CSC 180 Intelligent Systems (Fall 2022)

Dr. Haiquan Chen, Dept of Computer Scicence

California State University, Sacramento

Lab 0: Python and Python Syntax

Please follow the instructions in the slides to install Anaconda and TensorFLow

Anaconda is the world's most popular Python data science platform, with 1,000+ data science packages.

Test your enviroment by runing the following code:

In [1]:
```
# What version of Python do you have?

import tensorflow as tf
import sys
import sklearn as sk
import pandas as pd

print("Python {}".format(sys.version))
print('Pandas {}'.format(pd.__version__))
print('Scikit-Learn {}'.format(sk.__version__))
print()
print("Tensor Flow Version: {}".format(tf.__version__))
print("Keras Version: {}".format(tf.keras.__version__))
```

```
Python 3.8.8 (default, Apr 13 2021, 15:08:03) [MSC v.1916 64 bit (AMD64)]
Pandas 1.2.4
Scikit-Learn 0.24.1

Tensor Flow Version: 2.3.0
Keras Version: 2.4.0
```

Python Syntax

In [3]:
```
a = 4 # assign value 4 to variable a
```

In [4]:
```
print (a)
```

```
4
```

In [5]:
```
type(a)  # what is the type of a?
```

Out[5]:
```
int
```

Variable names in Python can contain alphanumerical characters `a-z`, `A-Z`, `0-9` and some

special characters such as `_`. Normal variable names must start with a letter.

By convension, variable names start with a lower-case letter, and Class names start with a capital letter.

In addition, there are a number of Python keywords that cannot be used as variable names. These keywords are:

```
and, as, assert, break, class, continue, def, del, elif, else, except,
exec, finally, for, from, global, if, import, in, is, lambda, not, or,
pass, print, raise, return, try, while, with, yield
```

NOTE

In [5]:
```
int a = 1;  # in C
```

```
  File "<ipython-input-5-d7c31ab15148>", line 1
    int a = 1;  # in C
        ^
SyntaxError: invalid syntax
```

In [7]:
```
c = 2.1
print (type(c))
```

```
<class 'float'>
```

Modules

Most of the functionality in Python is provided by *modules*.

The Python Standard Library is a large collection of modules to access the operating system, do file I/O, string management, network communication, and much more.

To use a module in a python program it first has to be imported. A module can be imported using the `import` statement. For example, to import the module `math`, which contains many standard mathematical functions, we can do:

In [8]:
```
import math
```

This includes the whole module and makes it available for use later in the program. For example, we can do:

In [10]:
```
math.cos(2 * math.pi)
```

Out[10]:
```
1.0
```

In notebook, use "?"

In [12]:
```
math.sqrt?
```

In [13]:
```
math.sqrt(10)
```

Out[13]:
```
3.1622776601683795
```

Operators and comparisons

Most operators and comparisons in Python work as one would expect:

- Arithmetic operators `+`, `-`, `*`, `/`, `//` (integer division), '**' power

In [14]:
```
print (1*2, 1+1, 1*3, 3//2, 2**2)
```

```
2 2 3 1 4
```

- Comparison operators `>`, `<`, `>=` (greater or equal), `<=` (less or equal), `==` equality, `is` identical.

In [15]:
```
2 > 1, 2 < 1
```

Out[15]:
```
(True, False)
```

In [16]:
```
# equality
1 == 1
```

Out[16]:
```
True
```

Compound types: Strings, List and dictionaries

Strings

Strings are the variable type that is used for storing text messages.

In [17]:
```
s = "Hello world"
# or
s = 'Hello world'
# or
s = '''Hello world'''

type(s)
```

Out[17]:
```
str
```

In [18]:
```
# length of the string: the number of characters
len(s)
```

Out[18]:
```
11
```

In [19]:
```
# replace a substring in a string with somethign else
s2 = s.replace("world", "csc180")
print (s2)
```

```
Hello csc180
```

We can index a character in a string using `[]`:

In [20]:
```
s[0]
```

Out[20]:
```
'H'
```

Indexing start at 0!

We can use extract a part of a string using the syntax `[start:stop]`, which extracts characters between index `start` and `stop`:

In [21]:
```
s[0:5]
```

Out[21]:
```
'Hello'
```

If we omit either (or both) of `start` or `stop` from `[start:stop]`, the default is the beginning and the end of the string, respectively:

In [22]:
```
s[:5]
```

Out[22]:
```
'Hello'
```

In [23]:
```
s[6:]
```

Out[23]:
```
'world'
```

In [24]:
```
s[:]
```

Out[24]:
```
'Hello world'
```

We can also define the step size using the syntax `[start:end:step]` (the default value for `step` is 1, as we saw above). This technique is called *string slicing*.

In [25]:
```
s[::1]
```

Out[25]:
```
'Hello world'
```

In [26]:
```
s[::2]
```

Out[26]:
```
'Hlowrd'
```

Python has a very rich set of functions for text processing. See for example

http://docs.python.org/2/library/string.html for more information.

In [27]:
```
"str1" + "str2" + "str3" # strings added with + are concatenated without space
```

Out[27]:
```
'str1str2str3'
```

C-style string formatting:

In [28]:
```
"value = %f" % 1.0       # we can use C-style string formatting
```

Out[28]:
```
'value = 1.000000'
```

In [29]:
```
# this formatting creates a string
"value1 = %.2f. value2 = %d" % (3.1415, 1.5)
```

Out[29]:
```
'value1 = 3.14. value2 = 1'
```

List

Lists are very similar to strings, except that ***each element can be of any type***.

The syntax for creating lists in Python is `[...]`:

In [30]:
```
l = [1, 2, 3, 4]
```

We can use the same slicing techniques to manipulate lists as we could use on strings:

In [31]:
```
print (l)

print (l[1:3])

print (l[::2])
```

```
[1, 2, 3, 4]
[2, 3]
[1, 3]
```

Elements in a list do not all have to be of the same type:

In [32]:
```
l = [1, 'a', 1.0, 1-1j]
l
```

Out[32]:
```
[1, 'a', 1.0, (1-1j)]
```

Python lists can be inhomogeneous and arbitrarily nested:

Lists play a very important role in Python, and are for example used in loops and other flow control structures (discussed below). There are number of convenient functions for generating lists of various types, for example the `range` function:

Convert a string to a list by type casting

In [33]:
```
# convert a string to a list by type casting:
s2 = list(s)
s2
```

Out[33]:
```
['H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
```

In [34]:
```
# sorting lists (inplace)
s2.sort()
print (s2)
```

```
[' ', 'H', 'd', 'e', 'l', 'l', 'l', 'o', 'o', 'r', 'w']
```

Adding, inserting, modifying, and removing elements from lists

In [35]:
```
# create a new empty list
l = []

# add an elements using `append`
l.append("A")
l.append("d")
l.append("d")

l
```

Out[35]:
```
['A', 'd', 'd']
```

We can modify lists by assigning new values to elements in the list. In technical jargon, lists are *mutable*.

In [36]:
```
l[1] = "p"
l[2] = "p"

l
```

Out[36]:
```
['A', 'p', 'p']
```

Insert at element a specific index using `insert`

In [37]:
```
l.insert(0, "i")
l.insert(1, "n")
l.insert(2, "s")
l.insert(3, "e")
l.insert(4, "r")
l.insert(5, "t")

l
```

Out[37]:
```
['i', 'n', 's', 'e', 'r', 't', 'A', 'p', 'p']
```

Remove first element with specific value using 'remove'

In [38]:
```
l.remove("A")
l
```

Out[38]:
```
['i', 'n', 's', 'e', 'r', 't', 'p', 'p']
```

Remove an element at a specific location using `del`:

In [39]:
```
del l[7]
del l[6]

l
```

Out[39]:
```
['i', 'n', 's', 'e', 'r', 't']
```

See `help(list)` for more details, or read the online documentation

Tuples

Tuples are like lists, except that they cannot be modified once created, that is they are ***immutable***.

In Python, tuples are created using the syntax `(..., ..., ...)`, or even `..., ...`:

In [40]:
```
point = (10, 20)

print (type(point))
```

```
<class 'tuple'>
```

If we try to assign a new value to an element in a tuple we get an error:

In [41]:
```
point[0] = 20
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-41-9734b1daa940> in <module>
----> 1 point[0] = 20

TypeError: 'tuple' object does not support item assignment
```

Dictionaries

Dictionaries are ***lists of key-value pairs***. The syntax for lists are `{key1 : value1, ...}`:

In [42]:
```
params = {"parameter1" : 1.0,
          "parameter2" : 2.0,
          "parameter3" : 3.0,}

# or equivalent

params = dict(parameter1=1.0, parameter2=2.0, parameter3=3.0)

params
```

Out[42]:
```
{'parameter1': 1.0, 'parameter2': 2.0, 'parameter3': 3.0}
```

In [43]:
```
#Or this:
```

```
params  = dict([['parameter1', 1.0], ['parameter2', 2.0], ['parameter3', 3.0]])
params
```

Out[43]:
```
{'parameter1': 1.0, 'parameter2': 2.0, 'parameter3': 3.0}
```

In [44]:
```
print ("parameter1 =", params["parameter1"])
print ("parameter2 =", params["parameter2"])
print ("parameter3 =", params["parameter3"])
```

```
parameter1 = 1.0
parameter2 = 2.0
parameter3 = 3.0
```

In [45]:
```
params["parameter1"] = "A"
params["parameter2"] = "B"

# add a new entry
params["parameter4"] = "D"

params
```

Out[45]:
```
{'parameter1': 'A', 'parameter2': 'B', 'parameter3': 3.0, 'parameter4': 'D'}
```

In [46]:
```
"parameter1" in params
```

Out[46]:
```
True
```

In [47]:
```
# delete a key
del params["parameter3"]
params
```

Out[47]:
```
{'parameter1': 'A', 'parameter2': 'B', 'parameter4': 'D'}
```

Control Flow

Conditional statements: if, elif, else

The Python syntax for conditional execution of code use the keywords `if`, `elif` (else if), `else`:

In [48]:
```
statement1 = False
statement2 = False

if statement1:
    print ("statement1 is True")

elif statement2:
    print ("statement2 is True")

else:
    print ("statement1 and statement2 are False")
```

```
statement1 and statement2 are False
```

Notice that in the Python programming language, ***program blocks are defined by their indentation level***.

Compare to the equivalent Java code:

```
if (statement1)
{
    printf("statement1 is True\n");
}
else if (statement2)
{
    printf("statement2 is True\n");
}
else
{
    printf("statement1 and statement2 are False\n");
}
```

In C blocks are defined by the enclosing curly brakets `{` and `}`. And the level of indentation (white space before the code statements) does not matter (completely optional).

But in Python, the extent of a code block is defined by the indentation level (ideally four white spaces).

This means that we have to **be careful to indent our code correctly**, or else we will get syntax errors.

Examples:

In [47]:
```
statement1 = False

if statement1:
    print ("printed if statement1 is True")

    print ("still inside the if block")
```

In [48]:
```
if statement1:
    print ("printed if statement1 is True")

print ("now outside the if block")
```

```
now outside the if block
```

Loops

In Python, loops can be programmed in a number of different ways. The most common is the `for` loop, which is used together with iterable objects, such as lists. The basic syntax is:

`for` loops:

In [49]:
```
for x in [1, 2, 3]:
    print (x)
```

```
1
2
3
```

The `for` loop iterates over the elements of the supplied list, and executes the containing block once for each element. Any kind of list can be used in the `for` loop. For example:

In [50]:
```
for x in range(4): # by default range start at 0
    print (x)
```

```
0
1
2
3
```

Note: `range(4)` does not include 4 !

In [51]:
```
for x in range(-3,3):
    print (x)
```

```
-3
-2
-1
0
1
2
```

In [52]:
```
for word in ["scientific", "computing", "with", "python"]:
    print (word)
```

```
scientific
computing
with
python
```

In [53]:
```
for x in 'Hello world!':
    print (x)
```

```
H
e
l
l
o

w
o
r
l
d
!
```

To iterate over key-value pairs of a dictionary:

In [54]:
```
for key, value in params.items():
    print (key, " = ", value)
```

```
parameter1  =  A
parameter2  =  B
parameter4  =  D
```

List comprehensions: Creating lists using `for` loops:

A convenient and compact way to initialize lists:

In [55]:
```
l1 = [x**2 for x in range(0,5)]

l1
```

Out[55]:
```
[0, 1, 4, 9, 16]
```

`while` loops:

In [56]:
```
i = 0

while i < 5:
    print (i)
    i = i + 1

print ("done")
```

```
0
1
2
3
4
done
```

Note that the `print "done"` statement is not part of the `while` loop body because of the difference in indentation.

Functions

A function in Python is defined using the keyword `def`, followed by a function name, a signature within parenthises `()`, and a colon `:`. The following code, with one additional level of indentation, is the function body.

In [57]:
```
def func0():
    print ("test")
```

In [58]:
```
func0()
```

```
test
```

Functions that returns a value use the `return` keyword:

In [59]:
```
def square(x):
    return x ** 2
```

In [60]:
```
square(4)
```

Out[60]: 16

Default argument and keyword arguments

In [61]:
```
def myfunc(x, p=2, debug=False):  # debug has a default value = False
    if debug:
        print ("evaluating myfunc for x =", x, "using exponent p =", p)
    return x**p
```

If we don't provide a value of the `debug` argument when calling the the function `myfunc` it defaults to the value provided in the function definition:

In [62]:
```
myfunc(5)
```

Out[62]: 25

In [63]:
```
myfunc(5, debug=True)
```

```
evaluating myfunc for x = 5 using exponent p = 2
```

Out[63]: 25

Classes

Classes are the key features of object-oriented programming. A class is a structure for representing an object and the operations that can be performed on the object.

In Python a class can contain *attributes* (variables) and *methods* (functions).

In python a class are defined almost like a functions, but using the `class` keyword, and the class definition usually contains a number of class method definitions (a function in a class).

Each instance method must have an argument `self` as it first argument.

Some instance method names have special meaning, for example:

- `__init__` : The name of the method that is invoked when the object is first created.
- `__str__` : A method that is invoked when a simple string representation of the class is needed, as for example when printed.

In [64]:
```
class Point:
```

```
    """
    Simple class for representing a point in a Cartesian coordinate system.
    """

    def __init__(self, x, y):
        """
        Create a new Point at x, y.
        """
        self.x = x
        self.y = y

    def translate(self, dx, dy):
        """
        Translate the point by dx and dy in the x and y direction.
        """
        self.x += dx
        self.y += dy

    def __str__(self):
        return "Point at [%f, %f]" % (self.x, self.y)
```

To create a new instance of a class:

In [65]:

```
p1 = Point(0, 0) # this will invoke the __init__ method in the Point class

print (p1)         # this will invode the __str__ method
```

```
Point at [0.000000, 0.000000]
```

To invoke an instance method in the class instance `p` :

In [66]:

```
p2 = Point(1, 1)

p1.translate(0.25, 1.5)

print (p1)
print (p2)
```

```
Point at [0.250000, 1.500000]
Point at [1.000000, 1.000000]
```

Static methods (using a decorator) and inheritance:

In [67]:

```
class MyClass(object):       # MyClass is a subclass of the superclass "object"

    @staticmethod
    def the_static_method(x):
        print (x)

MyClass.the_static_method(2) # outputs 2
```

```
2
```

Exceptions

In Python errors are managed with a special language construct called "Exceptions". When errors occur exceptions can be raised, which interrupts the normal program flow and fallback to somewhere else in the code where the closest try-except statements is defined.

To generate an exception we can use the `raise` statement, which takes an argument that must be an instance of the class `BaseExpection` or a class dervied from it.

In [23]:
```
raise Exception("description of the error")
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-23-c32f93e4dfa0> in <module>
----> 1 raise Exception("description of the error")

Exception: description of the error
```

In [24]:
```
try:
    print ("test")
    # generate an error: the variable test is not defined
    print (test)
except:
    print ("Caught an expection")
```

```
test
Caught an expection
```

To get information about the error, we can access the `Expection` class instance that describes the exception by using for example:

```
   except Exception as e:
```

In [25]:
```
try:
    print ("test")
    # generate an error: the variable test is not defined
    print (test)
except Exception as e:
    print ("Caught an expection:", e)
```

```
test
Caught an expection: name 'test' is not defined
```

CSC 180 Intelligent Systems (Fall 2022)

Dr. Haiquan Chen, Dept of Computer Scicence

California State University, Sacramento

Lab 1: Numpy, Scipy and Matplotlib - high-performance vector operations

1. Numpy

The `numpy` package (module) is used in almost all numerical computation using Python. It is a package that provide high-performance vector, matrix and higher-dimensional data structures for Python. It is implemented in C and Fortran so when calculations are vectorized (formulated with vectors and matrices), performance is very good.

To use `numpy` first import the module:

In [1]:
```
import numpy as np  # np is the most common import name for numpy
```

In the `numpy` package the terminology used for vectors, matrices and higher-dimensional data sets is *array*.

Creating `numpy` arrays (vectors or matrices)

There are a number of ways to initialize new numpy arrays, for example from

- a Python list or tuples
- using functions that are dedicated to generating numpy arrays, such as `arange`, `linspace`, etc.
- reading data from files

From lists

For example, to create new vector and matrix arrays from Python lists we can use the `numpy.array` function.

In [2]:
```
# a vector: the argument to the array function is a Python list
v = np.array([1, 2, 3, 4])
v
```

Out[2]:
```
array([1, 2, 3, 4])
```

In [3]:
```
# a matrix (or better a 2d array): the argument to the array function is a nested Pytho
M = np.array([[1, 2], [3, 4]])
M
```

Out[3]:
```
array([[1, 2],
       [3, 4]])
```

The `v` and `M` objects are both of the type `ndarray` that the `numpy` module provides.

In [4]:
```
type(v), type(M)
```

Out[4]:
```
(numpy.ndarray, numpy.ndarray)
```

The difference between the `v` and `M` arrays is only their shapes. We can get information about the shape of an array by using the `ndarray.shape` property.

In [5]:
```
v.shape
```

Out[5]:
```
(4,)
```

In [6]:
```
M.shape
```

Out[6]:
```
(2, 2)
```

The number of elements in the array is available through the `ndarray.size` property:

In [7]:
```
v.size
```

Out[7]:
```
4
```

In [8]:
```
M.size  # be careful matlab size is shape with Numpy
```

Out[8]:
```
4
```

***So far the `numpy.ndarray` looks much like a Python list.

Why not simply use Python lists for computations instead of creating a new array type?

There are several reasons:

- Numpy arrays are ***memory efficient*** and run ***much faster*** than Python lists in order to support ***big data***.

- Python lists DO NOT support ***mathematical functions*** such as matrix and dot multiplications, etc. Implementating such functions for Python lists would be very inefficient.
- Python lists are very general. They can contain any kind of object. *Numpy arrays are* ***homogeneous****. The type of the elements is determined when array is created.***

Using the `dtype` (data type) property of an `ndarray`, we can see what type each element of an array has:

In [9]:
```
M.dtype
```

Out[9]:
```
dtype('int32')
```

We get an error if we try to assign a value of the wrong type to an element in a numpy array:

In [10]:
```
M[0, 0] = "hello"
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-10-d137d88031df> in <module>
----> 1 M[0, 0] = "hello"

ValueError: invalid literal for int() with base 10: 'hello'
```

If we want, we can ***explicitly define the type of the array data*** when we create it, using the `dtype` keyword argument:

In [11]:
```
M = np.array([[1, 2], [3, 4]], dtype='float32')
M
```

Out[11]:
```
array([[1., 2.],
       [3., 4.]], dtype=float32)
```

Common type that can be used with `dtype` are: `int`, `float`, `complex`, `bool`, `object`, etc.

We can also explicitly define the bit size of the data types, for example: `int64`, `int16`, `float128`, `complex128`.

Using functions to generate arrays

For larger arrays it is inpractical to initialize the data manually, using explicit python lists. Instead we can use one of the many functions in `numpy` that generates arrays of different forms. Some of the more common are:

arange

In [12]:
```
# create a range

x = np.arange(0, 10, 1) # arguments: start, stop, step
x
```

Out[12]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [13]:
```
x = np.arange(-1, 1, 0.1)
x
```

Out[13]:
```
array([-1.00000000e+00, -9.00000000e-01, -8.00000000e-01, -7.00000000e-01,
       -6.00000000e-01, -5.00000000e-01, -4.00000000e-01, -3.00000000e-01,
       -2.00000000e-01, -1.00000000e-01, -2.22044605e-16,  1.00000000e-01,
        2.00000000e-01,  3.00000000e-01,  4.00000000e-01,  5.00000000e-01,
        6.00000000e-01,  7.00000000e-01,  8.00000000e-01,  9.00000000e-01])
```

linspace and logspace

In [14]:
```
# using linspace, both end points ARE included
np.linspace(0, 10, 25)
```

Out[14]:
```
array([ 0.        ,  0.41666667,  0.83333333,  1.25      ,  1.66666667,
        2.08333333,  2.5       ,  2.91666667,  3.33333333,  3.75      ,
        4.16666667,  4.58333333,  5.        ,  5.41666667,  5.83333333,
        6.25      ,  6.66666667,  7.08333333,  7.5       ,  7.91666667,
        8.33333333,  8.75      ,  9.16666667,  9.58333333, 10.        ])
```

In [16]:
```
np.logspace(0, 10, 10, base=10)
```

Out[16]:
```
array([1.00000000e+00, 1.29154967e+01, 1.66810054e+02, 2.15443469e+03,
       2.78255940e+04, 3.59381366e+05, 4.64158883e+06, 5.99484250e+07,
       7.74263683e+08, 1.00000000e+10])
```

Manipulating arrays

Indexing

We can index elements in an array using the square bracket and indices:

In [17]:
```
v = np.array([1, 2, 3, 4])

# v is a vector, and has only one dimension, taking one index
v[0]
```

Out[17]:
```
1
```

In [27]:
```
M = np.array([[1, 2], [3, 4]])

# M is a matrix, or a 2 dimensional array, taking two indices
M[1, 1]
```

Out[27]:
```
4
```

In [19]:
```
M
```

Out[19]:
```
array([[1, 2],
       [3, 4]])
```

In [20]:
```
M[1, :] # row 1
```

Out[20]:
```
array([3, 4])
```

In [20]:
```
M[:, 1] # column 1
```

Out[20]:
```
array([2, 4])
```

We can assign new values to elements in an array using indexing:

In [21]:
```
M[0, 0] = 10
M
```

Out[21]:
```
array([[10,  2],
       [ 3,  4]])
```

In [22]:
```
# also works for rows and columns
M[0, :] = 0
M[:, 1] = -1
```

In [23]:
```
M
```

Out[23]:
```
array([[ 0, -1],
       [ 3, -1]])
```

Index slicing

Index slicing is the technical name for the syntax `M[start:end:step]` to extract part of an array:

In [23]:
```
A = np.array([1,2,3,4,5])
A
```

Out[23]:
```
array([1, 2, 3, 4, 5])
```

In [24]:
```
A[1:3]
```

Out[24]:
```
array([2, 3])
```

Array slices are *mutable*

If they are assigned a new value the original array from which the slice was extracted is modified:

In [25]:
```
A[1:3] = [-2, -3]

A
```

Out[25]:
```
array([ 1, -2, -3,  4,  5])
```

We can omit any of the three parameters in `M[start:end:step]`:

In [26]:
```
A[::] # lower, upper, step all take the default values
```

Out[26]:
```
array([ 1, -2, -3,  4,  5])
```

In [27]:
```
A[::2] # step is 2, lower and upper defaults to the beginning and end of the array
```

Out[27]:
```
array([ 1, -3,  5])
```

In [28]:
```
A[:3] # first three elements
```

Out[28]:
```
array([ 1, -2, -3])
```

In [29]:
```
A[3:] # elements from index 3
```

Out[29]:
```
array([4, 5])
```

Negative indices counts from the end of the array (positive index from the begining):

In [31]:
```
A = np.array([1, 2, 3, 4, 5])
```

In [32]:
```
A[-1] # the last element in the array
```

Out[32]:
```
5
```

In [33]:
```
A[-3:] # the last three elements
```

Out[33]:
```
array([3, 4, 5])
```

Index slicing works exactly the same way for multidimensional arrays:

You may also use an array or list as an index:

In [2]:
```
A = np.array([[n+m*10 for n in range(5)] for m in range(5)])

A   # a matrix
```

Out[2]:
```
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])
```

In [34]:
```
row_indices = [1, 2, 3]
A[row_indices,:]
```

Out[34]:
```
array([[10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34]])
```

In [35]:
```
col_indices = [1, 2, -1] # remember, index -1 means the last element
```

```
A[:,col_indices]
```

Out[35]:
```
array([[ 1,  2,  4],
       [11, 12, 14],
       [21, 22, 24],
       [31, 32, 34],
       [41, 42, 44]])
```

In [36]:
```
# a block from the original array
A[1:4, 1:4]
```

Out[36]:
```
array([[11, 12, 13],
       [21, 22, 23],
       [31, 32, 33]])
```

Array-oriented programming

Vectorizing code is the key to writing code when handling big data. That means that as much as possible, a program should be written in terms of matrix (vector) operations.

Scalar-array operations

We can use the usual arithmetic operators to multiply, add, subtract, and divide arrays with scalar numbers.

In [3]:
```
v1 = np.arange(0, 5)
v1
```

Out[3]:
```
array([0, 1, 2, 3, 4])
```

In [4]:
```
v1 * 2
```

Out[4]:
```
array([0, 2, 4, 6, 8])
```

In [5]:
```
v1 + 2
```

Out[5]:
```
array([2, 3, 4, 5, 6])
```

Element-wise array-array operations

When we add, subtract, multiply and divide arrays with each other, the default behaviour is **element-wise** operations:

In [6]:
```
A
```

Out[6]:
```
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])
```

In [7]:
```
A * A # element-wise multiplication
```

Out[7]:
```
array([[   0,    1,    4,    9,   16],
       [ 100,  121,  144,  169,  196],
       [ 400,  441,  484,  529,  576],
       [ 900,  961, 1024, 1089, 1156],
       [1600, 1681, 1764, 1849, 1936]])
```

Matrix algebra

What about matrix mutiplication? We can use the `dot` function

In [8]:
```
np.dot(A, A)
```

Out[8]:
```
array([[ 300,  310,  320,  330,  340],
       [1300, 1360, 1420, 1480, 1540],
       [2300, 2410, 2520, 2630, 2740],
       [3300, 3460, 3620, 3780, 3940],
       [4300, 4510, 4720, 4930, 5140]])
```

Basic data analysis using numpy

Often it is useful to store datasets in Numpy arrays (matrix). Numpy provides a number of functions to calculate statistics of datasets in arrays.

In [9]:
```
A
```

Out[9]:
```
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])
```

In [10]:
```
np.shape(A)
```

Out[10]:
```
(5, 5)
```

mean

In [11]:
```
np.mean(A[:, 3])
```

Out[11]:
```
23.0
```

standard deviations and variance, min and max

In [12]:
```
np.std(A[:, 3]), np.var(A[:, 3])
```

Out[12]:
```
(14.142135623730951, 200.0)
```

In [13]:
```
# lowest value
A[:, 3].min()
```

Out[13]:
```
3
```

In [14]:
```
# highest value
A[:, 3].max()
```

Out[14]:
```
43
```

sum, prod

In [15]:
```
# sum up all elements
np.sum(A[:, 3])
```

Out[15]:
```
115
```

In [16]:
```
# product of all elements
np.prod(A[:, 3])
```

Out[16]:
```
1272843
```

Perform funcitons to entire matrix?

It is sometimes useful to apply the calculation to the entire array, and sometimes only on a row or column basis. Using the `axis` argument we can specify how these functions should behave:

In [17]:
```
from numpy import random

m = random.rand(3, 3)
m
```

Out[17]:
```
array([[0.45053935, 0.97441981, 0.59813365],
       [0.06822442, 0.29576215, 0.07293118],
       [0.65040022, 0.21297668, 0.74898193]])
```

In [18]:
```
# global max
m.max()
```

Out[18]:
```
0.974419813504629
```

In [19]:
```
# max in each column
m.max(axis=0)
```

Out[19]:
```
array([0.65040022, 0.97441981, 0.74898193])
```

In [20]:
```
# max in each row
m.max(axis=1)
```

Out[20]:
```
array([0.97441981, 0.29576215, 0.74898193])
```

Many other functions and methods in the `array` and `matrix` classes accept the same (optional)

`axis` keyword argument.

Reshaping arrays

The shape of an Numpy array can be modified without copying the underlaying data, which makes it a fast operation even for large arrays.

In [21]:
```
A
```

Out[21]:
```
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])
```

In [22]:
```
n, m = A.shape
n, m     # equivalent to print(n, m)
```

Out[22]:
```
(5, 5)
```

In [23]:
```
B = A.reshape((1, n*m))
B
```

Out[23]:
```
array([[ 0,  1,  2,  3,  4, 10, 11, 12, 13, 14, 20, 21, 22, 23, 24, 30,
        31, 32, 33, 34, 40, 41, 42, 43, 44]])
```

We can also use the function `flatten` to make a higher-dimensional array into a vector. But this function create a seperate copy of the underlying data.

In [24]:
```
B = A.flatten()  # because B's data is a seperate copy of A's,
B
```

Out[24]:
```
array([ 0,  1,  2,  3,  4, 10, 11, 12, 13, 14, 20, 21, 22, 23, 24, 30, 31,
       32, 33, 34, 40, 41, 42, 43, 44])
```

Type casting

Since Numpy arrays are *statically typed*, the type of an array does not change once created. But we can explicitly cast an array of some type to another using the `astype` functions. This always create a new array of new type:

In [28]:
```
M
```

Out[28]:
```
array([[1, 2],
       [3, 4]])
```

In [29]:
```
M.dtype
```

Out[29]:
```
dtype('int32')
```

In [30]:

```
M2 = M.astype('float64')

M2
```

Out[30]:
```
array([[1., 2.],
       [3., 4.]])
```

In [31]:
```
M2.dtype
```

Out[31]:
```
dtype('float64')
```

In [32]:
```
M3 = M.astype(bool)

M3
```

Out[32]:
```
array([[ True,  True],
       [ True,  True]])
```

2. Scipy: a library for scientific computing

It provides many user-friendly and efficient numerical routines such as stats, numerical integration and optimization.

In [33]:
```
import numpy as np
import scipy.spatial.distance as sp_dist


x = np.random.randint(0, 2, size = 20)      # Return random integers from low (inclusiv
y = np.random.randint(0, 2, size = 20)

print (x)
print (y)
print (sp_dist.cosine(x,y))
print (sp_dist.euclidean(x,y))
print (sp_dist.jaccard(x,y))
print (sp_dist.hamming(x,y))
```

```
[1 1 1 0 0 1 1 1 0 0 0 1 0 1 1 0 1 0 0 1]
[0 0 0 0 0 0 1 1 1 1 0 1 0 0 1 0 0 0 0 0]
0.507634036082669
3.0
0.6923076923076923
0.45
```

In [34]:
```
from scipy.stats import linregress

a = [15, 12, 8, 8, 7, 7, 7, 6, 5, 3]
b = [10, 25, 17, 11, 13, 17, 20, 13, 9, 15]

linregress(a, b)
```

Out[34]:
```
LinregressResult(slope=0.20833333333333331, intercept=13.375, rvalue=0.1449981545806851
8, pvalue=0.689401448116695, stderr=0.5026170462708364)
```

3. Matplotlib

(1) Use a magic command to enable figures in notebook!!!

%matplotlib inline

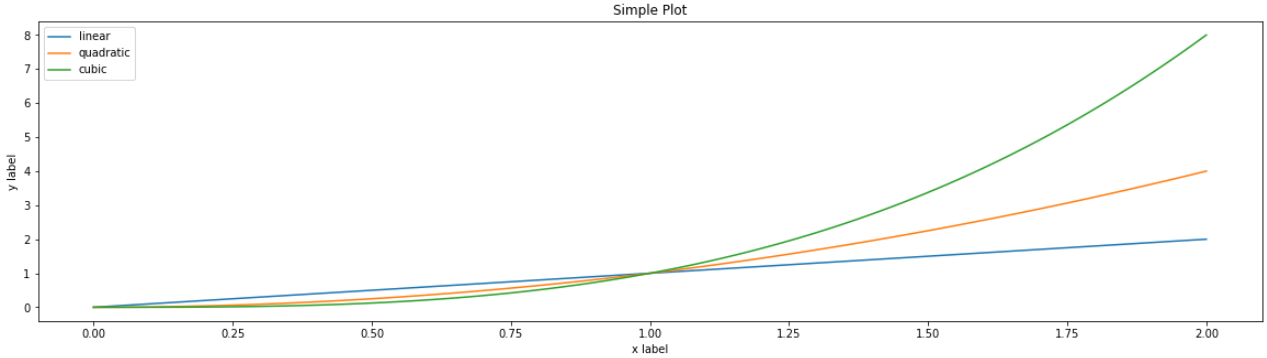
(2) Import

import matplotlib.pyplot as plt

In [35]:

```
# A magic command to enable matplotlib figures in notebook!!!
%matplotlib inline

# import
import matplotlib.pyplot as plt
import numpy as np


# create an empty figure

fig = plt.figure()  # an empty figure

plt.figure(figsize=(20,5))  # optional

x = np.linspace(0, 2, 100)

plt.plot(x, x, label='linear')
plt.plot(x, x**2, label='quadratic')
plt.plot(x, x**3, label='cubic')

plt.xlabel('x label')
plt.ylabel('y label')

plt.title("Simple Plot")

plt.legend()

plt.show()
```

```
<Figure size 432x288 with 0 Axes>
```



Generate random data

In [36]:

```
from numpy import random
```

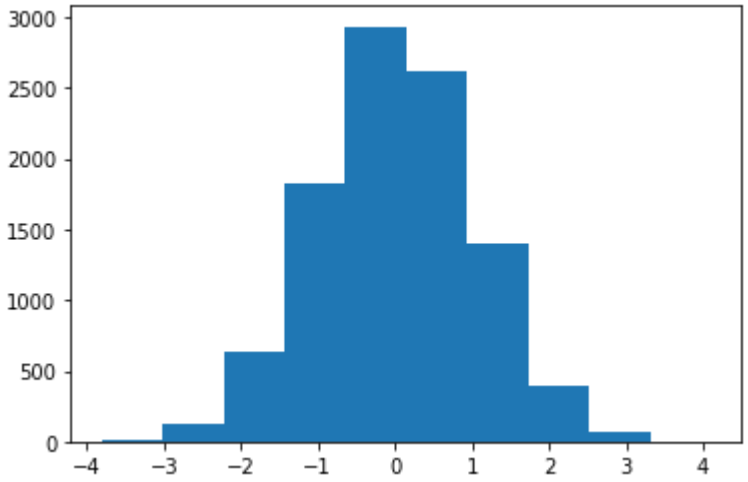
In [37]:
```
# uniform random numbers in [0,1]
random.rand(5,5)
```

Out[37]:
```
array([[0.67373676, 0.02403633, 0.5230291 , 0.67004306, 0.03264233],
       [0.82936216, 0.89950513, 0.64177194, 0.48986332, 0.63748397],
       [0.1685659 , 0.46961103, 0.24442622, 0.30736207, 0.74108466],
       [0.46596404, 0.94530707, 0.75338508, 0.76889138, 0.41892492],
       [0.72600649, 0.1809466 , 0.90475738, 0.55931287, 0.26711076]])
```

In [38]:
```
# standard normal distributed random numbers
random.randn(5,5)
```

Out[38]:
```
array([[ 0.29216707, -1.1448224 ,  1.81378403,  0.45844933, -0.02777439],
       [ 0.2530375 ,  0.00305307,  0.86589231,  0.05498005, -1.52839608],
       [ 0.90962229,  0.60770911, -0.4685774 ,  0.16719353, -0.93821694],
       [-1.31919377,  1.30187711, -2.15219761, -0.09135717, -1.05306543],
       [-0.10156332,  0.34248402,  0.22720028, -1.50388356,  1.65948624]])
```

In [39]:
```
x = random.randn(10000)
plt.hist(x)
plt.show()
```



In [40]:
```
np.random.randint(0, 2, size=10)   # Return random integers from low (inclusive) to hig
```

Out[40]:
```
array([0, 1, 1, 0, 0, 0, 0, 1, 0, 1])
```

In [41]:
```
np.random.randint(0, 5, size=(2, 4))   # Return random integers from low (inclusive) to
```

Out[41]:
```
array([[1, 1, 3, 2],
       [1, 2, 1, 2]])
```

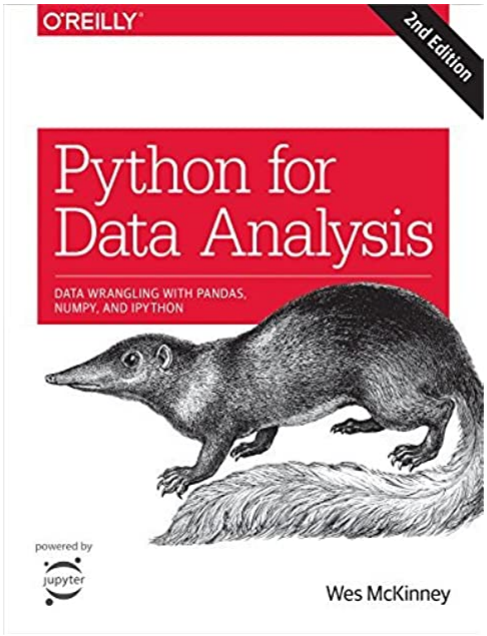
In [ ]:

CSC 180 Intelligent Systems (Fall 2022)

Dr. Haiquan Chen, Dept of Computer Scicence

California State University, Sacramento

Lab 2: Data Preprocessing using Pandas



1. What is pandas?

Pandas is an open source library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language. Pandas is built on top of Numpy.

- pandas main page
- How to install pandas? Anaconda distribution of Python (includes pandas)

Pandas gives you two data structures:

- Series: One-dimensional numpy array with a labeled index.
- Dataframe: A collection of multiple Series with the same labeled index. (Two-dimensional)

Online official documentation should be your best friend!

- pandas 10 minutes

conventional way to import pandas

import pandas as pd

2. How do I read a tabular data file into pandas?

In [1]:
```
import pandas as pd

orders = pd.read_csv('data/chipotle.tsv', sep='\t')
```

In [2]:
```
# examine the first 5 rows
orders.head()
```

Out[2]:

	order_id	quantity	item_name	choice_description	item_price
0	1	1	Chips and Fresh Tomato Salsa	NaN	$2.39
1	1	1	Izze	[Clementine]	$3.39
2	1	1	Nantucket Nectar	[Apple]	$3.39
3	1	1	Chips and Tomatillo-Green Chili Salsa	NaN	$2.39
4	2	2	Chicken Bowl	[Tomatillo-Red Chili Salsa (Hot), [Black Beans...	$16.98

In [3]:
```
# read a dataset of movie reviewers (modifying the default parameter values for read_ta
user_cols = ['user_id', 'age', 'gender', 'occupation', 'zip_code']
users = pd.read_csv('data/u.user', sep='|', header=None, names=user_cols)
```

In [4]:
```
# examine the first 5 rows
users.head()
```

Out[4]:

	user_id	age	gender	occupation	zip_code
0	1	24	M	technician	85711
1	2	53	F	other	94043
2	3	23	M	writer	32067
3	4	24	M	technician	43537
4	5	33	F	other	15213

In [5]:
```
users
```

Out[5]:

	user_id	age	gender	occupation	zip_code
0	1	24	M	technician	85711
1	2	53	F	other	94043
2	3	23	M	writer	32067
3	4	24	M	technician	43537
4	5	33	F	other	15213

	user_id	age	gender	occupation	zip_code
...	...	...	...	...	...
938	939	26	F	student	33319
939	940	32	M	administrator	02215
940	941	20	M	student	97229
941	942	48	F	librarian	78209
942	943	22	M	student	77841

943 rows × 5 columns

In [6]:
```
type(users)
```

Out[6]:
```
pandas.core.frame.DataFrame
```

3. How do I select a pandas Series from a DataFrame?

In [7]:
```
# read_csv is equivalent to read_table, except it assumes a comma separator
ufo = pd.read_csv('data/ufo.csv')
```

In [8]:
```
# examine the first 5 rows
ufo.head()
```

Out[8]:

	City	Colors Reported	Shape Reported	State	Time
0	Ithaca	NaN	TRIANGLE	NY	6/1/1930 22:00
1	Willingboro	NaN	OTHER	NJ	6/30/1930 20:00
2	Holyoke	NaN	OVAL	CO	2/15/1931 14:00
3	Abilene	NaN	DISK	KS	6/1/1931 13:00
4	New York Worlds Fair	NaN	LIGHT	NY	4/18/1933 19:00

In [9]:
```
# select the 'City' Series using bracket notation
ufo['City']

# or equivalently, use dot notation
ufo.City
```

Out[9]:
```
0                    Ithaca
1                Willingboro
2                    Holyoke
3                    Abilene
4        New York Worlds Fair
                 ...
18236             Grant Park
18237            Spirit Lake
18238            Eagle River
```

```
18239             Eagle River
18240                    Ybor
Name: City, Length: 18241, dtype: object
```

In [10]:

```
# create a new 'Location' Series (must use bracket notation to define the Series name)
ufo['Location'] = ufo.City + ', ' + ufo.State
ufo
```

Out[10]:

	City	Colors Reported	Shape Reported	State	Time	Location
0	Ithaca	NaN	TRIANGLE	NY	6/1/1930 22:00	Ithaca, NY
1	Willingboro	NaN	OTHER	NJ	6/30/1930 20:00	Willingboro, NJ
2	Holyoke	NaN	OVAL	CO	2/15/1931 14:00	Holyoke, CO
3	Abilene	NaN	DISK	KS	6/1/1931 13:00	Abilene, KS
4	New York Worlds Fair	NaN	LIGHT	NY	4/18/1933 19:00	New York Worlds Fair, NY
...	...	...	...	...	...	...
18236	Grant Park	NaN	TRIANGLE	IL	12/31/2000 23:00	Grant Park, IL
18237	Spirit Lake	NaN	DISK	IA	12/31/2000 23:00	Spirit Lake, IA
18238	Eagle River	NaN	NaN	WI	12/31/2000 23:45	Eagle River, WI
18239	Eagle River	RED	LIGHT	WI	12/31/2000 23:45	Eagle River, WI
18240	Ybor	NaN	OVAL	FL	12/31/2000 23:59	Ybor, FL

18241 rows × 6 columns

4. how to handle Null values, do basic stats and sort dataframes

It is possible to sort.

In [11]:

```
import numpy as np
import pandas as pd

pd.set_option('display.max_colwidth', -1)


df = pd.read_csv("./data/auto-mpg.csv")
df[0:5]
```

Out[11]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin	name

	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin	name
0	18.0	8	307.0	130	3504	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165	3693	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150	3436	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150	3433	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140	3449	10.5	70	1	ford torino

In [12]:
```
df.dtypes
```

Out[12]:
```
mpg             float64
cylinders         int64
displacement    float64
horsepower       object
weight            int64
acceleration    float64
year              int64
origin            int64
name             object
dtype: object
```

In [13]:
```
# na_values:   special strings to recognize as NA/NaN.
df = pd.read_csv("./data/auto-mpg.csv", na_values=['NA','?'])

# return rows with one or more nulls
df[df.isnull().any(axis=1)]
```

Out[13]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin	name
32	25.0	4	98.0	NaN	2046	19.0	71	1	ford pinto
126	21.0	6	200.0	NaN	2875	17.0	74	1	ford maverick
330	40.9	4	85.0	NaN	1835	17.3	80	2	renault lecar deluxe
336	23.6	4	140.0	NaN	2905	14.3	80	1	ford mustang cobra
354	34.5	4	100.0	NaN	2320	15.8	81	2	renault 18i
374	23.0	4	151.0	NaN	3035	20.5	82	1	amc concord dl

In [14]:
```
df['horsepower'].mean(), df['horsepower'].var(), df['horsepower'].std()
```

Out[14]:
```
(104.46938775510205, 1481.5693929745862, 38.49115993282855)
```

In [15]:
```
df = df.sort_values(by='name',ascending=True)
df
```

Out[15]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin	name

	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin	name
96	13.0	8	360.0	175.0	3821	11.0	73	1	amc ambassador brougham
9	15.0	8	390.0	190.0	3850	8.5	70	1	amc ambassador dpl
66	17.0	8	304.0	150.0	3672	11.5	72	1	amc ambassador sst
315	24.3	4	151.0	90.0	3003	20.1	80	1	amc concord
257	19.4	6	232.0	90.0	3210	17.2	78	1	amc concord
...	...	...	...	...	...	...	...	...	...
394	44.0	4	97.0	52.0	2130	24.6	82	2	vw pickup
309	41.5	4	98.0	76.0	2144	14.7	80	2	vw rabbit
197	29.0	4	90.0	70.0	1937	14.2	76	2	vw rabbit
325	44.3	4	90.0	48.0	2085	21.7	80	2	vw rabbit c (diesel)
293	31.9	4	89.0	71.0	1925	14.0	79	2	vw rabbit custom

398 rows × 9 columns

In [16]:
```
print("The first car is: {}".format(df['name'].iloc[0]))
```

```
The first car is: amc ambassador brougham
```

In [17]:
```
print("The first car is: {}".format(df['name'].loc[197]))

#iloc gets rows (or columns) at particular positions in the index (so it only takes int
#loc gets rows (or columns) with particular labels from the index.
```

```
The first car is: vw rabbit
```

5. Save a Dataframe

The following code performs a shuffle and then saves a new copy.

In [18]:
```
import os
import pandas as pd
import numpy as np

path = "./data/"

filename_read = os.path.join(path,"auto-mpg.csv")
filename_write = os.path.join(path,"auto-mpg-shuffle.csv")
df = pd.read_csv(filename_read,na_values=['NA','?'])
df = df.reindex(np.random.permutation(df.index))
```

```
df.to_csv(filename_write,index=False)    # Specify index = false to not write row number
print("Done")
```

```
Done
```

6. Drop Fields

Some fields are of no value to the neural network and can be dropped. The following code removes the name column from the MPG dataset.

In [19]:

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

path = "./data/"

filename_read = os.path.join(path,"auto-mpg.csv")
df = pd.read_csv(filename_read,na_values=['NA','?'])

print("Before drop: {}".format(df.columns))
df.drop('name', axis=1, inplace=True)
print("After drop: {}".format(df.columns))
df[0:5]
```

```
Before drop: Index(['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
       'acceleration', 'year', 'origin', 'name'],
      dtype='object')
After drop: Index(['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
       'acceleration', 'year', 'origin'],
      dtype='object')
```

Out[19]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin
0	18.0	8	307.0	130.0	3504	12.0	70	1
1	15.0	8	350.0	165.0	3693	11.5	70	1
2	18.0	8	318.0	150.0	3436	11.0	70	1
3	16.0	8	304.0	150.0	3433	12.0	70	1
4	17.0	8	302.0	140.0	3449	10.5	70	1

7. Calculated Fields

It is possible to add new fields to the dataframe that are calculated from the other fields. We can create a new column that gives the weight in kilograms. The equation to calculate a metric weight, given a weight in pounds is:

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

This can be used with the following Python code:

In [21]:

```
import os
import pandas as pd
```

```
import numpy as np

path = "./data/"

filename_read = os.path.join(path,"auto-mpg.csv")
df = pd.read_csv(filename_read,na_values=['NA','?'])
df.insert(1,'weight_kg',(df['weight']*0.45359237).astype(int))

#df['weight_kg']= df['weight']*0.45359237).astype(int))


df
```

Out[21]:

	mpg	weight_kg	cylinders	displacement	horsepower	weight	acceleration	year	origin	nam
0	18.0	1589	8	307.0	130.0	3504	12.0	70	1	chevrole chevell malib
1	15.0	1675	8	350.0	165.0	3693	11.5	70	1	buic skylar 32
2	18.0	1558	8	318.0	150.0	3436	11.0	70	1	plymout satellit
3	16.0	1557	8	304.0	150.0	3433	12.0	70	1	am rebel s
4	17.0	1564	8	302.0	140.0	3449	10.5	70	1	for torin
...	...	...	...	...	...	...	...	...	...	
393	27.0	1265	4	140.0	86.0	2790	15.6	82	1	for mustan
394	44.0	966	4	97.0	52.0	2130	24.6	82	2	v picku
395	32.0	1040	4	135.0	84.0	2295	11.6	82	1	dodg rampag
396	28.0	1190	4	120.0	79.0	2625	18.6	82	1	for range
397	31.0	1233	4	119.0	82.0	2720	19.4	82	1	chevy s 1

398 rows × 10 columns

8. Feature Normalization

A normalization allows numbers to be put in a standard form so that two values can easily be compared. One very common machine learning normalization is the Z-Score:

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

To calculate the Z-Score you need to also calculate the mean(μ) and the standard deviation (σ). The mean is calculated as follows:

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

The standard deviation is calculated as follows:

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

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 above/below -3/3 are very rare, these are outliers.

In [22]:
```
import os
import pandas as pd
import numpy as np
from scipy.stats import zscore

path = "./data/"

filename_read = os.path.join(path,"auto-mpg.csv")
df = pd.read_csv(filename_read,na_values=['NA','?'])
df['mpg'] = zscore(df['mpg'])
df
```

Out[22]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin	name
0	-0.706439	8	307.0	130.0	3504	12.0	70	1	chevrolet chevelle malibu
1	-1.090751	8	350.0	165.0	3693	11.5	70	1	buick skylark 320
2	-0.706439	8	318.0	150.0	3436	11.0	70	1	plymouth satellite
3	-0.962647	8	304.0	150.0	3433	12.0	70	1	amc rebel sst
4	-0.834543	8	302.0	140.0	3449	10.5	70	1	ford torino
...	...	...	...	...	...	...	...	...	...
393	0.446497	4	140.0	86.0	2790	15.6	82	1	ford mustang gl
394	2.624265	4	97.0	52.0	2130	24.6	82	2	vw pickup
395	1.087017	4	135.0	84.0	2295	11.6	82	1	dodge rampage
396	0.574601	4	120.0	79.0	2625	18.6	82	1	ford ranger
397	0.958913	4	119.0	82.0	2720	19.4	82	1	chevy s-10

398 rows × 9 columns

9. Missing Values

You can also simply drop any rows with any NA values. Another common practice is to replace missing values with the median value for that column. The following code replaces any NA values in horsepower with the median:

In [23]:
```
import os
import pandas as pd
import numpy as np
from scipy.stats import zscore

path = "./data/"

filename_read = os.path.join(path,"auto-mpg.csv")
df = pd.read_csv(filename_read,na_values=['NA','?'])
med = df['horsepower'].median()
df['horsepower'] = df['horsepower'].fillna(med)

# df = df.dropna() # you can also simply drop NA values
```

10. Concatenate Rows and Columns

Rows and columns can be concatenated together to form new data frames.

In [24]:
```
# Create a new dataframe from name and horsepower

import os
import pandas as pd
import numpy as np
from scipy.stats import zscore

path = "./data/"

filename_read = os.path.join(path,"auto-mpg.csv")
df = pd.read_csv(filename_read,na_values=['NA','?'])
col_horsepower = df['horsepower']
col_name = df['name']
result = pd.concat([col_name,col_horsepower],axis=1)
result
```

Out[24]:

	name	horsepower
0	chevrolet chevelle malibu	130.0
1	buick skylark 320	165.0
2	plymouth satellite	150.0
3	amc rebel sst	150.0
4	ford torino	140.0
...	...	...
393	ford mustang gl	86.0

	name	horsepower
394	vw pickup	52.0
395	dodge rampage	84.0
396	ford ranger	79.0
397	chevy s-10	82.0

398 rows × 2 columns

11. One-hot encoding in pandas (converting a categorical feature)

In [26]:
```
# read the training dataset from Kaggle's Titanic competition
train = pd.read_csv('data/titanic_train.csv')
train.head()
```

Out[26]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Eml
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Thayer)	female	38.0	1	0	PC 17599	71.2833	C85	
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	

In [27]:
```
# pass the DataFrame to 'get_dummies' and specify which columns to dummy (it drops the

pd.get_dummies(train, columns=['Sex', 'Embarked']).head()
```

Out[27]:

	PassengerId	Survived	Pclass	Name	Age	SibSp	Parch	Ticket	Fare	Cabin	Sex_female

	PassengerId	Survived	Pclass	Name	Age	SibSp	Parch	Ticket	Fare	Cabin	Sex_female
0	1	0	3	Braund, Mr. Owen Harris	22.0	1	0	A/5 21171	7.2500	NaN	0
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Thayer)	38.0	1	0	PC 17599	71.2833	C85	1
2	3	1	3	Heikkinen, Miss. Laina	26.0	0	0	STON/O2. 3101282	7.9250	NaN	1
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	35.0	1	0	113803	53.1000	C123	1
4	5	0	3	Allen, Mr. William Henry	35.0	0	0	373450	8.0500	NaN	0

Super super cool, right?

Let's define some helpful functions for Tensorflow (little gems)

These functions will be used to preprocess the data. They help you build the feature vectors in the format that TensorFlow expects from raw data.

(1) Encoding data:

- **encode_text_dummy** - Encode text fields as numeric, such as the iris species as a single field for each class. Three classes would become "0,0,1" "0,1,0" and "1,0,0". ***Encode non-target features this way***. ***(one hot encoding)***
- **encode_text_index** - Encode text fields to numeric, such as the iris species as a single numeric field as "0" "1" and "2". ***Encode the target field for a classification this way***. ***(label encoding)***

(2) Normalizing data:

- **encode_numeric_zscore** - Encode numeric values as a z-score. Neural networks deal well with "normalized" fields only.
- **encode_numeric_range** - Encode a column to a range between the given normalized_low and normalized_high.

(3) Dealing with missing data:

- **missing_median** - Fill all missing values with the median value.

(4) Removing outliers:

- **remove_outliers** - Remove outliers in a certain column with a value beyond X times SD

(5) Creating the ***feature vector*** and ***target vector*** that ***Tensorflow needs***. ***Notice that Tensorflow only takes Numpy array as input***

- **to_xy** - If all fields are numeric, this function can ***create X (feature matrix) and Y (response vector)*** in a format TensorFlow needs.

(6) Other utility functions:

- **hms_string** - Print out an elapsed time string.
- **chart_regression** - Display a chart to show how well a regression performs.

In [1]:

```
from collections.abc import Sequence
from sklearn import preprocessing
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import shutil
import os


# Encode text values to dummy variables(i.e. [1,0,0],[0,1,0],[0,0,1] for red,green,blue
def encode_text_dummy(df, name):
    dummies = pd.get_dummies(df[name])
    for x in dummies.columns:
        dummy_name = "{}-{}".format(name, x)
        df[dummy_name] = dummies[x]
    df.drop(name, axis=1, inplace=True)


# Encode text values to indexes(i.e. [1],[2],[3] for red,green,blue).
def encode_text_index(df, name):
    le = preprocessing.LabelEncoder()
    df[name] = le.fit_transform(df[name])
    return le.classes_


# Encode a numeric column as zscores
def encode_numeric_zscore(df, name, mean=None, sd=None):
    if mean is None:
        mean = df[name].mean()

    if sd is None:
        sd = df[name].std()

    df[name] = (df[name] - mean) / sd


# Convert all missing values in the specified column to the median
```

```
def missing_median(df, name):
    med = df[name].median()
    df[name] = df[name].fillna(med)


# Convert all missing values in the specified column to the default
def missing_default(df, name, default_value):
    df[name] = df[name].fillna(default_value)


# Convert a Pandas dataframe to the x,y inputs that TensorFlow needs
def to_xy(df, target):
    result = []
    for x in df.columns:
        if x != target:
            result.append(x)
    # find out the type of the target column.
    target_type = df[target].dtypes
    target_type = target_type[0] if isinstance(target_type, Sequence) else target_type
    # Encode to int for classification, float otherwise. TensorFlow likes 32 bits.
    if target_type in (np.int64, np.int32):
        # Classification
        dummies = pd.get_dummies(df[target])
        return df[result].values.astype(np.float32), dummies.values.astype(np.float32)
    else:
        # Regression
        return df[result].values.astype(np.float32), df[target].values.astype(np.float3

# 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)


# 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)
    a = plt.plot(t['y'].tolist(),label='expected')
    b = plt.plot(t['pred'].tolist(),label='prediction')
    plt.ylabel('output')
    plt.legend()
    plt.show()

# 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)


# Encode a column to a range between normalized_low and normalized_high.
def encode_numeric_range(df, name, normalized_low=-1, normalized_high=1,
                         data_low=None, data_high=None):
    if data_low is None:
        data_low = min(df[name])
        data_high = max(df[name])
```

```
    df[name] = ((df[name] - data_low) / (data_high - data_low)) * (normalized_high - no
```

Examples of label encoding, one hot encoding, and creating X/Y for TensorFlow

In [2]:
```
df=pd.read_csv("data/iris.csv", na_values=['NA','?'])
df
```

Out[2]:

	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
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

150 rows × 5 columns

TensorFlow only works with numpy arrays. So next we call to_xy() to convert data from dataframe to numpy arrays. Note: Apply lable encoding to the output feature before you call to_xy()!

In [3]:
```
encode_text_index(df,"species")    # encoding first before you call to_xy()

df
```

Out[3]:

	sepal_l	sepal_w	petal_l	petal_w	species
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	2

	sepal_l	sepal_w	petal_l	petal_w	species
146	6.3	2.5	5.0	1.9	2
147	6.5	3.0	5.2	2.0	2
148	6.2	3.4	5.4	2.3	2
149	5.9	3.0	5.1	1.8	2

150 rows × 5 columns

In [4]:
```
x,y = to_xy(df,"species")
```

In [5]:
```
x
```

Out[5]:
```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       [4.6, 3.1, 1.5, 0.2],
       [5. , 3.6, 1.4, 0.2],
       [5.4, 3.9, 1.7, 0.4],
       [4.6, 3.4, 1.4, 0.3],
       [5. , 3.4, 1.5, 0.2],
       [4.4, 2.9, 1.4, 0.2],
       [4.9, 3.1, 1.5, 0.1],
       [5.4, 3.7, 1.5, 0.2],
       [4.8, 3.4, 1.6, 0.2],
       [4.8, 3. , 1.4, 0.1],
       [4.3, 3. , 1.1, 0.1],
       [5.8, 4. , 1.2, 0.2],
       [5.7, 4.4, 1.5, 0.4],
       [5.4, 3.9, 1.3, 0.4],
       [5.1, 3.5, 1.4, 0.3],
       [5.7, 3.8, 1.7, 0.3],
       [5.1, 3.8, 1.5, 0.3],
       [5.4, 3.4, 1.7, 0.2],
       [5.1, 3.7, 1.5, 0.4],
       [4.6, 3.6, 1. , 0.2],
       [5.1, 3.3, 1.7, 0.5],
       [4.8, 3.4, 1.9, 0.2],
       [5. , 3. , 1.6, 0.2],
       [5. , 3.4, 1.6, 0.4],
       [5.2, 3.5, 1.5, 0.2],
       [5.2, 3.4, 1.4, 0.2],
       [4.7, 3.2, 1.6, 0.2],
       [4.8, 3.1, 1.6, 0.2],
       [5.4, 3.4, 1.5, 0.4],
       [5.2, 4.1, 1.5, 0.1],
       [5.5, 4.2, 1.4, 0.2],
       [4.9, 3.1, 1.5, 0.2],
       [5. , 3.2, 1.2, 0.2],
       [5.5, 3.5, 1.3, 0.2],
       [4.9, 3.6, 1.4, 0.1],
       [4.4, 3. , 1.3, 0.2],
       [5.1, 3.4, 1.5, 0.2],
       [5. , 3.5, 1.3, 0.3],
       [4.5, 2.3, 1.3, 0.3],
       [4.4, 3.2, 1.3, 0.2],
       [5. , 3.5, 1.6, 0.6],
```

```
[5.1, 3.8, 1.9, 0.4],
[4.8, 3. , 1.4, 0.3],
[5.1, 3.8, 1.6, 0.2],
[4.6, 3.2, 1.4, 0.2],
[5.3, 3.7, 1.5, 0.2],
[5. , 3.3, 1.4, 0.2],
[7. , 3.2, 4.7, 1.4],
[6.4, 3.2, 4.5, 1.5],
[6.9, 3.1, 4.9, 1.5],
[5.5, 2.3, 4. , 1.3],
[6.5, 2.8, 4.6, 1.5],
[5.7, 2.8, 4.5, 1.3],
[6.3, 3.3, 4.7, 1.6],
[4.9, 2.4, 3.3, 1. ],
[6.6, 2.9, 4.6, 1.3],
[5.2, 2.7, 3.9, 1.4],
[5. , 2. , 3.5, 1. ],
[5.9, 3. , 4.2, 1.5],
[6. , 2.2, 4. , 1. ],
[6.1, 2.9, 4.7, 1.4],
[5.6, 2.9, 3.6, 1.3],
[6.7, 3.1, 4.4, 1.4],
[5.6, 3. , 4.5, 1.5],
[5.8, 2.7, 4.1, 1. ],
[6.2, 2.2, 4.5, 1.5],
[5.6, 2.5, 3.9, 1.1],
[5.9, 3.2, 4.8, 1.8],
[6.1, 2.8, 4. , 1.3],
[6.3, 2.5, 4.9, 1.5],
[6.1, 2.8, 4.7, 1.2],
[6.4, 2.9, 4.3, 1.3],
[6.6, 3. , 4.4, 1.4],
[6.8, 2.8, 4.8, 1.4],
[6.7, 3. , 5. , 1.7],
[6. , 2.9, 4.5, 1.5],
[5.7, 2.6, 3.5, 1. ],
[5.5, 2.4, 3.8, 1.1],
[5.5, 2.4, 3.7, 1. ],
[5.8, 2.7, 3.9, 1.2],
[6. , 2.7, 5.1, 1.6],
[5.4, 3. , 4.5, 1.5],
[6. , 3.4, 4.5, 1.6],
[6.7, 3.1, 4.7, 1.5],
[6.3, 2.3, 4.4, 1.3],
[5.6, 3. , 4.1, 1.3],
[5.5, 2.5, 4. , 1.3],
[5.5, 2.6, 4.4, 1.2],
[6.1, 3. , 4.6, 1.4],
[5.8, 2.6, 4. , 1.2],
[5. , 2.3, 3.3, 1. ],
[5.6, 2.7, 4.2, 1.3],
[5.7, 3. , 4.2, 1.2],
[5.7, 2.9, 4.2, 1.3],
[6.2, 2.9, 4.3, 1.3],
[5.1, 2.5, 3. , 1.1],
[5.7, 2.8, 4.1, 1.3],
[6.3, 3.3, 6. , 2.5],
[5.8, 2.7, 5.1, 1.9],
[7.1, 3. , 5.9, 2.1],
[6.3, 2.9, 5.6, 1.8],
[6.5, 3. , 5.8, 2.2],
[7.6, 3. , 6.6, 2.1],
[4.9, 2.5, 4.5, 1.7],
[7.3, 2.9, 6.3, 1.8],
[6.7, 2.5, 5.8, 1.8],
```

```
       [7.2, 3.6, 6.1, 2.5],
       [6.5, 3.2, 5.1, 2. ],
       [6.4, 2.7, 5.3, 1.9],
       [6.8, 3. , 5.5, 2.1],
       [5.7, 2.5, 5. , 2. ],
       [5.8, 2.8, 5.1, 2.4],
       [6.4, 3.2, 5.3, 2.3],
       [6.5, 3. , 5.5, 1.8],
       [7.7, 3.8, 6.7, 2.2],
       [7.7, 2.6, 6.9, 2.3],
       [6. , 2.2, 5. , 1.5],
       [6.9, 3.2, 5.7, 2.3],
       [5.6, 2.8, 4.9, 2. ],
       [7.7, 2.8, 6.7, 2. ],
       [6.3, 2.7, 4.9, 1.8],
       [6.7, 3.3, 5.7, 2.1],
       [7.2, 3.2, 6. , 1.8],
       [6.2, 2.8, 4.8, 1.8],
       [6.1, 3. , 4.9, 1.8],
       [6.4, 2.8, 5.6, 2.1],
       [7.2, 3. , 5.8, 1.6],
       [7.4, 2.8, 6.1, 1.9],
       [7.9, 3.8, 6.4, 2. ],
       [6.4, 2.8, 5.6, 2.2],
       [6.3, 2.8, 5.1, 1.5],
       [6.1, 2.6, 5.6, 1.4],
       [7.7, 3. , 6.1, 2.3],
       [6.3, 3.4, 5.6, 2.4],
       [6.4, 3.1, 5.5, 1.8],
       [6. , 3. , 4.8, 1.8],
       [6.9, 3.1, 5.4, 2.1],
       [6.7, 3.1, 5.6, 2.4],
       [6.9, 3.1, 5.1, 2.3],
       [5.8, 2.7, 5.1, 1.9],
       [6.8, 3.2, 5.9, 2.3],
       [6.7, 3.3, 5.7, 2.5],
       [6.7, 3. , 5.2, 2.3],
       [6.3, 2.5, 5. , 1.9],
       [6.5, 3. , 5.2, 2. ],
       [6.2, 3.4, 5.4, 2.3],
       [5.9, 3. , 5.1, 1.8]], dtype=float32)
```

In [6]:

