,888

Several clues to neural network architecture become evident when examining the above structure.

We will now use the MobileNet to classify several image URLs below. You can add additional URLs of your own to see how well the MobileNet can classify.

In [5]:

```
%matplotlib inline
from PIL import Image, ImageFile
from matplotlib.pyplot import imshow
import requests
import numpy as np
from io import BytesIO
from IPython.display import display, HTML
from tensorflow.keras.applications.mobilenet import decode_predictions

IMAGE_WIDTH = 224
IMAGE_HEIGHT = 224
IMAGE_CHANNELS = 3

ROOT = "https://data.heatonresearch.com/data/t81-558/images/"

def make_square(img):
    cols,rows = img.size

    if rows>cols:
        pad = (rows-cols)/2
```

```
        img = img.crop((pad, 0, cols, cols))
    else:
        pad = (cols - rows) / 2
        img = img.crop((0, pad, rows, rows))

    return img

def classify_image(url):
    x = []
    ImageFile.LOAD_TRUNCATED_IMAGES = False
    response = requests.get(url)
    img = Image.open(BytesIO(response.content))
    img.load()
    img = img.resize((IMAGE_WIDTH, IMAGE_HEIGHT), Image.ANTIALIAS)

    x = image.img_to_array(img)
    x = np.expand_dims(x, axis=0)
    x = preprocess_input(x)
    x = x[:, :, :, :3] # maybe an alpha channel
    pred = model.predict(x)

    display(img)
    print(np.argmax(pred, axis=1))

    lst = decode_predictions(pred, top=5)
    for item in lst[0]:
        print(item)
```

We can now classify an example image. You can specify the URL of any image you wish to classify.

In [6]: `classify_image(ROOT+"soccer_ball.jpg")`



```
[805]
Downloading data from https://storage.googleapis.com/download.tensorflow.org/data/imagenet_class_index.json
40960/35363 [=====] - 0s 0us/step
49152/35363 [=====] - 0s 0us/step
('n04254680', 'soccer_ball', 0.9999938)
('n03530642', 'honeycomb', 3.862412e-06)
('n03255030', 'dumbbell', 4.442458e-07)
('n02782093', 'balloon', 3.7038987e-07)
('n04548280', 'wall_clock', 3.143911e-07)
```

In [7]:

```
classify_image(ROOT+"race_truck.jpg")
```



Overall, the neural network is doing quite well.

For many applications, MobileNet might be entirely acceptable as an image classifier. However, if you need to classify very specialized images, not in the 1,000 image types supported by imagenet, it is necessary to use transfer learning.

Using the Structure of ResNet

We will train a neural network to count the number of paper clips in images. We will make use of the structure of the ResNet neural network. There are several significant changes that we will make to ResNet to apply to this task. First, ResNet is a classifier; we wish to perform a regression to count. Secondly, we want to change the image resolution that ResNet uses. We will not use the weights from ResNet; changing this resolution invalidates the current weights. Thus, it will be necessary to retrain the network.

In [8]:

```
import os
URL = "https://github.com/jeffheaton/data-mirror/"
DOWNLOAD_SOURCE = URL+"releases/download/v1/paperclips.zip"
DOWNLOAD_NAME = DOWNLOAD_SOURCE[DOWNLOAD_SOURCE.rfind('/')+1:]

if COLAB:
    PATH = "/content"
else:
    # I used this locally on my machine, you may need different
    PATH = "/Users/jeff/temp"

EXTRACT_TARGET = os.path.join(PATH, "clips")
SOURCE = os.path.join(EXTRACT_TARGET, "paperclips")
```

```
[751]
('n04037443', 'racer', 0.7131951)
('n03100240', 'convertible', 0.100896776)
('n04285008', 'sports_car', 0.0770768)
('n03930630', 'pickup', 0.02635305)
('n02704792', 'amphibian', 0.011636169)
```

Next, we download the images. This part depends on the origin of your images. The

following code downloads images from a URL, where a ZIP file contains the images. The code unzips the ZIP file.

In [9]:

```
# HIDE OUTPUT
!wget -O {os.path.join(PATH, DOWNLOAD_NAME)} {DOWNLOAD_SOURCE}
!mkdir -p {SOURCE}
!mkdir -p {TARGET}
!mkdir -p {EXTRACT_TARGET}
!unzip -o -j -d {SOURCE} {os.path.join(PATH, DOWNLOAD_NAME)} >/dev/null

--2021-11-28 08:45:31-- https://github.com/jeffheaton/data-mirror/releases/download/v1/paperclips.zip
Resolving github.com (github.com)... 140.82.114.4
Connecting to github.com (github.com)|140.82.114.4|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://objects.githubusercontent.com/github-production-release-asset-2e65be/408419764/25830812-b9e6-4ddf-93b6-7932d9ef5982?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAIWNJYAX4CSVEH53A%2F20211128%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Date=20211128T084531Z&X-Amz-Expires=300&X-Amz-Signature=6db9f72de68b167bd44bfec7661073997b48ed04708a827f50189f622b0395cd&X-Amz-SignedHeaders=host&actor_id=0&key_id=0&repo_id=408419764&response-content-disposition=attachment%3B%20filename%3Dpaperclips.zip&response-content-type=application%2Foctet-stream [following]
--2021-11-28 08:45:31-- https://objects.githubusercontent.com/github-production-release-asset-2e65be/408419764/25830812-b9e6-4ddf-93b6-7932d9ef5982?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAIWNJYAX4CSVEH53A%2F20211128%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Date=20211128T084531Z&X-Amz-Expires=300&X-Amz-Signature=6db9f72de68b167bd44bfec7661073997b48ed04708a827f50189f622b0395cd&X-Amz-SignedHeaders=host&actor_id=0&key_id=0&repo_id=408419764&response-content-disposition=attachment%3B%20filename%3Dpaperclips.zip&response-content-type=application%2Foctet-stream
Resolving objects.githubusercontent.com (objects.githubusercontent.com)... 185.199.108.133, 185.199.109.133, 185.199.110.133, ...
Connecting to objects.githubusercontent.com (objects.githubusercontent.com)|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 163590691 (156M) [application/octet-stream]
Saving to: '/content/paperclips.zip'

/content/paperclips 100%[=====] 156.01M 25.5MB/s   in 5.9s

2021-11-28 08:45:37 (26.6 MB/s) - '/content/paperclips.zip' saved [163590691/163590691]
```

The labels are contained in a CSV file named **train.csv** for the regression. This file has just two labels, **id** and **clip_count**. The ID specifies the filename; for example, row id 1 corresponds to the file **clips-1.jpg**. The following code loads the labels for the training set and creates a new column, named **filename**, that contains the filename of each image, based on the **id** column.

In [11]:

```
df_train = pd.read_csv(os.path.join(SOURCE, "train.csv"))
df_train['filename'] = "clips-" + df_train.id.astype(str) + ".jpg"
```

We want to use early stopping. To do this, we need a validation set. We will break the data into 80 percent test data and 20 validation. Do not confuse this validation

data with the test set provided by Kaggle. This validation set is unique to your program and is for early stopping.

In [12]:

```
TRAIN_PCT = 0.9
TRAIN_CUT = int(len(df_train) * TRAIN_PCT)

df_train_cut = df_train[0:TRAIN_CUT]
df_validate_cut = df_train[TRAIN_CUT:]

print(f"Training size: {len(df_train_cut)}")
print(f"Validate size: {len(df_validate_cut)}")
```

Training size: 18000

Validate size: 2000

Next, we create the generators that will provide the images to the neural network during training. We normalize the images so that the RGB colors between 0-255 become ratios between 0 and 1. We also use the **flow_from_dataframe** generator to connect the Pandas dataframe to the actual image files. We see here a straightforward implementation; you might also wish to use some of the image transformations provided by the data generator.

The **HEIGHT** and **WIDTH** constants specify the dimensions to which the image will be scaled (or expanded). It is probably not a good idea to expand the images.

In [13]:

```
import tensorflow as tf
import keras.preprocessing
from keras.preprocessing import image
from keras.preprocessing.image import ImageDataGenerator

WIDTH = 256
HEIGHT = 256

training_datagen = ImageDataGenerator(
    rescale = 1./255,
    horizontal_flip=True,
    #vertical_flip=True,
    fill_mode='nearest')

train_generator = training_datagen.flow_from_dataframe(
    dataframe=df_train_cut,
    directory=SOURCE,
    x_col="filename",
    y_col="clip_count",
    target_size=(HEIGHT, WIDTH),
    # Keeping the training batch size small
    # USUALLY increases performance
    batch_size=32,
    class_mode='raw')

validation_datagen = ImageDataGenerator(rescale = 1./255)

val_generator = validation_datagen.flow_from_dataframe(
    dataframe=df_validate_cut,
    directory=SOURCE,
    x_col="filename",
    y_col="clip_count",
```

```
target_size=(HEIGHT, WIDTH),  
# Make the validation batch size as large as you  
# have memory for  
batch_size=256,  
class_mode='raw')
```

Found 18000 validated image filenames.

Found 2000 validated image filenames.

We will now use a ResNet neural network as a basis for our neural network. We will redefine both the input shape and output of the ResNet model, so we will not transfer the weights. Since we redefine the input, the weights are of minimal value. We begin by loading, from Keras, the ResNet50 network. We specify **include_top** as False because we will change the input resolution. We also specify **weights** as false because we must retrain the network after changing the top input layers.

In [14]:

```
from tensorflow.keras.applications.resnet50 import ResNet50  
from tensorflow.keras.layers import Input  
  
input_tensor = Input(shape=(HEIGHT, WIDTH, 3))  
  
base_model = ResNet50(  
    include_top=False, weights=None, input_tensor=input_tensor,  
    input_shape=None)
```

Now we must add a few layers to the end of the neural network so that it becomes a regression model.

In [15]:

```
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D  
from tensorflow.keras.models import Model  
  
x=base_model.output  
x=GlobalAveragePooling2D()(x)  
x=Dense(1024,activation='relu')(x)  
x=Dense(1024,activation='relu')(x)  
model=Model(inputs=base_model.input,outputs=Dense(1)(x))
```

We train like before; the only difference is that we do not define the entire neural network here.

In []:

```
from tensorflow.keras.callbacks import EarlyStopping  
from tensorflow.keras.metrics import RootMeanSquaredError  
  
# Important, calculate a valid step size for the validation dataset  
STEP_SIZE_VALID=val_generator.n//val_generator.batch_size  
  
model.compile(loss = 'mean_squared_error', optimizer='adam',  
              metrics=[RootMeanSquaredError(name="rmse")])  
monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3,  
                       patience=50, verbose=1, mode='auto',  
                       restore_best_weights=True)  
  
history = model.fit(train_generator, epochs=100, steps_per_epoch=250,  
                     validation_data = val_generator, callbacks=[monitor]  
                     verbose = 1, validation_steps=STEP_SIZE_VALID)
```

```
Epoch 1/100
250/250 [=====] - 70s 256ms/step - loss: 73.1411
- rmse: 8.5523 - val_loss: 701.4966 - val_rmse: 26.4858
Epoch 2/100
250/250 [=====] - 61s 243ms/step - loss: 27.7530
- rmse: 5.2681 - val_loss: 365.0618 - val_rmse: 19.1066
Epoch 3/100
250/250 [=====] - 61s 243ms/step - loss: 28.6821
- rmse: 5.3556 - val_loss: 130.9240 - val_rmse: 11.4422
Epoch 4/100
250/250 [=====] - 61s 243ms/step - loss: 18.8626
- rmse: 4.3431 - val_loss: 55.8694 - val_rmse: 7.4746
Epoch 5/100
250/250 [=====] - 61s 242ms/step - loss: 14.1957
- rmse: 3.7677 - val_loss: 554.3814 - val_rmse: 23.5453
Epoch 6/100
250/250 [=====] - 61s 242ms/step - loss: 12.8428
- rmse: 3.5837 - val_loss: 79.6855 - val_rmse: 8.9267
Epoch 7/100
250/250 [=====] - 61s 242ms/step - loss: 13.2751
- rmse: 3.6435 - val_loss: 316.9753 - val_rmse: 17.8038
Epoch 8/100
250/250 [=====] - 61s 242ms/step - loss: 11.9826
- rmse: 3.4616 - val_loss: 466.4104 - val_rmse: 21.5965
Epoch 9/100
250/250 [=====] - 61s 243ms/step - loss: 12.0956
- rmse: 3.4779 - val_loss: 4.5767 - val_rmse: 2.1393
Epoch 10/100
250/250 [=====] - 60s 242ms/step - loss: 9.6629 -
rmse: 3.1085 - val_loss: 82.4498 - val_rmse: 9.0802
Epoch 11/100
250/250 [=====] - 60s 242ms/step - loss: 6.0348 -
rmse: 2.4566 - val_loss: 134.9830 - val_rmse: 11.6182
Epoch 12/100
250/250 [=====] - 61s 242ms/step - loss: 9.1004 -
rmse: 3.0167 - val_loss: 13.1667 - val_rmse: 3.6286
Epoch 13/100
250/250 [=====] - 60s 242ms/step - loss: 9.2808 -
rmse: 3.0464 - val_loss: 372.9783 - val_rmse: 19.3126
Epoch 14/100
250/250 [=====] - 61s 242ms/step - loss: 5.7128 -
rmse: 2.3901 - val_loss: 26.7188 - val_rmse: 5.1690
Epoch 15/100
250/250 [=====] - 61s 242ms/step - loss: 5.8171 -
rmse: 2.4119 - val_loss: 15.2567 - val_rmse: 3.9060
Epoch 16/100
250/250 [=====] - 61s 242ms/step - loss: 5.2777 -
rmse: 2.2973 - val_loss: 61.7677 - val_rmse: 7.8592
Epoch 17/100
250/250 [=====] - 61s 242ms/step - loss: 8.9798 -
rmse: 2.9966 - val_loss: 116.6043 - val_rmse: 10.7983
Epoch 18/100
250/250 [=====] - 60s 242ms/step - loss: 6.0367 -
rmse: 2.4570 - val_loss: 11.1855 - val_rmse: 3.3445
Epoch 19/100
250/250 [=====] - 60s 241ms/step - loss: 7.0206 -
rmse: 2.6496 - val_loss: 157.4581 - val_rmse: 12.5482
Epoch 20/100
250/250 [=====] - 60s 240ms/step - loss: 7.4522 -
rmse: 2.7299 - val_loss: 210.9105 - val_rmse: 14.5228
Epoch 21/100
250/250 [=====] - 60s 241ms/step - loss: 10.0948
- rmse: 3.1772 - val_loss: 18.0399 - val_rmse: 4.2473
```

```
Epoch 22/100
250/250 [=====] - 61s 242ms/step - loss: 5.0645 -
rmse: 2.2505 - val_loss: 21.2375 - val_rmse: 4.6084
Epoch 23/100
250/250 [=====] - 60s 241ms/step - loss: 5.7177 -
rmse: 2.3912 - val_loss: 28.5047 - val_rmse: 5.3390
Epoch 24/100
250/250 [=====] - 60s 241ms/step - loss: 5.1064 -
rmse: 2.2597 - val_loss: 47.6090 - val_rmse: 6.8999
Epoch 25/100
250/250 [=====] - 60s 241ms/step - loss: 4.4656 -
rmse: 2.1132 - val_loss: 51.3690 - val_rmse: 7.1672
Epoch 26/100
250/250 [=====] - 60s 240ms/step - loss: 3.6463 -
rmse: 1.9095 - val_loss: 93.4837 - val_rmse: 9.6687
Epoch 27/100
250/250 [=====] - 60s 241ms/step - loss: 3.6216 -
rmse: 1.9031 - val_loss: 49.9860 - val_rmse: 7.0701
Epoch 28/100
250/250 [=====] - 60s 241ms/step - loss: 3.9304 -
rmse: 1.9825 - val_loss: 4.8757 - val_rmse: 2.2081
Epoch 29/100
250/250 [=====] - 60s 240ms/step - loss: 4.6160 -
rmse: 2.1485 - val_loss: 159.4939 - val_rmse: 12.6291
Epoch 30/100
250/250 [=====] - 60s 240ms/step - loss: 5.9745 -
rmse: 2.4443 - val_loss: 31.3900 - val_rmse: 5.6027
Epoch 31/100
250/250 [=====] - 60s 241ms/step - loss: 4.9073 -
rmse: 2.2152 - val_loss: 44.5920 - val_rmse: 6.6777
Epoch 32/100
250/250 [=====] - 60s 241ms/step - loss: 4.4296 -
rmse: 2.1047 - val_loss: 6.8120 - val_rmse: 2.6100
Epoch 33/100
250/250 [=====] - 60s 241ms/step - loss: 6.6059 -
rmse: 2.5702 - val_loss: 103.0320 - val_rmse: 10.1505
Epoch 34/100
250/250 [=====] - 60s 242ms/step - loss: 3.9264 -
rmse: 1.9815 - val_loss: 318.6042 - val_rmse: 17.8495
Epoch 35/100
250/250 [=====] - 61s 242ms/step - loss: 3.7293 -
rmse: 1.9311 - val_loss: 245.8616 - val_rmse: 15.6800
Epoch 36/100
250/250 [=====] - 61s 244ms/step - loss: 3.5809 -
rmse: 1.8923 - val_loss: 3.9251 - val_rmse: 1.9812
Epoch 37/100
250/250 [=====] - 61s 243ms/step - loss: 3.6419 -
rmse: 1.9084 - val_loss: 23.3965 - val_rmse: 4.8370
Epoch 38/100
250/250 [=====] - 61s 243ms/step - loss: 3.6437 -
rmse: 1.9089 - val_loss: 22.4549 - val_rmse: 4.7387
Epoch 39/100
250/250 [=====] - 61s 242ms/step - loss: 3.5197 -
rmse: 1.8761 - val_loss: 103.7435 - val_rmse: 10.1855
Epoch 40/100
250/250 [=====] - 61s 242ms/step - loss: 5.8539 -
rmse: 2.4195 - val_loss: 272.6473 - val_rmse: 16.5120
Epoch 41/100
250/250 [=====] - 61s 242ms/step - loss: 2.8808 -
rmse: 1.6973 - val_loss: 97.9878 - val_rmse: 9.8989
Epoch 42/100
250/250 [=====] - 61s 242ms/step - loss: 3.9501 -
rmse: 1.9875 - val_loss: 237.1111 - val_rmse: 15.3984
```

```
Epoch 43/100
250/250 [=====] - 61s 242ms/step - loss: 5.9793 -
rmse: 2.4453 - val_loss: 102.9308 - val_rmse: 10.1455
Epoch 44/100
250/250 [=====] - 61s 242ms/step - loss: 3.2876 -
rmse: 1.8132 - val_loss: 13.3443 - val_rmse: 3.6530
Epoch 45/100
250/250 [=====] - 61s 243ms/step - loss: 2.8473 -
rmse: 1.6874 - val_loss: 4.4881 - val_rmse: 2.1185
Epoch 46/100
250/250 [=====] - 61s 243ms/step - loss: 2.9382 -
rmse: 1.7141 - val_loss: 13.9019 - val_rmse: 3.7285
Epoch 47/100
250/250 [=====] - 61s 242ms/step - loss: 3.5568 -
rmse: 1.8860 - val_loss: 59.7056 - val_rmse: 7.7269
Epoch 48/100
250/250 [=====] - 61s 243ms/step - loss: 3.0542 -
rmse: 1.7476 - val_loss: 48.3846 - val_rmse: 6.9559
Epoch 49/100
250/250 [=====] - 61s 242ms/step - loss: 4.0696 -
rmse: 2.0173 - val_loss: 21.7313 - val_rmse: 4.6617
Epoch 50/100
250/250 [=====] - 61s 242ms/step - loss: 3.7122 -
rmse: 1.9267 - val_loss: 118.0979 - val_rmse: 10.8673
Epoch 51/100
250/250 [=====] - 61s 242ms/step - loss: 2.6829 -
rmse: 1.6379 - val_loss: 10.7841 - val_rmse: 3.2839
Epoch 52/100
250/250 [=====] - 61s 243ms/step - loss: 2.8031 -
rmse: 1.6742 - val_loss: 13.8609 - val_rmse: 3.7230
Epoch 53/100
250/250 [=====] - 61s 242ms/step - loss: 2.7726 -
rmse: 1.6651 - val_loss: 21.6723 - val_rmse: 4.6553
Epoch 54/100
250/250 [=====] - 61s 243ms/step - loss: 3.4007 -
rmse: 1.8441 - val_loss: 77.9120 - val_rmse: 8.8268
Epoch 55/100
250/250 [=====] - 61s 243ms/step - loss: 3.0227 -
rmse: 1.7386 - val_loss: 17.7745 - val_rmse: 4.2160
Epoch 56/100
250/250 [=====] - 61s 243ms/step - loss: 4.1116 -
rmse: 2.0277 - val_loss: 20.5534 - val_rmse: 4.5336
Epoch 57/100
250/250 [=====] - 61s 243ms/step - loss: 2.5196 -
rmse: 1.5873 - val_loss: 1.6131 - val_rmse: 1.2701
Epoch 58/100
250/250 [=====] - 61s 243ms/step - loss: 3.0357 -
rmse: 1.7423 - val_loss: 72.6971 - val_rmse: 8.5263
Epoch 59/100
250/250 [=====] - 61s 243ms/step - loss: 2.4410 -
rmse: 1.5624 - val_loss: 300.2112 - val_rmse: 17.3266
Epoch 60/100
250/250 [=====] - 61s 242ms/step - loss: 2.2377 -
rmse: 1.4959 - val_loss: 4.8804 - val_rmse: 2.2092
Epoch 61/100
250/250 [=====] - 61s 243ms/step - loss: 2.5355 -
rmse: 1.5923 - val_loss: 3.1464 - val_rmse: 1.7738
Epoch 62/100
250/250 [=====] - 61s 243ms/step - loss: 2.4223 -
rmse: 1.5564 - val_loss: 149.8977 - val_rmse: 12.2433
Epoch 63/100
250/250 [=====] - 61s 243ms/step - loss: 2.3303 -
rmse: 1.5265 - val_loss: 97.8213 - val_rmse: 9.8905
```

```
Epoch 64/100
250/250 [=====] - 61s 242ms/step - loss: 2.6361 -
rmse: 1.6236 - val_loss: 7.0856 - val_rmse: 2.6619
Epoch 65/100
250/250 [=====] - 61s 243ms/step - loss: 2.2068 -
rmse: 1.4855 - val_loss: 42.9824 - val_rmse: 6.5561
Epoch 66/100
250/250 [=====] - 61s 243ms/step - loss: 2.2291 -
rmse: 1.4930 - val_loss: 27.3345 - val_rmse: 5.2282
Epoch 67/100
250/250 [=====] - 61s 242ms/step - loss: 2.2970 -
rmse: 1.5156 - val_loss: 5.9973 - val_rmse: 2.4489
Epoch 68/100
250/250 [=====] - 61s 243ms/step - loss: 2.8215 -
rmse: 1.6797 - val_loss: 12.4237 - val_rmse: 3.5247
Epoch 69/100
250/250 [=====] - 61s 243ms/step - loss: 2.4158 -
rmse: 1.5543 - val_loss: 12.4950 - val_rmse: 3.5348
Epoch 70/100
250/250 [=====] - 61s 243ms/step - loss: 2.4888 -
rmse: 1.5776 - val_loss: 27.5749 - val_rmse: 5.2512
Epoch 71/100
250/250 [=====] - 61s 243ms/step - loss: 1.9211 -
rmse: 1.3860 - val_loss: 17.0489 - val_rmse: 4.1290
Epoch 72/100
250/250 [=====] - 61s 243ms/step - loss: 2.3726 -
rmse: 1.5403 - val_loss: 167.8536 - val_rmse: 12.9558
```

[Open in Colab](#)

T81-558: Applications of Deep Neural Networks

Module 6: Convolutional Neural Networks (CNN) for Computer Vision

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

Module 6 Material

- Part 6.1: Image Processing in Python [\[Video\]](#) [\[Notebook\]](#)
- Part 6.2: Using Convolutional Neural Networks [\[Video\]](#) [\[Notebook\]](#)
- Part 6.3: Using Pretrained Neural Networks with Keras [\[Video\]](#) [\[Notebook\]](#)
- **Part 6.4: Looking at Keras Generators and Image Augmentation** [\[Video\]](#) [\[Notebook\]](#)
- Part 6.5: Recognizing Multiple Images with YOLOv5 [\[Video\]](#) [\[Notebook\]](#)

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  
  
# Nicely formatted time string  
def hms_string(sec_elapsed):  
    h = int(sec_elapsed / (60 * 60))  
    m = int((sec_elapsed % (60 * 60)) / 60)  
    s = sec_elapsed % 60  
    return f"{h}:{m:02}:{s:05.2f}"
```

Note: using Google CoLab

Part 6.4: Inside Augmentation

The [ImageDataGenerator](#) class provides many options for image augmentation.

Deciding which augmentations to use can impact the effectiveness of your model. This part will visualize some of these augmentations that you might use to train your neural network. We begin by loading a sample image to augment.

In [2]:

```
import urllib.request
import shutil
from IPython.display import Image

URL = "https://github.com/jeffheaton/t81_558_deep_learning/" +\
      "blob/master/photos/landscape.jpg?raw=true"
LOCAL_IMG_FILE = "/content/landscape.jpg"

with urllib.request.urlopen(URL) as response, \
     open(LOCAL_IMG_FILE, 'wb') as out_file:
    shutil.copyfileobj(response, out_file)

Image(filename=LOCAL_IMG_FILE)
```

Out[2]:



Next, we introduce a simple utility function to visualize four images sampled from any generator.

In [3]:

```
from numpy import expand_dims
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.preprocessing.image import ImageDataGenerator
from matplotlib import pyplot
import matplotlib.pyplot as plt
import numpy as np
import matplotlib

def visualize_generator(img_file, gen):
```

```
# Load the requested image
img = load_img(img_file)
data = img_to_array(img)
samples = expand_dims(data, 0)

# Generate augmentations from the generator
it = gen.flow(samples, batch_size=1)
images = []
for i in range(4):
    batch = it.next()
    image = batch[0].astype('uint8')
    images.append(image)

images = np.array(images)

# Create a grid of 4 images from the generator
index, height, width, channels = images.shape
nrows = index//2

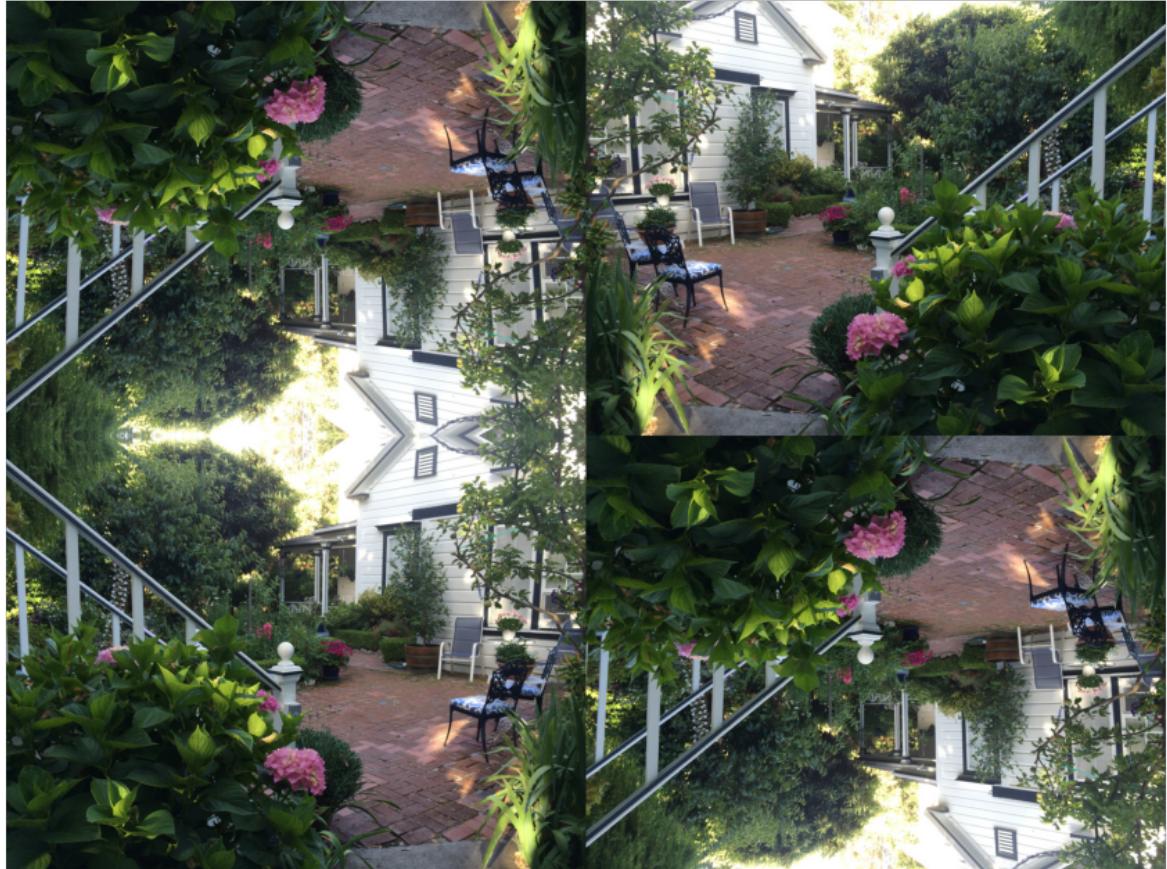
grid = (images.reshape(nrows, 2, height, width, channels)
        .swapaxes(1,2)
        .reshape(height*nrows, width*2, 3))

fig = plt.figure(figsize=(15., 15.))
plt.axis('off')
plt.imshow(grid)
```

We begin by flipping the image. Some images may not make sense to flip, such as this landscape. However, if you expect "noise" in your data where some images may be flipped, then this augmentation may be useful, even if it violates physical reality.

In [4]:

```
visualize_generator(
    LOCAL_IMG_FILE,
    ImageDataGenerator(horizontal_flip=True, vertical_flip=True))
```



Next, we will try moving the image. Notice how part of the image is missing? There are various ways to fill in the missing data, as controlled by **fill_mode**. In this case, we simply use the nearest pixel to fill. It is also possible to rotate images.

In [5]:

