1. Write the Python code to implement a single neuron.

Neural networks are the core of deep learning, a field that has practical applications in many different areas. Today neural networks are used for image classification, speech recognition, object detection, etc. Now, Let’s try to understand the basic unit behind all these states of art techniques.  
A single neuron transforms given input into some output. Depending on the given input and weights assigned to each input, decide whether the neuron fired or not. Let’s assume the neuron has 3 input connections and one output.

Neural networks (NN), also called artificial neural networks (ANN) are a subset of learning algorithms within the machine learning field that are loosely based on the concept of biological neural networks.

[Andrey Bulezyuk](https://www.liveedu.tv/andreybu/REaxr-machine-learning-model-python-sklearn-kera/oPGdP-machine-learning-model-python-sklearn-kera/), who is a German-based machine learning specialist with more than five years of experience, says that “neural networks are revolutionizing machine learning because they are capable of efficiently modeling sophisticated abstractions across an extensive range of disciplines and industries.”

Basically, an ANN comprises of the following components:

An input layer that receives data and pass it on

A hidden layer

An output layer

Weights between the layers

A deliberate activation function for every hidden layer. In this simple neural network Python tutorial, we’ll employ the Sigmoid activation function.

There are several types of neural networks. In this project, we are going to create the feed-forward or perception neural networks. This type of ANN relays data directly from the front to the back.

Training the feed-forward neurons often need back-propagation, which provides the network with corresponding set of inputs and outputs. When the input data is transmitted into the neuron, it is processed, and an output is generated.

Here is a diagram that shows the structure of a simple neural network:

And, the best way to understand how neural networks work is to learn how to build one from scratch (without using any library).

In this article, we’ll demonstrate how to use the Python programming language to create a simple neural network.

The problem

Here is a table that shows the problem.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Input | |  | Output |
| Training data 1 | 0 | 0 | 1 | 0 |
| Training data 2 | 1 | 1 | 1 | 1 |
| Training data 3 | 1 | 0 | 1 | 1 |
| Training data 4 | 0 | 1 | 1 | 0 |
|  | | | | |
| New Situation | 1 | 0 | 0 | ? |

We are going to train the neural network such that it can predict the correct output value when provided with a new set of data.

As you can see on the table, the value of the output is always equal to the first value in the input section. Therefore, we expect the value of the output (?) to be 1.

Let’s see if we can use some Python code to give the same result (You can peruse the code for this project at the end of this article before continuing with the reading).

Creating a NeuralNetwork Class

We’ll create a NeuralNetwork class in Python to train the neuron to give an accurate prediction. The class will also have other helper functions.

Even though we’ll not use a neural network library for this simple neural network example, we’ll import the numpy library to assist with the calculations.

The library comes with the following four important methods:

exp—for generating the natural exponential

array—for generating a matrix

dot—for multiplying matrices

random—for generating random numbers. Note that we’ll seed the random numbers to ensure their efficient distribution.

2.Write the Python code to implement ReLU.

Implementing ReLu function in Python

Applying Relu on (1.0) gives 1.0 Applying Relu on (-10.0) gives 0.0 Applying Relu on (0.0) gives 0.0 Applying Relu on (15.0) gives 15.0 Applying Relu on (-20.0) gives 0.0.

f'(x) = 1, x>=0 = 0, x<0.

f(x)= 0. ...

f'(x) = 1, x>=0 = 0.01, x<0.

def relu(x): if x>0 : return x else : return 0.01\*x

3.Write the Python code for a dense layer in terms of matrix multiplication.

Layers in the deep learning model can be considered as the architecture of the model. There can be various types of layers that can be used in the models. All of these different layers have their own importance based on their features. Like we use [LSTM](https://analyticsindiamag.com/a-complete-guide-to-lstm-architecture-and-its-use-in-text-classification/) layers mostly in the time series analysis or in the [NLP](https://analyticsindiamag.com/a-complete-guide-to-lstm-architecture-and-its-use-in-text-classification/) problems, [convolutional layers](https://analyticsindiamag.com/what-is-a-convolutional-layer/) in [image processing](https://analyticsindiamag.com/image-processing-with-opencv-in-python/), etc. A dense layer also referred to as a fully connected layer is a layer that is used in the final stages of the neural network. This layer helps in changing the dimensionality of the output from the preceding layer so that the model can easily define the relationship between the values of the data in which the model is working. In this article, we will discuss the dense layer in detail with its importance and work. The major points to be discussed in this article are listed below.

