Watson Studio democratizes machine learning and deep learning to accelerate infusion of AI in your business to drive innovation. Watson Studio provides a suite of tools and a collaborative environment for data scientists, developers and domain experts.

(http://cocl.us/pytorch_link_top)

Torch Tensors in 1D

Table of Contents

In this lab, you will learn the basics of tensor operations. Tensors are an essential part of PyTorch; there are complex mathematical objects in and of themselves. Fortunately, most of the intricacies are not necessary. In this section, you will compare them to vectors and numpy arrays.

Estimated Time Needed: **25 min**

Preparation

Import the following libraries that you'll use for this lab:

In [1]:

```
# These are the libraries will be used for this lab.

import torch
import numpy as np
import pandas as pd

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

Check PyTorch version:

In [2]:

```
torch.__version__
```

Out[2]:

```
'1.3.1'
```

This is the function for plotting diagrams. You will use this function to plot the vectors in Coordinate system.

In [3]:

```
# Plot vecotrs, please keep the parameters in the same length
# @param: Vectors = [{"vector": vector variable, "name": name of vector, "color": c
olor of the vector on diagram}]

def plotVec(vectors):
    ax = plt.axes()

    # For loop to draw the vectors
    for vec in vectors:
        ax.arrow(0, 0, *vec["vector"], head_width = 0.05,color = vec["color"], head
_length = 0.1)
        plt.text(*(vec["vector"] + 0.1), vec["name"])

    plt.ylim(-2,2)
    plt.xlim(-2,2)
```

Types and Shape

You can find the type of the following list of integers *[0, 1, 2, 3, 4]* by applying the constructor `torch.tensor()`:

In [4]:

```
# Convert a integer list with length 5 to a tensor

ints_to_tensor = torch.tensor([0, 1, 2, 3, 4])
print("The dtype of tensor object after converting it to tensor: ", ints_to_tensor.
dtype)
print("The type of tensor object after converting it to tensor: ", ints_to_tensor.t
ype())
```

```
The dtype of tensor object after converting it to tensor:  torch.int64
The type of tensor object after converting it to tensor:  torch.LongTen
sor
```

As a result, the integer list has been converted to a long tensor.

The Python type is still `torch.Tensor` :

In [5]:

```
type(ints_to_tensor)
```

Out[5]:

```
torch.Tensor
```

You can find the type of this float list *[0.0, 1.0, 2.0, 3.0, 4.0]* by applying the method `torch.tensor()` :

In [6]:

```
# Convert a float list with length 5 to a tensor

floats_to_tensor = torch.tensor([0.0, 1.0, 2.0, 3.0, 4.0])
print("The dtype of tensor object after converting it to tensor: ", floats_to_tenso
r.dtype)
print("The type of tensor object after converting it to tensor: ", floats_to_tensor
.type())
```

```
The dtype of tensor object after converting it to tensor:  torch.float3
2
The type of tensor object after converting it to tensor:  torch.FloatTe
nsor
```

The float list is converted to a float tensor.

In [7]:

```
list_floats=[0.0, 1.0, 2.0, 3.0, 4.0]

floats_int_tensor=torch.tensor(list_floats,dtype=torch.int64)
```

In [8]:

```
print("The dtype of tensor object is: ", floats_int_tensor.dtype)
print("The type of tensor object is: ", floats_int_tensor.type())
```

```
The dtype of tensor object is:  torch.int64
The type of tensor object is:  torch.LongTensor
```

Note: The elements in the list that will be converted to tensor must have the same type.

From the previous examples, you see that `torch.tensor()` converts the list to the tensor type, which is similar to the original list type. However, what if you want to convert the list to a certain tensor type? `torch` contains the methods required to do this conversion. The following code converts an integer list to float tensor:

In [9]:

```
# Convert a integer list with length 5 to float tensor

new_float_tensor = torch.FloatTensor([0, 1, 2, 3, 4])
new_float_tensor.type()
print("The type of the new_float_tensor:", new_float_tensor.type())
```

```
The type of the new_float_tensor: torch.FloatTensor
```

In [10]:

```
new_float_tensor = torch.FloatTensor([0, 1, 2, 3, 4])
```

You can also convert an existing tensor object (*tensor_obj*) to another tensor type. Convert the integer tensor to a float tensor:

In [11]:

```
# Another method to convert the integer list to float tensor

old_int_tensor = torch.tensor([0, 1, 2, 3, 4])
new_float_tensor = old_int_tensor.type(torch.FloatTensor)
print("The type of the new_float_tensor:", new_float_tensor.type())
```