```
y
```

Out[6]:

```
array([[1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
```

```
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
```

```
       [0., 1., 0.],
       [0., 1., 0.],
       [0., 1., 0.],
       [0., 1., 0.],
       [0., 1., 0.],
       [0., 1., 0.],
       [0., 1., 0.],
       [0., 1., 0.],
       [0., 1., 0.],
       [0., 1., 0.],
       [0., 1., 0.],
       [0., 1., 0.],
       [0., 1., 0.],
       [0., 1., 0.],
       [0., 1., 0.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.]], dtype=float32)
```

Reflection: why each output in y has three columns?

Training and Validation

The machine learning model will learn from the training data, but ultimately be evaluated based on the validation data.

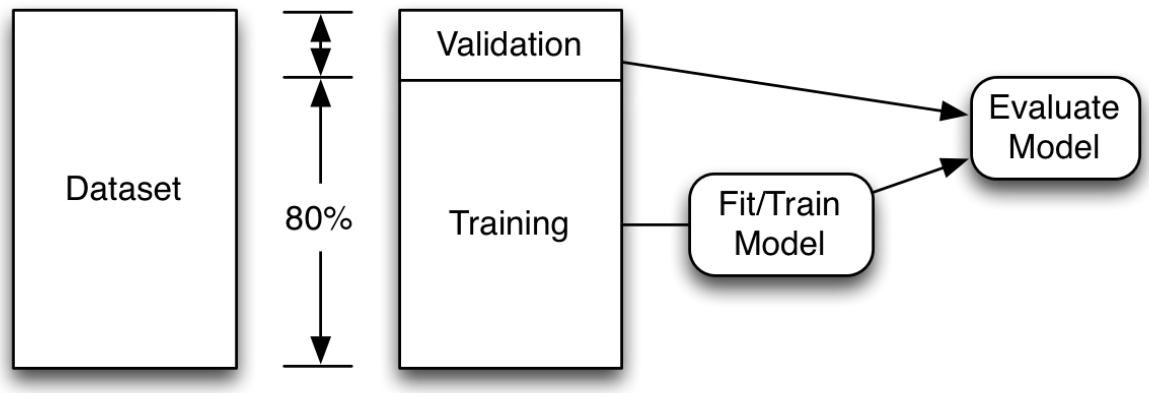
- **Training Data** - **In Sample Data** - The data that the machine learning model was fit to/created from.
- **Validation Data** - **Out of Sample Data** - The data that the machine learning model is evaluated upon after it is fit to the training data.

- **Training/Test Split** - The data are split according to some ratio between a training and validation (hold-out) set. Common ratios are 80% training and 20% validation.

Training/Test Split using train_test_split()

The code below performs a split of the MPG data into a training and validation set. The training set uses 80% of the data and the test(validation) set uses 20%.

The following image shows how a model is trained on 80% of the data and then validated against the remaining 20%.



In [7]:

```
import pandas as pd
import io
import numpy as np
import os
from sklearn.model_selection import train_test_split


path = "./data/"

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

df
```

Out[7]:

	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
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

150 rows × 5 columns

In [8]:

```
encode_text_index(df,"species")

df

# alternatively,we can use LabelEncoder() for label encoding
#from sklearn import preprocessing
#le = preprocessing.LabelEncoder()
```

Out[8]:

	sepal_l	sepal_w	petal_l	petal_w	species
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	2
146	6.3	2.5	5.0	1.9	2
147	6.5	3.0	5.2	2.0	2
148	6.2	3.4	5.4	2.3	2
149	5.9	3.0	5.1	1.8	2

150 rows × 5 columns

In [9]:

```
x,y = to_xy(df,"species")
```

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

In [10]:
```
x_train.shape
```

Out[10]: (112, 4)

In [11]:
```
y_train.shape
```

Out[11]: (112, 3)

In [12]:
```
x_test.shape
```

Out[12]: (38, 4)

In [13]:
```
y_test.shape
```

Out[13]: (38, 3)

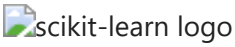
In [ ]:
```

```

CSC 180 Intelligent Systems (Fall 2022)

Dr. Haiquan Chen, Dept of Computer Scicence

California State University, Sacramento

Lab 3: Supervised Learning using Sklearn



Motivation problem: Iris flower dataset

150 observations where each obervation is of one of the following types of irises' (Setosa, Versicolour, and Virginica)

4 features (sepal length, sepal width, petal length, petal width)

Label variable is the iris species

Machine learning terminology

- Each row is an **observation** (also known as: object, sample, example, instance, record)
- Each column is a **feature** (also known as: predictor, attribute, independent variable, input)
- Each value we are predicting is the **response** (also known as: target, outcome, label, dependent variable)

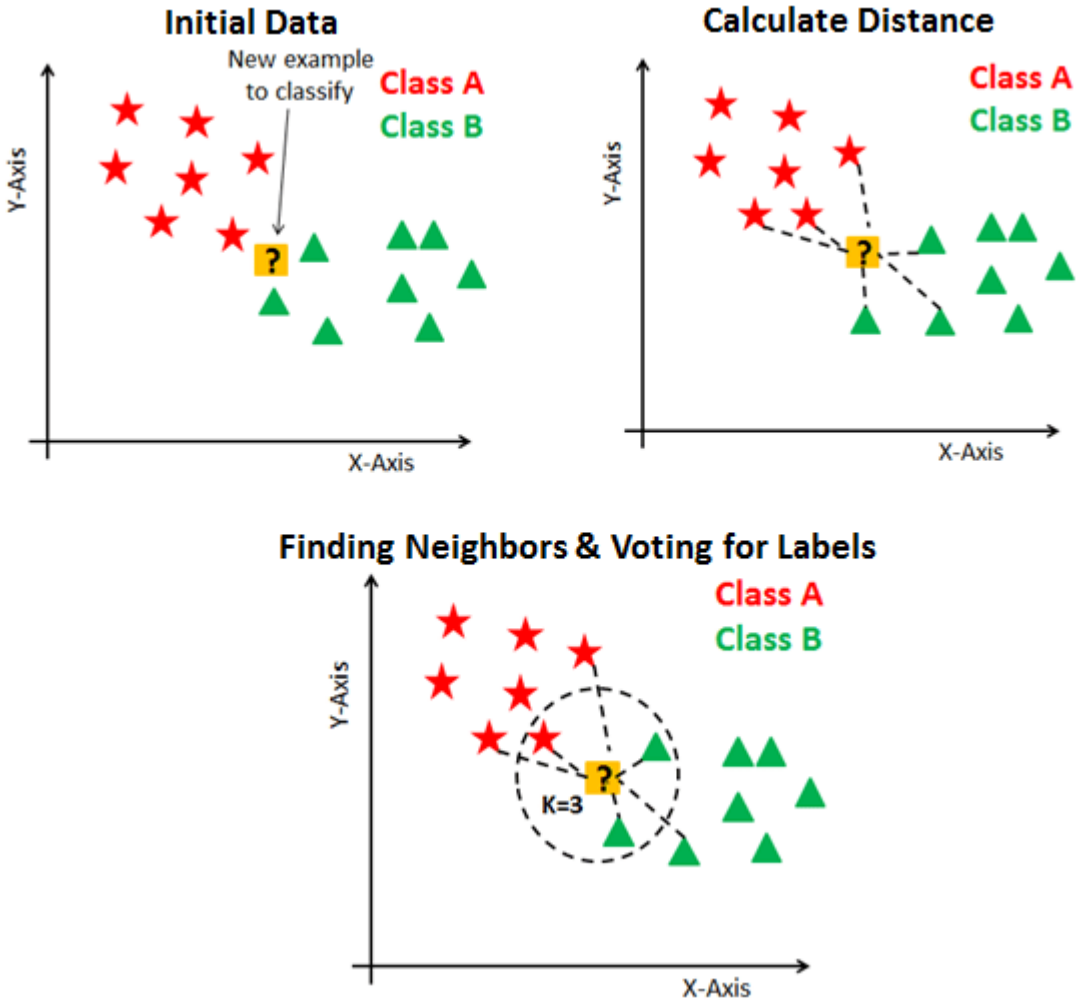
This is a classification problem since label is discrete!

Suppose we want to use K-nearest neighbors (KNN)

1. Pick a value for K.

2. For any *unknown iris*, find the top-K records in our data that are "nearest" to that iris.

3. Majoity vote. Use the *most popular species* of those K records as the predicted species for the unknown iris.



Prepare the training data in terms of X (feature matrix) and y (label/target vector)

X = input; y = output

Scikit-learn comes with a few datasets

In [1]:
```
# import load_iris function from datasets module
from sklearn.datasets import load_iris

iris = load_iris()
```

In [2]:
```
# store all the input features in a matrix "X"
X = iris.data
# print the shape of X
print(X.shape)
print(type(X))
```

```
(150, 4)
<class 'numpy.ndarray'>
```

In [3]:
```
X
```

Out[3]:
```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       [4.6, 3.1, 1.5, 0.2],
       [5. , 3.6, 1.4, 0.2],
       [5.4, 3.9, 1.7, 0.4],
       [4.6, 3.4, 1.4, 0.3],
       [5. , 3.4, 1.5, 0.2],
       [4.4, 2.9, 1.4, 0.2],
       [4.9, 3.1, 1.5, 0.1],
       [5.4, 3.7, 1.5, 0.2],
       [4.8, 3.4, 1.6, 0.2],
       [4.8, 3. , 1.4, 0.1],
       [4.3, 3. , 1.1, 0.1],
       [5.8, 4. , 1.2, 0.2],
       [5.7, 4.4, 1.5, 0.4],
       [5.4, 3.9, 1.3, 0.4],
       [5.1, 3.5, 1.4, 0.3],
       [5.7, 3.8, 1.7, 0.3],
       [5.1, 3.8, 1.5, 0.3],
       [5.4, 3.4, 1.7, 0.2],
       [5.1, 3.7, 1.5, 0.4],
       [4.6, 3.6, 1. , 0.2],
       [5.1, 3.3, 1.7, 0.5],
       [4.8, 3.4, 1.9, 0.2],
       [5. , 3. , 1.6, 0.2],
       [5. , 3.4, 1.6, 0.4],
       [5.2, 3.5, 1.5, 0.2],
       [5.2, 3.4, 1.4, 0.2],
       [4.7, 3.2, 1.6, 0.2],
       [4.8, 3.1, 1.6, 0.2],
       [5.4, 3.4, 1.5, 0.4],
       [5.2, 4.1, 1.5, 0.1],
       [5.5, 4.2, 1.4, 0.2],
       [4.9, 3.1, 1.5, 0.2],
       [5. , 3.2, 1.2, 0.2],
       [5.5, 3.5, 1.3, 0.2],
       [4.9, 3.6, 1.4, 0.1],
       [4.4, 3. , 1.3, 0.2],
       [5.1, 3.4, 1.5, 0.2],
       [5. , 3.5, 1.3, 0.3],
       [4.5, 2.3, 1.3, 0.3],
       [4.4, 3.2, 1.3, 0.2],
       [5. , 3.5, 1.6, 0.6],
       [5.1, 3.8, 1.9, 0.4],
       [4.8, 3. , 1.4, 0.3],
       [5.1, 3.8, 1.6, 0.2],
       [4.6, 3.2, 1.4, 0.2],
       [5.3, 3.7, 1.5, 0.2],
       [5. , 3.3, 1.4, 0.2],
       [7. , 3.2, 4.7, 1.4],
       [6.4, 3.2, 4.5, 1.5],
       [6.9, 3.1, 4.9, 1.5],
       [5.5, 2.3, 4. , 1.3],
       [6.5, 2.8, 4.6, 1.5],
       [5.7, 2.8, 4.5, 1.3],
       [6.3, 3.3, 4.7, 1.6],
       [4.9, 2.4, 3.3, 1. ],
       [6.6, 2.9, 4.6, 1.3],
       [5.2, 2.7, 3.9, 1.4],
       [5. , 2. , 3.5, 1. ],
       [5.9, 3. , 4.2, 1.5],
       [6. , 2.2, 4. , 1. ],
       [6.1, 2.9, 4.7, 1.4],
```

```
[5.6, 2.9, 3.6, 1.3],
[6.7, 3.1, 4.4, 1.4],
[5.6, 3. , 4.5, 1.5],
[5.8, 2.7, 4.1, 1. ],
[6.2, 2.2, 4.5, 1.5],
[5.6, 2.5, 3.9, 1.1],
[5.9, 3.2, 4.8, 1.8],
[6.1, 2.8, 4. , 1.3],
[6.3, 2.5, 4.9, 1.5],
[6.1, 2.8, 4.7, 1.2],
[6.4, 2.9, 4.3, 1.3],
[6.6, 3. , 4.4, 1.4],
[6.8, 2.8, 4.8, 1.4],
[6.7, 3. , 5. , 1.7],
[6. , 2.9, 4.5, 1.5],
[5.7, 2.6, 3.5, 1. ],
[5.5, 2.4, 3.8, 1.1],
[5.5, 2.4, 3.7, 1. ],
[5.8, 2.7, 3.9, 1.2],
[6. , 2.7, 5.1, 1.6],
[5.4, 3. , 4.5, 1.5],
[6. , 3.4, 4.5, 1.6],
[6.7, 3.1, 4.7, 1.5],
[6.3, 2.3, 4.4, 1.3],
[5.6, 3. , 4.1, 1.3],
[5.5, 2.5, 4. , 1.3],
[5.5, 2.6, 4.4, 1.2],
[6.1, 3. , 4.6, 1.4],
[5.8, 2.6, 4. , 1.2],
[5. , 2.3, 3.3, 1. ],
[5.6, 2.7, 4.2, 1.3],
[5.7, 3. , 4.2, 1.2],
[5.7, 2.9, 4.2, 1.3],
[6.2, 2.9, 4.3, 1.3],
[5.1, 2.5, 3. , 1.1],
[5.7, 2.8, 4.1, 1.3],
[6.3, 3.3, 6. , 2.5],
[5.8, 2.7, 5.1, 1.9],
[7.1, 3. , 5.9, 2.1],
[6.3, 2.9, 5.6, 1.8],
[6.5, 3. , 5.8, 2.2],
[7.6, 3. , 6.6, 2.1],
[4.9, 2.5, 4.5, 1.7],
[7.3, 2.9, 6.3, 1.8],
[6.7, 2.5, 5.8, 1.8],
[7.2, 3.6, 6.1, 2.5],
[6.5, 3.2, 5.1, 2. ],
[6.4, 2.7, 5.3, 1.9],
[6.8, 3. , 5.5, 2.1],
[5.7, 2.5, 5. , 2. ],
[5.8, 2.8, 5.1, 2.4],
[6.4, 3.2, 5.3, 2.3],
[6.5, 3. , 5.5, 1.8],
[7.7, 3.8, 6.7, 2.2],
[7.7, 2.6, 6.9, 2.3],
[6. , 2.2, 5. , 1.5],
[6.9, 3.2, 5.7, 2.3],
[5.6, 2.8, 4.9, 2. ],
[7.7, 2.8, 6.7, 2. ],
[6.3, 2.7, 4.9, 1.8],
[6.7, 3.3, 5.7, 2.1],
[7.2, 3.2, 6. , 1.8],
[6.2, 2.8, 4.8, 1.8],
[6.1, 3. , 4.9, 1.8],
[6.4, 2.8, 5.6, 2.1],
```

```
       [7.2, 3. , 5.8, 1.6],
       [7.4, 2.8, 6.1, 1.9],
       [7.9, 3.8, 6.4, 2. ],
       [6.4, 2.8, 5.6, 2.2],
       [6.3, 2.8, 5.1, 1.5],
       [6.1, 2.6, 5.6, 1.4],
       [7.7, 3. , 6.1, 2.3],
       [6.3, 3.4, 5.6, 2.4],
       [6.4, 3.1, 5.5, 1.8],
       [6. , 3. , 4.8, 1.8],
       [6.9, 3.1, 5.4, 2.1],
       [6.7, 3.1, 5.6, 2.4],
       [6.9, 3.1, 5.1, 2.3],
       [5.8, 2.7, 5.1, 1.9],
       [6.8, 3.2, 5.9, 2.3],
       [6.7, 3.3, 5.7, 2.5],
       [6.7, 3. , 5.2, 2.3],
       [6.3, 2.5, 5. , 1.9],
       [6.5, 3. , 5.2, 2. ],
       [6.2, 3.4, 5.4, 2.3],
       [5.9, 3. , 5.1, 1.8]])
```

In [4]:

```
# store response/lable vector in "y"
y = iris.target
# print the shape of y
print(y.shape)
```

```
(150,)
```

In [5]:

```
y
```

Out[5]:

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

y is already label-encoded: 0 for 'setosa', 1 for 'versicolor', 2 for 'virginica'

In [6]:

```
y
```

Out[6]:

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

In [6]:

```
iris.target_names
```

Out[6]:

```
array(['setosa', 'versicolor', 'virginica'], dtype='<U10')
```

Scikit-learn 4-step modeling pattern (ICFP)

Step 1: Import the *machine learning model* you plan to use.

Check official API documentation here https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

In [7]:
```
from sklearn.neighbors import KNeighborsClassifier
```

Step 2: Create an instance of the model with the parameters specified

In [9]:
```
knn = KNeighborsClassifier(n_neighbors=3)

# http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifi
```

- Name of the object does not matter
- All parameters not specified are set to their defaults

Step 3: Fit the model with data (aka "model training")

- Model is learning the relationship between ***X (Input Matrix)*** and ***y (Target Vector)*** ### Notice that X must be a Matrix, y must be 1-d array

In [10]:
```
knn.fit(X, y)     # X must be a Matrix,  y must be 1-d array
```

Out[10]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=3, p=2,
                     weights='uniform')
```

Step 4: Predict the target for a new record

- precict() takes a matrix (2D) as input

In [11]:
```
knn.predict([[3, 5, 4, 2]])            #  (sepal length, sepal width, petal length, pet
                                       # ('setosa', 'versicolor', 'virginica')
```

Out[11]:
```
array([1])
```

It is versicolor!!

- Returns a NumPy array
- Can predict for multiple observations at once

In [12]:
```
X_new = [[3, 5, 4, 2], [5, 4, 3, 2]]
```

In [13]:
```
knn.predict(X_new)
```

Out[13]:
```
array([1, 1])
```

Using a different value for K

In [14]:
```
# instantiate the model (using the value K=10)
knn = KNeighborsClassifier(n_neighbors=10)

# fit the model with data
knn.fit(X, y)

# predict the response for new observations
knn.predict(X_new)
```

Out[14]:
```
array([1, 1])
```

Using a different classification model

In [15]:
```
# import the class
from sklearn.linear_model import LogisticRegression

# instantiate the model
logreg = LogisticRegression(solver='lbfgs', multi_class='ovr')

# fit the model with data
logreg.fit(X, y)

# predict the response for new observations
logreg.predict(X_new)
```

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

In [ ]:
```

```

CSC 180 Intelligent Systems (Fall 2022)

Dr. Haiquan Chen, Dept of Computer Scicence

California State University, Sacramento

Lab 4: Natural Language Processing using Sklearn

In [3]:
```
import numpy as np
import pandas as pd

#import matplotlib.pyplot as plt
#%matplotlib inline
```

How to vectorize natural lauguage data?

Python offers a set of tools for extracting features:http://scikit-learn.org/stable/modules/feature_extraction.html

Bag-of-words Model:

CountVectorizer: transforms text into a "sparse matrix" where rows are text and columns are words, and values are occurrence values.

In [4]:
```
import sklearn.feature_extraction.text as sk_text

vectorizer = sk_text.CountVectorizer(min_df=1)
#vectorizer = sk_text.CountVectorizer(min_df=1, stop_words = 'english')

#min_df: ignore terms that have a document frequency < min_df.

corpus = ['This is the first document.',
          'this is the second second document.',
          'And the third one.',
          'Is this the first first first document?',
         ]

matrix = vectorizer.fit_transform(corpus)

print(type(matrix))          # Compressed Sparse Row matrix
print(matrix.toarray())        #  convert it to numpy array
```

```
<class 'scipy.sparse.csr.csr_matrix'>
[[0 1 1 1 0 0 1 0 1]
 [0 1 0 1 0 2 1 0 1]
 [1 0 0 0 1 0 1 1 0]
 [0 1 3 1 0 0 1 0 1]]
```

In [2]:
```
print(vectorizer.get_feature_names())
```

```
['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']
```

TF-IDF model:

TfIdfVectorizer: transforms text into a "sparse matrix" where rows are text and columns are words, and values are the tf-dif values.

More here: http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html#sklearn.feature_

In [4]:
```
vectorizer = sk_text.TfidfVectorizer(
                            #stop_words='english',
                            max_features = 1000,
                            min_df=1)


#max_features:  build a vocabulary that only consider the top max_features features ord

matrix = vectorizer.fit_transform(corpus)

print(type(matrix))          # Compressed Sparse Row matrix
print(matrix.toarray())        #  convert it to numpy array
```

```
<class 'scipy.sparse.csr.csr_matrix'>
[[0.         0.43877674 0.54197657 0.43877674 0.         0.
  0.35872874 0.         0.43877674]
 [0.         0.27230147 0.         0.27230147 0.         0.85322574
  0.22262429 0.         0.27230147]
 [0.55280532 0.         0.         0.         0.55280532 0.
  0.28847675 0.55280532 0.        ]
 [0.         0.23973261 0.88835239 0.23973261 0.         0.
  0.19599711 0.         0.23973261]]
```

In [5]:
```
print(vectorizer.get_feature_names())
```

```
['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']
```

In [6]:
```
vectorizer = sk_text.TfidfVectorizer(#stop_words='english',
                            #max_features = 1000,
                            min_df=2,
                            max_df=500
                            )

#min_df: ignore terms that have a document frequency < min_df.
#max_df: ignore terms that have a document frequency > max_df


matrix = vectorizer.fit_transform(corpus)
print(type(matrix))               # Compressed Sparse Row matrix

tfidf_data = matrix.toarray()     #  convert it to numpy array

print(tfidf_data)
```

```
<class 'scipy.sparse.csr.csr_matrix'>
[[0.43877674 0.54197657 0.43877674 0.35872874 0.43877674]
 [0.52210862 0.         0.52210862 0.42685801 0.52210862]]
```

```
[0.         0.         0.         1.         0.        ]
[0.23973261 0.88835239 0.23973261 0.19599711 0.23973261]]
```

In [7]:
```
print(vectorizer.get_feature_names())
```

```
['document', 'first', 'is', 'the', 'this']
```

Once you vectorize the text, you can send the data to models

In [8]:
```
tfidf_data
```

Out[8]:
```
array([[0.43877674, 0.54197657, 0.43877674, 0.35872874, 0.43877674],
       [0.52210862, 0.        , 0.52210862, 0.42685801, 0.52210862],
       [0.        , 0.        , 0.        , 1.        , 0.        ],
       [0.23973261, 0.88835239, 0.23973261, 0.19599711, 0.23973261]])
```

In [9]:
```
tfidf_data.shape
```

Out[9]:
```
(4, 5)
```

Another example:

In [6]:
```
document_0 = "Japan's prime minister, Shinzo Abe, is working towards healing the econom
document_1 = "Vladimir Putin is working hard to fix the economy in Russia as the Ruble
document_2 = "What's the future of Abenomics? We asked Shinzo Abe for his views"
document_3 = "Obama has eased sanctions on Cuba while accelerating those against the Ru
document_4 = "Vladimir Putin is riding a horse while hunting deer. Vladimir Putin alway


corpus = [document_0, document_1, document_2, document_3, document_4]
```

In [7]:
```
vectorizer = sk_text.TfidfVectorizer(stop_words='english',
                             max_features = 100,
                             min_df=1,
                             #max_df=5
                             )

#min_df: ignore terms that have a document frequency < min_df.
#max_df: ignore terms that have a document frequency > max_df


matrix = vectorizer.fit_transform(corpus)

tfidf_data = matrix.toarray()     #  convert it to numpy array

print(tfidf_data)
print(tfidf_data.shape)
print(vectorizer.get_feature_names())
print(len(vectorizer.get_feature_names()))
```

```
[[0.23684538 0.         0.         0.         0.29356375 0.
  0.         0.         0.         0.         0.29356375 0.
  0.         0.         0.23684538 0.         0.29356375 0.
  0.         0.         0.29356375 0.29356375 0.         0.29356375
  0.29356375 0.         0.         0.         0.         0.
```

```
  0.         0.23684538 0.         0.         0.29356375 0.
  0.29356375 0.         0.         0.23684538]
 [0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.29954105
  0.         0.37127341 0.         0.37127341 0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.29954105 0.         0.29954105 0.37127341 0.
  0.         0.         0.         0.37127341 0.         0.
  0.         0.         0.29954105 0.29954105]
 [0.36252618 0.44934185 0.         0.44934185 0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.36252618 0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.36252618 0.         0.         0.         0.
  0.         0.44934185 0.         0.        ]
 [0.         0.         0.31156077 0.         0.         0.
  0.31156077 0.31156077 0.         0.31156077 0.         0.25136527
  0.31156077 0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.31156077 0.
  0.         0.         0.         0.25136527 0.         0.31156077
  0.31156077 0.         0.         0.         0.         0.31156077
  0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.25643278
  0.         0.         0.25643278 0.         0.         0.
  0.         0.         0.         0.         0.         0.25643278
  0.25643278 0.25643278 0.         0.         0.         0.
  0.         0.4137767  0.51286555 0.         0.         0.
  0.         0.         0.25643278 0.         0.         0.
  0.         0.         0.4137767  0.        ]]
(5, 40)
['abe', 'abenomics', 'accelerating', 'asked', 'country', 'crazy', 'cuba', 'daily', 'dee
r', 'eased', 'economic', 'economy', 'falls', 'fix', 'future', 'hard', 'healing', 'hors
e', 'horses', 'hunting', 'japan', 'minister', 'obama', 'people', 'prime', 'putin', 'ridi
ng', 'ruble', 'russia', 'russian', 'sanctions', 'shinzo', 'things', 'tumbled', 'turmoi
l', 'value', 'view', 'views', 'vladimir', 'working']
40
```

In [ ]:

CSC 180 Intelligent Systems (Fall 2022)

Dr. Haiquan Chen, Dept of Computer Scicence

California State University, Sacramento

Lab 5: Tensorflow Introduction

Helpful Functions for Tensorflow (little gems)

The following functions will be used with TensorFlow to help preprocess the data. They allow you to build the feature vector for a neural network.

- Predictors/Inputs
  - Fill any missing inputs with the median for that column. Use **missing_median**.
  - Encode textual/categorical values with **encode_text_dummy**.
  - Encode numeric values with **encode_numeric_zscore**.
- Output
  - Discard rows with missing outputs.
  - Encode textual/categorical values with **encode_text_index**.
  - Do not encode output numeric values.
- Produce final feature vectors (x) and expected output (y) with **to_xy**.

In [1]:
```
from collections.abc import Sequence
from sklearn import preprocessing
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import shutil
import os


# Encode text values to dummy variables(i.e. [1,0,0],[0,1,0],[0,0,1] for red,green,blue
def encode_text_dummy(df, name):
    dummies = pd.get_dummies(df[name])
    for x in dummies.columns:
        dummy_name = "{}-{}".format(name, x)
        df[dummy_name] = dummies[x]
    df.drop(name, axis=1, inplace=True)



# Encode text values to indexes(i.e. [1],[2],[3] for red,green,blue).
def encode_text_index(df, name):
    le = preprocessing.LabelEncoder()
    df[name] = le.fit_transform(df[name])
    return le.classes_


# Encode a numeric column as zscores
def encode_numeric_zscore(df, name, mean=None, sd=None):
```

```
    if mean is None:
        mean = df[name].mean()

    if sd is None:
        sd = df[name].std()

    df[name] = (df[name] - mean) / sd


# Convert all missing values in the specified column to the median
def missing_median(df, name):
    med = df[name].median()
    df[name] = df[name].fillna(med)


# Convert all missing values in the specified column to the default
def missing_default(df, name, default_value):
    df[name] = df[name].fillna(default_value)


# Convert a Pandas dataframe to the x,y inputs that TensorFlow needs
def to_xy(df, target):
    result = []
    for x in df.columns:
        if x != target:
            result.append(x)
    # find out the type of the target column.
    target_type = df[target].dtypes
    target_type = target_type[0] if isinstance(target_type, Sequence) else target_type
    # Encode to int for classification, float otherwise. TensorFlow likes 32 bits.
    if target_type in (np.int64, np.int32):
        # Classification
        dummies = pd.get_dummies(df[target])
        return df[result].values.astype(np.float32), dummies.values.astype(np.float32)
    else:
        # Regression
        return df[result].values.astype(np.float32), df[target].values.astype(np.float3

# 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)


# 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)
    a = plt.plot(t['y'].tolist(),label='expected')
    b = plt.plot(t['pred'].tolist(),label='prediction')
    plt.ylabel('output')
    plt.legend()
    plt.show()

# 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)


# Encode a column to a range between normalized_low and normalized_high.
def encode_numeric_range(df, name, normalized_low=-1, normalized_high=1,
                         data_low=None, data_high=None):
    if data_low is None:
        data_low = min(df[name])
        data_high = max(df[name])

    df[name] = ((df[name] - data_low) / (data_high - data_low)) \
               * (normalized_high - normalized_low) + normalized_low
```

Classification or Regression

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.

Regression networks always have a single output neuron. Classification neural networks have an output neuron for each class.

These neurons are grouped into layers:

- **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 that occur between the input and output layers. Each hidden layer will usually have a bias neuron.

What version of TensorFlow do you have?

TensorFlow is very new and changing rapidly.

In [1]:
```
import tensorflow as tf
print("Tensor Flow Version: {}".format(tf.__version__))
```

```
Tensor Flow Version: 2.3.0
```

Why TensorFlow

- Supported by Google
- Works well on Linux/Mac
- Excellent GPU support
- Most popular today

Using Keras

Keras is an official high-end API, supported by Google, for Tensorflow that makes it much easier to create neural networks.

In [2]:
```
import tensorflow.keras
print("Keras Version: {}".format(tensorflow.keras.__version__))
```

```
Keras Version: 2.4.0
```

Example of TensorFlow Regression: MPG Prediction

This example shows how to encode the MPG dataset for regression. Notice that:

- Input has both numeric and categorical
- Input has missing values

In [4]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
import pandas as pd
import io
import requests
import numpy as np
from sklearn import metrics
path = "./data/"

filename_read = os.path.join(path,"auto-mpg.csv")
df = pd.read_csv(filename_read,na_values=['NA','?'])

df[0:5]
```

Out[4]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin	name
0	18.0	8	307.0	130.0	3504	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165.0	3693	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150.0	3436	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150.0	3433	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140.0	3449	10.5	70	1	ford torino

In [5]:
```
cars = df['name']

df.drop('name',1,inplace=True)

missing_median(df, 'horsepower')

encode_text_dummy(df, 'origin')
```

```
x,y = to_xy(df,"mpg")
```

In [6]:
```
x.shape
```

Out[6]:
```
(398, 9)
```

In [7]:
```
y.shape
```

Out[7]:
```
(398,)
```

In [8]:
```
x
```

Out[8]:
```
array([[  8., 307., 130., ...,   1.,   0.,   0.],
       [  8., 350., 165., ...,   1.,   0.,   0.],
       [  8., 318., 150., ...,   1.,   0.,   0.],
       ...,
       [  4., 135.,  84., ...,   1.,   0.,   0.],
       [  4., 120.,  79., ...,   1.,   0.,   0.],
       [  4., 119.,  82., ...,   1.,   0.,   0.]], dtype=float32)
```

In [9]:
```
y
```

Out[9]:
```
array([18. , 15. , 18. , 16. , 17. , 15. , 14. , 14. , 14. , 15. , 15. ,
       14. , 15. , 14. , 24. , 22. , 18. , 21. , 27. , 26. , 25. , 24. ,
       25. , 26. , 21. , 10. , 10. , 11. ,  9. , 27. , 28. , 25. , 25. ,
       19. , 16. , 17. , 19. , 18. , 14. , 14. , 14. , 14. , 12. , 13. ,
       13. , 18. , 22. , 19. , 18. , 23. , 28. , 30. , 30. , 31. , 35. ,
       27. , 26. , 24. , 25. , 23. , 20. , 21. , 13. , 14. , 15. , 14. ,
       17. , 11. , 13. , 12. , 13. , 19. , 15. , 13. , 13. , 14. , 18. ,
       22. , 21. , 26. , 22. , 28. , 23. , 28. , 27. , 13. , 14. , 13. ,
       14. , 15. , 12. , 13. , 13. , 14. , 13. , 12. , 13. , 18. , 16. ,
       18. , 18. , 23. , 26. , 11. , 12. , 13. , 12. , 18. , 20. , 21. ,
       22. , 18. , 19. , 21. , 26. , 15. , 16. , 29. , 24. , 20. , 19. ,
       15. , 24. , 20. , 11. , 20. , 21. , 19. , 15. , 31. , 26. , 32. ,
       25. , 16. , 16. , 18. , 16. , 13. , 14. , 14. , 14. , 29. , 26. ,
       26. , 31. , 32. , 28. , 24. , 26. , 24. , 26. , 31. , 19. , 18. ,
       15. , 15. , 16. , 15. , 16. , 14. , 17. , 16. , 15. , 18. , 21. ,
       20. , 13. , 29. , 23. , 20. , 23. , 24. , 25. , 24. , 18. , 29. ,
       19. , 23. , 23. , 22. , 25. , 33. , 28. , 25. , 25. , 26. , 27. ,
       17.5, 16. , 15.5, 14.5, 22. , 22. , 24. , 22.5, 29. , 24.5, 29. ,
       33. , 20. , 18. , 18.5, 17.5, 29.5, 32. , 28. , 26.5, 20. , 13. ,
       19. , 19. , 16.5, 16.5, 13. , 13. , 13. , 31.5, 30. , 36. , 25.5,
       33.5, 17.5, 17. , 15.5, 15. , 17.5, 20.5, 19. , 18.5, 16. , 15.5,
       15.5, 16. , 29. , 24.5, 26. , 25.5, 30.5, 33.5, 30. , 30.5, 22. ,
       21.5, 21.5, 43.1, 36.1, 32.8, 39.4, 36.1, 19.9, 19.4, 20.2, 19.2,
       20.5, 20.2, 25.1, 20.5, 19.4, 20.6, 20.8, 18.6, 18.1, 19.2, 17.7,
       18.1, 17.5, 30. , 27.5, 27.2, 30.9, 21.1, 23.2, 23.8, 23.9, 20.3,
       17. , 21.6, 16.2, 31.5, 29.5, 21.5, 19.8, 22.3, 20.2, 20.6, 17. ,
       17.6, 16.5, 18.2, 16.9, 15.5, 19.2, 18.5, 31.9, 34.1, 35.7, 27.4,
       25.4, 23. , 27.2, 23.9, 34.2, 34.5, 31.8, 37.3, 28.4, 28.8, 26.8,
       33.5, 41.5, 38.1, 32.1, 37.2, 28. , 26.4, 24.3, 19.1, 34.3, 29.8,
       31.3, 37. , 32.2, 46.6, 27.9, 40.8, 44.3, 43.4, 36.4, 30. , 44.6,
       40.9, 33.8, 29.8, 32.7, 23.7, 35. , 23.6, 32.4, 27.2, 26.6, 25.8,
       23.5, 30. , 39.1, 39. , 35.1, 32.3, 37. , 37.7, 34.1, 34.7, 34.4,
       29.9, 33. , 34.5, 33.7, 32.4, 32.9, 31.6, 28.1, 30.7, 25.4, 24.2,
       22.4, 26.6, 20.2, 17.6, 28. , 27. , 34. , 31. , 29. , 27. , 24. ,
```

```
       23. , 36. , 37. , 31. , 38. , 36. , 36. , 36. , 34. , 38. , 32. ,
       38. , 25. , 38. , 26. , 22. , 32. , 36. , 27. , 27. , 44. , 32. ,
       28. , 31. ], dtype=float32)
```

In [10]:

```
model = Sequential()

model.add(Dense(25, input_dim=x.shape[1], activation='relu')) # Hidden 1     #  why inp
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)    # Verbosity mode. 0 = silent, 1 = progress bar,
```

```
Train on 398 samples
Epoch 1/100
398/398 - 1s - loss: 121507.2316
Epoch 2/100
398/398 - 0s - loss: 32366.2281
Epoch 3/100
398/398 - 0s - loss: 6560.8209
Epoch 4/100
398/398 - 0s - loss: 458.7371
Epoch 5/100
398/398 - 0s - loss: 180.6163
Epoch 6/100
398/398 - 0s - loss: 211.6260
Epoch 7/100
398/398 - 0s - loss: 121.1665
Epoch 8/100
398/398 - 0s - loss: 94.5356
Epoch 9/100
398/398 - 0s - loss: 94.0867
Epoch 10/100
398/398 - 0s - loss: 93.0899
Epoch 11/100
398/398 - 0s - loss: 92.4133
Epoch 12/100
398/398 - 0s - loss: 92.4059
Epoch 13/100
398/398 - 0s - loss: 92.0073
Epoch 14/100
398/398 - 0s - loss: 91.9448
Epoch 15/100
398/398 - 0s - loss: 91.8758
Epoch 16/100
398/398 - 0s - loss: 91.3564
Epoch 17/100
398/398 - 0s - loss: 91.3151
Epoch 18/100
398/398 - 0s - loss: 90.8586
Epoch 19/100
398/398 - 0s - loss: 90.5491
Epoch 20/100
398/398 - 0s - loss: 90.2820
Epoch 21/100
398/398 - 0s - loss: 90.0387
Epoch 22/100
398/398 - 0s - loss: 89.7985
Epoch 23/100
398/398 - 0s - loss: 89.4040
Epoch 24/100
398/398 - 0s - loss: 89.2273
```

```
Epoch 25/100
398/398 - 0s - loss: 88.7804
Epoch 26/100
398/398 - 0s - loss: 88.6813
Epoch 27/100
398/398 - 0s - loss: 88.3879
Epoch 28/100
398/398 - 0s - loss: 87.8715
Epoch 29/100
398/398 - 0s - loss: 87.7440
Epoch 30/100
398/398 - 0s - loss: 87.4421
Epoch 31/100
398/398 - 0s - loss: 87.1539
Epoch 32/100
398/398 - 0s - loss: 86.5423
Epoch 33/100
398/398 - 0s - loss: 86.2999
Epoch 34/100
398/398 - 0s - loss: 86.1143
Epoch 35/100
398/398 - 0s - loss: 85.3741
Epoch 36/100
398/398 - 0s - loss: 85.2060
Epoch 37/100
398/398 - 0s - loss: 84.9943
Epoch 38/100
398/398 - 0s - loss: 84.5882
Epoch 39/100
398/398 - 0s - loss: 84.1278
Epoch 40/100
398/398 - 0s - loss: 83.5848
Epoch 41/100
398/398 - 0s - loss: 83.5920
Epoch 42/100
398/398 - 0s - loss: 82.8033
Epoch 43/100
398/398 - 0s - loss: 82.6984
Epoch 44/100
398/398 - 0s - loss: 82.3113
Epoch 45/100
398/398 - 0s - loss: 81.8689
Epoch 46/100
398/398 - 0s - loss: 81.4020
Epoch 47/100
398/398 - 0s - loss: 81.1136
Epoch 48/100
398/398 - 0s - loss: 80.5552
Epoch 49/100
398/398 - 0s - loss: 80.1284
Epoch 50/100
398/398 - 0s - loss: 79.8937
Epoch 51/100
398/398 - 0s - loss: 79.3550
Epoch 52/100
398/398 - 0s - loss: 78.9173
Epoch 53/100
398/398 - 0s - loss: 78.6873
Epoch 54/100
398/398 - 0s - loss: 79.1422
Epoch 55/100
398/398 - 0s - loss: 77.8925
Epoch 56/100
398/398 - 0s - loss: 77.4074
Epoch 57/100
```

```
398/398 - 0s - loss: 76.8481
Epoch 58/100
398/398 - 0s - loss: 76.5019
Epoch 59/100
398/398 - 0s - loss: 76.0518
Epoch 60/100
398/398 - 0s - loss: 75.8869
Epoch 61/100
398/398 - 0s - loss: 75.5191
Epoch 62/100
398/398 - 0s - loss: 74.7774
Epoch 63/100
398/398 - 0s - loss: 74.4661
Epoch 64/100
398/398 - 0s - loss: 74.1682
Epoch 65/100
398/398 - 0s - loss: 73.7136
Epoch 66/100
398/398 - 0s - loss: 73.0588
Epoch 67/100
398/398 - 0s - loss: 73.2522
Epoch 68/100
398/398 - 0s - loss: 72.2606
Epoch 69/100
398/398 - 0s - loss: 71.9769
Epoch 70/100
398/398 - 0s - loss: 72.1472
Epoch 71/100
398/398 - 0s - loss: 70.9268
Epoch 72/100
398/398 - 0s - loss: 70.8533
Epoch 73/100
398/398 - 0s - loss: 70.8924
Epoch 74/100
398/398 - 0s - loss: 70.3384
Epoch 75/100
398/398 - 0s - loss: 69.4024
Epoch 76/100
398/398 - 0s - loss: 68.8651
Epoch 77/100
398/398 - 0s - loss: 68.4875
Epoch 78/100
398/398 - 0s - loss: 68.2619
Epoch 79/100
398/398 - 0s - loss: 67.4107
Epoch 80/100
398/398 - 0s - loss: 67.2279
Epoch 81/100
398/398 - 0s - loss: 66.7233
Epoch 82/100
398/398 - 0s - loss: 66.4764
Epoch 83/100
398/398 - 0s - loss: 65.9702
Epoch 84/100
398/398 - 0s - loss: 66.0415
Epoch 85/100
398/398 - 0s - loss: 64.8395
Epoch 86/100
398/398 - 0s - loss: 64.9977
Epoch 87/100
398/398 - 0s - loss: 64.3088
Epoch 88/100
398/398 - 0s - loss: 63.7484
Epoch 89/100
398/398 - 0s - loss: 63.7661
```

```
Epoch 90/100
398/398 - 0s - loss: 63.4873
Epoch 91/100
398/398 - 0s - loss: 62.5178
Epoch 92/100
398/398 - 0s - loss: 62.2690
Epoch 93/100
398/398 - 0s - loss: 61.8146
Epoch 94/100
398/398 - 0s - loss: 62.0988
Epoch 95/100
398/398 - 0s - loss: 62.0469
Epoch 96/100
398/398 - 0s - loss: 60.7122
Epoch 97/100
398/398 - 0s - loss: 60.0722
Epoch 98/100
398/398 - 0s - loss: 59.7114
Epoch 99/100
398/398 - 0s - loss: 59.8432
Epoch 100/100
398/398 - 0s - loss: 59.0609
```

Out[10]: `<tensorflow.python.keras.callbacks.History at 0x24fe53f1488>`

Monitor the loss at each epoch

One line is produced 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 Juputer 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)

Use Trained Model to Make Regression Prediction

Next we will perform actual predictions. These predictions are assigned to the **pred** variable. These are all MPG predictions from the neural network.

Notice that the data to predict should be a 2D array!

Notice that the prediction result is also a 2D array!

Neural networks can return multiple values, so the result is always an array. Here the neural network only returns 1 value per prediction (there are 398 cars, so 398 predictions). However, a 2D array is needed because the neural network has the potential of returning more than one value.

In [11]:
```
pred = model.predict(x)
print("Shape: {}".format(pred.shape))
print(pred)
```

```
Shape: (398, 1)
[[16.663313]
 [19.3972  ]
 [18.922789]
```

[20.213316]
[18.547926]
[19.636305]
[21.223555]
[21.416792]
[22.258406]
[20.346022]
[16.993757]
[18.860365]
[12.481184]
[19.245634]
[25.240095]
[18.842756]
[18.969526]
[16.440336]
[24.809385]
[17.247107]
[24.801542]
[24.739271]
[26.028471]
[26.873915]
[17.428816]
[29.49037]
[31.002302]
[31.65312]
[31.193464]
[24.974562]
[21.884108]
[24.983686]
[25.938261]
[16.265944]
[19.784208]
[16.531263]
[14.39897]
[17.981398]
[20.77223]
[18.739027]
[18.637617]
[20.76814]
[22.16368]
[18.559217]
[20.308882]
[16.551115]
[19.442305]
[16.376722]
[13.934645]
[22.490599]
[23.393644]
[23.537992]
[23.294401]
[22.867834]
[23.015112]
[19.680502]
[22.422895]
[25.407728]
[23.800596]
[20.194029]
[22.763594]
[22.913206]
[21.083662]
[18.775343]
[21.069868]
[18.697737]
[21.032515]
[22.447584]

[20.016737]
[20.776411]
[21.496778]
[29.384245]
[21.625196]
[18.525373]
[21.316298]
[20.988142]
[28.799574]
[21.976927]
[25.160978]
[22.186201]
[23.244703]
[26.429731]
[25.568586]
[23.64807]
[25.267393]
[22.681196]
[21.197691]
[17.250319]
[20.250227]
[20.331953]
[21.652967]
[14.802443]
[20.252546]
[21.562178]
[23.108137]
[24.035048]
[21.001387]
[19.496565]
[16.990828]
[17.588774]
[14.194609]
[19.546812]
[18.050184]
[16.196821]
[18.780409]
[22.207716]
[24.393324]
[17.140226]
[26.079329]
[19.805677]
[26.153532]
[27.877531]
[23.28993]
[24.135786]
[25.732298]
[17.461805]
[27.950422]
[20.719938]
[21.359694]
[25.100798]
[28.91609]
[19.335066]
[27.956465]
[27.494764]
[22.368162]
[20.206297]
[19.314024]
[17.655027]
[17.188948]
[23.4214]
[22.714064]
[23.70033]
[20.562056]

[18.190199]
[18.993345]
[20.768797]
[21.107191]
[20.024824]
[22.124739]
[22.420698]
[23.069563]
[24.683134]
[22.875547]
[23.89279]
[20.167768]
[22.162092]
[23.664274]
[23.640303]
[21.576475]
[25.909483]
[26.07556]
[23.007597]
[18.37364]
[18.396421]
[13.126783]
[12.363935]
[18.995619]
[18.695295]
[22.47975]
[19.567   ]
[22.38365]
[19.625929]
[19.476637]
[19.838636]
[19.84509]
[17.402174]
[16.991255]
[23.629698]
[22.326551]
[17.849455]
[21.504011]
[25.035261]
[23.471266]
[26.481535]
[22.56967]
[22.411177]
[16.88893]
[26.385542]
[25.535612]
[26.905302]
[29.125929]
[19.646315]
[25.21385]
[23.167128]
[23.689695]
[24.604399]
[24.601072]
[21.249434]
[21.805586]
[17.401075]
[19.359816]
[19.273588]
[18.176771]
[17.85333]
[16.82916]
[21.001493]
[21.549055]
[22.595854]

[19.801786]
[20.438612]
[14.647245]
[19.862867]
[15.726942]
[21.72148]
[23.650343]
[23.79791]
[20.09448]
[27.568205]
[21.262587]
[26.883268]
[26.045424]
[28.974836]
[24.582045]
[17.799055]
[19.19911]
[20.92436]
[22.626402]
[23.320234]
[21.809408]
[25.53433]
[23.694754]
[21.475233]
[20.328413]
[21.063368]
[20.310804]
[19.766188]
[20.366224]
[20.556532]
[17.959684]
[20.005262]
[22.468916]
[22.075178]
[19.442976]
[23.57833]
[22.707151]
[24.38626]
[23.836576]
[21.480146]
[24.879438]
[22.199553]
[24.10179]
[24.812208]
[28.365065]
[32.474987]
[19.948744]
[21.34477]
[21.69615]
[24.316832]
[21.254202]
[18.588285]
[19.405287]
[20.193449]
[21.003355]
[20.980787]
[18.592756]
[23.711332]
[20.222898]
[17.401762]
[20.322798]
[18.916853]
[22.523085]
[20.597898]
[20.68498]

[30.59271]
[19.216063]
[20.230452]
[22.693708]
[25.106977]
[26.201384]
[23.275862]
[25.064268]
[25.458189]
[22.769331]
[26.457136]
[27.260176]
[28.764265]
[30.11511]
[30.967512]
[23.372114]
[22.808699]
[21.871504]
[19.048399]
[24.444471]
[17.788313]
[21.89836]
[19.409163]
[19.039564]
[16.89253]
[19.116285]
[20.990156]
[17.91316]
[21.234312]
[18.139631]
[23.303213]
[22.987654]
[24.185507]
[23.971998]
[20.465986]
[15.020094]
[22.747648]
[16.078444]
[22.383558]
[22.436964]
[23.564909]
[23.316114]
[23.225294]
[24.978834]
[25.371214]
[22.695463]
[24.121397]
[22.368177]
[23.192976]
[23.368925]
[23.457197]
[24.642653]
[24.539808]
[18.863066]
[24.758696]
[25.0408]
[23.465925]
[25.999327]
[23.646452]
[23.720045]
[25.687834]
[23.932585]
[20.523603]
[21.28131]
[22.893034]

[21.63606]
[22.65174]
[27.814856]
[23.21124]
[21.861547]
[29.375883]
[31.328474]
[24.976469]
[25.300077]
[23.446165]
[23.456556]
[22.48007]
[23.20381]
[24.865355]
[22.94708]
[22.380278]
[22.88722]
[22.487364]
[23.176908]
[23.813536]
[22.916716]
[23.507978]
[21.705137]
[22.39459]
[23.587507]
[23.396193]
[27.639555]
[23.69554]
[24.159025]
[27.933302]
[23.745756]
[24.29228]
[22.973783]
[26.873594]
[29.633848]
[21.760206]
[11.660292]
[19.946821]
[18.149137]
[26.922483]
[26.903624]
[26.29376]
[26.005278]
[23.72978]
[24.062223]
[25.486448]
[25.585417]
[23.196241]
[23.977873]
[23.795576]
[21.648314]
[23.71028]
[24.888182]
[23.85922]
[23.234587]
[23.35797]
[23.426268]
[23.542555]
[25.21089]
[14.170393]
[23.29997]
[20.772612]
[25.29048]
[23.088255]
[24.477644]

```
[24.203825]
[21.328932]
[22.78288 ]
[24.63066 ]
[25.52609 ]]
```

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

In [12]:
```
# Measure RMSE error.  RMSE is common for regression.
score = np.sqrt(metrics.mean_squared_error(pred,y))
print("Final score (RMSE): {}".format(score))
```

```
Final score (RMSE): 7.6590576171875
```

We can also print out the first 10 cars, with predictions and actual MPG.

In [13]:
```
# Sample predictions
for i in range(10):
    print("{}. Car name: {}, MPG: {}, predicted MPG: {}".format(i+1,cars[i],y[i],pred[i
```

```
1. Car name: chevrolet chevelle malibu, MPG: 18.0, predicted MPG: [16.663313]
2. Car name: buick skylark 320, MPG: 15.0, predicted MPG: [19.3972]
3. Car name: plymouth satellite, MPG: 18.0, predicted MPG: [18.922789]
4. Car name: amc rebel sst, MPG: 16.0, predicted MPG: [20.213316]
5. Car name: ford torino, MPG: 17.0, predicted MPG: [18.547926]
6. Car name: ford galaxie 500, MPG: 15.0, predicted MPG: [19.636305]
7. Car name: chevrolet impala, MPG: 14.0, predicted MPG: [21.223555]
8. Car name: plymouth fury iii, MPG: 14.0, predicted MPG: [21.416792]
9. Car name: pontiac catalina, MPG: 14.0, predicted MPG: [22.258406]
10. Car name: amc ambassador dpl, MPG: 15.0, predicted MPG: [20.346022]
```

Example of TensorFlow Classification: Iris

This is a very simple example of how to perform the Iris classification using TensorFlow. The iris.csv file is used.

In [14]:
```
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("data/iris.csv",na_values=['NA','?'])

species = encode_text_index(df,"species")

x,y = to_xy(df,"species")
```

In [15]:
```
x
```

Out[15]:
```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       [4.6, 3.1, 1.5, 0.2],
       [5. , 3.6, 1.4, 0.2],
       [5.4, 3.9, 1.7, 0.4],
       [4.6, 3.4, 1.4, 0.3],
       [5. , 3.4, 1.5, 0.2],
       [4.4, 2.9, 1.4, 0.2],
       [4.9, 3.1, 1.5, 0.1],
       [5.4, 3.7, 1.5, 0.2],
       [4.8, 3.4, 1.6, 0.2],
       [4.8, 3. , 1.4, 0.1],
       [4.3, 3. , 1.1, 0.1],
       [5.8, 4. , 1.2, 0.2],
       [5.7, 4.4, 1.5, 0.4],
       [5.4, 3.9, 1.3, 0.4],
       [5.1, 3.5, 1.4, 0.3],
       [5.7, 3.8, 1.7, 0.3],
       [5.1, 3.8, 1.5, 0.3],
       [5.4, 3.4, 1.7, 0.2],
       [5.1, 3.7, 1.5, 0.4],
       [4.6, 3.6, 1. , 0.2],
       [5.1, 3.3, 1.7, 0.5],
       [4.8, 3.4, 1.9, 0.2],
       [5. , 3. , 1.6, 0.2],
       [5. , 3.4, 1.6, 0.4],
       [5.2, 3.5, 1.5, 0.2],
       [5.2, 3.4, 1.4, 0.2],
       [4.7, 3.2, 1.6, 0.2],
       [4.8, 3.1, 1.6, 0.2],
       [5.4, 3.4, 1.5, 0.4],
       [5.2, 4.1, 1.5, 0.1],
       [5.5, 4.2, 1.4, 0.2],
       [4.9, 3.1, 1.5, 0.2],
       [5. , 3.2, 1.2, 0.2],
       [5.5, 3.5, 1.3, 0.2],
       [4.9, 3.6, 1.4, 0.1],
       [4.4, 3. , 1.3, 0.2],
       [5.1, 3.4, 1.5, 0.2],
       [5. , 3.5, 1.3, 0.3],
       [4.5, 2.3, 1.3, 0.3],
       [4.4, 3.2, 1.3, 0.2],
       [5. , 3.5, 1.6, 0.6],
       [5.1, 3.8, 1.9, 0.4],
       [4.8, 3. , 1.4, 0.3],
       [5.1, 3.8, 1.6, 0.2],
       [4.6, 3.2, 1.4, 0.2],
       [5.3, 3.7, 1.5, 0.2],
       [5. , 3.3, 1.4, 0.2],
       [7. , 3.2, 4.7, 1.4],
       [6.4, 3.2, 4.5, 1.5],
       [6.9, 3.1, 4.9, 1.5],
       [5.5, 2.3, 4. , 1.3],
       [6.5, 2.8, 4.6, 1.5],
       [5.7, 2.8, 4.5, 1.3],
       [6.3, 3.3, 4.7, 1.6],
       [4.9, 2.4, 3.3, 1. ],
       [6.6, 2.9, 4.6, 1.3],
       [5.2, 2.7, 3.9, 1.4],
       [5. , 2. , 3.5, 1. ],
       [5.9, 3. , 4.2, 1.5],
       [6. , 2.2, 4. , 1. ],
       [6.1, 2.9, 4.7, 1.4],
       [5.6, 2.9, 3.6, 1.3],
```

```
[6.7, 3.1, 4.4, 1.4],
[5.6, 3. , 4.5, 1.5],
[5.8, 2.7, 4.1, 1. ],
[6.2, 2.2, 4.5, 1.5],
[5.6, 2.5, 3.9, 1.1],
[5.9, 3.2, 4.8, 1.8],
[6.1, 2.8, 4. , 1.3],
[6.3, 2.5, 4.9, 1.5],
[6.1, 2.8, 4.7, 1.2],
[6.4, 2.9, 4.3, 1.3],
[6.6, 3. , 4.4, 1.4],
[6.8, 2.8, 4.8, 1.4],
[6.7, 3. , 5. , 1.7],
[6. , 2.9, 4.5, 1.5],
[5.7, 2.6, 3.5, 1. ],
[5.5, 2.4, 3.8, 1.1],
[5.5, 2.4, 3.7, 1. ],
[5.8, 2.7, 3.9, 1.2],
[6. , 2.7, 5.1, 1.6],
[5.4, 3. , 4.5, 1.5],
[6. , 3.4, 4.5, 1.6],
[6.7, 3.1, 4.7, 1.5],
[6.3, 2.3, 4.4, 1.3],
[5.6, 3. , 4.1, 1.3],
[5.5, 2.5, 4. , 1.3],
[5.5, 2.6, 4.4, 1.2],
[6.1, 3. , 4.6, 1.4],
[5.8, 2.6, 4. , 1.2],
[5. , 2.3, 3.3, 1. ],
[5.6, 2.7, 4.2, 1.3],
[5.7, 3. , 4.2, 1.2],
[5.7, 2.9, 4.2, 1.3],
[6.2, 2.9, 4.3, 1.3],
[5.1, 2.5, 3. , 1.1],
[5.7, 2.8, 4.1, 1.3],
[6.3, 3.3, 6. , 2.5],
[5.8, 2.7, 5.1, 1.9],
[7.1, 3. , 5.9, 2.1],
[6.3, 2.9, 5.6, 1.8],
[6.5, 3. , 5.8, 2.2],
[7.6, 3. , 6.6, 2.1],
[4.9, 2.5, 4.5, 1.7],
[7.3, 2.9, 6.3, 1.8],
[6.7, 2.5, 5.8, 1.8],
[7.2, 3.6, 6.1, 2.5],
[6.5, 3.2, 5.1, 2. ],
[6.4, 2.7, 5.3, 1.9],
[6.8, 3. , 5.5, 2.1],
[5.7, 2.5, 5. , 2. ],
[5.8, 2.8, 5.1, 2.4],
[6.4, 3.2, 5.3, 2.3],
[6.5, 3. , 5.5, 1.8],
[7.7, 3.8, 6.7, 2.2],
[7.7, 2.6, 6.9, 2.3],
[6. , 2.2, 5. , 1.5],
[6.9, 3.2, 5.7, 2.3],
[5.6, 2.8, 4.9, 2. ],
[7.7, 2.8, 6.7, 2. ],
[6.3, 2.7, 4.9, 1.8],
[6.7, 3.3, 5.7, 2.1],
[7.2, 3.2, 6. , 1.8],
[6.2, 2.8, 4.8, 1.8],
[6.1, 3. , 4.9, 1.8],
[6.4, 2.8, 5.6, 2.1],
[7.2, 3. , 5.8, 1.6],
```

```
       [7.4, 2.8, 6.1, 1.9],
       [7.9, 3.8, 6.4, 2. ],
       [6.4, 2.8, 5.6, 2.2],
       [6.3, 2.8, 5.1, 1.5],
       [6.1, 2.6, 5.6, 1.4],
       [7.7, 3. , 6.1, 2.3],
       [6.3, 3.4, 5.6, 2.4],
       [6.4, 3.1, 5.5, 1.8],
       [6. , 3. , 4.8, 1.8],
       [6.9, 3.1, 5.4, 2.1],
       [6.7, 3.1, 5.6, 2.4],
       [6.9, 3.1, 5.1, 2.3],
       [5.8, 2.7, 5.1, 1.9],
       [6.8, 3.2, 5.9, 2.3],
       [6.7, 3.3, 5.7, 2.5],
       [6.7, 3. , 5.2, 2.3],
       [6.3, 2.5, 5. , 1.9],
       [6.5, 3. , 5.2, 2. ],
       [6.2, 3.4, 5.4, 2.3],
       [5.9, 3. , 5.1, 1.8]], dtype=float32)
```

In [16]:
```
x.shape
```

Out[16]:
```
(150, 4)
```

In [17]:
```
y  #  This is one-hot encoding.  Only one value is 1.0 (hot)
```

Out[17]:
```
array([[1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
```

```
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
[0., 1., 0.],
```

```
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.]], dtype=float32)
```

In [18]:

```python
y.shape
```

Out[18]:

```
(150, 3)
```

In [19]:

```python
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: 1.8213
Epoch 2/100
150/150 - 0s - loss: 1.3406
Epoch 3/100
150/150 - 0s - loss: 1.0830
Epoch 4/100
150/150 - 0s - loss: 0.9423
Epoch 5/100
150/150 - 0s - loss: 0.8870
Epoch 6/100
150/150 - 0s - loss: 0.8404
Epoch 7/100
150/150 - 0s - loss: 0.7796
Epoch 8/100
150/150 - 0s - loss: 0.7184
Epoch 9/100
150/150 - 0s - loss: 0.6771
Epoch 10/100
150/150 - 0s - loss: 0.6432
Epoch 11/100
150/150 - 0s - loss: 0.6097
Epoch 12/100
150/150 - 0s - loss: 0.5772
Epoch 13/100
150/150 - 0s - loss: 0.5509
Epoch 14/100
150/150 - 0s - loss: 0.5278
Epoch 15/100
150/150 - 0s - loss: 0.5065
Epoch 16/100
150/150 - 0s - loss: 0.4881
Epoch 17/100
150/150 - 0s - loss: 0.4723
Epoch 18/100
150/150 - 0s - loss: 0.4565
Epoch 19/100
150/150 - 0s - loss: 0.4455
Epoch 20/100
150/150 - 0s - loss: 0.4302
Epoch 21/100
150/150 - 0s - loss: 0.4202
Epoch 22/100
150/150 - 0s - loss: 0.4092
Epoch 23/100
150/150 - 0s - loss: 0.4030
Epoch 24/100
150/150 - 0s - loss: 0.3884
Epoch 25/100
150/150 - 0s - loss: 0.3799
Epoch 26/100
150/150 - 0s - loss: 0.3609
Epoch 27/100
150/150 - 0s - loss: 0.3445
Epoch 28/100
150/150 - 0s - loss: 0.3314
Epoch 29/100
150/150 - 0s - loss: 0.3293
Epoch 30/100
150/150 - 0s - loss: 0.3107
```

```
Epoch 31/100
150/150 - 0s - loss: 0.3053
Epoch 32/100
150/150 - 0s - loss: 0.2957
Epoch 33/100
150/150 - 0s - loss: 0.2844
Epoch 34/100
150/150 - 0s - loss: 0.2776
Epoch 35/100
150/150 - 0s - loss: 0.2682
Epoch 36/100
150/150 - 0s - loss: 0.2612
Epoch 37/100
150/150 - 0s - loss: 0.2530
Epoch 38/100
150/150 - 0s - loss: 0.2459
Epoch 39/100
150/150 - 0s - loss: 0.2404
Epoch 40/100
150/150 - 0s - loss: 0.2324
Epoch 41/100
150/150 - 0s - loss: 0.2259
Epoch 42/100
150/150 - 0s - loss: 0.2214
Epoch 43/100
150/150 - 0s - loss: 0.2124
Epoch 44/100
150/150 - 0s - loss: 0.2097
Epoch 45/100
150/150 - 0s - loss: 0.2041
Epoch 46/100
150/150 - 0s - loss: 0.1978
Epoch 47/100
150/150 - 0s - loss: 0.1908
Epoch 48/100
150/150 - 0s - loss: 0.1894
Epoch 49/100
150/150 - 0s - loss: 0.1826
Epoch 50/100
150/150 - 0s - loss: 0.1799
Epoch 51/100
150/150 - 0s - loss: 0.1751
Epoch 52/100
150/150 - 0s - loss: 0.1690
Epoch 53/100
150/150 - 0s - loss: 0.1660
Epoch 54/100
150/150 - 0s - loss: 0.1648
Epoch 55/100
150/150 - 0s - loss: 0.1582
Epoch 56/100
150/150 - 0s - loss: 0.1544
Epoch 57/100
150/150 - 0s - loss: 0.1519
Epoch 58/100
150/150 - 0s - loss: 0.1476
Epoch 59/100
150/150 - 0s - loss: 0.1452
Epoch 60/100
150/150 - 0s - loss: 0.1427
Epoch 61/100
150/150 - 0s - loss: 0.1399
Epoch 62/100
150/150 - 0s - loss: 0.1376
Epoch 63/100
```

```
150/150 - 0s - loss: 0.1389
Epoch 64/100
150/150 - 0s - loss: 0.1327
Epoch 65/100
150/150 - 0s - loss: 0.1309
Epoch 66/100
150/150 - 0s - loss: 0.1282
Epoch 67/100
150/150 - 0s - loss: 0.1249
Epoch 68/100
150/150 - 0s - loss: 0.1245
Epoch 69/100
150/150 - 0s - loss: 0.1211
Epoch 70/100
150/150 - 0s - loss: 0.1212
Epoch 71/100
150/150 - 0s - loss: 0.1236
Epoch 72/100
150/150 - 0s - loss: 0.1186
Epoch 73/100
150/150 - 0s - loss: 0.1154
Epoch 74/100
150/150 - 0s - loss: 0.1127
Epoch 75/100
150/150 - 0s - loss: 0.1111
Epoch 76/100
150/150 - 0s - loss: 0.1099
Epoch 77/100
150/150 - 0s - loss: 0.1086
Epoch 78/100
150/150 - 0s - loss: 0.1068
Epoch 79/100
150/150 - 0s - loss: 0.1071
Epoch 80/100
150/150 - 0s - loss: 0.1034
Epoch 81/100
150/150 - 0s - loss: 0.1045
Epoch 82/100
150/150 - 0s - loss: 0.1035
Epoch 83/100
150/150 - 0s - loss: 0.1007
Epoch 84/100
150/150 - 0s - loss: 0.1002
Epoch 85/100
150/150 - 0s - loss: 0.0983
Epoch 86/100
150/150 - 0s - loss: 0.1027
Epoch 87/100
150/150 - 0s - loss: 0.0999
Epoch 88/100
150/150 - 0s - loss: 0.1018
Epoch 89/100
150/150 - 0s - loss: 0.0948
Epoch 90/100
150/150 - 0s - loss: 0.0934
Epoch 91/100
150/150 - 0s - loss: 0.0924
Epoch 92/100
150/150 - 0s - loss: 0.0916
Epoch 93/100
150/150 - 0s - loss: 0.0927
Epoch 94/100
150/150 - 0s - loss: 0.0898
Epoch 95/100
150/150 - 0s - loss: 0.0941
```