What is a Dense Layer?

In any [neural network](https://analyticsindiamag.com/a-beginners-guide-to-neural-network-pruning/), a dense layer is a layer that is deeply connected with its preceding layer which means the neurons of the layer are connected to every neuron of its preceding layer. This layer is the most commonly used layer in [artificial neural network networks](https://analyticsindiamag.com/artificial-neural-networks-in-r/).

The dense layer’s neuron in a model receives output from every neuron of its preceding layer, where neurons of the dense layer perform matrix-vector multiplication. Matrix vector multiplication is a procedure where the row vector of the output from the preceding layers is equal to the column vector of the dense layer. The general rule of matrix-vector multiplication is that the row vector must have as many columns like the column vector.

Where A is a (M x N) matrix and x is a (1 ???? N) matrix. Values under the matrix are the trained parameters of the preceding layers and also can be updated by the backpropagation. Backpropagation is the most commonly used algorithm for training the feedforward neural networks. Generally, backpropagation in a neural network computes the gradient of the loss function with respect to the weights of the network for single input or output. From the above intuition, we can say that the output coming from the dense layer will be an N-dimensional vector. We can see that it is reducing the dimension of the vectors. So basically a dense layer is used for changing the dimension of the vectors by using every neuron.

As discussed before, results from every neuron of the preceding layers go to every single neuron of the dense layer. So we can say that if the preceding layer outputs a (M x N) matrix by combining results from every neuron, this output goes through the dense layer where the count of neurons in a dense layer should be N. We can implement it using Keras, in the next part of the article we will see some of the major parameters of the dense layer using Keras with their definitions.

Dense Layer from Keras

Keras provide dense layers through the following syntax:

tf.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,

    \*\*kwargs

)

Keras Dense Layer Hyperparameters

As we can see a set of hyperparameters being used in the above syntax, let us try to understand their significance.

Units

Units are one of the most basic and necessary parameters of the Keras dense layer which defines the size of the output from the dense layer. It must be a positive integer since it represents the dimensionality of the output vector.

Activation

In neural networks, the activation function is a function that is used for the transformation of the input values of neurons. Basically, it introduces the non-linearity into the networks of neural networks so that the networks can learn the relationship between the input and output values.

If in this Keras layer no activation is defined it will consider the linear activation function. The following options are available as activation functions in Keras.

Relu function (activation = activations.relu) – rectified linear unit activation function

Sigmoid function(activation = activations.sigmoid) – Sigmoid activation function, sigmoid(x) = 1 / (1 + exp(-x)).

Softmax function(activation = activations.softmax) – softmax converts a vector of value to a probability distribution.

Softplus function(activation = activations.softplus) – Softplus activation function, softplus(x) = log(exp(x) + 1).

Softsign function(activation = activations.softplus) – Softsign activation function, softsign(x) = x / (abs(x) + 1).

Tanh function(activation = activation.tanh) – Hyperbolic tangent activation function

Selu function(activation = activations.selu) – Scaled Exponential Linear Unit (SELU).

Elu function(activation = activations.elu) – Exponential Linear Unit.

Exponential function(activation = activations.exponential) – Exponential Activation function.

use\_bias

 Use\_Bias parameter is used for deciding whether we want a dense layer to use a bias vector or not. It is a boolean parameter if not defined then use\_bias is set to true.

kernel\_initializer

This parameter is used for initializing the kernel weights matrix. The weight matrix is a matrix of weights that are multiplied with the input to extract relevant feature kernels.

bias\_initializer

This parameter is used for initializing the bias vector. A bias vector can be defined as the additional sets of weight that require no input and correspond to the output layer. By default, it is set as zeros.