```
visualize_generator(  
    LOCAL_IMG_FILE,  
    ImageDataGenerator(width_shift_range=[-200,200],  
        fill_mode='nearest'))
```



We can also adjust brightness.

In [6]:

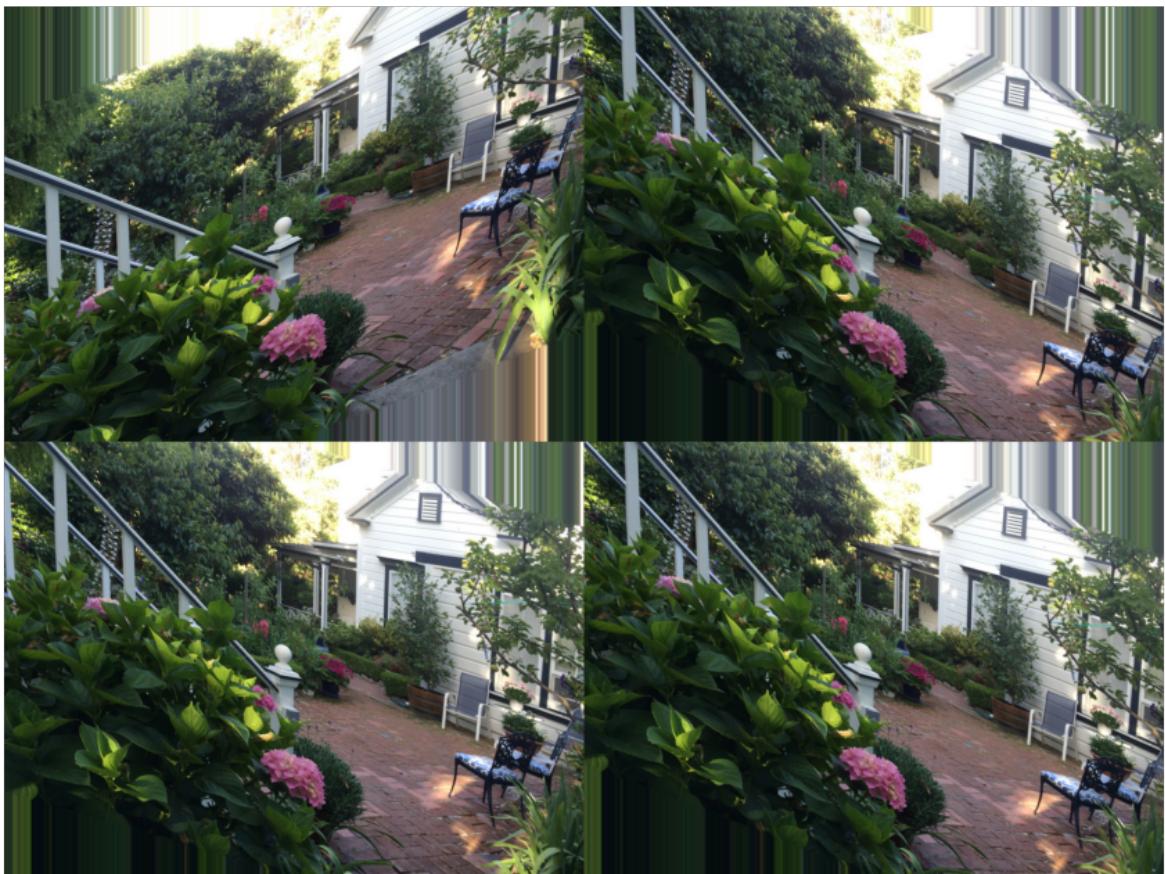
```
visualize_generator(  
    LOCAL_IMG_FILE,  
    ImageDataGenerator(brightness_range=[0,1]))  
  
# brightness_range=None, shear_range=0.0
```



Shearing may not be appropriate for all image types, it stretches the image.

In [7]:

```
visualize_generator(  
    LOCAL_IMG_FILE,  
    ImageDataGenerator(shear_range=30))
```



It is also possible to rotate images.

In [8]:

```
visualize_generator(  
    LOCAL_IMG_FILE,  
    ImageDataGenerator(rotation_range=30))
```



[!\[\]\(b131546fd809d628fa5c470f05cbf31d_img.jpg\) Open in Colab](#)

T81-558: Applications of Deep Neural Networks

Module 6: Convolutional Neural Networks (CNN) for Computer Vision

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

Module 6 Material

- Part 6.1: Image Processing in Python [\[Video\]](#) [\[Notebook\]](#)
- Part 6.2: Using Convolutional Neural Networks [\[Video\]](#) [\[Notebook\]](#)
- Part 6.3: Using Pretrained Neural Networks with Keras [\[Video\]](#) [\[Notebook\]](#)
- Part 6.4: Looking at Keras Generators and Image Augmentation [\[Video\]](#) [\[Notebook\]](#)
- **Part 6.5: Recognizing Multiple Images with YOLOv5** [\[Video\]](#) [\[Notebook\]](#)

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

Note: using Google CoLab
Colab only includes TensorFlow 2.x; %tensorflow_version has no effect.

Part 6.5: Recognizing Multiple Images with YOLO5

Programmers typically design convolutional neural networks to classify a single item centered in an image. However, as humans, we can recognize many items in

our field of view in real-time. It is advantageous to recognize multiple items in a single image. One of the most advanced means of doing this is YOLOv5. You Only Look Once (YOLO) was introduced by Joseph Redmon, who supported YOLO up through V3. [Cite:redmon2016you] The fact that YOLO must only look once speaks to the efficiency of the algorithm. In this context, to "look" means to perform one scan over the image. It is also possible to run YOLO on live video streams.

Joseph Redmon left computer vision to pursue other interests. The current version, YOLOv5 is supported by the startup company [Ultralytics](#), who released the open-source library that we use in this class.[Cite:zhu2021tph]

Researchers have trained YOLO on a variety of different computer image datasets. The version of YOLO weights used in this course is from the dataset Common Objects in Context (COCO). [Cite: lin2014microsoft] This dataset contains images labeled into 80 different classes. COCO is the source of the file coco.txt used in this module.

Using YOLO in Python

To use YOLO in Python, we will use the open-source library provided by Ultralytics.

- [YOLOv5 GitHub](#)

The code provided in this notebook works equally well when run either locally or from Google CoLab. It is easier to run YOLOv5 from CoLab, which is recommended for this course.

We begin by obtaining an image to classify.

In [2]:

```
import urllib.request
import shutil
from IPython.display import Image
!mkdir /content/images/

URL = "https://github.com/jeffheaton/t81_558_deep_learning"
URL += "/raw/master/photos/jeff_cook.jpg"
LOCAL_IMG_FILE = "/content/images/jeff_cook.jpg"

with urllib.request.urlopen(URL) as response, \
    open(LOCAL_IMG_FILE, 'wb') as out_file:
    shutil.copyfileobj(response, out_file)

Image(filename=LOCAL_IMG_FILE)
```

Out[2]:



Installing YOLOv5

YOLO is not available directly through either PIP or CONDA. Additionally, YOLO is not installed in Google CoLab by default. Therefore, whether you wish to use YOLO through CoLab or run it locally, you need to go through several steps to install it. This section describes the process of installing YOLO. The same steps apply to either CoLab or a local install. For CoLab, you must repeat these steps each time the system restarts your virtual environment. You must perform these steps only once for your virtual Python environment for a local install. If you are installing locally, install to the same virtual environment you created for this course. The following commands install YOLO directly from its GitHub repository.

In [3]:

```
import sys

!git clone https://github.com/ultralytics/yolov5 --tag 6.2 # clone
!mv /content/6.2 /content/yolov5
%pip install -qr /content/yolov5/requirements.txt # install
sys.path.insert(0, '/content/yolov5')

import torch
import utils
display = utils.notebook_init() # checks
```

YOLOv5 🚀 v6.2-195-gdf80e7c Python-3.7.14 torch-1.12.1+cu113 CUDA:0 (Tesla T4, 15110MiB)
Setup complete ✅ (2 CPUs, 12.7 GB RAM, 38.8/166.8 GB disk)

Next, we will run YOLO from the command line and classify the previously downloaded kitchen picture. You can run this classification on any image you choose.

In [4]:

```
# Prepare directories for YOLO command line
!rm -R /content/yolov5/runs/detect/*
!mkdir /content/images
```

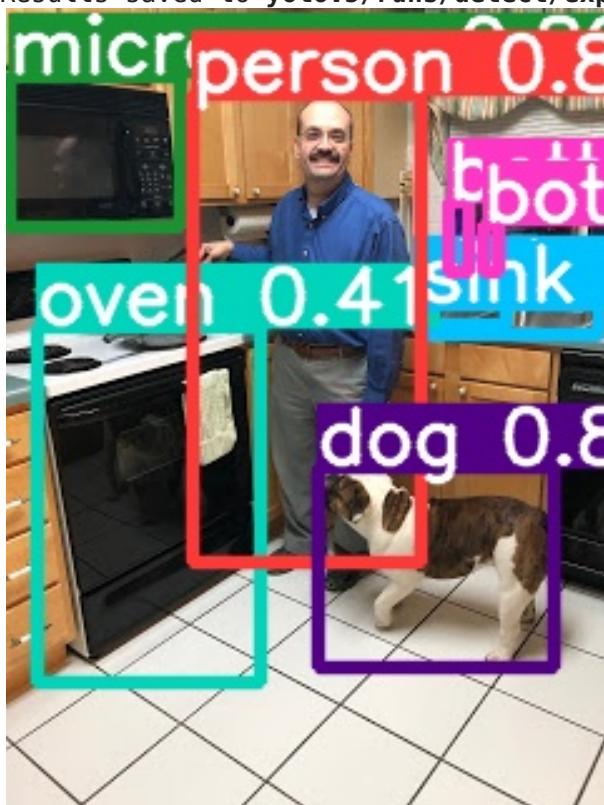
```
!cp /content/street/jeff_cook.jpg /content/images  
  
# Run YOLO to classify  
!python /content/yolov5/detect.py --weights yolov5s.pt --img 1024 \  
--conf 0.25 --source /content/images/  
  
# Display the images  
from IPython.display import Image  
  
URL = '/content/yolov5/runs/detect/exp/jeff_cook.jpg'  
Image(filename=URL, width=300)
```

```
rm: cannot remove '/content/yolov5/runs/detect/*': No such file or directory  
mkdir: cannot create directory '/content/images': File exists  
cp: cannot stat '/content/street/jeff_cook.jpg': No such file or directory  
detect: weights=['yolov5s.pt'], source=/content/images/, data=yolov5/data/  
coco128.yaml, imgsz=[1024, 1024], conf_thres=0.25, iou_thres=0.45, max_det  
=1000, device=, view_img=False, save_txt=False, save_conf=False, save_crop  
=False, nosave=False, classes=None, agnostic_nms=False, augment=False, vis  
ualize=False, update=False, project=yolov5/runs/detect, name=exp, exist_ok  
=False, line_thickness=3, hide_labels=False, hide_conf=False, half=False,  
dnn=False, vid_stride=1  
YOLOv5 🚀 v6.2-195-gdf80e7c Python-3.7.14 torch-1.12.1+cu113 CUDA:0 (Tesla  
T4, 15110MiB)
```

```
Downloading https://github.com/ultralytics/yolov5/releases/download/v6.2/y  
olov5s.pt to yolov5s.pt...  
100% 14.1M/14.1M [00:00<00:00, 236MB/s]
```

```
Fusing layers...  
YOLOv5s summary: 213 layers, 7225885 parameters, 0 gradients  
image 1/1 /content/images/jeff_cook.jpg: 1024x768 1 person, 1 dog, 2 bottl  
es, 1 microwave, 1 oven, 2 sinks, 21.3ms  
Speed: 0.8ms pre-process, 21.3ms inference, 41.7ms NMS per image at shape  
(1, 3, 1024, 1024)  
Results saved to yolov5/runs/detect/exp
```

Out[4]:



In [5]:

```
!ls /content/yolov5/
```

```
benchmarks.py      detect.py      models          runs        tutorial.ipynb
classify          export.py      __pycache__      segment     utils
CONTRIBUTING.md   hubconf.py    README.md       setup.cfg  val.py
data              LICENSE       requirements.txt train.py
```

Running YOLOv5

In addition to the command line execution, we just saw. The following code adds the downloaded YOLOv5 to Python's environment, allowing **yolov5** to be imported like a regular Python library.

In [6]:

```
import torch

# Model
yolo_model = torch.hub.load('ultralytics/yolov5', 'yolov5s') # or yolov5s

# Inference
results = yolo_model(LOCAL_IMG_FILE)

# Results
df = results.pandas().xyxy[0]
df
```

```
/usr/local/lib/python3.7/dist-packages/torch/hub.py:267: UserWarning: You
are about to download and run code from an untrusted repository. In a future
release, this won't be allowed. To add the repository to your trusted list,
change the command to {calling_fn}(..., trust_repo=False) and a command
prompt will appear asking for an explicit confirmation of trust, or load(..., trust_repo=True),
which will assume that the prompt is to be answered with 'yes'. You can also use load(..., trust_repo='check') which will only
prompt for confirmation if the repo is not already trusted. This will
eventually be the default behaviour
```

```
"You are about to download and run code from an untrusted repository. In
a future release, this won't "
Downloading: "https://github.com/ultralytics/yolov5/zipball/master" to /root/.cache/torch/hub/master.zip
YOLOv5 🚀 v6.2-195-gdf80e7c Python-3.7.14 torch-1.12.1+cu113 CUDA:0 (Tesla
T4, 15110MiB)
```

```
Fusing layers...
YOLOv5s summary: 213 layers, 7225885 parameters, 0 gradients
Adding AutoShape...
```

Out[6]:

	xmin	ymin	xmax	ymax	confidence	class	name
0	125.092232	182.010025	219.074036	264.044983	0.928736	16	dog
1	72.338425	36.174423	162.752075	229.957077	0.928245	0	person
2	0.428009	25.537472	68.613434	89.955139	0.891785	68	microwave
3	0.000000	98.033714	103.113159	266.426483	0.739207	69	oven
4	176.110916	76.847527	183.783249	105.030785	0.725925	39	bottle
5	189.972397	85.284508	196.409378	105.729591	0.593492	39	bottle
6	161.864563	115.693741	237.386475	131.211624	0.571422	71	sink
7	216.053223	137.275635	239.968109	230.737457	0.364453	69	oven
8	181.397934	82.266541	195.568832	105.023056	0.252385	39	bottle

It is important to note that the **yolo** class instantiated here is a callable object, which can fill the role of both an object and a function. Acting as a function, *yolo* returns a Pandas dataframe that contains the bounding boxes (xmin/xmax and ymin/ymax), confidence, and name/class of each item detected.

Your program should use these values to perform whatever actions you wish due to the input image. The following code displays the images detected above the threshold.

You can obtain the counts of images through the use of a Pandas groupby and pivot.

In [7]:

```
df2 = df[['name','class']].groupby(by=["name"]).count().reset_index()
df2.columns = ['name','count']
df2['image'] = 1
df2.pivot(index=['image'],columns='name',values='count').reset_index().f...
```

Out[7]:

	name	image	bottle	dog	microwave	oven	person	sink
0		1	3	1		1	2	1

Module 6 Assignment

You can find the first assignment here: [assignment 6](#)

[!\[\]\(298c686dd18c7f35bfe4a9fe554ff145_img.jpg\) Open in Colab](#)

T81-558: Applications of Deep Neural Networks

Module 7: Generative Adversarial Networks

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

Module 7 Material

- **Part 7.1: Introduction to GANs for Image and Data Generation** [\[Video\]](#) [\[Notebook\]](#)
- Part 7.2: Train StyleGAN3 with your Own Images [\[Video\]](#) [\[Notebook\]](#)
- Part 7.3: Exploring the StyleGAN Latent Vector [\[Video\]](#) [\[Notebook\]](#)
- Part 7.4: GANs to Enhance Old Photographs Deoldify [\[Video\]](#) [\[Notebook\]](#)
- Part 7.5: GANs for Tabular Synthetic Data Generation [\[Video\]](#) [\[Notebook\]](#)

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  
    drive.mount('/content/drive', force_remount=True)  
    COLAB = True  
    print("Note: using Google CoLab")  
except:  
    print("Note: not using Google CoLab")  
    COLAB = False
```

Mounted at /content/drive
Note: using Google CoLab

Part 7.1: Introduction to GANS for Image and Data Generation

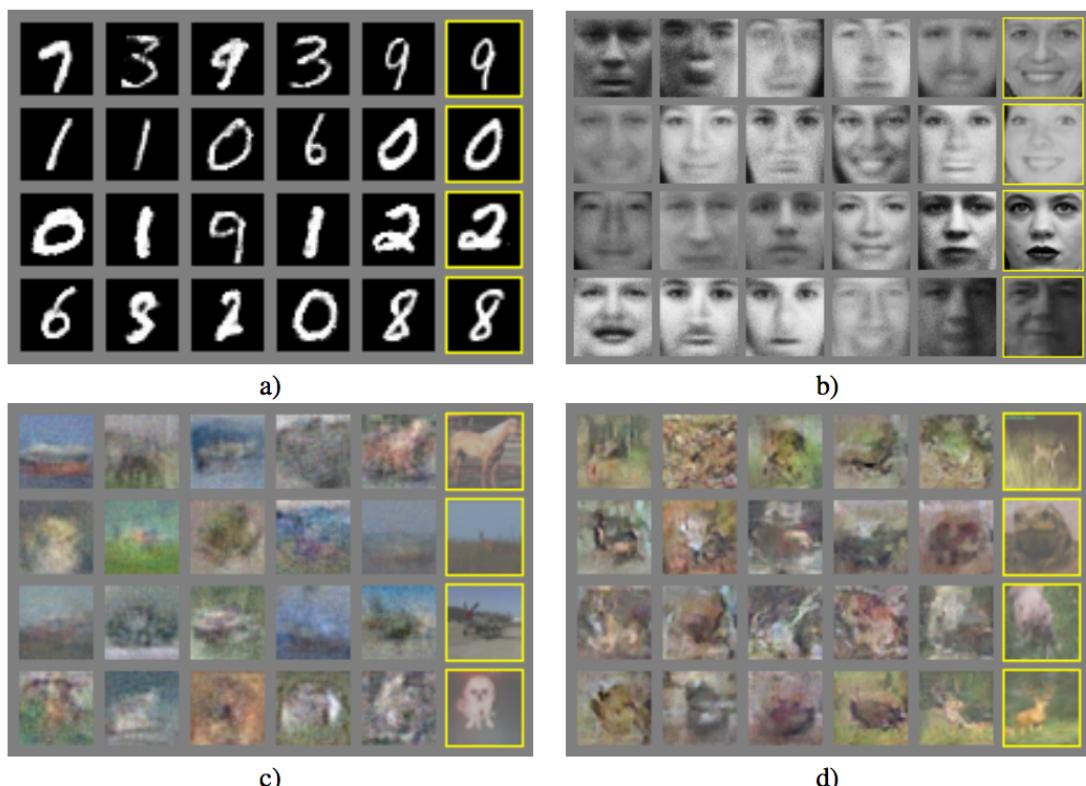
A generative adversarial network (GAN) is a class of machine learning systems invented by Ian Goodfellow in 2014. [\[Cite:goodfellow2014generative\]](#) Two neural networks compete with each other in a game. The GAN training algorithm starts

with a training set and learns to generate new data with the same distributions as the training set. For example, a GAN trained on photographs can generate new photographs that look at least superficially authentic to human observers, having many realistic characteristics.

This chapter makes use of the PyTorch framework rather than Keras/TensorFlow. While there are versions of [StyleGAN2-ADA that work with TensorFlow 1.0](#), NVIDIA has switched to PyTorch for StyleGAN. Running this notebook in this notebook in Google CoLab is the most straightforward means of completing this chapter. Because of this, I designed this notebook to run in Google CoLab. It will take some modifications if you wish to run it locally.

This original StyleGAN paper used neural networks to automatically generate images for several previously seen datasets: MINST and CIFAR. However, it also included the Toronto Face Dataset (a private dataset used by some researchers). You can see some of these images in Figure 7.GANS.

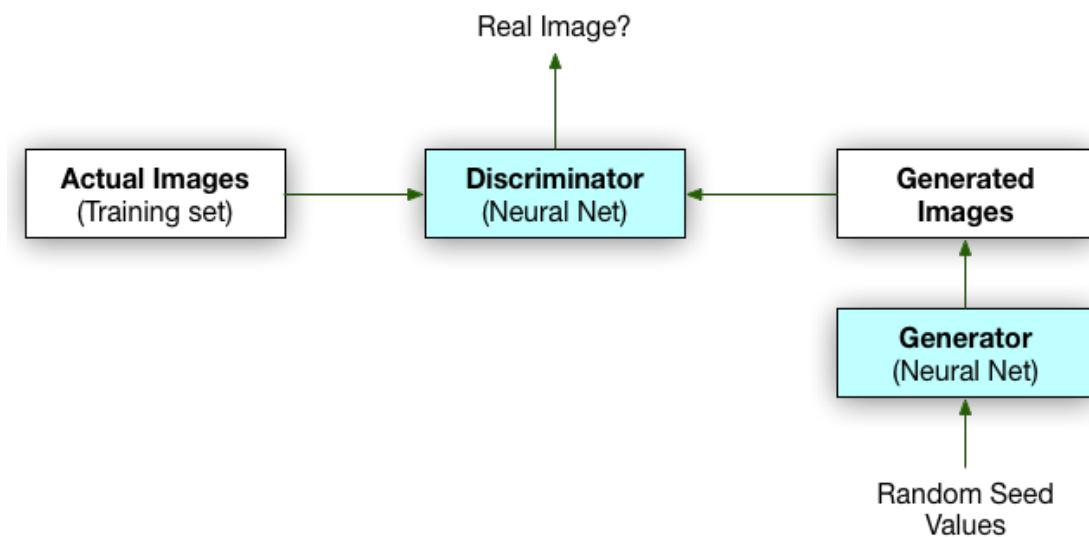
Figure 7.GANS: GAN Generated Images



Only sub-figure D made use of convolutional neural networks. Figures A-C make use of fully connected neural networks. As we will see in this module, the researchers significantly increased the role of convolutional neural networks for GANs.

We call a GAN a generative model because it generates new data. You can see the overall process in Figure 7.GAN-FLOW.

Figure 7.GAN-FLOW: GAN Structure



Face Generation with StyleGAN and Python

GANs have appeared frequently in the media, showcasing their ability to generate highly photorealistic faces. One significant step forward for realistic face generation was the NVIDIA StyleGAN series. NVIDIA introduced the original StyleGAN in 2018. [Cite:karras2019style] StyleGAN was followed by StyleGAN2 in 2019, which improved the quality of StyleGAN by removing certain artifacts. [Cite:karras2019analyzing] Most recently, in 2020, NVIDIA released StyleGAN2 adaptive discriminator augmentation (ADA), which will be the focus of this module. [Cite:karras2020training] We will see both how to train StyleGAN2 ADA on any arbitrary set of images; as well as use pretrained weights provided by NVIDIA. The NVIDIA weights allow us to generate high resolution photorealistic looking faces, such seen in Figure 7. STY-GAN.

Figure 7. STY-GAN: StyleGAN2 Generated Faces

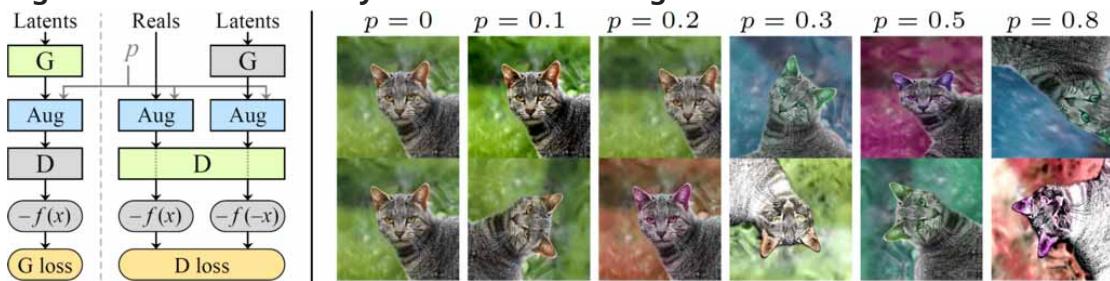


The above images were generated with StyleGAN2, using Google CoLab. Following the instructions in this section, you will be able to create faces like this of your own. StyleGAN2 images are usually 1,024 x 1,024 in resolution. An example of a full-resolution StyleGAN image can be [found here](#).

The primary advancement introduced by the adaptive discriminator augmentation is that the algorithm augments the training images in real-time. Image augmentation is a common technique in many convolution neural network applications. Augmentation has the effect of increasing the size of the training set. Where StyleGAN2 previously required over 30K images for an effective to develop an effective neural network; now much fewer are needed. I used 2K images to train

the fish generating GAN for this section. Figure 7.STY-GAN-ADA demonstrates the ADA process.

Figure 7.STY-GAN-ADA: StyleGAN2 ADA Training

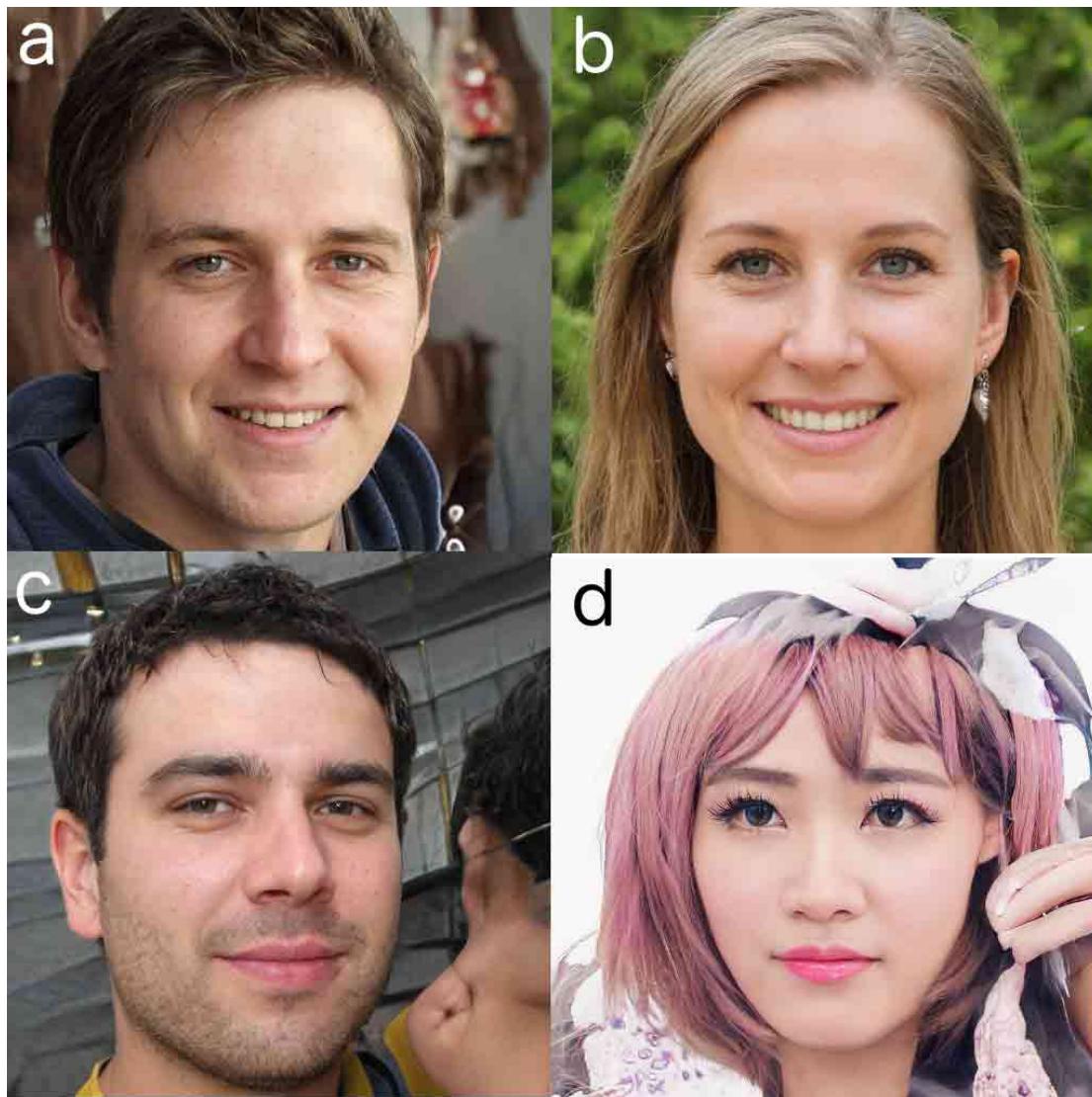


The figure shows the increasing probability of augmentation as p increases. For small image sets, the discriminator will generally memorize the image set unless the training algorithm makes use of augmentation. Once this memorization occurs, the discriminator is no longer providing useful information to the training of the generator.

While the above images look much more realistic than images generated earlier in this course, they are not perfect. Look at Figure 7.STYLEGAN2. There are usually several tell-tail signs that you are looking at a computer-generated image. One of the most obvious is usually the surreal, dream-like backgrounds. The background does not look obviously fake at first glance; however, upon closer inspection, you usually can't quite discern what a GAN-generated background is. Also, look at the image character's left eye. It is slightly unrealistic looking, especially near the eyelashes.

Look at the following GAN face. Can you spot any imperfections?

Figure 7.STYLEGAN2: StyleGAN2 Face



- Image A demonstrates the abstract backgrounds usually associated with a GAN-generated image.
- Image B exhibits issues that earrings often present for GANs. GANs sometimes have problems with symmetry, particularly earrings.
- Image C contains an abstract background and a highly distorted secondary image.
- Image D also contains a highly distorted secondary image that might be a hand.

Several websites allow you to generate GANs of your own without any software.

- [This Person Does not Exist](#)
- [Which Face is Real](#)

The first site generates high-resolution images of human faces. The second site presents a quiz to see if you can detect the difference between a real and fake human face image.

In this chapter, you will learn to create your own StyleGAN pictures using Python.

Generating High Rez GAN Faces with Google

CoLab

This notebook demonstrates how to run NVidia StyleGAN2 ADA inside a Google CoLab notebook. I suggest you use this to generate GAN faces from a pretrained model. If you try to train your own, you will run into compute limitations of Google CoLab. Make sure to run this code on a GPU instance. GPU is assumed.

First, we clone StyleGAN3 from GitHub.

In [2]:

```
# HIDE OUTPUT
!git clone https://github.com/NVlabs/stylegan3.git
!pip install ninja

Cloning into 'stylegan3'...
remote: Enumerating objects: 193, done.
remote: Counting objects: 100% (91/91), done.
remote: Compressing objects: 100% (39/39), done.
remote: Total 193 (delta 55), reused 52 (delta 52), pack-reused 102
Receiving objects: 100% (193/193), 4.17 MiB | 11.50 MiB/s, done.
Resolving deltas: 100% (89/89), done.
Collecting ninja
  Downloading ninja-1.10.2.3-py2.py3-none-manylinux_2_5_x86_64.manylinux1_
x86_64.whl (108 kB)
[██████████] 108 kB 4.3 MB/s
Installing collected packages: ninja
Successfully installed ninja-1.10.2.3
```

Verify that StyleGAN has been cloned.

In [3]:

```
!ls /content/stylegan3

avg_spectra.py Dockerfile gen_video.py metrics train.py
calc_metrics.py docs gui_utils README.md visualizer.py
dataset_tool.py environment.yml legacy.py torch_utils viz
dnnlib           gen_images.py LICENSE.txt training
```

Run StyleGAN From Command Line

Add the StyleGAN folder to Python so that you can import it. I based this code below on code from NVidia for the original StyleGAN paper. When you use StyleGAN you will generally create a GAN from a seed number. This seed is an integer, such as 6600, that will generate a unique image. The seed generates a latent vector containing 512 floating-point values. The GAN code uses the seed to generate these 512 values. The seed value is easier to represent in code than a 512 value vector; however, while a small change to the latent vector results in a slight change to the image, even a small change to the integer seed value will produce a radically different image.