```
The type of the new_float_tensor: torch.FloatTensor
```

The *tensor_obj*`.size()` helps you to find out the size of the *tensor_obj* . The *tensor_obj*`.ndimension()` shows the dimension of the tensor object.

In [12]:

```
# Introduce the tensor_obj.size() & tensor_ndimension.size() methods

print("The size of the new_float_tensor: ", new_float_tensor.size())
print("The dimension of the new_float_tensor: ",new_float_tensor.ndimension())
```

```
The size of the new_float_tensor:  torch.Size([5])
The dimension of the new_float_tensor:  1
```

The *tensor_obj*`.view(`*row, column*`)` is used for reshaping a tensor object.

What if you have a tensor object with `torch.Size([5])` as a `new_float_tensor` as shown in the previous example?
After you execute `new_float_tensor.view(5, 1)`, the size of `new_float_tensor` will be `torch.Size([5, 1])`.
This means that the tensor object `new_float_tensor` has been reshaped from a one-dimensional tensor object with 5 elements to a two-dimensional tensor object with 5 rows and 1 column.

In [13]:

```
# Introduce the tensor_obj.view(row, column) method

twoD_float_tensor = new_float_tensor.view(5, 1)
print("Original Size: ", new_float_tensor)
print("Size after view method", twoD_float_tensor)
```

```
Original Size:  tensor([0., 1., 2., 3., 4.])
Size after view method tensor([[0.],
        [1.],
        [2.],
        [3.],
        [4.]])
```

Note that the original size is 5. The tensor after reshaping becomes a 5X1 tensor analog to a column vector.

Note: The number of elements in a tensor must remain constant after applying view.

What if you have a tensor with dynamic size but you want to reshape it? You can use **-1** to do just that.

In [14]:

```
# Introduce the use of -1 in tensor_obj.view(row, column) method

twoD_float_tensor = new_float_tensor.view(-1, 1)
print("Original Size: ", new_float_tensor)
print("Size after view method", twoD_float_tensor)
```

```
Original Size:  tensor([0., 1., 2., 3., 4.])
Size after view method tensor([[0.],
        [1.],
        [2.],
        [3.],
        [4.]])
```

You get the same result as the previous example. The **-1** can represent any size. However, be careful because you can set only one argument as **-1**.

You can also convert a **numpy** array to a **tensor**, for example:

In [15]:

```
# Convert a numpy array to a tensor

numpy_array = np.array([0.0, 1.0, 2.0, 3.0, 4.0])
new_tensor = torch.from_numpy(numpy_array)

print("The dtype of new tensor: ", new_tensor.dtype)
print("The type of new tensor: ", new_tensor.type())
```

```
The dtype of new tensor:  torch.float64
The type of new tensor:  torch.DoubleTensor
```

Converting a **tensor** to a **numpy** is also supported in PyTorch. The syntax is shown below:

In [16]:

```
# Convert a tensor to a numpy array

back_to_numpy = new_tensor.numpy()
print("The numpy array from tensor: ", back_to_numpy)
print("The dtype of numpy array: ", back_to_numpy.dtype)
```

```
The numpy array from tensor:  [0. 1. 2. 3. 4.]
The dtype of numpy array:  float64
```

`back_to_numpy` and `new_tensor` still point to `numpy_array` . As a result if we change `numpy_array` both `back_to_numpy` and `new_tensor` will change. For example if we set all the elements in `numpy_array` to zeros, `back_to_numpy` and `new_tensor` will follow suit.

In [17]:

```
# Set all elements in numpy array to zero
numpy_array[:] = 0
print("The new tensor points to numpy_array : ", new_tensor)
print("and back to numpy array points to the tensor: ", back_to_numpy)
```

```
The new tensor points to numpy_array :  tensor([0., 0., 0., 0., 0.], dt
ype=torch.float64)
and back to numpy array points to the tensor:  [0. 0. 0. 0. 0.]
```

Pandas Series can also be converted by using the numpy array that is stored in `pandas_series.values` . Note that `pandas_series` can be any pandas_series object.

In [18]:

```
# Convert a panda series to a tensor

pandas_series=pd.Series([0.1, 2, 0.3, 10.1])
new_tensor=torch.from_numpy(pandas_series.values)
print("The new tensor from numpy array: ", new_tensor)
print("The dtype of new tensor: ", new_tensor.dtype)
print("The type of new tensor: ", new_tensor.type())
```