Kernel regularizer

This parameter is used for regularization of the kernel weight matrix if we have initialized any matrix in the kernal\_initializer.

 bias\_regularizer

This parameter is used for regularization of the bias vector if we have initialized any vector in the bias\_initializer. By default, it is set as none.

Activity\_regularizer

This parameter is used for the regularization of the activation function which we have defined in the activation parameter. It is applied to the output of the layer. By default, it is set as none.

kernal\_constraint

This parameter is used to apply the constraint function to the kernel weight matrix. By default, it is set as none.

Bias\_constraint

This parameter is used to apply the constraint function to the bias vector. By default, it is set as none.

Basic Operations with Dense Layer

As we have seen in the parameters we have three main attributes: activation function, weight matrix, and bias vector. Using these attributes a dense layer operation can be represented as:

Output = activation(dot(input, kernel) + bias)

Where if the input matrix for the dense layer has a rank of more than 2, then dot product between the kernel and input along the last axis of the input and zeroth axis of the kernel using the tf.tensordot calculated by the dense layer if the use\_bias is False.

How to Implement the Dense Layer?

In this section of the article, we will see how to implement a dense layer in a neural network with a single dense layer and a neural network with multiple dense layers.

A sequential model with a single dense layer.

import tensorflow

model = tensorflow.keras.models.Sequential()

model.add(tensorflow.keras.Input(shape=(16,)))

model.add(tensorflow.keras.layers.Dense(32, activation='relu'))

print(model.output\_shape)

print(model.compute\_output\_signature)

Output:

Here in the output, we can see that the output of the model is a size of (None,32) and we are using a single Keras layer and the signature of the output from the model is a sequential object.

4.Write the Python code for a dense layer in plain Python (that is, with list comprehensions and functionality built into Python).

Lists are a helpful and frequently used feature in Python.

And list comprehension gives you a way to create lists while writing more elegant code that is easy to read.

In this beginner-friendly article, I'll give an overview of how list comprehension works in Python. I'll also show plenty of code examples along the way.

Let's get started!

How to use a for loop to create a list in Python

One way to create a list in Python is by using a for loop.

For example, you can use the range() function to create a list of numbers ranging from 0 - 4.

#first create an empty list

my\_list = []

#iterate over the numbers 0 - 4 using the range() function

#range(5) creates an iterable, starting from 0 up to (but not including) 5

#Use the .append() method to add the numbers 0 - 4 to my\_list

for num in range(5):

my\_list.append(num)

#print my\_list

print(my\_list)

#output

#[0, 1, 2, 3, 4]

What if you already have a list of numbers, but want to create a new list with their squares?

You could again use a for loop, like so:

#initial list of numbers

numbers = [1,2,3,4,5,6]

#create a new,empty list to hold their squares

square\_numbers = []

#iterate over initial list

#multiply each number by itself

#use .append() method, to add the square to the new list, square\_numbers

for num in numbers:

square\_numbers.append(num \* num)

#print new list

print(square\_numbers)

#output

#[1, 4, 9, 16, 25, 36]

But there is a quicker and more succinct way to achieve the same results – by using list comprehension.

What is list comprehension in Python? A syntax overview

When you're analyzing and working with lists in Python, you'll often have to manipulate, modify, or perform calculations on every single item in the list, all at once.

You may also need to create new lists from scratch, or create a new list based on the values of an already existing list.

List comprehension is a fast, short, and elegant way to create lists compared to other iterative methods, like for loops.

The general syntax for list comprehension looks like this:

new\_list = [expression for variable in iterable]

Let's break it down:

List comprehensions start and end with opening and closing square brackets, [].

Then comes the expression or operation you'd like to perform and carry out on each value inside the current iterable. The results of these calculations enter the new list.

The expression is followed by a for clause.

variable is a temporary name you want to use for each item in the current list that is going through the iteration.

The in keyword is used to loop over the iterable.

iterable can be any Python object, such as a list, tuple, string and so on.