In [4]:

```
# HIDE OUTPUT
URL = "https://api.ngc.nvidia.com/v2/models/nvidia/research/\"\
"stylegan3/versions/1/files/stylegan3-r-ffhq-1024x1024.pkl"

!python /content/stylegan3/gen_images.py \
```

```
--network={URL} \
--outdir=/content/results --seeds=6600-6625
```

```
Loading networks from "https://api.ngc.nvidia.com/v2/models/nvidia/research/stylegan3/versions/1/files/stylegan3-r-ffhq-1024x1024.pkl"...
Downloading https://api.ngc.nvidia.com/v2/models/nvidia/research/stylegan3/versions/1/files/stylegan3-r-ffhq-1024x1024.pkl ... done
Generating image for seed 6600 (0/26) ...
Setting up PyTorch plugin "bias_act_plugin"... Done.
Setting up PyTorch plugin "filtered_lrelu_plugin"... Done.
Generating image for seed 6601 (1/26) ...
Generating image for seed 6602 (2/26) ...
Generating image for seed 6603 (3/26) ...
Generating image for seed 6604 (4/26) ...
Generating image for seed 6605 (5/26) ...
Generating image for seed 6606 (6/26) ...
Generating image for seed 6607 (7/26) ...
Generating image for seed 6608 (8/26) ...
Generating image for seed 6609 (9/26) ...
Generating image for seed 6610 (10/26) ...
Generating image for seed 6611 (11/26) ...
Generating image for seed 6612 (12/26) ...
Generating image for seed 6613 (13/26) ...
Generating image for seed 6614 (14/26) ...
Generating image for seed 6615 (15/26) ...
Generating image for seed 6616 (16/26) ...
Generating image for seed 6617 (17/26) ...
Generating image for seed 6618 (18/26) ...
Generating image for seed 6619 (19/26) ...
Generating image for seed 6620 (20/26) ...
Generating image for seed 6621 (21/26) ...
Generating image for seed 6622 (22/26) ...
Generating image for seed 6623 (23/26) ...
Generating image for seed 6624 (24/26) ...
Generating image for seed 6625 (25/26) ...
```

We can now display the images created.

In [5]:

```
!ls /content/results
```

```
seed6600.png  seed6606.png  seed6612.png  seed6618.png  seed6624.png
seed6601.png  seed6607.png  seed6613.png  seed6619.png  seed6625.png
seed6602.png  seed6608.png  seed6614.png  seed6620.png
seed6603.png  seed6609.png  seed6615.png  seed6621.png
seed6604.png  seed6610.png  seed6616.png  seed6622.png
seed6605.png  seed6611.png  seed6617.png  seed6623.png
```

Next, copy the images to a folder of your choice on GDrive.

In [6]:

```
!cp /content/results/* \
/content/drive/My\ Drive/projects/stylegan3
```

Run StyleGAN From Python Code

Add the StyleGAN folder to Python so that you can import it.

In [7]:

```
import sys
```

```
sys.path.insert(0, "/content/stylegan3")
import pickle
import os
import numpy as np
import PIL.Image
from IPython.display import Image
import matplotlib.pyplot as plt
import IPython.display
import torch
import dnnlib
import legacy

def seed2vec(G, seed):
    return np.random.RandomState(seed).randn(1, G.z_dim)

def display_image(image):
    plt.axis('off')
    plt.imshow(image)
    plt.show()

def generate_image(G, z, truncation_psi):
    # Render images for latents initialized from random seeds.
    Gs_kwargs = {
        'output_transform': dict(func=tflib.convert_images_to_uint8,
                                nchw_to_nhwc=True),
        'randomize_noise': False
    }
    if truncation_psi is not None:
        Gs_kwargs['truncation_psi'] = truncation_psi

    label = np.zeros([1] + G.input_shapes[1][1:])
    # [minibatch, height, width, channel]
    images = G.run(z, label, **Gs_kwargs)
    return images[0]

def get_label(G, device, class_idx):
    label = torch.zeros([1, G.c_dim], device=device)
    if G.c_dim != 0:
        if class_idx is None:
            ctx.fail("Must specify class label with --class when using \"\n"
                    "a conditional network")
            label[:, class_idx] = 1
    else:
        if class_idx is not None:
            print ("warn: --class=lbl ignored when running on \"\n"
                  "an unconditional network")
    return label

def generate_image(device, G, z, truncation_psi=1.0, noise_mode='const',
                  class_idx=None):
    z = torch.from_numpy(z).to(device)
    label = get_label(G, device, class_idx)
    img = G(z, label, truncation_psi=truncation_psi, noise_mode=noise_mode)
    img = (img.permute(0, 2, 3, 1) * 127.5 + 128).clamp(0, 255).to(torch.uint8)
    return PIL.Image.fromarray(img[0].cpu().numpy(), 'RGB')
```

In [8]:

```
#URL = "https://github.com/jeffheaton/pretrained-gan-fish/releases/" \
# "download/1.0.0/fish-gan-2020-12-09.pkl"
#URL = "https://github.com/jeffheaton/pretrained-merry-gan-mas/releases/" \
# "download/v1/christmas-gan-2020-12-03.pkl"
```

```
URL = "https://api.ngc.nvidia.com/v2/models/nvidia/research/stylegan3/"\
      "versions/1/files/stylegan3-r-ffhq-1024x1024.pkl"

print(f'Loading networks from "{URL}"...')
device = torch.device('cuda')
with dnnlib.util.open_url(URL) as f:
    G = legacy.load_network_pkl(f)['G_ema'].to(device) # type: ignore
```

Loading networks from "https://api.ngc.nvidia.com/v2/models/nvidia/research/stylegan3/versions/1/files/stylegan3-r-ffhq-1024x1024.pkl"...

We can now generate images from integer seed codes in Python.

In [9]:

```
# Choose your own starting and ending seed.
SEED_FROM = 1000
SEED_TO = 1003

# Generate the images for the seeds.
for i in range(SEED_FROM, SEED_TO):
    print(f"Seed {i}")
    z = seed2vec(G, i)
    img = generate_image(device, G, z)
    display_image(img)
```

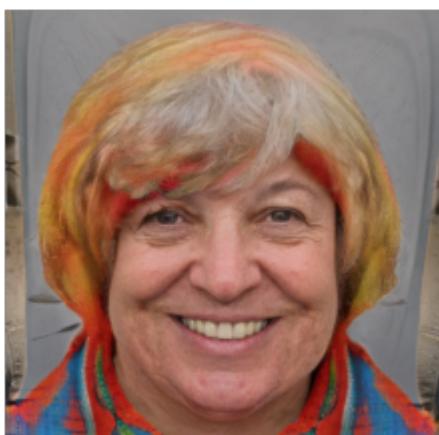
Seed 1000

Setting up PyTorch plugin "bias_act_plugin"... Done.

Setting up PyTorch plugin "filtered_lrelu_plugin"... Done.



Seed 1001



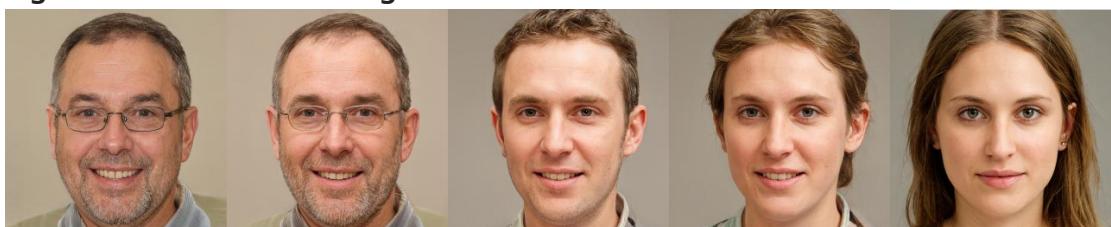
Seed 1002



Examining the Latent Vector

Figure 7.LVEC shows the effects of transforming the latent vector between two images. We accomplish this transformation by slowly moving one 512-value latent vector to another 512 vector. A high-dimension point between two latent vectors will appear similar to both of the two endpoint latent vectors. Images that have similar latent vectors will appear similar to each other.

Figure 7.LVEC: Transforming the Latent Vector



In [10]:

```
def expand_seed(seeds, vector_size):
    result = []

    for seed in seeds:
        rnd = np.random.RandomState(seed)
        result.append( rnd.randn(1, vector_size) )
    return result

#URL = "https://github.com/jeffheaton/pretrained-gan-fish/releases/" \
# "download/1.0.0/fish-gan-2020-12-09.pkl"
#URL = "https://github.com/jeffheaton/pretrained-merry-gan-mas/releases/" \
# "download/v1/christmas-gan-2020-12-03.pkl"
#URL = "https://nvlabs-fi-cdn.nvidia.com/stylegan2-ada/pretrained/ffhq.pkl"
URL = "https://api.ngc.nvidia.com/v2/models/nvidia/research/stylegan3/" \
    "versions/1/files/stylegan3-r-ffhq-1024x1024.pkl"

print(f'Loading networks from "{URL}"...')
device = torch.device('cuda')
with dnnlib.util.open_url(URL) as f:
    G = legacy.load_network_pkl(f)['G_ema'].to(device) # type: ignore

vector_size = G.z_dim
# range(8192,8300)
seeds = expand_seed([8192+1,8192+9], vector_size)
#generate_images(Gs, seeds, truncation_psi=0.5)
print(seeds[0].shape)
```

```
Loading networks from "https://api.ngc.nvidia.com/v2/models/nvidia/research/stylegan3/versions/1/files/stylegan3-r-ffhq-1024x1024.pkl"...
(1, 512)
```

The following code will move between the provided seeds. The constant STEPS specify how many frames there should be between each seed.

In [11]:

```
# HIDE OUTPUT
# Choose your seeds to morph through and the number of steps to
# take to get to each.

SEEDS = [6624,6618,6616] # Better for faces
#SEEDS = [1000,1003,1001] # Better for fish
STEPS = 100

# Remove any prior results
!rm /content/results/*

from tqdm.notebook import tqdm

os.makedirs("./results/", exist_ok=True)

# Generate the images for the video.
idx = 0
for i in range(len(SEEDS)-1):
    v1 = seed2vec(G, SEEDS[i])
    v2 = seed2vec(G, SEEDS[i+1])

    diff = v2 - v1
    step = diff / STEPS
    current = v1.copy()

    for j in tqdm(range(STEPS), desc=f"Seed {SEEDS[i]}"):
```

```
    current = current + step
    img = generate_image(device, G, current)
    img.save(f'./results/frame-{idx}.png')
    idx+=1

# Link the images into a video.
!ffmpeg -r 30 -i /content/results/frame-%d.png -vcodec mpeg4 -y movie.mp4
```

```
Seed 6624:  0%|          | 0/100 [00:00<?, ?it/s]
Seed 6618:  0%|          | 0/100 [00:00<?, ?it/s]
ffmpeg version 3.4.8-0ubuntu0.2 Copyright (c) 2000-2020 the FFmpeg developers
      built with gcc 7 (Ubuntu 7.5.0-3ubuntu1~18.04)
      configuration: --prefix=/usr --extra-version=0ubuntu0.2 --toolchain=hardened --libdir=/usr/lib/x86_64-linux-gnu --incdir=/usr/include/x86_64-linux-gnu --enable-gpl --disable-stripping --enable-avresample --enable-avisynth --enable-gnutls --enable-ladspa --enable-libass --enable-libbluray --enable-libbs2b --enable-libcaca --enable-libcdio --enable-libflite --enable-libfontconfig --enable-libfreetype --enable-libfribidi --enable-libgme --enable-libgsm --enable-libmp3lame --enable-libmysofa --enable-libopenjpeg --enable-libopenmpt --enable-libopus --enable-libpulse --enable-librubberband --enable-librsvg --enable-libshine --enable-libsnappy --enable-libsoxr --enable-libspeex --enable-libssh --enable-libtheora --enable-libtwolame --enable-libvorbis --enable-libvpx --enable-libwavpack --enable-libwebp --enable-libx265 --enable-libxml2 --enable-libxvid --enable-libzmq --enable-libzvbi --enable-omx --enable-openal --enable-opengl --enable-sdl2 --enable-libdc1394 --enable-libdrm --enable-libiec61883 --enable-chromaprint --enable-frei0r --enable-libopencv --enable-libx264 --enable-shared
      libavutil      55. 78.100 / 55. 78.100
      libavcodec     57.107.100 / 57.107.100
      libavformat    57. 83.100 / 57. 83.100
      libavdevice    57. 10.100 / 57. 10.100
      libavfilter     6.107.100 /  6.107.100
      libavresample   3.  7.  0 /  3.  7.  0
      libswscale       4.  8.100 /  4.  8.100
      libswresample   2.  9.100 /  2.  9.100
      libpostproc    54.  7.100 / 54.  7.100
Input #0, image2, from '/content/results/frame-%d.png':
  Duration: 00:00:08.00, start: 0.000000, bitrate: N/A
    Stream #0:0: Video: png, rgb24(pc), 1024x1024, 25 fps, 25 tbr, 25 tbn, 25 tbc
    Stream mapping:
      Stream #0:0 -> #0:0 (png (native) -> mpeg4 (native))
    Press [q] to stop, [?] for help
Output #0, mp4, to 'movie.mp4':
  Metadata:
    encoder         : Lavf57.83.100
  Stream #0:0: Video: mpeg4 (mp4v / 0x7634706D), yuv420p, 1024x1024, q=2-31, 200 kb/s, 30 fps, 15360 tbn, 30 tbc
    Metadata:
      encoder         : Lavc57.107.100 mpeg4
    Side data:
      cpb: bitrate max/min/avg: 0/0/200000 buffer size: 0 vbv_delay: -1
      frame= 200 fps= 43 q=31.0 Lsize= 1161kB time=00:00:06.63 bitrate=143
      3.4kbits/s speed=1.43x
      video:1159kB audio:0kB subtitle:0kB other streams:0kB global headers:0kB muxing overhead: 0.146784%
```

You can now download the generated video.

In [12]:

```
from google.colab import files  
files.download('movie.mp4')
```

Module 7 Assignment

You can find the first assignment here: [assignment 7](#)

[!\[\]\(99466b76c0dc586e961b41bf7eb1d94b_img.jpg\) Open in Colab](#)

T81-558: Applications of Deep Neural Networks

Module 7: Generative Adversarial Networks

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

Module 7 Material

- Part 7.1: Introduction to GANs for Image and Data Generation [\[Video\]](#) [\[Notebook\]](#)
- **Part 7.2: Train StyleGAN3 with your Own Images** [\[Video\]](#) [\[Notebook\]](#)
- Part 7.3: Exploring the StyleGAN Latent Vector [\[Video\]](#) [\[Notebook\]](#)
- Part 7.4: GANs to Enhance Old Photographs Deoldify [\[Video\]](#) [\[Notebook\]](#)
- Part 7.5: GANs for Tabular Synthetic Data Generation [\[Video\]](#) [\[Notebook\]](#)

Google CoLab Instructions

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

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 7.2: Train StyleGAN3 with your Images

Training GANs with StyleGAN is resource-intensive. The NVIDIA StyleGAN researchers used computers with eight high-end GPUs for the high-resolution face GANs trained by NVIDIA. The GPU used by NVIDIA is an A100, which has more memory and cores than the P100 or V100 offered by even Colab Pro+. In this part,

we will use StyleGAN2 to train rather than StyleGAN3. You can use networks trained with StyleGAN2 from StyleGAN3; however, StyleGAN3 usually is more effective at training than StyleGAN2.

Unfortunately, StyleGAN3 is compute-intensive and will perform slowly on any GPU that is not the latest Ampere technology. Because Colab does not provide such technology, I am keeping the training guide at the StyleGAN2 level. Switching to StyleGAN3 is relatively easy, as will be pointed out later.

Make sure that you are running this notebook with a GPU runtime. You can train GANs with either Google Colab Free or Pro. I recommend at least the Pro version due to better GPU instances, longer runtimes, and timeouts. Additionally, the capability of Google Colab Pro to run in the background is valuable when training GANs, as you can close your browser or reboot your laptop while training continues.

You will store your training data and trained neural networks to GDRIVE. For GANs, I lay out my GDRIVE like this:

- ./data/gan/images - RAW images I wish to train on.
- ./data/gan/datasets - Actual training datasets that I convert from the raw images.
- ./data/gan/experiments - The output from StyleGAN2, my image previews, and saved network snapshots.

You will mount the drive at the following location.

```
/content/drive/MyDrive/data
```

What Sort of GPU do you Have?

The type of GPU assigned to you by Colab will significantly affect your training time. Some sample times that I achieved with Colab are given here. I've found that Colab Pro generally starts you with a V100, however, if you run scripts non-stop for 24hrs straight for a few days in a row, you will generally be throttled back to a P100.

- 1024x1024 - V100 - 566 sec/tick (CoLab Pro)
- 1024x1024 - P100 - 1819 sec/tick (CoLab Pro)
- 1024x1024 - T4 - 2188 sec/tick (CoLab Free)

By comparison, a 1024x1024 GAN trained with StyleGAN3 on a V100 is 3087 sec/tick.

If you use Google CoLab Pro, generally, it will not disconnect before 24 hours, even if you (but not your script) are inactive. Free CoLab WILL disconnect a perfectly good running script if you do not interact for a few hours. The following describes how to circumvent this issue.

- [How to prevent Google Colab from disconnecting?](#)

Set Up New Environment

You will likely need to train for >24 hours. Colab will disconnect you. You must be prepared to restart training when this eventually happens. Training is divided into ticks, every so many ticks (50 by default), your neural network is evaluated, and a snapshot is saved. When CoLab shuts down, all training after the last snapshot is lost. It might seem desirable to snapshot after each tick; however, this snapshotting process itself takes nearly an hour. Learning an optimal snapshot size for your resolution and training data is important.

We will mount GDRIVE so that you will save your snapshots there. You must also place your training images in GDRIVE.

You must also install NVIDIA StyleGAN2 ADA PyTorch. We also need to downgrade PyTorch to a version that supports StyleGAN.

In []:

```
!pip uninstall jax jaxlib -y
!pip install "jax[cuda11_cudnn805]==0.3.10" -f https://storage.googleapis.com/jax-releases/jax_cuda11轮子
!pip install torch==1.8.1 torchvision==0.9.1
!git clone https://github.com/NVlabs/stylegan2-ada-pytorch.git
!pip install ninja
```

Find Your Files

The drive is mounted to the following location.

```
/content/drive/MyDrive/data
```

It might be helpful to use an `ls` command to establish the exact path for your images.

In []:

```
!ls /content/drive/MyDrive/data/gan/images
```

Convert Your Images

You must convert your images into a data set form that PyTorch can directly utilize. The following command converts your images and writes the resulting data set to another directory.

In []:

```
CMD = "python /content/stylegan2-ada-pytorch/dataset_tool.py \"\
--source /content/drive/MyDrive/data/gan/images/circuit \"\
--dest /content/drive/MyDrive/data/gan/dataset/circuit"
!{CMD}
```

You can use the following command to clear out the newly created dataset. If

something goes wrong and you need to clean up your images and rerun the above command, you should delete your partially completed dataset directory.

```
In [ ]: #!rm -R /content/drive/MyDrive/data/gan/dataset/circuit/*
```

Clean Up your Images

All images must have the same dimensions and color depth. This code can identify images that have issues.

```
In [ ]:
```

```
from os import listdir
from os.path import isfile, join
import os
from PIL import Image
from tqdm.notebook import tqdm

IMAGE_PATH = '/content/drive/MyDrive/data/gan/images/fish'
files = [f for f in listdir(IMAGE_PATH) if isfile(join(IMAGE_PATH, f))]

base_size = None
for file in tqdm(files):
    file2 = os.path.join(IMAGE_PATH, file)
    img = Image.open(file2)
    sz = img.size
    if base_size and sz!=base_size:
        print(f"Inconsistent size: {file2}")
    elif img.mode!='RGB':
        print(f"Inconsistent color format: {file2}")
    else:
        base_size = sz
```

Perform Initial Training

This code performs the initial training. Set SNAP low enough to get a snapshot before Colab forces you to quit.

```
In [ ]:
```

```
import os

# Modify these to suit your needs
EXPERIMENTS = "/content/drive/MyDrive/data/gan/experiments"
DATA = "/content/drive/MyDrive/data/gan/dataset/circuit"
SNAP = 10

# Build the command and run it
cmd = f"/usr/bin/python3 /content/stylegan2-ada-pytorch/train.py \"\n    --snap {SNAP} --outdir {EXPERIMENTS} --data {DATA}\n!{cmd}
```

Resume Training

You can now resume training after you are interrupted by something in the

pervious step.

In []:

```
import os

# Modify these to suit your needs
EXPERIMENTS = "/content/drive/MyDrive/data/gan/experiments"
NETWORK = "network-snapshot-000100.pkl"
RESUME = os.path.join(EXPERIMENTS, \
                      "00008-circuit-autol-resumecustom", NETWORK)
DATA = "/content/drive/MyDrive/data/gan/dataset/circuit"
SNAP = 10

# Build the command and run it
cmd = f"/usr/bin/python3 /content/stylegan2-ada-pytorch/train.py \"\
      f"--snap {SNAP} --resume {RESUME} --outdir {EXPERIMENTS} --data {DATA}\
      !{cmd}
```

[!\[\]\(f1dfa8502e6410ba89bc57583a22b54f_img.jpg\) Open in Colab](#)

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 7 Material

- Part 7.1: Introduction to GANs for Image and Data Generation [\[Video\]](#) [\[Notebook\]](#)
- Part 7.2: Train StyleGAN3 with your Own Images [\[Video\]](#) [\[Notebook\]](#)
- **Part 7.3: Exploring the StyleGAN Latent Vector** [\[Video\]](#) [\[Notebook\]](#)
- Part 7.4: GANs to Enhance Old Photographs Deoldify [\[Video\]](#) [\[Notebook\]](#)
- Part 7.5: GANs for Tabular Synthetic Data Generation [\[Video\]](#) [\[Notebook\]](#)

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: using Google CoLab

Part 7.3: Exploring the StyleGAN Latent Vector

StyleGAN seeds, such as 3000, are only random number seeds used to generate much longer 512-length latent vectors, which create the GAN image. If you make a small change to the seed, for example, change 3000 to 3001, StyleGAN will create an entirely different picture. However, if you make a small change to a few latent vector values, the image will only change slightly. In this part, we will see how we

can fine-tune the latent vector to control, to some degree, the resulting GAN image appearance.

Installing Needed Software

We begin by installing StyleGAN.

In [2]:

```
# HIDE OUTPUT
!git clone https://github.com/NVlabs/stylegan3.git
!pip install ninja

Cloning into 'stylegan3'...
remote: Enumerating objects: 193, done.
remote: Counting objects: 100% (193/193), done.
remote: Compressing objects: 100% (104/104), done.
remote: Total 193 (delta 90), reused 153 (delta 86), pack-reused 0
Receiving objects: 100% (193/193), 4.17 MiB | 5.24 MiB/s, done.
Resolving deltas: 100% (90/90), done.
Collecting ninja
  Downloading ninja-1.10.2.3-py2.py3-none-manylinux_2_5_x86_64.manylinux1_
x86_64.whl (108 kB)
[██████████] | 108 kB 13.5 MB/s
Installing collected packages: ninja
Successfully installed ninja-1.10.2.3
```

We will use the same functions introduced in the previous part to generate GAN seeds and images.

In [3]:

```
import sys
sys.path.insert(0, "/content/stylegan3")
import pickle
import os
import numpy as np
import PIL.Image
from IPython.display import Image
import matplotlib.pyplot as plt
import IPython.display
import torch
import dnnlib
import legacy

def seed2vec(G, seed):
    return np.random.RandomState(seed).randn(1, G.z_dim)

def display_image(image):
    plt.axis('off')
    plt.imshow(image)
    plt.show()

def generate_image(G, z, truncation_psi):
    # Render images for latents initialized from random seeds.
    Gs_kwargs = {
        'output_transform': dict(func=tflib.convert_images_to_uint8,
                                nchw_to_nhwc=True),
        'randomize_noise': False
    }
    if truncation_psi is not None:
        Gs_kwargs['truncation_psi'] = truncation_psi
```

```

label = np.zeros([1] + G.input_shapes[1][1:])
# [minibatch, height, width, channel]
images = G.run(z, label, **G_kwargs)
return images[0]

def get_label(G, device, class_idx):
    label = torch.zeros([1, G.c_dim], device=device)
    if G.c_dim != 0:
        if class_idx is None:
            ctx.fail('Must specify class label with --class'\
                     'when using a conditional network')
        label[:, class_idx] = 1
    else:
        if class_idx is not None:
            print ('warn: --class=lbl ignored when running'\
                  'on an unconditional network')
    return label

def generate_image(device, G, z, truncation_psi=1.0,
                   noise_mode='const', class_idx=None):
    z = torch.from_numpy(z).to(device)
    label = get_label(G, device, class_idx)
    img = G(z, label, truncation_psi=truncation_psi,
             noise_mode=noise_mode)
    img = (img.permute(0, 2, 3, 1) * 127.5 + 128) \
        .clamp(0, 255).to(torch.uint8)
    return PIL.Image.fromarray(img[0].cpu().numpy(), 'RGB')

```

Next, we load the NVIDIA FFHQ (faces) GAN. We could use any StyleGAN pretrained GAN network here.

In [4]:

```

# HIDE CODE

URL = "https://api.ngc.nvidia.com/v2/models/nvidia/research/"\
      "stylegan3/versions/1/files/stylegan3-r-ffhq-1024x1024.pkl"

print('Loading networks from "%s"...' % URL)
device = torch.device('cuda')
with dnnlib.util.open_url(URL) as fp:
    G = legacy.load_network_pkl(fp)['G_ema']\
        .requires_grad_(False).to(device)

```

```

Loading networks from "https://api.ngc.nvidia.com/v2/models/nvidia/research/stylegan3/versions/1/files/stylegan3-r-ffhq-1024x1024.pkl"...
Downloading https://api.ngc.nvidia.com/v2/models/nvidia/research/stylegan3/versions/1/files/stylegan3-r-ffhq-1024x1024.pkl ... done

```

Generate and View GANS from Seeds

We will begin by generating a few seeds to evaluate potential starting points for our fine-tuning. Try out different seeds ranges until you have a seed that looks close to what you wish to fine-tune.

In [5]:

```

# HIDE OUTPUT 1
# Choose your own starting and ending seed.
SEED_FROM = 4020

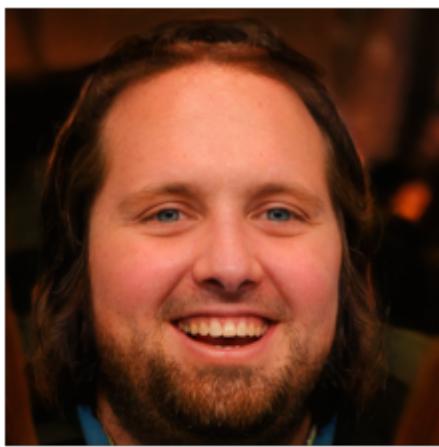
```

```
SEED_T0 = 4023

# Generate the images for the seeds.
for i in range(SEED_FROM, SEED_T0):
    print(f"Seed {i}")
    z = seed2vec(G, i)
    img = generate_image(device, G, z)
    display_image(img)
```

Seed 4020

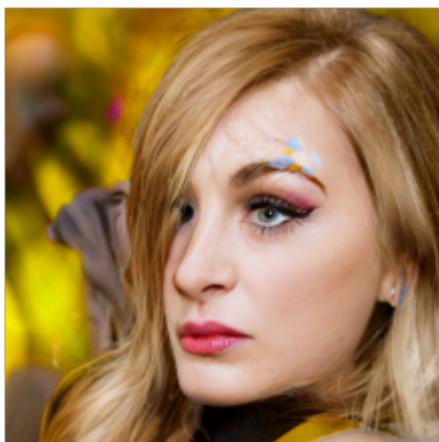
Setting up PyTorch plugin "bias_act_plugin"... Done.
Setting up PyTorch plugin "filtered_lrelu_plugin"... Done.



Seed 4021



Seed 4022



Fine-tune an Image

If you find a seed you like, you can fine-tune it by directly adjusting the latent vector.

First, choose the seed to fine-tune.

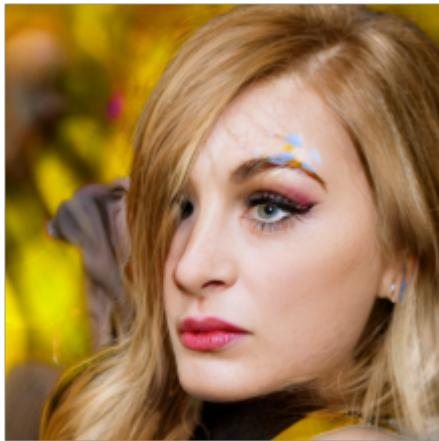
In [6]:

```
START_SEED = 4022  
  
current = seed2vec(G, START_SEED)
```

Next, generate and display the current vector. You will return to this point for each iteration of the finetuning.

In [7]:

```
img = generate_image(device, G, current)  
  
SCALE = 0.5  
display_image(img)
```



Choose an explore size; this is the number of different potential images chosen by moving in 10 different directions. Run this code once and then again anytime you wish to change the ten directions you are exploring. You might change the ten directions if you are no longer seeing improvements.

In [8]:

```
EXPLORE_SIZE = 25  
  
explore = []  
for i in range(EXPLORE_SIZE):  
    explore.append( np.random.rand(1, 512) - 0.5 )
```

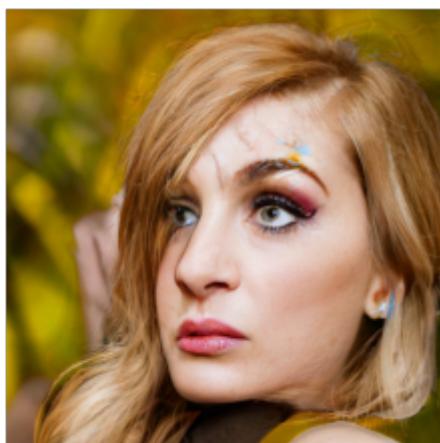
Each image displayed from running this code shows a potential direction that we can move in the latent vector. Choose one image that you like and change MOVE_DIRECTION to indicate this decision. Once you rerun the code, the code will give you a new set of potential directions. Continue this process until you have a latent vector that you like.