```
The new tensor from numpy array:  tensor([ 0.1000,  2.0000,  0.3000, 1
0.1000], dtype=torch.float64)
The dtype of new tensor:  torch.float64
The type of new tensor:  torch.DoubleTensor
```

consider the following tensor

In [19]:

```
this_tensor=torch.tensor([0,1, 2,3])
```

The method `item()` returns the value of this tensor as a standard Python number. This only works for one element.

In [20]:

```
this_tensor=torch.tensor([0,1, 2,3])

print("the first item is given by",this_tensor[0].item(),"the first tensor value is
given by ",this_tensor[0])
print("the second item is given by",this_tensor[1].item(),"the second tensor value
 is given by ",this_tensor[1])
print("the third  item is given by",this_tensor[2].item(),"the third tensor value i
s given by ",this_tensor[2])
```

```
the first item is given by 0 the first tensor value is given by  tensor
(0)
the second item is given by 1 the second tensor value is given by  tens
or(1)
the third  item is given by 2 the third tensor value is given by  tenso
r(2)
```

we can use the method `tolist()` to return a list

In [21]:

```
torch_to_list=this_tensor.tolist()

print('tensor:', this_tensor,"\nlist:",torch_to_list)
```

```
tensor: tensor([0, 1, 2, 3])
list: [0, 1, 2, 3]
```

Practice

Try to convert `your_tensor` to a 1X5 tensor.

In [27]:

```
# Practice: convert the following tensor to a tensor object with 1 row and 5 column
s

your_tensor = torch.tensor([1, 2, 3, 4, 5])
your_new_tensor = your_tensor.view(1,5)
print("old size:", your_tensor)
print("new size:", your_new_tensor)
```

```
old size: tensor([1, 2, 3, 4, 5])
new size: tensor([[1, 2, 3, 4, 5]])
```

Double-click **here** for the solution.

Indexing and Slicing

In Python, **the index starts with 0**. Therefore, the last index will always be 1 less than the length of the tensor object. You can access the value on a certain index by using the square bracket, for example:

In [28]:

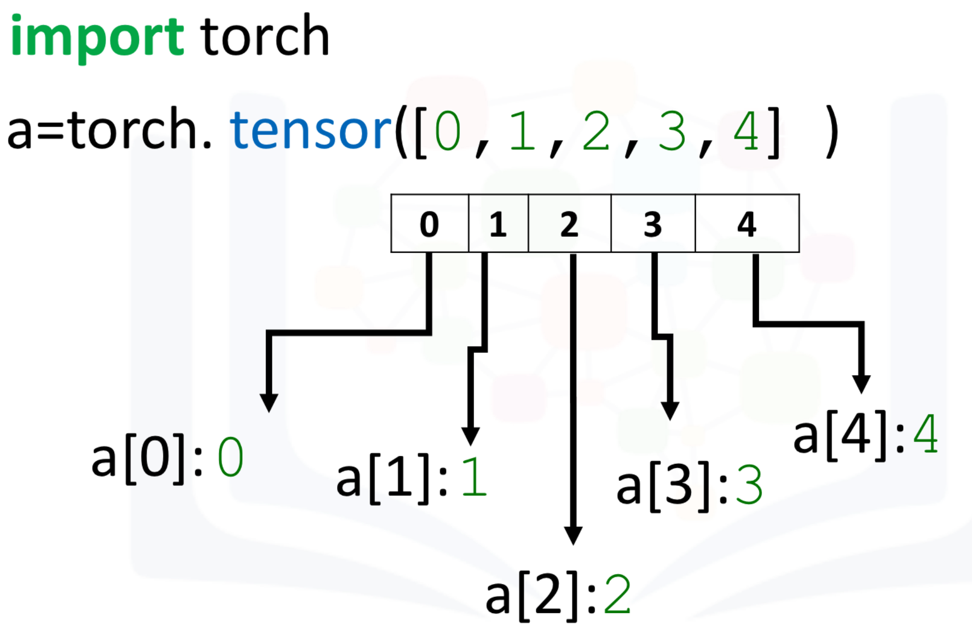
```
# A tensor for showing how the indexs work on tensors

index_tensor = torch.tensor([0, 1, 2, 3, 4])
print("The value on index 0:",index_tensor[0])
print("The value on index 1:",index_tensor[1])
print("The value on index 2:",index_tensor[2])
print("The value on index 3:",index_tensor[3])
print("The value on index 4:",index_tensor[4])
```