```
Epoch 96/100
150/150 - 0s - loss: 0.0882
Epoch 97/100
150/150 - 0s - loss: 0.0912
Epoch 98/100
150/150 - 0s - loss: 0.0904
Epoch 99/100
150/150 - 0s - loss: 0.0916
Epoch 100/100
150/150 - 0s - loss: 0.0865
```

Out[19]:
```
<tensorflow.python.keras.callbacks.History at 0x24fe71ff6c8>
```

In [20]:
```
# Print out number of species found:
print(species)
```

```
['Iris-setosa' 'Iris-versicolor' 'Iris-virginica']
```

Now that you have a neural network trained, we would like to be able to use it. There were 3 types of iris (Iris-setosa, Iris-versicolor, and Iris-virginica).

In [21]:
```
pred = model.predict(x)
pred.shape
```

Out[21]:
```
(150, 3)
```

In [22]:
```
pred
```

Out[22]:
```
array([[9.98260200e-01, 1.73469016e-03, 5.14739895e-06],
       [9.96438146e-01, 3.54828546e-03, 1.35634045e-05],
       [9.97133374e-01, 2.85392953e-03, 1.27558314e-05],
       [9.95246112e-01, 4.73011006e-03, 2.36744927e-05],
       [9.98356879e-01, 1.63782563e-03, 5.27151951e-06],
       [9.98313665e-01, 1.68274331e-03, 3.62309220e-06],
       [9.96958494e-01, 3.02718696e-03, 1.44079695e-05],
       [9.97697532e-01, 2.29479023e-03, 7.61940692e-06],
       [9.94080603e-01, 5.88311953e-03, 3.62918108e-05],
       [9.96756852e-01, 3.23098409e-03, 1.21421126e-05],
       [9.98736799e-01, 1.26052368e-03, 2.76404307e-06],
       [9.96742785e-01, 3.24425683e-03, 1.29060109e-05],
       [9.96532321e-01, 3.45332595e-03, 1.43490151e-05],
       [9.96504307e-01, 3.47297196e-03, 2.27021665e-05],
       [9.99484777e-01, 5.14454092e-04, 7.31225327e-07],
       [9.99450624e-01, 5.48501732e-04, 8.42264740e-07],
       [9.99032974e-01, 9.65072424e-04, 2.08119218e-06],
       [9.98107553e-01, 1.88675476e-03, 5.65809978e-06],
       [9.98719931e-01, 1.27797120e-03, 2.12409350e-06],
       [9.98485506e-01, 1.51019706e-03, 4.32172646e-06],
       [9.97356415e-01, 2.63745128e-03, 6.15282715e-06],
       [9.98102844e-01, 1.89174386e-03, 5.48767002e-06],
       [9.98590052e-01, 1.40353083e-03, 6.42923305e-06],
       [9.91988719e-01, 7.98540004e-03, 2.58874152e-05],
       [9.90868211e-01, 9.09298938e-03, 3.88435292e-05],
       [9.94109511e-01, 5.86942025e-03, 2.11333754e-05],
       [9.95998621e-01, 3.98787158e-03, 1.34807869e-05],
       [9.98192251e-01, 1.80293282e-03, 4.89810600e-06],
       [9.98157680e-01, 1.83727650e-03, 5.02618559e-06],
       [9.94968474e-01, 5.00859600e-03, 2.28890531e-05],
       [9.94358957e-01, 5.61713940e-03, 2.39159417e-05],
       [9.97803628e-01, 2.19122716e-03, 5.09901110e-06],
```

```
[9.99182522e-01, 8.15384032e-04, 2.02836259e-06],
[9.99404430e-01, 5.94490557e-04, 1.10551184e-06],
[9.96308327e-01, 3.67772183e-03, 1.40324073e-05],
[9.97838557e-01, 2.15422828e-03, 7.21466722e-06],
[9.98765707e-01, 1.23187446e-03, 2.49273694e-06],
[9.98394310e-01, 1.60008192e-03, 5.59931732e-06],
[9.95614052e-01, 4.35943389e-03, 2.64584414e-05],
[9.97837484e-01, 2.15604599e-03, 6.51979099e-06],
[9.98178720e-01, 1.81534700e-03, 5.94609764e-06],
[9.84010041e-01, 1.58906374e-02, 9.94141228e-05],
[9.96508062e-01, 3.47147998e-03, 2.04602293e-05],
[9.95005190e-01, 4.97800577e-03, 1.68088773e-05],
[9.95190978e-01, 4.79451194e-03, 1.44904197e-05],
[9.95650113e-01, 4.33144253e-03, 1.84020446e-05],
[9.98462677e-01, 1.53298979e-03, 4.36743949e-06],
[9.96580422e-01, 3.40280565e-03, 1.66957143e-05],
[9.98657346e-01, 1.33947516e-03, 3.22732490e-06],
[9.97657537e-01, 2.33457191e-03, 7.81111248e-06],
[9.50098445e-04, 9.94067073e-01, 4.98284074e-03],
[1.17485574e-03, 9.82445538e-01, 1.63796507e-02],
[5.22727845e-04, 9.71911371e-01, 2.75658481e-02],
[1.61551381e-03, 8.86358082e-01, 1.12026408e-01],
[7.23056204e-04, 9.44195032e-01, 5.50819412e-02],
[9.21764586e-04, 8.66791606e-01, 1.32286623e-01],
[8.30960926e-04, 9.36852694e-01, 6.23163395e-02],
[1.15724532e-02, 9.75152731e-01, 1.32748727e-02],
[8.59280815e-04, 9.87078190e-01, 1.20625561e-02],
[2.52606231e-03, 9.00780857e-01, 9.66930613e-02],
[3.96730518e-03, 9.63739455e-01, 3.22932266e-02],
[1.77378627e-03, 9.64507818e-01, 3.37183475e-02],
[1.92832865e-03, 9.87842083e-01, 1.02295969e-02],
[6.56856690e-04, 8.78738046e-01, 1.20605059e-01],
[9.67946462e-03, 9.81514513e-01, 8.80599394e-03],
[1.71202014e-03, 9.93175805e-01, 5.11222566e-03],
[8.66693095e-04, 7.59369731e-01, 2.39763618e-01],
[2.53432710e-03, 9.89483297e-01, 7.98239093e-03],
[4.26318235e-04, 6.70375288e-01, 3.29198331e-01],
[2.63097393e-03, 9.84042943e-01, 1.33261206e-02],
[3.08988470e-04, 4.54399139e-01, 5.45291901e-01],
[3.02226236e-03, 9.89500821e-01, 7.47689418e-03],
[2.04782671e-04, 5.25585055e-01, 4.74210113e-01],
[7.04103499e-04, 9.35994506e-01, 6.33014217e-02],
[1.67608052e-03, 9.91080940e-01, 7.24301999e-03],
[1.39352528e-03, 9.91295934e-01, 7.31047988e-03],
[5.28855075e-04, 9.65835750e-01, 3.36353481e-02],
[3.14703648e-04, 7.79807091e-01, 2.19878212e-01],
[9.07314650e-04, 8.86748731e-01, 1.12343989e-01],
[1.80195998e-02, 9.76381361e-01, 5.59896231e-03],
[2.92228744e-03, 9.82019961e-01, 1.50577463e-02],
[4.66023758e-03, 9.86074507e-01, 9.26528126e-03],
[3.41135962e-03, 9.87655997e-01, 8.93265381e-03],
[5.27733282e-05, 1.68224201e-01, 8.31723094e-01],
[7.40792835e-04, 6.35263860e-01, 3.63995343e-01],
[1.30550412e-03, 9.49788272e-01, 4.89061959e-02],
[7.54833804e-04, 9.78178024e-01, 2.10672207e-02],
[7.94588239e-04, 9.36850607e-01, 6.23547658e-02],
[2.36855820e-03, 9.77021158e-01, 2.06103120e-02],
[1.93685514e-03, 9.30698335e-01, 6.73648119e-02],
[9.80837969e-04, 8.38783026e-01, 1.60236165e-01],
[8.86148366e-04, 9.36213315e-01, 6.29005060e-02],
[2.19786214e-03, 9.84334171e-01, 1.34679154e-02],
[1.06805982e-02, 9.76889849e-01, 1.24295522e-02],
[1.55691046e-03, 9.30051208e-01, 6.83919117e-02],
[2.09527276e-03, 9.82824564e-01, 1.50801539e-02],
[1.83026621e-03, 9.67911720e-01, 3.02580688e-02],
```

```
       [1.49372208e-03, 9.87746358e-01, 1.07598454e-02],
       [4.25622985e-02, 9.48486090e-01, 8.95163603e-03],
       [2.03832029e-03, 9.71988440e-01, 2.59732567e-02],
       [4.33587104e-08, 6.15980593e-04, 9.99384046e-01],
       [7.07607160e-06, 2.47481111e-02, 9.75244820e-01],
       [2.26994007e-06, 3.12042739e-02, 9.68793452e-01],
       [4.90392131e-06, 3.51903886e-02, 9.64804649e-01],
       [4.47571267e-07, 5.06165996e-03, 9.94937897e-01],
       [1.50886848e-07, 6.73677726e-03, 9.93263185e-01],
       [2.88545798e-05, 3.51283439e-02, 9.64842856e-01],
       [1.32839125e-06, 3.29444148e-02, 9.67054188e-01],
       [1.26461646e-06, 1.62417293e-02, 9.83757019e-01],
       [8.72589681e-07, 1.32627739e-02, 9.86736357e-01],
       [1.19166762e-04, 3.34477723e-01, 6.65403128e-01],
       [1.14189197e-05, 5.96750155e-02, 9.40313578e-01],
       [1.03391894e-05, 6.99779391e-02, 9.30011749e-01],
       [2.92582445e-06, 1.00819869e-02, 9.89915133e-01],
       [7.23422204e-07, 2.83929543e-03, 9.97159958e-01],
       [7.89584738e-06, 3.43334079e-02, 9.65658665e-01],
       [2.02600604e-05, 1.19069695e-01, 8.80910099e-01],
       [9.29275416e-07, 3.24153267e-02, 9.67583776e-01],
       [3.16069637e-09, 3.08782270e-04, 9.99691248e-01],
       [3.31665942e-05, 1.12891093e-01, 8.87075722e-01],
       [2.97032602e-06, 2.68753897e-02, 9.73121643e-01],
       [1.10998162e-05, 2.62579415e-02, 9.73730922e-01],
       [9.75333094e-08, 5.51745109e-03, 9.94482398e-01],
       [1.20666853e-04, 3.05801004e-01, 6.94078326e-01],
       [6.63945957e-06, 5.11699095e-02, 9.48823452e-01],
       [1.36910348e-05, 1.77108690e-01, 8.22877586e-01],
       [2.02242329e-04, 4.00439978e-01, 5.99357784e-01],
       [1.76583533e-04, 3.58996093e-01, 6.40827239e-01],
       [9.65829486e-07, 8.39483831e-03, 9.91604269e-01],
       [4.58725081e-05, 4.41397816e-01, 5.58556259e-01],
       [2.40130362e-06, 4.82210554e-02, 9.51776505e-01],
       [1.88739541e-05, 3.75902355e-01, 6.24078751e-01],
       [5.67337338e-07, 5.14246151e-03, 9.94856954e-01],
       [1.73028020e-04, 5.16783357e-01, 4.83043700e-01],
       [8.49797016e-06, 5.90336546e-02, 9.40957844e-01],
       [1.31617935e-06, 2.91595776e-02, 9.70839083e-01],
       [1.16279614e-06, 7.55999424e-03, 9.92438853e-01],
       [2.15141754e-05, 1.17064789e-01, 8.82913709e-01],
       [2.30502075e-04, 3.91220778e-01, 6.08548760e-01],
       [3.17498234e-05, 1.71538502e-01, 8.28429818e-01],
       [1.29883756e-06, 1.07550882e-02, 9.89243627e-01],
       [6.35684846e-05, 2.21935704e-01, 7.78000832e-01],
       [7.07607160e-06, 2.47481111e-02, 9.75244820e-01],
       [6.76651382e-07, 8.50558374e-03, 9.91493702e-01],
       [8.35202115e-07, 7.44327111e-03, 9.92555916e-01],
       [1.65131914e-05, 7.14505836e-02, 9.28532958e-01],
       [2.54885326e-05, 9.06321108e-02, 9.09342408e-01],
       [4.13337584e-05, 1.56975016e-01, 8.42983723e-01],
       [4.90172306e-06, 2.16508899e-02, 9.78344202e-01],
       [4.09383792e-05, 1.14391126e-01, 8.85567904e-01]], dtype=float32)
```

In [23]:

```
# print y.
print(y)
```

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

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

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

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

The column (pred) with the highest probability is the prediction of the neural network.

Use argmax function to find the index of the maximum prediction for each row.

In [24]:
```
# Usually the column (pred) with the highest prediction is considered to be the predict
# to convert the predictions to the expected iris species.  The argmax function finds t
# for each row.

predict_classes = np.argmax(pred,axis=1)

true_classes = np.argmax(y,axis=1)

print("Predictions: {}".format(predict_classes))
print("True: {}".format(true_classes))
```

```
Predictions: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1
 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
True: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

In [25]:
```
# Of course it is very easy to turn these indexes back into iris species.  We just use
print(species[predict_classes[0:10]])
```

```
['Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa'
 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa']
```

In [26]:
```
#For all of the iris predictions, what percent were correct?

correct = metrics.accuracy_score(true_classes, predict_classes)
print("Accuracy: {}".format(correct))
```

```
Accuracy: 0.98
```

The code below performs two ad hoc predictions.

Remember x should be a 2D array!

In [27]:
```
# ad hoc prediction
```

```
sample_flower = np.array( [[5.0,3.0,4.0,2.0]], dtype=float)
pred = model.predict(sample_flower)
print(pred)
```

```
[[6.7408831e-04 2.8414682e-01 7.1517903e-01]]
```

In [22]:

```
pred = np.argmax(pred, axis=1)
print("Predict that {} is: {}".format(sample_flower,species[pred]))
```

```
Predict that [[5. 3. 4. 2.]] is: ['Iris-virginica']
```

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 [23]:

```
# predict two sample flowers
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)
```

```
[[2.4080558e-03 3.3512670e-01 6.6246527e-01]
 [9.8348677e-01 1.6512996e-02 2.3293094e-07]]
```

In [24]:

```
pred = np.argmax(pred, axis=1)
print("Predict that {} is: {}".format(sample_flower,species[pred]))
```

```
Predict that [[5.  3.  4.  2. ]
 [5.2 3.5 1.5 0.8]] is: ['Iris-virginica' 'Iris-setosa']
```

Load/Save Trained 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 they can be reloaded later. A reloaded neural network will not require retraining. Keras provides three formats for neural network saving.

- **YAML** - Stores the neural network structure (no weights) in the YAML file format.
- **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.

Usually you will want to save in HDF5.

In [25]:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
import pandas as pd
import io
import requests
import numpy as np
from sklearn import metrics
path = "./data/"
save_path = "./dnn/"

filename_read = os.path.join(path,"auto-mpg.csv")
df = pd.read_csv(filename_read,na_values=['NA','?'])
```

```
cars = df['name']
df.drop('name',1,inplace=True)
missing_median(df, 'horsepower')
x,y = to_xy(df,"mpg")
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(1)) # Output
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(x,y,verbose=0,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("Before save score (RMSE): {}".format(score))


# save entire network to HDF5 (save everything)
model.save(os.path.join(save_path,"network.hdf5"))
```

```
Before save score (RMSE): 6.450832843780518
```

Now we reload the network and perform another prediction. The RMSE should match the previous one exactly if the neural network was really saved and reloaded.

In [26]:

```
from tensorflow.keras.models import load_model

model2 = load_model(os.path.join(save_path,"network.hdf5"))
pred = model2.predict(x)
# Measure RMSE error.  RMSE is common for regression.
score = np.sqrt(metrics.mean_squared_error(pred,y))
print("After load score (RMSE): {}".format(score))
```

```
W0913 14:31:15.962442 14488 deprecation.py:506] From C:\ProgramData\Anaconda3\lib\site-p
ackages\tensorflow\python\ops\init_ops.py:97: calling GlorotUniform.__init__ (from tenso
rflow.python.ops.init_ops) with dtype is deprecated and will be removed in a future vers
ion.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the construct
or
W0913 14:31:15.963423 14488 deprecation.py:506] From C:\ProgramData\Anaconda3\lib\site-p
ackages\tensorflow\python\ops\init_ops.py:97: calling Zeros.__init__ (from tensorflow.py
thon.ops.init_ops) with dtype is deprecated and will be removed in a future version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the construct
or
After load score (RMSE): 6.450832843780518
```

References:

- Google Colab - Free web based platform that includes Python, Juypter Notebooks, and TensorFlow with free GPU support. No setup needed.
- IBM Cognitive Class Labs - Free web based platform that includes Python, Juypter Notebooks, and TensorFlow. No setup needed.

- Python Anaconda - Python distribution that includes many data science packages, such as Numpy, Scipy, Scikit-Learn, Pandas, and much more.
- TensorFlow - Google's mathematics package for deep learning.
- Kaggle - Competitive data science. Good source of sample data.
- T81-558: Applications of Deep Neural Networks. Instructor: Jeff Heaton

CSC 180 Intelligenct Systems (Fall 2022)

Dr. Haiquan Chen, Dept of Computer Scicence

California State University, Sacramento

Lab 6: Evaluating Neural Networks

Helpful Functions for Tensorflow (little gems)

The following functions will be used with TensorFlow to help preprocess the data. They allow you to build the feature vector for a neural network.

- Predictors/Inputs
  - Fill any missing inputs with the median for that column. Use **missing_median**.
  - Encode textual/categorical values with **encode_text_dummy**.
  - Encode numeric values with **encode_numeric_zscore**.
- Output
  - Discard rows with missing outputs.
  - Encode textual/categorical values with **encode_text_index**.
  - Do not encode output numeric values.
- Produce final feature vectors (x) and expected output (y) with **to_xy**.

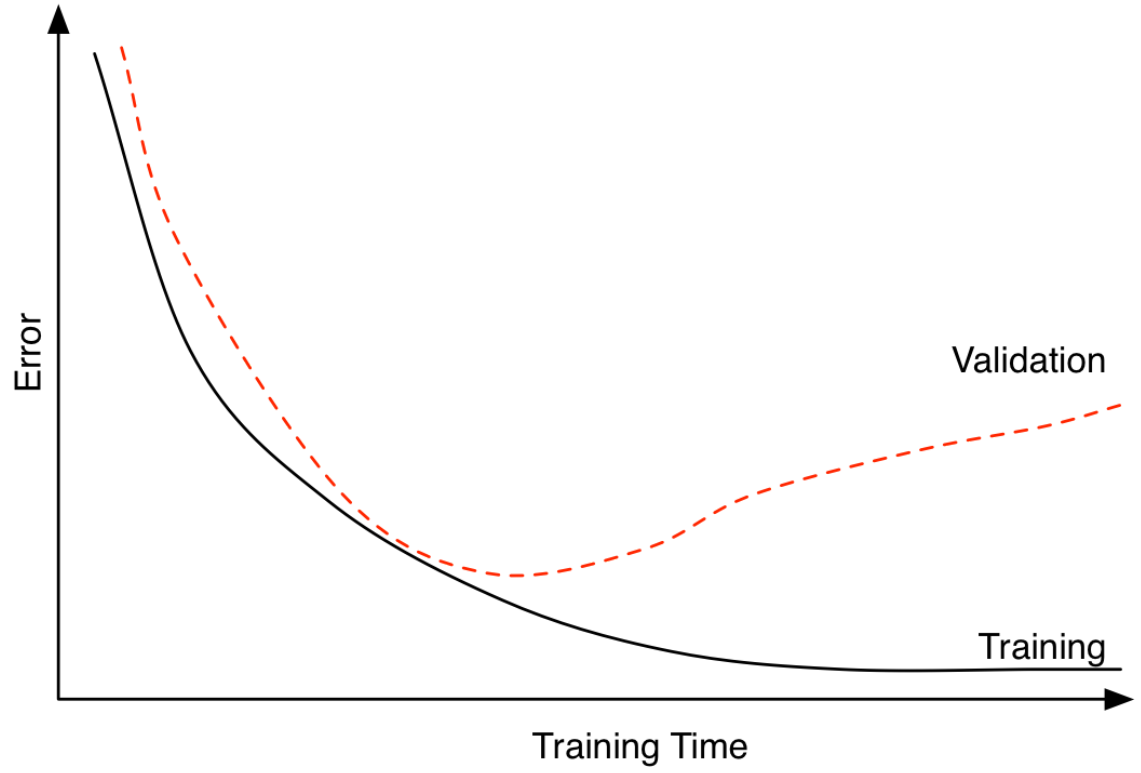
In [1]:
```
from collections.abc import Sequence
from sklearn import preprocessing
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import shutil
import os


# Encode text values to dummy variables(i.e. [1,0,0],[0,1,0],[0,0,1] for red,green,blue
def encode_text_dummy(df, name):
    dummies = pd.get_dummies(df[name])
    for x in dummies.columns:
        dummy_name = "{}-{}".format(name, x)
        df[dummy_name] = dummies[x]
    df.drop(name, axis=1, inplace=True)


# Encode text values to indexes(i.e. [1],[2],[3] for red,green,blue).
def encode_text_index(df, name):
    le = preprocessing.LabelEncoder()
    df[name] = le.fit_transform(df[name])
    return le.classes_


# Encode a numeric column as zscores
def encode_numeric_zscore(df, name, mean=None, sd=None):
    if mean is None:
```

```
        mean = df[name].mean()

    if sd is None:
        sd = df[name].std()

    df[name] = (df[name] - mean) / sd


# Convert all missing values in the specified column to the median
def missing_median(df, name):
    med = df[name].median()
    df[name] = df[name].fillna(med)


# Convert all missing values in the specified column to the default
def missing_default(df, name, default_value):
    df[name] = df[name].fillna(default_value)


# Convert a Pandas dataframe to the x,y inputs that TensorFlow needs
def to_xy(df, target):
    result = []
    for x in df.columns:
        if x != target:
            result.append(x)
    # find out the type of the target column.
    target_type = df[target].dtypes
    target_type = target_type[0] if isinstance(target_type, Sequence) else target_type
    # Encode to int for classification, float otherwise. TensorFlow likes 32 bits.
    if target_type in (np.int64, np.int32):
        # Classification
        dummies = pd.get_dummies(df[target])
        return df[result].values.astype(np.float32), dummies.values.astype(np.float32)
    else:
        # Regression
        return df[result].values.astype(np.float32), df[target].values.astype(np.float3

# 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)


# 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)
    a = plt.plot(t['y'].tolist(),label='expected')
    b = plt.plot(t['pred'].tolist(),label='prediction')
    plt.ylabel('output')
    plt.legend()
    plt.show()

# 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)
```

```
# Encode a column to a range between normalized_low and normalized_high.
def encode_numeric_range(df, name, normalized_low=-1, normalized_high=1,
                         data_low=None, data_high=None):
    if data_low is None:
        data_low = min(df[name])
        data_high = max(df[name])

    df[name] = ((df[name] - data_low) / (data_high - data_low)) \
               * (normalized_high - normalized_low) + normalized_low
```

Training with a Test Set with Early Stopping

Overfitting occurs when a neural network is trained to the point that it begins to memorize rather than generalize.



Split data into training and test using train_test_split

In [2]:

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

from sklearn.model_selection import train_test_split

from sklearn import metrics

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
```

```
from tensorflow.keras.callbacks import EarlyStopping

path = "./data/"

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

species = encode_text_index(df,"species")

x,y = to_xy(df,"species")

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

model = Sequential()

model.add(Dense(10, input_dim=x.shape[1], activation='relu'))
model.add(Dense(5,activation='relu'))
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=2, verbose=2, mode

# patience: number of epochs with no improvement after which training will be stopped

# The test set is checked during training to monitor progress for early stopping but is

model.fit(x_train, y_train, validation_data=(x_test,y_test), callbacks=[monitor], verbo
```

```
Train on 112 samples, validate on 38 samples
Epoch 1/1000
112/112 - 1s - loss: 1.2626 - val_loss: 1.2019
Epoch 2/1000
112/112 - 0s - loss: 1.2165 - val_loss: 1.1572
Epoch 3/1000
112/112 - 0s - loss: 1.1751 - val_loss: 1.1163
Epoch 4/1000
112/112 - 0s - loss: 1.1331 - val_loss: 1.0792
Epoch 5/1000
112/112 - 0s - loss: 1.0942 - val_loss: 1.0438
Epoch 6/1000
112/112 - 0s - loss: 1.0593 - val_loss: 1.0076
Epoch 7/1000
112/112 - 0s - loss: 1.0223 - val_loss: 0.9695
Epoch 8/1000
112/112 - 0s - loss: 0.9835 - val_loss: 0.9348
Epoch 9/1000
112/112 - 0s - loss: 0.9494 - val_loss: 0.9147
Epoch 10/1000
112/112 - 0s - loss: 0.9311 - val_loss: 0.8982
Epoch 11/1000
112/112 - 0s - loss: 0.9210 - val_loss: 0.8877
Epoch 12/1000
112/112 - 0s - loss: 0.9100 - val_loss: 0.8776
Epoch 13/1000
112/112 - 0s - loss: 0.9013 - val_loss: 0.8680
Epoch 14/1000
112/112 - 0s - loss: 0.8938 - val_loss: 0.8575
Epoch 15/1000
112/112 - 0s - loss: 0.8850 - val_loss: 0.8467
Epoch 16/1000
```

```
112/112 - 0s - loss: 0.8759 - val_loss: 0.8368
Epoch 17/1000
112/112 - 0s - loss: 0.8672 - val_loss: 0.8287
Epoch 18/1000
112/112 - 0s - loss: 0.8608 - val_loss: 0.8219
Epoch 19/1000
112/112 - 0s - loss: 0.8546 - val_loss: 0.8144
Epoch 20/1000
112/112 - 0s - loss: 0.8487 - val_loss: 0.8068
Epoch 21/1000
112/112 - 0s - loss: 0.8417 - val_loss: 0.7992
Epoch 22/1000
112/112 - 0s - loss: 0.8352 - val_loss: 0.7920
Epoch 23/1000
112/112 - 0s - loss: 0.8292 - val_loss: 0.7851
Epoch 24/1000
112/112 - 0s - loss: 0.8233 - val_loss: 0.7785
Epoch 25/1000
112/112 - 0s - loss: 0.8180 - val_loss: 0.7718
Epoch 26/1000
112/112 - 0s - loss: 0.8123 - val_loss: 0.7654
Epoch 27/1000
112/112 - 0s - loss: 0.8067 - val_loss: 0.7595
Epoch 28/1000
112/112 - 0s - loss: 0.8014 - val_loss: 0.7535
Epoch 29/1000
112/112 - 0s - loss: 0.7957 - val_loss: 0.7477
Epoch 30/1000
112/112 - 0s - loss: 0.7908 - val_loss: 0.7421
Epoch 31/1000
112/112 - 0s - loss: 0.7858 - val_loss: 0.7365
Epoch 32/1000
112/112 - 0s - loss: 0.7809 - val_loss: 0.7310
Epoch 33/1000
112/112 - 0s - loss: 0.7762 - val_loss: 0.7254
Epoch 34/1000
112/112 - 0s - loss: 0.7714 - val_loss: 0.7201
Epoch 35/1000
112/112 - 0s - loss: 0.7669 - val_loss: 0.7151
Epoch 36/1000
112/112 - 0s - loss: 0.7625 - val_loss: 0.7103
Epoch 37/1000
112/112 - 0s - loss: 0.7583 - val_loss: 0.7055
Epoch 38/1000
112/112 - 0s - loss: 0.7540 - val_loss: 0.7007
Epoch 39/1000
112/112 - 0s - loss: 0.7498 - val_loss: 0.6960
Epoch 40/1000
112/112 - 0s - loss: 0.7460 - val_loss: 0.6908
Epoch 41/1000
112/112 - 0s - loss: 0.7421 - val_loss: 0.6869
Epoch 42/1000
112/112 - 0s - loss: 0.7380 - val_loss: 0.6828
Epoch 43/1000
112/112 - 0s - loss: 0.7340 - val_loss: 0.6789
Epoch 44/1000
112/112 - 0s - loss: 0.7303 - val_loss: 0.6752
Epoch 45/1000
112/112 - 0s - loss: 0.7267 - val_loss: 0.6711
Epoch 46/1000
112/112 - 0s - loss: 0.7230 - val_loss: 0.6670
Epoch 47/1000
112/112 - 0s - loss: 0.7194 - val_loss: 0.6633
Epoch 48/1000
112/112 - 0s - loss: 0.7159 - val_loss: 0.6598
```

```
Epoch 49/1000
112/112 - 0s - loss: 0.7125 - val_loss: 0.6555
Epoch 50/1000
112/112 - 0s - loss: 0.7089 - val_loss: 0.6517
Epoch 51/1000
112/112 - 0s - loss: 0.7056 - val_loss: 0.6478
Epoch 52/1000
112/112 - 0s - loss: 0.7024 - val_loss: 0.6439
Epoch 53/1000
112/112 - 0s - loss: 0.6993 - val_loss: 0.6410
Epoch 54/1000
112/112 - 0s - loss: 0.6961 - val_loss: 0.6382
Epoch 55/1000
112/112 - 0s - loss: 0.6930 - val_loss: 0.6348
Epoch 56/1000
112/112 - 0s - loss: 0.6899 - val_loss: 0.6306
Epoch 57/1000
112/112 - 0s - loss: 0.6868 - val_loss: 0.6278
Epoch 58/1000
112/112 - 0s - loss: 0.6838 - val_loss: 0.6257
Epoch 59/1000
112/112 - 0s - loss: 0.6808 - val_loss: 0.6233
Epoch 60/1000
112/112 - 0s - loss: 0.6781 - val_loss: 0.6200
Epoch 61/1000
112/112 - 0s - loss: 0.6749 - val_loss: 0.6174
Epoch 62/1000
112/112 - 0s - loss: 0.6721 - val_loss: 0.6144
Epoch 63/1000
112/112 - 0s - loss: 0.6693 - val_loss: 0.6118
Epoch 64/1000
112/112 - 0s - loss: 0.6664 - val_loss: 0.6085
Epoch 65/1000
112/112 - 0s - loss: 0.6640 - val_loss: 0.6049
Epoch 66/1000
112/112 - 0s - loss: 0.6608 - val_loss: 0.6027
Epoch 67/1000
112/112 - 0s - loss: 0.6581 - val_loss: 0.6004
Epoch 68/1000
112/112 - 0s - loss: 0.6554 - val_loss: 0.5975
Epoch 69/1000
112/112 - 0s - loss: 0.6527 - val_loss: 0.5945
Epoch 70/1000
112/112 - 0s - loss: 0.6501 - val_loss: 0.5919
Epoch 71/1000
112/112 - 0s - loss: 0.6477 - val_loss: 0.5905
Epoch 72/1000
112/112 - 0s - loss: 0.6450 - val_loss: 0.5879
Epoch 73/1000
112/112 - 0s - loss: 0.6422 - val_loss: 0.5847
Epoch 74/1000
112/112 - 0s - loss: 0.6396 - val_loss: 0.5822
Epoch 75/1000
112/112 - 0s - loss: 0.6373 - val_loss: 0.5805
Epoch 76/1000
112/112 - 0s - loss: 0.6346 - val_loss: 0.5772
Epoch 77/1000
112/112 - 0s - loss: 0.6318 - val_loss: 0.5747
Epoch 78/1000
112/112 - 0s - loss: 0.6294 - val_loss: 0.5730
Epoch 79/1000
112/112 - 0s - loss: 0.6267 - val_loss: 0.5705
Epoch 80/1000
112/112 - 0s - loss: 0.6242 - val_loss: 0.5689
Epoch 81/1000
```

```
112/112 - 0s - loss: 0.6222 - val_loss: 0.5673
Epoch 82/1000
112/112 - 0s - loss: 0.6193 - val_loss: 0.5634
Epoch 83/1000
112/112 - 0s - loss: 0.6171 - val_loss: 0.5603
Epoch 84/1000
112/112 - 0s - loss: 0.6141 - val_loss: 0.5592
Epoch 85/1000
112/112 - 0s - loss: 0.6116 - val_loss: 0.5588
Epoch 86/1000
112/112 - 0s - loss: 0.6093 - val_loss: 0.5581
Epoch 87/1000
112/112 - 0s - loss: 0.6071 - val_loss: 0.5555
Epoch 88/1000
112/112 - 0s - loss: 0.6044 - val_loss: 0.5521
Epoch 89/1000
112/112 - 0s - loss: 0.6018 - val_loss: 0.5492
Epoch 90/1000
112/112 - 0s - loss: 0.5992 - val_loss: 0.5469
Epoch 91/1000
112/112 - 0s - loss: 0.5968 - val_loss: 0.5455
Epoch 92/1000
112/112 - 0s - loss: 0.5944 - val_loss: 0.5429
Epoch 93/1000
112/112 - 0s - loss: 0.5919 - val_loss: 0.5405
Epoch 94/1000
112/112 - 0s - loss: 0.5893 - val_loss: 0.5389
Epoch 95/1000
112/112 - 0s - loss: 0.5872 - val_loss: 0.5372
Epoch 96/1000
112/112 - 0s - loss: 0.5845 - val_loss: 0.5344
Epoch 97/1000
112/112 - 0s - loss: 0.5822 - val_loss: 0.5320
Epoch 98/1000
112/112 - 0s - loss: 0.5797 - val_loss: 0.5299
Epoch 99/1000
112/112 - 0s - loss: 0.5773 - val_loss: 0.5280
Epoch 100/1000
112/112 - 0s - loss: 0.5751 - val_loss: 0.5271
Epoch 101/1000
112/112 - 0s - loss: 0.5726 - val_loss: 0.5249
Epoch 102/1000
112/112 - 0s - loss: 0.5701 - val_loss: 0.5215
Epoch 103/1000
112/112 - 0s - loss: 0.5677 - val_loss: 0.5187
Epoch 104/1000
112/112 - 0s - loss: 0.5653 - val_loss: 0.5165
Epoch 105/1000
112/112 - 0s - loss: 0.5629 - val_loss: 0.5146
Epoch 106/1000
112/112 - 0s - loss: 0.5606 - val_loss: 0.5119
Epoch 107/1000
112/112 - 0s - loss: 0.5583 - val_loss: 0.5097
Epoch 108/1000
112/112 - 0s - loss: 0.5559 - val_loss: 0.5083
Epoch 109/1000
112/112 - 0s - loss: 0.5536 - val_loss: 0.5058
Epoch 110/1000
112/112 - 0s - loss: 0.5511 - val_loss: 0.5045
Epoch 111/1000
112/112 - 0s - loss: 0.5486 - val_loss: 0.5029
Epoch 112/1000
112/112 - 0s - loss: 0.5467 - val_loss: 0.5015
Epoch 113/1000
112/112 - 0s - loss: 0.5437 - val_loss: 0.4985
```

```
Epoch 114/1000
112/112 - 0s - loss: 0.5414 - val_loss: 0.4954
Epoch 115/1000
112/112 - 0s - loss: 0.5392 - val_loss: 0.4931
Epoch 116/1000
112/112 - 0s - loss: 0.5370 - val_loss: 0.4915
Epoch 117/1000
112/112 - 0s - loss: 0.5347 - val_loss: 0.4899
Epoch 118/1000
112/112 - 0s - loss: 0.5322 - val_loss: 0.4880
Epoch 119/1000
112/112 - 0s - loss: 0.5296 - val_loss: 0.4868
Epoch 120/1000
112/112 - 0s - loss: 0.5276 - val_loss: 0.4852
Epoch 121/1000
112/112 - 0s - loss: 0.5257 - val_loss: 0.4854
Epoch 122/1000
112/112 - 0s - loss: 0.5229 - val_loss: 0.4820
Epoch 123/1000
112/112 - 0s - loss: 0.5209 - val_loss: 0.4776
Epoch 124/1000
112/112 - 0s - loss: 0.5185 - val_loss: 0.4748
Epoch 125/1000
112/112 - 0s - loss: 0.5160 - val_loss: 0.4733
Epoch 126/1000
112/112 - 0s - loss: 0.5137 - val_loss: 0.4737
Epoch 127/1000
112/112 - 0s - loss: 0.5112 - val_loss: 0.4723
Epoch 00127: early stopping
```

Out[2]:
```
<tensorflow.python.keras.callbacks.History at 0x2a9e038f048>
```

Now that the neural network is trained, we can make predictions about the test set. The following code predicts the type of iris for test set and displays the first five irises.

In [3]:
```
pred = model.predict(x_test)
print(pred[0:5]) # print first five predictions
```

```
[[2.6873145e-01 5.3498286e-01 1.9628578e-01]
 [9.7115135e-01 2.8781431e-02 6.7221888e-05]
 [1.8109435e-01 3.6247870e-01 4.5642692e-01]
 [2.7048811e-01 5.2750272e-01 2.0200916e-01]
 [3.0047789e-01 5.6460226e-01 1.3491990e-01]]
```

Each line provides the probability that the iris is one of the 3 types of iris in the data set.

Saving Best Weights

It would be good idea to keep track of the most optimal weights during the entire training operation.

An additional monitor, ModelCheckpoint, is used and saves a copy of the neural network to **best_weights.hdf5** each time the validation score of the neural network improves.

Once training is done, we just reload this file and we have the optimal training weights that were found.

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

```
import requests
import numpy as np
import os
from sklearn.model_selection import train_test_split
from sklearn import metrics
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.callbacks import ModelCheckpoint

path = "./data/"

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

species = encode_text_index(df,"species")
x,y = to_xy(df,"species")

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

model = Sequential()
model.add(Dense(10, input_dim=x.shape[1], activation='relu'))
model.add(Dense(5,activation='relu'))
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

checkpointer = ModelCheckpoint(filepath="dnn/best_weights.hdf5", verbose=0, save_best_o

model.fit(x_train, y_train,validation_data=(x_test,y_test),callbacks=[monitor,checkpoin

model.load_weights('dnn/best_weights.hdf5') # load weights from best model
```

```
Train on 112 samples, validate on 38 samples
Epoch 1/1000
112/112 - 1s - loss: 1.7000 - val_loss: 1.8940
Epoch 2/1000
112/112 - 0s - loss: 1.6247 - val_loss: 1.7986
Epoch 3/1000
112/112 - 0s - loss: 1.5448 - val_loss: 1.7097
Epoch 4/1000
112/112 - 0s - loss: 1.4785 - val_loss: 1.6222
Epoch 5/1000
112/112 - 0s - loss: 1.4108 - val_loss: 1.5375
Epoch 6/1000
112/112 - 0s - loss: 1.3458 - val_loss: 1.4541
Epoch 7/1000
112/112 - 0s - loss: 1.2834 - val_loss: 1.3741
Epoch 8/1000
112/112 - 0s - loss: 1.2235 - val_loss: 1.2980
Epoch 9/1000
112/112 - 0s - loss: 1.1591 - val_loss: 1.2249
Epoch 10/1000
112/112 - 0s - loss: 1.0987 - val_loss: 1.1520
Epoch 11/1000
112/112 - 0s - loss: 1.0413 - val_loss: 1.0807
Epoch 12/1000
112/112 - 0s - loss: 0.9851 - val_loss: 1.0105
Epoch 13/1000
```

```
112/112 - 0s - loss: 0.9336 - val_loss: 0.9444
Epoch 14/1000
112/112 - 0s - loss: 0.8886 - val_loss: 0.8859
Epoch 15/1000
112/112 - 0s - loss: 0.8460 - val_loss: 0.8379
Epoch 16/1000
112/112 - 0s - loss: 0.8103 - val_loss: 0.7970
Epoch 17/1000
112/112 - 0s - loss: 0.7838 - val_loss: 0.7637
Epoch 18/1000
112/112 - 0s - loss: 0.7578 - val_loss: 0.7386
Epoch 19/1000
112/112 - 0s - loss: 0.7444 - val_loss: 0.7198
Epoch 20/1000
112/112 - 0s - loss: 0.7338 - val_loss: 0.7065
Epoch 21/1000
112/112 - 0s - loss: 0.7234 - val_loss: 0.6949
Epoch 22/1000
112/112 - 0s - loss: 0.7161 - val_loss: 0.6850
Epoch 23/1000
112/112 - 0s - loss: 0.7094 - val_loss: 0.6767
Epoch 24/1000
112/112 - 0s - loss: 0.7014 - val_loss: 0.6682
Epoch 25/1000
112/112 - 0s - loss: 0.6939 - val_loss: 0.6610
Epoch 26/1000
112/112 - 0s - loss: 0.6868 - val_loss: 0.6539
Epoch 27/1000
112/112 - 0s - loss: 0.6799 - val_loss: 0.6469
Epoch 28/1000
112/112 - 0s - loss: 0.6730 - val_loss: 0.6404
Epoch 29/1000
112/112 - 0s - loss: 0.6668 - val_loss: 0.6339
Epoch 30/1000
112/112 - 0s - loss: 0.6605 - val_loss: 0.6261
Epoch 31/1000
112/112 - 0s - loss: 0.6543 - val_loss: 0.6191
Epoch 32/1000
112/112 - 0s - loss: 0.6483 - val_loss: 0.6121
Epoch 33/1000
112/112 - 3s - loss: 0.6420 - val_loss: 0.6046
Epoch 34/1000
112/112 - 0s - loss: 0.6370 - val_loss: 0.5981
Epoch 35/1000
112/112 - 0s - loss: 0.6314 - val_loss: 0.5925
Epoch 36/1000
112/112 - 0s - loss: 0.6256 - val_loss: 0.5877
Epoch 37/1000
112/112 - 0s - loss: 0.6209 - val_loss: 0.5830
Epoch 38/1000
112/112 - 0s - loss: 0.6153 - val_loss: 0.5769
Epoch 39/1000
112/112 - 0s - loss: 0.6101 - val_loss: 0.5714
Epoch 40/1000
112/112 - 0s - loss: 0.6053 - val_loss: 0.5660
Epoch 41/1000
112/112 - 0s - loss: 0.6003 - val_loss: 0.5612
Epoch 42/1000
112/112 - 0s - loss: 0.5955 - val_loss: 0.5556
Epoch 43/1000
112/112 - 0s - loss: 0.5907 - val_loss: 0.5507
Epoch 44/1000
112/112 - 0s - loss: 0.5859 - val_loss: 0.5457
Epoch 45/1000
112/112 - 0s - loss: 0.5814 - val_loss: 0.5404
```

```
Epoch 46/1000
112/112 - 0s - loss: 0.5771 - val_loss: 0.5350
Epoch 47/1000
112/112 - 2s - loss: 0.5725 - val_loss: 0.5302
Epoch 48/1000
112/112 - 0s - loss: 0.5682 - val_loss: 0.5255
Epoch 49/1000
112/112 - 0s - loss: 0.5638 - val_loss: 0.5210
Epoch 50/1000
112/112 - 0s - loss: 0.5600 - val_loss: 0.5166
Epoch 51/1000
112/112 - 0s - loss: 0.5554 - val_loss: 0.5118
Epoch 52/1000
112/112 - 0s - loss: 0.5512 - val_loss: 0.5070
Epoch 53/1000
112/112 - 0s - loss: 0.5487 - val_loss: 0.5022
Epoch 54/1000
112/112 - 0s - loss: 0.5440 - val_loss: 0.4981
Epoch 55/1000
112/112 - 0s - loss: 0.5394 - val_loss: 0.4942
Epoch 56/1000
112/112 - 0s - loss: 0.5358 - val_loss: 0.4908
Epoch 57/1000
112/112 - 0s - loss: 0.5319 - val_loss: 0.4872
Epoch 58/1000
112/112 - 0s - loss: 0.5283 - val_loss: 0.4834
Epoch 59/1000
112/112 - 0s - loss: 0.5248 - val_loss: 0.4791
Epoch 60/1000
112/112 - 0s - loss: 0.5209 - val_loss: 0.4752
Epoch 61/1000
112/112 - 0s - loss: 0.5175 - val_loss: 0.4709
Epoch 62/1000
112/112 - 0s - loss: 0.5140 - val_loss: 0.4674
Epoch 63/1000
112/112 - 0s - loss: 0.5104 - val_loss: 0.4636
Epoch 64/1000
112/112 - 0s - loss: 0.5072 - val_loss: 0.4601
Epoch 65/1000
112/112 - 0s - loss: 0.5036 - val_loss: 0.4566
Epoch 66/1000
112/112 - 0s - loss: 0.5004 - val_loss: 0.4534
Epoch 67/1000
112/112 - 0s - loss: 0.4982 - val_loss: 0.4499
Epoch 68/1000
112/112 - 0s - loss: 0.4947 - val_loss: 0.4474
Epoch 69/1000
112/112 - 0s - loss: 0.4906 - val_loss: 0.4442
Epoch 70/1000
112/112 - 0s - loss: 0.4873 - val_loss: 0.4411
Epoch 71/1000
112/112 - 0s - loss: 0.4842 - val_loss: 0.4376
Epoch 72/1000
112/112 - 0s - loss: 0.4807 - val_loss: 0.4336
Epoch 73/1000
112/112 - 0s - loss: 0.4781 - val_loss: 0.4296
Epoch 74/1000
112/112 - 0s - loss: 0.4749 - val_loss: 0.4261
Epoch 75/1000
112/112 - 2s - loss: 0.4718 - val_loss: 0.4230
Epoch 76/1000
112/112 - 0s - loss: 0.4688 - val_loss: 0.4205
Epoch 77/1000
112/112 - 0s - loss: 0.4655 - val_loss: 0.4183
Epoch 78/1000
```

```
112/112 - 0s - loss: 0.4625 - val_loss: 0.4155
Epoch 79/1000
112/112 - 0s - loss: 0.4595 - val_loss: 0.4125
Epoch 80/1000
112/112 - 0s - loss: 0.4570 - val_loss: 0.4095
Epoch 81/1000
112/112 - 0s - loss: 0.4541 - val_loss: 0.4052
Epoch 82/1000
112/112 - 0s - loss: 0.4504 - val_loss: 0.4019
Epoch 83/1000
112/112 - 0s - loss: 0.4475 - val_loss: 0.3987
Epoch 84/1000
112/112 - 0s - loss: 0.4448 - val_loss: 0.3953
Epoch 85/1000
112/112 - 0s - loss: 0.4423 - val_loss: 0.3925
Epoch 86/1000
112/112 - 0s - loss: 0.4396 - val_loss: 0.3893
Epoch 87/1000
112/112 - 0s - loss: 0.4367 - val_loss: 0.3866
Epoch 88/1000
112/112 - 0s - loss: 0.4341 - val_loss: 0.3842
Epoch 89/1000
112/112 - 0s - loss: 0.4310 - val_loss: 0.3813
Epoch 90/1000
112/112 - 0s - loss: 0.4280 - val_loss: 0.3788
Epoch 91/1000
112/112 - 0s - loss: 0.4249 - val_loss: 0.3767
Epoch 92/1000
112/112 - 0s - loss: 0.4237 - val_loss: 0.3748
Epoch 93/1000
112/112 - 0s - loss: 0.4198 - val_loss: 0.3709
Epoch 94/1000
112/112 - 0s - loss: 0.4172 - val_loss: 0.3675
Epoch 95/1000
112/112 - 0s - loss: 0.4138 - val_loss: 0.3652
Epoch 96/1000
112/112 - 0s - loss: 0.4112 - val_loss: 0.3631
Epoch 97/1000
112/112 - 0s - loss: 0.4085 - val_loss: 0.3604
Epoch 98/1000
112/112 - 0s - loss: 0.4054 - val_loss: 0.3585
Epoch 99/1000
112/112 - 0s - loss: 0.4029 - val_loss: 0.3559
Epoch 100/1000
112/112 - 0s - loss: 0.4001 - val_loss: 0.3522
Epoch 101/1000
112/112 - 0s - loss: 0.3975 - val_loss: 0.3492
Epoch 102/1000
112/112 - 0s - loss: 0.3944 - val_loss: 0.3461
Epoch 103/1000
112/112 - 0s - loss: 0.3918 - val_loss: 0.3431
Epoch 104/1000
112/112 - 0s - loss: 0.3899 - val_loss: 0.3398
Epoch 105/1000
112/112 - 0s - loss: 0.3867 - val_loss: 0.3372
Epoch 106/1000
112/112 - 0s - loss: 0.3838 - val_loss: 0.3347
Epoch 107/1000
112/112 - 0s - loss: 0.3814 - val_loss: 0.3325
Epoch 108/1000
112/112 - 0s - loss: 0.3786 - val_loss: 0.3303
Epoch 109/1000
112/112 - 0s - loss: 0.3760 - val_loss: 0.3279
Epoch 110/1000
112/112 - 0s - loss: 0.3734 - val_loss: 0.3249
```

```
Epoch 111/1000
112/112 - 0s - loss: 0.3719 - val_loss: 0.3221
Epoch 112/1000
112/112 - 0s - loss: 0.3683 - val_loss: 0.3202
Epoch 113/1000
112/112 - 0s - loss: 0.3655 - val_loss: 0.3189
Epoch 114/1000
112/112 - 0s - loss: 0.3633 - val_loss: 0.3172
Epoch 115/1000
112/112 - 0s - loss: 0.3609 - val_loss: 0.3166
Epoch 116/1000
112/112 - 0s - loss: 0.3596 - val_loss: 0.3142
Epoch 117/1000
112/112 - 0s - loss: 0.3568 - val_loss: 0.3119
Epoch 118/1000
112/112 - 0s - loss: 0.3546 - val_loss: 0.3094
Epoch 119/1000
112/112 - 0s - loss: 0.3518 - val_loss: 0.3067
Epoch 120/1000
112/112 - 0s - loss: 0.3493 - val_loss: 0.3032
Epoch 121/1000
112/112 - 0s - loss: 0.3465 - val_loss: 0.2999
Epoch 122/1000
112/112 - 0s - loss: 0.3442 - val_loss: 0.2968
Epoch 123/1000
112/112 - 0s - loss: 0.3416 - val_loss: 0.2947
Epoch 124/1000
112/112 - 0s - loss: 0.3393 - val_loss: 0.2923
Epoch 125/1000
112/112 - 0s - loss: 0.3372 - val_loss: 0.2903
Epoch 126/1000
112/112 - 0s - loss: 0.3348 - val_loss: 0.2881
Epoch 127/1000
112/112 - 0s - loss: 0.3326 - val_loss: 0.2865
Epoch 128/1000
112/112 - 0s - loss: 0.3306 - val_loss: 0.2851
Epoch 129/1000
112/112 - 0s - loss: 0.3284 - val_loss: 0.2828
Epoch 130/1000
112/112 - 0s - loss: 0.3269 - val_loss: 0.2809
Epoch 131/1000
112/112 - 0s - loss: 0.3240 - val_loss: 0.2779
Epoch 132/1000
112/112 - 0s - loss: 0.3217 - val_loss: 0.2756
Epoch 133/1000
112/112 - 0s - loss: 0.3205 - val_loss: 0.2733
Epoch 134/1000
112/112 - 0s - loss: 0.3180 - val_loss: 0.2719
Epoch 135/1000
112/112 - 0s - loss: 0.3156 - val_loss: 0.2702
Epoch 136/1000
112/112 - 0s - loss: 0.3137 - val_loss: 0.2679
Epoch 137/1000
112/112 - 0s - loss: 0.3114 - val_loss: 0.2663
Epoch 138/1000
112/112 - 0s - loss: 0.3093 - val_loss: 0.2648
Epoch 139/1000
112/112 - 0s - loss: 0.3074 - val_loss: 0.2632
Epoch 140/1000
112/112 - 0s - loss: 0.3054 - val_loss: 0.2611
Epoch 141/1000
112/112 - 0s - loss: 0.3034 - val_loss: 0.2588
Epoch 142/1000
112/112 - 0s - loss: 0.3020 - val_loss: 0.2559
Epoch 143/1000
```

```
112/112 - 0s - loss: 0.2998 - val_loss: 0.2539
Epoch 144/1000
112/112 - 0s - loss: 0.2996 - val_loss: 0.2520
Epoch 145/1000
112/112 - 0s - loss: 0.2969 - val_loss: 0.2503
Epoch 146/1000
112/112 - 0s - loss: 0.2937 - val_loss: 0.2492
Epoch 147/1000
112/112 - 0s - loss: 0.2913 - val_loss: 0.2488
Epoch 148/1000
112/112 - 0s - loss: 0.2903 - val_loss: 0.2480
Epoch 149/1000
112/112 - 0s - loss: 0.2885 - val_loss: 0.2465
Epoch 150/1000
112/112 - 0s - loss: 0.2867 - val_loss: 0.2443
Epoch 151/1000
112/112 - 0s - loss: 0.2848 - val_loss: 0.2410
Epoch 152/1000
112/112 - 0s - loss: 0.2828 - val_loss: 0.2385
Epoch 153/1000
112/112 - 0s - loss: 0.2807 - val_loss: 0.2369
Epoch 154/1000
112/112 - 0s - loss: 0.2790 - val_loss: 0.2350
Epoch 155/1000
112/112 - 0s - loss: 0.2771 - val_loss: 0.2336
Epoch 156/1000
112/112 - 0s - loss: 0.2752 - val_loss: 0.2323
Epoch 157/1000
112/112 - 0s - loss: 0.2735 - val_loss: 0.2311
Epoch 158/1000
112/112 - 0s - loss: 0.2720 - val_loss: 0.2299
Epoch 159/1000
112/112 - 0s - loss: 0.2704 - val_loss: 0.2273
Epoch 160/1000
112/112 - 0s - loss: 0.2682 - val_loss: 0.2253
Epoch 161/1000
112/112 - 0s - loss: 0.2665 - val_loss: 0.2235
Epoch 162/1000
112/112 - 0s - loss: 0.2649 - val_loss: 0.2222
Epoch 163/1000
112/112 - 0s - loss: 0.2631 - val_loss: 0.2204
Epoch 164/1000
112/112 - 0s - loss: 0.2616 - val_loss: 0.2192
Epoch 165/1000
112/112 - 0s - loss: 0.2597 - val_loss: 0.2179
Epoch 166/1000
112/112 - 0s - loss: 0.2580 - val_loss: 0.2166
Epoch 167/1000
112/112 - 0s - loss: 0.2563 - val_loss: 0.2149
Epoch 168/1000
112/112 - 0s - loss: 0.2550 - val_loss: 0.2135
Epoch 169/1000
112/112 - 0s - loss: 0.2532 - val_loss: 0.2110
Epoch 170/1000
112/112 - 0s - loss: 0.2513 - val_loss: 0.2097
Epoch 171/1000
112/112 - 0s - loss: 0.2496 - val_loss: 0.2083
Epoch 172/1000
112/112 - 0s - loss: 0.2479 - val_loss: 0.2069
Epoch 173/1000
112/112 - 0s - loss: 0.2465 - val_loss: 0.2055
Epoch 174/1000
112/112 - 0s - loss: 0.2450 - val_loss: 0.2044
Epoch 175/1000
112/112 - 0s - loss: 0.2434 - val_loss: 0.2022
```

```
Epoch 176/1000
112/112 - 0s - loss: 0.2418 - val_loss: 0.2006
Epoch 177/1000
112/112 - 0s - loss: 0.2404 - val_loss: 0.1985
Epoch 178/1000
112/112 - 0s - loss: 0.2388 - val_loss: 0.1969
Epoch 179/1000
112/112 - 0s - loss: 0.2374 - val_loss: 0.1957
Epoch 180/1000
112/112 - 0s - loss: 0.2360 - val_loss: 0.1951
Epoch 181/1000
112/112 - 0s - loss: 0.2340 - val_loss: 0.1944
Epoch 182/1000
112/112 - 0s - loss: 0.2329 - val_loss: 0.1942
Epoch 183/1000
112/112 - 0s - loss: 0.2318 - val_loss: 0.1921
Epoch 184/1000
112/112 - 0s - loss: 0.2301 - val_loss: 0.1907
Epoch 185/1000
112/112 - 0s - loss: 0.2282 - val_loss: 0.1879
Epoch 186/1000
112/112 - 0s - loss: 0.2267 - val_loss: 0.1861
Epoch 187/1000
112/112 - 0s - loss: 0.2256 - val_loss: 0.1839
Epoch 188/1000
112/112 - 0s - loss: 0.2244 - val_loss: 0.1826
Epoch 189/1000
112/112 - 0s - loss: 0.2231 - val_loss: 0.1815
Epoch 190/1000
112/112 - 0s - loss: 0.2214 - val_loss: 0.1802
Epoch 191/1000
112/112 - 0s - loss: 0.2198 - val_loss: 0.1800
Epoch 192/1000
112/112 - 0s - loss: 0.2185 - val_loss: 0.1793
Epoch 193/1000
112/112 - 0s - loss: 0.2171 - val_loss: 0.1779
Epoch 194/1000
112/112 - 0s - loss: 0.2156 - val_loss: 0.1764
Epoch 195/1000
112/112 - 0s - loss: 0.2143 - val_loss: 0.1742
Epoch 196/1000
112/112 - 0s - loss: 0.2128 - val_loss: 0.1729
Epoch 197/1000
112/112 - 0s - loss: 0.2118 - val_loss: 0.1714
Epoch 198/1000
112/112 - 0s - loss: 0.2100 - val_loss: 0.1707
Epoch 199/1000
112/112 - 0s - loss: 0.2094 - val_loss: 0.1712
Epoch 200/1000
112/112 - 0s - loss: 0.2080 - val_loss: 0.1709
Epoch 201/1000
112/112 - 0s - loss: 0.2081 - val_loss: 0.1701
Epoch 202/1000
112/112 - 0s - loss: 0.2067 - val_loss: 0.1672
Epoch 203/1000
112/112 - 0s - loss: 0.2054 - val_loss: 0.1641
Epoch 204/1000
112/112 - 0s - loss: 0.2031 - val_loss: 0.1624
Epoch 205/1000
112/112 - 0s - loss: 0.2017 - val_loss: 0.1614
Epoch 206/1000
112/112 - 0s - loss: 0.2003 - val_loss: 0.1607
Epoch 207/1000
112/112 - 0s - loss: 0.2000 - val_loss: 0.1604
Epoch 208/1000
```

```
112/112 - 0s - loss: 0.1984 - val_loss: 0.1585
Epoch 209/1000
112/112 - 0s - loss: 0.1979 - val_loss: 0.1566
Epoch 210/1000
112/112 - 0s - loss: 0.1956 - val_loss: 0.1555
Epoch 211/1000
112/112 - 0s - loss: 0.1947 - val_loss: 0.1549
Epoch 212/1000
112/112 - 0s - loss: 0.1933 - val_loss: 0.1534
Epoch 213/1000
112/112 - 0s - loss: 0.1920 - val_loss: 0.1524
Epoch 214/1000
112/112 - 0s - loss: 0.1910 - val_loss: 0.1518
Epoch 215/1000
112/112 - 0s - loss: 0.1899 - val_loss: 0.1505
Epoch 216/1000
112/112 - 0s - loss: 0.1885 - val_loss: 0.1491
Epoch 217/1000
112/112 - 0s - loss: 0.1874 - val_loss: 0.1482
Epoch 218/1000
112/112 - 0s - loss: 0.1862 - val_loss: 0.1470
Epoch 219/1000
112/112 - 0s - loss: 0.1854 - val_loss: 0.1461
Epoch 220/1000
112/112 - 0s - loss: 0.1839 - val_loss: 0.1446
Epoch 221/1000
112/112 - 0s - loss: 0.1828 - val_loss: 0.1430
Epoch 222/1000
112/112 - 0s - loss: 0.1821 - val_loss: 0.1417
Epoch 223/1000
112/112 - 0s - loss: 0.1817 - val_loss: 0.1407
Epoch 224/1000
112/112 - 0s - loss: 0.1804 - val_loss: 0.1416
Epoch 225/1000
112/112 - 0s - loss: 0.1803 - val_loss: 0.1423
Epoch 226/1000
112/112 - 0s - loss: 0.1781 - val_loss: 0.1399
Epoch 227/1000
112/112 - 0s - loss: 0.1775 - val_loss: 0.1378
Epoch 228/1000
112/112 - 0s - loss: 0.1754 - val_loss: 0.1374
Epoch 229/1000
112/112 - 0s - loss: 0.1747 - val_loss: 0.1356
Epoch 230/1000
112/112 - 0s - loss: 0.1731 - val_loss: 0.1350
Epoch 231/1000
112/112 - 0s - loss: 0.1725 - val_loss: 0.1344
Epoch 232/1000
112/112 - 0s - loss: 0.1710 - val_loss: 0.1340
Epoch 233/1000
112/112 - 0s - loss: 0.1702 - val_loss: 0.1334
Epoch 234/1000
112/112 - 0s - loss: 0.1693 - val_loss: 0.1317
Epoch 235/1000
112/112 - 0s - loss: 0.1681 - val_loss: 0.1305
Epoch 236/1000
112/112 - 0s - loss: 0.1670 - val_loss: 0.1294
Epoch 237/1000
112/112 - 0s - loss: 0.1664 - val_loss: 0.1294
Epoch 238/1000
112/112 - 0s - loss: 0.1651 - val_loss: 0.1277
Epoch 239/1000
112/112 - 0s - loss: 0.1642 - val_loss: 0.1252
Epoch 240/1000
112/112 - 0s - loss: 0.1630 - val_loss: 0.1240
```

```
Epoch 241/1000
112/112 - 0s - loss: 0.1620 - val_loss: 0.1230
Epoch 242/1000
112/112 - 0s - loss: 0.1612 - val_loss: 0.1225
Epoch 243/1000
112/112 - 0s - loss: 0.1601 - val_loss: 0.1224
Epoch 244/1000
112/112 - 0s - loss: 0.1602 - val_loss: 0.1226
Epoch 245/1000
112/112 - 0s - loss: 0.1584 - val_loss: 0.1199
Epoch 246/1000
112/112 - 0s - loss: 0.1576 - val_loss: 0.1184
Epoch 247/1000
112/112 - 0s - loss: 0.1567 - val_loss: 0.1180
Epoch 248/1000
112/112 - 0s - loss: 0.1562 - val_loss: 0.1168
Epoch 249/1000
112/112 - 0s - loss: 0.1550 - val_loss: 0.1160
Epoch 250/1000
112/112 - 0s - loss: 0.1541 - val_loss: 0.1152
Epoch 251/1000
112/112 - 0s - loss: 0.1531 - val_loss: 0.1150
Epoch 252/1000
112/112 - 0s - loss: 0.1527 - val_loss: 0.1139
Epoch 253/1000
112/112 - 0s - loss: 0.1512 - val_loss: 0.1139
Epoch 254/1000
112/112 - 0s - loss: 0.1516 - val_loss: 0.1147
Epoch 255/1000
112/112 - 0s - loss: 0.1497 - val_loss: 0.1124
Epoch 256/1000
112/112 - 0s - loss: 0.1487 - val_loss: 0.1105
Epoch 257/1000
112/112 - 0s - loss: 0.1485 - val_loss: 0.1090
Epoch 258/1000
112/112 - 0s - loss: 0.1476 - val_loss: 0.1086
Epoch 259/1000
112/112 - 0s - loss: 0.1483 - val_loss: 0.1088
Epoch 260/1000
112/112 - 0s - loss: 0.1461 - val_loss: 0.1068
Epoch 261/1000
112/112 - 0s - loss: 0.1453 - val_loss: 0.1058
Epoch 262/1000
112/112 - 0s - loss: 0.1454 - val_loss: 0.1049
Epoch 263/1000
112/112 - 0s - loss: 0.1448 - val_loss: 0.1057
Epoch 264/1000
112/112 - 0s - loss: 0.1441 - val_loss: 0.1063
Epoch 265/1000
112/112 - 0s - loss: 0.1421 - val_loss: 0.1046
Epoch 266/1000
112/112 - 0s - loss: 0.1416 - val_loss: 0.1024
Epoch 267/1000
112/112 - 0s - loss: 0.1416 - val_loss: 0.1015
Epoch 268/1000
112/112 - 0s - loss: 0.1416 - val_loss: 0.1022
Epoch 269/1000
112/112 - 0s - loss: 0.1395 - val_loss: 0.1012
Epoch 270/1000
112/112 - 0s - loss: 0.1397 - val_loss: 0.1002
Epoch 271/1000
112/112 - 0s - loss: 0.1392 - val_loss: 0.1016
Epoch 272/1000
112/112 - 0s - loss: 0.1377 - val_loss: 0.1015
Epoch 273/1000
```

```
112/112 - 0s - loss: 0.1374 - val_loss: 0.1019
Epoch 274/1000
112/112 - 0s - loss: 0.1367 - val_loss: 0.1002
Epoch 00274: early stopping
```

Potential Keras Issue on Small Networks Regarding Saving Optimal Weights

You might occasionally see this error:

```
OSError: Unable to create file (Unable to open file: name =
'dnn/best_weights.hdf5', errno = 22, error message = 'invalid argument',
flags = 13, o_flags = 302)
```

Usually you can just run rerun the code and it goes away. This is an unfortnuate result of saving a file each time the validation score improves (as described in the previous section). If the errors improve two rapidly, you might try to save the file twice and get an error from these two saves overlapping. For larger neural networks this will not be a problem because each training step will take longer, allowing for plenty of time for the previous save to complete.