In [9]:

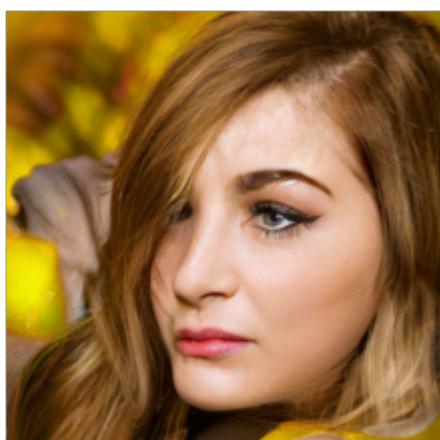
```
# HIDE OUTPUT 1  
# Choose the direction to move. Choose -1 for the initial iteration.  
MOVE_DIRECTION = -1  
SCALE = 0.5  
  
if MOVE_DIRECTION >=0:  
    current = current + explore[MOVE_DIRECTION]
```

```
for i, mv in enumerate(explore):
    print(f"Direction {i}")
    z = current + mv
    img = generate_image(device, G, z)
    display_image(img)
```

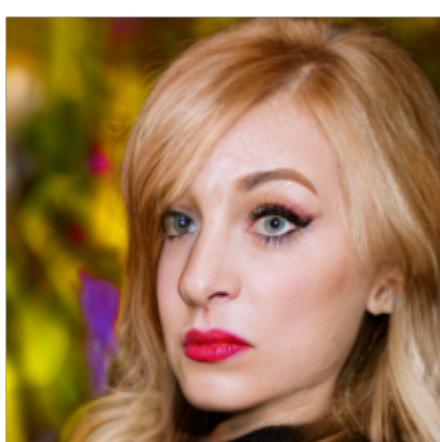
Direction 0



Direction 1



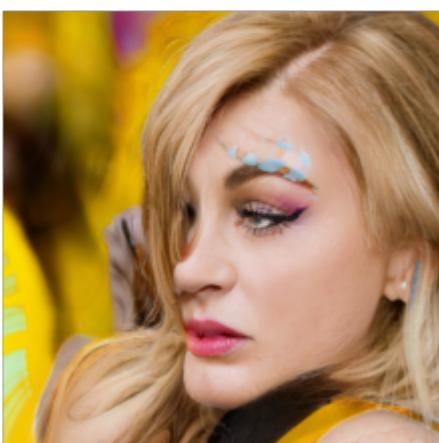
Direction 2



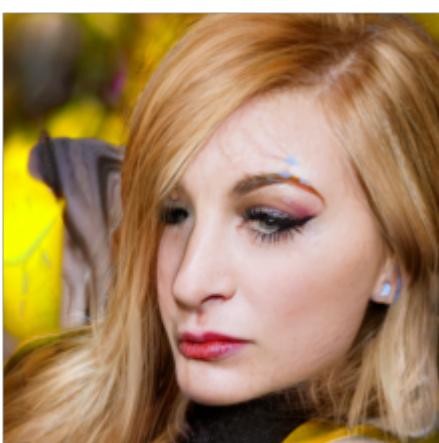
Direction 3



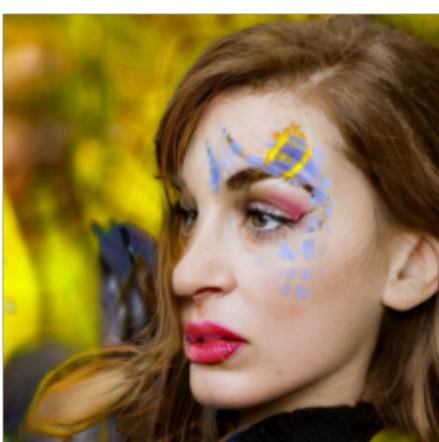
Direction 4



Direction 5



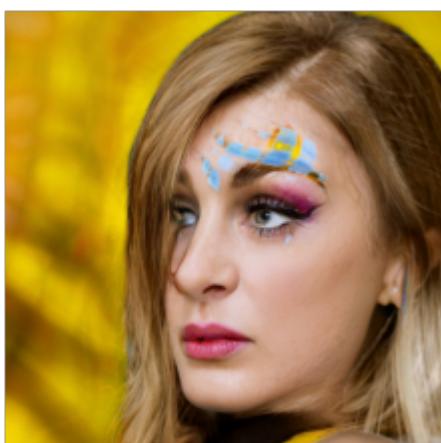
Direction 6



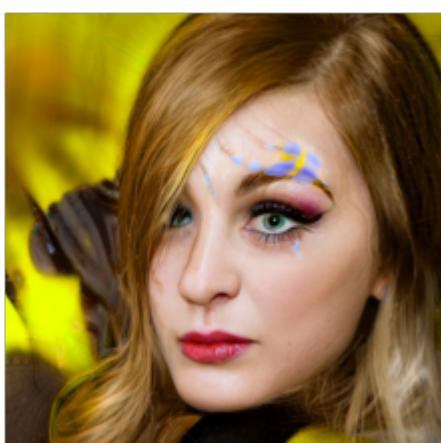
Direction 7



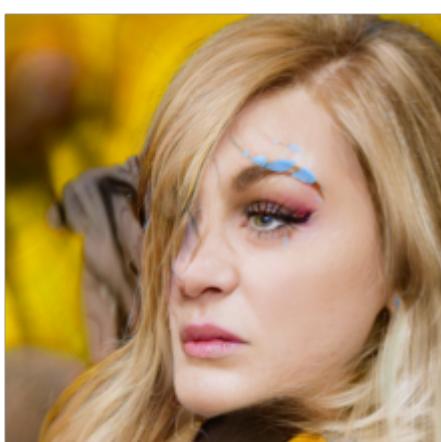
Direction 8



Direction 9



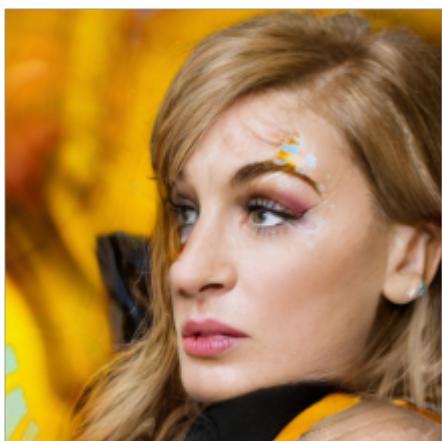
Direction 10



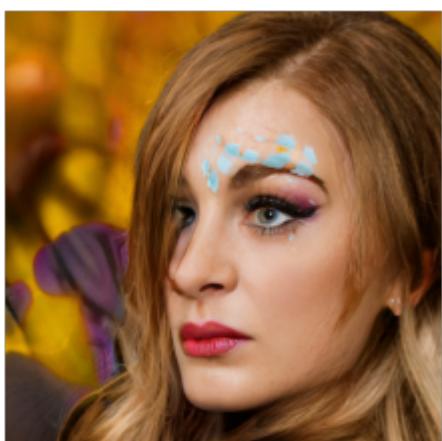
Direction 11



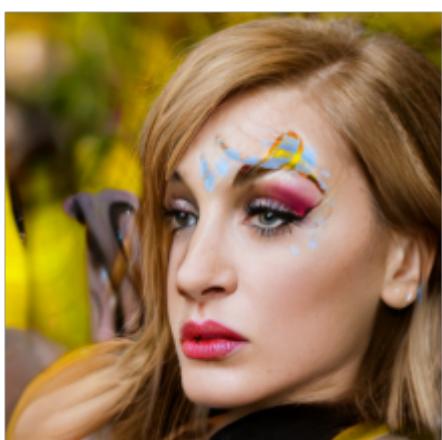
Direction 12



Direction 13



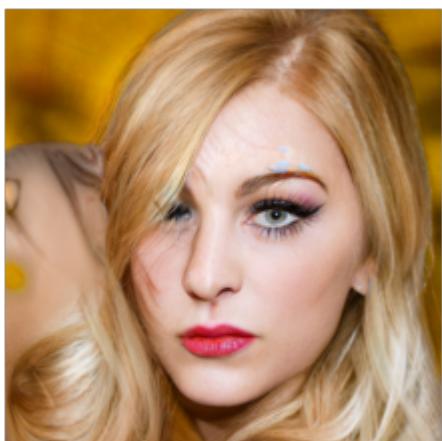
Direction 14



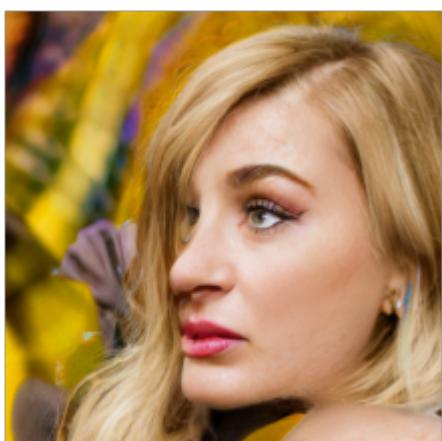
Direction 15



Direction 16



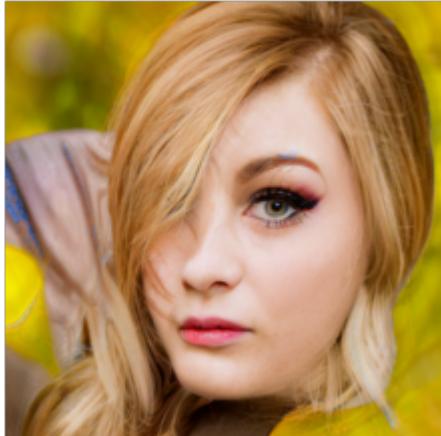
Direction 17



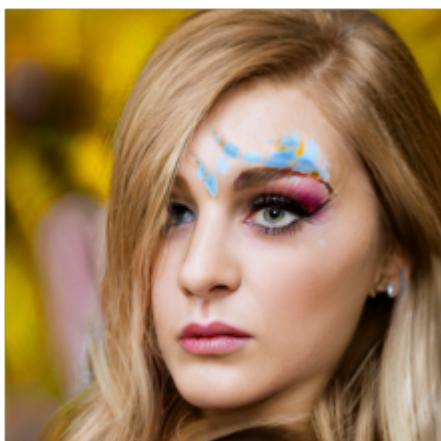
Direction 18



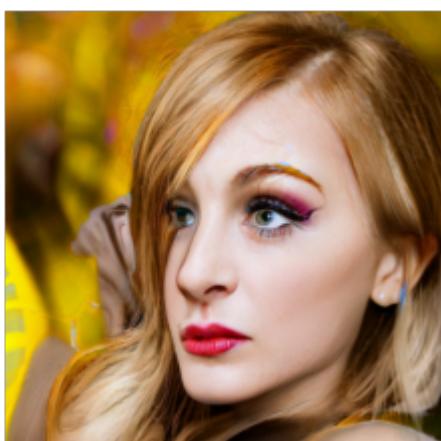
Direction 19



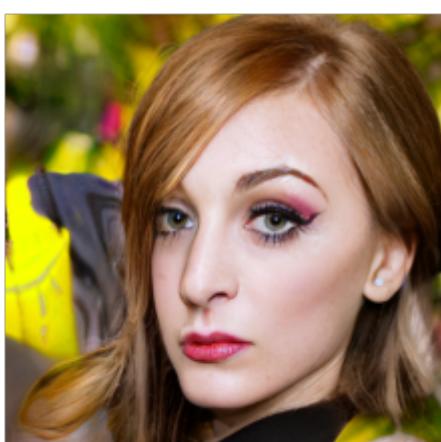
Direction 20



Direction 21



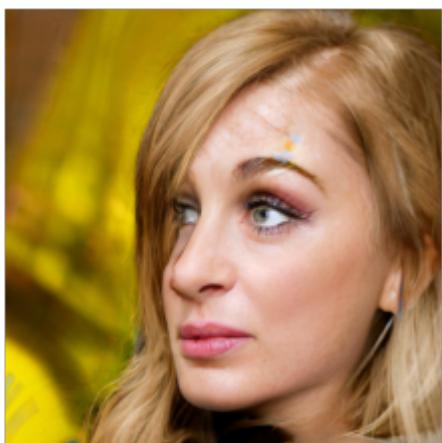
Direction 22



Direction 23



Direction 24



[!\[\]\(fd2a329d3115687029cbb97a83fdc8e6_img.jpg\) Open in Colab](#)

T81-558: Applications of Deep Neural Networks

Module 7: Generative Adversarial Networks

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

Module 7 Material

- Part 7.1: Introduction to GANs for Image and Data Generation [\[Video\]](#) [\[Notebook\]](#)
- Part 7.2: Train StyleGAN3 with your Own Images [\[Video\]](#) [\[Notebook\]](#)
- Part 7.3: Exploring the StyleGAN Latent Vector [\[Video\]](#) [\[Notebook\]](#)
- **Part 7.4: GANs to Enhance Old Photographs Deoldify** [\[Video\]](#) [\[Notebook\]](#)
- Part 7.5: GANs for Tabular Synthetic Data Generation [\[Video\]](#) [\[Notebook\]](#)

Part 7.4: GANS to Enhance Old Photographs Deoldify

For the last two parts of this module, we will examine two applications of GANs. The first application is named [deoldify](#), which uses a PyTorch-based GAN to transform old photographs into more modern-looking images. The complete [source code](#) to Deoldify is provided, along with several examples [notebooks](#) upon which I based this part.

Install Needed Software

We begin by cloning the deoldify repository.

In [1]:

```
# HIDE OUTPUT
!git clone https://github.com/jantic/DeOldify.git DeOldify
%cd DeOldify
```

```
Cloning into 'DeOldify'...
remote: Enumerating objects: 2344, done.
remote: Counting objects: 100% (116/116), done.
remote: Compressing objects: 100% (107/107), done.
remote: Total 2344 (delta 57), reused 29 (delta 9), pack-reused 2228
Receiving objects: 100% (2344/2344), 69.46 MiB | 41.02 MiB/s, done.
Resolving deltas: 100% (1064/1064), done.
/content/DeOldify
```

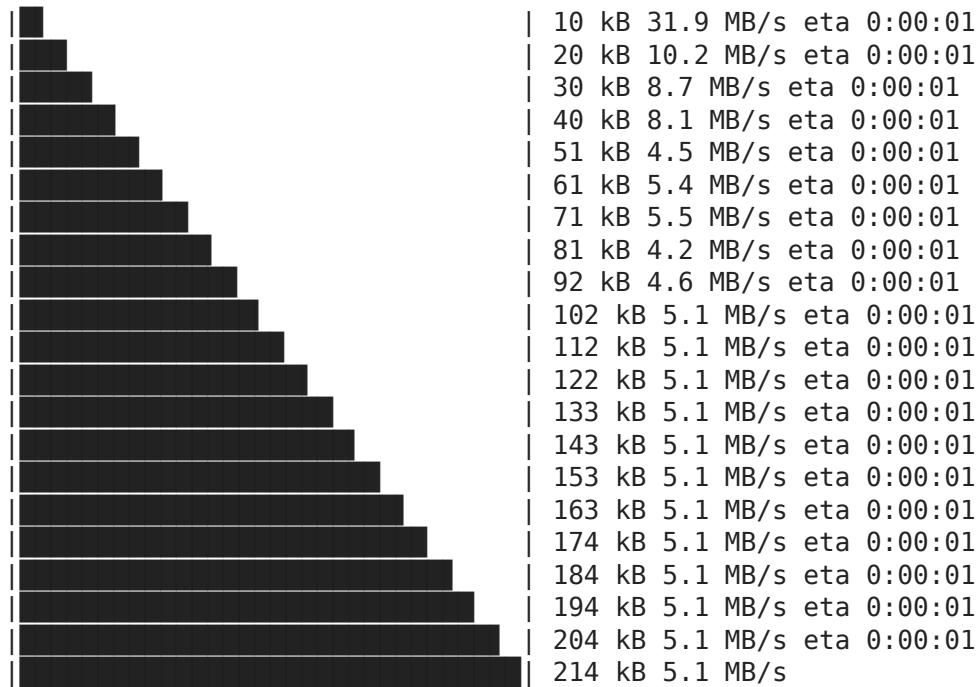
Install any additional Python packages needed.

In [2]:

```
# HIDE OUTPUT
!pip install -r colab_requirements.txt
```

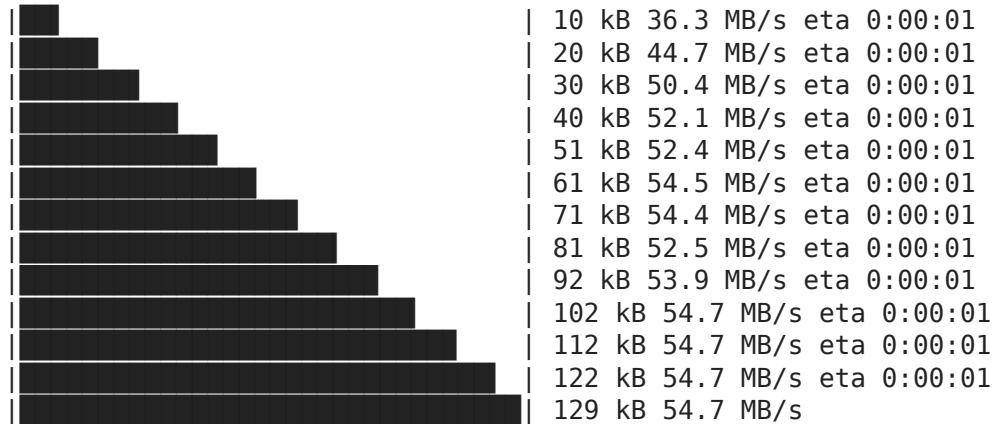
Collecting fastai==1.0.51

Downloading fastai-1.0.51-py3-none-any.whl (214 kB)



Collecting tensorboardX==1.6

Downloading tensorboardX-1.6-py2.py3-none-any.whl (129 kB)



Collecting ffmpeg-python

Downloading ffmpeg_python-0.2.0-py3-none-any.whl (25 kB)

Collecting youtube-dl>=2019.4.17

Downloading youtube_dl-2021.12.17-py2.py3-none-any.whl (1.9 MB)



Requirement already satisfied: opencv-python>=3.3.0.10 in /usr/local/lib/python3.7/dist-packages (from -r colab_requirements.txt (line 5)) (4.1.2.30)

Requirement already satisfied: pillow in /usr/local/lib/python3.7/dist-packages (from -r colab_requirements.txt (line 6)) (7.1.2)

Requirement already satisfied: tornado~>=5.1.0 in /usr/local/lib/python3.7/dist-packages (from -r colab_requirements.txt (line 7)) (5.1.1)

Requirement already satisfied: scipy in /usr/local/lib/python3.7/dist-packages (from fastai==1.0.51->-r colab_requirements.txt (line 1)) (1.4.1)

Requirement already satisfied: pandas in /usr/local/lib/python3.7/dist-packages (from fastai==1.0.51->-r colab_requirements.txt (line 1)) (1.3.5)

Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-packages (from fastai==1.0.51->-r colab_requirements.txt (line 1)) (3.2.2)

Requirement already satisfied: pyyaml in /usr/local/lib/python3.7/dist-packages (from fastai==1.0.51->-r colab_requirements.txt (line 1)) (3.13)

Requirement already satisfied: spacy>=2.0.18 in /usr/local/lib/python3.7/dist-packages (from fastai==1.0.51->-r colab_requirements.txt (line 1))

```
(2.2.4)
Requirement already satisfied: nvidia-ml-py3 in /usr/local/lib/python3.7/dist-packages (from fastai==1.0.51->-r colab_requirements.txt (line 1)) (7.352.0)
Requirement already satisfied: numexpr in /usr/local/lib/python3.7/dist-packages (from fastai==1.0.51->-r colab_requirements.txt (line 1)) (2.8.1)
Requirement already satisfied: beautifulsoup4 in /usr/local/lib/python3.7/dist-packages (from fastai==1.0.51->-r colab_requirements.txt (line 1)) (4.6.3)
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-packages (from fastai==1.0.51->-r colab_requirements.txt (line 1)) (2.23.0)
Requirement already satisfied: torchvision in /usr/local/lib/python3.7/dist-packages (from fastai==1.0.51->-r colab_requirements.txt (line 1)) (0.1.1+cull11)
Requirement already satisfied: torch>=1.0.0 in /usr/local/lib/python3.7/dist-packages (from fastai==1.0.51->-r colab_requirements.txt (line 1)) (1.1.0.0+cull11)
Collecting typing
  Downloading typing-3.7.4.3.tar.gz (78 kB)
    ██████████ | 78 kB 6.8 MB/s
Requirement already satisfied: fastprogress>=0.1.19 in /usr/local/lib/python3.7/dist-packages (from fastai==1.0.51->-r colab_requirements.txt (line 1)) (1.0.2)
Requirement already satisfied: bottleneck in /usr/local/lib/python3.7/dist-packages (from fastai==1.0.51->-r colab_requirements.txt (line 1)) (1.3.4)
Requirement already satisfied: numpy>=1.15 in /usr/local/lib/python3.7/dist-packages (from fastai==1.0.51->-r colab_requirements.txt (line 1)) (1.2.1.5)
Requirement already satisfied: packaging in /usr/local/lib/python3.7/dist-packages (from fastai==1.0.51->-r colab_requirements.txt (line 1)) (21.3)
Requirement already satisfied: six in /usr/local/lib/python3.7/dist-packages (from tensorboardX==1.6->-r colab_requirements.txt (line 2)) (1.15.0)
Requirement already satisfied: protobuf>=3.2.0 in /usr/local/lib/python3.7/dist-packages (from tensorboardX==1.6->-r colab_requirements.txt (line 2)) (3.17.3)
Requirement already satisfied: plac<1.2.0,>=0.9.6 in /usr/local/lib/python3.7/dist-packages (from spacy>=2.0.18->fastai==1.0.51->-r colab_requirements.txt (line 1)) (1.1.3)
Requirement already satisfied: catalogue<1.1.0,>=0.0.7 in /usr/local/lib/python3.7/dist-packages (from spacy>=2.0.18->fastai==1.0.51->-r colab_requirements.txt (line 1)) (1.0.0)
Requirement already satisfied: preshed<3.1.0,>=3.0.2 in /usr/local/lib/python3.7/dist-packages (from spacy>=2.0.18->fastai==1.0.51->-r colab_requirements.txt (line 1)) (3.0.6)
Requirement already satisfied: cymem<2.1.0,>=2.0.2 in /usr/local/lib/python3.7/dist-packages (from spacy>=2.0.18->fastai==1.0.51->-r colab_requirements.txt (line 1)) (2.0.6)
Requirement already satisfied: thinc==7.4.0 in /usr/local/lib/python3.7/dist-packages (from spacy>=2.0.18->fastai==1.0.51->-r colab_requirements.txt (line 1)) (7.4.0)
Requirement already satisfied: wasabi<1.1.0,>=0.4.0 in /usr/local/lib/python3.7/dist-packages (from spacy>=2.0.18->fastai==1.0.51->-r colab_requirements.txt (line 1)) (0.9.0)
Requirement already satisfied: srslly<1.1.0,>=1.0.2 in /usr/local/lib/python3.7/dist-packages (from spacy>=2.0.18->fastai==1.0.51->-r colab_requirements.txt (line 1)) (1.0.5)
Requirement already satisfied: setuptools in /usr/local/lib/python3.7/dist-packages (from spacy>=2.0.18->fastai==1.0.51->-r colab_requirements.txt (line 1)) (57.4.0)
Requirement already satisfied: murmurhash<1.1.0,>=0.28.0 in /usr/local/lib/python3.7/dist-packages (from spacy>=2.0.18->fastai==1.0.51->-r colab_requirements.txt (line 1)) (1.0.6)
```

```
Requirement already satisfied: blis<0.5.0,>=0.4.0 in /usr/local/lib/python3.7/dist-packages (from spacy>=2.0.18->fastai==1.0.51->-r colab_requirements.txt (line 1)) (0.4.1)
Requirement already satisfied: tqdm<5.0.0,>=4.38.0 in /usr/local/lib/python3.7/dist-packages (from spacy>=2.0.18->fastai==1.0.51->-r colab_requirements.txt (line 1)) (4.63.0)
Requirement already satisfied: importlib-metadata>=0.20 in /usr/local/lib/python3.7/dist-packages (from catalogue<1.1.0,>=0.0.7->spacy>=2.0.18->fastai==1.0.51->-r colab_requirements.txt (line 1)) (4.11.3)
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/dist-packages (from importlib-metadata>=0.20->catalogue<1.1.0,>=0.0.7->spacy>=2.0.18->fastai==1.0.51->-r colab_requirements.txt (line 1)) (3.7.0)
Requirement already satisfied: typing-extensions>=3.6.4 in /usr/local/lib/python3.7/dist-packages (from importlib-metadata>=0.20->catalogue<1.1.0,>=0.0.7->spacy>=2.0.18->fastai==1.0.51->-r colab_requirements.txt (line 1)) (3.10.0.2)
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/dist-packages (from requests->fastai==1.0.51->-r colab_requirements.txt (line 1)) (3.0.4)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/dist-packages (from requests->fastai==1.0.51->-r colab_requirements.txt (line 1)) (2021.10.8)
Requirement already satisfied: urllib3!=1.25.0,!>=1.25.1,<1.26,>=1.21.1 in /usr/local/lib/python3.7/dist-packages (from requests->fastai==1.0.51->-r colab_requirements.txt (line 1)) (1.24.3)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-packages (from requests->fastai==1.0.51->-r colab_requirements.txt (line 1)) (2.10)
Requirement already satisfied: future in /usr/local/lib/python3.7/dist-packages (from ffmpeg-python->-r colab_requirements.txt (line 3)) (0.16.0)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dist-packages (from matplotlib->fastai==1.0.51->-r colab_requirements.txt (line 1)) (0.11.0)
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.7/dist-packages (from matplotlib->fastai==1.0.51->-r colab_requirements.txt (line 1)) (2.8.2)
Requirement already satisfied: pyparsing!=2.0.4,!>=2.1.2,!>=2.1.6,>=2.0.1 in /usr/local/lib/python3.7/dist-packages (from matplotlib->fastai==1.0.51->-r colab_requirements.txt (line 1)) (3.0.7)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.7/dist-packages (from matplotlib->fastai==1.0.51->-r colab_requirements.txt (line 1)) (1.4.0)
Requirement already satisfied: pytz>=2017.3 in /usr/local/lib/python3.7/dist-packages (from pandas->fastai==1.0.51->-r colab_requirements.txt (line 1)) (2018.9)
Building wheels for collected packages: typing
  Building wheel for typing (setup.py) ... done
  Created wheel for typing: filename=typing-3.7.4.3-py3-none-any.whl size=26325 sha256=59cd22666e09f1de026cc1027fafc482d06516afe68f1f390a5815150309d329
  Stored in directory: /root/.cache/pip/wheels/35/f3/15/01aa6571f0a72ee6ae7b827c1491c37a1f72d686fd22b43b0e
Successfully built typing
Installing collected packages: typing, youtube-dl, tensorboardX, ffmpeg-python, fastai
  Attempting uninstall: fastai
    Found existing installation: fastai 1.0.61
    Uninstalling fastai-1.0.61:
      Successfully uninstalled fastai-1.0.61
Successfully installed fastai-1.0.51 ffmpeg-python-0.2.0 tensorboardX-1.6 typing-3.7.4.3 youtube-dl-2021.12.17
```

Install the pretrained weights for deoldify.

In [3]:

```
# HIDE OUTPUT
!mkdir './models/'
CMD = "wget https://data.deepai.org/deoldify/ColorizeArtistic_gen.pth"\n    "-O ./models/ColorizeArtistic_gen.pth"
!{CMD}

--2022-04-03 18:32:26--  https://data.deepai.org/deoldify/ColorizeArtistic_gen.pth
Resolving data.deepai.org (data.deepai.org)... 138.201.36.183
Connecting to data.deepai.org (data.deepai.org)|138.201.36.183|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 255144681 (243M) [application/octet-stream]
Saving to: './models/ColorizeArtistic_gen.pth'

./models/ColorizeAr 100%[=====] 243.32M  29.3MB/s   in 8.9s

2022-04-03 18:32:36 (27.4 MB/s) - './models/ColorizeArtistic_gen.pth' saved [255144681/255144681]
```

The authors of deoldify suggest that you might wish to include a watermark to let others know that AI-enhanced this picture. The following code downloads this standard watermark. The authors describe the watermark as follows:

"This places a watermark icon of a palette at the bottom left corner of the image. The authors intend this practice to be a standard way to convey to others viewing the image that AI colorizes it. We want to help promote this as a standard, especially as the technology continues to improve and the distinction between real and fake becomes harder to discern. This palette watermark practice was initiated and led by the MyHeritage in the MyHeritage In Color feature (which uses a newer version of DeOldify than what you're using here)."

In [4]:

```
# HIDE OUTPUT
CMD = "wget https://media.githubusercontent.com/media/jantic/\"\
      \"DeOldify/master/resource_images/watermark.png \"\
      \"-O /content/DeOldify/resource_images/watermark.png"
!{CMD}

--2022-04-03 18:32:36-- https://media.githubusercontent.com/media/jantic/
DeOldify/master/resource_images/watermark.png
Resolving media.githubusercontent.com (media.githubusercontent.com)... 185
.199.108.133, 185.199.109.133, 185.199.110.133, ...
Connecting to media.githubusercontent.com (media.githubusercontent.com)|18
5.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 9210 (9.0K) [image/png]
Saving to: '/content/DeOldify/resource_images/watermark.png'

/content/DeOldify/r 100%[=====] 8.99K --.-KB/s in 0s

2022-04-03 18:32:36 (84.3 MB/s) - '/content/DeOldify/resource_images/water
mark.png' saved [9210/9210]
```

Initialize Torch Device

First, we must initialize a Torch device. If we have a GPU available, we will detect it here. I assume that you will run this code from Google CoLab, with a GPU. It is possible to run this code from a local GPU; however, some modification will be necessary.

In [5]:

```
import sys

#NOTE: This must be the first call in order to work properly!
from deoldify import device
from deoldify.device_id import DeviceId
#choices: CPU, GPU0...GPU7
device.set(device=DeviceId.GPU0)

import torch

if not torch.cuda.is_available():
    print('GPU not available.')
else:
    print('Using GPU.')
```

Using GPU.

We can now call the model. I will enhance an image from my childhood, probably taken in the late 1970s. The picture shows three miniature schnauzers. My childhood dog (Scooby) is on the left, followed by his mom and sister. Overall, a stunning improvement. However, the red in the fire engine riding toy is lost, and the red color of the picnic table where the three dogs were sitting.

In [6]:

```
# HIDE OUTPUT
import fastai
from deoldify.visualize import *
```

```
import warnings
from urllib.parse import urlparse
import os

warnings.filterwarnings("ignore", category=UserWarning,
                      message=".*?Your .*? set is empty.*?")

URL = 'https://raw.githubusercontent.com/jeffheaton/' \
      't81_558_deep_learning/master/photos/scooby_family.jpg'

!wget {URL}

a = urlparse(URL)
before_file = os.path.basename(a.path)

RENDER_FACTOR = 35
WATERMARK = False

colorizer = get_image_colorizer(artistic=True)

after_image = colorizer.get_transformed_image(
    before_file, render_factor=RENDER_FACTOR,
    watermarked=WATERMARK)
#print("Starting image:")
```

```
--2022-04-03 18:32:43-- https://raw.githubusercontent.com/jeffheaton/t81_
558_deep_learning/master/photos/scooby_family.jpg
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199
.109.133, 185.199.111.133, 185.199.108.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.19
9.109.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 43719 (43K) [image/jpeg]
Saving to: 'scooby_family.jpg'
```

```
scooby_family.jpg      0%[                    ]          0  ---KB/s
scooby_family.jpg     100%[=====] 42.69K  ---KB/s    in 0.0
1s
```

```
2022-04-03 18:32:43 (4.19 MB/s) - 'scooby_family.jpg' saved [43719/43719]
```

```
Downloading: "https://download.pytorch.org/models/resnet34-b627a593.pth" t
o /root/.cache/torch/hub/checkpoints/resnet34-b627a593.pth
0%|          | 0.00/83.3M [00:00<?, ?B/s]
```

You can see the starting image here.

In [7]:

```
from IPython import display
display.Image(URL)
```

Out[7]:



You can see the deoldify version here. Please note that these two images will look similar in a black and white book. To see it in color, visit this [link](#).

In [8]:

after_image

Out[8]:



[!\[\]\(c80faf9537134fadfef77c3abdeeb358_img.jpg\) Open in Colab](#)

T81-558: Applications of Deep Neural Networks

Module 7: Generative Adversarial Networks

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

Module 7 Material

- Part 7.1: Introduction to GANs for Image and Data Generation [\[Video\]](#) [\[Notebook\]](#)
- Part 7.2: Train StyleGAN3 with your Own Images [\[Video\]](#) [\[Notebook\]](#)
- Part 7.3: Exploring the StyleGAN Latent Vector [\[Video\]](#) [\[Notebook\]](#)
- Part 7.4: GANs to Enhance Old Photographs Deoldify [\[Video\]](#) [\[Notebook\]](#)
- **Part 7.5: GANs for Tabular Synthetic Data Generation** [\[Video\]](#) [\[Notebook\]](#)

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

Note: using Google CoLab

Part 7.5: GANs for Tabular Synthetic Data Generation

Typically GANs are used to generate images. However, we can also generate tabular data from a GAN. In this part, we will use the Python tabgan utility to create fake data from tabular data. Specifically, we will use the Auto MPG dataset to train a GAN

to generate fake cars. [Cite:ashrapov2020tabular](#)

Installing Tabgan

Pytorch is the foundation of the tabgan neural network utility. The following code installs the needed software to run tabgan in Google Colab.