```
The value on index 0: tensor(0)
The value on index 1: tensor(1)
The value on index 2: tensor(2)
The value on index 3: tensor(3)
The value on index 4: tensor(4)
```

Note that the `index_tensor[5]` will create an error.

The index is shown in the following figure:



Now, you'll see how to change the values on certain indexes.

Suppose you have a tensor as shown here:

```
In [29]:
```

```python
# A tensor for showing how to change value according to the index

tensor_sample = torch.tensor([20, 1, 2, 3, 4])
```

Assign the value on index 0 as 100:

```
In [30]:
```

```python
# Change the value on the index 0 to 100

print("Inital value on index 0:", tensor_sample[0])
tensor_sample[0] = 100
print("Modified tensor:", tensor_sample)
```

```
Inital value on index 0: tensor(20)
Modified tensor: tensor([100,   1,   2,   3,   4])
```

As you can see, the value on index 0 changes. Change the value on index 4 to 0:

```
In [31]:
```

```python
# Change the value on the index 4 to 0

print("Inital value on index 4:", tensor_sample[4])
tensor_sample[4] = 0
print("Modified tensor:", tensor_sample)
```

```
Inital value on index 4: tensor(4)
Modified tensor: tensor([100,   1,   2,   3,   0])
```

The value on index 4 turns to 0.

If you are familiar with Python, you know that there is a feature called slicing on a list. Tensors support the same feature.

Get the subset of `tensor_sample` . The subset should contain the values in `tensor_sample` from index 1 to index 3.

In [32]:

```
# Slice tensor_sample

subset_tensor_sample = tensor_sample[1:4]
print("Original tensor sample: ", tensor_sample)
print("The subset of tensor sample:", subset_tensor_sample)
```

```
Original tensor sample:  tensor([100,   1,   2,   3,   0])
The subset of tensor sample: tensor([1, 2, 3])
```

As a result, the `subset_tensor_sample` returned only the values on index 1, index 2, and index 3. Then, it stored them in a `subset_tensor_sample` .

Note: The number on the left side of the colon represents the index of the first value. The number on the right side of the colon is always 1 larger than the index of the last value. For example, `tensor_sample[1:4]` means you get values from the index 1 to index 3 *(4-1)*.

As for assigning values to the certain index, you can also assign the value to the slices:

Change the value of `tensor_sample` from index 3 to index 4:

In [33]:

```
# Change the values on index 3 and index 4

print("Inital value on index 3 and index 4:", tensor_sample[3:5])
tensor_sample[3:5] = torch.tensor([300.0, 400.0])
print("Modified tensor:", tensor_sample)
```

```
Inital value on index 3 and index 4: tensor([3, 0])
Modified tensor: tensor([100,   1,   2, 300, 400])
```

The values on both index 3 and index 4 were changed. The values on other indexes remain the same.

You can also use a variable to contain the selected indexes and pass that variable to a tensor slice operation as a parameter, for example:

In [34]:

```
# Using variable to contain the selected index, and pass it to slice operation

selected_indexes = [3, 4]
subset_tensor_sample = tensor_sample[selected_indexes]
print("The inital tensor_sample", tensor_sample)
print("The subset of tensor_sample with the values on index 3 and 4: ", subset_tens
or_sample)
```

```
The inital tensor_sample tensor([100,   1,   2, 300, 400])
The subset of tensor_sample with the values on index 3 and 4:  tensor
([300, 400])
```

You can also assign one value to the selected indexes by using the variable. For example, assign 100,000 to all the `selected_indexes` :

In [35]:

```
#Using variable to assign the value to the selected indexes

print("The inital tensor_sample", tensor_sample)
selected_indexes = [1, 3]
tensor_sample[selected_indexes] = 100000
print("Modified tensor with one value: ", tensor_sample)
```

```
The inital tensor_sample tensor([100,   1,   2, 300, 400])
Modified tensor with one value:  tensor([   100, 100000,      2, 10000
0,    400])
```

The values on index 1 and index 3 were changed to 100,000. Others remain the same.

Note: You can use only one value for the assignment.

Practice

Try to change the values on index 3, 4, 7 of the following tensor to 0.

In [37]:

```
# Practice: Change the values on index 3, 4, 7 to 0

practice_tensor = torch.tensor([2, 7, 3, 4, 6, 2, 3, 1, 2])
slected_indexes = [3, 4, 7]
practice_tensor[selected_indexes] = 0
print(practice_tensor)
```

```
tensor([2, 0, 3, 0, 6, 2, 3, 1, 2])
```

Double-click **here** for the solution.

Tensor Functions

For this section, you'll work with some methods that you can apply to tensor objects.

Mean and Standard Deviation

You'll review the mean and standard deviation methods first. They are two basic statistical methods.

Create a tensor with values *[1.0, -1, 1, -1]*:

In [38]:

```
# Sample tensor for mathmatic calculation methods on tensor

math_tensor = torch.tensor([1.0, -1.0, 1, -1])
print("Tensor example: ", math_tensor)
```

```
Tensor example:  tensor([ 1., -1.,  1., -1.])
```

Here is the mean method:

In [39]:

```
#Calculate the mean for math_tensor

mean = math_tensor.mean()
print("The mean of math_tensor: ", mean)
```

```
The mean of math_tensor:  tensor(0.)
```

The standard deviation can also be calculated by using `tensor_obj.std()` :

In [40]:

```
#Calculate the standard deviation for math_tensor

standard_deviation = math_tensor.std()
print("The standard deviation of math_tensor: ", standard_deviation)
```

```
The standard deviation of math_tensor:  tensor(1.1547)
```

Max and Min

Now, you'll review another two useful methods: *tensor_obj*`.max()` and *tensor_obj*`.min()` . These two methods are used for finding the maximum value and the minimum value in the tensor.

Create a `max_min_tensor` :

In [41]:

```
# Sample for introducing max and min methods

max_min_tensor = torch.tensor([1, 1, 3, 5, 5])
print("Tensor example: ", max_min_tensor)
```

```
Tensor example:  tensor([1, 1, 3, 5, 5])
```

Note: There are two minimum numbers as 1 and two maximum numbers as 5 in the tensor. Can you guess how PyTorch is going to deal with the duplicates?

Apply *tensor_obj*`.max()` on `max_min_tensor` :

In [42]:

```
# Method for finding the maximum value in the tensor

max_val = max_min_tensor.max()
print("Maximum number in the tensor: ", max_val)
```

```
Maximum number in the tensor:  tensor(5)
```

The answer is `tensor(5)` . Therefore, the method *tensor_obj*`.max()` is grabbing the maximum value but not the elements that contain the maximum value in the tensor.

In [43]:

```
max_min_tensor.max()
```

Out[43]:

```
tensor(5)
```

Use *tensor_obj*`.min()` on `max_min_tensor` :

In [44]:

```
# Method for finding the minimum value in the tensor

min_val = max_min_tensor.min()
print("Minimum number in the tensor: ", min_val)
```

```
Minimum number in the tensor:  tensor(1)
```

The answer is `tensor(1)` . Therefore, the method *tensor_obj*`.min()` is grabbing the minimum value but not the elements that contain the minimum value in the tensor.

Sin

Sin is the trigonometric function of an angle. Again, you will not be introducedvto any mathematic functions. You'll focus on Python.

Create a tensor with 0, π/2 and π. Then, apply the sin function on the tensor. Notice here that the `sin()` is not a method of tensor object but is a function of torch:

In [45]:

```
# Method for calculating the sin result of each element in the tensor

pi_tensor = torch.tensor([0, np.pi/2, np.pi])
sin = torch.sin(pi_tensor)
print("The sin result of pi_tensor: ", sin)
```

```
The sin result of pi_tensor:  tensor([ 0.0000e+00,  1.0000e+00, -8.7423
e-08])
```

The resultant tensor `sin` contains the result of the `sin` function applied to each element in the `pi_tensor` .
This is different from the previous methods. For *tensor_obj*`.mean()` , *tensor_obj*`.std()` , *tensor_obj*`.max()` , and *tensor_obj*`.min()` , the result is a tensor with only one number because these are aggregate methods.
However, the `torch.sin()` is not. Therefore, the resultant tensors have the same length as the input tensor.

Create Tensor by `torch.linspace()`

A useful function for plotting mathematical functions is `torch.linspace()` . `torch.linspace()` returns evenly spaced numbers over a specified interval. You specify the starting point of the sequence and the ending point of the sequence. The parameter `steps` indicates the number of samples to generate. Now, you'll work with `steps = 5` .

In [46]:

```
# First try on using linspace to create tensor

len_5_tensor = torch.linspace(-2, 2, steps = 5)
print ("First Try on linspace", len_5_tensor)
```

```
First Try on linspace tensor([-2., -1.,  0.,  1.,  2.])
```

Assign `steps` with 9:

In [47]:

```
# Second try on using linspace to create tensor

len_9_tensor = torch.linspace(-2, 2, steps = 9)
print ("Second Try on linspace", len_9_tensor)
```