Evaluating Classification Models

(1) Calculate Classification Accuracy/Precision/Recall/F1-Score

By default, Keras will return the predicted probability for each class. We can change these prediction probabilities into the actual iris predicted with **argmax**.

In [5]:
```
pred = model.predict(x_test)
pred
```

Out[5]:
```
array([[4.71494487e-03, 8.45346868e-01, 1.49938241e-01],
       [9.91973162e-01, 8.02563876e-03, 1.13456713e-06],
       [3.45179842e-06, 1.14592100e-02, 9.88537431e-01],
       [4.64522187e-03, 7.54409015e-01, 2.40945816e-01],
       [3.53244087e-03, 9.27941024e-01, 6.85265660e-02],
       [9.86725211e-01, 1.32721653e-02, 2.55039140e-06],
       [1.58099979e-02, 9.57360506e-01, 2.68294606e-02],
       [6.04713976e-04, 2.81531900e-01, 7.17863441e-01],
       [3.46631696e-03, 7.80912101e-01, 2.15621606e-01],
       [1.09425150e-02, 9.57177758e-01, 3.18797417e-02],
       [8.09577235e-04, 2.81197578e-01, 7.17992783e-01],
       [9.80344415e-01, 1.96491349e-02, 6.47375873e-06],
       [9.89732563e-01, 1.02663469e-02, 1.04882076e-06],
       [9.82253373e-01, 1.77410170e-02, 5.62334071e-06],
       [9.92402792e-01, 7.59436050e-03, 2.86538329e-06],
       [4.05827072e-03, 7.64293194e-01, 2.31648564e-01],
       [2.61457735e-05, 1.92335658e-02, 9.80740309e-01],
       [1.11245075e-02, 9.46039975e-01, 4.28355858e-02],
       [5.10215200e-03, 7.19326735e-01, 2.75571018e-01],
       [3.78164477e-05, 2.46756002e-02, 9.75286603e-01],
       [9.83402491e-01, 1.65870525e-02, 1.04899545e-05],
       [1.19526382e-03, 3.07682246e-01, 6.91122472e-01],
       [9.84546602e-01, 1.54456282e-02, 7.71218220e-06],
       [5.48412863e-05, 3.42011787e-02, 9.65744019e-01],
       [2.20155925e-04, 2.72331923e-01, 7.27447867e-01],
```

```
       [2.76573730e-04, 1.37017280e-01, 8.62706184e-01],
       [1.12366797e-04, 8.86308253e-02, 9.11256909e-01],
       [2.92064469e-05, 2.47451868e-02, 9.75225627e-01],
       [9.78956521e-01, 2.10346375e-02, 8.85666213e-06],
       [9.80891228e-01, 1.90992802e-02, 9.43361738e-06],
       [9.91707861e-01, 8.28820188e-03, 3.87211685e-06],
       [9.96441185e-01, 3.55826062e-03, 6.00188741e-07],
       [5.96467266e-03, 9.71610248e-01, 2.24251188e-02],
       [9.87222135e-01, 1.27705336e-02, 7.25694008e-06],
       [9.84735966e-01, 1.52515899e-02, 1.24963326e-05],
       [6.12511649e-04, 2.30411321e-01, 7.68976212e-01],
       [5.57778357e-03, 9.09627855e-01, 8.47943202e-02],
       [9.89120841e-01, 1.08765354e-02, 2.57044030e-06]], dtype=float32)
```

In [6]:

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

```
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1
 0]
```

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

In [7]:

```
y_true= np.argmax(y_test,axis=1)

score = metrics.accuracy_score(y_true, pred)

print("Accuracy score: {}".format(score))
```

```
Accuracy score: 1.0
```

In [8]:

```
score = metrics.precision_score(y_true, pred, average= "weighted")
print("Precision score: {}".format(score))
```

```
Precision score: 1.0
```

In [9]:

```
score = metrics.recall_score(y_true, pred, average= "weighted")
print("Recall score: {}".format(score))
```

```
Recall score: 1.0
```

In [10]:

```
score = metrics.f1_score(y_true, pred, average= "weighted")
print("F1 score: {}".format(score))
```

```
F1 score: 1.0
```

In [11]:

```
print(metrics.classification_report(y_true, pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        15
           1       1.00      1.00      1.00        11
           2       1.00      1.00      1.00        12

    accuracy                           1.00        38
   macro avg       1.00      1.00      1.00        38
weighted avg       1.00      1.00      1.00        38
```

(2) Calculate Classification Cross-Entropy Loss (Log Loss)

Log loss is an error metric that is often used in place of accuracy for classification.

Log loss allows for "partial credit". For example, a model might be used to classify A, B and C. The correct answer might be A, however if the classification network chose B as having the highest probability, then accuracy gives the neural network no credit for this classification.

However, with log loss, the probability of the correct answer is added to the score. For example, the correct answer might be A, but if the neural network only predicted .4 probability of A being correct, then the value -log(.4) is added.

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\log(\hat{y}_{ij})$$

The following code shows the logloss scores that correspond to the average probablity for the correct item. The **pred** column specifies the average robability for the correct class. The **logloss** column specifies the log loss for that probability.

Calculating log loss

In [12]:
```
# Generate predictions
pred = model.predict(x_test)

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

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

```
Numpy array of predictions
[[4.7149449e-03 8.4534687e-01 1.4993824e-01]
 [9.9197316e-01 8.0256388e-03 1.1345671e-06]
 [3.4517984e-06 1.1459210e-02 9.8853743e-01]
 [4.6452219e-03 7.5440902e-01 2.4094582e-01]
 [3.5324409e-03 9.2794102e-01 6.8526566e-02]]

y_test:
[[0. 1. 0.]
 [1. 0. 0.]
 [0. 0. 1.]
 [0. 1. 0.]
 [0. 1. 0.]]
Log loss score: 0.100190236594675
```

Evaluating Regression Models

Regression results are evaluated differently than classification. Consider the following code.

In [13]:
```
from sklearn.model_selection import train_test_split
import pandas as pd
```

```
import numpy as np
from sklearn import metrics

path = "./data/"

filename_read = os.path.join(path,"auto-mpg.csv")
df = pd.read_csv(filename_read,na_values=['NA','?'])

cars = df['name']
df.drop('name',1,inplace=True)
missing_median(df, 'horsepower')

encode_text_dummy(df, 'origin')

x,y = to_xy(df,"mpg")

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

model = Sequential()
model.add(Dense(10, input_dim=x.shape[1], activation='relu'))
model.add(Dense(10))
model.add(Dense(10))
model.add(Dense(10))

model.add(Dense(1))  # 1 output neuron


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

monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3, patience=5, verbose=1, mode

model.fit(x_train,y_train, validation_data=(x_test,y_test),callbacks=[monitor],verbose=
```

```
Train on 298 samples, validate on 100 samples
Epoch 1/1000
298/298 - 1s - loss: 2822.0771 - val_loss: 1350.2199
Epoch 2/1000
298/298 - 0s - loss: 693.0042 - val_loss: 360.4646
Epoch 3/1000
298/298 - 0s - loss: 400.9638 - val_loss: 163.4880
Epoch 4/1000
298/298 - 0s - loss: 242.4459 - val_loss: 257.8002
Epoch 5/1000
298/298 - 0s - loss: 235.7542 - val_loss: 155.4853
Epoch 6/1000
298/298 - 0s - loss: 198.7946 - val_loss: 136.8882
Epoch 7/1000
298/298 - 0s - loss: 188.2564 - val_loss: 136.4962
Epoch 8/1000
298/298 - 0s - loss: 182.8997 - val_loss: 149.1577
Epoch 9/1000
298/298 - 0s - loss: 207.1337 - val_loss: 137.4306
Epoch 10/1000
298/298 - 0s - loss: 203.3414 - val_loss: 155.6793
Epoch 11/1000
298/298 - 0s - loss: 176.8696 - val_loss: 132.2497
Epoch 12/1000
298/298 - 0s - loss: 177.1177 - val_loss: 147.5683
Epoch 13/1000
298/298 - 0s - loss: 204.6960 - val_loss: 226.5532
Epoch 14/1000
```

```
298/298 - 0s - loss: 232.8534 - val_loss: 179.7078
Epoch 15/1000
298/298 - 0s - loss: 185.6255 - val_loss: 126.6233
Epoch 16/1000
298/298 - 0s - loss: 162.3117 - val_loss: 123.3192
Epoch 17/1000
298/298 - 0s - loss: 157.3667 - val_loss: 120.5006
Epoch 18/1000
298/298 - 0s - loss: 157.9454 - val_loss: 138.6774
Epoch 19/1000
298/298 - 0s - loss: 160.7731 - val_loss: 142.5329
Epoch 20/1000
298/298 - 0s - loss: 178.6901 - val_loss: 121.7760
Epoch 21/1000
298/298 - 0s - loss: 152.5645 - val_loss: 129.7472
Epoch 22/1000
298/298 - 0s - loss: 163.4855 - val_loss: 152.6662
Epoch 00022: early stopping
```

Out[13]:
```
<tensorflow.python.keras.callbacks.History at 0x2a9e025e488>
```

Mean Square Error

The mean square error 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. Low MSE values are desired.

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

In [14]:
```
# 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): 152.66622924804688
```

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. Low RMSE values are desired.

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

In [15]:
```
# 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): 12.355817794799805
```

Performance Improvement by Normalizing Features and Tuning Hyperparameters

There are many different settings that you can use for a neural network. These can affect performance. The following code changes some of these, beyond their default values:

- **activation:** relu, sigmoid, tanh
- **Layers and Neuron Counts**
- **optimizer:** adam, sgd, rmsprop, and others

In [16]:

```
import pandas as pd
import io
import requests
import numpy as np
import os
from sklearn.model_selection import train_test_split
from sklearn import metrics
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.callbacks import ModelCheckpoint

path = "./data/"
preprocess = True

filename_read = os.path.join(path,"auto-mpg.csv")
df = pd.read_csv(filename_read,na_values=['NA','?'])

# create feature vector
missing_median(df, 'horsepower')
encode_text_dummy(df, 'origin')
df.drop('name',1,inplace=True)

if preprocess:
    encode_numeric_zscore(df, 'horsepower')
    encode_numeric_zscore(df, 'weight')
    encode_numeric_zscore(df, 'cylinders')
    encode_numeric_zscore(df, 'displacement')
    encode_numeric_zscore(df, 'acceleration')
    encode_numeric_zscore(df, 'year')

# Encode to a 2D matrix for training
x,y = to_xy(df,'mpg')

# Split into train/test
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.20, random_state=

model = Sequential()
model.add(Dense(100, input_dim=x.shape[1], activation='relu'))
model.add(Dense(50, activation='relu'))
model.add(Dense(25, 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=2, mode

model.fit(x_train,y_train,validation_data=(x_test,y_test),callbacks=[monitor],verbose=2
```

```
Train on 318 samples, validate on 80 samples
```

```
Epoch 1/1000
318/318 - 1s - loss: 601.0686 - val_loss: 553.4452
Epoch 2/1000
318/318 - 0s - loss: 565.9083 - val_loss: 512.0234
Epoch 3/1000
318/318 - 0s - loss: 515.6146 - val_loss: 450.2371
Epoch 4/1000
318/318 - 0s - loss: 438.0986 - val_loss: 355.0346
Epoch 5/1000
318/318 - 0s - loss: 320.4278 - val_loss: 222.3230
Epoch 6/1000
318/318 - 0s - loss: 176.3422 - val_loss: 85.3547
Epoch 7/1000
318/318 - 0s - loss: 55.0570 - val_loss: 27.5880
Epoch 8/1000
318/318 - 0s - loss: 28.9309 - val_loss: 32.4941
Epoch 9/1000
318/318 - 0s - loss: 26.6477 - val_loss: 20.9192
Epoch 10/1000
318/318 - 0s - loss: 19.5922 - val_loss: 16.9989
Epoch 11/1000
318/318 - 0s - loss: 17.7273 - val_loss: 14.8337
Epoch 12/1000
318/318 - 0s - loss: 15.6191 - val_loss: 13.2944
Epoch 13/1000
318/318 - 0s - loss: 14.3838 - val_loss: 11.8893
Epoch 14/1000
318/318 - 0s - loss: 13.4566 - val_loss: 10.6609
Epoch 15/1000
318/318 - 0s - loss: 12.5942 - val_loss: 9.7406
Epoch 16/1000
318/318 - 0s - loss: 11.9475 - val_loss: 9.1340
Epoch 17/1000
318/318 - 0s - loss: 11.4326 - val_loss: 8.3586
Epoch 18/1000
318/318 - 0s - loss: 10.9700 - val_loss: 7.8489
Epoch 19/1000
318/318 - 0s - loss: 10.6577 - val_loss: 7.3039
Epoch 20/1000
318/318 - 0s - loss: 10.3515 - val_loss: 7.2740
Epoch 21/1000
318/318 - 0s - loss: 9.9101 - val_loss: 6.9260
Epoch 22/1000
318/318 - 0s - loss: 9.6372 - val_loss: 6.5449
Epoch 23/1000
318/318 - 0s - loss: 9.4455 - val_loss: 6.3419
Epoch 24/1000
318/318 - 0s - loss: 9.1688 - val_loss: 6.2565
Epoch 25/1000
318/318 - 0s - loss: 8.9981 - val_loss: 6.0536
Epoch 26/1000
318/318 - 0s - loss: 8.8751 - val_loss: 5.9385
Epoch 27/1000
318/318 - 0s - loss: 8.6403 - val_loss: 5.8216
Epoch 28/1000
318/318 - 0s - loss: 8.5033 - val_loss: 5.6967
Epoch 29/1000
318/318 - 0s - loss: 8.4988 - val_loss: 5.6640
Epoch 30/1000
318/318 - 0s - loss: 8.2294 - val_loss: 5.5286
Epoch 31/1000
318/318 - 0s - loss: 8.1688 - val_loss: 5.4936
Epoch 32/1000
318/318 - 0s - loss: 8.0905 - val_loss: 5.4366
Epoch 33/1000
```

```
318/318 - 0s - loss: 7.9708 - val_loss: 5.5161
Epoch 34/1000
318/318 - 0s - loss: 7.8841 - val_loss: 5.3118
Epoch 35/1000
318/318 - 0s - loss: 7.7685 - val_loss: 5.4275
Epoch 36/1000
318/318 - 0s - loss: 7.7268 - val_loss: 5.3087
Epoch 37/1000
318/318 - 0s - loss: 7.7221 - val_loss: 5.3711
Epoch 38/1000
318/318 - 0s - loss: 7.6125 - val_loss: 5.3627
Epoch 39/1000
318/318 - 0s - loss: 7.5003 - val_loss: 5.2197
Epoch 40/1000
318/318 - 0s - loss: 7.5055 - val_loss: 5.1032
Epoch 41/1000
318/318 - 0s - loss: 7.4188 - val_loss: 5.3052
Epoch 42/1000
318/318 - 0s - loss: 7.3521 - val_loss: 5.1058
Epoch 43/1000
318/318 - 0s - loss: 7.3433 - val_loss: 5.0612
Epoch 44/1000
318/318 - 0s - loss: 7.2301 - val_loss: 5.2147
Epoch 45/1000
318/318 - 0s - loss: 7.2460 - val_loss: 5.1271
Epoch 46/1000
318/318 - 0s - loss: 7.1538 - val_loss: 5.1805
Epoch 47/1000
318/318 - 0s - loss: 7.1032 - val_loss: 5.1091
Epoch 48/1000
318/318 - 0s - loss: 7.1453 - val_loss: 5.1802
Epoch 00048: early stopping
```

Out[16]:
```
<tensorflow.python.keras.callbacks.History at 0x2a9e5a71788>
```

In [17]:
```
# Predict and measure RMSE
pred = model.predict(x_test)
score = np.sqrt(metrics.mean_squared_error(pred,y_test))
print("Score (RMSE): {}".format(score))
```

```
Score (RMSE): 2.2759974002838135
```

In [18]:
```
# print out prediction
df_y = pd.DataFrame(y_test, columns=['ground_truth'])
df_pred = pd.DataFrame(pred, columns=['predicted'])
result = pd.concat([df_y, df_pred],axis=1)
result
```

Out[18]:

	ground_truth	predicted
0	33.0	33.709663
1	28.0	31.442051
2	19.0	20.359224
3	13.0	15.257220
4	14.0	13.430858
...	...	...

	ground_truth	predicted
75	19.9	21.596952
76	17.5	17.525774
77	28.0	31.057686
78	29.0	30.164536
79	17.0	18.622261

80 rows × 2 columns

In [ ]:

CSC 180 Intelligenct Systems (Fall 2022)

Dr. Haiquan Chen, Dept of Computer Scicence

California State University, Sacramento

Lab 7: Model Performance Visualization

Helpful Functions for Tensorflow

The following functions will be used with TensorFlow to help preprocess the data. They allow you to build the feature vector for a neural network.

- Predictors/Inputs
  - Fill any missing inputs with the median for that column. Use **missing_median**.
  - Encode textual/categorical values with **encode_text_dummy**.
  - Encode numeric values with **encode_numeric_zscore**.
- Output
  - Discard rows with missing outputs.
  - Encode textual/categorical values with **encode_text_index**.
  - Do not encode output numeric values.
- Produce final feature vectors (x) and expected output (y) with **to_xy**.

In [1]:

```
from collections.abc import Sequence
from sklearn import preprocessing
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import shutil
import os


# Encode text values to dummy variables(i.e. [1,0,0],[0,1,0],[0,0,1] for red,green,blue
def encode_text_dummy(df, name):
    dummies = pd.get_dummies(df[name])
    for x in dummies.columns:
        dummy_name = "{}-{}".format(name, x)
        df[dummy_name] = dummies[x]
    df.drop(name, axis=1, inplace=True)



# Encode text values to indexes(i.e. [1],[2],[3] for red,green,blue).
def encode_text_index(df, name):
    le = preprocessing.LabelEncoder()
    df[name] = le.fit_transform(df[name])
    return le.classes_


# Encode a numeric column as zscores
def encode_numeric_zscore(df, name, mean=None, sd=None):
```

```
    if mean is None:
        mean = df[name].mean()

    if sd is None:
        sd = df[name].std()

    df[name] = (df[name] - mean) / sd


# Convert all missing values in the specified column to the median
def missing_median(df, name):
    med = df[name].median()
    df[name] = df[name].fillna(med)


# Convert all missing values in the specified column to the default
def missing_default(df, name, default_value):
    df[name] = df[name].fillna(default_value)


# Convert a Pandas dataframe to the x,y inputs that TensorFlow needs
def to_xy(df, target):
    result = []
    for x in df.columns:
        if x != target:
            result.append(x)
    # find out the type of the target column.
    target_type = df[target].dtypes
    target_type = target_type[0] if isinstance(target_type, Sequence) else target_type
    # Encode to int for classification, float otherwise. TensorFlow likes 32 bits.
    if target_type in (np.int64, np.int32):
        # Classification
        dummies = pd.get_dummies(df[target])
        return df[result].values.astype(np.float32), dummies.values.astype(np.float32)
    else:
        # Regression
        return df[result].values.astype(np.float32), df[target].values.astype(np.float3

# 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)


# 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)
    a = plt.plot(t['y'].tolist(),label='expected')
    b = plt.plot(t['pred'].tolist(),label='prediction')
    plt.ylabel('output')
    plt.legend()
    plt.show()

# 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)


# Encode a column to a range between normalized_low and normalized_high.
def encode_numeric_range(df, name, normalized_low=-1, normalized_high=1,
                         data_low=None, data_high=None):
    if data_low is None:
        data_low = min(df[name])
        data_high = max(df[name])

    df[name] = ((df[name] - data_low) / (data_high - data_low)) \
               * (normalized_high - normalized_low) + normalized_low
```

Metrics:

For any type of classification neural network:

- **Accuracy**
- **Precision**
- **Recall**
- **F1 Score**
- **Log Loss**
- **Confusion Matrix**
- **ROC Curve**

For regression neural networks:

- **RMSE**
- **R2 Score**
- **Lift Chart**

Check Sklearn API here http://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics

The code used to produce Confusion matrix and ROC curve is shown here:

In [2]:
```
%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, thresholds = 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()
```

Visualizing binary classification models

Binary classification is used to create a model that classifies between only two classes. These two classes are often called "positive" and "negative".

In [3]:
```
import os
import pandas as pd
from sklearn.model_selection import train_test_split
import tensorflow as tf
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 tensorflow.keras.callbacks import ModelCheckpoint

# Set the desired TensorFlow output level for this example
# tf.logging.set_verbosity(tf.logging.ERROR)

path = "./data/"

filename = os.path.join(path,"wcbreast_wdbc.csv")
df = pd.read_csv(filename,na_values=['NA','?'])
df
```

Out[3]:

	id	diagnosis	mean_radius	mean_texture	mean_perimeter	mean_area	mean_smoothness
0	842302	M	17.99	10.38	122.80	1001.0	0.11840
1	842517	M	20.57	17.77	132.90	1326.0	0.08474
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960
3	84348301	M	11.42	20.38	77.58	386.1	0.14250
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030
...	...	...	...	...	...	...	...

	id	diagnosis	mean_radius	mean_texture	mean_perimeter	mean_area	mean_smoothness
564	926424	M	21.56	22.39	142.00	1479.0	0.11100
565	926682	M	20.13	28.25	131.20	1261.0	0.09780
566	926954	M	16.60	28.08	108.30	858.1	0.08455
567	927241	M	20.60	29.33	140.10	1265.0	0.11780
568	92751	B	7.76	24.54	47.92	181.0	0.05263

569 rows × 32 columns

In [4]:

```
# Encode feature vector
df.drop('id',axis=1,inplace=True)

diagnosis = encode_text_index(df,'diagnosis')

# Create x & y for training and testing
x, y = to_xy(df,'diagnosis')

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

Avoid local optimum: Jump out of local optimum by using a loop!!

In [5]:

```
# Define ModelCheckpoint outside the loop
checkpointer = ModelCheckpoint(filepath="dnn/best_weights.hdf5", verbose=0, save_best_o

for i in range(5):
    print(i)

    # Build network
    model = Sequential()
    model.add(Dense(20, input_dim=x.shape[1], activation='relu'))
    model.add(Dense(10, activation='relu'))
    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,

    model.fit(x_train,y_train,validation_data=(x_test,y_test),callbacks=[monitor,checkp


print('Training finished...Loading the best model')
print()
model.load_weights('dnn/best_weights.hdf5') # load weights from best model

# Measure accuracy
pred = model.predict(x_test)
pred = np.argmax(pred,axis=1)

y_true = np.argmax(y_test,axis=1)
```

```
score = metrics.accuracy_score(y_true, pred)
print("Final accuracy: {}".format(score))
```

```
0
Train on 426 samples, validate on 143 samples
Epoch 1/100
426/426 - 1s - loss: 4.4151 - val_loss: 2.5804
Epoch 2/100
426/426 - 0s - loss: 1.5916 - val_loss: 0.6562
Epoch 3/100
426/426 - 0s - loss: 0.8587 - val_loss: 0.4273
Epoch 4/100
426/426 - 0s - loss: 0.7448 - val_loss: 0.3727
Epoch 5/100
426/426 - 0s - loss: 0.6496 - val_loss: 0.3541
Epoch 6/100
426/426 - 0s - loss: 0.5669 - val_loss: 0.3508
Epoch 7/100
426/426 - 0s - loss: 0.5552 - val_loss: 0.3231
Epoch 8/100
426/426 - 0s - loss: 0.5346 - val_loss: 0.3607
Epoch 9/100
426/426 - 0s - loss: 0.5671 - val_loss: 0.2962
Epoch 10/100
426/426 - 0s - loss: 0.5394 - val_loss: 0.3094
Epoch 11/100
426/426 - 0s - loss: 0.4770 - val_loss: 0.3632
Epoch 12/100
426/426 - 0s - loss: 0.4409 - val_loss: 0.2749
Epoch 13/100
426/426 - 0s - loss: 0.4572 - val_loss: 0.3383
Epoch 14/100
426/426 - 0s - loss: 0.4456 - val_loss: 0.2441
Epoch 15/100
426/426 - 0s - loss: 0.4123 - val_loss: 0.2601
Epoch 16/100
426/426 - 0s - loss: 0.4430 - val_loss: 0.2919
Epoch 17/100
426/426 - 0s - loss: 0.3890 - val_loss: 0.2416
Epoch 18/100
426/426 - 0s - loss: 0.3851 - val_loss: 0.2531
Epoch 19/100
426/426 - 0s - loss: 0.3499 - val_loss: 0.2241
Epoch 20/100
426/426 - 0s - loss: 0.4549 - val_loss: 0.4044
Epoch 21/100
426/426 - 0s - loss: 0.4449 - val_loss: 0.3063
Epoch 22/100
426/426 - 0s - loss: 0.3711 - val_loss: 0.2237
Epoch 23/100
426/426 - 0s - loss: 0.3439 - val_loss: 0.2221
Epoch 24/100
426/426 - 0s - loss: 0.3588 - val_loss: 0.3490
Epoch 25/100
426/426 - 0s - loss: 0.4601 - val_loss: 0.2681
Epoch 26/100
426/426 - 0s - loss: 0.3695 - val_loss: 0.2140
Epoch 27/100
426/426 - 0s - loss: 0.3208 - val_loss: 0.1958
Epoch 28/100
426/426 - 0s - loss: 0.3237 - val_loss: 0.2346
Epoch 29/100
426/426 - 0s - loss: 0.3214 - val_loss: 0.2294
Epoch 30/100
```

```
426/426 - 0s - loss: 0.2949 - val_loss: 0.2323
Epoch 31/100
426/426 - 0s - loss: 0.3401 - val_loss: 0.1940
Epoch 32/100
426/426 - 0s - loss: 0.3119 - val_loss: 0.4313
Epoch 33/100
426/426 - 0s - loss: 0.3958 - val_loss: 0.2455
Epoch 34/100
426/426 - 0s - loss: 0.3496 - val_loss: 0.2481
Epoch 35/100
426/426 - 0s - loss: 0.3428 - val_loss: 0.3572
Epoch 36/100
426/426 - 0s - loss: 0.4322 - val_loss: 0.2398
Epoch 00036: early stopping
1
Train on 426 samples, validate on 143 samples
Epoch 1/100
426/426 - 1s - loss: 16.6402 - val_loss: 11.6081
Epoch 2/100
426/426 - 0s - loss: 9.9227 - val_loss: 10.1240
Epoch 3/100
426/426 - 0s - loss: 7.1303 - val_loss: 5.5062
Epoch 4/100
426/426 - 0s - loss: 4.2666 - val_loss: 3.2782
Epoch 5/100
426/426 - 0s - loss: 2.3395 - val_loss: 1.3534
Epoch 6/100
426/426 - 0s - loss: 1.1500 - val_loss: 0.7426
Epoch 7/100
426/426 - 0s - loss: 0.7474 - val_loss: 0.6337
Epoch 8/100
426/426 - 0s - loss: 0.6036 - val_loss: 0.5516
Epoch 9/100
426/426 - 0s - loss: 0.6552 - val_loss: 0.8535
Epoch 10/100
426/426 - 0s - loss: 0.6772 - val_loss: 0.3718
Epoch 11/100
426/426 - 0s - loss: 0.5342 - val_loss: 0.3379
Epoch 12/100
426/426 - 0s - loss: 0.5778 - val_loss: 0.3256
Epoch 13/100
426/426 - 0s - loss: 0.4513 - val_loss: 0.2943
Epoch 14/100
426/426 - 0s - loss: 0.4248 - val_loss: 0.2874
Epoch 15/100
426/426 - 0s - loss: 0.4353 - val_loss: 0.3231
Epoch 16/100
426/426 - 0s - loss: 0.4269 - val_loss: 0.2983
Epoch 17/100
426/426 - 0s - loss: 0.6471 - val_loss: 0.3370
Epoch 18/100
426/426 - 0s - loss: 0.5199 - val_loss: 0.3309
Epoch 19/100
426/426 - 0s - loss: 0.5517 - val_loss: 0.2594
Epoch 20/100
426/426 - 0s - loss: 0.6355 - val_loss: 0.5331
Epoch 21/100
426/426 - 0s - loss: 0.6806 - val_loss: 0.2391
Epoch 22/100
426/426 - 0s - loss: 0.4212 - val_loss: 0.3326
Epoch 23/100
426/426 - 0s - loss: 0.4243 - val_loss: 0.2575
Epoch 24/100
426/426 - 0s - loss: 0.4316 - val_loss: 0.2402
Epoch 25/100
```

```
426/426 - 0s - loss: 0.3672 - val_loss: 0.2199
Epoch 26/100
426/426 - 0s - loss: 0.3593 - val_loss: 0.2109
Epoch 27/100
426/426 - 0s - loss: 0.3707 - val_loss: 0.2374
Epoch 28/100
426/426 - 0s - loss: 0.3349 - val_loss: 0.2067
Epoch 29/100
426/426 - 0s - loss: 0.3367 - val_loss: 0.2659
Epoch 30/100
426/426 - 0s - loss: 0.5170 - val_loss: 0.2347
Epoch 31/100
426/426 - 0s - loss: 0.3905 - val_loss: 0.2017
Epoch 32/100
426/426 - 0s - loss: 0.3765 - val_loss: 0.2100
Epoch 33/100
426/426 - 0s - loss: 0.3345 - val_loss: 0.5103
Epoch 34/100
426/426 - 0s - loss: 0.4328 - val_loss: 0.2037
Epoch 35/100
426/426 - 0s - loss: 0.4490 - val_loss: 0.4133
Epoch 36/100
426/426 - 0s - loss: 0.3916 - val_loss: 0.2162
Epoch 00036: early stopping
2
Train on 426 samples, validate on 143 samples
Epoch 1/100
426/426 - 1s - loss: 140.0453 - val_loss: 84.1908
Epoch 2/100
426/426 - 0s - loss: 37.3763 - val_loss: 23.7865
Epoch 3/100
426/426 - 0s - loss: 15.8859 - val_loss: 5.0944
Epoch 4/100
426/426 - 0s - loss: 5.5456 - val_loss: 2.7180
Epoch 5/100
426/426 - 0s - loss: 3.8559 - val_loss: 2.0681
Epoch 6/100
426/426 - 0s - loss: 3.3988 - val_loss: 1.5430
Epoch 7/100
426/426 - 0s - loss: 2.9946 - val_loss: 1.2967
Epoch 8/100
426/426 - 0s - loss: 2.6914 - val_loss: 1.2433
Epoch 9/100
426/426 - 0s - loss: 2.4643 - val_loss: 0.9970
Epoch 10/100
426/426 - 0s - loss: 2.3429 - val_loss: 1.3227
Epoch 11/100
426/426 - 0s - loss: 2.1003 - val_loss: 0.8048
Epoch 12/100
426/426 - 0s - loss: 1.9858 - val_loss: 0.7849
Epoch 13/100
426/426 - 0s - loss: 1.7708 - val_loss: 0.6896
Epoch 14/100
426/426 - 0s - loss: 1.7347 - val_loss: 0.7031
Epoch 15/100
426/426 - 0s - loss: 1.4962 - val_loss: 0.5859
Epoch 16/100
426/426 - 0s - loss: 1.4093 - val_loss: 0.5634
Epoch 17/100
426/426 - 0s - loss: 1.3626 - val_loss: 1.1728
Epoch 18/100
426/426 - 0s - loss: 1.6747 - val_loss: 0.7019
Epoch 19/100
426/426 - 0s - loss: 1.2334 - val_loss: 0.9939
Epoch 20/100
```

```
426/426 - 0s - loss: 1.5506 - val_loss: 0.4691
Epoch 21/100
426/426 - 0s - loss: 1.1359 - val_loss: 0.5004
Epoch 22/100
426/426 - 0s - loss: 1.0159 - val_loss: 0.5254
Epoch 23/100
426/426 - 0s - loss: 1.0580 - val_loss: 0.4112
Epoch 24/100
426/426 - 0s - loss: 0.7844 - val_loss: 0.3480
Epoch 25/100
426/426 - 0s - loss: 0.7714 - val_loss: 0.3511
Epoch 26/100
426/426 - 0s - loss: 0.7594 - val_loss: 0.3750
Epoch 27/100
426/426 - 0s - loss: 0.8030 - val_loss: 0.3395
Epoch 28/100
426/426 - 0s - loss: 0.9757 - val_loss: 0.3042
Epoch 29/100
426/426 - 0s - loss: 1.2164 - val_loss: 0.3895
Epoch 30/100
426/426 - 0s - loss: 1.1253 - val_loss: 0.5347
Epoch 31/100
426/426 - 0s - loss: 0.6765 - val_loss: 0.5635
Epoch 32/100
426/426 - 0s - loss: 0.7655 - val_loss: 0.3442
Epoch 33/100
426/426 - 0s - loss: 0.6530 - val_loss: 0.3834
Epoch 00033: early stopping
3
Train on 426 samples, validate on 143 samples
Epoch 1/100
426/426 - 1s - loss: 51.5292 - val_loss: 44.5696
Epoch 2/100
426/426 - 0s - loss: 27.7306 - val_loss: 18.7962
Epoch 3/100
426/426 - 0s - loss: 13.3844 - val_loss: 9.4738
Epoch 4/100
426/426 - 0s - loss: 6.3632 - val_loss: 6.7713
Epoch 5/100
426/426 - 0s - loss: 4.1523 - val_loss: 2.1981
Epoch 6/100
426/426 - 0s - loss: 1.9751 - val_loss: 0.9571
Epoch 7/100
426/426 - 0s - loss: 1.2544 - val_loss: 0.8941
Epoch 8/100
426/426 - 0s - loss: 1.0736 - val_loss: 0.8026
Epoch 9/100
426/426 - 0s - loss: 1.1392 - val_loss: 0.5592
Epoch 10/100
426/426 - 0s - loss: 0.8205 - val_loss: 0.5856
Epoch 11/100
426/426 - 0s - loss: 0.7939 - val_loss: 0.5245
Epoch 12/100
426/426 - 0s - loss: 0.7357 - val_loss: 0.5183
Epoch 13/100
426/426 - 0s - loss: 0.7112 - val_loss: 0.4966
Epoch 14/100
426/426 - 0s - loss: 0.6484 - val_loss: 0.4588
Epoch 15/100
426/426 - 0s - loss: 0.6565 - val_loss: 0.4486
Epoch 16/100
426/426 - 0s - loss: 0.6317 - val_loss: 0.5389
Epoch 17/100
426/426 - 0s - loss: 0.6735 - val_loss: 0.3908
Epoch 18/100
```

```
426/426 - 0s - loss: 0.5898 - val_loss: 0.3850
Epoch 19/100
426/426 - 0s - loss: 0.6077 - val_loss: 0.4060
Epoch 20/100
426/426 - 0s - loss: 0.7253 - val_loss: 0.3749
Epoch 21/100
426/426 - 0s - loss: 0.8737 - val_loss: 0.4253
Epoch 22/100
426/426 - 0s - loss: 0.6678 - val_loss: 0.3436
Epoch 23/100
426/426 - 0s - loss: 0.5993 - val_loss: 0.6804
Epoch 24/100
426/426 - 0s - loss: 0.9390 - val_loss: 0.3372
Epoch 25/100
426/426 - 0s - loss: 1.4522 - val_loss: 0.7979
Epoch 26/100
426/426 - 0s - loss: 0.8561 - val_loss: 0.4104
Epoch 27/100
426/426 - 0s - loss: 0.5829 - val_loss: 0.3902
Epoch 28/100
426/426 - 0s - loss: 0.5834 - val_loss: 0.2788
Epoch 29/100
426/426 - 0s - loss: 0.5876 - val_loss: 0.3040
Epoch 30/100
426/426 - 0s - loss: 0.5332 - val_loss: 0.2728
Epoch 31/100
426/426 - 0s - loss: 0.5034 - val_loss: 0.3641
Epoch 32/100
426/426 - 0s - loss: 0.8005 - val_loss: 0.4341
Epoch 33/100
426/426 - 0s - loss: 0.7290 - val_loss: 0.2803
Epoch 34/100
426/426 - 0s - loss: 0.9187 - val_loss: 0.7626
Epoch 35/100
426/426 - 0s - loss: 0.7343 - val_loss: 0.3471
Epoch 00035: early stopping
4
Train on 426 samples, validate on 143 samples
Epoch 1/100
426/426 - 1s - loss: 20.1441 - val_loss: 11.3783
Epoch 2/100
426/426 - 0s - loss: 5.3454 - val_loss: 1.7324
Epoch 3/100
426/426 - 0s - loss: 2.1027 - val_loss: 0.9286
Epoch 4/100
426/426 - 0s - loss: 1.2160 - val_loss: 0.8796
Epoch 5/100
426/426 - 0s - loss: 1.1174 - val_loss: 0.8843
Epoch 6/100
426/426 - 0s - loss: 1.2031 - val_loss: 0.6544
Epoch 7/100
426/426 - 0s - loss: 1.0960 - val_loss: 0.6334
Epoch 8/100
426/426 - 0s - loss: 1.0578 - val_loss: 0.5880
Epoch 9/100
426/426 - 0s - loss: 0.9772 - val_loss: 0.5805
Epoch 10/100
426/426 - 0s - loss: 0.8840 - val_loss: 0.5547
Epoch 11/100
426/426 - 0s - loss: 0.8803 - val_loss: 0.5242
Epoch 12/100
426/426 - 0s - loss: 0.8443 - val_loss: 0.5241
Epoch 13/100
426/426 - 0s - loss: 0.8515 - val_loss: 0.4916
Epoch 14/100
```

```
426/426 - 0s - loss: 0.7795 - val_loss: 0.4772
Epoch 15/100
426/426 - 0s - loss: 0.7708 - val_loss: 0.4941
Epoch 16/100
426/426 - 0s - loss: 0.7781 - val_loss: 0.4695
Epoch 17/100
426/426 - 0s - loss: 0.7871 - val_loss: 0.5474
Epoch 18/100
426/426 - 0s - loss: 0.6994 - val_loss: 0.4316
Epoch 19/100
426/426 - 0s - loss: 0.6408 - val_loss: 0.4250
Epoch 20/100
426/426 - 0s - loss: 0.6661 - val_loss: 0.4016
Epoch 21/100
426/426 - 0s - loss: 0.6245 - val_loss: 0.4140
Epoch 22/100
426/426 - 0s - loss: 0.6878 - val_loss: 0.5130
Epoch 23/100
426/426 - 0s - loss: 0.6285 - val_loss: 0.6703
Epoch 24/100
426/426 - 0s - loss: 0.6063 - val_loss: 0.3994
Epoch 25/100
426/426 - 0s - loss: 0.5453 - val_loss: 0.3650
Epoch 26/100
426/426 - 0s - loss: 0.4991 - val_loss: 0.4949
Epoch 27/100
426/426 - 0s - loss: 0.6014 - val_loss: 0.4418
Epoch 28/100
426/426 - 0s - loss: 0.5104 - val_loss: 0.4087
Epoch 29/100
426/426 - 0s - loss: 0.5993 - val_loss: 0.3626
Epoch 30/100
426/426 - 0s - loss: 0.5133 - val_loss: 0.4179
Epoch 31/100
426/426 - 0s - loss: 0.6985 - val_loss: 0.3556
Epoch 32/100
426/426 - 0s - loss: 0.7182 - val_loss: 0.2826
Epoch 33/100
426/426 - 0s - loss: 0.5304 - val_loss: 0.2922
Epoch 34/100
426/426 - 0s - loss: 0.4624 - val_loss: 0.6323
Epoch 35/100
426/426 - 0s - loss: 0.6950 - val_loss: 0.2824
Epoch 36/100
426/426 - 0s - loss: 0.4784 - val_loss: 0.2396
Epoch 37/100
426/426 - 0s - loss: 0.4140 - val_loss: 0.2361
Epoch 38/100
426/426 - 0s - loss: 0.3663 - val_loss: 0.4633
Epoch 39/100
426/426 - 0s - loss: 0.5574 - val_loss: 0.3524
Epoch 40/100
426/426 - 0s - loss: 0.4447 - val_loss: 0.3057
Epoch 41/100
426/426 - 0s - loss: 0.4195 - val_loss: 0.3352
Epoch 42/100
426/426 - 0s - loss: 0.4791 - val_loss: 0.3881
Epoch 00042: early stopping
Training finished...Loading the best model

Final accuracy: 0.9300699300699301
```

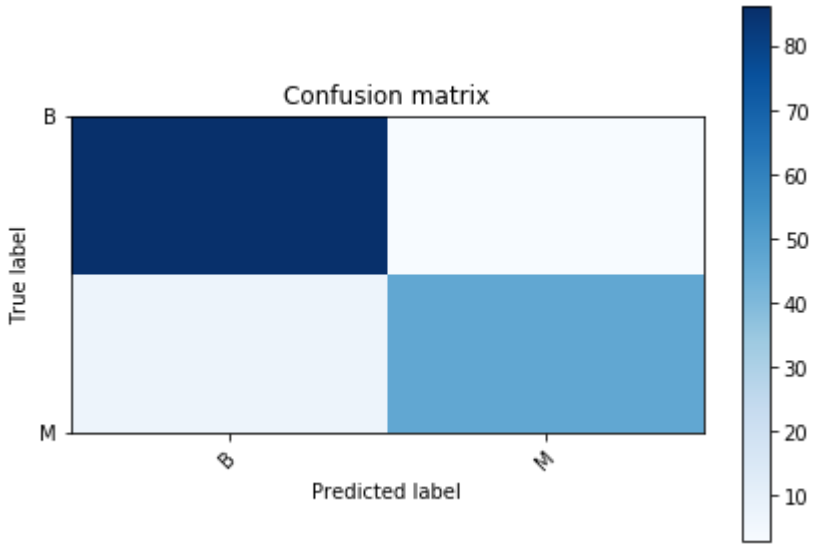
Confusion Matrix

The confusion matrix is a common visualization for both binary and multi-class classification problems. The following code generates a confusion matrix:

In [6]:

```
import numpy as np

from sklearn import svm, datasets
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report

# Compute confusion matrix
cm = confusion_matrix(y_true, pred)
print(cm)


print('Plotting confusion matrix')

plt.figure()
plot_confusion_matrix(cm, diagnosis)
plt.show()

print(classification_report(y_true, pred))
```

```
[[86  3]
 [ 7 47]]
Plotting confusion matrix
```



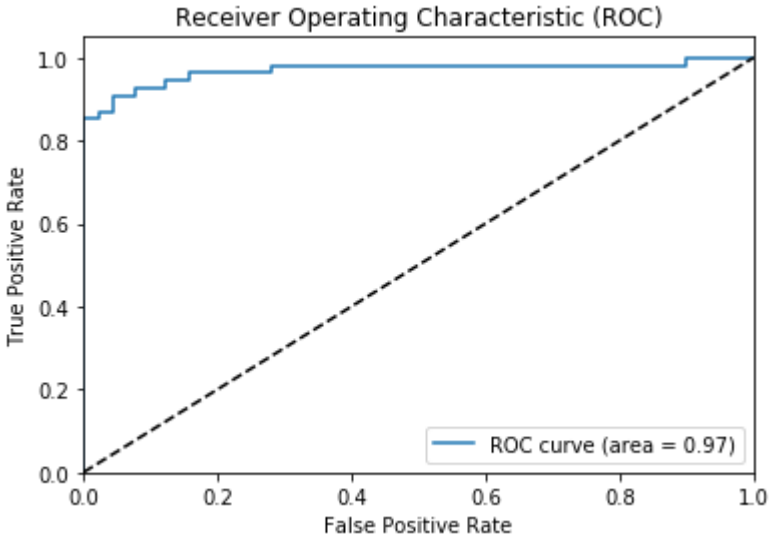

```
              precision    recall  f1-score   support

           0       0.92      0.97      0.95        89
           1       0.94      0.87      0.90        54

    accuracy                           0.93       143
   macro avg       0.93      0.92      0.92       143
weighted avg       0.93      0.93      0.93       143
```

ROC Curves

An ROC curve measures how good a model is regardless of the cutoff.

The following code shows an ROC chart for the breast cancer neural network. The area under the curve (AUC) is also an important measure. The larger the AUC, the better.

In [7]:
```
pred = model.predict(x_test)
pred = pred[:,1] # Only positive class (M)
plot_roc(pred,y_true)
```



Visualizing multi-class classification models

We've already seen multi-class classification, with the iris dataset. Confusion matrixes work just fine with 3 classes. The following code generates a confusion matrix for iris.

In [8]:
```
import pandas as pd
import io
import requests
import numpy as np
import os
from sklearn.model_selection import train_test_split
from sklearn import metrics
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.callbacks import ModelCheckpoint

path = "./data/"

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

species = encode_text_index(df,"species")
x,y = to_xy(df,"species")

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

model = Sequential()
model.add(Dense(20, input_dim=x.shape[1], activation='relu'))
```

```
model.add(Dense(10))
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=0, mode
checkpointer = ModelCheckpoint(filepath="dnn/best_weights.hdf5", verbose=0, save_best_o

model.fit(x_train,y_train,validation_data=(x_test,y_test),callbacks=[monitor,checkpoint

model.load_weights('dnn/best_weights.hdf5') # load weights from best model
```

```
Train on 112 samples, validate on 38 samples
Epoch 1/100
112/112 - 1s - loss: 1.9001 - val_loss: 1.5804
Epoch 2/100
112/112 - 0s - loss: 1.6818 - val_loss: 1.4099
Epoch 3/100
112/112 - 0s - loss: 1.4953 - val_loss: 1.2737
Epoch 4/100
112/112 - 0s - loss: 1.3389 - val_loss: 1.1674
Epoch 5/100
112/112 - 0s - loss: 1.2259 - val_loss: 1.0840
Epoch 6/100
112/112 - 0s - loss: 1.1251 - val_loss: 1.0211
Epoch 7/100
112/112 - 0s - loss: 1.0536 - val_loss: 0.9721
Epoch 8/100
112/112 - 0s - loss: 0.9980 - val_loss: 0.9367
Epoch 9/100
112/112 - 0s - loss: 0.9557 - val_loss: 0.9131
Epoch 10/100
112/112 - 0s - loss: 0.9270 - val_loss: 0.8982
Epoch 11/100
112/112 - 0s - loss: 0.9063 - val_loss: 0.8870
Epoch 12/100
112/112 - 0s - loss: 0.8933 - val_loss: 0.8764
Epoch 13/100
112/112 - 0s - loss: 0.8808 - val_loss: 0.8637
Epoch 14/100
112/112 - 0s - loss: 0.8681 - val_loss: 0.8491
Epoch 15/100
112/112 - 0s - loss: 0.8539 - val_loss: 0.8320
Epoch 16/100
112/112 - 0s - loss: 0.8388 - val_loss: 0.8145
Epoch 17/100
112/112 - 0s - loss: 0.8231 - val_loss: 0.7977
Epoch 18/100
112/112 - 0s - loss: 0.8082 - val_loss: 0.7803
Epoch 19/100
112/112 - 0s - loss: 0.7930 - val_loss: 0.7633
Epoch 20/100
112/112 - 0s - loss: 0.7788 - val_loss: 0.7457
Epoch 21/100
112/112 - 0s - loss: 0.7642 - val_loss: 0.7290
Epoch 22/100
112/112 - 0s - loss: 0.7503 - val_loss: 0.7131
Epoch 23/100
112/112 - 0s - loss: 0.7362 - val_loss: 0.6980
Epoch 24/100
112/112 - 0s - loss: 0.7228 - val_loss: 0.6830
Epoch 25/100
112/112 - 0s - loss: 0.7096 - val_loss: 0.6680
Epoch 26/100
112/112 - 0s - loss: 0.6966 - val_loss: 0.6536
```

```
Epoch 27/100
112/112 - 0s - loss: 0.6841 - val_loss: 0.6405
Epoch 28/100
112/112 - 0s - loss: 0.6715 - val_loss: 0.6279
Epoch 29/100
112/112 - 0s - loss: 0.6606 - val_loss: 0.6152
Epoch 30/100
112/112 - 0s - loss: 0.6485 - val_loss: 0.6039
Epoch 31/100
112/112 - 0s - loss: 0.6378 - val_loss: 0.5931
Epoch 32/100
112/112 - 0s - loss: 0.6276 - val_loss: 0.5819
Epoch 33/100
112/112 - 0s - loss: 0.6180 - val_loss: 0.5717
Epoch 34/100
112/112 - 0s - loss: 0.6082 - val_loss: 0.5620
Epoch 35/100
112/112 - 0s - loss: 0.5992 - val_loss: 0.5521
Epoch 36/100
112/112 - 0s - loss: 0.5912 - val_loss: 0.5430
Epoch 37/100
112/112 - 0s - loss: 0.5818 - val_loss: 0.5339
Epoch 38/100
112/112 - 0s - loss: 0.5753 - val_loss: 0.5256
Epoch 39/100
112/112 - 0s - loss: 0.5671 - val_loss: 0.5172
Epoch 40/100
112/112 - 0s - loss: 0.5589 - val_loss: 0.5094
Epoch 41/100
112/112 - 0s - loss: 0.5514 - val_loss: 0.5021
Epoch 42/100
112/112 - 0s - loss: 0.5439 - val_loss: 0.4948
Epoch 43/100
112/112 - 0s - loss: 0.5373 - val_loss: 0.4878
Epoch 44/100
112/112 - 1s - loss: 0.5299 - val_loss: 0.4812
Epoch 45/100
112/112 - 1s - loss: 0.5235 - val_loss: 0.4748
Epoch 46/100
112/112 - 0s - loss: 0.5171 - val_loss: 0.4685
Epoch 47/100
112/112 - 0s - loss: 0.5115 - val_loss: 0.4625
Epoch 48/100
112/112 - 0s - loss: 0.5044 - val_loss: 0.4551
Epoch 49/100
112/112 - 0s - loss: 0.4976 - val_loss: 0.4482
Epoch 50/100
112/112 - 0s - loss: 0.4927 - val_loss: 0.4419
Epoch 51/100
112/112 - 1s - loss: 0.4867 - val_loss: 0.4358
Epoch 52/100
112/112 - 1s - loss: 0.4810 - val_loss: 0.4299
Epoch 53/100
112/112 - 0s - loss: 0.4741 - val_loss: 0.4246
Epoch 54/100
112/112 - 0s - loss: 0.4687 - val_loss: 0.4198
Epoch 55/100
112/112 - 0s - loss: 0.4647 - val_loss: 0.4153
Epoch 56/100
112/112 - 0s - loss: 0.4590 - val_loss: 0.4094
Epoch 57/100
112/112 - 0s - loss: 0.4540 - val_loss: 0.4037
Epoch 58/100
112/112 - 0s - loss: 0.4480 - val_loss: 0.3980
Epoch 59/100
```

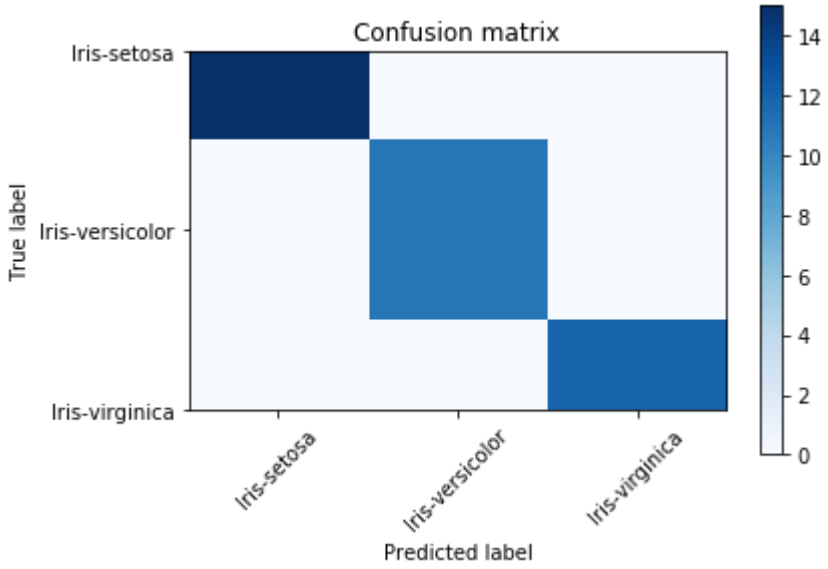
```
112/112 - 0s - loss: 0.4429 - val_loss: 0.3931
Epoch 60/100
112/112 - 0s - loss: 0.4389 - val_loss: 0.3883
Epoch 61/100
112/112 - 0s - loss: 0.4338 - val_loss: 0.3836
Epoch 62/100
112/112 - 0s - loss: 0.4289 - val_loss: 0.3792
Epoch 63/100
112/112 - 0s - loss: 0.4237 - val_loss: 0.3748
Epoch 64/100
112/112 - 0s - loss: 0.4191 - val_loss: 0.3705
Epoch 65/100
112/112 - 0s - loss: 0.4147 - val_loss: 0.3661
Epoch 66/100
112/112 - 0s - loss: 0.4105 - val_loss: 0.3622
Epoch 67/100
112/112 - 0s - loss: 0.4059 - val_loss: 0.3580
Epoch 68/100
112/112 - 0s - loss: 0.4013 - val_loss: 0.3536
Epoch 69/100
112/112 - 0s - loss: 0.3974 - val_loss: 0.3489
Epoch 70/100
112/112 - 0s - loss: 0.3925 - val_loss: 0.3449
Epoch 71/100
112/112 - 0s - loss: 0.3884 - val_loss: 0.3410
Epoch 72/100
112/112 - 0s - loss: 0.3849 - val_loss: 0.3375
Epoch 73/100
112/112 - 0s - loss: 0.3805 - val_loss: 0.3326
Epoch 74/100
112/112 - 0s - loss: 0.3759 - val_loss: 0.3285
Epoch 75/100
112/112 - 0s - loss: 0.3736 - val_loss: 0.3246
Epoch 76/100
112/112 - 0s - loss: 0.3685 - val_loss: 0.3207
Epoch 77/100
112/112 - 0s - loss: 0.3642 - val_loss: 0.3172
Epoch 78/100
112/112 - 0s - loss: 0.3601 - val_loss: 0.3141
Epoch 79/100
112/112 - 0s - loss: 0.3570 - val_loss: 0.3107
Epoch 80/100
112/112 - 0s - loss: 0.3531 - val_loss: 0.3065
Epoch 81/100
112/112 - 0s - loss: 0.3496 - val_loss: 0.3030
Epoch 82/100
112/112 - 0s - loss: 0.3465 - val_loss: 0.2993
Epoch 83/100
112/112 - 0s - loss: 0.3441 - val_loss: 0.2959
Epoch 84/100
112/112 - 0s - loss: 0.3373 - val_loss: 0.2935
Epoch 85/100
112/112 - 0s - loss: 0.3344 - val_loss: 0.2909
Epoch 86/100
112/112 - 0s - loss: 0.3315 - val_loss: 0.2867
Epoch 87/100
112/112 - 0s - loss: 0.3274 - val_loss: 0.2825
Epoch 88/100
112/112 - 0s - loss: 0.3236 - val_loss: 0.2789
Epoch 89/100
112/112 - 0s - loss: 0.3202 - val_loss: 0.2757
Epoch 90/100
112/112 - 0s - loss: 0.3189 - val_loss: 0.2734
Epoch 91/100
112/112 - 0s - loss: 0.3134 - val_loss: 0.2695
```

```
Epoch 92/100
112/112 - 0s - loss: 0.3100 - val_loss: 0.2662
Epoch 93/100
112/112 - 0s - loss: 0.3068 - val_loss: 0.2631
Epoch 94/100
112/112 - 0s - loss: 0.3032 - val_loss: 0.2603
Epoch 95/100
112/112 - 0s - loss: 0.3004 - val_loss: 0.2572
Epoch 96/100
112/112 - 0s - loss: 0.2974 - val_loss: 0.2537
Epoch 97/100
112/112 - 0s - loss: 0.2938 - val_loss: 0.2507
Epoch 98/100
112/112 - 0s - loss: 0.2901 - val_loss: 0.2478
Epoch 99/100
112/112 - 0s - loss: 0.2866 - val_loss: 0.2453
Epoch 100/100
112/112 - 0s - loss: 0.2841 - val_loss: 0.2431
```

In [9]:

```
import numpy as np

from sklearn import svm, datasets
from sklearn.metrics import confusion_matrix, classification_report

pred = model.predict(x_test)
pred = np.argmax(pred,axis=1)
y_true = np.argmax(y_test,axis=1)

# Compute confusion matrix
cm = confusion_matrix(y_true, pred)
print(cm)

print('Ploting confusion matrix')
plt.figure()
plot_confusion_matrix(cm, species)
plt.show()


print(classification_report(y_true, pred))
```

```
[[15  0  0]
 [ 0 11  0]
 [ 0  0 12]]
Ploting confusion matrix
```



	precision	recall	f1-score	support
0	1.00	1.00	1.00	15
1	1.00	1.00	1.00	11
2	1.00	1.00	1.00	12
accuracy			1.00	38
macro avg	1.00	1.00	1.00	38
weighted avg	1.00	1.00	1.00	38

Visualizing regression models

We've already seen regression with the MPG dataset. Regression uses its own set of visualizations, one of the most common is the lift chart. The following code generates a lift chart.