In [2]:

```
# HIDE OUTPUT
CMD = "wget https://raw.githubusercontent.com/Diyago/\\"\
    "GAN-for-tabular-data/master/requirements.txt"

!{CMD}
!pip install -r requirements.txt
!pip install tabgan
```

```
--2022-04-03 18:53:04-- https://raw.githubusercontent.com/Diyago/GAN-for-tabular-data/master/requirements.txt
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.108.133, 185.199.109.133, 185.199.110.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 197 [text/plain]
Saving to: 'requirements.txt.1'

requirements.txt.1 100%[=====] 197 ---KB/s in 0s

2022-04-03 18:53:04 (8.18 MB/s) - 'requirements.txt.1' saved [197/197]

Requirement already satisfied: scipy==1.4.1 in /usr/local/lib/python3.7/dist-packages (from -r requirements.txt (line 1)) (1.4.1)
Requirement already satisfied: category_encoders==2.1.0 in /usr/local/lib/python3.7/dist-packages (from -r requirements.txt (line 2)) (2.1.0)
Requirement already satisfied: numpy==1.18.1 in /usr/local/lib/python3.7/dist-packages (from -r requirements.txt (line 3)) (1.18.1)
Requirement already satisfied: torch==1.6.0 in /usr/local/lib/python3.7/dist-packages (from -r requirements.txt (line 4)) (1.6.0)
Requirement already satisfied: pandas==1.2.2 in /usr/local/lib/python3.7/dist-packages (from -r requirements.txt (line 5)) (1.2.2)
Requirement already satisfied: lightgbm==2.3.1 in /usr/local/lib/python3.7/dist-packages (from -r requirements.txt (line 6)) (2.3.1)
Requirement already satisfied: scikit_learn==0.23.2 in /usr/local/lib/python3.7/dist-packages (from -r requirements.txt (line 7)) (0.23.2)
Requirement already satisfied: torchvision>=0.4.2 in /usr/local/lib/python3.7/dist-packages (from -r requirements.txt (line 8)) (0.7.0)
Requirement already satisfied: python-dateutil==2.8.1 in /usr/local/lib/python3.7/dist-packages (from -r requirements.txt (line 11)) (2.8.1)
Requirement already satisfied: tqdm==4.61.1 in /usr/local/lib/python3.7/dist-packages (from -r requirements.txt (line 12)) (4.61.1)
Requirement already satisfied: patsy>=0.4.1 in /usr/local/lib/python3.7/dist-packages (from category_encoders==2.1.0->-r requirements.txt (line 2)) (0.5.2)
Requirement already satisfied: statsmodels>=0.6.1 in /usr/local/lib/python3.7/dist-packages (from category_encoders==2.1.0->-r requirements.txt (line 2)) (0.10.2)
Requirement already satisfied: future in /usr/local/lib/python3.7/dist-packages (from torch==1.6.0->-r requirements.txt (line 4)) (0.16.0)
Requirement already satisfied: pytz>=2017.3 in /usr/local/lib/python3.7/dist-packages (from pandas==1.2.2->-r requirements.txt (line 5)) (2018.9)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.7/dist-packages (from scikit_learn==0.23.2->-r requirements.txt (line 7)) (3.1.0)
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.7/dist-packages (from scikit_learn==0.23.2->-r requirements.txt (line 7)) (1.1.0)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-packages (from python-dateutil==2.8.1->-r requirements.txt (line 11)) (1.15.0)
Requirement already satisfied: pillow>=4.1.1 in /usr/local/lib/python3.7/dist-packages (from torchvision>=0.4.2->-r requirements.txt (line 8)) (7.1.2)
Requirement already satisfied: tabgan in /usr/local/lib/python3.7/dist-packages (1.2.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (from tabgan) (1.18.1)
Requirement already satisfied: pandas in /usr/local/lib/python3.7/dist-packages (from tabgan) (1.2.2)
Requirement already satisfied: tqdm in /usr/local/lib/python3.7/dist-pac
```

```
ges (from tabgan) (4.61.1)
Requirement already satisfied: scikit-learn==0.23.2 in /usr/local/lib/python3.7/dist-packages (from tabgan) (0.23.2)
Requirement already satisfied: python-dateutil in /usr/local/lib/python3.7/dist-packages (from tabgan) (2.8.1)
Requirement already satisfied: category-encoders in /usr/local/lib/python3.7/dist-packages (from tabgan) (2.1.0)
Requirement already satisfied: lightgbm in /usr/local/lib/python3.7/dist-packages (from tabgan) (2.3.1)
Requirement already satisfied: torchvision in /usr/local/lib/python3.7/dist-packages (from tabgan) (0.7.0)
Requirement already satisfied: torch in /usr/local/lib/python3.7/dist-packages (from tabgan) (1.6.0)
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.7/dist-packages (from scikit-learn==0.23.2->tabgan) (1.1.0)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.7/dist-packages (from scikit-learn==0.23.2->tabgan) (3.1.0)
Requirement already satisfied: scipy>=0.19.1 in /usr/local/lib/python3.7/dist-packages (from scikit-learn==0.23.2->tabgan) (1.4.1)
Requirement already satisfied: patsy>=0.4.1 in /usr/local/lib/python3.7/dist-packages (from category-encoders->tabgan) (0.5.2)
Requirement already satisfied: statsmodels>=0.6.1 in /usr/local/lib/python3.7/dist-packages (from category-encoders->tabgan) (0.10.2)
Requirement already satisfied: pytz>=2017.3 in /usr/local/lib/python3.7/dist-packages (from pandas->tabgan) (2018.9)
Requirement already satisfied: six in /usr/local/lib/python3.7/dist-packages (from patsy>=0.4.1->category-encoders->tabgan) (1.15.0)
Requirement already satisfied: future in /usr/local/lib/python3.7/dist-packages (from torch->tabgan) (0.16.0)
Requirement already satisfied: pillow>=4.1.1 in /usr/local/lib/python3.7/dist-packages (from torchvision->tabgan) (7.1.2)
```

Note, after installing; you may see this message:

- You must restart the runtime in order to use newly installed versions.

If so, click the "restart runtime" button just under the message. Then rerun this notebook, and you should not receive further issues.

Loading the Auto MPG Data and Training a Neural Network

We will begin by generating fake data for the Auto MPG dataset we have previously seen. The tabgan library can generate categorical (textual) and continuous (numeric) data. However, it cannot generate unstructured data, such as the name of the automobile. Car names, such as "AMC Rebel SST" cannot be replicated by the GAN, because every row has a different car name; it is a textual but non-categorical value.

The following code is similar to what we have seen before. We load the AutoMPG dataset. The tabgan library requires Pandas dataframe to train. Because of this, we keep both the Pandas and Numpy values.

In [3]:

```
# HIDE OUTPUT
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
```

```
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.model_selection import train_test_split
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', '?'])

COLS_USED = ['cylinders', 'displacement', 'horsepower', 'weight',
             'acceleration', 'year', 'origin','mpg']
COLS_TRAIN = ['cylinders', 'displacement', 'horsepower', 'weight',
              'acceleration', 'year', 'origin']

df = df[COLS_USED]

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

# Split into training and test sets
df_x_train, df_x_test, df_y_train, df_y_test = train_test_split(
    df.drop("mpg", axis=1),
    df["mpg"],
    test_size=0.20,
    #shuffle=False,
    random_state=42,
)

# Create dataframe versions for tabular GAN
df_x_test, df_y_test = df_x_test.reset_index(drop=True), \
    df_y_test.reset_index(drop=True)
df_y_train = pd.DataFrame(df_y_train)
df_y_test = pd.DataFrame(df_y_test)

# Pandas to Numpy
x_train = df_x_train.values
x_test = df_x_test.values
y_train = df_y_train.values
y_test = df_y_test.values

# Build the neural network
model = Sequential()
# Hidden 1
model.add(Dense(50, input_dim=x_train.shape[1], activation='relu'))
model.add(Dense(25, activation='relu')) # Hidden 2
model.add(Dense(12, 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)
```

```
Epoch 1/1000
10/10 - 1s - loss: 139176.5625 - val_loss: 40689.0703 - 1s/epoch - 148ms/step
Epoch 2/1000
10/10 - 0s - loss: 19372.2285 - val_loss: 3346.7378 - 108ms/epoch - 11ms/step
Epoch 3/1000
10/10 - 0s - loss: 873.7932 - val_loss: 769.1017 - 109ms/epoch - 11ms/step
Epoch 4/1000
10/10 - 0s - loss: 1485.8730 - val_loss: 1525.9556 - 136ms/epoch - 14ms/step
Epoch 5/1000
10/10 - 0s - loss: 866.6918 - val_loss: 195.6039 - 155ms/epoch - 15ms/step
Epoch 6/1000
10/10 - 0s - loss: 142.9136 - val_loss: 177.2400 - 96ms/epoch - 10ms/step
Epoch 7/1000
10/10 - 0s - loss: 193.9373 - val_loss: 142.7312 - 113ms/epoch - 11ms/step
Epoch 8/1000
10/10 - 0s - loss: 116.1862 - val_loss: 89.0451 - 79ms/epoch - 8ms/step
Epoch 9/1000
10/10 - 0s - loss: 106.6868 - val_loss: 95.9191 - 174ms/epoch - 17ms/step
Epoch 10/1000
10/10 - 0s - loss: 104.5894 - val_loss: 87.7888 - 111ms/epoch - 11ms/step
Epoch 11/1000
10/10 - 0s - loss: 100.0589 - val_loss: 88.2749 - 96ms/epoch - 10ms/step
Epoch 12/1000
10/10 - 0s - loss: 99.6257 - val_loss: 87.3040 - 115ms/epoch - 11ms/step
Epoch 13/1000
10/10 - 0s - loss: 99.4177 - val_loss: 86.6027 - 103ms/epoch - 10ms/step
Epoch 14/1000
10/10 - 0s - loss: 98.5141 - val_loss: 86.1316 - 97ms/epoch - 10ms/step
Epoch 15/1000
10/10 - 0s - loss: 98.0732 - val_loss: 85.9742 - 108ms/epoch - 11ms/step
Epoch 16/1000
10/10 - 0s - loss: 97.4856 - val_loss: 85.1393 - 76ms/epoch - 8ms/step
Epoch 17/1000
10/10 - 0s - loss: 97.1630 - val_loss: 84.7279 - 85ms/epoch - 8ms/step
Epoch 18/1000
10/10 - 0s - loss: 96.7964 - val_loss: 84.2021 - 160ms/epoch - 16ms/step
Epoch 19/1000
10/10 - 0s - loss: 96.3048 - val_loss: 83.9871 - 92ms/epoch - 9ms/step
Epoch 20/1000
10/10 - 0s - loss: 95.4462 - val_loss: 83.0952 - 121ms/epoch - 12ms/step
Epoch 21/1000
10/10 - 0s - loss: 94.9444 - val_loss: 82.6417 - 91ms/epoch - 9ms/step
Epoch 22/1000
10/10 - 0s - loss: 94.3362 - val_loss: 82.0355 - 132ms/epoch - 13ms/step
Epoch 23/1000
10/10 - 0s - loss: 93.8082 - val_loss: 81.6199 - 89ms/epoch - 9ms/step
Epoch 24/1000
10/10 - 0s - loss: 93.2513 - val_loss: 81.1020 - 82ms/epoch - 8ms/step
Epoch 25/1000
10/10 - 0s - loss: 92.6264 - val_loss: 80.3359 - 121ms/epoch - 12ms/step
Epoch 26/1000
10/10 - 0s - loss: 92.2328 - val_loss: 79.8444 - 121ms/epoch - 12ms/step
Epoch 27/1000
10/10 - 0s - loss: 91.4926 - val_loss: 79.1404 - 109ms/epoch - 11ms/step
Epoch 28/1000
10/10 - 0s - loss: 90.7999 - val_loss: 78.6531 - 118ms/epoch - 12ms/step
Epoch 29/1000
10/10 - 0s - loss: 90.1882 - val_loss: 78.1106 - 112ms/epoch - 11ms/step
Epoch 30/1000
10/10 - 0s - loss: 89.8745 - val_loss: 77.8685 - 116ms/epoch - 12ms/step
```

```
Epoch 31/1000
10/10 - 0s - loss: 89.4765 - val_loss: 76.8118 - 94ms/epoch - 9ms/step
Epoch 32/1000
10/10 - 0s - loss: 88.4912 - val_loss: 76.5078 - 87ms/epoch - 9ms/step
Epoch 33/1000
10/10 - 0s - loss: 88.0864 - val_loss: 75.5026 - 102ms/epoch - 10ms/step
Epoch 34/1000
10/10 - 0s - loss: 86.9415 - val_loss: 75.0887 - 90ms/epoch - 9ms/step
Epoch 35/1000
10/10 - 0s - loss: 86.7026 - val_loss: 74.8265 - 129ms/epoch - 13ms/step
Epoch 36/1000
10/10 - 0s - loss: 86.5384 - val_loss: 73.6916 - 97ms/epoch - 10ms/step
Epoch 37/1000
10/10 - 0s - loss: 85.6226 - val_loss: 73.5788 - 105ms/epoch - 11ms/step
Epoch 38/1000
10/10 - 0s - loss: 84.6683 - val_loss: 72.4751 - 91ms/epoch - 9ms/step
Epoch 39/1000
10/10 - 0s - loss: 83.8491 - val_loss: 71.7716 - 90ms/epoch - 9ms/step
Epoch 40/1000
10/10 - 0s - loss: 83.1613 - val_loss: 71.0936 - 119ms/epoch - 12ms/step
Epoch 41/1000
10/10 - 0s - loss: 82.5631 - val_loss: 70.6658 - 89ms/epoch - 9ms/step
Epoch 42/1000
10/10 - 0s - loss: 81.8695 - val_loss: 69.8167 - 163ms/epoch - 16ms/step
Epoch 43/1000
10/10 - 0s - loss: 81.1869 - val_loss: 69.2964 - 91ms/epoch - 9ms/step
Epoch 44/1000
10/10 - 0s - loss: 80.8101 - val_loss: 68.9843 - 88ms/epoch - 9ms/step
Epoch 45/1000
10/10 - 0s - loss: 80.6469 - val_loss: 67.9292 - 143ms/epoch - 14ms/step
Epoch 46/1000
10/10 - 0s - loss: 79.4096 - val_loss: 67.6057 - 102ms/epoch - 10ms/step
Epoch 47/1000
10/10 - 0s - loss: 78.5745 - val_loss: 66.5229 - 69ms/epoch - 7ms/step
Epoch 48/1000
10/10 - 0s - loss: 78.8939 - val_loss: 66.6657 - 95ms/epoch - 10ms/step
Epoch 49/1000
10/10 - 0s - loss: 76.9754 - val_loss: 65.2553 - 106ms/epoch - 11ms/step
Epoch 50/1000
10/10 - 0s - loss: 76.6228 - val_loss: 64.6849 - 81ms/epoch - 8ms/step
Epoch 51/1000
10/10 - 0s - loss: 76.0204 - val_loss: 64.7692 - 120ms/epoch - 12ms/step
Epoch 52/1000
10/10 - 0s - loss: 74.8868 - val_loss: 63.3094 - 119ms/epoch - 12ms/step
Epoch 53/1000
10/10 - 0s - loss: 74.4092 - val_loss: 62.8904 - 136ms/epoch - 14ms/step
Epoch 54/1000
10/10 - 0s - loss: 73.6486 - val_loss: 62.5721 - 98ms/epoch - 10ms/step
Epoch 55/1000
10/10 - 0s - loss: 72.7242 - val_loss: 61.3689 - 131ms/epoch - 13ms/step
Epoch 56/1000
10/10 - 0s - loss: 72.2849 - val_loss: 61.0335 - 120ms/epoch - 12ms/step
Epoch 57/1000
10/10 - 0s - loss: 72.1777 - val_loss: 60.2657 - 163ms/epoch - 16ms/step
Epoch 58/1000
10/10 - 0s - loss: 71.7879 - val_loss: 59.4650 - 127ms/epoch - 13ms/step
Epoch 59/1000
10/10 - 0s - loss: 71.8203 - val_loss: 60.6488 - 83ms/epoch - 8ms/step
Epoch 60/1000
10/10 - 0s - loss: 69.9323 - val_loss: 58.5242 - 135ms/epoch - 13ms/step
Epoch 61/1000
10/10 - 0s - loss: 70.4658 - val_loss: 58.6250 - 153ms/epoch - 15ms/step
Epoch 62/1000
```

```
10/10 - 0s - loss: 68.6058 - val_loss: 57.0953 - 202ms/epoch - 20ms/step
Epoch 63/1000
10/10 - 0s - loss: 67.7657 - val_loss: 56.8579 - 110ms/epoch - 11ms/step
Epoch 64/1000
10/10 - 0s - loss: 67.2709 - val_loss: 56.0743 - 134ms/epoch - 13ms/step
Epoch 65/1000
10/10 - 0s - loss: 66.5735 - val_loss: 55.5872 - 115ms/epoch - 12ms/step
Epoch 66/1000
10/10 - 0s - loss: 66.1336 - val_loss: 54.8934 - 84ms/epoch - 8ms/step
Epoch 67/1000
10/10 - 0s - loss: 65.7582 - val_loss: 54.4984 - 142ms/epoch - 14ms/step
Epoch 68/1000
10/10 - 0s - loss: 65.1338 - val_loss: 53.6615 - 151ms/epoch - 15ms/step
Epoch 69/1000
10/10 - 0s - loss: 63.7764 - val_loss: 54.1908 - 107ms/epoch - 11ms/step
Epoch 70/1000
10/10 - 0s - loss: 63.6102 - val_loss: 52.6200 - 88ms/epoch - 9ms/step
Epoch 71/1000
10/10 - 0s - loss: 62.9163 - val_loss: 52.3956 - 92ms/epoch - 9ms/step
Epoch 72/1000
10/10 - 0s - loss: 62.3272 - val_loss: 51.6602 - 99ms/epoch - 10ms/step
Epoch 73/1000
10/10 - 0s - loss: 63.4992 - val_loss: 51.2628 - 161ms/epoch - 16ms/step
Epoch 74/1000
10/10 - 0s - loss: 62.8709 - val_loss: 50.4873 - 111ms/epoch - 11ms/step
Epoch 75/1000
10/10 - 0s - loss: 60.9686 - val_loss: 51.4177 - 82ms/epoch - 8ms/step
Epoch 76/1000
10/10 - 0s - loss: 59.7037 - val_loss: 49.4762 - 89ms/epoch - 9ms/step
Epoch 77/1000
10/10 - 0s - loss: 59.5827 - val_loss: 49.6083 - 121ms/epoch - 12ms/step
Epoch 78/1000
10/10 - 0s - loss: 59.7795 - val_loss: 48.5477 - 115ms/epoch - 11ms/step
Epoch 79/1000
10/10 - 0s - loss: 58.4787 - val_loss: 48.2142 - 113ms/epoch - 11ms/step
Epoch 80/1000
10/10 - 0s - loss: 58.1287 - val_loss: 48.3080 - 93ms/epoch - 9ms/step
Epoch 81/1000
10/10 - 0s - loss: 57.5325 - val_loss: 47.2556 - 77ms/epoch - 8ms/step
Epoch 82/1000
10/10 - 0s - loss: 56.7754 - val_loss: 47.1473 - 109ms/epoch - 11ms/step
Epoch 83/1000
10/10 - 0s - loss: 56.4003 - val_loss: 46.8065 - 101ms/epoch - 10ms/step
Epoch 84/1000
10/10 - 0s - loss: 56.7061 - val_loss: 45.9990 - 185ms/epoch - 18ms/step
Epoch 85/1000
10/10 - 0s - loss: 56.1603 - val_loss: 45.8791 - 197ms/epoch - 20ms/step
Epoch 86/1000
10/10 - 0s - loss: 55.1062 - val_loss: 45.0677 - 126ms/epoch - 13ms/step
Epoch 87/1000
10/10 - 0s - loss: 54.8889 - val_loss: 44.7583 - 120ms/epoch - 12ms/step
Epoch 88/1000
10/10 - 0s - loss: 54.1313 - val_loss: 44.8168 - 82ms/epoch - 8ms/step
Epoch 89/1000
10/10 - 0s - loss: 53.5392 - val_loss: 44.2517 - 76ms/epoch - 8ms/step
Epoch 90/1000
10/10 - 0s - loss: 53.6703 - val_loss: 44.3647 - 86ms/epoch - 9ms/step
Epoch 91/1000
10/10 - 0s - loss: 53.1194 - val_loss: 43.2339 - 164ms/epoch - 16ms/step
Epoch 92/1000
10/10 - 0s - loss: 52.4162 - val_loss: 43.0597 - 115ms/epoch - 12ms/step
Epoch 93/1000
10/10 - 0s - loss: 52.0441 - val_loss: 42.6480 - 83ms/epoch - 8ms/step
```

```
Epoch 94/1000
10/10 - 0s - loss: 51.5989 - val_loss: 42.4377 - 42ms/epoch - 4ms/step
Epoch 95/1000
10/10 - 0s - loss: 51.3697 - val_loss: 41.9103 - 48ms/epoch - 5ms/step
Epoch 96/1000
10/10 - 0s - loss: 51.3203 - val_loss: 41.6717 - 55ms/epoch - 6ms/step
Epoch 97/1000
10/10 - 0s - loss: 51.1632 - val_loss: 41.1382 - 65ms/epoch - 7ms/step
Epoch 98/1000
10/10 - 0s - loss: 50.2788 - val_loss: 40.7239 - 47ms/epoch - 5ms/step
Epoch 99/1000
10/10 - 0s - loss: 49.7021 - val_loss: 40.8461 - 50ms/epoch - 5ms/step
Epoch 100/1000
10/10 - 0s - loss: 50.6249 - val_loss: 40.8916 - 67ms/epoch - 7ms/step
Epoch 101/1000
10/10 - 0s - loss: 49.9470 - val_loss: 40.2612 - 47ms/epoch - 5ms/step
Epoch 102/1000
10/10 - 0s - loss: 49.3327 - val_loss: 39.3642 - 53ms/epoch - 5ms/step
Epoch 103/1000
10/10 - 0s - loss: 49.4299 - val_loss: 39.4563 - 67ms/epoch - 7ms/step
Epoch 104/1000
10/10 - 0s - loss: 48.4410 - val_loss: 39.5185 - 44ms/epoch - 4ms/step
Epoch 105/1000
10/10 - 0s - loss: 47.7434 - val_loss: 39.1029 - 49ms/epoch - 5ms/step
Epoch 106/1000
10/10 - 0s - loss: 47.3096 - val_loss: 38.3037 - 64ms/epoch - 6ms/step
Epoch 107/1000
10/10 - 0s - loss: 47.3403 - val_loss: 38.1661 - 50ms/epoch - 5ms/step
Epoch 108/1000
10/10 - 0s - loss: 47.3158 - val_loss: 39.3938 - 44ms/epoch - 4ms/step
Epoch 109/1000
10/10 - 0s - loss: 47.2465 - val_loss: 37.4724 - 63ms/epoch - 6ms/step
Epoch 110/1000
10/10 - 0s - loss: 46.1793 - val_loss: 38.2548 - 46ms/epoch - 5ms/step
Epoch 111/1000
10/10 - 0s - loss: 45.9742 - val_loss: 37.9052 - 48ms/epoch - 5ms/step
Epoch 112/1000
10/10 - 0s - loss: 46.8534 - val_loss: 36.6737 - 51ms/epoch - 5ms/step
Epoch 113/1000
10/10 - 0s - loss: 45.6568 - val_loss: 37.2436 - 43ms/epoch - 4ms/step
Epoch 114/1000
10/10 - 0s - loss: 46.1722 - val_loss: 37.9826 - 59ms/epoch - 6ms/step
Epoch 115/1000
10/10 - 0s - loss: 45.0864 - val_loss: 36.1506 - 45ms/epoch - 5ms/step
Epoch 116/1000
10/10 - 0s - loss: 44.5590 - val_loss: 36.2634 - 42ms/epoch - 4ms/step
Epoch 117/1000
10/10 - 0s - loss: 44.0101 - val_loss: 36.1932 - 50ms/epoch - 5ms/step
Epoch 118/1000
10/10 - 0s - loss: 44.5253 - val_loss: 36.1185 - 55ms/epoch - 6ms/step
Epoch 119/1000
10/10 - 0s - loss: 43.6802 - val_loss: 35.3576 - 49ms/epoch - 5ms/step
Epoch 120/1000
10/10 - 0s - loss: 43.8521 - val_loss: 35.2081 - 63ms/epoch - 6ms/step
Epoch 121/1000
10/10 - 0s - loss: 42.8944 - val_loss: 35.2362 - 63ms/epoch - 6ms/step
Epoch 122/1000
10/10 - 0s - loss: 43.0618 - val_loss: 34.6546 - 46ms/epoch - 5ms/step
Epoch 123/1000
10/10 - 0s - loss: 42.5577 - val_loss: 34.5727 - 46ms/epoch - 5ms/step
Epoch 124/1000
10/10 - 0s - loss: 42.0112 - val_loss: 35.2444 - 47ms/epoch - 5ms/step
Epoch 125/1000
```

```
10/10 - 0s - loss: 41.5351 - val_loss: 33.8780 - 50ms/epoch - 5ms/step
Epoch 126/1000
10/10 - 0s - loss: 43.1731 - val_loss: 36.7196 - 42ms/epoch - 4ms/step
Epoch 127/1000
10/10 - 0s - loss: 44.9588 - val_loss: 34.4649 - 67ms/epoch - 7ms/step
Epoch 128/1000
10/10 - 0s - loss: 41.6290 - val_loss: 35.5199 - 74ms/epoch - 7ms/step
Epoch 129/1000
10/10 - 0s - loss: 40.7516 - val_loss: 33.2187 - 43ms/epoch - 4ms/step
Epoch 130/1000
10/10 - 0s - loss: 42.1431 - val_loss: 35.9299 - 65ms/epoch - 6ms/step
Epoch 131/1000
10/10 - 0s - loss: 40.5715 - val_loss: 32.7406 - 45ms/epoch - 4ms/step
Epoch 132/1000
10/10 - 0s - loss: 40.3439 - val_loss: 32.6067 - 71ms/epoch - 7ms/step
Epoch 133/1000
10/10 - 0s - loss: 39.9940 - val_loss: 32.7292 - 47ms/epoch - 5ms/step
Epoch 134/1000
10/10 - 0s - loss: 40.0634 - val_loss: 32.1410 - 48ms/epoch - 5ms/step
Epoch 135/1000
10/10 - 0s - loss: 40.4516 - val_loss: 32.3407 - 69ms/epoch - 7ms/step
Epoch 136/1000
10/10 - 0s - loss: 39.1197 - val_loss: 32.0680 - 61ms/epoch - 6ms/step
Epoch 137/1000
10/10 - 0s - loss: 38.6692 - val_loss: 31.8778 - 51ms/epoch - 5ms/step
Epoch 138/1000
10/10 - 0s - loss: 38.7935 - val_loss: 32.9095 - 59ms/epoch - 6ms/step
Epoch 139/1000
10/10 - 0s - loss: 38.6567 - val_loss: 31.1871 - 54ms/epoch - 5ms/step
Epoch 140/1000
10/10 - 0s - loss: 38.1735 - val_loss: 31.1331 - 63ms/epoch - 6ms/step
Epoch 141/1000
10/10 - 0s - loss: 37.6601 - val_loss: 31.2389 - 44ms/epoch - 4ms/step
Epoch 142/1000
10/10 - 0s - loss: 37.5095 - val_loss: 31.1678 - 62ms/epoch - 6ms/step
Epoch 143/1000
10/10 - 0s - loss: 37.3672 - val_loss: 30.4313 - 61ms/epoch - 6ms/step
Epoch 144/1000
10/10 - 0s - loss: 37.9671 - val_loss: 30.2334 - 50ms/epoch - 5ms/step
Epoch 145/1000
10/10 - 0s - loss: 37.7869 - val_loss: 32.6676 - 61ms/epoch - 6ms/step
Epoch 146/1000
10/10 - 0s - loss: 37.3247 - val_loss: 30.0956 - 49ms/epoch - 5ms/step
Epoch 147/1000
10/10 - 0s - loss: 37.0411 - val_loss: 29.7544 - 63ms/epoch - 6ms/step
Epoch 148/1000
10/10 - 0s - loss: 37.8974 - val_loss: 34.1898 - 55ms/epoch - 6ms/step
Epoch 149/1000
10/10 - 0s - loss: 35.6341 - val_loss: 31.0651 - 50ms/epoch - 5ms/step
Epoch 150/1000
10/10 - 0s - loss: 38.9956 - val_loss: 32.1005 - 52ms/epoch - 5ms/step
Epoch 151/1000
10/10 - 0s - loss: 35.7875 - val_loss: 28.9734 - 65ms/epoch - 7ms/step
Epoch 152/1000
10/10 - 0s - loss: 35.7318 - val_loss: 29.1119 - 48ms/epoch - 5ms/step
Epoch 153/1000
10/10 - 0s - loss: 35.2600 - val_loss: 28.6848 - 65ms/epoch - 6ms/step
Epoch 154/1000
10/10 - 0s - loss: 35.9957 - val_loss: 29.1977 - 42ms/epoch - 4ms/step
Epoch 155/1000
10/10 - 0s - loss: 35.7540 - val_loss: 29.7204 - 72ms/epoch - 7ms/step
Epoch 156/1000
10/10 - 0s - loss: 34.8676 - val_loss: 28.1050 - 44ms/epoch - 4ms/step
```

```
Epoch 157/1000
10/10 - 0s - loss: 34.6044 - val_loss: 29.6049 - 49ms/epoch - 5ms/step
Epoch 158/1000
10/10 - 0s - loss: 34.8734 - val_loss: 27.8684 - 49ms/epoch - 5ms/step
Epoch 159/1000
10/10 - 0s - loss: 34.2168 - val_loss: 27.5564 - 61ms/epoch - 6ms/step
Epoch 160/1000
10/10 - 0s - loss: 34.3384 - val_loss: 27.3708 - 64ms/epoch - 6ms/step
Epoch 161/1000
10/10 - 0s - loss: 33.9496 - val_loss: 28.5652 - 47ms/epoch - 5ms/step
Epoch 162/1000
10/10 - 0s - loss: 33.4599 - val_loss: 27.3174 - 58ms/epoch - 6ms/step
Epoch 163/1000
10/10 - 0s - loss: 33.8438 - val_loss: 27.6048 - 45ms/epoch - 5ms/step
Epoch 164/1000
10/10 - 0s - loss: 33.1440 - val_loss: 26.6754 - 71ms/epoch - 7ms/step
Epoch 165/1000
10/10 - 0s - loss: 33.6024 - val_loss: 28.4600 - 45ms/epoch - 4ms/step
Epoch 166/1000
10/10 - 0s - loss: 33.0155 - val_loss: 26.3480 - 75ms/epoch - 8ms/step
Epoch 167/1000
10/10 - 0s - loss: 33.6239 - val_loss: 27.4919 - 43ms/epoch - 4ms/step
Epoch 168/1000
10/10 - 0s - loss: 33.7240 - val_loss: 28.3199 - 51ms/epoch - 5ms/step
Epoch 169/1000
10/10 - 0s - loss: 33.5670 - val_loss: 25.8991 - 62ms/epoch - 6ms/step
Epoch 170/1000
10/10 - 0s - loss: 31.7415 - val_loss: 26.0897 - 62ms/epoch - 6ms/step
Epoch 171/1000
10/10 - 0s - loss: 31.5303 - val_loss: 25.4196 - 49ms/epoch - 5ms/step
Epoch 172/1000
10/10 - 0s - loss: 31.8498 - val_loss: 26.9220 - 50ms/epoch - 5ms/step
Epoch 173/1000
10/10 - 0s - loss: 31.6775 - val_loss: 25.7945 - 62ms/epoch - 6ms/step
Epoch 174/1000
10/10 - 0s - loss: 31.3026 - val_loss: 25.3169 - 53ms/epoch - 5ms/step
Epoch 175/1000
10/10 - 0s - loss: 30.8672 - val_loss: 25.7407 - 69ms/epoch - 7ms/step
Epoch 176/1000
10/10 - 0s - loss: 31.5805 - val_loss: 24.5653 - 72ms/epoch - 7ms/step
Epoch 177/1000
10/10 - 0s - loss: 31.6575 - val_loss: 24.7597 - 53ms/epoch - 5ms/step
Epoch 178/1000
10/10 - 0s - loss: 31.0366 - val_loss: 26.0440 - 61ms/epoch - 6ms/step
Epoch 179/1000
10/10 - 0s - loss: 30.4223 - val_loss: 24.2567 - 65ms/epoch - 7ms/step
Epoch 180/1000
10/10 - 0s - loss: 29.6591 - val_loss: 24.0481 - 62ms/epoch - 6ms/step
Epoch 181/1000
10/10 - 0s - loss: 29.5555 - val_loss: 23.9970 - 65ms/epoch - 7ms/step
Epoch 182/1000
10/10 - 0s - loss: 29.4170 - val_loss: 23.5796 - 58ms/epoch - 6ms/step
Epoch 183/1000
10/10 - 0s - loss: 29.2324 - val_loss: 24.1796 - 52ms/epoch - 5ms/step
Epoch 184/1000
10/10 - 0s - loss: 28.8220 - val_loss: 23.2315 - 46ms/epoch - 5ms/step
Epoch 185/1000
10/10 - 0s - loss: 29.0582 - val_loss: 24.1037 - 61ms/epoch - 6ms/step
Epoch 186/1000
10/10 - 0s - loss: 28.7244 - val_loss: 22.9213 - 64ms/epoch - 6ms/step
Epoch 187/1000
10/10 - 0s - loss: 28.4226 - val_loss: 23.6453 - 52ms/epoch - 5ms/step
Epoch 188/1000
```

```
10/10 - 0s - loss: 30.4988 - val_loss: 22.3928 - 67ms/epoch - 7ms/step
Epoch 189/1000
10/10 - 0s - loss: 29.2073 - val_loss: 22.8565 - 46ms/epoch - 5ms/step
Epoch 190/1000
10/10 - 0s - loss: 27.8428 - val_loss: 24.4894 - 61ms/epoch - 6ms/step
Epoch 191/1000
10/10 - 0s - loss: 28.7934 - val_loss: 21.9677 - 46ms/epoch - 5ms/step
Epoch 192/1000
10/10 - 0s - loss: 29.0026 - val_loss: 23.3321 - 68ms/epoch - 7ms/step
Epoch 193/1000
10/10 - 0s - loss: 28.7747 - val_loss: 21.6816 - 60ms/epoch - 6ms/step
Epoch 194/1000
10/10 - 0s - loss: 27.4620 - val_loss: 21.3625 - 48ms/epoch - 5ms/step
Epoch 195/1000
10/10 - 0s - loss: 26.9269 - val_loss: 21.5949 - 46ms/epoch - 5ms/step
Epoch 196/1000
10/10 - 0s - loss: 27.0653 - val_loss: 21.2440 - 66ms/epoch - 7ms/step
Epoch 197/1000
10/10 - 0s - loss: 26.5345 - val_loss: 21.7376 - 60ms/epoch - 6ms/step
Epoch 198/1000
10/10 - 0s - loss: 26.6457 - val_loss: 20.7840 - 43ms/epoch - 4ms/step
Epoch 199/1000
10/10 - 0s - loss: 26.1900 - val_loss: 20.5610 - 63ms/epoch - 6ms/step
Epoch 200/1000
10/10 - 0s - loss: 26.4989 - val_loss: 21.2801 - 73ms/epoch - 7ms/step
Epoch 201/1000
10/10 - 0s - loss: 25.7045 - val_loss: 20.3552 - 51ms/epoch - 5ms/step
Epoch 202/1000
10/10 - 0s - loss: 26.2127 - val_loss: 20.3249 - 47ms/epoch - 5ms/step
Epoch 203/1000
10/10 - 0s - loss: 26.5411 - val_loss: 23.3969 - 43ms/epoch - 4ms/step
Epoch 204/1000
10/10 - 0s - loss: 26.0485 - val_loss: 19.6083 - 52ms/epoch - 5ms/step
Epoch 205/1000
10/10 - 0s - loss: 25.7512 - val_loss: 21.4974 - 61ms/epoch - 6ms/step
Epoch 206/1000
10/10 - 0s - loss: 27.9661 - val_loss: 24.6909 - 73ms/epoch - 7ms/step
Epoch 207/1000
10/10 - 0s - loss: 27.1431 - val_loss: 20.3076 - 41ms/epoch - 4ms/step
Epoch 208/1000
10/10 - 0s - loss: 25.5981 - val_loss: 20.7630 - 43ms/epoch - 4ms/step
Epoch 209/1000
10/10 - 0s - loss: 25.4804 - val_loss: 18.8038 - 65ms/epoch - 7ms/step
Epoch 210/1000
10/10 - 0s - loss: 26.6374 - val_loss: 26.9133 - 63ms/epoch - 6ms/step
Epoch 211/1000
10/10 - 0s - loss: 26.2150 - val_loss: 18.8805 - 48ms/epoch - 5ms/step
Epoch 212/1000
10/10 - 0s - loss: 25.7188 - val_loss: 20.6097 - 50ms/epoch - 5ms/step
Epoch 213/1000
10/10 - 0s - loss: 24.9249 - val_loss: 18.8219 - 50ms/epoch - 5ms/step
Epoch 214/1000
Restoring model weights from the end of the best epoch: 209.
10/10 - 0s - loss: 24.0144 - val_loss: 19.2638 - 50ms/epoch - 5ms/step
Epoch 214: early stopping
```