```
Second Try on linspace tensor([-2.0000, -1.5000, -1.0000, -0.5000,  0.0
000,  0.5000,  1.0000,  1.5000,
         2.0000])
```

Use both `torch.linspace()` and `torch.sin()` to construct a tensor that contains the 100 sin result in range from 0 (0 degree) to 2π (360 degree):

In [48]:

```
# Construct the tensor within 0 to 360 degree

pi_tensor = torch.linspace(0, 2*np.pi, 100)
sin_result = torch.sin(pi_tensor)
```

Plot the result to get a clearer picture. You must cast the tensor to a numpy array before plotting it.

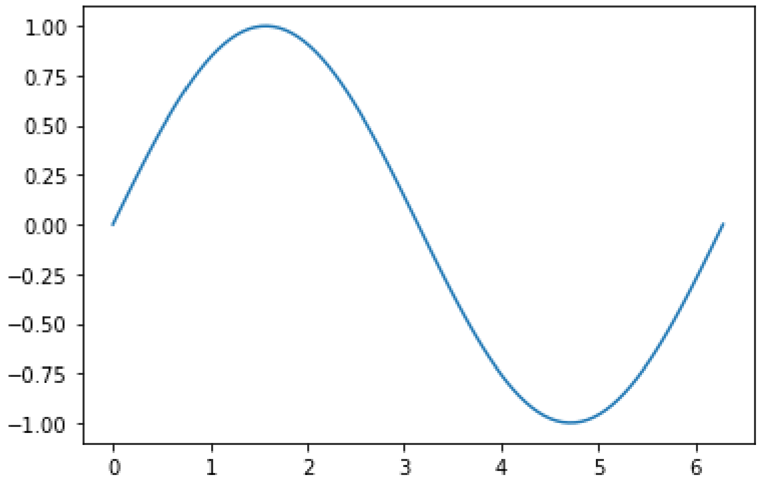
In [49]:

```
# Plot sin_result

plt.plot(pi_tensor.numpy(), sin_result.numpy())
```

Out[49]:

```
[<matplotlib.lines.Line2D at 0x7fbc51a2e4e0>]
```



If you know the trigonometric function, you will notice this is the diagram of the sin result in the range 0 to 360 degrees.

Practice

Construct a tensor with 25 steps in the range 0 to π/2. Print out the Maximum and Minimum number. Also, plot a graph showing the diagram that shows the result.

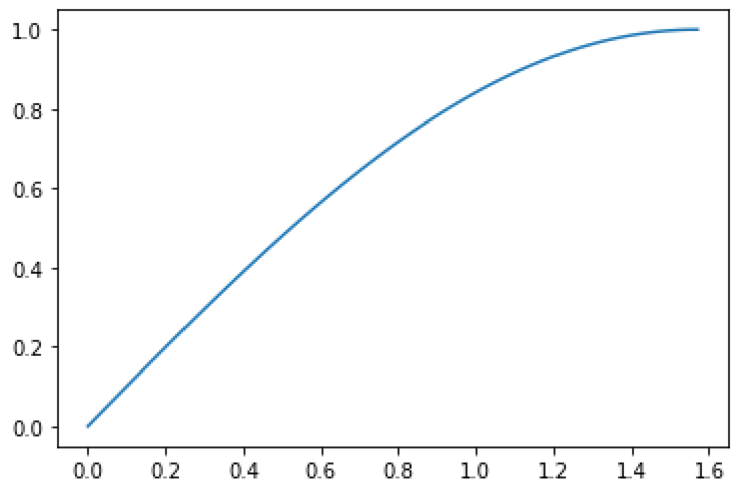
In [53]:

```
# Practice: Create your tensor, print max and min number, plot the sin result diagr
am
value = torch.linspace(0, np.pi/2, 100)
print("Maximum = ", value.max())
print("Minimum = ", value.min())
sin_value = torch.sin(value)
plt.plot(value.numpy(), sin_value.numpy())
```

```
Maximum =  tensor(1.5708)
Minimum =  tensor(0.)
```

Out[53]:

```
[<matplotlib.lines.Line2D at 0x7fbc51921208>]
```



Double-click **here** for the solution.

Tensor Operations

In the following section, you'll work with operations that you can apply to a tensor.

Tensor Addition

You can perform addition between two tensors.

Create a tensor `u` with 1 dimension and 2 elements. Then, create another tensor `v` with the same number of dimensions and the same number of elements:

In [54]:

```
# Create two sample tensors

u = torch.tensor([1, 0])
v = torch.tensor([0, 1])
```

Add `u` and `v` together:

In [55]:

```
# Add u and v

w = u + v
print("The result tensor: ", w)
```

```
The result tensor:  tensor([1, 1])
```

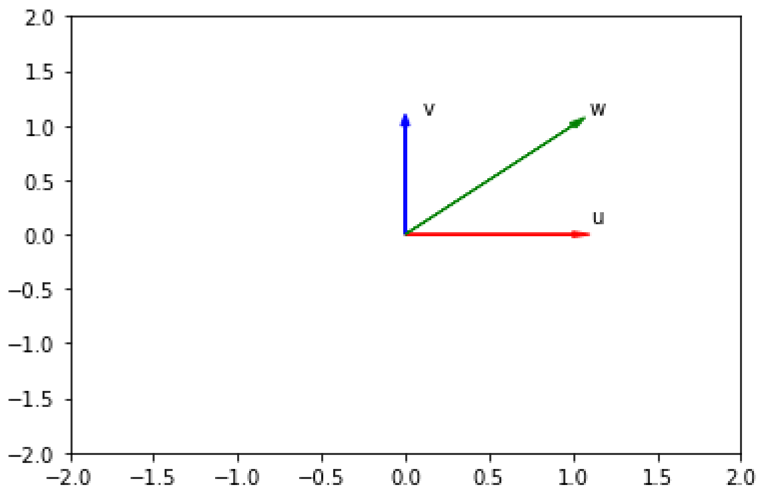
The result is `tensor([1, 1])`. The behavior is *[1 + 0, 0 + 1]*.

Plot the result to to get a clearer picture.

In [56]:

```
# Plot u, v, w

plotVec([
    {"vector": u.numpy(), "name": 'u', "color": 'r'},
    {"vector": v.numpy(), "name": 'v', "color": 'b'},
    {"vector": w.numpy(), "name": 'w', "color": 'g'}
])
```



Try

Implement the tensor subtraction with `u` and `v` as u-v.

In [58]:

```
# Try by yourself to get a result of u-v

u = torch.tensor([1, 0])
v = torch.tensor([0, 1])
d = u-v
print(d)
```

```
tensor([ 1, -1])
```

Double-click **here** for the solution.

Tensors must be of the same data type to perform addition as well as other operations.If you uncomment the following code and try to run it you will get an error as the two tensors are of two different data types.

In [59]:

```
torch.tensor([1,2,3],dtype=torch.int64)+torch.tensor([1,2,3],dtype=torch.float6464)
```

```
---------------------------------------------------------------------------
----
AttributeError                            Traceback (most recent call l
ast)
<ipython-input-59-c6bf7292bee5> in <module>
----> 1 torch.tensor([1,2,3],dtype=torch.int64)+torch.tensor([1,2,3],dt
ype=torch.float6464)

AttributeError: module 'torch' has no attribute 'float6464'
```

You can add a scalar to the tensor. Use `u` as the sample tensor:

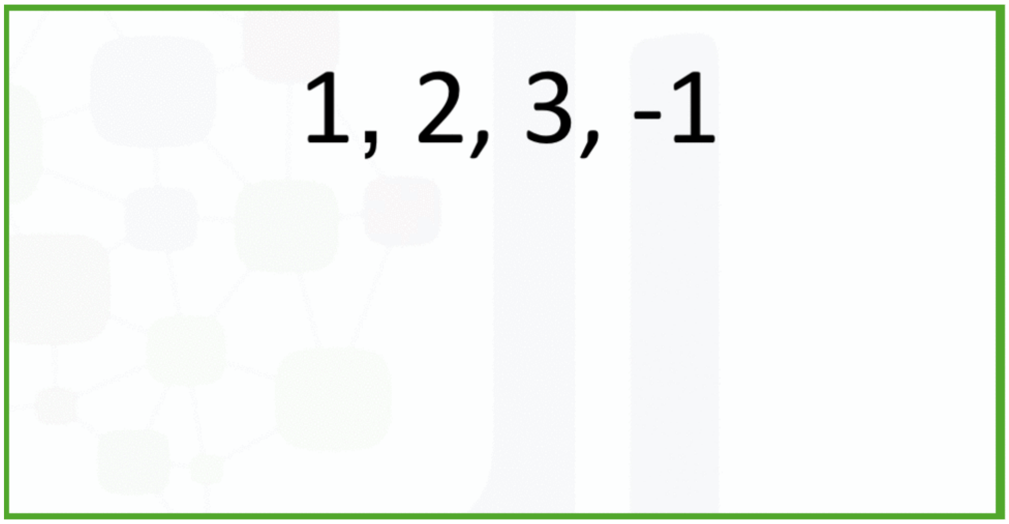
In [60]:

```
# tensor + scalar

u = torch.tensor([1, 2, 3, -1])
v = u + 1
print ("Addition Result: ", v)
```

```
Addition Result:  tensor([2, 3, 4, 0])
```

The result is simply adding 1 to each element in tensor `u` as shown in the following image:



Tensor Multiplication

Now, you'll review the multiplication between a tensor and a scalar.

Create a tensor with value `[1, 2]` and then multiply it by 2:

In [61]:

```
# tensor * scalar

u = torch.tensor([1, 2])
v = 2 * u
print("The result of 2 * u: ", v)
```

```
The result of 2 * u:  tensor([2, 4])
```

The result is `tensor([2, 4])`, so the code `2 * u` multiplies each element in the tensor by 2. This is how you get the product between a vector or matrix and a scalar in linear algebra.

You can use multiplication between two tensors.

Create two tensors `u` and `v` and then multiply them together:

In [62]:

```
# tensor * tensor

u = torch.tensor([1, 2])
v = torch.tensor([3, 2])
w = u * v
print ("The result of u * v", w)
```

```
The result of u * v tensor([3, 4])
```

The result is simply `tensor([3, 4])` . This result is achieved by multiplying every element in `u` with the corresponding element in the same position `v` , which is similar to *[1 * 3, 2 * 2]*.

Dot Product

The dot product is a special operation for a vector that you can use in Torch.

Here is the dot product of the two tensors `u` and `v` :

In [63]:

```
# Calculate dot product of u, v

u = torch.tensor([1, 2])
v = torch.tensor([3, 2])

print("Dot Product of u, v:", torch.dot(u,v))
```

```
Dot Product of u, v: tensor(7)
```

The result is `tensor(7)` . The function is *1 x 3 + 2 x 2 = 7*.

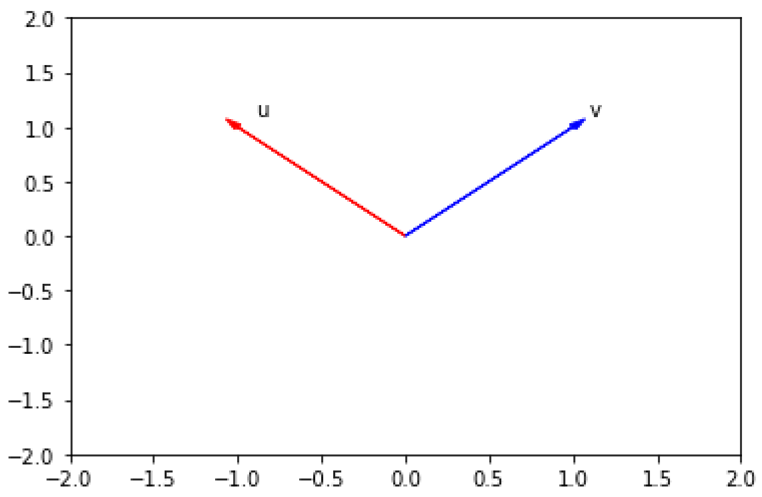
Practice

Convert the list *[-1, 1]* and *[1, 1]* to tensors `u` and `v` . Then, plot the tensor `u` and `v` as a vector by using the function `plotVec` and find the dot product:

In [67]:

```python
# Practice: calculate the dot product of u and v, and plot out two vectors
u = torch.tensor([-1, 1])
v = torch.tensor([1, 1])
plotVec([
    {"vector": u.numpy(), "name": 'u', "color": 'r'},
    {"vector": v.numpy(), "name": 'v', "color": 'b'}
])
print("The Dot Product is:", np.dot(u,v))
```

```
The Dot Product is: 0
```



Double-click **here** for the solution.

See Broadcasting (https://docs.scipy.org/doc/numpy-1.13.0/user/basics.broadcasting.html) for more information on numpy that is similar to PyTorch.

About the Authors:

Joseph Santarcangelo (https://www.linkedin.com/in/joseph-s-50398b136/) has a PhD in Electrical Engineering, his research focused on using machine learning, signal processing, and computer vision to determine how videos impact human cognition. Joseph has been working for IBM since he completed his PhD.

Other contributors: Michelle Carey (https://www.linkedin.com/in/michelleccarey/), Mavis Zhou (www.linkedin.com/in/jiahui-mavis-zhou-a4537814a)