In [10]:

```
%matplotlib inline
from matplotlib.pyplot import figure, show
from sklearn.model_selection import train_test_split
import pandas as pd
import numpy as np
from sklearn import metrics
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.callbacks import ModelCheckpoint


path = "./data/"
preprocess = False

filename_read = os.path.join(path,"auto-mpg.csv")
df = pd.read_csv(filename_read,na_values=['NA','?'])

# create feature vector
missing_median(df, 'horsepower')
encode_text_dummy(df, 'origin')
df.drop('name',1,inplace=True)
```

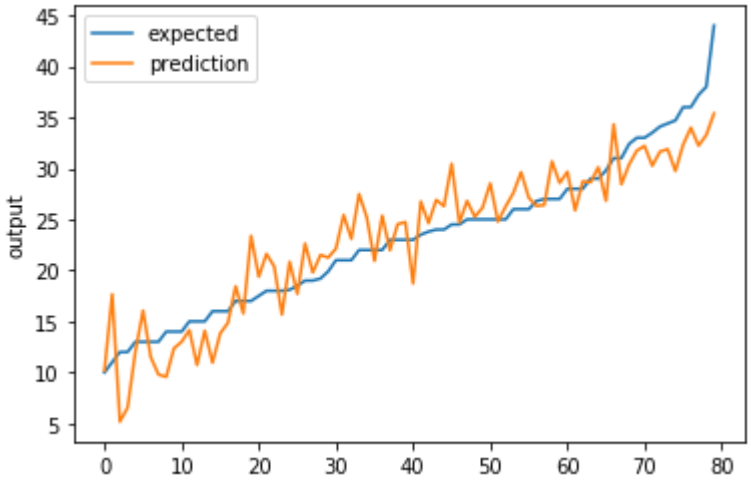
```
if preprocess:
    encode_numeric_zscore(df, 'horsepower')
    encode_numeric_zscore(df, 'weight')
    encode_numeric_zscore(df, 'cylinders')
    encode_numeric_zscore(df, 'displacement')
    encode_numeric_zscore(df, 'acceleration')
    encode_numeric_zscore(df, 'year')

# Encode to a 2D matrix for training
x,y = to_xy(df,'mpg')

# Split into train/test
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.20, random_state=

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')

monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3, patience=5, verbose=1, mode

checkpointer = ModelCheckpoint(filepath="dnn/best_weights.hdf5", verbose=0, save_best_o

model.fit(x_train,y_train,validation_data=(x_test,y_test),callbacks=[monitor,checkpoint

model.load_weights('dnn/best_weights.hdf5') # load weights from best model

# Predict and measure RMSE
pred = model.predict(x_test)

score = np.sqrt(metrics.mean_squared_error(pred,y_test))
print("Score (RMSE): {}".format(score))

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

```
Train on 318 samples, validate on 80 samples
Epoch 1/200
318/318 - 1s - loss: 962.6505 - val_loss: 642.0232
Epoch 2/200
318/318 - 0s - loss: 645.7761 - val_loss: 514.5989
Epoch 3/200
318/318 - 0s - loss: 603.3425 - val_loss: 682.1464
Epoch 4/200
318/318 - 0s - loss: 493.0617 - val_loss: 539.9942
Epoch 5/200
318/318 - 0s - loss: 432.1128 - val_loss: 416.7689
Epoch 6/200
318/318 - 0s - loss: 398.3095 - val_loss: 389.0527
Epoch 7/200
318/318 - 0s - loss: 369.1969 - val_loss: 359.7113
Epoch 8/200
318/318 - 0s - loss: 342.1037 - val_loss: 469.8026
Epoch 9/200
318/318 - 0s - loss: 319.6010 - val_loss: 347.4602
Epoch 10/200
318/318 - 0s - loss: 288.3704 - val_loss: 323.3895
Epoch 11/200
318/318 - 0s - loss: 259.9005 - val_loss: 282.6863
```

```
Epoch 12/200
318/318 - 0s - loss: 237.9831 - val_loss: 273.5553
Epoch 13/200
318/318 - 0s - loss: 220.7566 - val_loss: 275.8234
Epoch 14/200
318/318 - 0s - loss: 193.5468 - val_loss: 194.8694
Epoch 15/200
318/318 - 0s - loss: 202.3762 - val_loss: 176.3358
Epoch 16/200
318/318 - 0s - loss: 200.1138 - val_loss: 150.2110
Epoch 17/200
318/318 - 0s - loss: 171.8594 - val_loss: 139.6666
Epoch 18/200
318/318 - 0s - loss: 141.9661 - val_loss: 139.5058
Epoch 19/200
318/318 - 0s - loss: 130.1029 - val_loss: 147.9603
Epoch 20/200
318/318 - 0s - loss: 116.7589 - val_loss: 131.9817
Epoch 21/200
318/318 - 0s - loss: 112.4108 - val_loss: 145.7085
Epoch 22/200
318/318 - 0s - loss: 109.2915 - val_loss: 136.2665
Epoch 23/200
318/318 - 0s - loss: 104.4306 - val_loss: 105.8651
Epoch 24/200
318/318 - 0s - loss: 87.8622 - val_loss: 100.9069
Epoch 25/200
318/318 - 0s - loss: 81.4030 - val_loss: 69.6995
Epoch 26/200
318/318 - 0s - loss: 82.3979 - val_loss: 79.9601
Epoch 27/200
318/318 - 0s - loss: 74.3895 - val_loss: 66.4708
Epoch 28/200
318/318 - 0s - loss: 71.5869 - val_loss: 57.4273
Epoch 29/200
318/318 - 0s - loss: 69.4866 - val_loss: 63.5906
Epoch 30/200
318/318 - 0s - loss: 65.9844 - val_loss: 60.1135
Epoch 31/200
318/318 - 0s - loss: 62.6553 - val_loss: 65.6620
Epoch 32/200
318/318 - 0s - loss: 65.5772 - val_loss: 70.1613
Epoch 33/200
318/318 - 0s - loss: 66.3648 - val_loss: 47.7548
Epoch 34/200
318/318 - 0s - loss: 58.5009 - val_loss: 45.5102
Epoch 35/200
318/318 - 0s - loss: 61.3097 - val_loss: 40.2617
Epoch 36/200
318/318 - 0s - loss: 53.5297 - val_loss: 49.1750
Epoch 37/200
318/318 - 0s - loss: 47.3526 - val_loss: 46.6208
Epoch 38/200
318/318 - 0s - loss: 45.9297 - val_loss: 46.9137
Epoch 39/200
318/318 - 0s - loss: 45.9319 - val_loss: 34.4890
Epoch 40/200
318/318 - 0s - loss: 41.2115 - val_loss: 34.2023
Epoch 41/200
318/318 - 0s - loss: 39.3765 - val_loss: 36.1540
Epoch 42/200
318/318 - 0s - loss: 37.6734 - val_loss: 35.0276
Epoch 43/200
318/318 - 0s - loss: 37.8926 - val_loss: 32.2073
Epoch 44/200
```

```
318/318 - 0s - loss: 38.2995 - val_loss: 38.6576
Epoch 45/200
318/318 - 0s - loss: 38.6509 - val_loss: 30.5439
Epoch 46/200
318/318 - 0s - loss: 37.7902 - val_loss: 24.7672
Epoch 47/200
318/318 - 0s - loss: 31.4216 - val_loss: 29.5557
Epoch 48/200
318/318 - 0s - loss: 31.7731 - val_loss: 22.9608
Epoch 49/200
318/318 - 0s - loss: 29.6976 - val_loss: 21.9876
Epoch 50/200
318/318 - 0s - loss: 29.6814 - val_loss: 21.3329
Epoch 51/200
318/318 - 0s - loss: 28.9098 - val_loss: 36.5654
Epoch 52/200
318/318 - 0s - loss: 29.8849 - val_loss: 19.7274
Epoch 53/200
318/318 - 0s - loss: 27.5458 - val_loss: 21.0733
Epoch 54/200
318/318 - 0s - loss: 27.6939 - val_loss: 21.0293
Epoch 55/200
318/318 - 0s - loss: 24.4605 - val_loss: 24.1571
Epoch 56/200
318/318 - 0s - loss: 25.1465 - val_loss: 16.9978
Epoch 57/200
318/318 - 0s - loss: 24.3304 - val_loss: 14.8233
Epoch 58/200
318/318 - 0s - loss: 21.8261 - val_loss: 13.5130
Epoch 59/200
318/318 - 0s - loss: 22.4305 - val_loss: 23.3325
Epoch 60/200
318/318 - 0s - loss: 21.0812 - val_loss: 12.2197
Epoch 61/200
318/318 - 0s - loss: 18.8462 - val_loss: 13.0993
Epoch 62/200
318/318 - 0s - loss: 19.6042 - val_loss: 18.9276
Epoch 63/200
318/318 - 0s - loss: 20.3421 - val_loss: 12.6360
Epoch 64/200
318/318 - 0s - loss: 20.7013 - val_loss: 12.1762
Epoch 65/200
318/318 - 0s - loss: 19.0361 - val_loss: 18.4308
Epoch 66/200
318/318 - 0s - loss: 21.1469 - val_loss: 20.4251
Epoch 67/200
318/318 - 0s - loss: 16.3913 - val_loss: 11.1122
Epoch 68/200
318/318 - 0s - loss: 19.0560 - val_loss: 10.4707
Epoch 69/200
318/318 - 0s - loss: 15.5505 - val_loss: 10.2372
Epoch 70/200
318/318 - 0s - loss: 15.5035 - val_loss: 10.2266
Epoch 71/200
318/318 - 0s - loss: 17.4508 - val_loss: 9.9466
Epoch 72/200
318/318 - 0s - loss: 14.4202 - val_loss: 9.8846
Epoch 73/200
318/318 - 0s - loss: 14.6699 - val_loss: 12.3281
Epoch 74/200
318/318 - 0s - loss: 15.7262 - val_loss: 10.0174
Epoch 75/200
318/318 - 0s - loss: 14.4330 - val_loss: 9.8473
Epoch 76/200
318/318 - 0s - loss: 14.7686 - val_loss: 10.2549
```

```
Epoch 77/200
318/318 - 0s - loss: 14.5364 - val_loss: 9.6068
Epoch 78/200
318/318 - 0s - loss: 15.1435 - val_loss: 14.8097
Epoch 79/200
318/318 - 0s - loss: 18.3983 - val_loss: 10.8555
Epoch 80/200
318/318 - 0s - loss: 16.9326 - val_loss: 9.5497
Epoch 81/200
318/318 - 0s - loss: 18.0410 - val_loss: 11.9185
Epoch 82/200
318/318 - 0s - loss: 16.4152 - val_loss: 14.4787
Epoch 83/200
318/318 - 0s - loss: 14.6777 - val_loss: 9.5993
Epoch 84/200
318/318 - 0s - loss: 16.0771 - val_loss: 9.8323
Epoch 85/200
318/318 - 0s - loss: 15.4665 - val_loss: 9.6624
Epoch 00085: early stopping
Score (RMSE): 3.0902559757232666
```



The lift chart as shown above was achieved by performing the following actions:

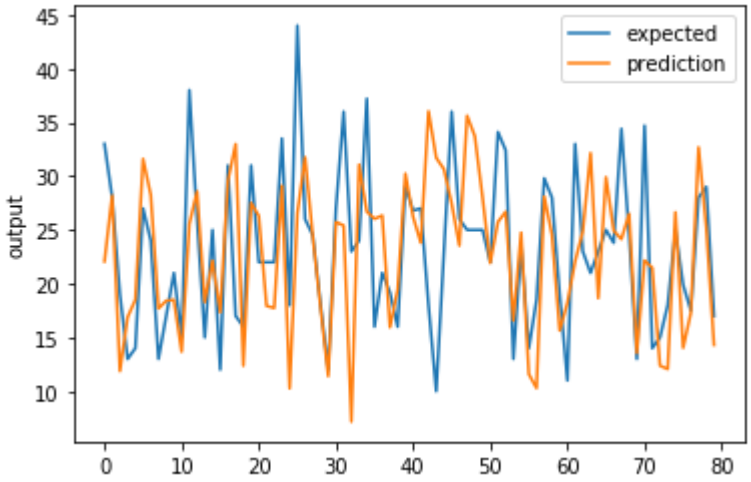
- Sort the data by expected output. Plot the blue line above. Blue line is always increasing.
- For every point on the x-axis plot the predicted value for that same data point. This is the orange line above.
- The x-axis is the index of test data after sorting. The index always starts low and ends high.

Reading a lift chart:

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

Regression lift chart without sorting on expected output (y_test)

In [23]:
```
chart_regression(pred.flatten(),y_test,sort=False)
```



References:

- Google Colab - Free web based platform that includes Python, Juypter Notebooks, and TensorFlow with free GPU support. No setup needed.
- IBM Cognitive Class Labs - Free web based platform that includes Python, Juypter Notebooks, and TensorFlow. No setup needed.
- Python Anaconda - Python distribution that includes many data science packages, such as Numpy, Scipy, Scikit-Learn, Pandas, and much more.
- TensorFlow - Google's mathematics package for deep learning.
- Kaggle - Competitive data science. Good source of sample data.
- T81-558: Applications of Deep Neural Networks. Instructor: Jeff Heaton

CSC 180 Intelligent Systems (Fall 2022)

Dr. Haiquan Chen, Dept of Computer Scicence

California State University, Sacramento

Lab 8: Hyper-parameter Tuning for Backpropagation

Helpful Functions for Tensorflow (Little Gems)

The following functions will be used with TensorFlow to help preprocess the data. They allow you to build the feature vector for a neural network.

- Predictors/Inputs
  - Fill any missing inputs with the median for that column. Use **missing_median**.
  - Encode textual/categorical values with **encode_text_dummy**.
  - Encode numeric values with **encode_numeric_zscore**.
- Output
  - Discard rows with missing outputs.
  - Encode textual/categorical values with **encode_text_index**.
  - Do not encode output numeric values.
- Produce final feature vectors (x) and expected output (y) with **to_xy**.

In [1]:

```
from collections.abc import Sequence
from sklearn import preprocessing
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd


# Encode text values to dummy variables(i.e. [1,0,0],[0,1,0],[0,0,1] for red,green,blue
def encode_text_dummy(df, name):
    dummies = pd.get_dummies(df[name])
    for x in dummies.columns:
        dummy_name = "{}-{}".format(name, x)
        df[dummy_name] = dummies[x]
    df.drop(name, axis=1, inplace=True)


# Encode text values to indexes(i.e. [1],[2],[3] for red,green,blue).
def encode_text_index(df, name):
    le = preprocessing.LabelEncoder()
    df[name] = le.fit_transform(df[name])
    return le.classes_


# Encode a numeric column as zscores
def encode_numeric_zscore(df, name, mean=None, sd=None):
    if mean is None:
        mean = df[name].mean()
```

```
    if sd is None:
        sd = df[name].std()

    df[name] = (df[name] - mean) / sd


# Convert all missing values in the specified column to the median
def missing_median(df, name):
    med = df[name].median()
    df[name] = df[name].fillna(med)


# Convert all missing values in the specified column to the default
def missing_default(df, name, default_value):
    df[name] = df[name].fillna(default_value)


# Convert a Pandas dataframe to the x,y inputs that TensorFlow needs
def to_xy(df, target):
    result = []
    for x in df.columns:
        if x != target:
            result.append(x)
    # find out the type of the target column.
    target_type = df[target].dtypes
    target_type = target_type[0] if isinstance(target_type, Sequence) else target_type
    # Encode to int for classification, float otherwise. TensorFlow likes 32 bits.
    if target_type in (np.int64, np.int32):
        # Classification
        dummies = pd.get_dummies(df[target])
        return df[result].values.astype(np.float32), dummies.values.astype(np.float32)
    else:
        # Regression
        return df[result].values.astype(np.float32), df[target].values.astype(np.float3

# 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)


# 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)
    a = plt.plot(t['y'].tolist(),label='expected')
    b = plt.plot(t['pred'].tolist(),label='prediction')
    plt.ylabel('output')
    plt.legend()
    plt.show()

# 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)
```

```
# Encode a column to a range between normalized_low and normalized_high.
def encode_numeric_range(df, name, normalized_low=-1, normalized_high=1,
                         data_low=None, data_high=None):
    if data_low is None:
        data_low = min(df[name])
        data_high = max(df[name])

    df[name] = ((df[name] - data_low) / (data_high - data_low)) \
               * (normalized_high - normalized_low) + normalized_low
```

Learning rate

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

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

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

- Too low of a learning rate will usually converge to a good solution; however, the process will be very slow.
- 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.

Batch size

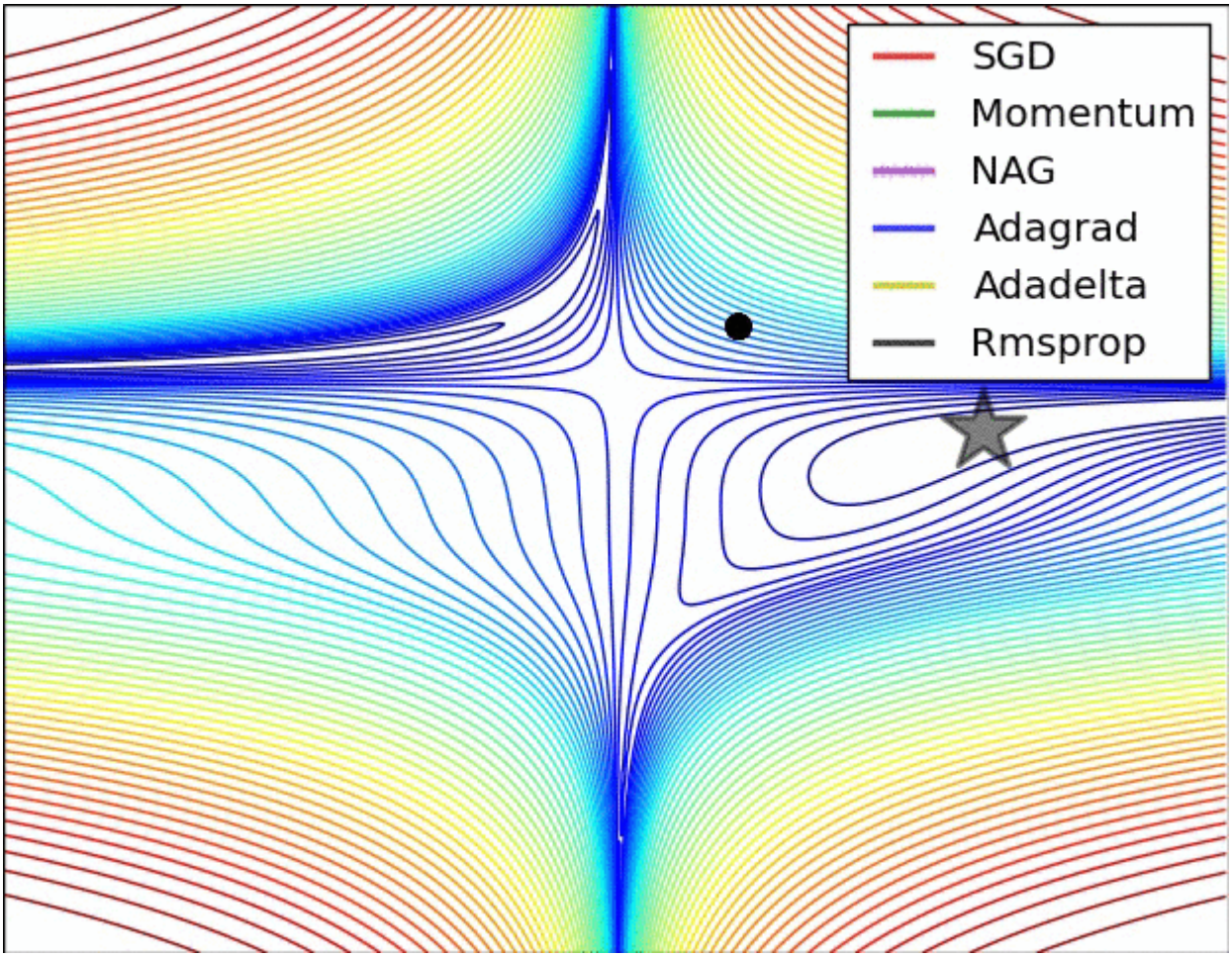
Number of samples per gradient update. In keras, you may set the batch_size parameter in function fit()

https://keras.io/models/model/

batch_size: Integer or None. Number of samples per gradient update. If unspecified, batch_size will default to 32.

Update rules (optimizers)

The following image shows how each of these algorithms train (image credits: author):



An optimizer is one of the two arguments required when you compile a Keras model using compile().

Specifying the Update Rule (Optimizer) in Tensorflow

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

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

https://keras.io/optimizers/

You can either instantiate an optimizer or you can call it by its name. In the latter case, the default parameters for the optimizer will be used.

In [7]:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras import optimizers
from tensorflow.keras.layers import Dense, Activation

model = Sequential()
```

```
model.add(Dense(10, input_dim=10, activation='relu'))
model.add(Dense(1))

sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)

#adam = optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None, decay=0.0, am

model.compile(loss='mean_squared_error', optimizer=sgd)
```

Using all Default Parameters for a Particular Optimizer

In [8]:

```
# pass optimizer by name: default parameters will be used
model.compile(loss='mean_squared_error', optimizer='sgd')
```

A Complete Example:

In [2]:

```
%matplotlib inline
from matplotlib.pyplot import figure, show
from sklearn.model_selection import train_test_split
import pandas as pd
import os
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 tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras import optimizers

path = "./data/"
preprocess = True

filename_read = os.path.join(path,"auto-mpg.csv")
df = pd.read_csv(filename_read,na_values=['NA','?'])

# create feature vector
missing_median(df, 'horsepower')
encode_text_dummy(df, 'origin')
df.drop('name',1,inplace=True)
if preprocess:
    encode_numeric_zscore(df, 'horsepower')
    encode_numeric_zscore(df, 'weight')
    encode_numeric_zscore(df, 'cylinders')
    encode_numeric_zscore(df, 'displacement')
    encode_numeric_zscore(df, 'acceleration')
    encode_numeric_zscore(df, 'year')

# Encode to a 2D matrix for training
x,y = to_xy(df,'mpg')

# Split into train/test
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.20, random_state=

model = Sequential()
model.add(Dense(10, input_dim=x.shape[1], activation='relu'))
```

```
#model.add(Dense(10, activation='tanh'))
model.add(Dense(1, activation='relu'))

adam = optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None, decay=0.0, ams
model.compile(loss='mean_squared_error', optimizer=adam)

monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3, patience=5, verbose=1, mode
checkpointer = ModelCheckpoint(filepath="dnn/best_weights.hdf5", verbose=0, save_best_o

# batch_size: Integer or None. Number of samples per gradient update. If unspecified, b
model.fit(x_train,y_train,validation_data=(x_test,y_test), batch_size= 128, callbacks=[
model.load_weights('dnn/best_weights.hdf5') # load weights from best model

# Predict and measure RMSE
pred = model.predict(x_test)
score = np.sqrt(metrics.mean_squared_error(pred,y_test))
print("Score (RMSE): {}".format(score))

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

```
Train on 318 samples, validate on 80 samples
Epoch 1/1000
318/318 - 1s - loss: 583.3396 - val_loss: 554.9752
Epoch 2/1000
318/318 - 0s - loss: 581.4490 - val_loss: 553.0596
Epoch 3/1000
318/318 - 0s - loss: 579.5466 - val_loss: 551.1432
Epoch 4/1000
318/318 - 0s - loss: 577.6013 - val_loss: 549.2197
Epoch 5/1000
318/318 - 0s - loss: 575.6504 - val_loss: 547.2726
Epoch 6/1000
318/318 - 0s - loss: 573.6275 - val_loss: 545.3247
Epoch 7/1000
318/318 - 0s - loss: 571.6635 - val_loss: 543.3593
Epoch 8/1000
318/318 - 0s - loss: 569.6459 - val_loss: 541.3770
Epoch 9/1000
318/318 - 0s - loss: 567.6753 - val_loss: 539.3716
Epoch 10/1000
318/318 - 0s - loss: 565.6091 - val_loss: 537.3660
Epoch 11/1000
318/318 - 0s - loss: 563.6237 - val_loss: 535.3466
Epoch 12/1000
318/318 - 0s - loss: 561.5554 - val_loss: 533.3208
Epoch 13/1000
318/318 - 0s - loss: 559.5352 - val_loss: 531.2704
Epoch 14/1000
318/318 - 0s - loss: 557.4393 - val_loss: 529.1824
Epoch 15/1000
318/318 - 0s - loss: 555.3522 - val_loss: 527.0941
Epoch 16/1000
318/318 - 0s - loss: 553.2898 - val_loss: 524.9888
Epoch 17/1000
318/318 - 0s - loss: 551.1793 - val_loss: 522.8790
Epoch 18/1000
318/318 - 0s - loss: 549.0388 - val_loss: 520.7588
Epoch 19/1000
318/318 - 0s - loss: 546.8838 - val_loss: 518.6345
Epoch 20/1000
318/318 - 0s - loss: 544.7423 - val_loss: 516.5007
Epoch 21/1000
```

```
318/318 - 0s - loss: 542.5649 - val_loss: 514.3525
Epoch 22/1000
318/318 - 0s - loss: 540.3514 - val_loss: 512.1932
Epoch 23/1000
318/318 - 0s - loss: 538.1843 - val_loss: 510.0158
Epoch 24/1000
318/318 - 0s - loss: 535.9233 - val_loss: 507.8369
Epoch 25/1000
318/318 - 0s - loss: 533.6708 - val_loss: 505.6515
Epoch 26/1000
318/318 - 0s - loss: 531.4252 - val_loss: 503.4474
Epoch 27/1000
318/318 - 0s - loss: 529.1694 - val_loss: 501.2291
Epoch 28/1000
318/318 - 0s - loss: 526.8548 - val_loss: 499.0079
Epoch 29/1000
318/318 - 0s - loss: 524.5812 - val_loss: 496.7724
Epoch 30/1000
318/318 - 0s - loss: 522.2689 - val_loss: 494.5275
Epoch 31/1000
318/318 - 0s - loss: 519.9370 - val_loss: 492.2832
Epoch 32/1000
318/318 - 0s - loss: 517.6089 - val_loss: 490.0208
Epoch 33/1000
318/318 - 0s - loss: 515.2238 - val_loss: 487.7358
Epoch 34/1000
318/318 - 0s - loss: 512.9050 - val_loss: 485.4189
Epoch 35/1000
318/318 - 0s - loss: 510.4694 - val_loss: 483.0887
Epoch 36/1000
318/318 - 0s - loss: 508.0268 - val_loss: 480.7535
Epoch 37/1000
318/318 - 0s - loss: 505.6202 - val_loss: 478.3907
Epoch 38/1000
318/318 - 0s - loss: 503.1594 - val_loss: 476.0142
Epoch 39/1000
318/318 - 0s - loss: 500.7070 - val_loss: 473.6102
Epoch 40/1000
318/318 - 0s - loss: 498.1839 - val_loss: 471.1977
Epoch 41/1000
318/318 - 0s - loss: 495.6168 - val_loss: 468.7710
Epoch 42/1000
318/318 - 0s - loss: 493.0881 - val_loss: 466.3180
Epoch 43/1000
318/318 - 0s - loss: 490.5550 - val_loss: 463.8461
Epoch 44/1000
318/318 - 0s - loss: 487.9712 - val_loss: 461.3619
Epoch 45/1000
318/318 - 0s - loss: 485.3675 - val_loss: 458.8527
Epoch 46/1000
318/318 - 0s - loss: 482.7499 - val_loss: 456.3112
Epoch 47/1000
318/318 - 0s - loss: 480.0763 - val_loss: 453.7496
Epoch 48/1000
318/318 - 0s - loss: 477.4364 - val_loss: 451.1632
Epoch 49/1000
318/318 - 0s - loss: 474.7857 - val_loss: 448.5591
Epoch 50/1000
318/318 - 0s - loss: 472.0544 - val_loss: 445.9526
Epoch 51/1000
318/318 - 0s - loss: 469.3259 - val_loss: 443.3394
Epoch 52/1000
318/318 - 0s - loss: 466.5942 - val_loss: 440.7042
Epoch 53/1000
318/318 - 0s - loss: 463.8126 - val_loss: 438.0496
```

```
Epoch 54/1000
318/318 - 0s - loss: 461.0007 - val_loss: 435.3737
Epoch 55/1000
318/318 - 0s - loss: 458.2342 - val_loss: 432.6544
Epoch 56/1000
318/318 - 0s - loss: 455.4270 - val_loss: 429.9185
Epoch 57/1000
318/318 - 0s - loss: 452.5897 - val_loss: 427.1783
Epoch 58/1000
318/318 - 0s - loss: 449.6842 - val_loss: 424.4415
Epoch 59/1000
318/318 - 0s - loss: 446.8254 - val_loss: 421.6812
Epoch 60/1000
318/318 - 0s - loss: 443.9161 - val_loss: 418.8994
Epoch 61/1000
318/318 - 0s - loss: 440.9996 - val_loss: 416.0827
Epoch 62/1000
318/318 - 0s - loss: 438.0433 - val_loss: 413.2372
Epoch 63/1000
318/318 - 0s - loss: 435.0499 - val_loss: 410.3677
Epoch 64/1000
318/318 - 0s - loss: 431.9728 - val_loss: 407.4745
Epoch 65/1000
318/318 - 0s - loss: 428.9319 - val_loss: 404.5426
Epoch 66/1000
318/318 - 0s - loss: 425.7892 - val_loss: 401.5786
Epoch 67/1000
318/318 - 0s - loss: 422.7177 - val_loss: 398.5778
Epoch 68/1000
318/318 - 0s - loss: 419.5245 - val_loss: 395.5622
Epoch 69/1000
318/318 - 0s - loss: 416.3195 - val_loss: 392.5351
Epoch 70/1000
318/318 - 0s - loss: 413.0015 - val_loss: 389.4974
Epoch 71/1000
318/318 - 0s - loss: 409.8431 - val_loss: 386.4048
Epoch 72/1000
318/318 - 0s - loss: 406.5742 - val_loss: 383.2970
Epoch 73/1000
318/318 - 0s - loss: 403.2780 - val_loss: 380.1741
Epoch 74/1000
318/318 - 0s - loss: 399.9279 - val_loss: 377.0459
Epoch 75/1000
318/318 - 0s - loss: 396.6577 - val_loss: 373.8931
Epoch 76/1000
318/318 - 0s - loss: 393.1923 - val_loss: 370.7549
Epoch 77/1000
318/318 - 0s - loss: 389.8535 - val_loss: 367.5815
Epoch 78/1000
318/318 - 0s - loss: 386.5445 - val_loss: 364.3679
Epoch 79/1000
318/318 - 0s - loss: 383.0581 - val_loss: 361.1656
Epoch 80/1000
318/318 - 0s - loss: 379.6950 - val_loss: 357.9610
Epoch 81/1000
318/318 - 0s - loss: 376.2602 - val_loss: 354.7613
Epoch 82/1000
318/318 - 0s - loss: 372.8501 - val_loss: 351.5692
Epoch 83/1000
318/318 - 0s - loss: 369.4778 - val_loss: 348.3654
Epoch 84/1000
318/318 - 0s - loss: 366.0749 - val_loss: 345.1559
Epoch 85/1000
318/318 - 0s - loss: 362.5831 - val_loss: 341.9652
Epoch 86/1000
```

```
318/318 - 0s - loss: 359.2409 - val_loss: 338.7522
Epoch 87/1000
318/318 - 0s - loss: 355.7882 - val_loss: 335.5191
Epoch 88/1000
318/318 - 0s - loss: 352.3505 - val_loss: 332.2711
Epoch 89/1000
318/318 - 0s - loss: 348.8457 - val_loss: 329.0228
Epoch 90/1000
318/318 - 0s - loss: 345.3891 - val_loss: 325.7637
Epoch 91/1000
318/318 - 0s - loss: 341.9474 - val_loss: 322.4886
Epoch 92/1000
318/318 - 0s - loss: 338.3673 - val_loss: 319.2385
Epoch 93/1000
318/318 - 0s - loss: 334.9200 - val_loss: 315.9599
Epoch 94/1000
318/318 - 0s - loss: 331.4216 - val_loss: 312.6861
Epoch 95/1000
318/318 - 0s - loss: 327.8681 - val_loss: 309.4349
Epoch 96/1000
318/318 - 0s - loss: 324.4956 - val_loss: 306.1825
Epoch 97/1000
318/318 - 0s - loss: 320.9243 - val_loss: 302.9519
Epoch 98/1000
318/318 - 0s - loss: 317.4768 - val_loss: 299.7247
Epoch 99/1000
318/318 - 0s - loss: 314.0223 - val_loss: 296.4930
Epoch 100/1000
318/318 - 0s - loss: 310.5150 - val_loss: 293.2668
Epoch 101/1000
318/318 - 0s - loss: 307.0870 - val_loss: 290.0377
Epoch 102/1000
318/318 - 0s - loss: 303.6841 - val_loss: 286.8167
Epoch 103/1000
318/318 - 0s - loss: 300.2434 - val_loss: 283.6325
Epoch 104/1000
318/318 - 0s - loss: 296.7830 - val_loss: 280.4800
Epoch 105/1000
318/318 - 0s - loss: 293.3725 - val_loss: 277.3343
Epoch 106/1000
318/318 - 0s - loss: 290.0112 - val_loss: 274.1909
Epoch 107/1000
318/318 - 0s - loss: 286.6444 - val_loss: 271.0579
Epoch 108/1000
318/318 - 0s - loss: 283.2574 - val_loss: 267.9325
Epoch 109/1000
318/318 - 0s - loss: 279.8754 - val_loss: 264.7981
Epoch 110/1000
318/318 - 0s - loss: 276.5421 - val_loss: 261.6469
Epoch 111/1000
318/318 - 0s - loss: 273.1423 - val_loss: 258.5191
Epoch 112/1000
318/318 - 0s - loss: 269.7744 - val_loss: 255.4024
Epoch 113/1000
318/318 - 0s - loss: 266.4590 - val_loss: 252.2951
Epoch 114/1000
318/318 - 0s - loss: 263.2026 - val_loss: 249.1993
Epoch 115/1000
318/318 - 0s - loss: 259.8743 - val_loss: 246.1344
Epoch 116/1000
318/318 - 0s - loss: 256.5655 - val_loss: 243.0927
Epoch 117/1000
318/318 - 0s - loss: 253.3164 - val_loss: 240.0736
Epoch 118/1000
318/318 - 0s - loss: 250.0206 - val_loss: 237.0840
```

```
Epoch 119/1000
318/318 - 0s - loss: 246.8333 - val_loss: 234.0988
Epoch 120/1000
318/318 - 0s - loss: 243.6765 - val_loss: 231.1272
Epoch 121/1000
318/318 - 0s - loss: 240.4614 - val_loss: 228.1880
Epoch 122/1000
318/318 - 0s - loss: 237.3047 - val_loss: 225.2592
Epoch 123/1000
318/318 - 0s - loss: 234.1693 - val_loss: 222.3397
Epoch 124/1000
318/318 - 0s - loss: 231.0326 - val_loss: 219.4513
Epoch 125/1000
318/318 - 0s - loss: 227.9162 - val_loss: 216.6000
Epoch 126/1000
318/318 - 0s - loss: 224.9294 - val_loss: 213.7709
Epoch 127/1000
318/318 - 0s - loss: 221.8394 - val_loss: 211.0020
Epoch 128/1000
318/318 - 0s - loss: 218.8784 - val_loss: 208.2583
Epoch 129/1000
318/318 - 0s - loss: 215.9449 - val_loss: 205.5476
Epoch 130/1000
318/318 - 0s - loss: 212.9632 - val_loss: 202.8846
Epoch 131/1000
318/318 - 0s - loss: 210.1351 - val_loss: 200.2228
Epoch 132/1000
318/318 - 0s - loss: 207.2387 - val_loss: 197.5829
Epoch 133/1000
318/318 - 0s - loss: 204.4039 - val_loss: 194.9537
Epoch 134/1000
318/318 - 0s - loss: 201.5977 - val_loss: 192.3352
Epoch 135/1000
318/318 - 0s - loss: 198.7337 - val_loss: 189.7414
Epoch 136/1000
318/318 - 0s - loss: 195.9719 - val_loss: 187.1578
Epoch 137/1000
318/318 - 0s - loss: 193.1836 - val_loss: 184.5990
Epoch 138/1000
318/318 - 0s - loss: 190.3872 - val_loss: 182.0724
Epoch 139/1000
318/318 - 0s - loss: 187.6884 - val_loss: 179.5652
Epoch 140/1000
318/318 - 0s - loss: 184.9422 - val_loss: 177.0889
Epoch 141/1000
318/318 - 0s - loss: 182.3097 - val_loss: 174.6212
Epoch 142/1000
318/318 - 0s - loss: 179.6404 - val_loss: 172.1819
Epoch 143/1000
318/318 - 0s - loss: 177.0570 - val_loss: 169.7650
Epoch 144/1000
318/318 - 0s - loss: 174.3920 - val_loss: 167.3963
Epoch 145/1000
318/318 - 0s - loss: 171.8865 - val_loss: 165.0401
Epoch 146/1000
318/318 - 0s - loss: 169.2902 - val_loss: 162.7265
Epoch 147/1000
318/318 - 0s - loss: 166.8195 - val_loss: 160.4276
Epoch 148/1000
318/318 - 0s - loss: 164.3407 - val_loss: 158.1475
Epoch 149/1000
318/318 - 0s - loss: 161.8757 - val_loss: 155.8944
Epoch 150/1000
318/318 - 0s - loss: 159.4534 - val_loss: 153.6720
Epoch 151/1000
```

```
318/318 - 0s - loss: 157.0682 - val_loss: 151.4751
Epoch 152/1000
318/318 - 0s - loss: 154.6696 - val_loss: 149.3142
Epoch 153/1000
318/318 - 0s - loss: 152.3533 - val_loss: 147.1818
Epoch 154/1000
318/318 - 0s - loss: 150.0763 - val_loss: 145.0799
Epoch 155/1000
318/318 - 0s - loss: 147.7909 - val_loss: 143.0179
Epoch 156/1000
318/318 - 0s - loss: 145.5866 - val_loss: 140.9695
Epoch 157/1000
318/318 - 0s - loss: 143.3644 - val_loss: 138.9524
Epoch 158/1000
318/318 - 0s - loss: 141.2176 - val_loss: 136.9695
Epoch 159/1000
318/318 - 0s - loss: 139.0844 - val_loss: 135.0211
Epoch 160/1000
318/318 - 0s - loss: 136.9344 - val_loss: 133.1113
Epoch 161/1000
318/318 - 0s - loss: 134.8651 - val_loss: 131.2181
Epoch 162/1000
318/318 - 0s - loss: 132.8381 - val_loss: 129.3404
Epoch 163/1000
318/318 - 0s - loss: 130.8106 - val_loss: 127.4903
Epoch 164/1000
318/318 - 0s - loss: 128.8378 - val_loss: 125.6613
Epoch 165/1000
318/318 - 0s - loss: 126.8453 - val_loss: 123.8653
Epoch 166/1000
318/318 - 0s - loss: 124.8965 - val_loss: 122.0964
Epoch 167/1000
318/318 - 0s - loss: 122.9986 - val_loss: 120.3454
Epoch 168/1000
318/318 - 0s - loss: 121.0636 - val_loss: 118.6292
Epoch 169/1000
318/318 - 0s - loss: 119.1962 - val_loss: 116.9346
Epoch 170/1000
318/318 - 0s - loss: 117.3713 - val_loss: 115.2594
Epoch 171/1000
318/318 - 0s - loss: 115.5546 - val_loss: 113.6047
Epoch 172/1000
318/318 - 0s - loss: 113.7971 - val_loss: 111.9714
Epoch 173/1000
318/318 - 0s - loss: 112.0345 - val_loss: 110.3685
Epoch 174/1000
318/318 - 0s - loss: 110.3075 - val_loss: 108.8019
Epoch 175/1000
318/318 - 0s - loss: 108.6055 - val_loss: 107.2665
Epoch 176/1000
318/318 - 0s - loss: 106.9266 - val_loss: 105.7636
Epoch 177/1000
318/318 - 0s - loss: 105.3246 - val_loss: 104.2772
Epoch 178/1000
318/318 - 0s - loss: 103.6966 - val_loss: 102.8152
Epoch 179/1000
318/318 - 0s - loss: 102.0915 - val_loss: 101.3756
Epoch 180/1000
318/318 - 0s - loss: 100.5660 - val_loss: 99.9419
Epoch 181/1000
318/318 - 0s - loss: 99.0203 - val_loss: 98.5306
Epoch 182/1000
318/318 - 0s - loss: 97.4890 - val_loss: 97.1436
Epoch 183/1000
318/318 - 0s - loss: 96.0098 - val_loss: 95.7766
```

```
Epoch 184/1000
318/318 - 0s - loss: 94.5644 - val_loss: 94.4349
Epoch 185/1000
318/318 - 0s - loss: 93.1156 - val_loss: 93.1280
Epoch 186/1000
318/318 - 0s - loss: 91.7092 - val_loss: 91.8474
Epoch 187/1000
318/318 - 0s - loss: 90.3196 - val_loss: 90.5896
Epoch 188/1000
318/318 - 0s - loss: 88.9836 - val_loss: 89.3508
Epoch 189/1000
318/318 - 0s - loss: 87.6579 - val_loss: 88.1351
Epoch 190/1000
318/318 - 0s - loss: 86.3518 - val_loss: 86.9454
Epoch 191/1000
318/318 - 0s - loss: 85.0350 - val_loss: 85.7871
Epoch 192/1000
318/318 - 0s - loss: 83.8583 - val_loss: 84.6306
Epoch 193/1000
318/318 - 0s - loss: 82.6288 - val_loss: 83.5009
Epoch 194/1000
318/318 - 0s - loss: 81.4256 - val_loss: 82.4009
Epoch 195/1000
318/318 - 0s - loss: 80.2236 - val_loss: 81.3284
Epoch 196/1000
318/318 - 0s - loss: 79.0659 - val_loss: 80.2723
Epoch 197/1000
318/318 - 0s - loss: 77.9480 - val_loss: 79.2254
Epoch 198/1000
318/318 - 0s - loss: 76.8280 - val_loss: 78.1904
Epoch 199/1000
318/318 - 0s - loss: 75.7676 - val_loss: 77.1719
Epoch 200/1000
318/318 - 0s - loss: 74.6523 - val_loss: 76.1893
Epoch 201/1000
318/318 - 0s - loss: 73.6241 - val_loss: 75.2215
Epoch 202/1000
318/318 - 0s - loss: 72.5831 - val_loss: 74.2767
Epoch 203/1000
318/318 - 0s - loss: 71.5761 - val_loss: 73.3436
Epoch 204/1000
318/318 - 0s - loss: 70.6089 - val_loss: 72.4183
Epoch 205/1000
318/318 - 0s - loss: 69.6258 - val_loss: 71.5188
Epoch 206/1000
318/318 - 0s - loss: 68.6793 - val_loss: 70.6389
Epoch 207/1000
318/318 - 0s - loss: 67.7610 - val_loss: 69.7771
Epoch 208/1000
318/318 - 0s - loss: 66.8671 - val_loss: 68.9356
Epoch 209/1000
318/318 - 0s - loss: 65.9628 - val_loss: 68.1171
Epoch 210/1000
318/318 - 0s - loss: 65.1203 - val_loss: 67.3099
Epoch 211/1000
318/318 - 0s - loss: 64.2674 - val_loss: 66.5215
Epoch 212/1000
318/318 - 0s - loss: 63.4438 - val_loss: 65.7420
Epoch 213/1000
318/318 - 0s - loss: 62.6098 - val_loss: 64.9773
Epoch 214/1000
318/318 - 0s - loss: 61.8298 - val_loss: 64.2217
Epoch 215/1000
318/318 - 0s - loss: 61.0305 - val_loss: 63.4852
Epoch 216/1000
```

```
318/318 - 0s - loss: 60.2711 - val_loss: 62.7575
Epoch 217/1000
318/318 - 0s - loss: 59.5085 - val_loss: 62.0480
Epoch 218/1000
318/318 - 0s - loss: 58.8041 - val_loss: 61.3498
Epoch 219/1000
318/318 - 0s - loss: 58.0402 - val_loss: 60.6790
Epoch 220/1000
318/318 - 0s - loss: 57.3567 - val_loss: 60.0185
Epoch 221/1000
318/318 - 0s - loss: 56.6677 - val_loss: 59.3677
Epoch 222/1000
318/318 - 0s - loss: 55.9847 - val_loss: 58.7282
Epoch 223/1000
318/318 - 0s - loss: 55.3318 - val_loss: 58.0984
Epoch 224/1000
318/318 - 0s - loss: 54.6838 - val_loss: 57.4841
Epoch 225/1000
318/318 - 0s - loss: 54.0328 - val_loss: 56.8881
Epoch 226/1000
318/318 - 0s - loss: 53.4169 - val_loss: 56.3021
Epoch 227/1000
318/318 - 0s - loss: 52.8098 - val_loss: 55.7287
Epoch 228/1000
318/318 - 0s - loss: 52.2265 - val_loss: 55.1674
Epoch 229/1000
318/318 - 0s - loss: 51.6424 - val_loss: 54.6182
Epoch 230/1000
318/318 - 0s - loss: 51.0649 - val_loss: 54.0830
Epoch 231/1000
318/318 - 0s - loss: 50.5225 - val_loss: 53.5569
Epoch 232/1000
318/318 - 0s - loss: 49.9616 - val_loss: 53.0447
Epoch 233/1000
318/318 - 0s - loss: 49.4544 - val_loss: 52.5397
Epoch 234/1000
318/318 - 0s - loss: 48.9270 - val_loss: 52.0464
Epoch 235/1000
318/318 - 0s - loss: 48.4364 - val_loss: 51.5661
Epoch 236/1000
318/318 - 0s - loss: 47.9228 - val_loss: 51.0984
Epoch 237/1000
318/318 - 0s - loss: 47.4535 - val_loss: 50.6360
Epoch 238/1000
318/318 - 0s - loss: 46.9769 - val_loss: 50.1859
Epoch 239/1000
318/318 - 0s - loss: 46.5342 - val_loss: 49.7429
Epoch 240/1000
318/318 - 0s - loss: 46.0501 - val_loss: 49.3126
Epoch 241/1000
318/318 - 0s - loss: 45.6267 - val_loss: 48.8844
Epoch 242/1000
318/318 - 0s - loss: 45.1838 - val_loss: 48.4654
Epoch 243/1000
318/318 - 0s - loss: 44.7768 - val_loss: 48.0506
Epoch 244/1000
318/318 - 0s - loss: 44.3346 - val_loss: 47.6511
Epoch 245/1000
318/318 - 0s - loss: 43.9320 - val_loss: 47.2547
Epoch 246/1000
318/318 - 0s - loss: 43.5354 - val_loss: 46.8638
Epoch 247/1000
318/318 - 0s - loss: 43.1384 - val_loss: 46.4800
Epoch 248/1000
318/318 - 0s - loss: 42.7603 - val_loss: 46.1007
```

```
Epoch 249/1000
318/318 - 0s - loss: 42.3731 - val_loss: 45.7297
Epoch 250/1000
318/318 - 0s - loss: 42.0045 - val_loss: 45.3625
Epoch 251/1000
318/318 - 0s - loss: 41.6415 - val_loss: 45.0014
Epoch 252/1000
318/318 - 0s - loss: 41.2909 - val_loss: 44.6477
Epoch 253/1000
318/318 - 0s - loss: 40.9320 - val_loss: 44.3055
Epoch 254/1000
318/318 - 0s - loss: 40.5966 - val_loss: 43.9688
Epoch 255/1000
318/318 - 0s - loss: 40.2660 - val_loss: 43.6396
Epoch 256/1000
318/318 - 0s - loss: 39.9283 - val_loss: 43.3193
Epoch 257/1000
318/318 - 0s - loss: 39.6181 - val_loss: 43.0027
Epoch 258/1000
318/318 - 0s - loss: 39.3068 - val_loss: 42.6948
Epoch 259/1000
318/318 - 0s - loss: 39.0079 - val_loss: 42.3932
Epoch 260/1000
318/318 - 0s - loss: 38.7091 - val_loss: 42.0955
Epoch 261/1000
318/318 - 0s - loss: 38.4137 - val_loss: 41.8035
Epoch 262/1000
318/318 - 0s - loss: 38.1350 - val_loss: 41.5163
Epoch 263/1000
318/318 - 0s - loss: 37.8530 - val_loss: 41.2362
Epoch 264/1000
318/318 - 0s - loss: 37.5645 - val_loss: 40.9625
Epoch 265/1000
318/318 - 0s - loss: 37.2919 - val_loss: 40.6914
Epoch 266/1000
318/318 - 0s - loss: 37.0302 - val_loss: 40.4198
Epoch 267/1000
318/318 - 0s - loss: 36.7715 - val_loss: 40.1497
Epoch 268/1000
318/318 - 0s - loss: 36.4961 - val_loss: 39.8843
Epoch 269/1000
318/318 - 0s - loss: 36.2442 - val_loss: 39.6218
Epoch 270/1000
318/318 - 0s - loss: 35.9863 - val_loss: 39.3647
Epoch 271/1000
318/318 - 0s - loss: 35.7382 - val_loss: 39.1140
Epoch 272/1000
318/318 - 0s - loss: 35.4889 - val_loss: 38.8679
Epoch 273/1000
318/318 - 0s - loss: 35.2512 - val_loss: 38.6244
Epoch 274/1000
318/318 - 0s - loss: 35.0252 - val_loss: 38.3823
Epoch 275/1000
318/318 - 0s - loss: 34.7952 - val_loss: 38.1457
Epoch 276/1000
318/318 - 0s - loss: 34.5699 - val_loss: 37.9168
Epoch 277/1000
318/318 - 0s - loss: 34.3519 - val_loss: 37.6913
Epoch 278/1000
318/318 - 0s - loss: 34.1331 - val_loss: 37.4663
Epoch 279/1000
318/318 - 0s - loss: 33.9260 - val_loss: 37.2443
Epoch 280/1000
318/318 - 0s - loss: 33.7035 - val_loss: 37.0284
Epoch 281/1000
```

```
318/318 - 0s - loss: 33.4929 - val_loss: 36.8121
Epoch 282/1000
318/318 - 0s - loss: 33.2883 - val_loss: 36.5953
Epoch 283/1000
318/318 - 0s - loss: 33.0858 - val_loss: 36.3798
Epoch 284/1000
318/318 - 0s - loss: 32.8854 - val_loss: 36.1662
Epoch 285/1000
318/318 - 0s - loss: 32.6857 - val_loss: 35.9557
Epoch 286/1000
318/318 - 0s - loss: 32.4808 - val_loss: 35.7496
Epoch 287/1000
318/318 - 0s - loss: 32.2963 - val_loss: 35.5425
Epoch 288/1000
318/318 - 0s - loss: 32.1063 - val_loss: 35.3394
Epoch 289/1000
318/318 - 0s - loss: 31.9098 - val_loss: 35.1371
Epoch 290/1000
318/318 - 0s - loss: 31.7257 - val_loss: 34.9346
Epoch 291/1000
318/318 - 0s - loss: 31.5363 - val_loss: 34.7353
Epoch 292/1000
318/318 - 0s - loss: 31.3602 - val_loss: 34.5362
Epoch 293/1000
318/318 - 0s - loss: 31.1849 - val_loss: 34.3392
Epoch 294/1000
318/318 - 0s - loss: 31.0042 - val_loss: 34.1497
Epoch 295/1000
318/318 - 0s - loss: 30.8281 - val_loss: 33.9597
Epoch 296/1000
318/318 - 0s - loss: 30.6522 - val_loss: 33.7714
Epoch 297/1000
318/318 - 0s - loss: 30.4787 - val_loss: 33.5825
Epoch 298/1000
318/318 - 0s - loss: 30.3159 - val_loss: 33.3914
Epoch 299/1000
318/318 - 0s - loss: 30.1399 - val_loss: 33.2015
Epoch 300/1000
318/318 - 0s - loss: 29.9727 - val_loss: 33.0120
Epoch 301/1000
318/318 - 0s - loss: 29.8059 - val_loss: 32.8258
Epoch 302/1000
318/318 - 0s - loss: 29.6375 - val_loss: 32.6439
Epoch 303/1000
318/318 - 0s - loss: 29.4773 - val_loss: 32.4600
Epoch 304/1000
318/318 - 0s - loss: 29.3151 - val_loss: 32.2787
Epoch 305/1000
318/318 - 0s - loss: 29.1549 - val_loss: 32.1000
Epoch 306/1000
318/318 - 0s - loss: 28.9986 - val_loss: 31.9261
Epoch 307/1000
318/318 - 0s - loss: 28.8397 - val_loss: 31.7545
Epoch 308/1000
318/318 - 0s - loss: 28.6879 - val_loss: 31.5820
Epoch 309/1000
318/318 - 0s - loss: 28.5372 - val_loss: 31.4120
Epoch 310/1000
318/318 - 0s - loss: 28.3860 - val_loss: 31.2449
Epoch 311/1000
318/318 - 0s - loss: 28.2402 - val_loss: 31.0769
Epoch 312/1000
318/318 - 0s - loss: 28.0881 - val_loss: 30.9082
Epoch 313/1000
318/318 - 0s - loss: 27.9429 - val_loss: 30.7407
```

```
Epoch 314/1000
318/318 - 0s - loss: 27.8001 - val_loss: 30.5744
Epoch 315/1000
318/318 - 0s - loss: 27.6502 - val_loss: 30.4084
Epoch 316/1000
318/318 - 0s - loss: 27.5025 - val_loss: 30.2449
Epoch 317/1000
318/318 - 0s - loss: 27.3589 - val_loss: 30.0809
Epoch 318/1000
318/318 - 0s - loss: 27.2226 - val_loss: 29.9185
Epoch 319/1000
318/318 - 0s - loss: 27.0784 - val_loss: 29.7562
Epoch 320/1000
318/318 - 0s - loss: 26.9451 - val_loss: 29.5955
Epoch 321/1000
318/318 - 0s - loss: 26.8015 - val_loss: 29.4393
Epoch 322/1000
318/318 - 0s - loss: 26.6679 - val_loss: 29.2823
Epoch 323/1000
318/318 - 0s - loss: 26.5255 - val_loss: 29.1262
Epoch 324/1000
318/318 - 0s - loss: 26.3884 - val_loss: 28.9695
Epoch 325/1000
318/318 - 0s - loss: 26.2618 - val_loss: 28.8124
Epoch 326/1000
318/318 - 0s - loss: 26.1225 - val_loss: 28.6550
Epoch 327/1000
318/318 - 0s - loss: 25.9905 - val_loss: 28.4975
Epoch 328/1000
318/318 - 0s - loss: 25.8585 - val_loss: 28.3432
Epoch 329/1000
318/318 - 0s - loss: 25.7246 - val_loss: 28.1896
Epoch 330/1000
318/318 - 0s - loss: 25.5946 - val_loss: 28.0370
Epoch 331/1000
318/318 - 0s - loss: 25.4619 - val_loss: 27.8841
Epoch 332/1000
318/318 - 0s - loss: 25.3306 - val_loss: 27.7302
Epoch 333/1000
318/318 - 0s - loss: 25.2019 - val_loss: 27.5759
Epoch 334/1000
318/318 - 0s - loss: 25.0681 - val_loss: 27.4250
Epoch 335/1000
318/318 - 0s - loss: 24.9393 - val_loss: 27.2735
Epoch 336/1000
318/318 - 0s - loss: 24.8154 - val_loss: 27.1235
Epoch 337/1000
318/318 - 0s - loss: 24.6874 - val_loss: 26.9739
Epoch 338/1000
318/318 - 0s - loss: 24.5546 - val_loss: 26.8267
Epoch 339/1000
318/318 - 0s - loss: 24.4376 - val_loss: 26.6776
Epoch 340/1000
318/318 - 0s - loss: 24.3060 - val_loss: 26.5336
Epoch 341/1000
318/318 - 0s - loss: 24.1888 - val_loss: 26.3870
Epoch 342/1000
318/318 - 0s - loss: 24.0641 - val_loss: 26.2416
Epoch 343/1000
318/318 - 0s - loss: 23.9418 - val_loss: 26.0991
Epoch 344/1000
318/318 - 0s - loss: 23.8256 - val_loss: 25.9567
Epoch 345/1000
318/318 - 0s - loss: 23.7078 - val_loss: 25.8155
Epoch 346/1000
```

```
318/318 - 0s - loss: 23.5909 - val_loss: 25.6770
Epoch 347/1000
318/318 - 0s - loss: 23.4693 - val_loss: 25.5384
Epoch 348/1000
318/318 - 0s - loss: 23.3591 - val_loss: 25.3990
Epoch 349/1000
318/318 - 0s - loss: 23.2381 - val_loss: 25.2608
Epoch 350/1000
318/318 - 0s - loss: 23.1243 - val_loss: 25.1196
Epoch 351/1000
318/318 - 0s - loss: 23.0107 - val_loss: 24.9822
Epoch 352/1000
318/318 - 0s - loss: 22.8991 - val_loss: 24.8433
Epoch 353/1000
318/318 - 0s - loss: 22.7856 - val_loss: 24.7057
Epoch 354/1000
318/318 - 0s - loss: 22.6726 - val_loss: 24.5666
Epoch 355/1000
318/318 - 0s - loss: 22.5618 - val_loss: 24.4261
Epoch 356/1000
318/318 - 0s - loss: 22.4477 - val_loss: 24.2885
Epoch 357/1000
318/318 - 0s - loss: 22.3358 - val_loss: 24.1502
Epoch 358/1000
318/318 - 0s - loss: 22.2219 - val_loss: 24.0130
Epoch 359/1000
318/318 - 0s - loss: 22.1151 - val_loss: 23.8763
Epoch 360/1000
318/318 - 0s - loss: 21.9990 - val_loss: 23.7419
Epoch 361/1000
318/318 - 0s - loss: 21.8934 - val_loss: 23.6050
Epoch 362/1000
318/318 - 0s - loss: 21.7861 - val_loss: 23.4674
Epoch 363/1000
318/318 - 0s - loss: 21.6746 - val_loss: 23.3317
Epoch 364/1000
318/318 - 0s - loss: 21.5713 - val_loss: 23.1940
Epoch 365/1000
318/318 - 0s - loss: 21.4606 - val_loss: 23.0593
Epoch 366/1000
318/318 - 0s - loss: 21.3560 - val_loss: 22.9249
Epoch 367/1000
318/318 - 0s - loss: 21.2515 - val_loss: 22.7907
Epoch 368/1000
318/318 - 0s - loss: 21.1523 - val_loss: 22.6576
Epoch 369/1000
318/318 - 0s - loss: 21.0417 - val_loss: 22.5262
Epoch 370/1000
318/318 - 0s - loss: 20.9387 - val_loss: 22.3964
Epoch 371/1000
318/318 - 0s - loss: 20.8343 - val_loss: 22.2655
Epoch 372/1000
318/318 - 0s - loss: 20.7317 - val_loss: 22.1314
Epoch 373/1000
318/318 - 0s - loss: 20.6274 - val_loss: 21.9998
Epoch 374/1000
318/318 - 0s - loss: 20.5276 - val_loss: 21.8692
Epoch 375/1000
318/318 - 0s - loss: 20.4215 - val_loss: 21.7389
Epoch 376/1000
318/318 - 0s - loss: 20.3199 - val_loss: 21.6079
Epoch 377/1000
318/318 - 0s - loss: 20.2221 - val_loss: 21.4771
Epoch 378/1000
318/318 - 0s - loss: 20.1173 - val_loss: 21.3496
```

```
Epoch 379/1000
318/318 - 0s - loss: 20.0175 - val_loss: 21.2218
Epoch 380/1000
318/318 - 0s - loss: 19.9149 - val_loss: 21.0937
Epoch 381/1000
318/318 - 0s - loss: 19.8195 - val_loss: 20.9645
Epoch 382/1000
318/318 - 0s - loss: 19.7179 - val_loss: 20.8392
Epoch 383/1000
318/318 - 0s - loss: 19.6176 - val_loss: 20.7146
Epoch 384/1000
318/318 - 0s - loss: 19.5188 - val_loss: 20.5916
Epoch 385/1000
318/318 - 0s - loss: 19.4249 - val_loss: 20.4682
Epoch 386/1000
318/318 - 0s - loss: 19.3215 - val_loss: 20.3475
Epoch 387/1000
318/318 - 0s - loss: 19.2297 - val_loss: 20.2247
Epoch 388/1000
318/318 - 0s - loss: 19.1313 - val_loss: 20.1032
Epoch 389/1000
318/318 - 0s - loss: 19.0348 - val_loss: 19.9824
Epoch 390/1000
318/318 - 0s - loss: 18.9406 - val_loss: 19.8624
Epoch 391/1000
318/318 - 0s - loss: 18.8441 - val_loss: 19.7438
Epoch 392/1000
318/318 - 0s - loss: 18.7498 - val_loss: 19.6223
Epoch 393/1000
318/318 - 0s - loss: 18.6545 - val_loss: 19.5031
Epoch 394/1000
318/318 - 0s - loss: 18.5634 - val_loss: 19.3841
Epoch 395/1000
318/318 - 0s - loss: 18.4662 - val_loss: 19.2671
Epoch 396/1000
318/318 - 0s - loss: 18.3760 - val_loss: 19.1517
Epoch 397/1000
318/318 - 0s - loss: 18.2823 - val_loss: 19.0372
Epoch 398/1000
318/318 - 0s - loss: 18.1921 - val_loss: 18.9219
Epoch 399/1000
318/318 - 0s - loss: 18.1080 - val_loss: 18.8074
Epoch 400/1000
318/318 - 0s - loss: 18.0155 - val_loss: 18.6963
Epoch 401/1000
318/318 - 0s - loss: 17.9217 - val_loss: 18.5891
Epoch 402/1000
318/318 - 0s - loss: 17.8417 - val_loss: 18.4795
Epoch 403/1000
318/318 - 0s - loss: 17.7509 - val_loss: 18.3715
Epoch 404/1000
318/318 - 0s - loss: 17.6676 - val_loss: 18.2624
Epoch 405/1000
318/318 - 0s - loss: 17.5842 - val_loss: 18.1532
Epoch 406/1000
318/318 - 0s - loss: 17.4983 - val_loss: 18.0469
Epoch 407/1000
318/318 - 0s - loss: 17.4130 - val_loss: 17.9438
Epoch 408/1000
318/318 - 0s - loss: 17.3365 - val_loss: 17.8425
Epoch 409/1000
318/318 - 0s - loss: 17.2514 - val_loss: 17.7411
Epoch 410/1000
318/318 - 0s - loss: 17.1755 - val_loss: 17.6387
Epoch 411/1000
```

```
318/318 - 0s - loss: 17.0925 - val_loss: 17.5363
Epoch 412/1000
318/318 - 0s - loss: 17.0149 - val_loss: 17.4321
Epoch 413/1000
318/318 - 0s - loss: 16.9307 - val_loss: 17.3297
Epoch 414/1000
318/318 - 0s - loss: 16.8524 - val_loss: 17.2268
Epoch 415/1000
318/318 - 0s - loss: 16.7783 - val_loss: 17.1243
Epoch 416/1000
318/318 - 0s - loss: 16.6989 - val_loss: 17.0230
Epoch 417/1000
318/318 - 0s - loss: 16.6188 - val_loss: 16.9211
Epoch 418/1000
318/318 - 0s - loss: 16.5442 - val_loss: 16.8181
Epoch 419/1000
318/318 - 0s - loss: 16.4669 - val_loss: 16.7171
Epoch 420/1000
318/318 - 0s - loss: 16.3923 - val_loss: 16.6167
Epoch 421/1000
318/318 - 0s - loss: 16.3185 - val_loss: 16.5166
Epoch 422/1000
318/318 - 0s - loss: 16.2425 - val_loss: 16.4175
Epoch 423/1000
318/318 - 0s - loss: 16.1677 - val_loss: 16.3202
Epoch 424/1000
318/318 - 0s - loss: 16.0978 - val_loss: 16.2235
Epoch 425/1000
318/318 - 0s - loss: 16.0188 - val_loss: 16.1290
Epoch 426/1000
318/318 - 0s - loss: 15.9492 - val_loss: 16.0323
Epoch 427/1000
318/318 - 0s - loss: 15.8774 - val_loss: 15.9370
Epoch 428/1000
318/318 - 0s - loss: 15.8076 - val_loss: 15.8400
Epoch 429/1000
318/318 - 0s - loss: 15.7331 - val_loss: 15.7454
Epoch 430/1000
318/318 - 0s - loss: 15.6664 - val_loss: 15.6493
Epoch 431/1000
318/318 - 0s - loss: 15.5924 - val_loss: 15.5566
Epoch 432/1000
318/318 - 0s - loss: 15.5296 - val_loss: 15.4623
Epoch 433/1000
318/318 - 0s - loss: 15.4579 - val_loss: 15.3721
Epoch 434/1000
318/318 - 0s - loss: 15.3905 - val_loss: 15.2840
Epoch 435/1000
318/318 - 0s - loss: 15.3244 - val_loss: 15.1976
Epoch 436/1000
318/318 - 0s - loss: 15.2592 - val_loss: 15.1134
Epoch 437/1000
318/318 - 0s - loss: 15.1949 - val_loss: 15.0288
Epoch 438/1000
318/318 - 0s - loss: 15.1299 - val_loss: 14.9439
Epoch 439/1000
318/318 - 0s - loss: 15.0664 - val_loss: 14.8601
Epoch 440/1000
318/318 - 0s - loss: 14.9996 - val_loss: 14.7766
Epoch 441/1000
318/318 - 0s - loss: 14.9370 - val_loss: 14.6945
Epoch 442/1000
318/318 - 0s - loss: 14.8749 - val_loss: 14.6078
Epoch 443/1000
318/318 - 0s - loss: 14.8103 - val_loss: 14.5225
```

```
Epoch 444/1000
318/318 - 0s - loss: 14.7526 - val_loss: 14.4379
Epoch 445/1000
318/318 - 0s - loss: 14.6908 - val_loss: 14.3521
Epoch 446/1000
318/318 - 0s - loss: 14.6272 - val_loss: 14.2703
Epoch 447/1000
318/318 - 0s - loss: 14.5679 - val_loss: 14.1878
Epoch 448/1000
318/318 - 0s - loss: 14.5068 - val_loss: 14.1063
Epoch 449/1000
318/318 - 0s - loss: 14.4507 - val_loss: 14.0259
Epoch 450/1000
318/318 - 0s - loss: 14.3942 - val_loss: 13.9459
Epoch 451/1000
318/318 - 0s - loss: 14.3371 - val_loss: 13.8685
Epoch 452/1000
318/318 - 0s - loss: 14.2769 - val_loss: 13.7930
Epoch 453/1000
318/318 - 0s - loss: 14.2208 - val_loss: 13.7169
Epoch 454/1000
318/318 - 0s - loss: 14.1676 - val_loss: 13.6398
Epoch 455/1000
318/318 - 0s - loss: 14.1094 - val_loss: 13.5650
Epoch 456/1000
318/318 - 0s - loss: 14.0555 - val_loss: 13.4905
Epoch 457/1000
318/318 - 0s - loss: 14.0011 - val_loss: 13.4159
Epoch 458/1000
318/318 - 0s - loss: 13.9477 - val_loss: 13.3420
Epoch 459/1000
318/318 - 0s - loss: 13.8928 - val_loss: 13.2698
Epoch 460/1000
318/318 - 0s - loss: 13.8406 - val_loss: 13.1980
Epoch 461/1000
318/318 - 0s - loss: 13.7876 - val_loss: 13.1246
Epoch 462/1000
318/318 - 0s - loss: 13.7332 - val_loss: 13.0532
Epoch 463/1000
318/318 - 0s - loss: 13.6863 - val_loss: 12.9792
Epoch 464/1000
318/318 - 0s - loss: 13.6315 - val_loss: 12.9069
Epoch 465/1000
318/318 - 0s - loss: 13.5776 - val_loss: 12.8355
Epoch 466/1000
318/318 - 0s - loss: 13.5290 - val_loss: 12.7626
Epoch 467/1000
318/318 - 0s - loss: 13.4772 - val_loss: 12.6919
Epoch 468/1000
318/318 - 0s - loss: 13.4255 - val_loss: 12.6207
Epoch 469/1000
318/318 - 0s - loss: 13.3765 - val_loss: 12.5500
Epoch 470/1000
318/318 - 0s - loss: 13.3257 - val_loss: 12.4799
Epoch 471/1000
318/318 - 0s - loss: 13.2738 - val_loss: 12.4127
Epoch 472/1000
318/318 - 0s - loss: 13.2301 - val_loss: 12.3437
Epoch 473/1000
318/318 - 0s - loss: 13.1804 - val_loss: 12.2769
Epoch 474/1000
318/318 - 0s - loss: 13.1318 - val_loss: 12.2121
Epoch 475/1000
318/318 - 0s - loss: 13.0843 - val_loss: 12.1488
Epoch 476/1000
```

```
318/318 - 0s - loss: 13.0393 - val_loss: 12.0863
Epoch 477/1000
318/318 - 0s - loss: 12.9913 - val_loss: 12.0235
Epoch 478/1000
318/318 - 0s - loss: 12.9471 - val_loss: 11.9604
Epoch 479/1000
318/318 - 0s - loss: 12.9023 - val_loss: 11.8985
Epoch 480/1000
318/318 - 0s - loss: 12.8554 - val_loss: 11.8370
Epoch 481/1000
318/318 - 0s - loss: 12.8094 - val_loss: 11.7751
Epoch 482/1000
318/318 - 0s - loss: 12.7654 - val_loss: 11.7130
Epoch 483/1000
318/318 - 0s - loss: 12.7215 - val_loss: 11.6524
Epoch 484/1000
318/318 - 0s - loss: 12.6770 - val_loss: 11.5925
Epoch 485/1000
318/318 - 0s - loss: 12.6341 - val_loss: 11.5327
Epoch 486/1000
318/318 - 0s - loss: 12.5917 - val_loss: 11.4731
Epoch 487/1000
318/318 - 0s - loss: 12.5446 - val_loss: 11.4136
Epoch 488/1000
318/318 - 0s - loss: 12.5075 - val_loss: 11.3517
Epoch 489/1000
318/318 - 0s - loss: 12.4656 - val_loss: 11.2922
Epoch 490/1000
318/318 - 0s - loss: 12.4223 - val_loss: 11.2354
Epoch 491/1000
318/318 - 0s - loss: 12.3811 - val_loss: 11.1783
Epoch 492/1000
318/318 - 0s - loss: 12.3403 - val_loss: 11.1244
Epoch 493/1000
318/318 - 0s - loss: 12.3013 - val_loss: 11.0695
Epoch 494/1000
318/318 - 0s - loss: 12.2638 - val_loss: 11.0155
Epoch 495/1000
318/318 - 0s - loss: 12.2217 - val_loss: 10.9641
Epoch 496/1000
318/318 - 0s - loss: 12.1845 - val_loss: 10.9139
Epoch 497/1000
318/318 - 0s - loss: 12.1462 - val_loss: 10.8619
Epoch 498/1000
318/318 - 0s - loss: 12.1064 - val_loss: 10.8104
Epoch 499/1000
318/318 - 0s - loss: 12.0711 - val_loss: 10.7581
Epoch 500/1000
318/318 - 0s - loss: 12.0313 - val_loss: 10.7044
Epoch 501/1000
318/318 - 0s - loss: 11.9927 - val_loss: 10.6496
Epoch 502/1000
318/318 - 0s - loss: 11.9564 - val_loss: 10.5947
Epoch 503/1000
318/318 - 0s - loss: 11.9212 - val_loss: 10.5382
Epoch 504/1000
318/318 - 0s - loss: 11.8839 - val_loss: 10.4847
Epoch 505/1000
318/318 - 0s - loss: 11.8450 - val_loss: 10.4342
Epoch 506/1000
318/318 - 0s - loss: 11.8104 - val_loss: 10.3848
Epoch 507/1000
318/318 - 0s - loss: 11.7742 - val_loss: 10.3362
Epoch 508/1000
318/318 - 0s - loss: 11.7404 - val_loss: 10.2879
```