Out[3]: <keras.callbacks.History at 0x7f126e090b90>

We now evaluate the trained neural network to see the RMSE. We will use this trained neural network to compare the accuracy between the original data and the GAN-generated data. We will later see that you can use such comparisons for anomaly detection. We can use this technique can be used for security systems. If a

neural network trained on original data does not perform well on new data, then the new data may be suspect or fake.

In [4]:

```
pred = model.predict(x_test)
score = np.sqrt(metrics.mean_squared_error(pred,y_test))
print("Final score (RMSE): {}".format(score))
```

Final score (RMSE): 4.33633936452545

Training a GAN for Auto MPG

Next, we will train the GAN to generate fake data from the original MPG data. There are quite a few options that you can fine-tune for the GAN. The example presented here uses most of the default values. These are the usual hyperparameters that must be tuned for any model and require some experimentation for optimal results. To learn more about tabgan refer to its paper or this [Medium article](#), written by the creator of tabgan.

In [5]:

```
from tabgan.sampler import GANGenerator
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split

gen_x, gen_y = GANGenerator(gen_x_times=1.1, cat_cols=None,
                             bot_filter_quantile=0.001, top_filter_quantile=0.999, \
                             is_post_process=True,
                             adversarial_model_params={
                                 "metrics": "rmse", "max_depth": 2, "max_bin": 100,
                                 "learning_rate": 0.02, "random_state": \
                                 42, "n_estimators": 500,
                             }, pregeneration_frac=2, only_generated_data=False,\ 
                             gan_params = {"batch_size": 500, "patience": 25, \
                             "epochs" : 500,}).generate_data_pipe(df_x_train, df_y_train,\ 
                             df_x_test, deep_copy=True, only_adversarial=False, \
                             use_adversarial=True)
```

Fitting CTGAN transformers for each column: 0% | 0/8 [00:00<?, ?it/s]
Training CTGAN, epochs:: 0% | 0/500 [00:00<?, ?it/s]

Note: if you receive an error running the above code, you likely need to restart the runtime. You should have a "restart runtime" button in the output from the second cell. Once you restart the runtime, rerun all of the cells. This step is necessary as tabgan requires specific versions of some packages.

Evaluating the GAN Results

If we display the results, we can see that the GAN-generated data looks similar to the original. Some values, typically whole numbers in the original data, have fractional values in the synthetic data.

In [6]:

gen_x

Out[6]:

	cylinders	displacement	horsepower	weight	acceleration	year	origin
0	5	296.949632	106.872450	2133	18.323035	73	2
1	5	247.744505	97.532052	2233	19.490136	75	2
2	4	259.648421	108.111921	2424	19.898952	79	3
3	5	319.208637	93.764364	2054	19.420225	78	3
4	4	386.237667	129.837418	1951	20.989091	82	2
...
542	8	304.000000	150.000000	3672	11.500000	72	1
543	8	304.000000	150.000000	3433	12.000000	70	1
544	4	98.000000	80.000000	2164	15.000000	72	1
545	4	97.500000	80.000000	2126	17.000000	72	1
546	5	138.526374	68.958515	2497	13.495784	71	1

547 rows × 7 columns

Finally, we present the synthetic data to the previously trained neural network to see how accurately we can predict the synthetic targets. As we can see, you lose some RMSE accuracy by going to synthetic data.

In [7]:

```
# Predict
pred = model.predict(gen_x.values)
score = np.sqrt(metrics.mean_squared_error(pred, gen_y.values))
print("Final score (RMSE): {}".format(score))
```

Final score (RMSE): 9.083745225633098

T81-558: Applications of Deep Neural Networks

Module 8: Kaggle Data Sets

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

Module 8 Material

- **Part 8.1: Introduction to Kaggle** [\[Video\]](#) [\[Notebook\]](#)
- Part 8.2: Building Ensembles with Scikit-Learn and Keras [\[Video\]](#) [\[Notebook\]](#)
- Part 8.3: How Should you Architect Your Keras Neural Network: Hyperparameters [\[Video\]](#) [\[Notebook\]](#)
- Part 8.4: Bayesian Hyperparameter Optimization for Keras [\[Video\]](#) [\[Notebook\]](#)
- Part 8.5: Current Semester's Kaggle [\[Video\]](#) [\[Notebook\]](#)

Part 8.1: Introduction to Kaggle

[Kaggle](#) runs competitions where data scientists compete to provide the best model to fit the data. A simple project to get started with Kaggle is the [Titanic data set](#). Most Kaggle competitions end on a specific date. Website organizers have scheduled the Titanic competition to end on December 31, 20xx (with the year usually rolling forward). However, they have already extended the deadline several times, and an extension beyond 2014 is also possible. Second, the Titanic data set is considered a tutorial data set. There is no prize, and your score in the competition does not count towards becoming a Kaggle Master.

Kaggle Ranks

You achieve Kaggle ranks by earning gold, silver, and bronze medals.

- [Kaggle Top Users](#)
- [Current Top Kaggle User's Profile Page](#)
- [Jeff Heaton's \(your instructor\) Kaggle Profile](#)

- [Current Kaggle Ranking System](#)

Typical Kaggle Competition

A typical Kaggle competition will have several components. Consider the Titanic tutorial:

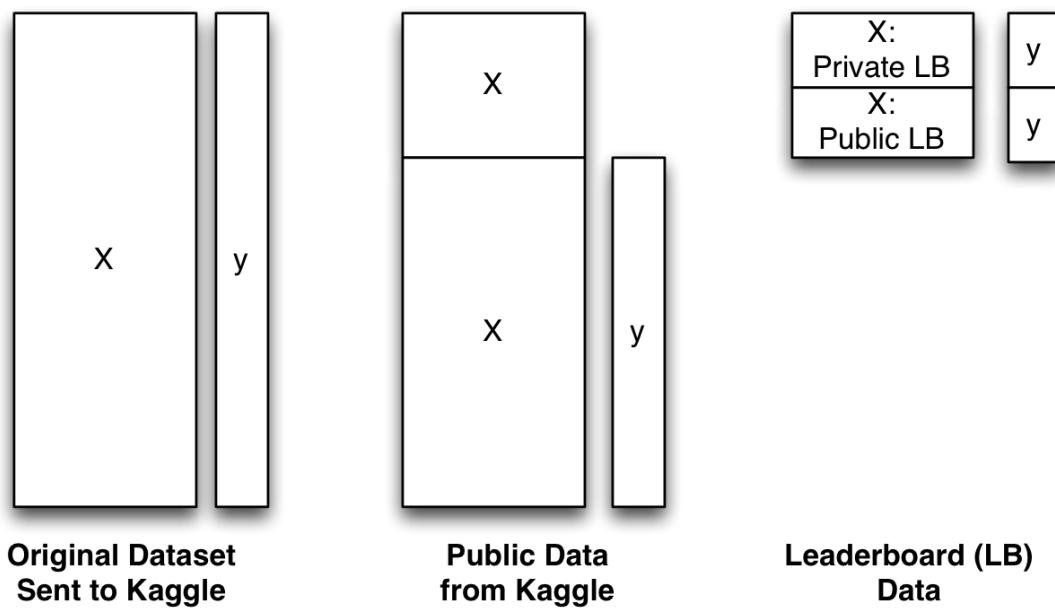
- [Competition Summary Page](#)
- [Data Page](#)
- [Evaluation Description Page](#)
- [Leaderboard](#)

How Kaggle Competition Scoring

Kaggle is provided with a data set by the competition sponsor, as seen in Figure 8.SCORE. Kaggle divides this data set as follows:

- **Complete Data Set** - This is the complete data set.
 - **Training Data Set** - This dataset provides both the inputs and the outcomes for the training portion of the data set.
 - **Test Data Set** - This dataset provides the complete test data; however, it does not give the outcomes. Your submission file should contain the predicted results for this data set.
 - **Public Leaderboard** - Kaggle does not tell you what part of the test data set contributes to the public leaderboard. Your public score is calculated based on this part of the data set.
 - **Private Leaderboard** - Likewise, Kaggle does not tell you what part of the test data set contributes to the public leaderboard. Your final score/rank is calculated based on this part. You do not see your private leaderboard score until the end.

Figure 8.SCORE: How Kaggle Competition Scoring



Preparing a Kaggle Submission

You do not submit the code to your solution to Kaggle. For competitions, you are scored entirely on the accuracy of your submission file. A Kaggle submission file is always a CSV file that contains the **Id** of the row you are predicting and the answer. For the titanic competition, a submission file looks something like this:

```
PassengerId,Survived  
892,0  
893,1  
894,1  
895,0  
896,0  
897,1  
...
```

The above file states the prediction for each of the various passengers. You should only predict on ID's that are in the test file. Likewise, you should render a prediction for every row in the test file. Some competitions will have different formats for their answers. For example, a multi-classification will usually have a column for each class and your predictions for each class.

Select Kaggle Competitions

There have been many exciting competitions on Kaggle; these are some of my favorites. Some select predictive modeling competitions which use tabular data include:

- Otto Group Product Classification Challenge
- Galaxy Zoo - The Galaxy Challenge
- Practice Fusion Diabetes Classification
- Predicting a Biological Response

Many Kaggle competitions include computer vision datasets, such as:

- Diabetic Retinopathy Detection
- Cats vs Dogs
- State Farm Distracted Driver Detection

Module 8 Assignment

You can find the first assignment here: [assignment 8](#)

In []:

T81-558: Applications of Deep Neural Networks

Module 8: Kaggle Data Sets

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

Module 8 Material

- Part 8.1: Introduction to Kaggle [\[Video\]](#) [\[Notebook\]](#)
- **Part 8.2: Building Ensembles with Scikit-Learn and Keras** [\[Video\]](#) [\[Notebook\]](#)
- Part 8.3: How Should you Architect Your Keras Neural Network: Hyperparameters [\[Video\]](#) [\[Notebook\]](#)
- Part 8.4: Bayesian Hyperparameter Optimization for Keras [\[Video\]](#) [\[Notebook\]](#)
- Part 8.5: Current Semester's Kaggle [\[Video\]](#) [\[Notebook\]](#)

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  
  
# Nicely formatted time string  
def hms_string(sec_elapsed):  
    h = int(sec_elapsed / (60 * 60))  
    m = int((sec_elapsed % (60 * 60)) / 60)
```

```
s = sec_elapsed % 60
return "{}:{}{:>02} {:>05.2f} ".format(h, m, s)
```

Mounted at /content/drive

Note: using Google CoLab

Part 8.2: Building Ensembles with Scikit-Learn and Keras

Evaluating Feature Importance

Feature importance tells us how important each feature (from the feature/import vector) is to predicting a neural network or another model. There are many different ways to evaluate the feature importance of neural networks. The following paper presents an excellent (and readable) overview of the various means of assessing the significance of neural network inputs/features.

- An accurate comparison of methods for quantifying variable importance in artificial neural networks using simulated data [Cite:olden2004accurate].
Ecological Modelling, 178(3), 389-397.

In summary, the following methods are available to neural networks:

- Connection Weights Algorithm
- Partial Derivatives
- Input Perturbation
- Sensitivity Analysis
- Forward Stepwise Addition
- Improved Stepwise Selection 1
- Backward Stepwise Elimination
- Improved Stepwise Selection

For this chapter, we will use the input Perturbation feature ranking algorithm. This algorithm will work with any regression or classification network. In the next section, I provide an implementation of the input perturbation algorithm for scikit-learn. This code implements a function below that will work with any scikit-learn model.

[Leo Breiman](#) provided this algorithm in his seminal paper on random forests. [Citebreiman2001random:] Although he presented this algorithm in conjunction with random forests, it is model-independent and appropriate for any supervised learning model. This algorithm, known as the input perturbation algorithm, works by evaluating a trained model's accuracy with each input individually shuffled from a data set. Shuffling an input causes it to become useless—effectively removing it from the model. More important inputs will produce a less accurate

score when they are removed by shuffling them. This process makes sense because important features will contribute to the model's accuracy. I first presented the TensorFlow implementation of this algorithm in the following paper.

- Early stabilizing feature importance for TensorFlow deep neural networks[Cite:heaton2017early]

This algorithm will use log loss to evaluate a classification problem and RMSE for regression.

```
In [2]: from sklearn import metrics
import scipy as sp
import numpy as np
import math
from sklearn import metrics

def perturbation_rank(model, x, y, names, regression):
    errors = []

    for i in range(x.shape[1]):
        hold = np.array(x[:, i])
        np.random.shuffle(x[:, i])

        if regression:
            pred = model.predict(x)
            error = metrics.mean_squared_error(y, pred)
        else:
            pred = model.predict(x)
            error = metrics.log_loss(y, pred)

        errors.append(error)
        x[:, i] = hold

    max_error = np.max(errors)
    importance = [e/max_error for e in errors]

    data = {'name':names,'error':errors,'importance':importance}
    result = pd.DataFrame(data, columns = ['name','error','importance'])
    result.sort_values(by=['importance'], ascending=[0], inplace=True)
    result.reset_index(inplace=True, drop=True)
    return result
```

Classification and Input Perturbation Ranking

We now look at the code to perform perturbation ranking for a classification neural network. The implementation technique is slightly different for classification vs. regression, so I must provide two different implementations. The primary difference between classification and regression is how we evaluate the accuracy of the neural network in each of these two network types. We will

use the Root Mean Square (RMSE) error calculation, whereas we will use log loss for classification.

The code presented below creates a classification neural network that will predict the classic iris dataset.

```
In [3]: # HIDE OUTPUT
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
from sklearn.model_selection import train_test_split

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 train/test
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')
model.fit(x_train,y_train,verbose=2,epochs=100)
```

Epoch 1/100
4/4 - 1s - loss: 2.0814 - 1s/epoch - 292ms/step
Epoch 2/100
4/4 - 0s - loss: 1.6125 - 14ms/epoch - 4ms/step
Epoch 3/100
4/4 - 0s - loss: 1.3316 - 26ms/epoch - 7ms/step
Epoch 4/100
4/4 - 0s - loss: 1.2246 - 13ms/epoch - 3ms/step
Epoch 5/100
4/4 - 0s - loss: 1.1989 - 13ms/epoch - 3ms/step
Epoch 6/100
4/4 - 0s - loss: 1.1349 - 14ms/epoch - 4ms/step
Epoch 7/100
4/4 - 0s - loss: 1.0543 - 21ms/epoch - 5ms/step
Epoch 8/100
4/4 - 0s - loss: 0.9987 - 25ms/epoch - 6ms/step
Epoch 9/100
4/4 - 0s - loss: 0.9449 - 20ms/epoch - 5ms/step
Epoch 10/100
4/4 - 0s - loss: 0.9032 - 16ms/epoch - 4ms/step
Epoch 11/100
4/4 - 0s - loss: 0.8623 - 20ms/epoch - 5ms/step
Epoch 12/100
4/4 - 0s - loss: 0.8274 - 12ms/epoch - 3ms/step
Epoch 13/100
4/4 - 0s - loss: 0.8013 - 18ms/epoch - 4ms/step
Epoch 14/100
4/4 - 0s - loss: 0.7718 - 18ms/epoch - 5ms/step
Epoch 15/100
4/4 - 0s - loss: 0.7426 - 19ms/epoch - 5ms/step
Epoch 16/100
4/4 - 0s - loss: 0.7163 - 13ms/epoch - 3ms/step
Epoch 17/100
4/4 - 0s - loss: 0.6933 - 13ms/epoch - 3ms/step
Epoch 18/100
4/4 - 0s - loss: 0.6689 - 14ms/epoch - 3ms/step
Epoch 19/100
4/4 - 0s - loss: 0.6488 - 11ms/epoch - 3ms/step
Epoch 20/100
4/4 - 0s - loss: 0.6294 - 11ms/epoch - 3ms/step
Epoch 21/100
4/4 - 0s - loss: 0.6094 - 20ms/epoch - 5ms/step
Epoch 22/100
4/4 - 0s - loss: 0.5911 - 18ms/epoch - 4ms/step
Epoch 23/100
4/4 - 0s - loss: 0.5725 - 16ms/epoch - 4ms/step
Epoch 24/100
4/4 - 0s - loss: 0.5550 - 13ms/epoch - 3ms/step
Epoch 25/100
4/4 - 0s - loss: 0.5389 - 14ms/epoch - 3ms/step
Epoch 26/100
4/4 - 0s - loss: 0.5207 - 15ms/epoch - 4ms/step
Epoch 27/100
4/4 - 0s - loss: 0.5041 - 14ms/epoch - 4ms/step
Epoch 28/100
4/4 - 0s - loss: 0.4901 - 14ms/epoch - 3ms/step

Epoch 29/100
4/4 - 0s - loss: 0.4765 - 14ms/epoch - 4ms/step
Epoch 30/100
4/4 - 0s - loss: 0.4619 - 16ms/epoch - 4ms/step
Epoch 31/100
4/4 - 0s - loss: 0.4489 - 16ms/epoch - 4ms/step
Epoch 32/100
4/4 - 0s - loss: 0.4366 - 13ms/epoch - 3ms/step
Epoch 33/100
4/4 - 0s - loss: 0.4243 - 13ms/epoch - 3ms/step
Epoch 34/100
4/4 - 0s - loss: 0.4124 - 14ms/epoch - 3ms/step
Epoch 35/100
4/4 - 0s - loss: 0.4015 - 14ms/epoch - 3ms/step
Epoch 36/100
4/4 - 0s - loss: 0.3917 - 21ms/epoch - 5ms/step
Epoch 37/100
4/4 - 0s - loss: 0.3826 - 30ms/epoch - 7ms/step
Epoch 38/100
4/4 - 0s - loss: 0.3713 - 18ms/epoch - 4ms/step
Epoch 39/100
4/4 - 0s - loss: 0.3621 - 16ms/epoch - 4ms/step
Epoch 40/100
4/4 - 0s - loss: 0.3543 - 14ms/epoch - 4ms/step
Epoch 41/100
4/4 - 0s - loss: 0.3460 - 14ms/epoch - 4ms/step
Epoch 42/100
4/4 - 0s - loss: 0.3385 - 28ms/epoch - 7ms/step
Epoch 43/100
4/4 - 0s - loss: 0.3280 - 31ms/epoch - 8ms/step
Epoch 44/100
4/4 - 0s - loss: 0.3211 - 15ms/epoch - 4ms/step
Epoch 45/100
4/4 - 0s - loss: 0.3144 - 14ms/epoch - 3ms/step
Epoch 46/100
4/4 - 0s - loss: 0.3068 - 15ms/epoch - 4ms/step
Epoch 47/100
4/4 - 0s - loss: 0.2992 - 19ms/epoch - 5ms/step
Epoch 48/100
4/4 - 0s - loss: 0.2922 - 18ms/epoch - 4ms/step
Epoch 49/100
4/4 - 0s - loss: 0.2847 - 37ms/epoch - 9ms/step
Epoch 50/100
4/4 - 0s - loss: 0.2803 - 13ms/epoch - 3ms/step
Epoch 51/100
4/4 - 0s - loss: 0.2756 - 13ms/epoch - 3ms/step
Epoch 52/100
4/4 - 0s - loss: 0.2665 - 18ms/epoch - 4ms/step
Epoch 53/100
4/4 - 0s - loss: 0.2632 - 16ms/epoch - 4ms/step
Epoch 54/100
4/4 - 0s - loss: 0.2571 - 20ms/epoch - 5ms/step
Epoch 55/100
4/4 - 0s - loss: 0.2499 - 17ms/epoch - 4ms/step
Epoch 56/100
4/4 - 0s - loss: 0.2458 - 17ms/epoch - 4ms/step

Epoch 57/100
4/4 - 0s - loss: 0.2399 - 23ms/epoch - 6ms/step
Epoch 58/100
4/4 - 0s - loss: 0.2340 - 16ms/epoch - 4ms/step
Epoch 59/100
4/4 - 0s - loss: 0.2318 - 16ms/epoch - 4ms/step
Epoch 60/100
4/4 - 0s - loss: 0.2225 - 12ms/epoch - 3ms/step
Epoch 61/100
4/4 - 0s - loss: 0.2266 - 15ms/epoch - 4ms/step
Epoch 62/100
4/4 - 0s - loss: 0.2178 - 12ms/epoch - 3ms/step
Epoch 63/100
4/4 - 0s - loss: 0.2116 - 15ms/epoch - 4ms/step
Epoch 64/100
4/4 - 0s - loss: 0.2137 - 21ms/epoch - 5ms/step
Epoch 65/100
4/4 - 0s - loss: 0.2030 - 17ms/epoch - 4ms/step
Epoch 66/100
4/4 - 0s - loss: 0.2041 - 14ms/epoch - 3ms/step
Epoch 67/100
4/4 - 0s - loss: 0.2001 - 15ms/epoch - 4ms/step
Epoch 68/100
4/4 - 0s - loss: 0.1919 - 25ms/epoch - 6ms/step
Epoch 69/100
4/4 - 0s - loss: 0.1894 - 23ms/epoch - 6ms/step
Epoch 70/100
4/4 - 0s - loss: 0.1863 - 17ms/epoch - 4ms/step
Epoch 71/100
4/4 - 0s - loss: 0.1823 - 16ms/epoch - 4ms/step
Epoch 72/100
4/4 - 0s - loss: 0.1790 - 24ms/epoch - 6ms/step
Epoch 73/100
4/4 - 0s - loss: 0.1780 - 16ms/epoch - 4ms/step
Epoch 74/100
4/4 - 0s - loss: 0.1755 - 15ms/epoch - 4ms/step
Epoch 75/100
4/4 - 0s - loss: 0.1719 - 31ms/epoch - 8ms/step
Epoch 76/100
4/4 - 0s - loss: 0.1767 - 21ms/epoch - 5ms/step
Epoch 77/100
4/4 - 0s - loss: 0.1694 - 17ms/epoch - 4ms/step
Epoch 78/100
4/4 - 0s - loss: 0.1655 - 27ms/epoch - 7ms/step
Epoch 79/100
4/4 - 0s - loss: 0.1634 - 23ms/epoch - 6ms/step
Epoch 80/100
4/4 - 0s - loss: 0.1566 - 17ms/epoch - 4ms/step
Epoch 81/100
4/4 - 0s - loss: 0.1563 - 17ms/epoch - 4ms/step
Epoch 82/100
4/4 - 0s - loss: 0.1536 - 15ms/epoch - 4ms/step
Epoch 83/100
4/4 - 0s - loss: 0.1504 - 18ms/epoch - 4ms/step
Epoch 84/100
4/4 - 0s - loss: 0.1502 - 16ms/epoch - 4ms/step

```
Epoch 85/100
4/4 - 0s - loss: 0.1469 - 17ms/epoch - 4ms/step
Epoch 86/100
4/4 - 0s - loss: 0.1448 - 28ms/epoch - 7ms/step
Epoch 87/100
4/4 - 0s - loss: 0.1424 - 23ms/epoch - 6ms/step
Epoch 88/100
4/4 - 0s - loss: 0.1401 - 25ms/epoch - 6ms/step
Epoch 89/100
4/4 - 0s - loss: 0.1386 - 47ms/epoch - 12ms/step
Epoch 90/100
4/4 - 0s - loss: 0.1365 - 30ms/epoch - 7ms/step
Epoch 91/100
4/4 - 0s - loss: 0.1383 - 41ms/epoch - 10ms/step
Epoch 92/100
4/4 - 0s - loss: 0.1332 - 12ms/epoch - 3ms/step
Epoch 93/100
4/4 - 0s - loss: 0.1311 - 20ms/epoch - 5ms/step
Epoch 94/100
4/4 - 0s - loss: 0.1320 - 20ms/epoch - 5ms/step
Epoch 95/100
4/4 - 0s - loss: 0.1302 - 17ms/epoch - 4ms/step
Epoch 96/100
4/4 - 0s - loss: 0.1311 - 19ms/epoch - 5ms/step
Epoch 97/100
4/4 - 0s - loss: 0.1248 - 14ms/epoch - 3ms/step
Epoch 98/100
4/4 - 0s - loss: 0.1254 - 12ms/epoch - 3ms/step
Epoch 99/100
4/4 - 0s - loss: 0.1275 - 13ms/epoch - 3ms/step
Epoch 100/100
4/4 - 0s - loss: 0.1225 - 41ms/epoch - 10ms/step
```

Out[3]: <keras.callbacks.History at 0x7fc869fd2950>

Next, we evaluate the accuracy of the trained model. Here we see that the neural network performs great, with an accuracy of 1.0. We might fear overfitting with such high accuracy for a more complex dataset. However, for this example, we are more interested in determining the importance of each column.

```
In [4]: 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

We are now ready to call the input perturbation algorithm. First, we extract the column names and remove the target column. The target column is not important, as it is the objective, not one of the inputs. In supervised learning, the target is of the utmost importance.