```
Epoch 509/1000
318/318 - 0s - loss: 11.7086 - val_loss: 10.2412
Epoch 510/1000
318/318 - 0s - loss: 11.6743 - val_loss: 10.1954
Epoch 511/1000
318/318 - 0s - loss: 11.6401 - val_loss: 10.1528
Epoch 512/1000
318/318 - 0s - loss: 11.6077 - val_loss: 10.1085
Epoch 513/1000
318/318 - 0s - loss: 11.5756 - val_loss: 10.0620
Epoch 514/1000
318/318 - 0s - loss: 11.5433 - val_loss: 10.0173
Epoch 515/1000
318/318 - 0s - loss: 11.5118 - val_loss: 9.9715
Epoch 516/1000
318/318 - 0s - loss: 11.4773 - val_loss: 9.9251
Epoch 517/1000
318/318 - 0s - loss: 11.4465 - val_loss: 9.8777
Epoch 518/1000
318/318 - 0s - loss: 11.4145 - val_loss: 9.8319
Epoch 519/1000
318/318 - 0s - loss: 11.3820 - val_loss: 9.7894
Epoch 520/1000
318/318 - 0s - loss: 11.3530 - val_loss: 9.7458
Epoch 521/1000
318/318 - 0s - loss: 11.3207 - val_loss: 9.7028
Epoch 522/1000
318/318 - 0s - loss: 11.2901 - val_loss: 9.6599
Epoch 523/1000
318/318 - 0s - loss: 11.2610 - val_loss: 9.6191
Epoch 524/1000
318/318 - 0s - loss: 11.2299 - val_loss: 9.5794
Epoch 525/1000
318/318 - 0s - loss: 11.2010 - val_loss: 9.5398
Epoch 526/1000
318/318 - 0s - loss: 11.1727 - val_loss: 9.4984
Epoch 527/1000
318/318 - 0s - loss: 11.1428 - val_loss: 9.4569
Epoch 528/1000
318/318 - 0s - loss: 11.1149 - val_loss: 9.4141
Epoch 529/1000
318/318 - 0s - loss: 11.0849 - val_loss: 9.3720
Epoch 530/1000
318/318 - 0s - loss: 11.0580 - val_loss: 9.3310
Epoch 531/1000
318/318 - 0s - loss: 11.0283 - val_loss: 9.2923
Epoch 532/1000
318/318 - 0s - loss: 11.0032 - val_loss: 9.2539
Epoch 533/1000
318/318 - 0s - loss: 10.9736 - val_loss: 9.2165
Epoch 534/1000
318/318 - 0s - loss: 10.9497 - val_loss: 9.1775
Epoch 535/1000
318/318 - 0s - loss: 10.9196 - val_loss: 9.1374
Epoch 536/1000
318/318 - 0s - loss: 10.8937 - val_loss: 9.0963
Epoch 537/1000
318/318 - 0s - loss: 10.8662 - val_loss: 9.0577
Epoch 538/1000
318/318 - 0s - loss: 10.8409 - val_loss: 9.0185
Epoch 539/1000
318/318 - 0s - loss: 10.8128 - val_loss: 8.9821
Epoch 540/1000
318/318 - 0s - loss: 10.7867 - val_loss: 8.9457
Epoch 541/1000
```

```
318/318 - 0s - loss: 10.7619 - val_loss: 8.9094
Epoch 542/1000
318/318 - 0s - loss: 10.7366 - val_loss: 8.8723
Epoch 543/1000
318/318 - 0s - loss: 10.7088 - val_loss: 8.8359
Epoch 544/1000
318/318 - 0s - loss: 10.6849 - val_loss: 8.8001
Epoch 545/1000
318/318 - 0s - loss: 10.6606 - val_loss: 8.7653
Epoch 546/1000
318/318 - 0s - loss: 10.6375 - val_loss: 8.7286
Epoch 547/1000
318/318 - 0s - loss: 10.6115 - val_loss: 8.6955
Epoch 548/1000
318/318 - 0s - loss: 10.5874 - val_loss: 8.6630
Epoch 549/1000
318/318 - 0s - loss: 10.5627 - val_loss: 8.6321
Epoch 550/1000
318/318 - 0s - loss: 10.5414 - val_loss: 8.6015
Epoch 551/1000
318/318 - 0s - loss: 10.5176 - val_loss: 8.5693
Epoch 552/1000
318/318 - 0s - loss: 10.4941 - val_loss: 8.5400
Epoch 553/1000
318/318 - 0s - loss: 10.4710 - val_loss: 8.5101
Epoch 554/1000
318/318 - 0s - loss: 10.4505 - val_loss: 8.4784
Epoch 555/1000
318/318 - 0s - loss: 10.4290 - val_loss: 8.4492
Epoch 556/1000
318/318 - 0s - loss: 10.4070 - val_loss: 8.4205
Epoch 557/1000
318/318 - 0s - loss: 10.3868 - val_loss: 8.3897
Epoch 558/1000
318/318 - 0s - loss: 10.3660 - val_loss: 8.3598
Epoch 559/1000
318/318 - 0s - loss: 10.3464 - val_loss: 8.3295
Epoch 560/1000
318/318 - 0s - loss: 10.3246 - val_loss: 8.3006
Epoch 561/1000
318/318 - 0s - loss: 10.3057 - val_loss: 8.2721
Epoch 562/1000
318/318 - 0s - loss: 10.2845 - val_loss: 8.2453
Epoch 563/1000
318/318 - 0s - loss: 10.2649 - val_loss: 8.2198
Epoch 564/1000
318/318 - 0s - loss: 10.2475 - val_loss: 8.1922
Epoch 565/1000
318/318 - 0s - loss: 10.2254 - val_loss: 8.1658
Epoch 566/1000
318/318 - 0s - loss: 10.2052 - val_loss: 8.1394
Epoch 567/1000
318/318 - 0s - loss: 10.1887 - val_loss: 8.1101
Epoch 568/1000
318/318 - 0s - loss: 10.1673 - val_loss: 8.0799
Epoch 569/1000
318/318 - 0s - loss: 10.1480 - val_loss: 8.0516
Epoch 570/1000
318/318 - 0s - loss: 10.1286 - val_loss: 8.0213
Epoch 571/1000
318/318 - 0s - loss: 10.1099 - val_loss: 7.9930
Epoch 572/1000
318/318 - 0s - loss: 10.0905 - val_loss: 7.9637
Epoch 573/1000
318/318 - 0s - loss: 10.0711 - val_loss: 7.9351
```

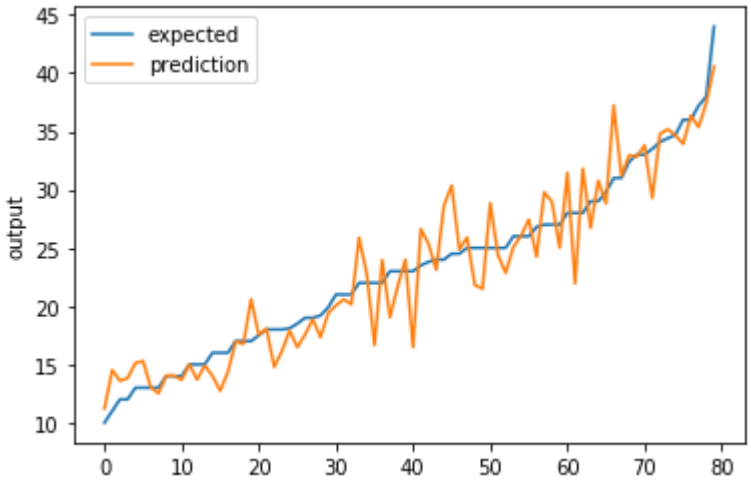
```
Epoch 574/1000
318/318 - 0s - loss: 10.0538 - val_loss: 7.9063
Epoch 575/1000
318/318 - 0s - loss: 10.0341 - val_loss: 7.8794
Epoch 576/1000
318/318 - 0s - loss: 10.0164 - val_loss: 7.8521
Epoch 577/1000
318/318 - 0s - loss: 9.9977 - val_loss: 7.8264
Epoch 578/1000
318/318 - 0s - loss: 9.9815 - val_loss: 7.8007
Epoch 579/1000
318/318 - 0s - loss: 9.9631 - val_loss: 7.7762
Epoch 580/1000
318/318 - 0s - loss: 9.9468 - val_loss: 7.7522
Epoch 581/1000
318/318 - 0s - loss: 9.9291 - val_loss: 7.7277
Epoch 582/1000
318/318 - 0s - loss: 9.9129 - val_loss: 7.7036
Epoch 583/1000
318/318 - 0s - loss: 9.8953 - val_loss: 7.6818
Epoch 584/1000
318/318 - 0s - loss: 9.8783 - val_loss: 7.6589
Epoch 585/1000
318/318 - 0s - loss: 9.8636 - val_loss: 7.6363
Epoch 586/1000
318/318 - 0s - loss: 9.8468 - val_loss: 7.6165
Epoch 587/1000
318/318 - 0s - loss: 9.8317 - val_loss: 7.5972
Epoch 588/1000
318/318 - 0s - loss: 9.8139 - val_loss: 7.5749
Epoch 589/1000
318/318 - 0s - loss: 9.7994 - val_loss: 7.5539
Epoch 590/1000
318/318 - 0s - loss: 9.7837 - val_loss: 7.5304
Epoch 591/1000
318/318 - 0s - loss: 9.7681 - val_loss: 7.5070
Epoch 592/1000
318/318 - 0s - loss: 9.7522 - val_loss: 7.4846
Epoch 593/1000
318/318 - 0s - loss: 9.7366 - val_loss: 7.4616
Epoch 594/1000
318/318 - 0s - loss: 9.7211 - val_loss: 7.4387
Epoch 595/1000
318/318 - 0s - loss: 9.7071 - val_loss: 7.4161
Epoch 596/1000
318/318 - 0s - loss: 9.6930 - val_loss: 7.3945
Epoch 597/1000
318/318 - 0s - loss: 9.6781 - val_loss: 7.3730
Epoch 598/1000
318/318 - 0s - loss: 9.6655 - val_loss: 7.3542
Epoch 599/1000
318/318 - 0s - loss: 9.6495 - val_loss: 7.3352
Epoch 600/1000
318/318 - 0s - loss: 9.6352 - val_loss: 7.3170
Epoch 601/1000
318/318 - 0s - loss: 9.6232 - val_loss: 7.2997
Epoch 602/1000
318/318 - 0s - loss: 9.6100 - val_loss: 7.2771
Epoch 603/1000
318/318 - 0s - loss: 9.5962 - val_loss: 7.2577
Epoch 604/1000
318/318 - 0s - loss: 9.5821 - val_loss: 7.2381
Epoch 605/1000
318/318 - 0s - loss: 9.5699 - val_loss: 7.2184
Epoch 606/1000
```

```
318/318 - 0s - loss: 9.5567 - val_loss: 7.2020
Epoch 607/1000
318/318 - 0s - loss: 9.5439 - val_loss: 7.1847
Epoch 608/1000
318/318 - 0s - loss: 9.5308 - val_loss: 7.1656
Epoch 609/1000
318/318 - 0s - loss: 9.5194 - val_loss: 7.1489
Epoch 610/1000
318/318 - 0s - loss: 9.5069 - val_loss: 7.1305
Epoch 611/1000
318/318 - 0s - loss: 9.4938 - val_loss: 7.1130
Epoch 612/1000
318/318 - 0s - loss: 9.4825 - val_loss: 7.0969
Epoch 613/1000
318/318 - 0s - loss: 9.4708 - val_loss: 7.0799
Epoch 614/1000
318/318 - 0s - loss: 9.4576 - val_loss: 7.0642
Epoch 615/1000
318/318 - 0s - loss: 9.4474 - val_loss: 7.0456
Epoch 616/1000
318/318 - 0s - loss: 9.4360 - val_loss: 7.0310
Epoch 617/1000
318/318 - 0s - loss: 9.4233 - val_loss: 7.0122
Epoch 618/1000
318/318 - 0s - loss: 9.4116 - val_loss: 6.9933
Epoch 619/1000
318/318 - 0s - loss: 9.3997 - val_loss: 6.9768
Epoch 620/1000
318/318 - 0s - loss: 9.3887 - val_loss: 6.9626
Epoch 621/1000
318/318 - 0s - loss: 9.3774 - val_loss: 6.9488
Epoch 622/1000
318/318 - 0s - loss: 9.3664 - val_loss: 6.9316
Epoch 623/1000
318/318 - 0s - loss: 9.3553 - val_loss: 6.9150
Epoch 624/1000
318/318 - 0s - loss: 9.3454 - val_loss: 6.8992
Epoch 625/1000
318/318 - 0s - loss: 9.3349 - val_loss: 6.8840
Epoch 626/1000
318/318 - 0s - loss: 9.3235 - val_loss: 6.8731
Epoch 627/1000
318/318 - 0s - loss: 9.3133 - val_loss: 6.8624
Epoch 628/1000
318/318 - 0s - loss: 9.3038 - val_loss: 6.8510
Epoch 629/1000
318/318 - 0s - loss: 9.2928 - val_loss: 6.8399
Epoch 630/1000
318/318 - 0s - loss: 9.2833 - val_loss: 6.8300
Epoch 631/1000
318/318 - 0s - loss: 9.2737 - val_loss: 6.8212
Epoch 632/1000
318/318 - 0s - loss: 9.2639 - val_loss: 6.8114
Epoch 633/1000
318/318 - 0s - loss: 9.2532 - val_loss: 6.7996
Epoch 634/1000
318/318 - 0s - loss: 9.2442 - val_loss: 6.7875
Epoch 635/1000
318/318 - 0s - loss: 9.2352 - val_loss: 6.7778
Epoch 636/1000
318/318 - 0s - loss: 9.2251 - val_loss: 6.7641
Epoch 637/1000
318/318 - 0s - loss: 9.2171 - val_loss: 6.7499
Epoch 638/1000
318/318 - 0s - loss: 9.2075 - val_loss: 6.7345
```

```
Epoch 639/1000
318/318 - 0s - loss: 9.1982 - val_loss: 6.7184
Epoch 640/1000
318/318 - 0s - loss: 9.1887 - val_loss: 6.7039
Epoch 641/1000
318/318 - 0s - loss: 9.1790 - val_loss: 6.6888
Epoch 642/1000
318/318 - 0s - loss: 9.1713 - val_loss: 6.6728
Epoch 643/1000
318/318 - 0s - loss: 9.1621 - val_loss: 6.6576
Epoch 644/1000
318/318 - 0s - loss: 9.1533 - val_loss: 6.6438
Epoch 645/1000
318/318 - 0s - loss: 9.1440 - val_loss: 6.6284
Epoch 646/1000
318/318 - 0s - loss: 9.1359 - val_loss: 6.6143
Epoch 647/1000
318/318 - 0s - loss: 9.1275 - val_loss: 6.5998
Epoch 648/1000
318/318 - 0s - loss: 9.1189 - val_loss: 6.5829
Epoch 649/1000
318/318 - 0s - loss: 9.1125 - val_loss: 6.5664
Epoch 650/1000
318/318 - 0s - loss: 9.1029 - val_loss: 6.5550
Epoch 651/1000
318/318 - 0s - loss: 9.0957 - val_loss: 6.5420
Epoch 652/1000
318/318 - 0s - loss: 9.0866 - val_loss: 6.5302
Epoch 653/1000
318/318 - 0s - loss: 9.0788 - val_loss: 6.5193
Epoch 654/1000
318/318 - 0s - loss: 9.0718 - val_loss: 6.5084
Epoch 655/1000
318/318 - 0s - loss: 9.0652 - val_loss: 6.4982
Epoch 656/1000
318/318 - 0s - loss: 9.0562 - val_loss: 6.4907
Epoch 657/1000
318/318 - 0s - loss: 9.0509 - val_loss: 6.4832
Epoch 658/1000
318/318 - 0s - loss: 9.0421 - val_loss: 6.4718
Epoch 659/1000
318/318 - 0s - loss: 9.0349 - val_loss: 6.4623
Epoch 660/1000
318/318 - 0s - loss: 9.0279 - val_loss: 6.4538
Epoch 661/1000
318/318 - 0s - loss: 9.0205 - val_loss: 6.4443
Epoch 662/1000
318/318 - 0s - loss: 9.0138 - val_loss: 6.4334
Epoch 663/1000
318/318 - 0s - loss: 9.0080 - val_loss: 6.4181
Epoch 664/1000
318/318 - 0s - loss: 9.0011 - val_loss: 6.4084
Epoch 665/1000
318/318 - 0s - loss: 8.9931 - val_loss: 6.3994
Epoch 666/1000
318/318 - 0s - loss: 8.9860 - val_loss: 6.3870
Epoch 667/1000
318/318 - 0s - loss: 8.9793 - val_loss: 6.3782
Epoch 668/1000
318/318 - 0s - loss: 8.9728 - val_loss: 6.3703
Epoch 669/1000
318/318 - 0s - loss: 8.9679 - val_loss: 6.3629
Epoch 670/1000
318/318 - 0s - loss: 8.9599 - val_loss: 6.3540
Epoch 671/1000
```

```
318/318 - 0s - loss: 8.9545 - val_loss: 6.3461
Epoch 672/1000
318/318 - 0s - loss: 8.9487 - val_loss: 6.3368
Epoch 673/1000
318/318 - 0s - loss: 8.9424 - val_loss: 6.3309
Epoch 674/1000
318/318 - 0s - loss: 8.9358 - val_loss: 6.3244
Epoch 675/1000
318/318 - 0s - loss: 8.9305 - val_loss: 6.3166
Epoch 676/1000
318/318 - 0s - loss: 8.9259 - val_loss: 6.3096
Epoch 677/1000
318/318 - 0s - loss: 8.9192 - val_loss: 6.3018
Epoch 678/1000
318/318 - 0s - loss: 8.9137 - val_loss: 6.2952
Epoch 679/1000
318/318 - 0s - loss: 8.9076 - val_loss: 6.2867
Epoch 680/1000
318/318 - 0s - loss: 8.9023 - val_loss: 6.2765
Epoch 681/1000
318/318 - 0s - loss: 8.8968 - val_loss: 6.2686
Epoch 682/1000
318/318 - 0s - loss: 8.8924 - val_loss: 6.2594
Epoch 683/1000
318/318 - 0s - loss: 8.8848 - val_loss: 6.2542
Epoch 684/1000
318/318 - 0s - loss: 8.8796 - val_loss: 6.2474
Epoch 685/1000
318/318 - 0s - loss: 8.8736 - val_loss: 6.2398
Epoch 686/1000
318/318 - 0s - loss: 8.8685 - val_loss: 6.2309
Epoch 687/1000
318/318 - 0s - loss: 8.8636 - val_loss: 6.2217
Epoch 688/1000
318/318 - 0s - loss: 8.8573 - val_loss: 6.2131
Epoch 689/1000
318/318 - 0s - loss: 8.8516 - val_loss: 6.2032
Epoch 690/1000
318/318 - 0s - loss: 8.8463 - val_loss: 6.1925
Epoch 691/1000
318/318 - 0s - loss: 8.8415 - val_loss: 6.1834
Epoch 692/1000
318/318 - 0s - loss: 8.8357 - val_loss: 6.1786
Epoch 693/1000
318/318 - 0s - loss: 8.8306 - val_loss: 6.1798
Epoch 694/1000
318/318 - 0s - loss: 8.8258 - val_loss: 6.1772
Epoch 695/1000
318/318 - 0s - loss: 8.8199 - val_loss: 6.1693
Epoch 696/1000
318/318 - 0s - loss: 8.8159 - val_loss: 6.1607
Epoch 697/1000
318/318 - 0s - loss: 8.8095 - val_loss: 6.1543
Epoch 698/1000
318/318 - 0s - loss: 8.8057 - val_loss: 6.1427
Epoch 699/1000
318/318 - 0s - loss: 8.7997 - val_loss: 6.1351
Epoch 700/1000
318/318 - 0s - loss: 8.7950 - val_loss: 6.1272
Epoch 701/1000
318/318 - 0s - loss: 8.7910 - val_loss: 6.1176
Epoch 702/1000
318/318 - 0s - loss: 8.7857 - val_loss: 6.1112
Epoch 703/1000
318/318 - 0s - loss: 8.7817 - val_loss: 6.1067
```

```
Epoch 704/1000
318/318 - 0s - loss: 8.7771 - val_loss: 6.0987
Epoch 705/1000
318/318 - 0s - loss: 8.7715 - val_loss: 6.0893
Epoch 706/1000
318/318 - 0s - loss: 8.7683 - val_loss: 6.0807
Epoch 707/1000
318/318 - 0s - loss: 8.7625 - val_loss: 6.0725
Epoch 708/1000
318/318 - 0s - loss: 8.7591 - val_loss: 6.0641
Epoch 709/1000
318/318 - 0s - loss: 8.7540 - val_loss: 6.0586
Epoch 710/1000
318/318 - 0s - loss: 8.7508 - val_loss: 6.0545
Epoch 711/1000
318/318 - 0s - loss: 8.7460 - val_loss: 6.0580
Epoch 712/1000
318/318 - 0s - loss: 8.7409 - val_loss: 6.0587
Epoch 713/1000
318/318 - 0s - loss: 8.7364 - val_loss: 6.0582
Epoch 714/1000
318/318 - 0s - loss: 8.7326 - val_loss: 6.0578
Epoch 715/1000
318/318 - 0s - loss: 8.7293 - val_loss: 6.0553
Epoch 00715: early stopping
Score (RMSE): 2.460583448410034
```



Try varying optimizer paramters and batch size after class.

References:

- Google Colab - Free web based platform that includes Python, Juypter Notebooks, and TensorFlow with free GPU support. No setup needed.
- IBM Cognitive Class Labs - Free web based platform that includes Python, Juypter Notebooks, and TensorFlow. No setup needed.
- Python Anaconda - Python distribution that includes many data science packages, such as Numpy, Scipy, Scikit-Learn, Pandas, and much more.
- TensorFlow - Google's mathematics package for deep learning.
- Kaggle - Competitive data science. Good source of sample data.
- T81-558: Applications of Deep Neural Networks. Instructor: Jeff Heaton

CSC 180 Intelligent Systems (Fall 2022)

Dr. Haiquan Chen, Dept of Computer Scicence

California State University, Sacramento

Lab 9: Convolutional Neural Networks

Helpful Functions for Tensorflow (Little Gems)

The following functions will be used with TensorFlow to help preprocess the data. They allow you to build the feature vector for a neural network.

- Predictors/Inputs
  - Fill any missing inputs with the median for that column. Use **missing_median**.
  - Encode textual/categorical values with **encode_text_dummy**.
  - Encode numeric values with **encode_numeric_zscore**.
- Output
  - Discard rows with missing outputs.
  - Encode textual/categorical values with **encode_text_index**.
  - Do not encode output numeric values.
- Produce final feature vectors (x) and expected output (y) with **to_xy**.

In [1]:
```
from collections.abc import Sequence
from sklearn import preprocessing
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import shutil
import os


# Encode text values to dummy variables(i.e. [1,0,0],[0,1,0],[0,0,1] for red,green,blue
def encode_text_dummy(df, name):
    dummies = pd.get_dummies(df[name])
    for x in dummies.columns:
        dummy_name = "{}-{}".format(name, x)
        df[dummy_name] = dummies[x]
    df.drop(name, axis=1, inplace=True)


# Encode text values to indexes(i.e. [1],[2],[3] for red,green,blue).
def encode_text_index(df, name):
    le = preprocessing.LabelEncoder()
    df[name] = le.fit_transform(df[name])
    return le.classes_


# Encode a numeric column as zscores
def encode_numeric_zscore(df, name, mean=None, sd=None):
    if mean is None:
```

```
        mean = df[name].mean()

    if sd is None:
        sd = df[name].std()

    df[name] = (df[name] - mean) / sd


# Convert all missing values in the specified column to the median
def missing_median(df, name):
    med = df[name].median()
    df[name] = df[name].fillna(med)


# Convert all missing values in the specified column to the default
def missing_default(df, name, default_value):
    df[name] = df[name].fillna(default_value)


# Convert a Pandas dataframe to the x,y inputs that TensorFlow needs
def to_xy(df, target):
    result = []
    for x in df.columns:
        if x != target:
            result.append(x)
    # find out the type of the target column.
    target_type = df[target].dtypes
    target_type = target_type[0] if isinstance(target_type, Sequence) else target_type
    # Encode to int for classification, float otherwise. TensorFlow likes 32 bits.
    if target_type in (np.int64, np.int32):
        # Classification
        dummies = pd.get_dummies(df[target])
        return df[result].values.astype(np.float32), dummies.values.astype(np.float32)
    else:
        # Regression
        return df[result].values.astype(np.float32), df[target].values.astype(np.float3

# 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)


# 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)
    a = plt.plot(t['y'].tolist(),label='expected')
    b = plt.plot(t['pred'].tolist(),label='prediction')
    plt.ylabel('output')
    plt.legend()
    plt.show()

# 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)
```

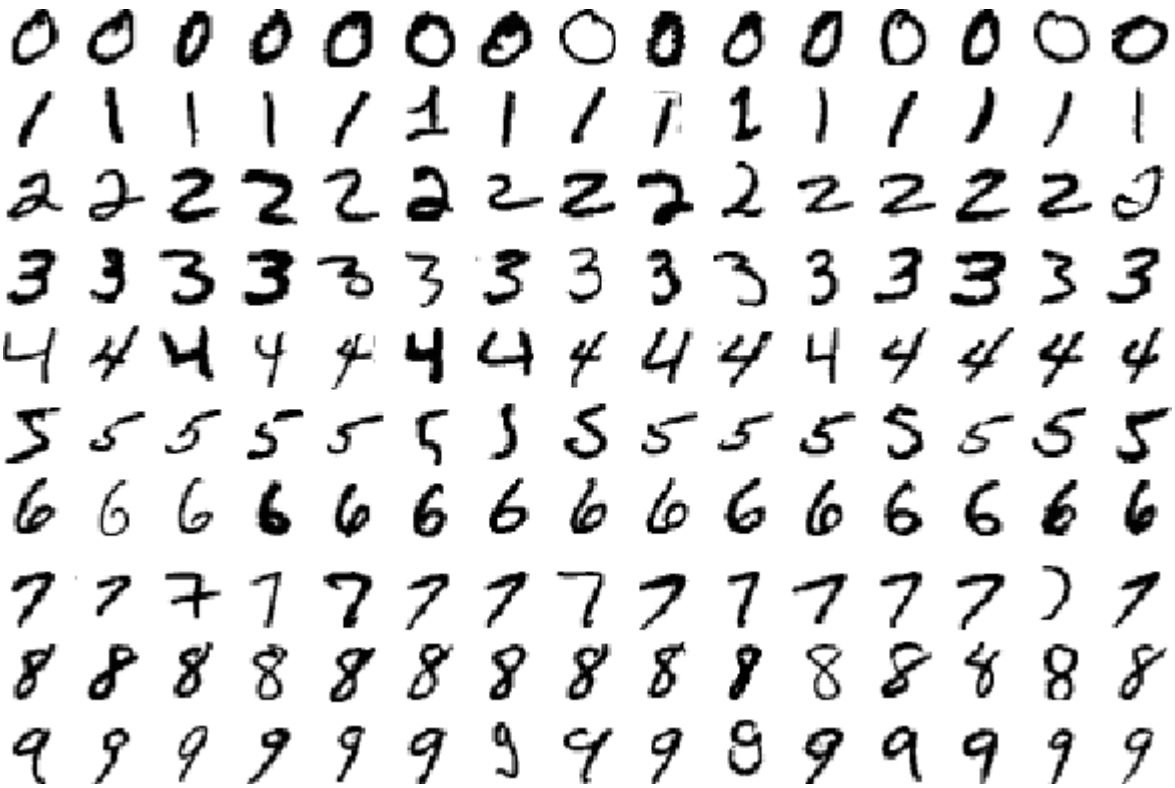
```
# Encode a column to a range between normalized_low and normalized_high.
def encode_numeric_range(df, name, normalized_low=-1, normalized_high=1,
                         data_low=None, data_high=None):
    if data_low is None:
        data_low = min(df[name])
        data_high = max(df[name])

    df[name] = ((df[name] - data_low) / (data_high - data_low)) \
               * (normalized_high - normalized_low) + normalized_low
```

Computer Vision Data Sets

There are many data sets for computer vision. Two of the most popular are the MNIST digits data set and the CIFAR image data sets.

MNIST Digits Data Set

The MNIST Digits Data Set is very popular in the neural network research community. A sample of it can be seen here:



Convolutional Neural Networks (CNNs)

The convolutional neural network (CNN) is a neural network technology that has profoundly impacted the area of computer vision (CV). In CNN, we use the following layer types:

- **Dense Layers** - Fully connected layers.
- **Convolution Layers** - Used to scan across images.
- **Max Pooling Layers** - Used to downsample images.
- **Dropout Layer** - Prevent overfitting
- **Flatten Layer** - Transform data of any shape into one dimensional

Convolution Layers

For the convolutional layer, We must specify the following hyper-parameters :

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

A filter is a square-shaped object that scans over the image. The more filters that we give to a convolutional layer, the more features it can detect.

Max Pooling Layers

Max-pool layers downsample the data. This technique can avoid overfitting.

Typically, you can always place a max-pool layer immediately following convolutional layer.

Example of Training CNNs with Tensorflow

The following sections describe how to use TensorFlow/Keras with CNNs.

Access to Data Sets

Keras provides built in access classes for MNIST which is already separated into two sets:

- **train** - Neural network will be trained with this.
- **test** - Used for validation.

In [2]:
```
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.datasets import mnist


(x_train, y_train), (x_test, y_test) = mnist.load_data()
print("Shape of x_train: {}".format(x_train.shape))
print("Shape of y_train: {}".format(y_train.shape))
print()
```

```
print("Shape of x_test: {}".format(x_test.shape))
print("Shape of y_test: {}".format(y_test.shape))
```

```
Shape of x_train: (60000, 28, 28)
Shape of y_train: (60000,)

Shape of x_test: (10000, 28, 28)
Shape of y_test: (10000,)
```

In [3]:
```
print("y_train: {}".format(y_train))
```

```
y_train: [5 0 4 ... 5 6 8]
```

Display the Digits

The following code shows what the MNIST files contain.

In [4]:
```
# Single MNIST digit
first = x_train[0]

pd.DataFrame(first)
```

Out[4]:

	0	1	2	3	4	5	6	7	8	9	...	18	19	20	21	22	23	24	25	26	27
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	...	175	26	166	255	247	127	0	0	0	0
6	0	0	0	0	0	0	0	0	30	36	...	225	172	253	242	195	64	0	0	0	0
7	0	0	0	0	0	0	0	49	238	253	...	93	82	82	56	39	0	0	0	0	0
8	0	0	0	0	0	0	0	18	219	253	...	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	80	156	...	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	14	...	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	...	25	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	...	150	27	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	...	253	187	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	...	253	249	64	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	...	253	207	2	0	0	0	0	0	0	0

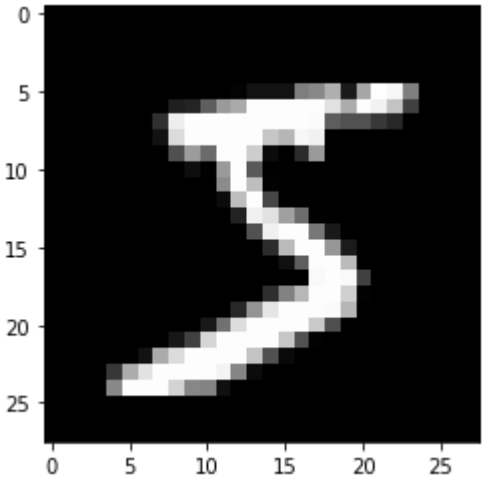
	0	1	2	3	4	5	6	7	8	9	...	18	19	20	21	22	23	24	25	26	27
19	0	0	0	0	0	0	0	0	0	0	...	250	182	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	...	78	0	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	23	66	...	0	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	18	171	219	253	...	0	0	0	0	0	0	0	0	0	0
23	0	0	0	0	55	172	226	253	253	253	...	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	136	253	253	253	212	135	...	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
27	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

28 rows × 28 columns

In [7]:

```
%matplotlib inline
import matplotlib.pyplot as plt

sample = 0    #  change this number to select another sample
digit = x_train[sample]
#print(type(digit))
#print(digit.shape)


plt.imshow(digit, cmap='gray')
print("Image (#{}): Which is digit '{}'".format(sample,y_train[sample]))
```

```
Image (#0): Which is digit '5'
```



Define CNN

In [8]:

```
# Load modules
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
```

```
from tensorflow.keras.layers import Dense, Dropout, Flatten
from tensorflow.keras.layers import Conv2D, MaxPooling2D
```

In [9]:
```
# Define batch_size and # of epochs
batch_size = 128
```

In [10]:
```
# define input image dimensions
img_rows, img_cols = 28, 28
```

Let's create x (images) first.

x must be 4D array

You must explicitly declare the depth of the input image. For example, a full-color image with all 3 RGB channels will have a depth of 3.

In other words, we need to transform our dataset from the shape (n, rows, cols) to (n, rows, cols, depth).

Our MNIST images only have a depth of 1, but we must explicitly declare that.

In [11]:
```
x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
```

In [12]:
```
print(x_train.shape)
print(x_test.shape)
```

```
(60000, 28, 28, 1)
(10000, 28, 28, 1)
```

In [13]:
```
x_train.dtype
```

Out[13]:
```
dtype('uint8')
```

In [14]:
```
# convert to float32 for normalization
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
```

In [15]:
```
# normalize the data values to the range [0, 1]
x_train /= 255
x_test /= 255
```

In [16]:
```
print('x_train shape:', x_train.shape)
print('x_test shape:', x_test.shape)
print("Training samples: {}".format(x_train.shape[0]))
print("Test samples: {}".format(x_test.shape[0]))
```

```
x_train shape: (60000, 28, 28, 1)
```

```
x_test shape: (10000, 28, 28, 1)
Training samples: 60000
Test samples: 10000
```

Now we have x ready. Let's create y (class labels)

In [17]:
```
print(y_train.shape)
print(y_train[:10])
```

```
(60000,)
[5 0 4 1 9 2 1 3 1 4]
```

For classification, TensorFlow requires y in one hot-encoded format.

Let's use function ***tf.keras.utils.to_categorical()*** to converts a class vector (integers) to its one hot-encoded format

In [18]:
```
num_classes = 10

# Converts a class vector (integers) to binary class matrix.   One-hot encoding!  Use w
y_train = tf.keras.utils.to_categorical(y_train, num_classes)
y_test = tf.keras.utils.to_categorical(y_test, num_classes)
```

In [20]:
```
pd.DataFrame(y_train)
```

Out[20]:

	0	1	2	3	4	5	6	7	8	9
0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
1	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
3	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
...	...	...	...	...	...	...	...	...	...	...
59995	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
59996	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
59997	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
59998	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
59999	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0

60000 rows × 10 columns

In [21]:
```
pd.DataFrame(y_test)
```

Out[21]:

	0	1	2	3	4	5	6	7	8	9
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0

	0	1	2	3	4	5	6	7	8	9
1	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...
9995	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
9996	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
9997	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
9998	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
9999	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0

10000 rows × 10 columns

In [22]:
```
y_train.shape, y_test.shape
```

Out[22]:
```
((60000, 10), (10000, 10))
```

We are ready to define the CNN architecture now!

Let's start by declaring a sequential model format:

In [23]:
```
model = Sequential()
```

Next, we add the input layer.

The Conv2D layer creates a convolution kernel that is convolved with the layer input to produce outputs. Provide the following:

- The number of convolution filters (neurons) to use
- kernel_size specifies the height and width of the 2D convolution window
- strides: An tuple of 2 integers, specifying the strides of the convolution along the height and width
- padding: one of "valid" or "same". ***valid***: no padding. ***same***: zero padding.
- ***input_shape: should be the shape of 1 sample.*** In this case, it should (28, 28, 1) that corresponds to the (rows, cols, depth) of each digit image. When using Conv2D layer as the first layer in a model, you must provide input_shape, e.g. input_shape=(128, 128, 3) for 128x128 RGB pictures.

In [24]:
```
input_shape = (img_rows, img_cols, 1)
```

```
model.add(Conv2D(32, kernel_size=(3, 3), strides=(1, 1), padding='valid',
                 activation='relu',
                 input_shape=input_shape))    #  in this case, input_shape = (img_rows,
```

In [25]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 32)        320
=================================================================
Total params: 320
Trainable params: 320
Non-trainable params: 0
_________________________________________________________________
```

Reflection:

Why the output shape is (None, 26, 26, 32) ?

The first dimension in a keras model is always the batch size.

Why the depth is 32?

Note that number of filters from previous layer become the number of channels for current layer's input image.

How 320 was calculated?

total_params* = *filter_height* $ $filter_width input_image_channels$ **$** **number_of_filters + number_of_filters***

Next, we can simply add more layers to our model

In [26]:
```
# Note that number of filters from previous layer become the number of channels for cur

model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=None))
model.add(Dropout(0.25))
```

MaxPooling2D is a way to reduce the number of parameters in our model.

Here we slide a 2x2 pooling filter across the previous layer and taking the max of the 4 values in the 2x2 filter.

strides: Integer, or None. Factor by which to downscale. E.g. 2 will halve the input. If None, it will default to pool_size.

Dropout layer is a method for regularizing our model in order to prevent overfitting

So far, for model parameters, we've added two Convolution layers. To complete our model architecture, let's add a fully connected layer and then the output layer:

In [27]:
```
model.add(Flatten())

model.add(Dense(128, activation='relu'))

model.add(Dropout(0.5))

model.add(Dense(num_classes, activation='softmax'))
```

For Dense layers, the first parameter is the output size of the layer.

Note that the final layer has an output size of 10, corresponding to the 10 classes of digits.

Why use flatten layer here?

Also note that Convolution layers must be flattened (made 1-dimensional) before passing them to the fully connected Dense layer.

In [28]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 32)        320
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 24, 24, 64)        18496
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 12, 12, 64)        0
_________________________________________________________________
dropout (Dropout)            (None, 12, 12, 64)        0
_________________________________________________________________
flatten (Flatten)            (None, 9216)              0
_________________________________________________________________
dense (Dense)                (None, 128)               1179776
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 1,199,882
Trainable params: 1,199,882
Non-trainable params: 0
_________________________________________________________________
```

Now all we need to do is to compile the model by defining the loss function and the optimizer, and then we'll be ready to train it.

Here for each epoch, we show not only "log loss" but also "accuracy" by using metrics=['accuracy']

In [29]:
```
from tensorflow.keras.optimizers import Adam

# show not only log loss but also accuracy for each epoch using metrics=['accuracy']
```

```
model.compile(loss=tf.keras.losses.categorical_crossentropy, optimizer=Adam(lr=0.001, d
```

In [30]:
```
x_train.shape
```

Out[30]:
```
(60000, 28, 28, 1)
```

Training/Fitting CNN

To fit the model, declare the batch size and number of epochs to train for, then pass in our training data.

This can take a while. You can also use a variety of callbacks to set early-stopping rules.

Check this while you are waiting ...

http://scs.ryerson.ca/~aharley/vis/conv/

In [31]:
```
import time

start_time = time.time()

# 1% of dataset

model.fit(x_train[0:600], y_train[0:600],
          batch_size=batch_size,
          epochs=10,
          verbose=2,
          validation_data=(x_test[0:100], y_test[0:100]))

elapsed_time = time.time() - start_time
print("Elapsed time: {}".format(hms_string(elapsed_time)))
```

```
Train on 600 samples, validate on 100 samples
Epoch 1/10
600/600 - 2s - loss: 2.0121 - accuracy: 0.3333 - val_loss: 1.3157 - val_accuracy: 0.7300
Epoch 2/10
600/600 - 1s - loss: 1.1199 - accuracy: 0.6367 - val_loss: 0.7664 - val_accuracy: 0.7300
Epoch 3/10
600/600 - 1s - loss: 0.8121 - accuracy: 0.7567 - val_loss: 0.6911 - val_accuracy: 0.7500
Epoch 4/10
600/600 - 1s - loss: 0.5879 - accuracy: 0.8183 - val_loss: 0.4993 - val_accuracy: 0.8300
Epoch 5/10
600/600 - 1s - loss: 0.5051 - accuracy: 0.8433 - val_loss: 0.4285 - val_accuracy: 0.8500
Epoch 6/10
600/600 - 1s - loss: 0.4165 - accuracy: 0.8717 - val_loss: 0.3696 - val_accuracy: 0.8600
Epoch 7/10
600/600 - 1s - loss: 0.3384 - accuracy: 0.8983 - val_loss: 0.2968 - val_accuracy: 0.9000
Epoch 8/10
600/600 - 1s - loss: 0.3106 - accuracy: 0.9067 - val_loss: 0.2734 - val_accuracy: 0.9300
Epoch 9/10
600/600 - 1s - loss: 0.2669 - accuracy: 0.9167 - val_loss: 0.2305 - val_accuracy: 0.9500
Epoch 10/10
600/600 - 1s - loss: 0.2198 - accuracy: 0.9383 - val_loss: 0.1958 - val_accuracy: 0.9600
Elapsed time: 0:00:09.08
```

Evaluate Accuracy in Tensorflow

In [32]:
```
# evaluate() computes the loss and accuracy
score = model.evaluate(x_test[0:100], y_test[0:100], verbose=0)
score
```

Out[32]:
```
[0.19584740400314332, 0.96]
```

In [33]:
```
print('Test loss: {}'.format(score[0]))
print('Test accuracy: {}'.format(score[1]))
```

```
Test loss: 0.19584740400314332
Test accuracy: 0.9599999785423279
```

Evaluate Other Metrics using Sklearn

In [34]:
```
from sklearn import metrics

y_true = np.argmax(y_test[0:500],axis=1)
pred = model.predict(x_test[0:500])
pred = np.argmax(pred,axis=1)


score = metrics.accuracy_score(y_true, pred)
print('Accuracy: {}'.format(score))


f1 = metrics.f1_score(y_true, pred, average='weighted')
print('Averaged F1: {}'.format(f1))


print(metrics.classification_report(y_true, pred))
```

```
Accuracy: 0.894
Averaged F1: 0.8934272757532905
              precision    recall  f1-score   support

           0       0.88      1.00      0.93        42
           1       0.99      1.00      0.99        67
           2       0.93      0.91      0.92        55
           3       0.93      0.91      0.92        45
           4       0.90      0.82      0.86        55
           5       0.87      0.82      0.85        50
           6       0.86      0.88      0.87        43
           7       0.83      0.88      0.85        49
           8       0.94      0.78      0.85        40
           9       0.82      0.91      0.86        54

    accuracy                           0.89       500
   macro avg       0.89      0.89      0.89       500
weighted avg       0.90      0.89      0.89       500
```

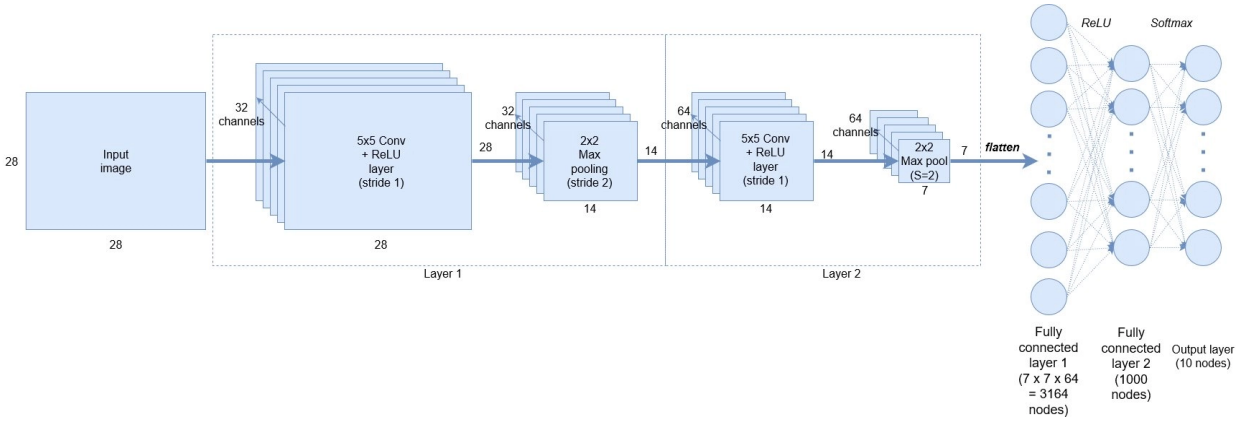
How to choose network archectures (# of layers, type of each layer, etc..)?

When you're just starting out, you should replicate proven architectures from academic papers or use existing examples. Here's a link to many example implementations in Keras/TensorFlow.

Proven and example implementation code for different applications for Keras/TensorFlow: https://keras.io/examples/

Please learn from those examples

For example, given this design,



we can replicate the model by using the following code:

In [35]:
```
model = Sequential()
model.add(Conv2D(32, kernel_size=(5, 5), strides=(1, 1),
                 activation='relu', padding='same',
                 input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Conv2D(64, (5, 5), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(1000, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))
```

In [36]:
```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 24, 24, 32)        832
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 12, 12, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 8, 8, 64)          51264
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 4, 4, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 1024)              0
_________________________________________________________________
dense_2 (Dense)              (None, 1000)              1025000
_________________________________________________________________
dense_3 (Dense)              (None, 10)                10010
=================================================================
```

```
Total params: 1,087,106
Trainable params: 1,087,106
Non-trainable params: 0
_________________________________________________________________
```

CNN can handle ANY data types other than images! The key is how you would view each sample as an image.

In [36]:

```
# x - inputs, 10000 samples of 128-dimensional vectors
# y - labels, 10000 samples of scalars from the set {0, 1, 2}

x = np.random.rand(10000, 128).astype("float32")
y = np.random.randint(3, size=(10000,1))


x
```

Out[36]:

```
array([[0.7638855 , 0.50253135, 0.7278408 , ..., 0.42962995, 0.19472997,
        0.3446575 ],
       [0.04477839, 0.58569694, 0.44175926, ..., 0.24039394, 0.29363278,
        0.1879978 ],
       [0.09363969, 0.5243248 , 0.30365306, ..., 0.8539248 , 0.570944  ,
        0.97114235],
       ...,
       [0.85849   , 0.6094952 , 0.38002342, ..., 0.8979433 , 0.42865613,
        0.99542755],
       [0.5361261 , 0.14472172, 0.08990218, ..., 0.80912924, 0.4939246 ,
        0.56559795],
       [0.91307765, 0.68031937, 0.65873593, ..., 0.44492722, 0.9569515 ,
        0.2510431 ]], dtype=float32)
```

In [37]:

```
# process the data to fit in a keras CNN properly,  input data needs to be (N, X, Y, C)
# N - number of samples
# C - number of channels per sample
# (X, Y) - sample size


x = x.reshape((10000, 1, 128, 1))

# output labels should be one-hot vectors - ie,
# 0 -> [0, 0, 1]
# 1 -> [0, 1, 0]
# 2 -> [1, 0, 0]
# this operation changes the shape of y from (10000,1) to (10000, 3)

y_one_hot = tf.keras.utils.to_categorical(y, 3)


# split your data to training and test
```

In [38]:

```
x
```

Out[38]:

```
array([[[[0.7638855 ],
         [0.50253135],
         [0.7278408 ],
         ...,
         [0.42962995],
```

```
        [0.19472997],
        [0.3446575 ]]],


       [[[0.04477839],
        [0.58569694],
        [0.44175926],
        ...,
        [0.24039394],
        [0.29363278],
        [0.1879978 ]]],


       [[[0.09363969],
        [0.5243248 ],
        [0.30365306],
        ...,
        [0.8539248 ],
        [0.570944  ],
        [0.97114235]]],


       ...,


       [[[0.85849   ],
        [0.6094952 ],
        [0.38002342],
        ...,
        [0.8979433 ],
        [0.42865613],
        [0.99542755]]],


       [[[0.5361261 ],
        [0.14472172],
        [0.08990218],
        ...,
        [0.80912924],
        [0.4939246 ],
        [0.56559795]]],


       [[[0.91307765],
        [0.68031937],
        [0.65873593],
        ...,
        [0.44492722],
        [0.9569515 ],
        [0.2510431 ]]]], dtype=float32)
```

In [39]:
```
y_one_hot
```

Out[39]:
```
array([[1., 0., 0.],
       [0., 0., 1.],
       [1., 0., 0.],
       ...,
       [0., 1., 0.],
       [1., 0., 0.],
       [1., 0., 0.]], dtype=float32)
```

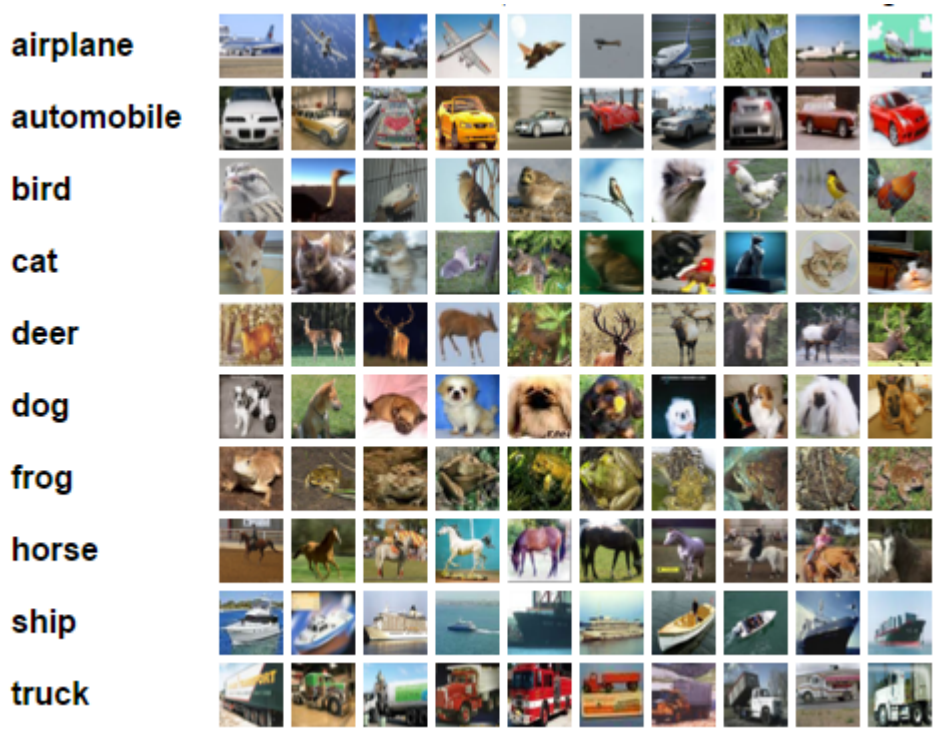
In [44]:
```
# define a CNN
```

```
cnn = Sequential()
cnn.add(Conv2D(64, kernel_size=(1, 3), strides=(1, 1),
                 activation='relu',
                 input_shape=(1, 128, 1)))

# the above code is equivalent to
# model.add(Conv1D(64, kernel_size=3, strides=1, activation='relu', input_shape=(128, 1

cnn.add(MaxPooling2D(pool_size=(1,2)))

cnn.add(Conv2D(128, kernel_size=(1, 3), strides=(1, 1),
                 activation='relu'))
cnn.add(MaxPooling2D(pool_size=(1,2)))


cnn.add(Flatten())
cnn.add(Dense(1024, activation="relu"))
cnn.add(Dropout(0.5))
cnn.add(Dense(3, activation="softmax"))

# define optimizer and objective, compile cnn

cnn.compile(loss="categorical_crossentropy", optimizer="adam")
```

Use CNN to handle color (RGB) images

CIFAR Data Set

The CIFAR-10 and CIFAR-100 datasets are also frequently used by the neural network research community.



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

In [37]:
```
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Activation, Flatten
from tensorflow.keras.layers import Conv2D, MaxPooling2D
import os

num_classes = 10

# The data, split between train and test sets:
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

In [38]:
```
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
```

```
x_train shape: (50000, 32, 32, 3)
50000 train samples
10000 test samples
```

In [39]:
```
x_train
```

Out[39]:
```
array([[[[ 59,  62,  63],
         [ 43,  46,  45],
         [ 50,  48,  43],
         ...,
         [158, 132, 108],
         [152, 125, 102],
         [148, 124, 103]],

        [[ 16,  20,  20],
         [  0,   0,   0],
         [ 18,   8,   0],
         ...,
         [123,  88,  55],
         [119,  83,  50],
         [122,  87,  57]],

        [[ 25,  24,  21],
         [ 16,   7,   0],
         [ 49,  27,   8],
         ...,
         [118,  84,  50],
         [120,  84,  50],
         [109,  73,  42]],

        ...,

        [[208, 170,  96],
         [201, 153,  34],
         [198, 161,  26],
         ...,
         [160, 133,  70],
         [ 56,  31,   7],
         [ 53,  34,  20]],
```

```
 [[180, 139,  96],
  [173, 123,  42],
  [186, 144,  30],
  ...,
  [184, 148,  94],
  [ 97,  62,  34],
  [ 83,  53,  34]],

 [[177, 144, 116],
  [168, 129,  94],
  [179, 142,  87],
  ...,
  [216, 184, 140],
  [151, 118,  84],
  [123,  92,  72]]],


[[[154, 177, 187],
  [126, 137, 136],
  [105, 104,  95],
  ...,
  [ 91,  95,  71],
  [ 87,  90,  71],
  [ 79,  81,  70]],

 [[140, 160, 169],
  [145, 153, 154],
  [125, 125, 118],
  ...,
  [ 96,  99,  78],
  [ 77,  80,  62],
  [ 71,  73,  61]],

 [[140, 155, 164],
  [139, 146, 149],
  [115, 115, 112],
  ...,
  [ 79,  82,  64],
  [ 68,  70,  55],
  [ 67,  69,  55]],

 ...,

 [[175, 167, 166],
  [156, 154, 160],
  [154, 160, 170],
  ...,
  [ 42,  34,  36],
  [ 61,  53,  57],
  [ 93,  83,  91]],

 [[165, 154, 128],
  [156, 152, 130],
  [159, 161, 142],
  ...,
  [103,  93,  96],
  [123, 114, 120],
  [131, 121, 131]],

 [[163, 148, 120],
  [158, 148, 122],
  [163, 156, 133],
  ...,
  [143, 133, 139],
```

```
  [143, 134, 142],
  [143, 133, 144]]],


[[[255, 255, 255],
  [253, 253, 253],
  [253, 253, 253],
  ...,
  [253, 253, 253],
  [253, 253, 253],
  [253, 253, 253]],

 [[255, 255, 255],
  [255, 255, 255],
  [255, 255, 255],
  ...,
  [255, 255, 255],
  [255, 255, 255],
  [255, 255, 255]],

 [[255, 255, 255],
  [254, 254, 254],
  [254, 254, 254],
  ...,
  [254, 254, 254],
  [254, 254, 254],
  [254, 254, 254]],

 ...,

 [[113, 120, 112],
  [111, 118, 111],
  [105, 112, 106],
  ...,
  [ 72,  81,  80],
  [ 72,  80,  79],
  [ 72,  80,  79]],

 [[111, 118, 110],
  [104, 111, 104],
  [ 99, 106,  98],
  ...,
  [ 68,  75,  73],
  [ 70,  76,  75],
  [ 78,  84,  82]],

 [[106, 113, 105],
  [ 99, 106,  98],
  [ 95, 102,  94],
  ...,
  [ 78,  85,  83],
  [ 79,  85,  83],
  [ 80,  86,  84]]],


...,


[[[ 35, 178, 235],
  [ 40, 176, 239],
  [ 42, 176, 241],
  ...,
  [ 99, 177, 219],
  [ 79, 147, 197],
  [ 89, 148, 189]],
```

```
 [[ 57, 182, 234],
  [ 44, 184, 250],
  [ 50, 183, 240],
  ...,
  [156, 182, 200],
  [141, 177, 206],
  [116, 149, 175]],

 [[ 98, 197, 237],
  [ 64, 189, 252],
  [ 69, 192, 245],
  ...,
  [188, 195, 206],
  [119, 135, 147],
  [ 61,  79,  90]],

 ...,

 [[ 73,  79,  77],
  [ 53,  63,  68],
  [ 54,  68,  80],
  ...,
  [ 17,  40,  64],
  [ 21,  36,  51],
  [ 33,  48,  49]],

 [[ 61,  68,  75],
  [ 55,  70,  86],
  [ 57,  79, 103],
  ...,
  [ 24,  48,  72],
  [ 17,  35,  53],
  [  7,  23,  32]],

 [[ 44,  56,  73],
  [ 46,  66,  88],
  [ 49,  77, 105],
  ...,
  [ 27,  52,  77],
  [ 21,  43,  66],
  [ 12,  31,  50]]],


[[[189, 211, 240],
  [186, 208, 236],
  [185, 207, 235],
  ...,
  [175, 195, 224],
  [172, 194, 222],
  [169, 194, 220]],

 [[194, 210, 239],
  [191, 207, 236],
  [190, 206, 235],
  ...,
  [173, 192, 220],
  [171, 191, 218],
  [167, 190, 216]],

 [[208, 219, 244],
  [205, 216, 240],
  [204, 215, 239],
  ...,
  [175, 191, 217],
```

```
  [172, 190, 216],
  [169, 191, 215]],

 ...,

 [[207, 199, 181],
  [203, 195, 175],
  [203, 196, 173],
  ...,
  [135, 132, 127],
  [162, 158, 150],
  [168, 163, 151]],

 [[198, 190, 170],
  [189, 181, 159],
  [180, 172, 147],
  ...,
  [178, 171, 160],
  [175, 169, 156],
  [175, 169, 154]],

 [[198, 189, 173],
  [189, 181, 162],
  [178, 170, 149],
  ...,
  [195, 184, 169],
  [196, 189, 171],
  [195, 190, 171]]],


[[[229, 229, 239],
  [236, 237, 247],
  [234, 236, 247],
  ...,
  [217, 219, 233],
  [221, 223, 234],
  [222, 223, 233]],

 [[222, 221, 229],
  [239, 239, 249],
  [233, 234, 246],
  ...,
  [223, 223, 236],
  [227, 228, 238],
  [210, 211, 220]],

 [[213, 206, 211],
  [234, 232, 239],
  [231, 233, 244],
  ...,
  [220, 220, 232],
  [220, 219, 232],
  [202, 203, 215]],

 ...,

 [[150, 143, 135],
  [140, 135, 127],
  [132, 127, 120],
  ...,
  [224, 222, 218],
  [230, 228, 225],
  [241, 241, 238]],

 [[137, 132, 126],
```

```
         [130, 127, 120],
         [125, 121, 115],
         ...,
         [181, 180, 178],
         [202, 201, 198],
         [212, 211, 207]],

        [[122, 119, 114],
         [118, 116, 110],
         [120, 116, 111],
         ...,
         [179, 177, 173],
         [164, 164, 162],
         [163, 163, 161]]], dtype=uint8)
```

In [40]:

```
# Convert class vectors to one hot format
y_train = tf.keras.utils.to_categorical(y_train, num_classes)
y_test = tf.keras.utils.to_categorical(y_test, num_classes)
```

In [41]:

```
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

x_train /= 255
x_test /= 255
```

In [42]:

```
model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same',
                 input_shape=x_train.shape[1:]))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Dense(num_classes, activation="softmax"))
```

In [43]:

```
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_4 (Conv2D)            (None, 32, 32, 32)        896
_________________________________________________________________
activation (Activation)      (None, 32, 32, 32)        0
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 16, 16, 32)        0
_________________________________________________________________
dropout_2 (Dropout)          (None, 16, 16, 32)        0
```

```
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 16, 16, 64)        18496
_________________________________________________________________
activation_1 (Activation)    (None, 16, 16, 64)        0
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 8, 8, 64)          0
_________________________________________________________________
dropout_3 (Dropout)          (None, 8, 8, 64)          0
_________________________________________________________________
flatten_2 (Flatten)          (None, 4096)              0
_________________________________________________________________
dense_4 (Dense)              (None, 512)               2097664
_________________________________________________________________
activation_2 (Activation)    (None, 512)               0
_________________________________________________________________
dropout_4 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_5 (Dense)              (None, 10)                5130
=================================================================
Total params: 2,122,186
Trainable params: 2,122,186
Non-trainable params: 0
_________________________________________________________________
```

References:

- Google Colab - Free web based platform that includes Python, Juypter Notebooks, and TensorFlow with free GPU support. No setup needed.
- IBM Cognitive Class Labs - Free web based platform that includes Python, Juypter Notebooks, and TensorFlow. No setup needed.
- Python Anaconda - Python distribution that includes many data science packages, such as Numpy, Scipy, Scikit-Learn, Pandas, and much more.
- TensorFlow - Google's mathematics package for deep learning.
- Kaggle - Competitive data science. Good source of sample data.
- T81-558: Applications of Deep Neural Networks. Instructor: Jeff Heaton

CSC 180 Intelligent Systems (Fall 2022)

Dr. Haiquan Chen, Dept of Computer Scicence

California State University, Sacramento

Lab 10: Regularization and Feature Importance Analysis

Helpful Functions for Tensorflow (Little Gems)

The following functions will be used with TensorFlow to help preprocess the data. They allow you to build the feature vector for a neural network.

- Predictors/Inputs
  - Fill any missing inputs with the median for that column. Use **missing_median**.
  - Encode textual/categorical values with **encode_text_dummy**.
  - Encode numeric values with **encode_numeric_zscore**.
- Output
  - Discard rows with missing outputs.
  - Encode textual/categorical values with **encode_text_index**.
  - Do not encode output numeric values.
- Produce final feature vectors (x) and expected output (y) with **to_xy**.

In [1]:
```
from collections.abc import Sequence
from sklearn import preprocessing
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import os


# Encode text values to dummy variables(i.e. [1,0,0],[0,1,0],[0,0,1] for red,green,blue
def encode_text_dummy(df, name):
    dummies = pd.get_dummies(df[name])
    for x in dummies.columns:
        dummy_name = "{}-{}".format(name, x)
        df[dummy_name] = dummies[x]
    df.drop(name, axis=1, inplace=True)


# Encode text values to indexes(i.e. [1],[2],[3] for red,green,blue).
def encode_text_index(df, name):
    le = preprocessing.LabelEncoder()
    df[name] = le.fit_transform(df[name])
    return le.classes_


# Encode a numeric column as zscores
def encode_numeric_zscore(df, name, mean=None, sd=None):
    if mean is None:
        mean = df[name].mean()
```

```
    if sd is None:
        sd = df[name].std()

    df[name] = (df[name] - mean) / sd


# Convert all missing values in the specified column to the median
def missing_median(df, name):
    med = df[name].median()
    df[name] = df[name].fillna(med)


# Convert all missing values in the specified column to the default
def missing_default(df, name, default_value):
    df[name] = df[name].fillna(default_value)


# Convert a Pandas dataframe to the x,y inputs that TensorFlow needs
def to_xy(df, target):
    result = []
    for x in df.columns:
        if x != target:
            result.append(x)
    # find out the type of the target column.
    target_type = df[target].dtypes
    target_type = target_type[0] if isinstance(target_type, Sequence) else target_type
    # Encode to int for classification, float otherwise. TensorFlow likes 32 bits.
    if target_type in (np.int64, np.int32):
        # Classification
        dummies = pd.get_dummies(df[target])
        return df[result].values.astype(np.float32), dummies.values.astype(np.float32)
    else:
        # Regression
        return df[result].values.astype(np.float32), df[target].values.astype(np.float3

# 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)


# 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)
    a = plt.plot(t['y'].tolist(),label='expected')
    b = plt.plot(t['pred'].tolist(),label='prediction')
    plt.ylabel('output')
    plt.legend()
    plt.show()

# 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)
```

```
# Encode a column to a range between normalized_low and normalized_high.
def encode_numeric_range(df, name, normalized_low=-1, normalized_high=1,
                         data_low=None, data_high=None):
    if data_low is None:
        data_low = min(df[name])
        data_high = max(df[name])

    df[name] = ((df[name] - data_low) / (data_high - data_low)) \
               * (normalized_high - normalized_low) + normalized_low
```

L1 and L2 regularization techniques used in linear regression

L1 and L2 regularization are two common regularization techniques.

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

https://www.kaggle.com/uciml/autompg-dataset/home

The labeling on the origin column is 1 for domestic, 2 for Europe and 3 for Asia

In [3]:
```
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

path = "./data/"

filename_read = os.path.join(path,"auto-mpg.csv")
df = pd.read_csv(filename_read,na_values=['NA','?'])
df[0:20]
```

Out[3]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin	name
0	18.0	8	307.0	130.0	3504	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165.0	3693	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150.0	3436	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150.0	3433	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140.0	3449	10.5	70	1	ford torino
5	15.0	8	429.0	198.0	4341	10.0	70	1	ford galaxie 500
6	14.0	8	454.0	220.0	4354	9.0	70	1	chevrolet impala
7	14.0	8	440.0	215.0	4312	8.5	70	1	plymouth fury iii

	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin	name
8	14.0	8	455.0	225.0	4425	10.0	70	1	pontiac catalina
9	15.0	8	390.0	190.0	3850	8.5	70	1	amc ambassador dpl
10	15.0	8	383.0	170.0	3563	10.0	70	1	dodge challenger se
11	14.0	8	340.0	160.0	3609	8.0	70	1	plymouth 'cuda 340
12	15.0	8	400.0	150.0	3761	9.5	70	1	chevrolet monte carlo
13	14.0	8	455.0	225.0	3086	10.0	70	1	buick estate wagon (sw)
14	24.0	4	113.0	95.0	2372	15.0	70	3	toyota corona mark ii
15	22.0	6	198.0	95.0	2833	15.5	70	1	plymouth duster
16	18.0	6	199.0	97.0	2774	15.5	70	1	amc hornet
17	21.0	6	200.0	85.0	2587	16.0	70	1	ford maverick
18	27.0	4	97.0	88.0	2130	14.5	70	3	datsun pl510
19	26.0	4	97.0	46.0	1835	20.5	70	2	volkswagen 1131 deluxe sedan

In [4]:

```
# create feature vector
missing_median(df, 'horsepower')
df.drop('name', 1, inplace=True)

encode_text_dummy(df, 'origin')
df[0:20]
```

Out[4]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin-1	origin-2	origin-3
0	18.0	8	307.0	130.0	3504	12.0	70	1	0	0
1	15.0	8	350.0	165.0	3693	11.5	70	1	0	0
2	18.0	8	318.0	150.0	3436	11.0	70	1	0	0
3	16.0	8	304.0	150.0	3433	12.0	70	1	0	0
4	17.0	8	302.0	140.0	3449	10.5	70	1	0	0
5	15.0	8	429.0	198.0	4341	10.0	70	1	0	0
6	14.0	8	454.0	220.0	4354	9.0	70	1	0	0
7	14.0	8	440.0	215.0	4312	8.5	70	1	0	0
8	14.0	8	455.0	225.0	4425	10.0	70	1	0	0
9	15.0	8	390.0	190.0	3850	8.5	70	1	0	0

	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin-1	origin-2	origin-3
10	15.0	8	383.0	170.0	3563	10.0	70	1	0	0
11	14.0	8	340.0	160.0	3609	8.0	70	1	0	0
12	15.0	8	400.0	150.0	3761	9.5	70	1	0	0
13	14.0	8	455.0	225.0	3086	10.0	70	1	0	0
14	24.0	4	113.0	95.0	2372	15.0	70	0	0	1
15	22.0	6	198.0	95.0	2833	15.5	70	1	0	0
16	18.0	6	199.0	97.0	2774	15.5	70	1	0	0
17	21.0	6	200.0	85.0	2587	16.0	70	1	0	0
18	27.0	4	97.0	88.0	2130	14.5	70	0	0	1
19	26.0	4	97.0	46.0	1835	20.5	70	0	1	0

In [5]:
```
# Encode to a 2D matrix for training
x,y = to_xy(df,'mpg')

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

Naive Linear Regression

To understand L1/L2 regularization, it is good to start with linear regression. L1/L2 were first introduced for linear regression. They can also be used for neural networks.

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 [6]:
```
import sklearn
from sklearn.linear_model import LinearRegression

# Create linear regression
regressor = 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("Final score (RMSE): {}".format(score))
```

```
Final score (RMSE): 2.937156915664673
```

In [7]:
```
names = list(df.columns.values)

print(names)
```

```
['mpg', 'cylinders', 'displacement', 'horsepower', 'weight', 'acceleration', 'year', 'or
igin-1', 'origin-2', 'origin-3']
```

In [8]:
```
regressor.coef_
```

Out[8]:
```
array([-0.41654268,  0.02445622, -0.00778466, -0.00747326,  0.13812245,
        0.8012743 , -1.448374  ,  0.8185648 ,  0.6298092 ], dtype=float32)
```

In [9]:
```
regressor.intercept_
```

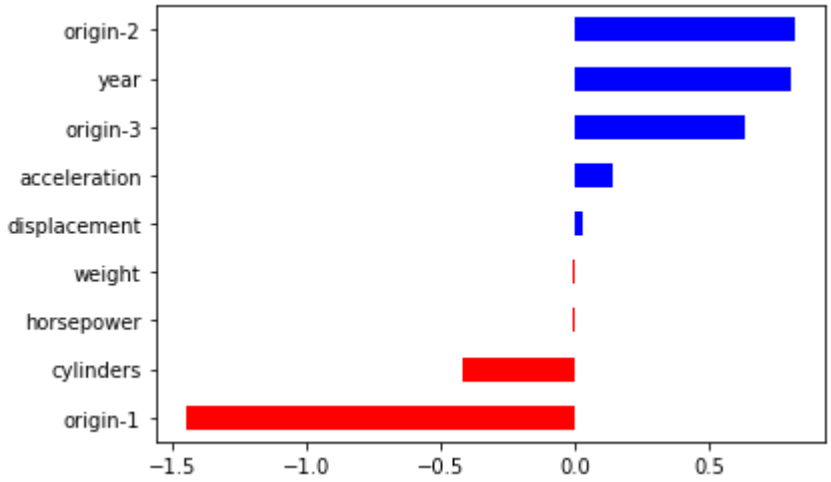
Out[9]:
```
-18.257578
```

In [10]:
```
# Simple function to evaluate the coefficients of a regression

%matplotlib inline
from IPython.display import display

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("Intercept: {}".format(intercept))
    r['coef'].plot(kind='barh', color=r['positive'].map({True: 'b', False: 'r'}))
```

In [11]:
```
names.remove("mpg")

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

	coef	positive
origin-1	-1.448374	False
cylinders	-0.416543	False
horsepower	-0.007785	False
weight	-0.007473	False
displacement	0.024456	True
acceleration	0.138122	True
origin-3	0.629809	True
year	0.801274	True
origin-2	0.818565	True

```
Intercept: -18.257577896118164
```



L1 (Lasso) Regularization

L1 Regularization, also called LASSO (Least Absolute Shrinkage and Selection Operator) should be used to create sparsity in the neural network.

L1 algorithm will push many weight connections to near 0.

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

The lower weight values will typically lead to less overfitting.

Minimization objective = SSE (Sum of Squared Error) + α * (Sum of Absolute Value of Coefficients)

When α is 0, Lasso regression produces the same coefficients as a linear regression. When α is very very large, all coefficients are zero.

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

In [12]:

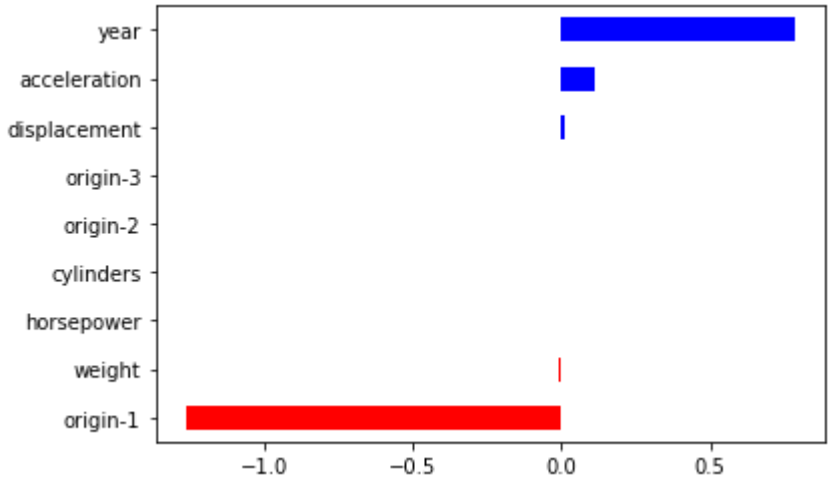
```
import sklearn
from sklearn.linear_model import Lasso

# Create linear regression
regressor = Lasso(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("Final score (RMSE): {}".format(score))
```

```
names = list(df.columns.values)
names.remove("mpg")

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

```
Final score (RMSE): 3.040905714035034
```

	coef	positive
origin-1	-1.264474	False
weight	-0.007458	False
horsepower	-0.002797	False
cylinders	-0.000000	True
origin-2	0.000000	True
origin-3	0.000000	True
displacement	0.013005	True
acceleration	0.113760	True
year	0.787195	True

```
Intercept: -17.271265029907227
```



Note: If your data set has a large number of input features that may not be needed, L1 regularization can help to detect and ignore unnecessary features.

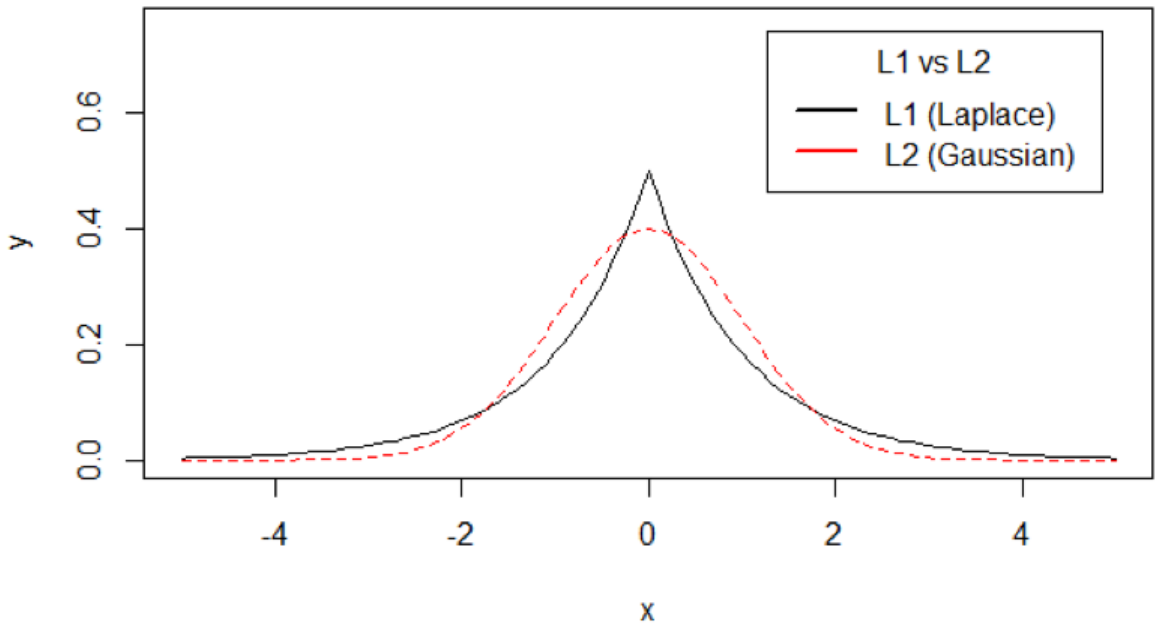
L2 (Ridge) Regularization

Use Ridge/L2 regularization when you prefer low weight values. ***The lower weight values will typically lead to less overfitting.***

Minimization objective = SSE (Sum of Squared Error) + α * (Sum of Square of Coefficients)

When α is 0, L2 regression produces the same coefficients as a linear regression. When α is very very large, all coefficients are zero.

L1 will force the weights into a pattern similar to a laplace distribution; the L2 will force the weights into a pattern similar to a Gaussian distribution***, as demonstrated the following:



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

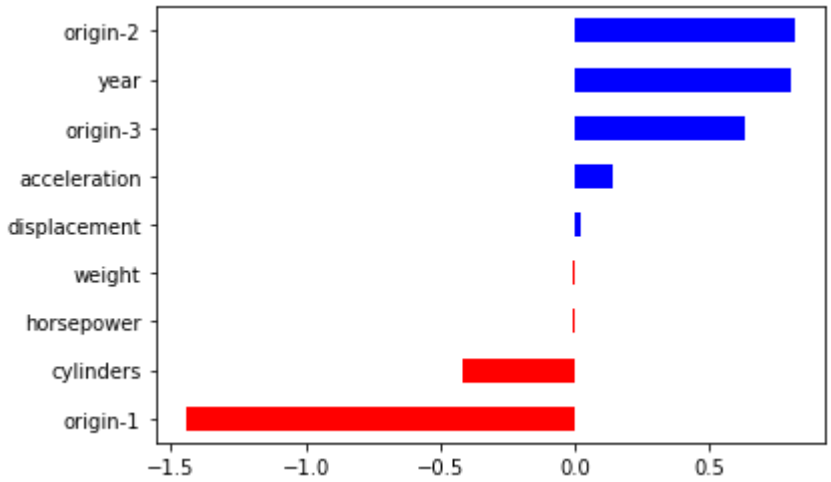
In [13]:
```
import sklearn
from sklearn.linear_model import Ridge

# Create linear regression
regressor = Ridge(alpha=0.1)

# 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): {}".format(score))

names = list(df.columns.values)
names.remove("mpg")

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

```
Final score (RMSE): 2.937572479248047
```

```
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear_model\ridge.py:147: LinAlgWarn
ing: Ill-conditioned matrix (rcond=3.88411e-10): result may not be accurate.
  overwrite_a=True).T
```

	coef	positive
origin-1	-1.444754	False
cylinders	-0.415851	False
horsepower	-0.007759	False
weight	-0.007473	False
displacement	0.024423	True
acceleration	0.138104	True
origin-3	0.629556	True
year	0.801221	True
origin-2	0.817690	True

```
Intercept: -18.25544548034668
```



ElasticNet: Linear regression with mixing L1 with L2.

- α: Constant for penalty (regularization).
- l1_ratio : The ElasticNet mixing parameter, with 0 <= l1_ratio <= 1. For l1_ratio = 0 the penalty is an L2 penalty. For l1_ratio = 1 it is an L1 penalty.

If you want the following: a $*$ L1 + b $*$ L2, set α = a + b and l1_ratio = a / (a + b)

In [14]:

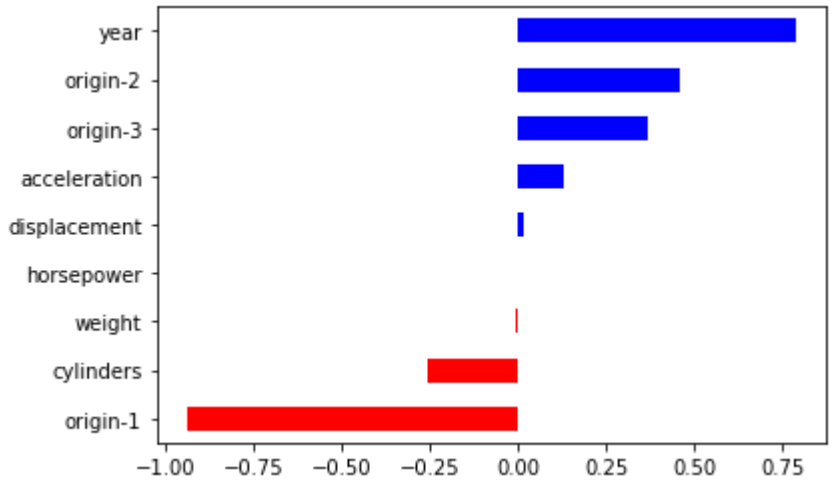
```
import sklearn
from sklearn.linear_model import ElasticNet

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

# Fit/train ElasticNet
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): {}".format(score))

names = list(df.columns.values)
names.remove("mpg")

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

```
Final score (RMSE): 3.031985282897949
```

	coef	positive
origin-1	-0.938924	False
cylinders	-0.257568	False
weight	-0.007462	False
horsepower	-0.002896	False
displacement	0.017533	True
acceleration	0.131320	True
origin-3	0.369088	True
origin-2	0.458725	True
year	0.788913	True

```
Intercept: -17.4800968170166
```



You can also apply such L1/L2 to get the feature importance for a classification problem. This can be done by training a Logistic Regression model and observe the coefficient learned for each feature.

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

The following is the sample code, which is just a little bit different than the way we did before for regression. In this code, we remove y-intercept. Note you should define x_train, y_train, x_test, y_test first to run the code.

In [ ]:
```
import sklearn
from sklearn.linear_model import LogisticRegression

# Create linear regression
regressor = LogisticRegression(penalty='elasticnet', solver='saga', l1_ratio=.1)

# Fit/train linear regression
regressor.fit(x_train,y_train)
```

In [3]:
```
# Simple function to evaluate the coefficients of a classification

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

In [ ]:
```
# Create the plot for the importance of each feature

names.remove("your_output_feature_goes_here")

report_coef(
  names,
  regressor.coef_[0])
```

L1/L2 in TensorFlow?

L1 and L2 regularization work by adding a weight penalty to the neural network training.

This penalty push the connection weights to small values.

In [16]:
```
########################################
# TensorFlow with L1/L2 for Regression
########################################
%matplotlib inline
from matplotlib.pyplot import figure, show
import tensorflow as tf
from sklearn.model_selection import train_test_split
import pandas as pd
import os
import numpy as np
from sklearn import metrics
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras import regularizers
from tensorflow.keras.models import Sequential
```

```
path = "./data/"


filename_read = os.path.join(path,"auto-mpg.csv")
df = pd.read_csv(filename_read,na_values=['NA','?'])

df.drop('name',1,inplace=True)
missing_median(df, 'horsepower')
encode_text_dummy(df, 'origin')
x,y = to_xy(df,"mpg")

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

model = Sequential()
```

You can set three regularization params for Dense, Conv1D, Conv2D and Conv3D.

keras.layers.Dense(units, activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None)

kernel_regularizer : Regularizer function applied to the kernel weights matrix. kernel regularizer will constantly decay the weights.

activity_regularizer : Regularizer function applied to the output of the layer. Activity regularizer will tend to make the output of the layer smaller.

bias_regularizer : Regularizer function applied to the bias vector.

https://keras.io/regularizers/

In [17]:
```
model.add(Dense(50, input_dim=x.shape[1], activation='relu'))
model.add(Dense(25, activation='relu'))

model.add(Dense(10,
                kernel_regularizer=regularizers.l1(0.01),
                activity_regularizer=regularizers.l2(0.01), activation='relu'))
```

In [18]:
```
model.add(Dense(1))   #  output layer

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

monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3, patience=5, verbose=1, mode

model.fit(x_train,y_train,validation_data=(x_test,y_test),callbacks=[monitor],verbose=2

pred = model.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): {}".format(score))
```

```
Train on 298 samples, validate on 100 samples
Epoch 1/100
298/298 - 1s - loss: 2508.1172 - val_loss: 1275.8206
Epoch 2/100
298/298 - 0s - loss: 1303.1712 - val_loss: 869.2227
Epoch 3/100
298/298 - 0s - loss: 792.3850 - val_loss: 586.1918
Epoch 4/100
298/298 - 0s - loss: 532.9360 - val_loss: 371.8480
Epoch 5/100
298/298 - 0s - loss: 337.8894 - val_loss: 226.0897
Epoch 6/100
298/298 - 0s - loss: 207.8855 - val_loss: 159.2584
Epoch 7/100
298/298 - 0s - loss: 162.6639 - val_loss: 125.0794
Epoch 8/100
298/298 - 0s - loss: 137.1347 - val_loss: 101.6161
Epoch 9/100
298/298 - 0s - loss: 129.7431 - val_loss: 102.7746
Epoch 10/100
298/298 - 0s - loss: 127.6511 - val_loss: 127.9809
Epoch 11/100
298/298 - 0s - loss: 129.9824 - val_loss: 95.9977
Epoch 12/100
298/298 - 0s - loss: 116.1929 - val_loss: 92.9806
Epoch 13/100
298/298 - 0s - loss: 112.7371 - val_loss: 95.0644
Epoch 14/100
298/298 - 0s - loss: 114.5570 - val_loss: 102.2564
Epoch 15/100
298/298 - 0s - loss: 112.0532 - val_loss: 97.3432
Epoch 16/100
298/298 - 0s - loss: 108.1101 - val_loss: 88.1528
Epoch 17/100
298/298 - 0s - loss: 104.0433 - val_loss: 88.8537
Epoch 18/100
298/298 - 0s - loss: 103.2577 - val_loss: 91.5117
Epoch 19/100
298/298 - 0s - loss: 102.0508 - val_loss: 85.6565
Epoch 20/100
298/298 - 0s - loss: 101.1060 - val_loss: 86.8096
Epoch 21/100
298/298 - 0s - loss: 100.7697 - val_loss: 87.4128
Epoch 22/100
298/298 - 0s - loss: 95.3531 - val_loss: 81.6606
Epoch 23/100
298/298 - 0s - loss: 98.5861 - val_loss: 81.1300
Epoch 24/100
298/298 - 0s - loss: 100.0143 - val_loss: 76.8305
Epoch 25/100
298/298 - 0s - loss: 99.9374 - val_loss: 72.2235
Epoch 26/100
298/298 - 0s - loss: 93.8908 - val_loss: 74.1515
Epoch 27/100
298/298 - 0s - loss: 85.0537 - val_loss: 66.1000
Epoch 28/100
298/298 - 0s - loss: 80.7311 - val_loss: 76.0448
Epoch 29/100
298/298 - 0s - loss: 83.3073 - val_loss: 66.7718
Epoch 30/100
298/298 - 0s - loss: 78.6246 - val_loss: 60.5064
Epoch 31/100
298/298 - 0s - loss: 73.7203 - val_loss: 67.0910
Epoch 32/100
```

```
298/298 - 0s - loss: 76.7629 - val_loss: 62.5588
Epoch 33/100
298/298 - 0s - loss: 68.5358 - val_loss: 58.3848
Epoch 34/100
298/298 - 0s - loss: 68.1094 - val_loss: 56.1986
Epoch 35/100
298/298 - 0s - loss: 64.3417 - val_loss: 63.9140
Epoch 36/100
298/298 - 0s - loss: 68.5390 - val_loss: 57.4534
Epoch 37/100
298/298 - 0s - loss: 64.0035 - val_loss: 58.3295
Epoch 38/100
298/298 - 0s - loss: 64.8909 - val_loss: 52.0084
Epoch 39/100
298/298 - 0s - loss: 58.4444 - val_loss: 51.7633
Epoch 40/100
298/298 - 0s - loss: 58.0050 - val_loss: 48.8300
Epoch 41/100
298/298 - 0s - loss: 55.9996 - val_loss: 57.1945
Epoch 42/100
298/298 - 0s - loss: 63.3284 - val_loss: 58.3766
Epoch 43/100
298/298 - 0s - loss: 61.1094 - val_loss: 47.9380
Epoch 44/100
298/298 - 0s - loss: 55.8948 - val_loss: 78.0072
Epoch 45/100
298/298 - 0s - loss: 62.3721 - val_loss: 44.1973
Epoch 46/100
298/298 - 0s - loss: 54.0161 - val_loss: 45.4007
Epoch 47/100
298/298 - 0s - loss: 50.1273 - val_loss: 42.4345
Epoch 48/100
298/298 - 0s - loss: 51.5962 - val_loss: 43.4980
Epoch 49/100
298/298 - 0s - loss: 48.6538 - val_loss: 40.3132
Epoch 50/100
298/298 - 0s - loss: 46.3198 - val_loss: 46.2679
Epoch 51/100
298/298 - 0s - loss: 47.4810 - val_loss: 41.8057
Epoch 52/100
298/298 - 0s - loss: 45.0930 - val_loss: 80.2193
Epoch 53/100
298/298 - 0s - loss: 60.7243 - val_loss: 56.3160
Epoch 54/100
298/298 - 0s - loss: 47.9144 - val_loss: 49.3288
Epoch 00054: early stopping
Final score (RMSE): 5.595123767852783
```

Dropout Layer (A dedicated layer for regularization)

Each dropout layer will drop neurons in its previous layer.

To create a dropout layer, specify dropout probability. The dropout probability indicates the likelihood of a neuron dropping out for every batch during training. Typically this value is 0.1 to 0.5.

Actually, a certain percentage of neurons will be masked during each training iteration. All neurons return after training is complete.

Animation that shows how dropout works

A dropout layer can be added between any two hidden layers to reduce overfitting.

TensorFlow Code with Dropout

In [19]:

```
############################################
# TensorFlow with Dropout for Regression
############################################


import tensorflow as tf
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 tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras import regularizers
from tensorflow.keras.models import Sequential

path = "./data/"


filename_read = os.path.join(path,"auto-mpg.csv")
df = pd.read_csv(filename_read,na_values=['NA','?'])

df.drop('name',1,inplace=True)

missing_median(df, 'horsepower')

encode_text_dummy(df, 'origin')

df[0:20]
```

Out[19]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin-1	origin-2	origin-3
0	18.0	8	307.0	130.0	3504	12.0	70	1	0	0
1	15.0	8	350.0	165.0	3693	11.5	70	1	0	0
2	18.0	8	318.0	150.0	3436	11.0	70	1	0	0
3	16.0	8	304.0	150.0	3433	12.0	70	1	0	0
4	17.0	8	302.0	140.0	3449	10.5	70	1	0	0
5	15.0	8	429.0	198.0	4341	10.0	70	1	0	0
6	14.0	8	454.0	220.0	4354	9.0	70	1	0	0
7	14.0	8	440.0	215.0	4312	8.5	70	1	0	0
8	14.0	8	455.0	225.0	4425	10.0	70	1	0	0
9	15.0	8	390.0	190.0	3850	8.5	70	1	0	0
10	15.0	8	383.0	170.0	3563	10.0	70	1	0	0

	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin-1	origin-2	origin-3
11	14.0	8	340.0	160.0	3609	8.0	70	1	0	0
12	15.0	8	400.0	150.0	3761	9.5	70	1	0	0
13	14.0	8	455.0	225.0	3086	10.0	70	1	0	0
14	24.0	4	113.0	95.0	2372	15.0	70	0	0	1
15	22.0	6	198.0	95.0	2833	15.5	70	1	0	0
16	18.0	6	199.0	97.0	2774	15.5	70	1	0	0
17	21.0	6	200.0	85.0	2587	16.0	70	1	0	0
18	27.0	4	97.0	88.0	2130	14.5	70	0	0	1
19	26.0	4	97.0	46.0	1835	20.5	70	0	1	0

In [30]:

```
x,y = to_xy(df,"mpg")
```

In [20]:

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

model = Sequential()
model.add(Dense(50, input_dim=x.shape[1]))
model.add(Dropout(0.1))

model.add(Dense(25, 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=1, mode
model.fit(x_train,y_train,validation_data=(x_test,y_test),callbacks=[monitor],verbose=2

pred = model.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): {}".format(score))
```

```
Train on 298 samples, validate on 100 samples
Epoch 1/1000
298/298 - 1s - loss: 5890.8973 - val_loss: 485.8900
Epoch 2/1000
298/298 - 0s - loss: 975.8011 - val_loss: 152.0852
Epoch 3/1000
298/298 - 0s - loss: 643.4201 - val_loss: 174.1349
Epoch 4/1000
298/298 - 0s - loss: 411.8962 - val_loss: 152.1615
Epoch 5/1000
298/298 - 0s - loss: 432.3020 - val_loss: 158.2570
Epoch 6/1000
298/298 - 0s - loss: 341.5154 - val_loss: 153.1775
Epoch 7/1000
298/298 - 0s - loss: 415.0385 - val_loss: 163.9636
```

```
Epoch 00007: early stopping
Final score (RMSE): 12.804829597473145
```

In [21]:

```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 50)                500
_________________________________________________________________
dropout (Dropout)            (None, 50)                0
_________________________________________________________________
dense_5 (Dense)              (None, 25)                1275
_________________________________________________________________
dense_6 (Dense)              (None, 10)                260
_________________________________________________________________
dense_7 (Dense)              (None, 1)                 11
=================================================================
Total params: 2,046
Trainable params: 2,046
Non-trainable params: 0
_________________________________________________________________
```

Reflection: How 500 and 1275 were calcuated?

1275 = 50*25 + 25

References:

- Google Colab - Free web based platform that includes Python, Juypter Notebooks, and TensorFlow with free GPU support. No setup needed.
- IBM Cognitive Class Labs - Free web based platform that includes Python, Juypter Notebooks, and TensorFlow. No setup needed.
- Python Anaconda - Python distribution that includes many data science packages, such as Numpy, Scipy, Scikit-Learn, Pandas, and much more.
- TensorFlow - Google's mathematics package for deep learning.
- Kaggle - Competitive data science. Good source of sample data.
- T81-558: Applications of Deep Neural Networks. Instructor: Jeff Heaton

CSC 180 Intelligent Systems (Fall 2022)

Dr. Haiquan Chen, Dept of Computer Scicence

California State University, Sacramento

Lab 11 Multi-output Regression and Auto-encoder

Helpful Functions for Tensorflow (Little Gems)

The following functions will be used with TensorFlow to help preprocess the data. They allow you to build the feature vector for a neural network.

- Predictors/Inputs
  - Fill any missing inputs with the median for that column. Use **missing_median**.
  - Encode textual/categorical values with **encode_text_dummy**.
  - Encode numeric values with **encode_numeric_zscore**.
- Output
  - Discard rows with missing outputs.
  - Encode textual/categorical values with **encode_text_index**.
  - Do not encode output numeric values.
- Produce final feature vectors (x) and expected output (y) with **to_xy**.

In [1]:
```
from collections.abc import Sequence
from sklearn import preprocessing
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import shutil
import os


# Encode text values to dummy variables(i.e. [1,0,0],[0,1,0],[0,0,1] for red,green,blue
def encode_text_dummy(df, name):
    dummies = pd.get_dummies(df[name])
    for x in dummies.columns:
        dummy_name = "{}-{}".format(name, x)
        df[dummy_name] = dummies[x]
    df.drop(name, axis=1, inplace=True)


# Encode text values to a single dummy variable.  The new columns (which do not replace
# at every location where the original column (name) matches each of the target_values.
# each target value.
def encode_text_single_dummy(df, name, target_values):
    for tv in target_values:
        l = list(df[name].astype(str))
        l = [1 if str(x) == str(tv) else 0 for x in l]
        name2 = "{}-{}".format(name, tv)
        df[name2] = l
```

```
# Encode text values to indexes(i.e. [1],[2],[3] for red,green,blue).
def encode_text_index(df, name):
    le = preprocessing.LabelEncoder()
    df[name] = le.fit_transform(df[name])
    return le.classes_


# Encode a numeric column as zscores
def encode_numeric_zscore(df, name, mean=None, sd=None):
    if mean is None:
        mean = df[name].mean()

    if sd is None:
        sd = df[name].std()

    df[name] = (df[name] - mean) / sd


# Convert all missing values in the specified column to the median
def missing_median(df, name):
    med = df[name].median()
    df[name] = df[name].fillna(med)


# Convert all missing values in the specified column to the default
def missing_default(df, name, default_value):
    df[name] = df[name].fillna(default_value)


# Convert a Pandas dataframe to the x,y inputs that TensorFlow needs
def to_xy(df, target):
    result = []
    for x in df.columns:
        if x != target:
            result.append(x)
    # find out the type of the target column.
    target_type = df[target].dtypes
    target_type = target_type[0] if isinstance(target_type, Sequence) else target_type
    # Encode to int for classification, float otherwise. TensorFlow likes 32 bits.
    if target_type in (np.int64, np.int32):
        # Classification
        dummies = pd.get_dummies(df[target])
        return df[result].values.astype(np.float32), dummies.values.astype(np.float32)
    else:
        # Regression
        return df[result].values.astype(np.float32), df[target].values.astype(np.float3

# 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)


# 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)
    a = plt.plot(t['y'].tolist(),label='expected')
    b = plt.plot(t['pred'].tolist(),label='prediction')
    plt.ylabel('output')
    plt.legend()
    plt.show()

# 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)


# Encode a column to a range between normalized_low and normalized_high.
def encode_numeric_range(df, name, normalized_low=-1, normalized_high=1,
                         data_low=None, data_high=None):
    if data_low is None:
        data_low = min(df[name])
        data_high = max(df[name])

    df[name] = ((df[name] - data_low) / (data_high - data_low)) \
               * (normalized_high - normalized_low) + normalized_low
```

Multi-Output Regression

Unlike most models, neural networks can provide multiple regression outputs. This allows a neural network to generate multiple outputs for the same input.

The following program uses a multi-output regression to predict both sin and cos from the same input data.

In [ ]:
```
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
from sklearn import metrics

x = np.sort((360 * np.random.rand(1000, 1)), axis=0)
y = np.array([np.pi * np.sin(x*(np.pi/180.0)).ravel(), np.pi * np.cos(x*(np.pi/180.0)).
```

In [2]:
```
x
```

Out[2]:
```
array([[  0.76741007],
       [  1.20739101],
       [  1.26095589],
       [  1.95544325],
       [  2.11602737],
       [  2.32218691],
       [  2.79272069],
       [  2.88119717],
       [  2.92739995],
       [  3.09465913],
       [  3.24189245],
```

```
[  3.79349791],
[  3.83791635],
[  3.85995538],
[  4.41416281],
[  4.49907736],
[  4.6258348 ],
[  5.15829531],
[  5.49057741],
[  5.52451371],
[  5.74834633],
[  6.64627623],
[  6.95896943],
[  7.01130322],
[  7.15840642],
[  7.41783892],
[  8.8064663 ],
[  9.93088256],
[  9.9527325 ],
[ 10.02590637],
[ 10.41755706],
[ 10.63030855],
[ 10.8152463 ],
[ 10.93999845],
[ 11.0460664 ],
[ 11.57839974],
[ 11.87776076],
[ 11.96355149],
[ 12.02957403],
[ 12.3822962 ],
[ 12.52550704],
[ 12.97942037],
[ 13.19410645],
[ 13.52827925],
[ 14.11414156],
[ 14.31402641],
[ 14.37881673],
[ 14.39550545],
[ 14.76748447],
[ 15.4079243 ],
[ 15.62677946],
[ 16.14606158],
[ 16.20654168],
[ 17.33309834],
[ 17.39499583],
[ 17.43663359],
[ 17.71647447],
[ 17.93624552],
[ 18.1178753 ],
[ 20.56696972],
[ 21.0376968 ],
[ 21.14761237],
[ 22.29280134],
[ 22.79716016],
[ 22.90191234],
[ 23.07902779],
[ 23.86307845],
[ 23.8912769 ],
[ 23.97073891],
[ 23.97237507],
[ 24.73505216],
[ 24.7853756 ],
[ 25.06133486],
[ 25.91096628],
[ 26.01666295],
[ 26.40203136],
```

```
[ 26.89546199],
[ 27.51873742],
[ 27.98477237],
[ 29.05893274],
[ 29.12660818],
[ 29.54146107],
[ 29.60626106],
[ 29.63607657],
[ 29.91671111],
[ 30.90012303],
[ 31.25726262],
[ 31.33392699],
[ 31.66712659],
[ 32.03778848],
[ 32.75632806],
[ 33.40591709],
[ 33.68173656],
[ 34.17245739],
[ 34.70147398],
[ 35.24419571],
[ 35.47962886],
[ 35.81604814],
[ 35.84673954],
[ 36.74678516],
[ 37.04199162],
[ 37.61999869],
[ 37.768224  ],
[ 37.94283046],
[ 38.01783826],
[ 38.0957667 ],
[ 38.32644356],
[ 38.37785408],
[ 38.69029069],
[ 39.20836941],
[ 39.43389818],
[ 39.51072269],
[ 39.593457  ],
[ 40.94931646],
[ 40.99319034],
[ 41.05301191],
[ 41.35015261],
[ 41.52808924],
[ 42.00067151],
[ 42.7625997 ],
[ 42.81237575],
[ 42.82662555],
[ 43.86183613],
[ 44.32825048],
[ 44.65273811],
[ 44.66986812],
[ 44.84038063],
[ 44.96688355],
[ 45.01087931],
[ 45.12600604],
[ 45.49953139],
[ 45.50353646],
[ 45.61737307],
[ 45.70763526],
[ 45.86285629],
[ 45.98582521],
[ 46.29910405],
[ 46.38861256],
[ 47.17120004],
[ 47.34038206],
[ 47.88939188],
```

```
[ 49.63436794],
[ 50.12384962],
[ 50.53377808],
[ 50.88868113],
[ 51.11936753],
[ 51.14244384],
[ 51.14307044],
[ 51.18316736],
[ 51.26891824],
[ 51.58408321],
[ 51.98743365],
[ 52.14465146],
[ 53.62624715],
[ 54.00975391],
[ 54.09887016],
[ 54.1669181 ],
[ 54.46468381],
[ 55.28524767],
[ 55.54024549],
[ 56.15008372],
[ 56.25157152],
[ 56.40687366],
[ 56.59889375],
[ 56.66596588],
[ 57.16083996],
[ 57.40979199],
[ 57.61420644],
[ 57.89050058],
[ 58.01810698],
[ 58.06554473],
[ 58.33662   ],
[ 58.82713418],
[ 58.91853465],
[ 59.35965702],
[ 59.56647356],
[ 59.62190347],
[ 60.25163852],
[ 60.62924772],
[ 61.64609228],
[ 62.55387826],
[ 63.20398754],
[ 63.63774171],
[ 63.81270861],
[ 63.98001538],
[ 64.25128064],
[ 64.33812258],
[ 64.78077811],
[ 65.15790914],
[ 65.57948166],
[ 65.65895954],
[ 65.9575641 ],
[ 65.98371565],
[ 67.08950146],
[ 67.22585222],
[ 67.4735909 ],
[ 68.24097619],
[ 68.25032859],
[ 68.28908052],
[ 68.95604327],
[ 69.44687802],
[ 69.87382264],
[ 70.59074012],
[ 70.72301048],
[ 71.07711761],
[ 71.59912372],
```

```
[ 71.85810439],
[ 71.92557229],
[ 72.17704221],
[ 72.48198957],
[ 72.80320497],
[ 73.05551043],
[ 73.13163103],
[ 73.37306056],
[ 73.40624712],
[ 74.55073205],
[ 74.56422514],
[ 74.5694559 ],
[ 74.64350244],
[ 75.43362578],
[ 76.34101087],
[ 77.81311879],
[ 78.51593519],
[ 79.57305563],
[ 79.97710149],
[ 79.9803631 ],
[ 80.22421422],
[ 80.71389058],
[ 80.81202319],
[ 80.91914129],
[ 80.96835511],
[ 81.43901503],
[ 81.62033251],
[ 82.22082206],
[ 82.7987293 ],
[ 83.74065861],
[ 84.14702703],
[ 84.73078781],
[ 85.1004471 ],
[ 85.10964441],
[ 85.15798919],
[ 85.62541339],
[ 86.10214181],
[ 86.79277957],
[ 86.81077121],
[ 87.08421366],
[ 87.24915212],
[ 88.07572559],
[ 88.21559746],
[ 88.2927323 ],
[ 89.47214117],
[ 89.90038449],
[ 89.97149164],
[ 90.09909203],
[ 90.16155392],
[ 90.92429898],
[ 90.97294047],
[ 91.68056202],
[ 93.52171379],
[ 93.99092034],
[ 95.04835809],
[ 95.22005721],
[ 96.03882086],
[ 97.13515446],
[ 97.22449284],
[ 97.48131318],
[ 98.09144387],
[ 98.18570823],
[ 98.36358311],
[ 98.40712884],
[ 98.4866011 ],
```

```
[ 98.66135898],
[ 98.80968165],
[ 99.01871408],
[ 99.57860983],
[ 99.73731195],
[ 99.9733581 ],
[100.38949418],
[100.50789214],
[100.97318022],
[101.05447982],
[101.36957195],
[101.57657838],
[102.70310216],
[103.17306241],
[103.32070521],
[103.38470476],
[103.72588606],
[103.7937935 ],
[103.98424377],
[104.52458845],
[105.02886196],
[105.18415731],
[105.23206801],
[105.55985443],
[106.73025453],
[107.01511461],
[107.52915116],
[108.24485793],
[108.5521851 ],
[108.99339461],
[109.47951957],
[109.76218125],
[110.07610606],
[110.17769605],
[111.17849219],
[111.71041577],
[112.87331876],
[112.89634989],
[113.79551989],
[114.25053132],
[114.56102935],
[114.9188916 ],
[115.1715118 ],
[115.7573052 ],
[116.36601933],
[116.488523  ],
[116.53276311],
[117.2990934 ],
[117.64149164],
[117.68674555],
[117.78495254],
[117.9588105 ],
[118.02335112],
[118.85595626],
[118.88812445],
[119.17048648],
[119.36932905],
[119.97349999],
[120.74572079],
[121.34635092],
[121.36061066],
[121.44812806],
[122.02905177],
[122.08647897],
[122.24760618],
```

[122.41330428],
[122.76214869],
[123.70605463],
[123.95485483],
[126.14893138],
[126.87726832],
[127.28236598],
[128.12231428],
[128.24220373],
[128.38535946],
[128.42596608],
[128.44238184],
[128.45154285],
[128.47878692],
[128.62021185],
[128.69350248],
[128.94352646],
[129.67532622],
[129.9261211],
[130.93094857],
[131.07506574],
[131.6473475],
[131.95256046],
[132.41793008],
[132.53997376],
[133.29871368],
[134.47796398],
[134.79761998],
[136.38454719],
[136.40018405],
[136.6273648],
[137.70609129],
[138.57846148],
[139.00930917],
[140.02567909],
[140.05669924],
[140.07317032],
[140.3501771],
[141.15733346],
[141.33483603],
[141.99482291],
[142.01965205],
[142.09592986],
[143.28236129],
[143.76807683],
[144.38869401],
[144.43228207],
[144.54213547],
[145.27437461],
[145.27641038],
[145.95472923],
[146.21007568],
[146.68800546],
[147.16420516],
[147.88933565],
[147.92936251],
[148.13925346],
[148.2276445],
[148.35582469],
[148.80724833],
[149.08370129],
[149.124848],
[149.29790115],
[149.40641679],
[149.63504653],

[150.54580556],
[150.55664533],
[151.08585129],
[152.22402273],
[152.48437989],
[152.9067255],
[153.36755767],
[153.5397327],
[154.38269892],
[155.12379673],
[155.21645038],
[155.31838004],
[155.43516348],
[155.43858789],
[155.55010713],
[155.90896054],
[156.55755373],
[156.62534509],
[157.16057884],
[157.92480969],
[159.06516614],
[159.22280474],
[159.60807849],
[159.8636203],
[160.57008274],
[161.30050889],
[161.79259455],
[162.43988365],
[162.45188178],
[163.06698943],
[164.1016372],
[164.12921824],
[164.47320867],
[164.59028841],
[164.98440157],
[165.61022767],
[166.10195612],
[166.43692339],
[167.31910287],
[167.34938147],
[167.94750197],
[168.12331349],
[168.22228675],
[168.42117637],
[168.63713605],
[169.26999762],
[169.72257735],
[169.75551521],
[170.03555761],
[170.93098168],
[171.11847681],
[172.22594185],
[172.38733501],
[172.50025475],
[172.53533703],
[172.543172],
[173.08047367],
[173.10848962],
[173.18022505],
[173.22690535],
[173.67846069],
[173.81861733],
[174.26589837],
[175.34051472],
[175.46045811],

```
[175.78055751],
[175.84120837],
[175.91195314],
[176.12465278],
[176.83023955],
[176.86624623],
[177.40190685],
[178.02921032],
[178.16130002],
[178.21441335],
[178.60464123],
[178.6804289 ],
[178.90505761],
[179.0859963 ],
[179.36685178],
[180.45438727],
[180.5644432 ],
[180.74802448],
[181.42963521],
[181.90634493],
[181.94811059],
[182.10094089],
[182.11256215],
[182.62270706],
[182.76487724],
[183.29875899],
[183.82882859],
[183.84124755],
[184.76434039],
[184.82019916],
[186.15120224],
[186.68394645],
[186.68513723],
[187.37730752],
[187.67672545],
[187.96307384],
[188.40025921],
[188.59409703],
[188.87732824],
[188.9962307 ],
[189.33942856],
[189.44798538],
[189.5679598 ],
[189.5760384 ],
[189.76363308],
[190.00491171],
[190.41990857],
[190.46773648],
[190.53182998],
[190.90621924],
[190.95245527],
[191.49492155],
[191.68889447],
[192.1602829 ],
[192.43774291],
[192.83030568],
[192.89672069],
[193.09881687],
[194.25295874],
[194.87507127],
[194.96819859],
[195.15003229],
[195.66178663],
[195.73368452],
[195.90839247],
```

[196.27762147],
[196.9492414],
[197.11529152],
[197.43590592],
[197.60315146],
[198.47081304],
[198.71116772],
[198.82843028],
[198.9093711],
[199.3704655],
[199.39546856],
[199.77276897],
[200.09107612],
[200.18725707],
[201.29908421],
[201.95869071],
[201.9963187],
[202.01979044],
[202.38776105],
[203.40446312],
[203.5286441],
[203.5773873],
[204.17792475],
[204.47978942],
[204.64774638],
[206.35980025],
[206.49706716],
[206.79594652],
[206.96592923],
[207.81881152],
[208.03524938],
[208.37582274],
[208.53427795],
[208.89666649],
[209.19535261],
[209.67891672],
[209.90894604],
[210.66331748],
[210.95066885],
[212.22735008],
[212.62736294],
[212.72630003],
[213.41627321],
[213.52039729],
[213.56272171],
[213.6519792],
[213.69608797],
[213.77366555],
[213.93928443],
[214.59930664],
[214.96812555],
[215.560775],
[215.69253972],
[215.89207312],
[216.12769033],
[216.25580854],
[216.44700269],
[216.99541538],
[217.51988733],
[218.3487918],
[218.52567989],
[218.54075323],
[218.56717458],
[218.82106391],
[219.04743419],

[219.23348951],
[219.8562352],
[221.31184302],
[221.85032737],
[222.17837443],
[222.449057],
[222.52723918],
[222.54542825],
[223.76511316],
[224.07745674],
[224.4362662],
[224.6038843],
[224.76268887],
[224.87671005],
[225.73121842],
[225.94252497],
[227.16784673],
[227.25832686],
[228.57569497],
[230.76399194],
[230.84047448],
[231.10489734],
[231.41190628],
[232.21338449],
[233.13683858],
[233.30195246],
[233.44988888],
[233.6678669],
[234.0930764],
[234.09833537],
[234.51371755],
[234.61523468],
[234.98663308],
[235.08310369],
[235.10685867],
[235.54837943],
[235.75643684],
[237.3823348],
[237.56247575],
[237.61412329],
[238.18895929],
[239.00625014],
[239.0213306],
[239.35970744],
[240.10050085],
[240.56086482],
[240.8753096],
[241.20941004],
[241.36683257],
[241.36703153],
[241.77358195],
[241.97152279],
[242.58567845],
[242.81874834],
[242.87474854],
[242.88847162],
[243.38169408],
[243.93010604],
[244.22226006],
[244.39295073],
[244.84455005],
[245.97831548],
[246.40898613],
[246.46390603],
[246.96557382],

[247.03590693],
[247.19277315],
[247.43399537],
[247.46460164],
[247.47568931],
[247.72642831],
[247.78312083],
[247.9931494],
[248.2228674],
[248.3465785],
[248.40136124],
[249.09894053],
[249.12153606],
[249.44471023],
[249.45408791],
[249.52708178],
[250.12447774],
[250.14138447],
[250.15335069],
[250.34585376],
[250.52364025],
[250.64468056],
[250.73018194],
[250.87129706],
[250.95491766],
[251.23105003],
[251.77159217],
[251.8738552],
[252.5087175],
[253.15085061],
[253.19406833],
[253.55678308],
[254.21217337],
[254.30696302],
[254.895257],
[255.08567168],
[255.46810737],
[256.19498122],
[256.20874393],
[256.63903496],
[256.73871213],
[257.34509735],
[257.56403514],
[257.80044856],
[257.90844626],
[258.00636926],
[258.07067929],
[259.36767539],
[259.86835562],
[260.16273508],
[260.19176189],
[260.27934587],
[260.74044322],
[260.83053971],
[261.27461813],
[261.54980226],
[261.73601446],
[262.20935336],
[262.41370741],
[263.39265562],
[263.43873771],
[263.71461852],
[264.5459762],
[265.10511316],
[265.51791071],

```
[265.53077177],
[265.85094839],
[265.86783366],
[266.04471718],
[266.3299425 ],
[266.79050587],
[266.87662225],
[268.64469248],
[268.77290935],
[269.83792368],
[270.38867962],
[271.07830514],
[271.33570872],
[271.46530539],
[271.77982151],
[272.00576196],
[272.51118505],
[272.87661083],
[273.08124593],
[273.20905196],
[273.57845605],
[273.88057196],
[274.13320073],
[274.43844221],
[274.86640222],
[275.04565505],
[275.59421778],
[275.6212721 ],
[276.15628375],
[276.64339646],
[276.73651319],
[277.16435763],
[277.26729996],
[277.42730207],
[277.51656983],
[277.54247335],
[277.97401972],
[278.08377991],
[278.19815217],
[278.9746191 ],
[279.16527857],
[279.8890091 ],
[280.04653013],
[280.13741832],
[280.16151732],
[280.30663934],
[280.56842223],
[280.79341836],
[281.6676837 ],
[281.74679448],
[282.01263797],
[282.92198682],
[282.93612656],
[283.54278019],
[283.71847342],
[283.99704328],
[284.61523124],
[284.81881141],
[284.83190087],
[285.07386659],
[285.76871112],
[286.16821366],
[286.34558047],
[286.35649233],
[286.6821375 ],
```

[287.11626177],
[287.28964866],
[287.38245501],
[287.86593814],
[287.90593713],
[287.98429185],
[288.38339619],
[288.41466795],
[288.51183212],
[289.0078231],
[289.0553056],
[289.20491862],
[289.94469393],
[290.06206613],
[290.28829381],
[291.34530364],
[291.38130342],
[291.88460885],
[292.41211122],
[292.45969545],
[293.09030774],
[293.53719033],
[293.66103963],
[294.12049766],
[294.57996367],
[294.66071253],
[294.68881317],
[294.96828605],
[295.5760217],
[295.72965805],
[296.0286249],
[296.10489702],
[296.34847886],
[296.48077358],
[296.48897897],
[297.64707296],
[298.09246041],
[298.23432896],
[298.42295203],
[298.82842165],
[299.4380061],
[299.59389187],
[299.83768631],
[300.7899139],
[300.98999179],
[300.99736391],
[301.02886947],
[301.08975782],
[301.12491139],
[301.25904648],
[301.3150667],
[301.43654802],
[301.70335427],
[302.06007442],
[302.22187818],
[302.38016379],
[302.43233572],
[302.58494352],
[302.62127864],
[303.92090605],
[304.55454201],
[304.57820893],
[305.30562889],
[305.36819131],
[305.38514208],

[305.42222324],
[306.50207476],
[306.5529001],
[307.01530228],
[307.22885767],
[307.25531089],
[307.35034125],
[307.63216989],
[308.02608976],
[308.03765165],
[308.05494541],
[308.3251408],
[308.93458872],
[309.02792369],
[309.14438459],
[309.58653484],
[309.74122657],
[310.00107501],
[310.40160913],
[310.57011778],
[310.57824378],
[311.18584585],
[311.44278957],
[311.764766],
[311.97452323],
[312.07434322],
[312.15405294],
[312.27407556],
[312.96432865],
[314.28588433],
[314.30973511],
[314.45987703],
[314.91633446],
[314.96236797],
[315.01897349],
[316.19909241],
[316.4036141],
[316.50693566],
[316.6224538],
[316.8878742],
[317.12859017],
[317.13119605],
[318.11976126],
[318.43361957],
[319.05642959],
[319.48269315],
[319.93318835],
[321.81989246],
[324.0270153],
[324.95940777],
[325.57688676],
[325.95056354],
[326.12572867],
[326.34046161],
[326.59561804],
[326.77532359],
[326.85647353],
[327.04711489],
[327.23347322],
[327.48690894],
[327.7610551],
[327.86590552],
[328.12323972],
[328.79480623],
[329.25347991],

[329.49379126],
[329.54061793],
[329.86479254],
[330.29685672],
[330.38383778],
[330.90428333],
[331.62914663],
[331.69897348],
[331.74833232],
[332.53932601],
[335.70897475],
[336.3943047],
[336.86931023],
[336.88576537],
[337.32275919],
[337.71737368],
[337.89063471],
[338.86665246],
[339.05951642],
[339.35837126],
[339.40223529],
[339.55656998],
[340.24615701],
[340.88608255],
[341.28537023],
[341.54571362],
[341.89712219],
[342.14100382],
[342.33277415],
[343.15494947],
[343.34491502],
[343.44951911],
[344.37640217],
[345.21337697],
[345.2388759],
[345.46452698],
[345.75670562],
[345.8896704],
[346.03250303],
[346.88037835],
[347.61218883],
[348.09391492],
[348.1545918],
[348.93512723],
[349.37854821],
[349.40679744],
[349.605368],
[349.72450414],
[350.22344439],
[350.81029512],
[350.89837579],
[350.91810538],
[351.14379452],
[351.35514578],
[351.58854616],
[351.89177316],
[352.06331876],
[352.23006478],
[352.49835866],
[353.14334672],
[353.41740421],
[353.68948168],
[353.78961952],
[353.85863756],
[353.90100945],

```
       [354.25792942],
       [354.27774377],
       [355.04393834],
       [355.98710772],
       [356.08540357],
       [356.11485496],
       [356.47944141],
       [357.1305418 ],
       [357.62042391],
       [357.66762988],
       [357.90271561],
       [358.21535936],
       [358.87864426],
       [359.47100262]])
```

In [3]:
```
y
```

Out[3]:
```
array([[ 0.04207671,  3.14131087],
       [ 0.06619772,  3.14089514],
       [ 0.06913406,  3.14083188],
       ...,
       [-0.09783805,  3.14006881],
       [-0.06148128,  3.140991  ],
       [-0.02900512,  3.14145875]])
```

In [4]:
```
model = Sequential()

model.add(Dense(100, input_dim=x.shape[1], activation='relu'))
model.add(Dense(50, activation='relu'))
model.add(Dense(25, activation='relu'))

model.add(Dense(2)) # Two output neurons

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

model.fit(x,y,verbose=0,batch_size=32,epochs=100)

model.summary()

# Fit regression DNN model.
pred = model.predict(x)
```

```
WARNING:tensorflow:From C:\Users\chenh\Anaconda3\lib\site-packages\tensorflow\python\ops
\init_ops.py:1251: calling VarianceScaling.__init__ (from tensorflow.python.ops.init_op
s) with dtype is deprecated and will be removed in a future version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the construct
or
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 100)               200
_________________________________________________________________
dense_1 (Dense)              (None, 50)                5050
_________________________________________________________________
dense_2 (Dense)              (None, 25)                1275
_________________________________________________________________
dense_3 (Dense)              (None, 2)                 52
=================================================================
Total params: 6,577
```

```
Trainable params: 6,577
Non-trainable params: 0
_________________________________________________________________
```

In [5]:

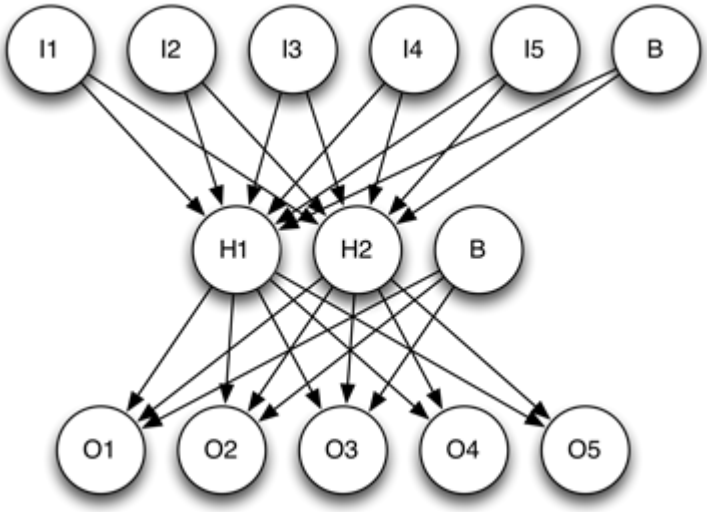
```
score = np.sqrt(metrics.mean_squared_error(pred, y))

print("Score (RMSE): {}".format(score))
print()
print()

print("Predicted:")
print(np.array(pred[0:5]))

print("Expected:")
print(np.array(y[0:5]))
```

```
Score (RMSE): 0.48017674058553966


Predicted:
[[ 4.0448204e-02  3.0171065e+00]
 [ 1.9946113e-02  3.0377409e+00]
 [ 1.7449871e-02  3.0402532e+00]
 [-2.6774257e-03  3.0690594e+00]
 [-9.3712658e-03  3.0755870e+00]]
Expected:
[[0.04207671 3.14131087]
 [0.06619772 3.14089514]
 [0.06913406 3.14083188]
 [0.10719836 3.13976319]
 [0.11599781 3.13945042]]
```

Simple Auto Encoder

An auto encoder is a neural network that has the same number of input neurons as output neurons.

The hidden layers of the neural network will have fewer neurons than the input/output neurons. Because there are fewer neurons, the auto-encoder must learn to encode the input to the fewer hidden neurons. The predictors (x) and output (y) are exactly the same in an auto encoder. Because of this, auto encoders are said to be unsupervised.



The following program demonstrates an auto encoder that learns to encode an image.

Reading Images in Python using Pillow

https://pillow.readthedocs.io/en/stable/index.html

The following code uses Pillow to load and display an image in RGB color model.

In [3]:

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

%matplotlib inline

url = "https://www.csus.edu/news/files/1_Campus_Spring_Flowers_Students_FB_20150330_001

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

print(np.asarray(img))
print('shape: ', np.asarray(img).shape)

img
```

```
[[[ 55  59   0]
  [ 73  77  18]
  [ 54  60   0]
  ...
  [ 72  78   4]
  [ 75  81   9]
  [ 55  61   0]]

 [[102 104  41]
  [ 53  58   0]
  [ 66  71   4]
```

```
          ...
          [ 63  70   0]
          [ 82  88  16]
          [ 96 102  32]]

         [[129 130  62]
          [ 75  76   6]
          [ 83  87  13]
          ...
          [ 95 102  24]
          [123 129  55]
          [ 62  68   0]]

         ...

         [[ 71  53  31]
          [ 88  70  48]
          [ 55  38  18]
          ...
          [ 84  81  28]
          [165 162 109]
          [155 152 101]]

         [[100  79  62]
          [ 89  68  51]
          [ 95  76  59]
          ...
          [124 119  64]
          [134 129  74]
          [153 147  95]]

         [[154 131 117]
          [142 120 106]
          [150 128 115]
          ...
          [115 110  54]
          [106 101  45]
          [112 107  51]]]
        shape:  (682, 1024, 3)
```

Out[3]:

Creating Images at pixel level

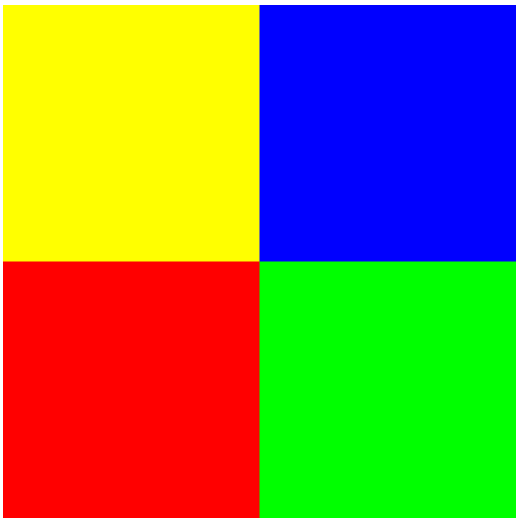
Pillow can also be used to create an image from a 3D numpy cube. The rows and columns specify the pixels. The depth, of 3, specifies red, green and blue. Here a simple image is created.

In [4]:

```
from PIL import Image
import numpy as np

w, h = 256, 256
data = np.zeros((h, w, 3), dtype=np.uint8)

# Yellow
for row in range(128):
    for col in range(128):
        data[row,col] = [255,255,0]

# Red
for row in range(128):
    for col in range(128):
        data[row+128,col] = [255,0,0]

# Green
for row in range(128):
    for col in range(128):
        data[row+128,col+128] = [0,255,0]

# Blue
for row in range(128):
    for col in range(128):
        data[row,col+128] = [0,0,255]
```

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

Out[4]:



Transform Images at the pixel level

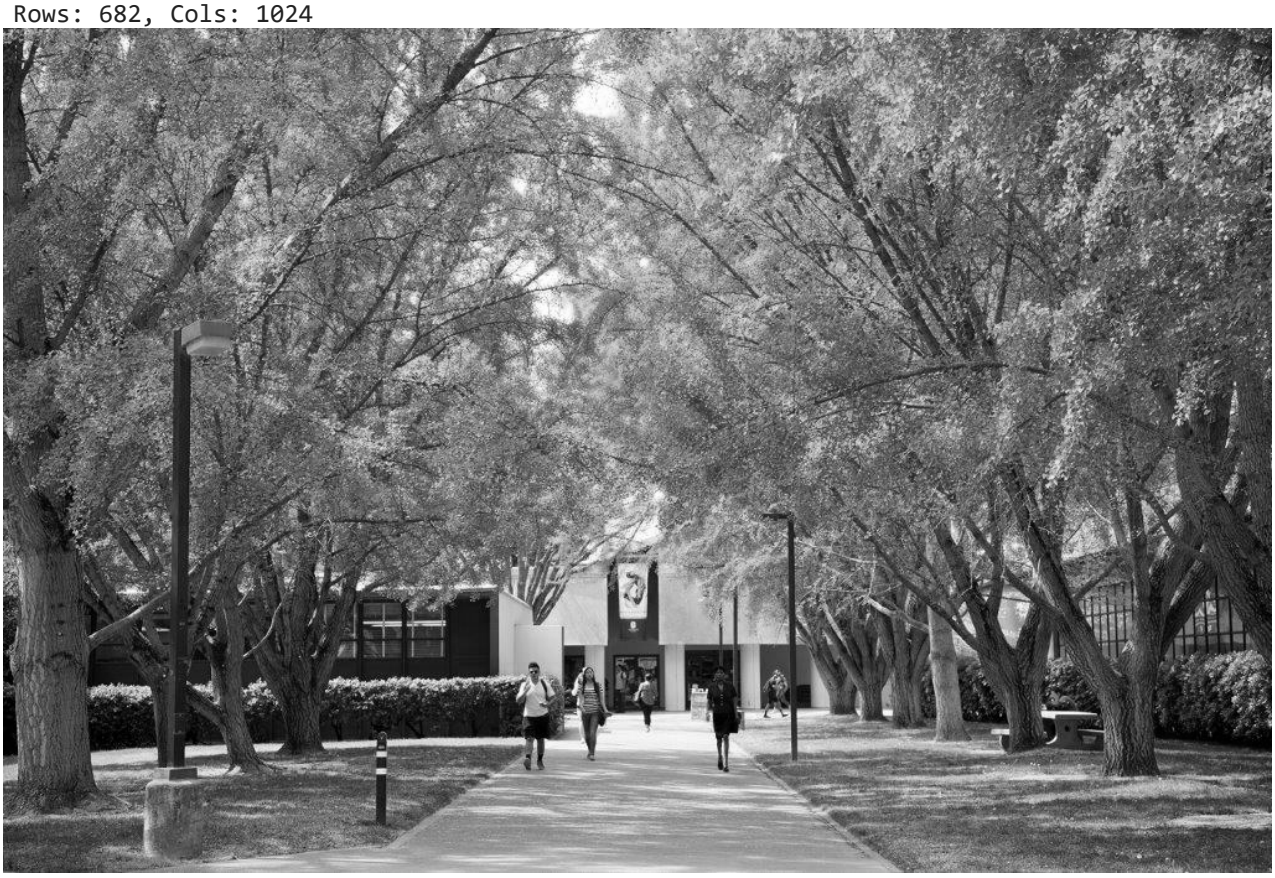
We can combine the last two programs and modify images. Here we take the mean color of each pixel and form a grayscale image.

In [5]:

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

%matplotlib inline

url = "https://www.csus.edu/news/files/1_Campus_Spring_Flowers_Students_FB_20150330_001

response = requests.get(url)
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: 682, Cols: 1024

Out[5]:



Demo about using auto encoders to handle noise

Adding Noise to an Image

Auto encoders can handle noise. First it is important 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.