We can see the importance displayed in the following table. The most important column is always 1.0, and lesser columns will continue in a downward trend. The least important column will have the lowest rank.

```
In [5]: # Rank the features
from IPython.display import display, HTML

names = list(df.columns) # x+y column names
names.remove("species") # remove the target(y)
rank = perturbation_rank(model, x_test, y_test, names, False)
display(rank)
```

	name	error	importance
0	petal_l	2.609378	1.000000
1	petal_w	0.480387	0.184100
2	sepal_l	0.223239	0.085553
3	sepal_w	0.128518	0.049252

Regression and Input Perturbation Ranking

We now see how to use input perturbation ranking for a regression neural network. We will use the MPG dataset as a demonstration. The code below loads the MPG dataset and creates a regression neural network for this dataset. The code trains the neural network and calculates an RMSE evaluation.

```
In [6]: # HIDE OUTPUT
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from sklearn.model_selection import train_test_split
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

# Split into train/test
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')
model.fit(x_train,y_train,verbose=2,epochs=100)

# Predict
pred = model.predict(x)
```

Epoch 1/100
10/10 - 1s - loss: 328433.8125 - 898ms/epoch - 90ms/step
Epoch 2/100
10/10 - 0s - loss: 78914.6406 - 26ms/epoch - 3ms/step
Epoch 3/100
10/10 - 0s - loss: 5371.1025 - 50ms/epoch - 5ms/step
Epoch 4/100
10/10 - 0s - loss: 4021.9753 - 34ms/epoch - 3ms/step
Epoch 5/100
10/10 - 0s - loss: 4438.5728 - 33ms/epoch - 3ms/step
Epoch 6/100
10/10 - 0s - loss: 1030.3115 - 34ms/epoch - 3ms/step
Epoch 7/100
10/10 - 0s - loss: 594.9177 - 31ms/epoch - 3ms/step
Epoch 8/100
10/10 - 0s - loss: 655.3908 - 31ms/epoch - 3ms/step
Epoch 9/100
10/10 - 0s - loss: 465.0457 - 25ms/epoch - 2ms/step
Epoch 10/100
10/10 - 0s - loss: 458.7520 - 30ms/epoch - 3ms/step
Epoch 11/100
10/10 - 0s - loss: 452.4102 - 22ms/epoch - 2ms/step
Epoch 12/100
10/10 - 0s - loss: 439.8730 - 25ms/epoch - 3ms/step
Epoch 13/100
10/10 - 0s - loss: 434.8245 - 27ms/epoch - 3ms/step
Epoch 14/100
10/10 - 0s - loss: 433.7303 - 25ms/epoch - 3ms/step
Epoch 15/100
10/10 - 0s - loss: 427.2859 - 46ms/epoch - 5ms/step
Epoch 16/100
10/10 - 0s - loss: 424.1164 - 50ms/epoch - 5ms/step
Epoch 17/100
10/10 - 0s - loss: 422.3007 - 42ms/epoch - 4ms/step
Epoch 18/100
10/10 - 0s - loss: 418.4877 - 31ms/epoch - 3ms/step
Epoch 19/100
10/10 - 0s - loss: 414.2283 - 23ms/epoch - 2ms/step
Epoch 20/100
10/10 - 0s - loss: 410.2691 - 34ms/epoch - 3ms/step
Epoch 21/100
10/10 - 0s - loss: 407.0490 - 29ms/epoch - 3ms/step
Epoch 22/100
10/10 - 0s - loss: 406.2433 - 46ms/epoch - 5ms/step
Epoch 23/100
10/10 - 0s - loss: 399.7404 - 37ms/epoch - 4ms/step
Epoch 24/100
10/10 - 0s - loss: 396.3280 - 66ms/epoch - 7ms/step
Epoch 25/100
10/10 - 0s - loss: 391.0629 - 28ms/epoch - 3ms/step
Epoch 26/100
10/10 - 0s - loss: 387.3203 - 26ms/epoch - 3ms/step
Epoch 27/100
10/10 - 0s - loss: 382.7670 - 54ms/epoch - 5ms/step
Epoch 28/100
10/10 - 0s - loss: 380.6316 - 21ms/epoch - 2ms/step

Epoch 29/100
10/10 - 0s - loss: 375.9518 - 30ms/epoch - 3ms/step
Epoch 30/100
10/10 - 0s - loss: 372.7001 - 24ms/epoch - 2ms/step
Epoch 31/100
10/10 - 0s - loss: 366.7871 - 24ms/epoch - 2ms/step
Epoch 32/100
10/10 - 0s - loss: 363.4180 - 42ms/epoch - 4ms/step
Epoch 33/100
10/10 - 0s - loss: 359.6006 - 47ms/epoch - 5ms/step
Epoch 34/100
10/10 - 0s - loss: 359.4055 - 46ms/epoch - 5ms/step
Epoch 35/100
10/10 - 0s - loss: 350.7181 - 29ms/epoch - 3ms/step
Epoch 36/100
10/10 - 0s - loss: 348.6260 - 42ms/epoch - 4ms/step
Epoch 37/100
10/10 - 0s - loss: 343.6122 - 28ms/epoch - 3ms/step
Epoch 38/100
10/10 - 0s - loss: 339.6165 - 32ms/epoch - 3ms/step
Epoch 39/100
10/10 - 0s - loss: 334.5634 - 32ms/epoch - 3ms/step
Epoch 40/100
10/10 - 0s - loss: 332.6061 - 34ms/epoch - 3ms/step
Epoch 41/100
10/10 - 0s - loss: 326.7434 - 22ms/epoch - 2ms/step
Epoch 42/100
10/10 - 0s - loss: 323.8063 - 40ms/epoch - 4ms/step
Epoch 43/100
10/10 - 0s - loss: 320.2585 - 29ms/epoch - 3ms/step
Epoch 44/100
10/10 - 0s - loss: 315.3609 - 23ms/epoch - 2ms/step
Epoch 45/100
10/10 - 0s - loss: 311.4920 - 23ms/epoch - 2ms/step
Epoch 46/100
10/10 - 0s - loss: 308.9212 - 29ms/epoch - 3ms/step
Epoch 47/100
10/10 - 0s - loss: 303.1410 - 24ms/epoch - 2ms/step
Epoch 48/100
10/10 - 0s - loss: 299.9317 - 24ms/epoch - 2ms/step
Epoch 49/100
10/10 - 0s - loss: 294.4305 - 23ms/epoch - 2ms/step
Epoch 50/100
10/10 - 0s - loss: 291.4469 - 24ms/epoch - 2ms/step
Epoch 51/100
10/10 - 0s - loss: 287.3263 - 41ms/epoch - 4ms/step
Epoch 52/100
10/10 - 0s - loss: 284.3096 - 49ms/epoch - 5ms/step
Epoch 53/100
10/10 - 0s - loss: 280.5522 - 30ms/epoch - 3ms/step
Epoch 54/100
10/10 - 0s - loss: 276.1487 - 26ms/epoch - 3ms/step
Epoch 55/100
10/10 - 0s - loss: 271.3444 - 42ms/epoch - 4ms/step
Epoch 56/100
10/10 - 0s - loss: 280.0936 - 33ms/epoch - 3ms/step

Epoch 57/100
10/10 - 0s - loss: 263.7166 - 40ms/epoch - 4ms/step
Epoch 58/100
10/10 - 0s - loss: 261.6750 - 56ms/epoch - 6ms/step
Epoch 59/100
10/10 - 0s - loss: 258.5714 - 45ms/epoch - 4ms/step
Epoch 60/100
10/10 - 0s - loss: 252.6791 - 31ms/epoch - 3ms/step
Epoch 61/100
10/10 - 0s - loss: 250.1348 - 53ms/epoch - 5ms/step
Epoch 62/100
10/10 - 0s - loss: 246.4157 - 72ms/epoch - 7ms/step
Epoch 63/100
10/10 - 0s - loss: 242.3768 - 46ms/epoch - 5ms/step
Epoch 64/100
10/10 - 0s - loss: 238.7874 - 28ms/epoch - 3ms/step
Epoch 65/100
10/10 - 0s - loss: 235.8578 - 42ms/epoch - 4ms/step
Epoch 66/100
10/10 - 0s - loss: 233.7492 - 24ms/epoch - 2ms/step
Epoch 67/100
10/10 - 0s - loss: 229.0066 - 26ms/epoch - 3ms/step
Epoch 68/100
10/10 - 0s - loss: 225.7449 - 25ms/epoch - 3ms/step
Epoch 69/100
10/10 - 0s - loss: 223.5038 - 25ms/epoch - 2ms/step
Epoch 70/100
10/10 - 0s - loss: 219.9561 - 39ms/epoch - 4ms/step
Epoch 71/100
10/10 - 0s - loss: 215.1055 - 58ms/epoch - 6ms/step
Epoch 72/100
10/10 - 0s - loss: 211.9364 - 39ms/epoch - 4ms/step
Epoch 73/100
10/10 - 0s - loss: 208.1019 - 55ms/epoch - 5ms/step
Epoch 74/100
10/10 - 0s - loss: 207.4119 - 34ms/epoch - 3ms/step
Epoch 75/100
10/10 - 0s - loss: 206.8693 - 40ms/epoch - 4ms/step
Epoch 76/100
10/10 - 0s - loss: 197.9749 - 49ms/epoch - 5ms/step
Epoch 77/100
10/10 - 0s - loss: 196.9090 - 34ms/epoch - 3ms/step
Epoch 78/100
10/10 - 0s - loss: 192.6349 - 45ms/epoch - 4ms/step
Epoch 79/100
10/10 - 0s - loss: 189.6783 - 31ms/epoch - 3ms/step
Epoch 80/100
10/10 - 0s - loss: 186.6584 - 25ms/epoch - 2ms/step
Epoch 81/100
10/10 - 0s - loss: 186.1920 - 29ms/epoch - 3ms/step
Epoch 82/100
10/10 - 0s - loss: 181.1735 - 31ms/epoch - 3ms/step
Epoch 83/100
10/10 - 0s - loss: 177.9338 - 51ms/epoch - 5ms/step
Epoch 84/100
10/10 - 0s - loss: 174.6662 - 87ms/epoch - 9ms/step

```
Epoch 85/100
10/10 - 0s - loss: 172.9421 - 90ms/epoch - 9ms/step
Epoch 86/100
10/10 - 0s - loss: 169.1906 - 58ms/epoch - 6ms/step
Epoch 87/100
10/10 - 0s - loss: 166.4181 - 57ms/epoch - 6ms/step
Epoch 88/100
10/10 - 0s - loss: 163.7466 - 36ms/epoch - 4ms/step
Epoch 89/100
10/10 - 0s - loss: 161.4653 - 29ms/epoch - 3ms/step
Epoch 90/100
10/10 - 0s - loss: 158.6274 - 30ms/epoch - 3ms/step
Epoch 91/100
10/10 - 0s - loss: 159.4237 - 32ms/epoch - 3ms/step
Epoch 92/100
10/10 - 0s - loss: 159.2035 - 31ms/epoch - 3ms/step
Epoch 93/100
10/10 - 0s - loss: 150.2793 - 38ms/epoch - 4ms/step
Epoch 94/100
10/10 - 0s - loss: 148.9276 - 36ms/epoch - 4ms/step
Epoch 95/100
10/10 - 0s - loss: 146.7706 - 34ms/epoch - 3ms/step
Epoch 96/100
10/10 - 0s - loss: 144.4946 - 29ms/epoch - 3ms/step
Epoch 97/100
10/10 - 0s - loss: 141.5782 - 28ms/epoch - 3ms/step
Epoch 98/100
10/10 - 0s - loss: 139.3355 - 27ms/epoch - 3ms/step
Epoch 99/100
10/10 - 0s - loss: 136.9762 - 56ms/epoch - 6ms/step
Epoch 100/100
10/10 - 0s - loss: 135.6660 - 23ms/epoch - 2ms/step
```

Just as before, we extract the column names and discard the target. We can now create a ranking of the importance of each of the input features. The feature with a ranking of 1.0 is the most important.

```
In [7]: # Rank the features
from IPython.display import display, HTML

names = list(df.columns) # x+y column names
names.remove("name")
names.remove("mpg") # remove the target(y)
rank = perturbation_rank(model, x_test, y_test, names, True)
display(rank)
```

	name	error	importance
0	displacement	139.657598	1.000000
1	acceleration	139.261508	0.997164
2	origin	134.637690	0.964056
3	year	134.177126	0.960758
4	cylinders	132.747246	0.950519
5	horsepower	121.501102	0.869993
6	weight	75.244610	0.538779

Biological Response with Neural Network

The following sections will demonstrate how to use feature importance ranking and ensembling with a more complex dataset. Ensembling is the process where you combine multiple models for greater accuracy. Kaggle competition winners frequently make use of ensembling for high-ranking solutions.

We will use the biological response dataset, a Kaggle dataset, where there is an unusually high number of columns. Because of the large number of columns, it is essential to use feature ranking to determine the importance of these columns. We begin by loading the dataset and preprocessing. This Kaggle dataset is a binary classification problem. You must predict if certain conditions will cause a biological response.

- Predicting a Biological Response

```
In [8]: import pandas as pd
import os
import numpy as np
from sklearn import metrics
from scipy.stats import zscore
from sklearn.model_selection import KFold
from IPython.display import HTML, display

URL = "https://data.heatonresearch.com/data/t81-558/kaggle/"

df_train = pd.read_csv(
    URL+"bio_train.csv",
    na_values=['NA', '?'])

df_test = pd.read_csv(
    URL+"bio_test.csv",
    na_values=['NA', '?'])

activity_classes = df_train['Activity']
```

A large number of columns is evident when we display the shape of the dataset.

```
In [9]: print(df_train.shape)  
(3751, 1777)
```

The following code constructs a classification neural network and trains it for the biological response dataset. Once trained, the accuracy is measured.

```
In [10]: import os  
import pandas as pd  
import tensorflow as tf  
from tensorflow.keras.models import Sequential  
from tensorflow.keras.layers import Dense, Activation  
from sklearn.model_selection import train_test_split  
from tensorflow.keras.callbacks import EarlyStopping  
import numpy as np  
import sklearn  
  
# Encode feature vector  
# Convert to numpy - Classification  
x_columns = df_train.columns.drop('Activity')  
x = df_train[x_columns].values  
y = df_train['Activity'].values # Classification  
x_submit = df_test[x_columns].values.astype(np.float32)  
  
# Split into train/test  
x_train, x_test, y_train, y_test = train_test_split(  
    x, y, test_size=0.25, random_state=42)  
  
print("Fitting/Training...")  
model = Sequential()  
model.add(Dense(25, input_dim=x.shape[1], activation='relu'))  
model.add(Dense(10))  
model.add(Dense(1,activation='sigmoid'))  
model.compile(loss='binary_crossentropy', optimizer='adam')  
monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3,  
                       patience=5, verbose=1, mode='auto')  
model.fit(x_train,y_train,validation_data=(x_test,y_test),  
          callbacks=[monitor],verbose=0,epochs=1000)  
print("Fitting done...")  
  
# Predict  
pred = model.predict(x_test).flatten()  
  
# Clip so that min is never exactly 0, max never 1  
pred = np.clip(pred,a_min=1e-6,a_max=(1-1e-6))  
print("Validation logloss: {}".format(  
    sklearn.metrics.log_loss(y_test,pred)))  
  
# Evaluate success using accuracy  
pred = pred>0.5 # If greater than 0.5 probability, then true  
score = metrics.accuracy_score(y_test, pred)
```

```

print("Validation accuracy score: {}".format(score))

# Build real submit file
pred_submit = model.predict(x_submit)

# Clip so that min is never exactly 0, max never 1 (would be a NaN score)
pred = np.clip(pred,a_min=1e-6,a_max=(1-1e-6))
submit_df = pd.DataFrame({'MoleculeId':[x+1 for x \
    in range(len(pred_submit))], 'PredictedProbability':\n        pred_submit.flatten()})
submit_df.to_csv("submit.csv", index=False)

```

Fitting/Training...
Epoch 7: early stopping
Fitting done...
Validation logloss: 0.5564708781752792
Validation accuracy score: 0.7515991471215352

What Features/Columns are Important

The following uses perturbation ranking to evaluate the neural network.

```
In [11]: # Rank the features
from IPython.display import display, HTML

names = list(df_train.columns) # x+y column names
names.remove("Activity") # remove the target(y)
rank = perturbation_rank(model, x_test, y_test, names, False)
display(rank[0:10])
```

	name	error	importance
0	D27	0.603974	1.000000
1	D1049	0.565997	0.937122
2	D51	0.565883	0.936934
3	D998	0.563872	0.933604
4	D1059	0.563745	0.933394
5	D961	0.563723	0.933357
6	D1407	0.563532	0.933041
7	D1309	0.562244	0.930908
8	D1100	0.561902	0.930341
9	D1275	0.561659	0.929940

Neural Network Ensemble

A neural network ensemble combines neural network predictions with other models. The program determines the exact blend of these models by logistic regression. The following code performs this blend for a classification. If you present the final predictions from the ensemble to Kaggle, you will see that the result is very accurate.

```
In [12]: # HIDE OUTPUT
import numpy as np
import os
import pandas as pd
import math
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import StratifiedKFold
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression

SHUFFLE = False
FOLDS = 10

def build_ann(input_size, classes, neurons):
    model = Sequential()
    model.add(Dense(neurons, input_dim=input_size, activation='relu'))
    model.add(Dense(1))
    model.add(Dense(classes, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam')
    return model

def mlogloss(y_test, preds):
    epsilon = 1e-15
    sum = 0
    for row in zip(preds, y_test):
        x = row[0][row[1]]
        x = max(epsilon, x)
        x = min(1-epsilon, x)
        sum+=math.log(x)
    return( (-1/len(preds))*sum)

def stretch(y):
    return (y - y.min()) / (y.max() - y.min())

def blend_ensemble(x, y, x_submit):
    kf = StratifiedKFold(FOLDS)
    folds = list(kf.split(x,y))

    models = [
        KerasClassifier(build_fn=build_ann, neurons=20,
                       input_size=x.shape[1], classes=2),
        KNeighborsClassifier(n_neighbors=3),
        RandomForestClassifier(n_estimators=100, n_jobs=-1,
                              criterion='gini'),
```

```

        RandomForestClassifier(n_estimators=100, n_jobs=-1,
                               criterion='entropy'),
        ExtraTreesClassifier(n_estimators=100, n_jobs=-1,
                               criterion='gini'),
        ExtraTreesClassifier(n_estimators=100, n_jobs=-1,
                               criterion='entropy'),
        GradientBoostingClassifier(learning_rate=0.05,
                                   subsample=0.5, max_depth=6, n_estimators=50)]


dataset_blend_train = np.zeros((x.shape[0], len(models)))
dataset_blend_test = np.zeros((x_submit.shape[0], len(models)))

for j, model in enumerate(models):
    print("Model: {} : {}".format(j, model) )
    fold_sums = np.zeros((x_submit.shape[0], len(folds)))
    total_loss = 0
    for i, (train, test) in enumerate(folds):
        x_train = x[train]
        y_train = y[train]
        x_test = x[test]
        y_test = y[test]
        model.fit(x_train, y_train)
        pred = np.array(model.predict_proba(x_test))
        dataset_blend_train[test, j] = pred[:, 1]
        pred2 = np.array(model.predict_proba(x_submit))
        fold_sums[:, i] = pred2[:, 1]
        loss = mlogloss(y_test, pred)
        total_loss+=loss
        print("Fold #{}: loss={}".format(i,loss))
    print("{}: Mean loss={:.2f} ({:.2f})".format(model.__class__.__name__,
                                                total_loss/len(folds)))
    dataset_blend_test[:, j] = fold_sums.mean(1)

print()
print("Blending models.")
blend = LogisticRegression(solver='lbfgs')
blend.fit(dataset_blend_train, y)
return blend.predict_proba(dataset_blend_test)

if __name__ == '__main__':
    np.random.seed(42) # seed to shuffle the train set

    print("Loading data...")
    URL = "https://data.heatonresearch.com/data/t81-558/kaggle/"

    df_train = pd.read_csv(
        URL+"bio_train.csv",
        na_values=['NA', '?'])

    df_submit = pd.read_csv(
        URL+"bio_test.csv",
        na_values=['NA', '?'])

    predictors = list(df_train.columns.values)
    predictors.remove('Activity')

```

```
x = df_train[predictors].values
y = df_train['Activity']
x_submit = df_submit.values

if SHUFFLE:
    idx = np.random.permutation(y.size)
    x = x[idx]
    y = y[idx]

submit_data = blend_ensemble(x, y, x_submit)
submit_data = stretch(submit_data)

#####
# Build submit file
#####
ids = [id+1 for id in range(submit_data.shape[0])]
submit_df = pd.DataFrame({'MoleculeId': ids,
                          'PredictedProbability':
                            submit_data[:, 1]},
                          columns=['MoleculeId',
                                    'PredictedProbability'])
submit_df.to_csv("submit.csv", index=False)
```

Loading data...

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:44: Deprecation
Warning: KerasClassifier is deprecated, use Sci-Keras (https://github.com/adriangb/scikeras) instead. See https://www.adriangb.com/scikeras/stable/migration.html for help migrating.
```

```
Model: 0 : <keras.wrappers.scikit_learn.KerasClassifier object at 0x7fc86980  
9610>  
106/106 [=====] - 1s 2ms/step - loss: 0.6048  
Fold #0: loss=0.5544745638322883  
106/106 [=====] - 1s 2ms/step - loss: 0.6046  
Fold #1: loss=0.5684765604955473  
106/106 [=====] - 1s 2ms/step - loss: 0.5943  
Fold #2: loss=0.5214491621944897  
106/106 [=====] - 1s 2ms/step - loss: 0.6301  
Fold #3: loss=0.5264746750391351  
106/106 [=====] - 1s 2ms/step - loss: 0.5905  
Fold #4: loss=0.5327822461352748  
106/106 [=====] - 1s 2ms/step - loss: 0.5993  
Fold #5: loss=0.5800157462831582  
106/106 [=====] - 1s 2ms/step - loss: 0.5877  
Fold #6: loss=0.5189563830365144  
106/106 [=====] - 1s 2ms/step - loss: 0.6038  
Fold #7: loss=0.5625417655617023  
106/106 [=====] - 1s 2ms/step - loss: 0.5935  
Fold #8: loss=0.5238374326475557  
106/106 [=====] - 1s 2ms/step - loss: 0.5991  
Fold #9: loss=0.5322226787930878  
KerasClassifier: Mean loss=0.5421231214018752  
Model: 1 : KNeighborsClassifier(n_neighbors=3)  
Fold #0: loss=3.606678388314123  
Fold #1: loss=2.2256421551487593  
Fold #2: loss=3.6815437059542186  
Fold #3: loss=2.416161292225968  
Fold #4: loss=4.442472310149748  
Fold #5: loss=4.321350530738247  
Fold #6: loss=3.400455469543658  
Fold #7: loss=3.1724147110842513  
Fold #8: loss=2.117356283193681  
Fold #9: loss=3.0532135963322586  
KNeighborsClassifier: Mean loss=3.243728844268491  
Model: 2 : RandomForestClassifier(n_jobs=-1)  
Fold #0: loss=0.4657177982691548  
Fold #1: loss=0.4346825805694879  
Fold #2: loss=0.4593868993445528  
Fold #3: loss=0.41674899522216713  
Fold #4: loss=0.4851849131056564  
Fold #5: loss=0.48473291073937  
Fold #6: loss=0.41274608628217674  
Fold #7: loss=0.47405291219252377  
Fold #8: loss=0.44974230059938286  
Fold #9: loss=0.46340159258241087  
RandomForestClassifier: Mean loss=0.45463969889068834  
Model: 3 : RandomForestClassifier(criterion='entropy', n_jobs=-1)  
Fold #0: loss=0.4511847247326708  
Fold #1: loss=0.42707704254926593  
Fold #2: loss=0.5550335199035183  
Fold #3: loss=0.42186970733328516  
Fold #4: loss=0.4794331756190797  
Fold #5: loss=0.4730559509802762  
Fold #6: loss=0.41116235817215196  
Fold #7: loss=0.46835919493314265
```

```
Fold #8: loss=0.4496144890690015
Fold #9: loss=0.4625902934553457
RandomForestClassifier: Mean loss=0.4599380456747738
Model: 4 : ExtraTreesClassifier(n_jobs=-1)
Fold #0: loss=0.45496751079363495
Fold #1: loss=0.5013051157905043
Fold #2: loss=0.5886179891724027
Fold #3: loss=0.41646902160044674
Fold #4: loss=0.4957910697444236
Fold #5: loss=0.4773401028797005
Fold #6: loss=0.41935061504547827
Fold #7: loss=0.5757908399174205
Fold #8: loss=0.4585195863412778
Fold #9: loss=0.6210675972963805
ExtraTreesClassifier: Mean loss=0.500921944858167
Model: 5 : ExtraTreesClassifier(criterion='entropy', n_jobs=-1)
Fold #0: loss=0.44825346440152214
Fold #1: loss=0.40764412171784686
Fold #2: loss=0.5819367378417363
Fold #3: loss=0.4140589874942631
Fold #4: loss=0.4923489720481471
Fold #5: loss=0.5744429921555051
Fold #6: loss=0.42334390524742155
Fold #7: loss=0.6409291880353659
Fold #8: loss=0.45627884947155956
Fold #9: loss=0.466653395317917
ExtraTreesClassifier: Mean loss=0.49058906137312847
Model: 6 : GradientBoostingClassifier(learning_rate=0.05, max_depth=6, n_estimators=50,
                                         subsample=0.5)
Fold #0: loss=0.4789324034433162
Fold #1: loss=0.4565636914381977
Fold #2: loss=0.47057741836357014
Fold #3: loss=0.4436328438944843
Fold #4: loss=0.4883293501002484
Fold #5: loss=0.4843521206311074
Fold #6: loss=0.4436043855503229
Fold #7: loss=0.45977393398397765
Fold #8: loss=0.46632256794136323
Fold #9: loss=0.4703354072414907
GradientBoostingClassifier: Mean loss=0.4662424122588078
```

Blending models.

In [12]:

T81-558: Applications of Deep Neural Networks

Module 8: Kaggle Data Sets

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

Module 8 Material

- Part 8.1: Introduction to Kaggle [\[Video\]](#) [\[Notebook\]](#)
- Part 8.2: Building Ensembles with Scikit-Learn and Keras [\[Video\]](#) [\[Notebook\]](#)
- **Part 8.3: How Should you Architect Your Keras Neural Network: Hyperparameters** [\[Video\]](#) [\[Notebook\]](#)
- Part 8.4: Bayesian Hyperparameter Optimization for Keras [\[Video\]](#) [\[Notebook\]](#)
- Part 8.5: Current Semester's Kaggle [\[Video\]](#) [\[Notebook\]](#)

Google CoLab Instructions

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

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

# Nicely formatted time string
def hms_string(sec_elapsed):
    h = int(sec_elapsed / (60 * 60))
    m = int((sec_elapsed % (60 * 60)) / 60)
    s = sec_elapsed % 60
    return "{}:{}{:02d}{}".format(h, m, s)
```

Note: not using Google CoLab

Part 8.3: Architecting Network: Hyperparameters

You have probably noticed several hyperparameters introduced previously in this course that you need to choose for your neural network. The number of layers, neuron counts per layer, layer types, and activation functions are all choices you must make to optimize your neural network. Some of the categories of hyperparameters for you to choose from coming from the following list:

- Number of Hidden Layers and Neuron Counts
- Activation Functions
- Advanced Activation Functions
- Regularization: L1, L2, Dropout
- Batch Normalization
- Training Parameters

The following sections will introduce each of these categories for Keras. While I will provide some general guidelines for hyperparameter selection, no two tasks are the same. You will benefit from experimentation with these values to determine what works best for your neural network. In the next part, we will see how machine learning can select some of these values independently.

Number of Hidden Layers and Neuron Counts

The structure of Keras layers is perhaps the hyperparameters that most become aware of first. How many layers should you have? How many neurons are on each layer? What activation function and layer type should you use? These are all questions that come up when designing a neural network. There are many different [types of layer](#) in Keras, listed here:

- **Activation** - You can also add activation functions as layers. Using the activation layer is the same as specifying the activation function as part of a Dense (or other) layer type.
- **ActivityRegularization** Used to add L1/L2 regularization outside of a layer. You can specify L1 and L2 as part of a Dense (or other) layer type.
- **Dense** - The original neural network layer type. In this layer type, every neuron connects to the next layer. The input vector is one-dimensional, and placing specific inputs next does not affect each other.
- **Dropout** - Dropout consists of randomly setting a fraction rate of input units to 0 at each update during training time, which helps prevent overfitting. Dropout only occurs during training.

- **Flatten** - Flattens the input to 1D and does not affect the batch size.
- **Input** - A Keras tensor is a tensor object from the underlying back end (Theano, TensorFlow, or CNTK), which we augment with specific attributes to build a Keras by knowing the inputs and outputs of the model.
- **Lambda** - Wraps arbitrary expression as a Layer object.
- **Masking** - Masks a sequence using a mask value to skip timesteps.
- **Permute** - Permutes the input dimensions according to a given pattern. Useful for tasks such as connecting RNNs and convolutional networks.
- **RepeatVector** - Repeats the input n times.
- **Reshape** - Similar to Numpy reshapes.
- **SpatialDropout1D** - This version performs the same function as Dropout; however, it drops entire 1D feature maps instead of individual elements.
- **SpatialDropout2D** - This version performs the same function as Dropout; however, it drops entire 2D feature maps instead of individual elements
- **SpatialDropout3D** - This version performs the same function as Dropout; however, it drops entire 3D feature maps instead of individual elements.

There is always trial and error for choosing a good number of neurons and hidden layers. Generally, the number of neurons on each layer will be larger closer to the hidden layer and smaller towards the output layer. This configuration gives the neural network a somewhat triangular or trapezoid appearance.

Activation Functions

Activation functions are a choice that you must make for each layer. Generally, you can follow this guideline:

- Hidden Layers - RELU
- Output Layer - Softmax for classification, linear for regression.

Some of the common activation functions in Keras are listed here:

- **softmax** - Used for multi-class classification. Ensures all output neurons behave as probabilities and sum to 1.0.
- **elu** - Exponential linear unit. Exponential Linear Unit or its widely known name ELU is a function that tends to converge cost to zero faster and produce more accurate results. Can produce negative outputs.
- **selu** - Scaled Exponential Linear Unit (SELU), essentially **elu** multiplied by a scaling constant.
- **softplus** - Softplus activation function. $\log(\exp(x) + 1)$ [Introduced](#) in 2001.
- **softsign** Softsign activation function. $x/(abs(x) + 1)$ Similar to tanh, but not widely used.

- **relu** - Very popular neural network activation function. Used for hidden layers, cannot output negative values. No trainable parameters.
- **tanh** Classic neural network activation function, though often replaced by relu family on modern networks.
- **sigmoid** - Classic neural network activation. Often used on output layer of a binary classifier.
- **hard_sigmoid** - Less computationally expensive variant of sigmoid.
- **exponential** - Exponential (base e) activation function.
- **linear** - Pass-through activation function. Usually used on the output layer of a regression neural network.

For more information about Keras activation functions refer to the following:

- [Keras Activation Functions](#)
- [Activation Function Cheat Sheets](#)

Advanced Activation Functions

Hyperparameters are not changed when the neural network trains. You, the network designer, must define the hyperparameters. The neural network learns regular parameters during neural network training. Neural network weights are the most common type of regular parameter. The "[advanced activation functions](#)," as Keras call them, also contain parameters that the network will learn during training. These activation functions may give you better performance than RELU.

- **LeakyReLU** - Leaky version of a Rectified Linear Unit. It allows a small gradient when the unit is not active, controlled by alpha hyperparameter.
- **PReLU** - Parametric Rectified Linear Unit, learns the alpha hyperparameter.

Regularization: L1, L2, Dropout

- [Keras Regularization](#)
- [Keras Dropout](#)

Batch Normalization

- [Keras Batch Normalization](#)
- Ioffe, S., & Szegedy, C. (2015). [Batch normalization: Accelerating deep network training by reducing internal covariate shift](#). *arXiv preprint arXiv:1502.03167*.

Normalize the activations of the previous layer at each batch, i.e. applies a transformation that maintains the mean activation close to 0 and the activation standard deviation close to 1. Can allow learning rate to be larger.

Training Parameters

- Keras Optimizers
- **Batch Size** - Usually small, such as 32 or so.
- **Learning Rate** - Usually small, 1e-3 or so.