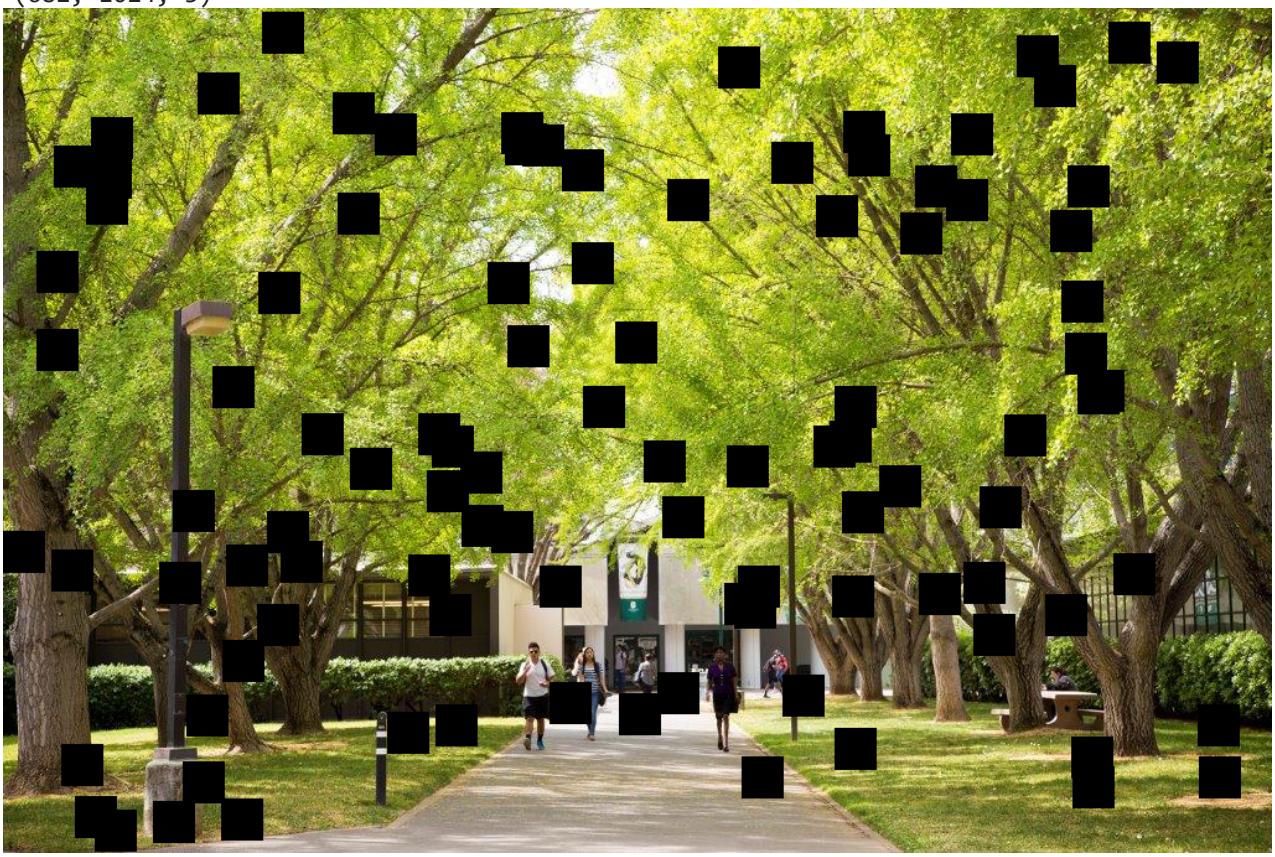
In [6]:

```
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 dimension

    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://www.csus.edu/news/files/1_Campus_Spring_Flowers_Students_FB_20150330_001

response = requests.get(url)
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 by adding noise
img_array_noise = add_noise(img_array)

# make sure it is in uint8
img_array_noise= img_array_noise.astype(np.uint8)

print(img_array_noise.shape)

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

```
Rows: 682, Cols: 1024
(682, 1024, 3)
```

Out[6]:



Denoising Autoencoder

A denoising auto encoder is designed to remove noise from input signals.

To do this ***the y becomes each image while the x becomes a version of y with noise added.***

- Downsampling images using resize() for code efficiency.

https://pillow.readthedocs.io/en/3.1.x/reference/Image.html

- The following code reads a sequence of images and resize them to the same size using ***img.resize()***. Image.LANCZOS is one of the high-quality downsampling filters you can use with resize().

We create 10 noisy versions of each image. The network is trained to convert noisy data (x) to the origional input (y).

In [8]:

```
%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://www.planetware.com/photos-large/USCA/california-san-francisco-golden-gate-
    "https://upload.wikimedia.org/wikipedia/commons/8/87/Golden_Gate_Bridge_Aerial.jpg"
    "https://www.travelbook.de/data/uploads/2017/09/gettyimages-585577624_1504953116-10
    "https://avatars.mds.yandex.net/get-pdb/27625/ea43da96-acce-4910-860b-27f8b3612dd9/
    "https://www.visittheusa.com/sites/default/files/styles/hero_m_1300x700/public/imag
]


x = []
y = []
loaded_images = []

for url in images:
    response = requests.get(url)
    img = Image.open(BytesIO(response.content))
    img.load()
    img = img.resize((256,256), Image.LANCZOS)

    loaded_images.append(img)
    print(url)
    display(img)
    for i in range(10):
        img_array = np.asarray(img)
        img_array_noise = add_noise(img_array)

        img_array = img_array.flatten()
        img_array = img_array.astype(np.float32)
        img_array = (img_array-256)/256

        img_array_noise = img_array_noise.flatten()
        img_array_noise = img_array_noise.astype(np.float32)
        img_array_noise = (img_array_noise-256)/256

        x.append(img_array_noise)
        y.append(img_array)
```

```
x = np.array(x)
y = np.array(y)

print(x.shape)
print(y.shape)
```

https://www.planetware.com/photos-large/USCA/california-san-francisco-golden-gate-bridge.jpg

https://upload.wikimedia.org/wikipedia/commons/8/87/Golden_Gate_Bridge_Aerial.jpg

https://www.travelbook.de/data/uploads/2017/09/gettyimages-585577624_1504953116-1040x690.jpg

https://avatars.mds.yandex.net/get-pdb/27625/ea43da96-acce-4910-860b-27f8b3612dd9/orig

https://www.visittheusa.com/sites/default/files/styles/hero_m_1300x700/public/images/hero_media_image/2017-05/23b0b0b9caaa07ee409b693da9bf9003.jpeg?itok=QUmOkiy_

```
(50, 196608)
(50, 196608)
```

Why (50, 196608)?

In [11]:

```
%matplotlib inline
from PIL import Image, ImageFile
import requests
from io import BytesIO
from sklearn import metrics
import numpy as np
import pandas as pd
import tensorflow as tf
from IPython.display import display, HTML


from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping

# Fit regression DNN model.
print("Creating/Training neural network")
model = Sequential()
model.add(Dense(100, input_dim=x.shape[1], activation='relu'))
model.add(Dense(50, activation='relu'))
```

```
model.add(Dense(100, activation='relu'))
model.add(Dense(x.shape[1])) # Multiple output neurons
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(x,y,verbose=1,epochs=200)

print("Neural network trained")
```

```
Creating/Training neural network
Epoch 1/200
50/50 [==============================] - 2s 43ms/sample - loss: 0.2797
Epoch 2/200
50/50 [==============================] - ETA: 0s - loss: 0.362 - 0s 8ms/sample - loss:
0.2690
Epoch 3/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.1177
Epoch 4/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0682
Epoch 5/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0550
Epoch 6/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0428
Epoch 7/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0419
Epoch 8/200
50/50 [==============================] - 0s 9ms/sample - loss: 0.0405
Epoch 9/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0386
Epoch 10/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0364
Epoch 11/200
50/50 [==============================] - 0s 9ms/sample - loss: 0.0360
Epoch 12/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0350
Epoch 13/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0333
Epoch 14/200
50/50 [==============================] - 0s 9ms/sample - loss: 0.0312
Epoch 15/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0293
Epoch 16/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0267
Epoch 17/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0247
Epoch 18/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0229
Epoch 19/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0216
Epoch 20/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0213
Epoch 21/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0212
Epoch 22/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0200
Epoch 23/200
50/50 [==============================] - 0s 9ms/sample - loss: 0.0187
Epoch 24/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0181
Epoch 25/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0165
Epoch 26/200
50/50 [==============================] - 0s 9ms/sample - loss: 0.0152
Epoch 27/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0147
Epoch 28/200
```

```
50/50 [==============================] - 0s 8ms/sample - loss: 0.0132
Epoch 29/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0119
Epoch 30/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0098
Epoch 31/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0081
Epoch 32/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0072
Epoch 33/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0057
Epoch 34/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0041
Epoch 35/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0032
Epoch 36/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0028
Epoch 37/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0029
Epoch 38/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0020
Epoch 39/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0016
Epoch 40/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0012
Epoch 41/200
50/50 [==============================] - 0s 9ms/sample - loss: 9.7609e-04
Epoch 42/200
50/50 [==============================] - 0s 8ms/sample - loss: 7.8794e-04
Epoch 43/200
50/50 [==============================] - 0s 8ms/sample - loss: 7.3140e-04
Epoch 44/200
50/50 [==============================] - 0s 8ms/sample - loss: 5.8793e-04
Epoch 45/200
50/50 [==============================] - 0s 8ms/sample - loss: 5.5378e-04
Epoch 46/200
50/50 [==============================] - 0s 8ms/sample - loss: 4.4545e-04
Epoch 47/200
50/50 [==============================] - 0s 8ms/sample - loss: 6.8475e-04
Epoch 48/200
50/50 [==============================] - 0s 8ms/sample - loss: 3.3254e-04
Epoch 49/200
50/50 [==============================] - 0s 8ms/sample - loss: 6.9560e-04
Epoch 50/200
50/50 [==============================] - 0s 8ms/sample - loss: 8.0177e-04
Epoch 51/200
50/50 [==============================] - 0s 8ms/sample - loss: 5.2261e-04
Epoch 52/200
50/50 [==============================] - 0s 8ms/sample - loss: 5.1893e-04
Epoch 53/200
50/50 [==============================] - 0s 8ms/sample - loss: 5.5882e-04
Epoch 54/200
50/50 [==============================] - 0s 8ms/sample - loss: 4.3982e-04
Epoch 55/200
50/50 [==============================] - 0s 9ms/sample - loss: 1.8339e-04
Epoch 56/200
50/50 [==============================] - 0s 8ms/sample - loss: 3.5883e-04
Epoch 57/200
50/50 [==============================] - 0s 8ms/sample - loss: 1.6857e-04
Epoch 58/200
50/50 [==============================] - 0s 8ms/sample - loss: 2.2602e-04
Epoch 59/200
50/50 [==============================] - 0s 8ms/sample - loss: 1.3599e-04
Epoch 60/200
50/50 [==============================] - 0s 8ms/sample - loss: 2.5501e-04
```

```
Epoch 61/200
50/50 [==============================] - 0s 9ms/sample - loss: 3.3145e-04
Epoch 62/200
50/50 [==============================] - 0s 9ms/sample - loss: 2.7080e-04
Epoch 63/200
50/50 [==============================] - 0s 9ms/sample - loss: 3.6519e-04
Epoch 64/200
50/50 [==============================] - 0s 8ms/sample - loss: 1.7133e-04
Epoch 65/200
50/50 [==============================] - 0s 8ms/sample - loss: 3.3761e-04
Epoch 66/200
50/50 [==============================] - 0s 8ms/sample - loss: 2.7367e-04
Epoch 67/200
50/50 [==============================] - 0s 8ms/sample - loss: 6.4144e-04
Epoch 68/200
50/50 [==============================] - 0s 8ms/sample - loss: 4.4272e-04
Epoch 69/200
50/50 [==============================] - 0s 8ms/sample - loss: 5.7949e-04
Epoch 70/200
50/50 [==============================] - 0s 8ms/sample - loss: 5.9376e-04
Epoch 71/200
50/50 [==============================] - 0s 8ms/sample - loss: 5.8315e-04
Epoch 72/200
50/50 [==============================] - 0s 8ms/sample - loss: 6.8826e-04
Epoch 73/200
50/50 [==============================] - 0s 8ms/sample - loss: 7.4079e-04
Epoch 74/200
50/50 [==============================] - ETA: 0s - loss: 9.4263e-0 - 0s 8ms/sample - los
s: 8.5122e-04
Epoch 75/200
50/50 [==============================] - 0s 8ms/sample - loss: 6.5188e-04
Epoch 76/200
50/50 [==============================] - 0s 8ms/sample - loss: 8.7707e-04
Epoch 77/200
50/50 [==============================] - 0s 8ms/sample - loss: 8.3961e-04
Epoch 78/200
50/50 [==============================] - 0s 8ms/sample - loss: 7.6588e-04
Epoch 79/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0013
Epoch 80/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0019
Epoch 81/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0015
Epoch 82/200
50/50 [==============================] - 0s 9ms/sample - loss: 0.0049
Epoch 83/200
50/50 [==============================] - 0s 9ms/sample - loss: 0.0037
Epoch 84/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0034
Epoch 85/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0027
Epoch 86/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0033
Epoch 87/200
50/50 [==============================] - 0s 9ms/sample - loss: 0.0025
Epoch 88/200
50/50 [==============================] - 0s 9ms/sample - loss: 0.0015
Epoch 89/200
50/50 [==============================] - 0s 8ms/sample - loss: 0.0013
Epoch 90/200
50/50 [==============================] - 0s 8ms/sample - loss: 8.6775e-04
Epoch 91/200
50/50 [==============================] - ETA: 0s - loss: 0.001 - 0s 8ms/sample - loss:
0.0010
Epoch 92/200
```

```
50/50 [==============================] - 0s 8ms/sample - loss: 5.3819e-04
Epoch 93/200
50/50 [==============================] - 0s 8ms/sample - loss: 8.4201e-04
Epoch 94/200
50/50 [==============================] - 0s 8ms/sample - loss: 4.1348e-04
Epoch 95/200
50/50 [==============================] - 0s 9ms/sample - loss: 4.4456e-04
Epoch 96/200
50/50 [==============================] - 0s 8ms/sample - loss: 4.9698e-04
Epoch 97/200
50/50 [==============================] - 0s 9ms/sample - loss: 3.2668e-04
Epoch 98/200
50/50 [==============================] - 0s 8ms/sample - loss: 3.9916e-04
Epoch 99/200
50/50 [==============================] - 0s 8ms/sample - loss: 3.1461e-04
Epoch 100/200
50/50 [==============================] - 0s 8ms/sample - loss: 2.8347e-04
Epoch 101/200
50/50 [==============================] - 0s 8ms/sample - loss: 4.8089e-04
Epoch 102/200
50/50 [==============================] - 0s 8ms/sample - loss: 3.0220e-04
Epoch 103/200
50/50 [==============================] - 0s 8ms/sample - loss: 3.8438e-04
Epoch 104/200
50/50 [==============================] - 0s 8ms/sample - loss: 7.1435e-04
Epoch 105/200
50/50 [==============================] - 0s 8ms/sample - loss: 2.6637e-04
Epoch 106/200
50/50 [==============================] - 0s 8ms/sample - loss: 2.9666e-04
Epoch 107/200
50/50 [==============================] - 0s 9ms/sample - loss: 4.2167e-04
Epoch 108/200
50/50 [==============================] - 0s 8ms/sample - loss: 5.4720e-04
Epoch 109/200
50/50 [==============================] - 0s 9ms/sample - loss: 5.7166e-04
Epoch 110/200
50/50 [==============================] - 0s 9ms/sample - loss: 4.6331e-04
Epoch 111/200
50/50 [==============================] - 0s 9ms/sample - loss: 0.0011
Epoch 112/200
50/50 [==============================] - 0s 8ms/sample - loss: 8.7956e-04
Epoch 113/200
50/50 [==============================] - 0s 8ms/sample - loss: 5.4062e-04
Epoch 114/200
50/50 [==============================] - 0s 8ms/sample - loss: 3.7140e-04
Epoch 115/200
50/50 [==============================] - 0s 8ms/sample - loss: 3.7188e-04
Epoch 116/200
50/50 [==============================] - 0s 8ms/sample - loss: 3.9435e-04
Epoch 117/200
50/50 [==============================] - 0s 8ms/sample - loss: 2.8453e-04
Epoch 118/200
50/50 [==============================] - 0s 8ms/sample - loss: 2.0109e-04
Epoch 119/200
50/50 [==============================] - 0s 8ms/sample - loss: 2.4461e-04
Epoch 120/200
50/50 [==============================] - 0s 8ms/sample - loss: 1.8169e-04
Epoch 121/200
50/50 [==============================] - 0s 8ms/sample - loss: 2.9210e-04
Epoch 122/200
50/50 [==============================] - 0s 8ms/sample - loss: 1.7043e-04
Epoch 123/200
50/50 [==============================] - 0s 8ms/sample - loss: 1.4859e-04
Epoch 124/200
50/50 [==============================] - 0s 8ms/sample - loss: 1.7207e-04
```

```
Epoch 125/200
50/50 [==============================] - 0s 8ms/sample - loss: 1.6223e-04
Epoch 126/200
50/50 [==============================] - 0s 8ms/sample - loss: 1.8606e-04
Epoch 127/200
50/50 [==============================] - 0s 8ms/sample - loss: 1.1607e-04
Epoch 128/200
50/50 [==============================] - 0s 8ms/sample - loss: 1.1448e-04
Epoch 129/200
50/50 [==============================] - 0s 8ms/sample - loss: 9.7022e-05
Epoch 130/200
50/50 [==============================] - 0s 8ms/sample - loss: 1.0157e-04
Epoch 131/200
50/50 [==============================] - 0s 8ms/sample - loss: 8.1065e-05
Epoch 132/200
50/50 [==============================] - 0s 8ms/sample - loss: 6.0812e-05
Epoch 133/200
50/50 [==============================] - 0s 8ms/sample - loss: 4.6335e-05
Epoch 134/200
50/50 [==============================] - 0s 8ms/sample - loss: 5.4217e-05
Epoch 135/200
50/50 [==============================] - 0s 8ms/sample - loss: 3.0557e-05
Epoch 136/200
50/50 [==============================] - 0s 8ms/sample - loss: 2.9973e-05
Epoch 137/200
50/50 [==============================] - 0s 9ms/sample - loss: 2.0584e-05
Epoch 138/200
50/50 [==============================] - 0s 9ms/sample - loss: 1.9720e-05
Epoch 139/200
50/50 [==============================] - 0s 8ms/sample - loss: 1.7230e-05
Epoch 140/200
50/50 [==============================] - 0s 8ms/sample - loss: 1.5907e-05
Epoch 141/200
50/50 [==============================] - 0s 9ms/sample - loss: 1.0041e-05
Epoch 142/200
50/50 [==============================] - 0s 9ms/sample - loss: 8.6813e-06
Epoch 143/200
50/50 [==============================] - 0s 8ms/sample - loss: 9.0939e-06
Epoch 144/200
50/50 [==============================] - 0s 8ms/sample - loss: 9.0373e-06
Epoch 145/200
50/50 [==============================] - 0s 8ms/sample - loss: 7.4690e-06
Epoch 146/200
50/50 [==============================] - 0s 8ms/sample - loss: 7.0640e-06
Epoch 147/200
50/50 [==============================] - 0s 8ms/sample - loss: 6.7721e-06
Epoch 148/200
50/50 [==============================] - 0s 8ms/sample - loss: 1.5557e-05
Epoch 149/200
50/50 [==============================] - 0s 8ms/sample - loss: 2.5333e-05
Epoch 150/200
50/50 [==============================] - 0s 8ms/sample - loss: 1.2711e-05
Epoch 151/200
50/50 [==============================] - 0s 9ms/sample - loss: 1.1842e-05
Epoch 152/200
50/50 [==============================] - 0s 8ms/sample - loss: 1.2089e-05
Epoch 153/200
50/50 [==============================] - 0s 8ms/sample - loss: 9.4116e-06
Epoch 154/200
50/50 [==============================] - 0s 8ms/sample - loss: 9.8509e-06
Epoch 155/200
50/50 [==============================] - 0s 8ms/sample - loss: 8.6884e-06
Epoch 156/200
50/50 [==============================] - 0s 8ms/sample - loss: 6.8937e-06
Epoch 157/200
```

```
50/50 [==============================] - 0s 8ms/sample - loss: 4.7930e-06
Epoch 158/200
50/50 [==============================] - 0s 8ms/sample - loss: 6.5700e-06
Epoch 159/200
50/50 [==============================] - 0s 8ms/sample - loss: 5.9231e-06
Epoch 160/200
50/50 [==============================] - 0s 8ms/sample - loss: 5.0707e-06
Epoch 161/200
50/50 [==============================] - 0s 8ms/sample - loss: 4.9965e-06
Epoch 162/200
50/50 [==============================] - 0s 8ms/sample - loss: 5.1275e-06
Epoch 163/200
50/50 [==============================] - 0s 8ms/sample - loss: 5.0150e-06
Epoch 164/200
50/50 [==============================] - 0s 8ms/sample - loss: 4.0587e-06
Epoch 165/200
50/50 [==============================] - 0s 10ms/sample - loss: 4.9166e-06
Epoch 166/200
50/50 [==============================] - 0s 8ms/sample - loss: 5.6138e-06
Epoch 167/200
50/50 [==============================] - 0s 8ms/sample - loss: 4.4482e-06
Epoch 168/200
50/50 [==============================] - 0s 8ms/sample - loss: 3.7801e-06
Epoch 169/200
50/50 [==============================] - 0s 8ms/sample - loss: 3.0704e-06
Epoch 170/200
50/50 [==============================] - 0s 8ms/sample - loss: 3.3689e-06
Epoch 171/200
50/50 [==============================] - 0s 8ms/sample - loss: 3.1442e-06
Epoch 172/200
50/50 [==============================] - 0s 8ms/sample - loss: 2.8123e-06
Epoch 173/200
50/50 [==============================] - 0s 8ms/sample - loss: 2.3541e-06
Epoch 174/200
50/50 [==============================] - 0s 8ms/sample - loss: 2.3896e-06
Epoch 175/200
50/50 [==============================] - 0s 8ms/sample - loss: 2.0795e-06
Epoch 176/200
50/50 [==============================] - 0s 8ms/sample - loss: 1.8403e-06
Epoch 177/200
50/50 [==============================] - 0s 9ms/sample - loss: 1.6135e-06
Epoch 178/200
50/50 [==============================] - 0s 8ms/sample - loss: 1.8864e-06
Epoch 179/200
50/50 [==============================] - 0s 9ms/sample - loss: 2.2061e-06
Epoch 180/200
50/50 [==============================] - 0s 8ms/sample - loss: 1.6671e-06
Epoch 181/200
50/50 [==============================] - 0s 8ms/sample - loss: 1.9014e-06
Epoch 182/200
50/50 [==============================] - 0s 8ms/sample - loss: 1.5751e-06
Epoch 183/200
50/50 [==============================] - 0s 8ms/sample - loss: 1.4745e-06
Epoch 184/200
50/50 [==============================] - 0s 8ms/sample - loss: 1.3730e-06
Epoch 185/200
50/50 [==============================] - 0s 8ms/sample - loss: 1.1353e-06
Epoch 186/200
50/50 [==============================] - 0s 8ms/sample - loss: 1.1657e-06
Epoch 187/200
50/50 [==============================] - 0s 8ms/sample - loss: 1.0173e-06
Epoch 188/200
50/50 [==============================] - 0s 8ms/sample - loss: 1.1215e-06
Epoch 189/200
50/50 [==============================] - 0s 8ms/sample - loss: 9.8063e-07
```

```
Epoch 190/200
50/50 [==============================] - 0s 8ms/sample - loss: 9.1018e-07
Epoch 191/200
50/50 [==============================] - 0s 8ms/sample - loss: 9.0401e-07
Epoch 192/200
50/50 [==============================] - 0s 8ms/sample - loss: 1.0289e-06
Epoch 193/200
50/50 [==============================] - 0s 8ms/sample - loss: 9.8815e-07
Epoch 194/200
50/50 [==============================] - 0s 8ms/sample - loss: 9.4954e-07
Epoch 195/200
50/50 [==============================] - 0s 8ms/sample - loss: 8.3764e-07
Epoch 196/200
50/50 [==============================] - 0s 8ms/sample - loss: 8.1650e-07
Epoch 197/200
50/50 [==============================] - 0s 8ms/sample - loss: 7.3275e-07
Epoch 198/200
50/50 [==============================] - 0s 8ms/sample - loss: 6.8749e-07
Epoch 199/200
50/50 [==============================] - 0s 8ms/sample - loss: 6.7253e-07
Epoch 200/200
50/50 [==============================] - 0s 8ms/sample - loss: 6.9318e-07
Neural network trained
```

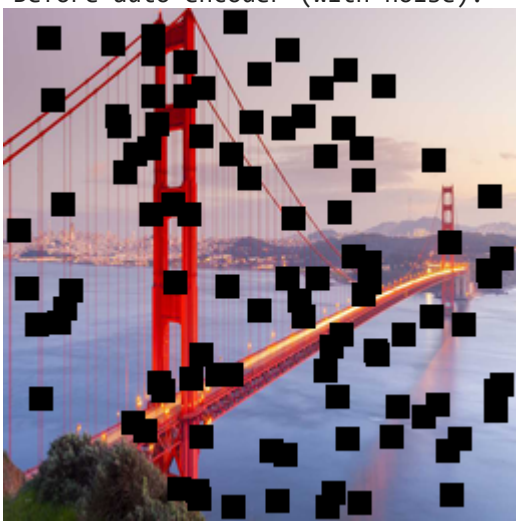
In [12]:

```
for z in range(10):
    print("*** Trial {}".format(z+1))

    # Choose random image
    i = np.random.randint(len(loaded_images))
    img = loaded_images[i]
    img_array = np.asarray(img)
    cols, rows = img.size

    # Add noise
    img_array_noise = add_noise(img_array)

    #Display noisy image
    img2 = img_array_noise.astype(np.uint8)
    img2 = Image.fromarray(img2, 'RGB')
    print("Before auto encoder (with noise):")
    display(img2)

    # Send noisy image to auto encoder
    img_array_noise = img_array_noise.flatten()
    img_array_noise = img_array_noise.astype(np.float32)
    img_array_noise = (img_array_noise-256)/256
    img_array_noise = np.array([img_array_noise])
    pred = model.predict(img_array_noise)[0]

    # Display neural result
    img_array2 = pred.reshape(rows,cols,3)
    img_array2 = (img_array2*256)+256
    img_array2 = img_array2.astype(np.uint8)
    img2 = Image.fromarray(img_array2, 'RGB')
    print("After auto encoder noise removal")
    display(img2)
```

```
*** Trial 1
Before auto encoder (with noise):
```

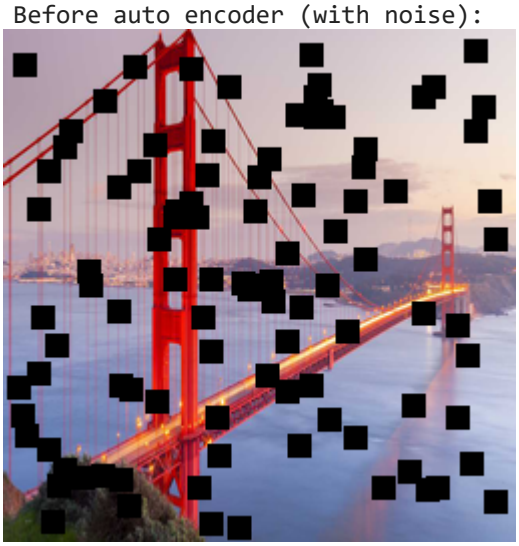
After auto encoder noise removal

*** Trial 2
Before auto encoder (with noise):

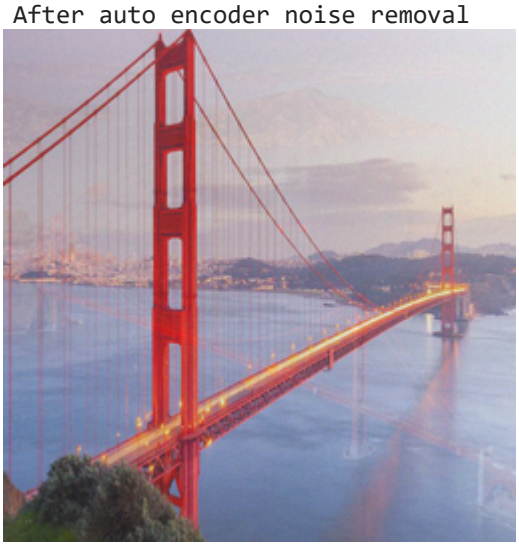


After auto encoder noise removal

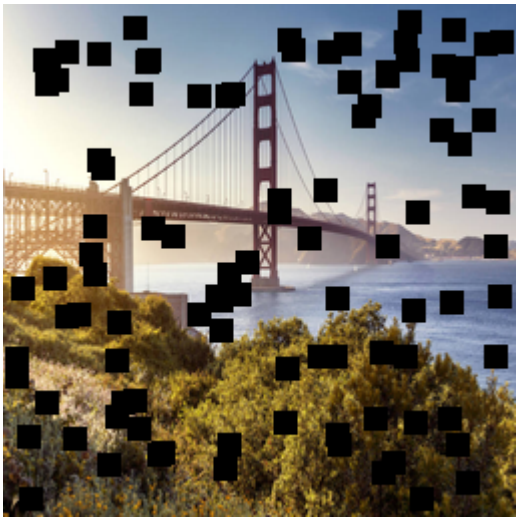
*** Trial 3
Before auto encoder (with noise):



After auto encoder noise removal



*** Trial 4
Before auto encoder (with noise):

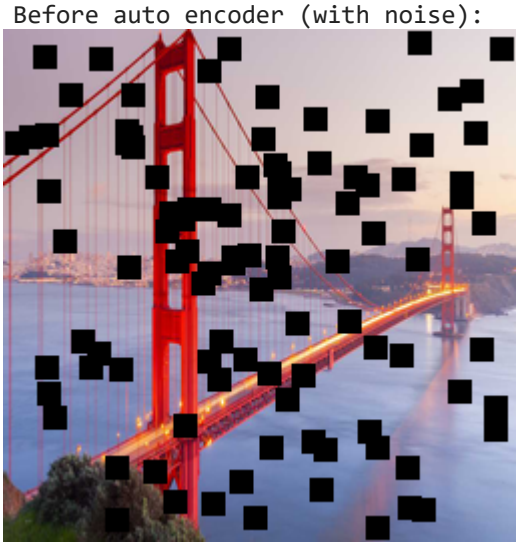


After auto encoder noise removal

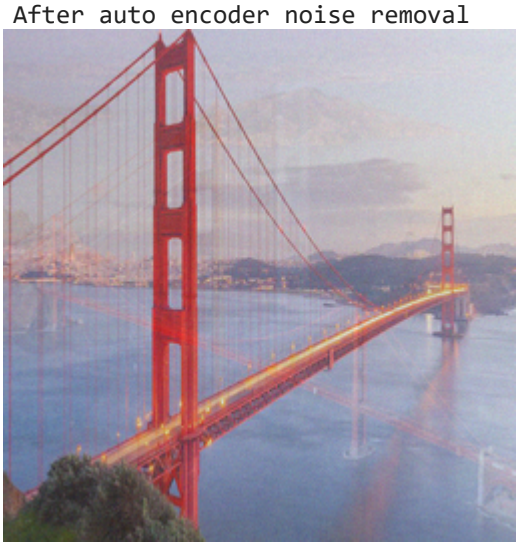
*** Trial 5
Before auto encoder (with noise):

After auto encoder noise removal

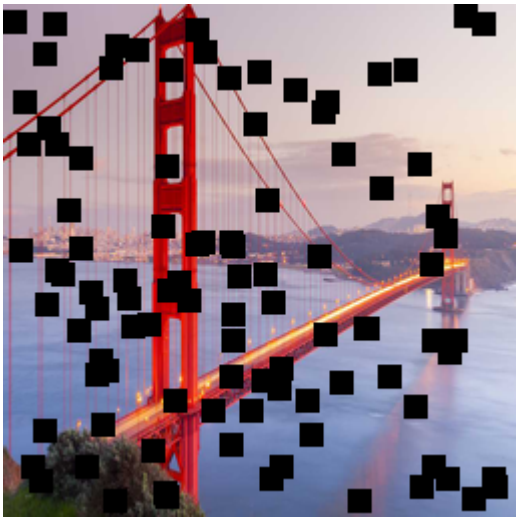
*** Trial 6
Before auto encoder (with noise):



After auto encoder noise removal



*** Trial 7
Before auto encoder (with noise):

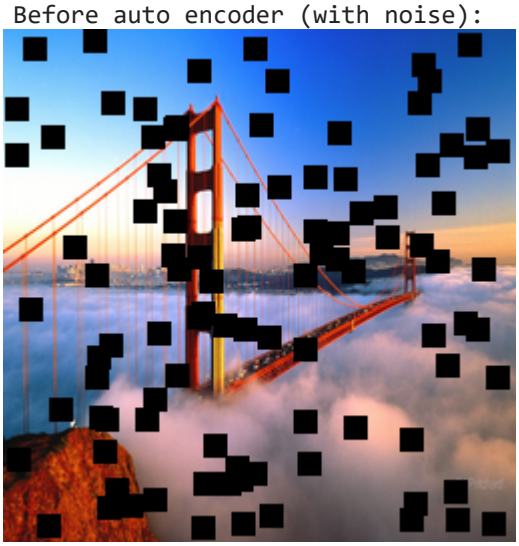


After auto encoder noise removal

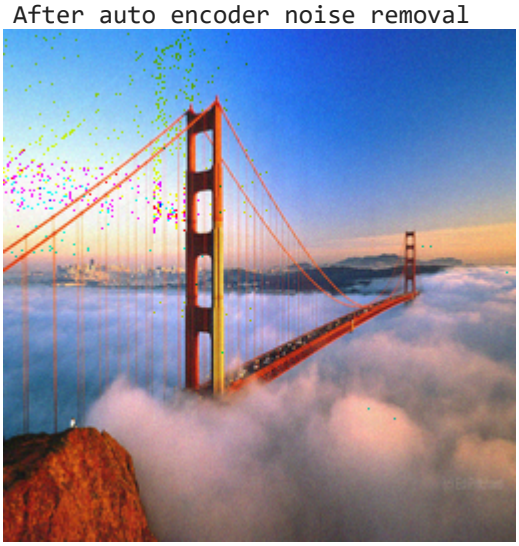
*** Trial 8
Before auto encoder (with noise):

After auto encoder noise removal

*** Trial 9
Before auto encoder (with noise):



After auto encoder noise removal



*** Trial 10
Before auto encoder (with noise):

After auto encoder noise removal

In [ ]:

CSC 180 Intelligent Systems (Fall 2022)

Dr. Haiquan Chen, Dept of Computer Scicence ¶

California State University, Sacramento

Lab 12： Distributed Evolutionary Algorithms in Python (DEAP) Basics

Elements to take into account using evolutionary algorithms

- **Individual representation** (binary, Gray, floating-point, etc.);
- **evaluation** and **fitness assignment**;
- **mating selection**, that establishes a partial order of individuals in the population using their fitness function value as reference and determines the degree at which individuals in the population will take part in the generation of new (offspring) individuals.
- **variation**, that applies a range of evolution-inspired operators, like crossover, mutation, etc., to synthesize offspring individuals from the current (parent) population. This process is supposed to prime the fittest individuals so they play a bigger role in the generation of the offspring.
- **environmental selection**, that merges the parent and offspring individuals to produce the population that will be used in the next iteration. This process often involves the deletion of some individuals using a given criterion in order to keep the amount of individuals bellow a certain threshold.
- **stopping criterion**, that determines when the algorithm shoulod be stopped, either because the optimum was reach or because the optimization process is not progressing.

Hence a 'general' evolutionary algorithm can be described as

```python
def evolutionary_algorithm():
    'Pseudocode of an evolutionary algorithm'
    populations = [] # a list with all the populations

    populations[0] =  initialize_population(pop_size)
    t = 0

    while not stop_criterion(populations[t]):
        fitnesses = evaluate(populations[t])
        offspring = matting_and_variation(populations[t],
                                          fitnesses)
        populations[t+1] = environmental_selection(
                                          populations[t],
                                          offspring)
        t = t+1
```

Python libraries for evolutionary computation

- PaGMO/PyGMO
- Inspyred
- **Distributed Evolutionary Algorithms in Python (DEAP)**

DISTRIBUTED EVOLUTIONARY ALGORITHMS IN PYTHON

- Open source Python library with,
- genetic algorithm using any representation;
- evolutionary strategies (including CMA-ES);
- multi-objective optimization from the start;
- co-evolution (cooperative and competitive) of multiple populations;
- parallelization of the evaluations (and more) using SCOOP;
- statistics keeping, and;
- benchmarks module containing some common test functions.
- [https://github.com/DEAP/deap] (https://github.com/DEAP/deap)

To install: pip install deap

Let's take the One Max problem as an example

Maximize the number of ones in a binary string (list, vector, etc.).

- More formally, from the set of binary strings of length n,
$$\mathcal{S} = \{s_1, \ldots, s_n\}, \text{ with } s_i = \{0, 1\}.$$
- Find $s^* \in \mathcal{S}$ such that

$$s^* = \arg\max_{s \in S} \sum_{i=1}^{n} s_i.$$

Its clear that the optimum is an *all-ones* string.

Coding the problem

Imports

We first get the required modules for our evolution.

- `random` gives us a way to generate random bits;
- `base` gives us access to the Toolbox and base Fitness;
- `creator` allows us to create our types;
- `tools` grants us access to the operators bank;
- `algorithms` enables us some ready generic evolutionary loops.

In [2]:
```
import random
import numpy as np
import pandas as pd
from deap import base, creator, tools, algorithms
```

Step 1: Create the Fitness class and the Individual class using create()

First step with DEAP is to create the required types. Usually the types created are the fitness and the individual.

Fitness Class

For the OneMax problem, we want to have a solution with as many ones as possible. Thus we need a maximizing fitness.

The create() function takes at least two arguments, a name for the newly created class and a base class. Any subsequent argument becomes an attribute of the class.

In [7]:
```
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
```

The weights attribute must be a ***tuple*** so that multi-objective and single objective fitnesses can be treated the same way. A FitnessMulti would be created the same way but using:

```
creator.create("FitnessMulti", base.Fitness, weights=(-1.0, 1.0))
```

This code produces a fitness that minimizes the first objective and maximize the second one. The weights can also be used to vary the importance of each objective one against another. This means that the weights can be any real number and only the sign is used to determine if a maximization or minimization is done.

Individual Class

For the OneMax problem, we want to have a solution with as many ones as possible. Thus we need a maximizing fitness and a individual that is a `list`.

Next we create an Individual class, using the creator, that will inherit from the standard list type and have a fitness attribute.

In [8]:
```
creator.create("Individual", list, fitness=creator.FitnessMax)
```

The created classes are made available in the creator module. We can instantiate directly objects of the created class like follow. This step is not required in an algorithm, the instatiation will be automated later.

You may use the following code to test if you create the individual class successfully

In [12]:
```
ind = creator.Individual([1, 0, 1, 1, 0])

print(ind)
print(type(ind))
print(type(ind.fitness))
```

```
[1, 0, 1, 1, 0]
<class 'deap.creator.Individual'>
<class 'deap.creator.FitnessMax'>
```

Step 2: Based on the individual representation, define your functions so DEAP can know how to instantiate each individual and the first population

- ***Functions are defined/registered using register() method, which takes at least two arguments; an alias and a function assigned to this alias. Any subsequent argument is passed to the function when called.***

If each individual is a list of boolean values:

In [13]:
```
toolbox = base.Toolbox()

toolbox.register("attr_bool", random.randint, 0, 1)
# random.randint(0, 1)  yields 0 or 1

toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox.attr
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
```

This first one, `attr_bool`, calls randint from the random module with arguments (0, 1) to create an integer in the interval $[0, 1]$.

The second function, `individual` will use the `initRepeat` function made available in the tools module to fill an `Individual` class with what is produced by 10 calls to the previously defined `attr_bool` function.

The same thing is done for the population function.

You may use the following code to test:

For example, calling every function individually shows how it proceeds.

In [14]:
```
bit = toolbox.attr_bool()
print((type(bit), bit))
```

```
(<class 'int'>, 1)
```

In [15]:
```
ind = toolbox.individual()
print((type(ind), len(ind), ind))
```

```
(<class 'deap.creator.Individual'>, 10, [0, 0, 0, 0, 0, 0, 0, 0, 0, 1])
```

In [16]:
```
pop = toolbox.population(n=3)
print((type(pop), len(pop), pop))
```

```
(<class 'list'>, 3, [[1, 0, 1, 0, 0, 0, 1, 1, 1, 1], [0, 0, 1, 0, 0, 0, 1, 0,
0, 1], [1, 1, 0, 1, 1, 1, 1, 1, 1, 1]])
```

If each individual is a list of float values:

In [17]:
```
toolbox1 = base.Toolbox()
toolbox1.register("attr_float", random.random)
toolbox1.register("individual", tools.initRepeat, creator.Individual,
                  toolbox1.attr_float, n=10)
toolbox1.register("population", tools.initRepeat, list, toolbox1.individual)
```

You may use the following code to test:

In [18]:
```
ind1 = toolbox1.individual()
print((type(ind1), len(ind1), ind1))
```

```
(<class 'deap.creator.Individual'>, 10, [0.4883608822945319, 0.821948542363906
8, 0.7755907619590585, 0.6009482199267582, 0.12340454173572957, 0.6824654792017
064, 0.4622005086554041, 0.04772036455219064, 0.05389758864248173, 0.5595009058
544583])
```

In [19]:
```
pop1 = toolbox1.population(n=3)
print((type(pop1), len(pop1), pop1))
```

```
(<class 'list'>, 3, [[0.4912154424591759, 0.3868728230182915, 0.811764912737215
5, 0.8803683796599934, 0.4124857983415052, 0.5320015185842386, 0.75323766428666
96, 0.9681128816876409, 0.2214110550270738, 0.7464874403540728], [0.21280456487
98319, 0.9855422254496641, 0.5169295149357216, 0.5101446923352828, 0.8829693593
426222, 0.3597981155927509, 0.5465330990115123, 0.5303235115832728, 0.260016132
3112383, 0.5830999479847632], [0.8994030032617878, 0.4968378683482826, 0.783794
494406394, 0.13961526220566478, 0.5267334890783276, 0.1568200068320046, 0.82642
42308511949, 0.46242253674122447, 0.12127732007002401, 0.2799075800889038]])
```

If each individual is a list of integer values:

In [20]:
```
toolbox2 = base.Toolbox()
toolbox2.register("attr_int", random.randint, 0, 10)
toolbox2.register("individual", tools.initRepeat, creator.Individual,
                  toolbox2.attr_int, n=10)
toolbox2.register("population", tools.initRepeat, list, toolbox2.individual)
```

You may use the following code to test:

In [21]:
```
ind2 = toolbox2.individual()
print(ind2)
```

```
[1, 1, 9, 6, 9, 5, 9, 9, 3, 6]
```

In [22]:
```
pop2 = toolbox2.population(n=3)
print((type(pop2), len(pop2), pop2))
```

```
(<class 'list'>, 3, [[1, 9, 3, 10, 6, 1, 1, 5, 0, 2], [5, 7, 3, 7, 4, 10, 7, 4,
6, 0], [6, 5, 9, 8, 3, 1, 4, 4, 2, 1]])
```

If each individual is a permutation:

An individual for the permutation representation is almost similar to the general list individual. In fact they both inherit from the basic list type. The only difference is that instead of filling the list with a series of floats, we need to generate a random permutation and provide that permutation to the individual.

In [23]:
```
toolbox3 = base.Toolbox()

toolbox3.register("indices", np.random.permutation, 10)
# np.random.permutation (10) return array([0, 6, 4, 8, 5, 2, 9, 1, 3, 7])

toolbox3.register("individual", tools.initIterate, creator.Individual,
                  toolbox3.indices)
toolbox3.register("population", tools.initRepeat, list, toolbox3.individual)
```

You may use the following code to test:

In [24]:
```
ind1 = toolbox3.individual()
print(ind1)
```

```
[1, 0, 2, 4, 3, 6, 7, 8, 9, 5]
```

In [25]:
```
pop3 = toolbox3.population(n=3)
print((type(pop3), len(pop3), pop3))
```

```
(<class 'list'>, 3, [[9, 6, 3, 5, 7, 2, 4, 8, 1, 0], [2, 1, 3, 8, 4, 0, 7, 9,
5, 6], [3, 9, 6, 5, 8, 2, 1, 0, 7, 4]])
```

Step 3: Define fitness evaluation function fo each individual

The evaluation function is pretty simple for the OneMax problem, we need to count the number of ones in an individual.

Notice that *the returned value must be an tuple, equal to the number of objectives.*

Notice the comma following sum()!!

In [26]:
```
def evalOneMax(individual):
    return (sum(individual),)      # return type should be tuple


# E.g., sum([1, 0, 1, 0, 1, 1, 0, 1, 1, 0]) returns 6
```

Step 4: Define genetic operators

- The usual names for the operators are mate(), mutate(), evaluate() and select(). Registering the operators and their default arguments in the toolbox is done as follow.

`All possible operaters` : https://deap.readthedocs.io/en/master/api/tools.html (https://deap.readthedocs.io/en/master/api/tools.html)

In [27]:
```
toolbox.register("evaluate", evalOneMax)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutFlipBit, indpb=0.10)
toolbox.register("select", tools.selTournament, tournsize=3)
```

The evaluation is given the alias `evaluate`. `evalOneMax` will be called with the signature `toolbox.evaluate(individual)` that takes the individual as input, which will be given later.

The two points crossover function is registered the same way under the alias `mate`.

The mutation set the `independent probability of each attribute to be mutated` (`indpb`) to 0.1. In the algorithms, the mutate() function is called with the signature `toolbox.mutate(individual)`

Finally, the selection operator is `K-way turnament selection`, registered under the name `select` and the size of the tournament set to 3.

We can for example mutate an individual and expect 10% of its attributes to be flipped.

You may use the following code to test:

In [28]:
```
ind = toolbox.individual()
print(ind)

toolbox.mutate(ind)
print(ind)
```

```
[1, 0, 0, 1, 1, 0, 0, 0, 1, 1]
[0, 1, 0, 1, 1, 0, 0, 0, 1, 1]
```

Step 5: Choose your algorithm and let evolution go!

https://deap.readthedocs.io/en/master/api/algo.html (https://deap.readthedocs.io/en/master/api/algo.html)

- There are several algorithms implemented in the algorithms module.
- In order to setup a toolbox for an algorithm, you must register the desired operators under a specified names.

The *simple evolutionary algorithm* takes 5 arguments, a population, a toolbox, a probability of cross-overing two individuals (`cxpb`), a probability of mutating each individual (`mutpb`) and a number of generations to accomplish (`ngen`).

Once the evolution is finished the population contains the individuals from the last generation.

In [29]:
```
import matplotlib.pyplot as plt
%matplotlib inline

pop = toolbox.population(n=100)

pop, log= algorithms.eaSimple(pop, toolbox, cxpb=0.5, mutpb=0.2, ngen=10)

# nevals:  the number of evalutions for each generation
```

```
gen     nevals
0       100
1       61
2       65
3       64
4       58
5       67
6       58
7       52
8       60
9       72
10      62
```

In [31]:
```python
# Print out the last generation

pop
```

Out[31]:
```
[[1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 0, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 0, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 0, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 0, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 0, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
```

```
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 0, 1, 1, 0, 1, 0],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 0, 1, 1, 1, 1, 1, 1, 0],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 0, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 0, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 0, 1, 0, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 0, 1, 1, 1, 1, 1, 1, 1, 0],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 0, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]
```

In [36]:
```
best_ind = tools.selBest(pop, k=1)[0]

print('Best individual is:', best_ind)
```

```
Best individual is: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

In [37]:
```
print('With fitness: ', toolbox.evaluate(best_ind))
```

```
With fitness:  (10,)
```

Note: Computing Statistics and Hall of Fame

Here we will employ some helpful tools such as Statistics and a Hall of Fame.

- The `statistics` are computed using numpy functions on the population, and the `hall of fame` keeps track of the best individuals that ever appeared during the evolution.

To do Statistics, one need to register the desired statistic functions inside the statistic object.

In [39]:
```
stats = tools.Statistics(key=lambda ind: ind.fitness.values)
```

The statistics object is created using a key as first argument. This key must be supplied a function that will later be applied to the data on which the statistics are computed. The previous code sample uses the fitness.values attribute of each element.

In [40]:
```
import numpy as np

stats.register("avg", np.mean)
stats.register("min", np.min)
stats.register("max", np.max)
```

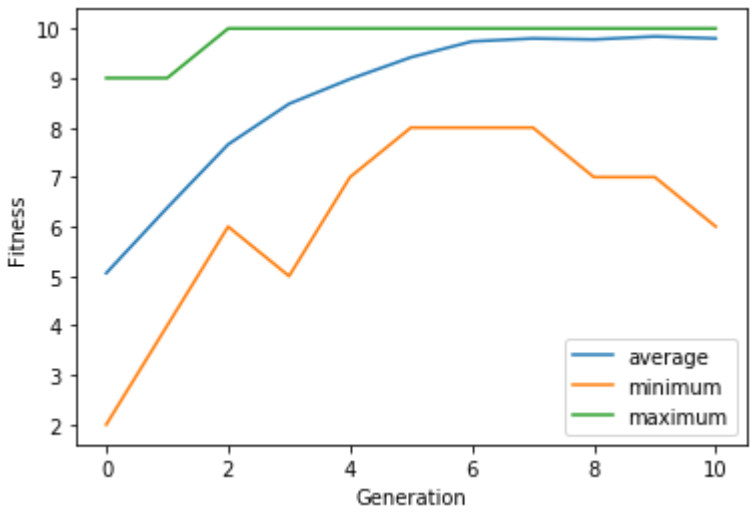
- The statistical functions are now registered.
- The register function expects an alias as first argument and a function operating on vectors as second argument.
- Any subsequent argument is passed to the function when called. The creation of the statistics object is now complete.

In [41]:
```
pop = toolbox.population(n=50)

hof = tools.HallOfFame(maxsize=1)

pop, log = algorithms.eaSimple(pop, toolbox, cxpb=0.5, mutpb=0.2, ngen=10,
                               stats=stats, halloffame=hof, verbose=True)
```

```
gen     nevals  avg     min     max
0       50      5.06    2       9
1       29      6.38    4       9
2       26      7.66    6       10
3       24      8.48    5       10
4       26      8.98    7       10
5       29      9.42    8       10
6       31      9.74    8       10
7       34      9.8     8       10
8       33      9.78    7       10
9       20      9.84    7       10
10      22      9.8     6       10
```

- Statistics will automatically be computed on the population every generation.
- The verbose argument prints the statistics on screen while the optimization takes place.
- Once the algorithm returns, the final population and a Logbook are returned.

In [42]:
```
gen, avg, min_, max_ = log.select("gen", "avg", "min", "max")

plt.plot(gen, avg, label="average")
plt.plot(gen, min_, label="minimum")
plt.plot(gen, max_, label="maximum")
plt.xlabel("Generation")
plt.ylabel("Fitness")
plt.legend(loc="lower right")
plt.show()
```



In [43]:
```
record = stats.compile(pop)
record
```

Out[43]:
```
{'avg': 9.8, 'min': 6.0, 'max': 10.0}
```

Using Hall of Fame

Hall of Fame preserves the best individuals that appeared during an evolution. At every generation, it scans the population and saves the individuals in a separate archive that does not interact with the population. If the best solution disappears during the evolution, it will still be available in the hall of fame.

The hall of fame proposes a list interface where the individuals are sorted in descending order of fitness. Thus, the fittest solution can be retrieved by accessing the list's first element.

In [44]:
```
print("Best individual is: ", hof[0])

print("\nwith fitness: ", hof[0].fitness)
```

```
Best individual is:  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

with fitness:  (10.0,)
```

Super cool, right?

Inside the black box: Controlling everything instead of using the traditional black-box approach

DEAP encourages users to rapidly build their own algorithms. With the different tools provided by DEAP, it is possible to design a nimble algorithm that tackles most problems at hand.

Let's put everythong together and the "canned" eaSimple function can be replaced by lines 26-53.

In [33]:

```
import random
from deap import algorithms, base, creator, tools

creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", list, fitness=creator.FitnessMax)

def evalOneMax(individual):
    return (sum(individual),)      # return type should be tuple



toolbox = base.Toolbox()
toolbox.register("attr_bool", random.randint, 0, 1)
toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox.attr
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

toolbox.register("evaluate", evalOneMax)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutFlipBit, indpb=0.10)
toolbox.register("select", tools.selTournament, tournsize=3)


pop = toolbox.population(n=10)


# begininng of the eaSimple function

ngen, cxpb, mutpb = 40, 0.5, 0.2
fitnesses = toolbox.map(toolbox.evaluate, pop)

for ind, fit in zip(pop, fitnesses):
    ind.fitness.values = fit

for g in range(ngen):
    pop = toolbox.select(pop, k=len(pop))
    pop = [toolbox.clone(ind) for ind in pop]

    for child1, child2 in zip(pop[::2], pop[1::2]):
        if random.random() < cxpb:
            toolbox.mate(child1, child2)
            del child1.fitness.values, child2.fitness.values

    for mutant in pop:
        if random.random() < mutpb:
            toolbox.mutate(mutant)
            del mutant.fitness.values

    invalids = [ind for ind in pop if not ind.fitness.valid]
    fitnesses = toolbox.map(toolbox.evaluate, invalids)
    for ind, fit in zip(invalids, fitnesses):
        ind.fitness.values = fit

# end of the eaSimple function
```

```
best_ind = tools.selBest(pop, k=1)[0]

print('Best individual is:', best_ind)
print('With fitness: ', toolbox.evaluate(best_ind))


print()
print ("Last generation:")
for n in range(10):
    print (pop[n], pop[n].fitness.values)
```

```
Best individual is: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
With fitness:  (10,)

Last generation:
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1] (10.0,)
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1] (10.0,)
[0, 1, 1, 0, 1, 1, 1, 1, 1, 1] (8.0,)
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1] (10.0,)
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1] (10.0,)
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1] (10.0,)
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1] (10.0,)
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1] (10.0,)
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1] (10.0,)
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1] (10.0,)
```

First, the evaluation is applied to every individual in the population by the map function contained in every toolbox. Next, a loop over both the population and the evaluated fitnesses sets each individual's fitness value.

Thereafter, the generational loop begins. It starts by selecting k individuals from the population, followed by the duplication of the population by the clone tool. Then, the crossover is applied to a portion of consecutive individuals . Each modified individual sees its fitness invalidated by the deletion of its value. Finally, a percentage of the population is mutated and their fitness values is also deleted.

Only freshly produced individuals have to be evaluated; they are filtered by their fitness validity.

This version of the program provides control over the application order and the number of operators, among many other aspects.

CSC 180 Intelligent Systems (Fall 2022)

Dr. Haiquan Chen, Dept of Computer Scicence

California State University, Sacramento

Lab 13： Building Expert Systems using Fuzzy Inference and skfuzzy API

Install the skfuzzy API

pip install scikit-fuzzy

The Tipping Problem

The 'tipping problem' is commonly used to illustrate the power of fuzzy logic principles to generate complex behavior from a compact, intuitive set of expert rules.

Let's create a fuzzy control system which models how you might choose to tip at a restaurant. When tipping, you consider the service and food quality rated between 0 and 10. You use this to leave a tip of between 0 and 25%.

We would formulate this problem as:

- Antecednets (Inputs)
  - `service`
    - Universe (ie, crisp value range): How good was the service of the wait staff, on a scale of 0 to 10?
    - Fuzzy set (ie, fuzzy value range): poor, acceptable, amazing
  - `food quality`
    - Universe: How tasty was the food, on a scale of 0 to 10?
    - Fuzzy set: bad, decent, great
- Consequents (Outputs)
  - `tip`
    - Universe: How much should we tip, on a scale of 0% to 25%
    - Fuzzy set: low, medium, high
- Rules on Fuzzy numbers
  - IF the *service* was good *or* the *food quality* was good, THEN the tip will be high.
  - IF the *service* was average, THEN the tip will be medium.
  - IF the *service* was poor *and* the *food quality* was poor THEN the tip will be low.
- Usage
  - If I tell this controller that I rated:
    - the service as 9.8, and

    - the quality as 6.5,
- it would recommend I leave:
    - a 20.2% tip.

First, let's define fuzzy numbers

In [1]:
```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt

import skfuzzy as fuzz
from skfuzzy import control as ctrl

%matplotlib inline
```

In [2]:
```
# New Antecedent/Consequent objects hold universe variables and membership
# functions
quality = ctrl.Antecedent(np.arange(0, 11, 1), 'quality')
service = ctrl.Antecedent(np.arange(0, 11, 1), 'service')
tip = ctrl.Consequent(np.arange(0, 26, 1), 'tip')
```

trimf(): Triangular-shaped built-in membership function.

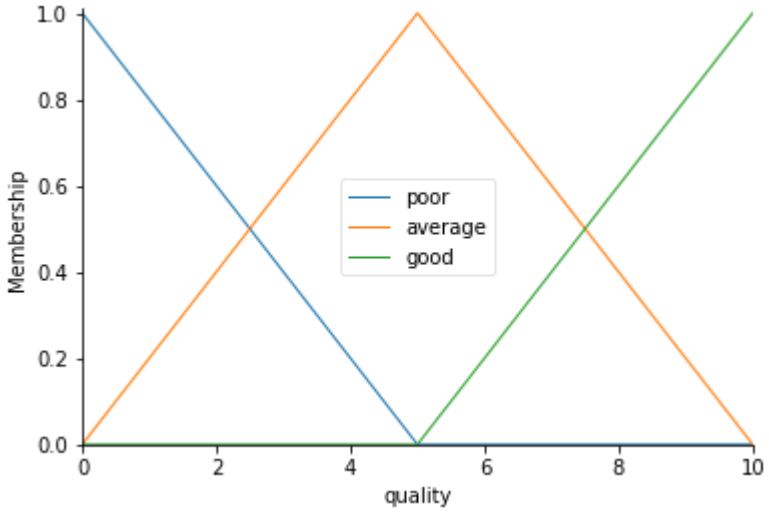
Syntax: y = trimf(x,[a b c])

The parameters a and c locate the "feet" of the triangle and the parameter c locates the peak.

In [3]:
```
quality['poor'] = fuzz.trimf(quality.universe, [0, 0, 5])
quality['average'] = fuzz.trimf(quality.universe, [0, 5, 10])
quality['good'] = fuzz.trimf(quality.universe, [5, 10, 10])
```

In [4]:
```
# You can see how these look with .view()
quality.view()
```

```
C:\Users\chenh\Anaconda3\lib\site-packages\skfuzzy\control\fuzzyvariable.py:12
2: UserWarning: Matplotlib is currently using module://ipykernel.pylab.backend_
inline, which is a non-GUI backend, so cannot show the figure.
  fig.show()
```

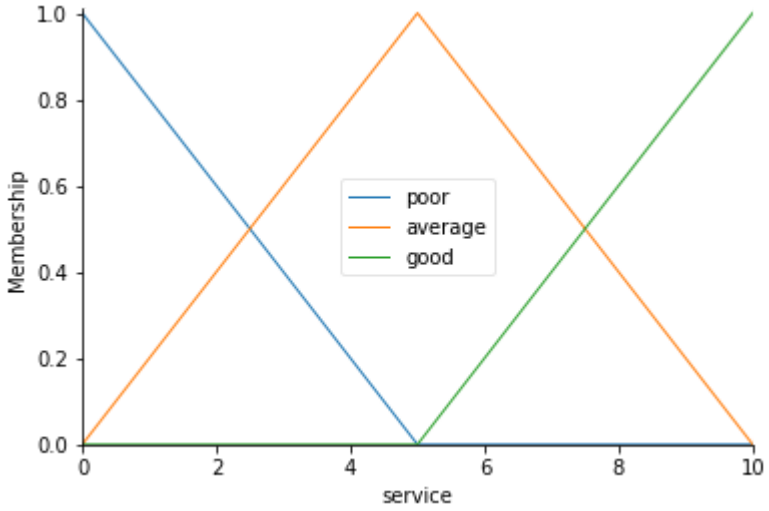


Now we have defined three fuzzy numbers for quality on a scale of 0 to 10

In [10]:
```
service['poor'] = fuzz.trimf(service.universe, [0, 0, 5])
service['average'] = fuzz.trimf(service.universe, [0, 5, 10])
service['good'] = fuzz.trimf(service.universe, [5, 10, 10])
```

In [11]:
```
# You can see how these look with .view()
service.view()
```

```
C:\Users\chenh\Anaconda3\lib\site-packages\skfuzzy\control\fuzzyvariable.py:12
2: UserWarning: Matplotlib is currently using module://ipykernel.pylab.backend_
inline, which is a non-GUI backend, so cannot show the figure.
  fig.show()
```

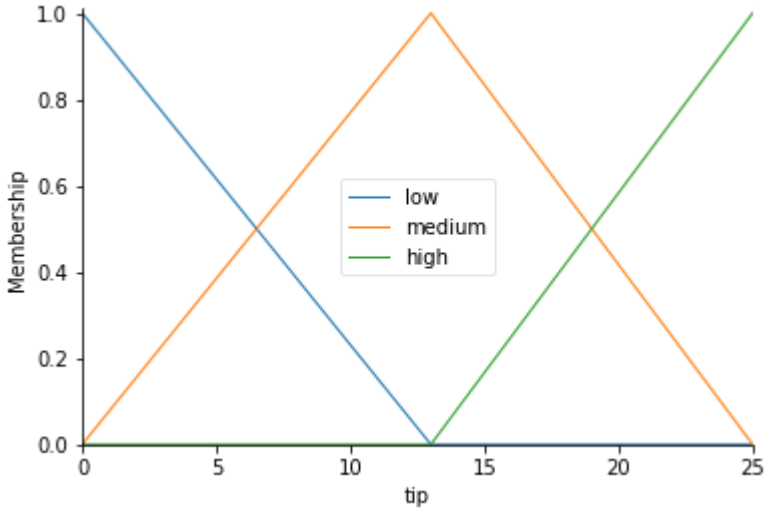


Now we have defined three fuzzy numbers for service on a scale of 0 to 10

In [12]:
```
### So far we have defined three fuzzy numbers for quality on a scale of 0 to 10#
# Pythonic API
tip['low'] = fuzz.trimf(tip.universe, [0, 0, 13])
tip['medium'] = fuzz.trimf(tip.universe, [0, 13, 25])
tip['high'] = fuzz.trimf(tip.universe, [13, 25, 25])
```

In [13]:
```
# You can see how these look with .view()
tip.view()
```

```
C:\Users\chenh\Anaconda3\lib\site-packages\skfuzzy\control\fuzzyvariable.py:12
2: UserWarning: Matplotlib is currently using module://ipykernel.pylab.backend_
inline, which is a non-GUI backend, so cannot show the figure.
  fig.show()
```



Similarly we have defined three fuzzy numbers for tip on a scale of 0 to 10

Second, define fuzzy rules

Now, to make these triangles useful, we define the fuzzy relationship between input and output variables. For the purposes of our example, consider 3 simple rules:

1. If the food is poor OR the service is poor, then the tip will be low

2. If the service is average, then the tip will be medium

3. If the food is good OR the service is good, then the tip will be high.

Most people would agree on these rules, but the rules are fuzzy. Mapping the imprecise rules into a defined, actionable tip is a challenge. This is the kind of task at which fuzzy logic excels.

In [14]:
```
rule1 = ctrl.Rule(quality['poor'] | service['poor'], tip['low'])
rule2 = ctrl.Rule(service['average'], tip['medium'])
rule3 = ctrl.Rule(service['good'] | quality['good'], tip['high'])
```

Third, model creation and simulation

Now that we have our rules defined, we can simply create a control system via:

In [15]:
```
tipping_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])
```

In order to simulate this control system, we will create a ControlSystemSimulation. Think of this object representing our controller applied to a specific set of cirucmstances. For tipping, this might be tipping Sharon at the local brew-pub. We would create another ControlSystemSimulation when we're trying to apply our tipping_ctrl for Travis at the cafe because the inputs would be different.

In [16]:
```
tipping = ctrl.ControlSystemSimulation(tipping_ctrl)
```

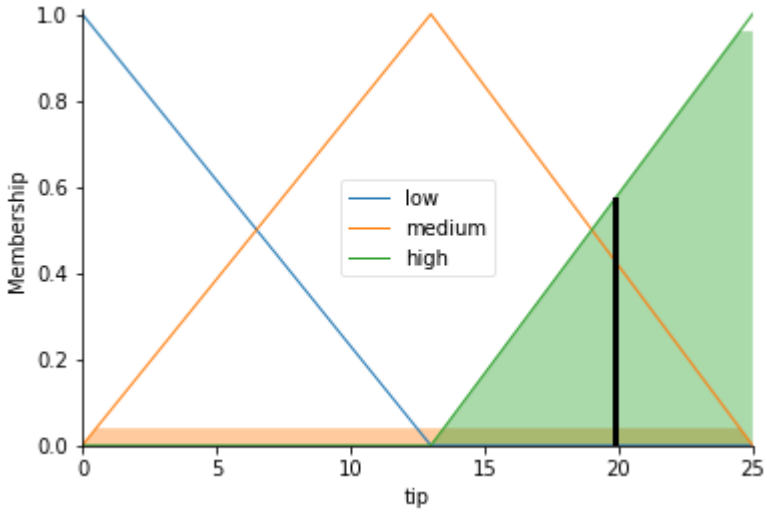
We can now simulate our control system by simply specifying the inputs and calling the compute method. Suppose we rated the quality 6.5 out of 10 and the service 9.8 of 10.

In [17]:
```
# Pass inputs to the ControlSystem using Antecedent labels with Pythonic API
# Note: if you like passing many inputs all at once, use .inputs(dict_of_data)
tipping.input['quality'] = 6.5
tipping.input['service'] = 9.8

# Crunch the numbers
tipping.compute()
```

Once computed, we can view the result as well as visualize it.

In [18]:
```
print(tipping.output['tip'])

tip.view(sim=tipping)
```

```
19.847607361963192

C:\Users\chenh\Anaconda3\lib\site-packages\skfuzzy\control\fuzzyvariable.py:12
2: UserWarning: Matplotlib is currently using module://ipykernel.pylab.backend_
inline, which is a non-GUI backend, so cannot show the figure.
  fig.show()
```



The resulting suggested tip is 19.85%.

References:

https://pythonhosted.org/scikit-fuzzy/ (https://pythonhosted.org/scikit-fuzzy/)