In []:



T81-558: Applications of Deep Neural Networks

Module 8: Kaggle Data Sets

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

Module 8 Material

- Part 8.1: Introduction to Kaggle [\[Video\]](#) [\[Notebook\]](#)
- Part 8.2: Building Ensembles with Scikit-Learn and Keras [\[Video\]](#) [\[Notebook\]](#)
- Part 8.3: How Should you Architect Your Keras Neural Network: Hyperparameters [\[Video\]](#) [\[Notebook\]](#)
- **Part 8.4: Bayesian Hyperparameter Optimization for Keras** [\[Video\]](#) [\[Notebook\]](#)
- Part 8.5: Current Semester's Kaggle [\[Video\]](#) [\[Notebook\]](#)

Google CoLab Instructions

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

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

# Nicely formatted time string
def hms_string(sec_elapsed):
    h = int(sec_elapsed / (60 * 60))
    m = int((sec_elapsed % (60 * 60)) / 60)
    s = sec_elapsed % 60
    return "{}:{}{:02}:{:>05.2f}".format(h, m, s)
```

Note: using Google CoLab

Part 8.4: Bayesian Hyperparameter Optimization for Keras

Bayesian Hyperparameter Optimization is a method of finding hyperparameters more efficiently than a grid search. Because each candidate set of hyperparameters requires a retraining of the neural network, it is best to keep the number of candidate sets to a minimum. Bayesian Hyperparameter Optimization achieves this by training a model to predict good candidate sets of hyperparameters. [Cite:snoek2012practical]

- bayesian-optimization
- hyperopt
- spearmint

```
In [2]: # Ignore useless W0819 warnings generated by TensorFlow 2.0.
# Hopefully can remove this ignore in the future.
# See https://github.com/tensorflow/tensorflow/issues/31308
import logging, os
logging.disable(logging.WARNING)
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "3"

import pandas as pd
from scipy.stats import zscore

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

# Generate dummies for job
df = pd.concat([df,pd.get_dummies(df['job'],prefix="job")],axis=1)
df.drop('job', axis=1, inplace=True)

# Generate dummies for area
df = pd.concat([df,pd.get_dummies(df['area'],prefix="area")],axis=1)
df.drop('area', axis=1, inplace=True)

# Missing values for income
med = df['income'].median()
df['income'] = df['income'].fillna(med)

# Standardize ranges
df['income'] = zscore(df['income'])
df['aspect'] = zscore(df['aspect'])
df['save_rate'] = zscore(df['save_rate'])
df['age'] = zscore(df['age'])
df['subscriptions'] = zscore(df['subscriptions'])

# Convert to numpy - Classification
x_columns = df.columns.drop('product').drop('id')
```

```
x = df[x_columns].values
dummies = pd.get_dummies(df['product']) # Classification
products = dummies.columns
y = dummies.values
```

Now that we've preprocessed the data, we can begin the hyperparameter optimization. We start by creating a function that generates the model based on just three parameters. Bayesian optimization works on a vector of numbers, not on a problematic notion like how many layers and neurons are on each layer. To represent this complex neuron structure as a vector, we use several numbers to describe this structure.

- **dropout** - The dropout percent for each layer.
- **neuronPct** - What percent of our fixed 5,000 maximum number of neurons do we wish to use? This parameter specifies the total count of neurons in the entire network.
- **neuronShrink** - Neural networks usually start with more neurons on the first hidden layer and then decrease this count for additional layers. This percent specifies how much to shrink subsequent layers based on the previous layer. We stop adding more layers once we run out of neurons (the count specified by neuronPct).

These three numbers define the structure of the neural network. The commands in the below code show exactly how the program constructs the network.

In [3]:

```
import pandas as pd
import os
import numpy as np
import time
import tensorflow.keras.
import statistics
import tensorflow.keras
from sklearn import metrics
from sklearn.model_selection import StratifiedKFold
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation, Dropout, InputLayer
from tensorflow.keras import regularizers
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.model_selection import ShuffleSplit
from tensorflow.keras.layers import LeakyReLU, PReLU
from tensorflow.keras.optimizers import Adam

def generate_model(dropout, neuronPct, neuronShrink):
    # We start with some percent of 5000 starting neurons on
    # the first hidden layer.
    neuronCount = int(neuronPct * 5000)

    # Construct neural network
    model = Sequential()
```

```

# So long as there would have been at least 25 neurons and
# fewer than 10
# layers, create a new layer.
layer = 0
while neuronCount > 25 and layer < 10:
    # The first (0th) layer needs an input input_dim(neuronCount)
    if layer == 0:
        model.add(Dense(neuronCount,
                      input_dim=x.shape[1],
                      activation=PReLU()))
    else:
        model.add(Dense(neuronCount, activation=PReLU()))
    layer += 1

    # Add dropout after each hidden layer
    model.add(Dropout(dropout))

    # Shrink neuron count for each layer
    neuronCount = neuronCount * neuronShrink

model.add(Dense(y.shape[1], activation='softmax')) # Output
return model

```

We can test this code to see how it creates a neural network based on three such parameters.

In [4]:

```
# Generate a model and see what the resulting structure looks like.
model = generate_model(dropout=0.2, neuronPct=0.1, neuronShrink=0.25)
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
<hr/>		
dense (Dense)	(None, 500)	24500
dropout (Dropout)	(None, 500)	0
dense_1 (Dense)	(None, 125)	62750
dropout_1 (Dropout)	(None, 125)	0
dense_2 (Dense)	(None, 31)	3937
dropout_2 (Dropout)	(None, 31)	0
dense_3 (Dense)	(None, 7)	224
<hr/>		
Total params: 91,411		
Trainable params: 91,411		
Non-trainable params: 0		

We will now create a function to evaluate the neural network using three such parameters. We use bootstrapping because one training run might have "bad luck" with the assigned random weights. We use this function to train and then evaluate the neural network.

In [5]:

```
SPLITS = 2
EPOCHS = 500
PATIENCE = 10

def evaluate_network(dropout, learning_rate, neuronPct, neuronShrink):
    # Bootstrap

    # for Classification
    boot = StratifiedShuffleSplit(n_splits=SPLITS, test_size=0.1)
    # for Regression
    # boot = ShuffleSplit(n_splits=SPLITS, test_size=0.1)

    # Track progress
    mean_benchmark = []
    epochs_needed = []
    num = 0

    # Loop through samples
    for train, test in boot.split(x, df['product']):
        start_time = time.time()
        num+=1

        # Split train and test
        x_train = x[train]
        y_train = y[train]
        x_test = x[test]
        y_test = y[test]

        model = generate_model(dropout, neuronPct, neuronShrink)
        model.compile(loss='categorical_crossentropy',
                      optimizer=Adam(learning_rate=learning_rate))
        monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3,
                               patience=PATIENCE, verbose=0, mode='auto',
                               restore_best_weights=True)

        # Train on the bootstrap sample
        model.fit(x_train,y_train,validation_data=(x_test,y_test),
                  callbacks=[monitor],verbose=0,epochs=EPOCHS)
        epochs = monitor.stopped_epoch
        epochs_needed.append(epochs)

        # Predict on the out of boot (validation)
        pred = model.predict(x_test)

        # Measure this bootstrap's log loss
        y_compare = np.argmax(y_test, axis=1) # For log loss calculation
        score = metrics.log_loss(y_compare, pred)
        mean_benchmark.append(score)
        m1 = statistics.mean(mean_benchmark)
```

```

m2 = statistics.mean(epochs_needed)
mdev = statistics.pstdev(mean_benchmark)

# Record this iteration
time_took = time.time() - start_time

tensorflow.keras.backend.clear_session()

return (-m1)

```

You can try any combination of our three hyperparameters, plus the learning rate, to see how effective these four numbers are. Of course, our goal is not to manually choose different combinations of these four hyperparameters; we seek to automate.

```
In [6]: print(evaluate_network(
    dropout=0.2,
    learning_rate=1e-3,
    neuronPct=0.2,
    neuronShrink=0.2))
```

-0.6668764846259546

First, we must install the Bayesian optimization package if we are in Colab.

```
In [7]: # HIDE OUTPUT
!pip install bayesian-optimization
```

```
Requirement already satisfied: bayesian-optimization in /usr/local/lib/python3.7/dist-packages (1.2.0)
Requirement already satisfied: scipy>=0.14.0 in /usr/local/lib/python3.7/dist-packages (from bayesian-optimization) (1.4.1)
Requirement already satisfied: scikit-learn>=0.18.0 in /usr/local/lib/python3.7/dist-packages (from bayesian-optimization) (1.0.2)
Requirement already satisfied: numpy>=1.9.0 in /usr/local/lib/python3.7/dist-packages (from bayesian-optimization) (1.21.5)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.7/dist-packages (from scikit-learn>=0.18.0->bayesian-optimization) (3.1.0)
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.7/dist-packages (from scikit-learn>=0.18.0->bayesian-optimization) (1.1.0)
```

We will now automate this process. We define the bounds for each of these four hyperparameters and begin the Bayesian optimization. Once the program finishes, the best combination of hyperparameters found is displayed. The **optimize** function accepts two parameters that will significantly impact how long the process takes to complete:

- **n_iter** - How many steps of Bayesian optimization that you want to perform. The more steps, the more likely you will find a reasonable maximum.
- **init_points**: How many steps of random exploration that you want to perform. Random exploration can help by diversifying the exploration space.

```
In [8]: from bayes_opt import BayesianOptimization
import time

# Suppress NaN warnings
import warnings
warnings.filterwarnings("ignore", category =RuntimeWarning)

# Bounded region of parameter space
pbounds = {'dropout': (0.0, 0.499),
            'learning_rate': (0.0, 0.1),
            'neuronPct': (0.01, 1),
            'neuronShrink': (0.01, 1)
            }

optimizer = BayesianOptimization(
    f=evaluate_network,
    pbounds=pbounds,
    verbose=2, # verbose = 1 prints only when a maximum
    # is observed, verbose = 0 is silent
    random_state=1,
)

start_time = time.time()
optimizer.maximize(init_points=10, n_iter=20,)
time_took = time.time() - start_time

print(f"Total runtime: {hms_string(time_took)}")
print(optimizer.max)
```

iter	target	dropout	learni...	neuronPct	neuron...
1	-0.8092	0.2081	0.07203	0.01011	0.3093
2	-0.7167	0.07323	0.009234	0.1944	0.3521
3	-17.87	0.198	0.05388	0.425	0.6884
4	-0.8022	0.102	0.08781	0.03711	0.6738
5	-0.9209	0.2082	0.05587	0.149	0.2061
6	-17.96	0.3996	0.09683	0.3203	0.6954
7	-4.223	0.4373	0.08946	0.09419	0.04866
8	-0.7025	0.08475	0.08781	0.1074	0.4269
9	-8.666	0.478	0.05332	0.695	0.3224
10	-9.785	0.3426	0.08346	0.02811	0.7526
11	-4.881	0.0	0.0	0.01	0.01
12	-21.59	0.2208	0.04135	0.5523	0.7468
13	-1.819	0.0	0.0	1.0	0.01
14	-33.33	0.01058	0.08079	0.9652	0.7051
15	-1.418	0.4963	0.02476	0.9744	0.01896
16	-1.876	0.1247	0.0	0.5781	0.01
17	-1.898	0.0	0.0	0.01	1.0
18	-17.7	0.1621	0.06358	0.4065	0.754
19	-2.674	0.0	0.0	0.01	0.5464
20	-1.931	0.499	0.0	0.5538	0.01
21	-3.402	0.004722	0.05502	0.1704	0.483
22	-2.8	0.08639	0.0838	0.04668	0.6864
23	-20.98	0.1168	0.0447	0.5546	0.9497
24	-4.565	0.2554	0.1	0.8418	0.01
25	-4.724	0.0	0.1	0.3368	0.01
26	-0.6956	0.2505	0.007623	0.01265	0.523
27	-0.7139	0.2967	0.01162	0.3735	0.01708
28	-2.145	0.499	0.0	0.3053	0.2207
29	-2.069	0.0	0.0	0.4808	0.2473
30	-0.7155	0.4082	0.01635	0.02488	0.1694

Total runtime: 1:36:11.56

```
{'target': -0.6955536706512794, 'params': {'dropout': 0.2504561773412203, 'learning_rate': 0.0076232346709142924, 'neuronPct': 0.012648791521811826, 'neuronShrink': 0.5229748831552032}}
```

As you can see, the algorithm performed 30 total iterations. This total iteration count includes ten random and 20 optimization iterations.



T81-558: Applications of Deep Neural Networks

Module 8: Kaggle Data Sets

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

Module 8 Material

- Part 8.1: Introduction to Kaggle [\[Video\]](#) [\[Notebook\]](#)
- Part 8.2: Building Ensembles with Scikit-Learn and Keras [\[Video\]](#) [\[Notebook\]](#)
- Part 8.3: How Should you Architect Your Keras Neural Network: Hyperparameters [\[Video\]](#) [\[Notebook\]](#)
- Part 8.4: Bayesian Hyperparameter Optimization for Keras [\[Video\]](#) [\[Notebook\]](#)
- **Part 8.5: Current Semester's Kaggle** [\[Video\]](#) [\[Notebook\]](#)

Google CoLab Instructions

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

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

# Nicely formatted time string
def hms_string(sec_elapsed):
    h = int(sec_elapsed / (60 * 60))
    m = int((sec_elapsed % (60 * 60)) / 60)
    s = sec_elapsed % 60
    return "{}:{}{:02}:{:>05.2f}".format(h, m, s)
```

Note: using Google CoLab

Part 8.5: Current Semester's Kaggle

Kaggke competition site for current semester:

- [Spring 2023 Kaggle Assignment](#)

Previous Kaggle competition sites for this class (NOT this semester's assignment, feel free to use code):

- [Fall 2022 Kaggle Assignment](#)
- [Spring 2022 Kaggle Assignment](#)
- [Fall 2021 Kaggle Assignment](#)
- [Spring 2021 Kaggle Assignment](#)
- [Fall 2020 Kaggle Assignment](#)
- [Spring 2020 Kaggle Assignment](#)
- [Fall 2019 Kaggle Assignment](#)
- [Spring 2019 Kaggle Assignment](#)
- [Fall 2018 Kaggle Assignment](#)
- [Spring 2018 Kaggle Assignment](#)
- [Fall 2017 Kaggle Assignment](#)
- [Spring 2017 Kaggle Assignment](#)
- [Fall 2016 Kaggle Assignment](#)

Iris as a Kaggle Competition

If I used the Iris data as a Kaggle, I would give you the following three files:

- [kaggle_iris_test.csv](#) - The data that Kaggle will evaluate you on. It contains only input; you must provide answers. (contains x)
- [kaggle_iris_train.csv](#) - The data that you will use to train. (contains x and y)
- [kaggle_iris_sample.csv](#) - A sample submission for Kaggle. (contains x and y)

Important features of the Kaggle iris files (that differ from how we've previously seen files):

- The iris species is already index encoded.
- Your training data is in a separate file.
- You will load the test data to generate a submission file.

The following program generates a submission file for "Iris Kaggle". You can use it as a starting point for assignment 3.

In [2]:

```
import os
import pandas as pd
```

```

from sklearn.model_selection import train_test_split
import tensorflow as tf
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping

df_train = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/datasets/+"\
    "kaggle_iris_train.csv", na_values=['NA', '?'])

# Encode feature vector
df_train.drop('id', axis=1, inplace=True)

num_classes = len(df_train.groupby('species').species.nunique())

print("Number of classes: {}".format(num_classes))

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

# Split into train/test
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=45)

# Train, with early stopping
model = Sequential()
model.add(Dense(50, input_dim=x.shape[1], activation='relu'))
model.add(Dense(25))
model.add(Dense(y.shape[1], activation='softmax'))
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=0,epochs=1000)

```

Number of classes: 3
Restoring model weights from the end of the best epoch: 103.
Epoch 108: early stopping

Out[2]: <keras.callbacks.History at 0x7f05e7452710>

Now that we've trained the neural network, we can check its log loss.

In [3]:

```

from sklearn import metrics

# Calculate multi log loss error
pred = model.predict(x_test)
score = metrics.log_loss(y_test, pred)
print("Log loss score: {}".format(score))

```

```
Log loss score: 0.10988010508939623
```

Now we are ready to generate the Kaggle submission file. We will use the iris test data that does not contain a y target value. It is our job to predict this value and submit it to Kaggle.

```
In [4]: # Generate Kaggle submit file

# Encode feature vector
df_test = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/datasets/+\n    \"kaggle_iris_test.csv\", na_values=['NA','?'])

# Convert to numpy - Classification
ids = df_test['id']
df_test.drop('id', axis=1, inplace=True)
x = df_test[['sepal_l', 'sepal_w', 'petal_l', 'petal_w']].values
y = dummies.values

# Generate predictions
pred = model.predict(x)
#pred

# Create submission data set

df_submit = pd.DataFrame(pred)
df_submit.insert(0,'id',ids)
df_submit.columns = ['id','species-0','species-1','species-2']

# Write submit file locally
df_submit.to_csv("iris_submit.csv", index=False)

print(df_submit[:5])
```

	id	species-0	species-1	species-2
0	100	0.022300	0.777859	0.199841
1	101	0.001309	0.273849	0.724842
2	102	0.001153	0.319349	0.679498
3	103	0.958006	0.041989	0.000005
4	104	0.976932	0.023066	0.000002

MPG as a Kaggle Competition (Regression)

If the Auto MPG data were used as a Kaggle, you would be given the following three files:

- [kaggle_mpg_test.csv](#) - The data that Kaggle will evaluate you on. Contains only input, you must provide answers. (contains x)
- [kaggle_mpg_train.csv](#) - The data that you will use to train. (contains x and y)
- [kaggle_mpg_sample.csv](#) - A sample submission for Kaggle. (contains x and y)

Important features of the Kaggle iris files (that differ from how we've previously seen files):

The following program generates a submission file for "MPG Kaggle".

In [5]:

```
# HIDE OUTPUT
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from sklearn.model_selection import train_test_split
from tensorflow.keras.callbacks import EarlyStopping
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/datasets/+\\" +
    "kaggle_auto_train.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 train/test
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),
          verbose=2,callbacks=[monitor],epochs=1000)

# Predict
pred = model.predict(x_test)
```

Epoch 1/1000
9/9 - 1s - loss: 1797.5945 - val_loss: 1272.4421 - 1s/epoch - 144ms/step
Epoch 2/1000
9/9 - 0s - loss: 574.7726 - val_loss: 734.3082 - 92ms/epoch - 10ms/step
Epoch 3/1000
9/9 - 0s - loss: 487.3118 - val_loss: 446.3558 - 76ms/epoch - 8ms/step
Epoch 4/1000
9/9 - 0s - loss: 326.7128 - val_loss: 321.7191 - 96ms/epoch - 11ms/step
Epoch 5/1000
9/9 - 0s - loss: 294.8217 - val_loss: 271.3473 - 70ms/epoch - 8ms/step
Epoch 6/1000
9/9 - 0s - loss: 259.8376 - val_loss: 239.6796 - 116ms/epoch - 13ms/step
Epoch 7/1000
9/9 - 0s - loss: 250.4708 - val_loss: 227.4295 - 73ms/epoch - 8ms/step
Epoch 8/1000
9/9 - 0s - loss: 227.1252 - val_loss: 198.4167 - 125ms/epoch - 14ms/step
Epoch 9/1000
9/9 - 0s - loss: 225.6681 - val_loss: 195.5055 - 95ms/epoch - 11ms/step
Epoch 10/1000
9/9 - 0s - loss: 209.1198 - val_loss: 184.1092 - 121ms/epoch - 13ms/step
Epoch 11/1000
9/9 - 0s - loss: 195.4801 - val_loss: 176.0311 - 108ms/epoch - 12ms/step
Epoch 12/1000
9/9 - 0s - loss: 198.6493 - val_loss: 168.1613 - 163ms/epoch - 18ms/step
Epoch 13/1000
9/9 - 0s - loss: 198.5606 - val_loss: 196.0306 - 114ms/epoch - 13ms/step
Epoch 14/1000
9/9 - 0s - loss: 184.3067 - val_loss: 179.8450 - 99ms/epoch - 11ms/step
Epoch 15/1000
9/9 - 0s - loss: 178.6627 - val_loss: 148.1014 - 80ms/epoch - 9ms/step
Epoch 16/1000
9/9 - 0s - loss: 154.0201 - val_loss: 129.9253 - 74ms/epoch - 8ms/step
Epoch 17/1000
9/9 - 0s - loss: 145.2373 - val_loss: 124.0609 - 79ms/epoch - 9ms/step
Epoch 18/1000
9/9 - 0s - loss: 140.0318 - val_loss: 116.7844 - 86ms/epoch - 10ms/step
Epoch 19/1000
9/9 - 0s - loss: 135.1688 - val_loss: 115.0745 - 136ms/epoch - 15ms/step
Epoch 20/1000
9/9 - 0s - loss: 132.8391 - val_loss: 106.9831 - 169ms/epoch - 19ms/step
Epoch 21/1000
9/9 - 0s - loss: 123.6673 - val_loss: 105.7211 - 95ms/epoch - 11ms/step
Epoch 22/1000
9/9 - 0s - loss: 123.7169 - val_loss: 99.6713 - 112ms/epoch - 12ms/step
Epoch 23/1000
9/9 - 0s - loss: 118.0815 - val_loss: 96.0683 - 150ms/epoch - 17ms/step
Epoch 24/1000
9/9 - 0s - loss: 114.6363 - val_loss: 99.1486 - 153ms/epoch - 17ms/step
Epoch 25/1000
9/9 - 0s - loss: 112.3965 - val_loss: 93.8642 - 180ms/epoch - 20ms/step
Epoch 26/1000
9/9 - 0s - loss: 111.2470 - val_loss: 88.3417 - 139ms/epoch - 15ms/step
Epoch 27/1000
9/9 - 0s - loss: 107.8639 - val_loss: 86.7927 - 135ms/epoch - 15ms/step
Epoch 28/1000
9/9 - 0s - loss: 103.0426 - val_loss: 89.0441 - 101ms/epoch - 11ms/step

Epoch 29/1000
9/9 - 0s - loss: 110.6277 - val_loss: 82.4294 - 159ms/epoch - 18ms/step
Epoch 30/1000
9/9 - 0s - loss: 100.3681 - val_loss: 90.8037 - 82ms/epoch - 9ms/step
Epoch 31/1000
9/9 - 0s - loss: 105.4711 - val_loss: 79.2106 - 76ms/epoch - 8ms/step
Epoch 32/1000
9/9 - 0s - loss: 98.7603 - val_loss: 79.9620 - 73ms/epoch - 8ms/step
Epoch 33/1000
9/9 - 0s - loss: 94.7678 - val_loss: 76.8616 - 78ms/epoch - 9ms/step
Epoch 34/1000
9/9 - 0s - loss: 93.8199 - val_loss: 77.0823 - 76ms/epoch - 8ms/step
Epoch 35/1000
9/9 - 0s - loss: 94.8746 - val_loss: 73.9967 - 62ms/epoch - 7ms/step
Epoch 36/1000
9/9 - 0s - loss: 95.3178 - val_loss: 73.0059 - 60ms/epoch - 7ms/step
Epoch 37/1000
9/9 - 0s - loss: 91.1315 - val_loss: 80.8389 - 57ms/epoch - 6ms/step
Epoch 38/1000
9/9 - 0s - loss: 96.4810 - val_loss: 77.8854 - 59ms/epoch - 7ms/step
Epoch 39/1000
9/9 - 0s - loss: 91.1039 - val_loss: 69.9539 - 40ms/epoch - 4ms/step
Epoch 40/1000
9/9 - 0s - loss: 86.9596 - val_loss: 69.3511 - 43ms/epoch - 5ms/step
Epoch 41/1000
9/9 - 0s - loss: 87.6142 - val_loss: 70.1390 - 57ms/epoch - 6ms/step
Epoch 42/1000
9/9 - 0s - loss: 88.0185 - val_loss: 73.6168 - 38ms/epoch - 4ms/step
Epoch 43/1000
9/9 - 0s - loss: 92.8655 - val_loss: 67.5213 - 38ms/epoch - 4ms/step
Epoch 44/1000
9/9 - 0s - loss: 88.5278 - val_loss: 69.9708 - 59ms/epoch - 7ms/step
Epoch 45/1000
9/9 - 0s - loss: 82.9339 - val_loss: 70.3786 - 39ms/epoch - 4ms/step
Epoch 46/1000
9/9 - 0s - loss: 81.7092 - val_loss: 63.3550 - 59ms/epoch - 7ms/step
Epoch 47/1000
9/9 - 0s - loss: 81.1514 - val_loss: 78.7681 - 59ms/epoch - 7ms/step
Epoch 48/1000
9/9 - 0s - loss: 99.3562 - val_loss: 62.8894 - 64ms/epoch - 7ms/step
Epoch 49/1000
9/9 - 0s - loss: 96.8292 - val_loss: 67.8047 - 55ms/epoch - 6ms/step
Epoch 50/1000
9/9 - 0s - loss: 88.7995 - val_loss: 67.5249 - 56ms/epoch - 6ms/step
Epoch 51/1000
9/9 - 0s - loss: 80.6064 - val_loss: 96.2975 - 58ms/epoch - 6ms/step
Epoch 52/1000
9/9 - 0s - loss: 95.2732 - val_loss: 62.4323 - 39ms/epoch - 4ms/step
Epoch 53/1000
9/9 - 0s - loss: 75.1992 - val_loss: 64.0174 - 39ms/epoch - 4ms/step
Epoch 54/1000
9/9 - 0s - loss: 75.5173 - val_loss: 57.8594 - 40ms/epoch - 4ms/step
Epoch 55/1000
9/9 - 0s - loss: 72.6369 - val_loss: 56.2216 - 47ms/epoch - 5ms/step
Epoch 56/1000
9/9 - 0s - loss: 72.8636 - val_loss: 55.3956 - 54ms/epoch - 6ms/step

Epoch 57/1000
9/9 - 0s - loss: 69.0251 - val_loss: 70.7940 - 56ms/epoch - 6ms/step
Epoch 58/1000
9/9 - 0s - loss: 75.8152 - val_loss: 63.7728 - 37ms/epoch - 4ms/step
Epoch 59/1000
9/9 - 0s - loss: 71.6866 - val_loss: 59.5908 - 41ms/epoch - 5ms/step
Epoch 60/1000
9/9 - 0s - loss: 69.3349 - val_loss: 52.7848 - 38ms/epoch - 4ms/step
Epoch 61/1000
9/9 - 0s - loss: 67.8410 - val_loss: 53.5977 - 54ms/epoch - 6ms/step
Epoch 62/1000
9/9 - 0s - loss: 68.4640 - val_loss: 53.6664 - 39ms/epoch - 4ms/step
Epoch 63/1000
9/9 - 0s - loss: 63.7229 - val_loss: 52.4224 - 44ms/epoch - 5ms/step
Epoch 64/1000
9/9 - 0s - loss: 69.8485 - val_loss: 59.1973 - 53ms/epoch - 6ms/step
Epoch 65/1000
9/9 - 0s - loss: 75.7193 - val_loss: 70.1342 - 37ms/epoch - 4ms/step
Epoch 66/1000
9/9 - 0s - loss: 87.7418 - val_loss: 55.3687 - 38ms/epoch - 4ms/step
Epoch 67/1000
9/9 - 0s - loss: 72.8599 - val_loss: 52.9028 - 44ms/epoch - 5ms/step
Epoch 68/1000
9/9 - 0s - loss: 69.9528 - val_loss: 49.9109 - 38ms/epoch - 4ms/step
Epoch 69/1000
9/9 - 0s - loss: 62.7782 - val_loss: 46.6361 - 39ms/epoch - 4ms/step
Epoch 70/1000
9/9 - 0s - loss: 58.4024 - val_loss: 50.8190 - 38ms/epoch - 4ms/step
Epoch 71/1000
9/9 - 0s - loss: 63.5687 - val_loss: 46.6161 - 44ms/epoch - 5ms/step
Epoch 72/1000
9/9 - 0s - loss: 65.9290 - val_loss: 47.1278 - 40ms/epoch - 4ms/step
Epoch 73/1000
9/9 - 0s - loss: 74.9235 - val_loss: 61.1282 - 42ms/epoch - 5ms/step
Epoch 74/1000
9/9 - 0s - loss: 63.6773 - val_loss: 45.0233 - 39ms/epoch - 4ms/step
Epoch 75/1000
9/9 - 0s - loss: 55.8287 - val_loss: 59.8986 - 41ms/epoch - 5ms/step
Epoch 76/1000
9/9 - 0s - loss: 58.9969 - val_loss: 52.0535 - 39ms/epoch - 4ms/step
Epoch 77/1000
9/9 - 0s - loss: 60.7104 - val_loss: 43.0530 - 46ms/epoch - 5ms/step
Epoch 78/1000
9/9 - 0s - loss: 59.7358 - val_loss: 45.3669 - 41ms/epoch - 5ms/step
Epoch 79/1000
9/9 - 0s - loss: 60.9792 - val_loss: 40.7967 - 41ms/epoch - 5ms/step
Epoch 80/1000
9/9 - 0s - loss: 58.0294 - val_loss: 49.0612 - 42ms/epoch - 5ms/step
Epoch 81/1000
9/9 - 0s - loss: 57.6733 - val_loss: 41.7604 - 44ms/epoch - 5ms/step
Epoch 82/1000
9/9 - 0s - loss: 50.3309 - val_loss: 39.1461 - 38ms/epoch - 4ms/step
Epoch 83/1000
9/9 - 0s - loss: 54.2316 - val_loss: 40.8561 - 36ms/epoch - 4ms/step
Epoch 84/1000
9/9 - 0s - loss: 66.4084 - val_loss: 38.1869 - 60ms/epoch - 7ms/step

```
Epoch 85/1000
9/9 - 0s - loss: 50.0778 - val_loss: 37.8852 - 56ms/epoch - 6ms/step
Epoch 86/1000
9/9 - 0s - loss: 47.0763 - val_loss: 37.3743 - 39ms/epoch - 4ms/step
Epoch 87/1000
9/9 - 0s - loss: 46.1752 - val_loss: 45.8444 - 45ms/epoch - 5ms/step
Epoch 88/1000
9/9 - 0s - loss: 49.4047 - val_loss: 37.3778 - 40ms/epoch - 4ms/step
Epoch 89/1000
9/9 - 0s - loss: 46.5478 - val_loss: 36.2859 - 38ms/epoch - 4ms/step
Epoch 90/1000
9/9 - 0s - loss: 44.7429 - val_loss: 47.2213 - 38ms/epoch - 4ms/step
Epoch 91/1000
9/9 - 0s - loss: 49.7726 - val_loss: 52.5501 - 42ms/epoch - 5ms/step
Epoch 92/1000
9/9 - 0s - loss: 53.5449 - val_loss: 62.3078 - 57ms/epoch - 6ms/step
Epoch 93/1000
9/9 - 0s - loss: 54.7558 - val_loss: 51.2010 - 43ms/epoch - 5ms/step
Epoch 94/1000
Restoring model weights from the end of the best epoch: 89.
9/9 - 0s - loss: 52.3631 - val_loss: 42.2640 - 69ms/epoch - 8ms/step
Epoch 94: early stopping
```

Now that we've trained the neural network, we can check its RMSE error.

```
In [6]: import numpy as np

# Measure RMSE error. RMSE is common for regression.
score = np.sqrt(metrics.mean_squared_error(pred,y_test))
print("Final score (RMSE): {}".format(score))
```

```
Final score (RMSE): 6.023776405947501
```

Now we are ready to generate the Kaggle submission file. We will use the MPG test data that does not contain a y target value. It is our job to predict this value and submit it to Kaggle.

```
In [7]: import pandas as pd

# Generate Kaggle submit file

# Encode feature vector
df_test = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/datasets/+"\
    "kaggle_auto_test.csv", na_values=['NA','?'])

# Convert to numpy - regression
ids = df_test['id']
df_test.drop('id', axis=1, inplace=True)

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

x = df_test[['cylinders', 'displacement', 'horsepower', 'weight',
```

```
'acceleration', 'year', 'origin']].values

# Generate predictions
pred = model.predict(x)
#pred

# Create submission data set

df_submit = pd.DataFrame(pred)
df_submit.insert(0,'id',ids)
df_submit.columns = ['id','mpg']

# Write submit file locally
df_submit.to_csv("auto_submit.csv", index=False)

print(df_submit[:5])
```

	<code>id</code>	<code>mpg</code>
0	350	27.158819
1	351	24.450621
2	352	24.913355
3	353	26.994867
4	354	26.669268