From the iteration that was performed and the calculations that took place on each item during the iteration, new values were created which are saved to a variable, in this case new\_list. The old list (or other object) will remain unchanged.

There can be an optional if statement and additional for clause.

How to use list comprehension in Python

Using the same example from earlier on, here is how you'd create a new list of numbers from 0 - 4 with the range() function in just one single line, using list comprehension:

new\_list = [num for num in range(5)]

print(new\_list)

#output

#[0, 1, 2, 3, 4]

This has the same output as the for loop example, but with significantly less code!

Let's break it down:

the iterable in this case is a sequence of numbers from 0 to 4, using range(5). range() constructs a list of numbers.

You use the in keyword to iterate over the numbers.

The num following the for clause is a variable, a temporary name for each value in the iterable. So num would be equal to 0 in the first iteration, then num would be equal to 1 in the next iteration and so on, until it reached and equalled the number 4, where the iteration would stop.

The num before the for clause is an expression for each item in the sequence.

Finally, the new list (or other iterable) that is created gets stored in the variable new\_list.

You can even perform mathematical operations on the items contained in the iterable and the result will be added to the new list:

new\_list = [num \* 2 for num in range(5)]

print(new\_list)

#output

#[0, 2, 4, 6, 8]

Here each number in range(5) will be multiplied by two and the new value will be stored in the variable new\_list.

What if you had a pre-existing list where you wanted to manipulate and modify each item in it? This would be similar to the example from earlier on, where we created a list of squares.

Again, you can achieve that with just one line of code, using list comprehension:

#initial list

numbers = [1,2,3,4,5,6]

#new list

#num \* num is the operation that takes place to create the squares

square\_numbers = [num \* num for num in numbers]

print(square\_numbers)

#output

[1, 4, 9, 16, 25, 36]

How to use conditionals with list comprehension in Python

Optionally, you can use an if statement with a list comprehension.

The general syntax looks like this:

new\_list = [expression for variable in iterable if condition == True]

Conditionals act as a filter and add an extra check for additional precision and customisation when creating a new list.

This means that the value in the expression has to meet certain criteria and a certain condition you speficy, in order to go in the new list.

new\_list = [num for num in range(50) if num % 2 == 0]

print(new\_list)

#output

#[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48]

In the example above, only the values where the condition num % 2 == 0 is checked and evaluates to True will enter new\_list.

The modulo operator is used on every single one of the numbers in the sequence of numbers starting from 0 and ending in 49.

If the remainder of the numbers when divided by 2 is 0, then and only then does it enter the list.

So in this case, it creates a list of only even numbers.

You can then make it as specific as you want.

For example, you could add more than one condition, like so:

new\_list = [num for num in range(50) if num > 20 and num % 2 == 0]

print(new\_list)

#output

#[22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48]

In this example, there are two conditions num > 20 and num % 2 == 0.

The and operator indicates that both have to be met in order for the value to be added to the new list.

The values that don't meet the conditions are excluded and are not added.

5.What is the “hidden size” of a layer?

The size of the hidden layer is normally between the size of the input and output-. It should be should be 2/3 the size of the input layerplus the size of the o/p layer The number of hidden neurons should be less than twice the size of the input layer.

6.What does the t method do in PyTorch?

Expects input to be <= 2-D tensor and transposes dimensions 0 and 1. 0-D and 1-D tensors are returned as is. When input is a 2-D tensor this is equivalent to transpose(input, 0, 1) .

Returns a tensor that is a transposed version of input. The given dimensions dim0 and dim1 are swapped.

If input is a strided tensor then the resulting out tensor shares its underlying storage with the input tensor, so changing the content of one would change the content of the other.

If input is a [sparse tensor](https://pytorch.org/docs/stable/sparse.html#sparse-docs) then the resulting out tensor does not share the underlying storage with the input tensor.

If input is a [sparse tensor](https://pytorch.org/docs/stable/sparse.html#sparse-docs) with compressed layout (SparseCSR, SparseBSR, SparseCSC or SparseBSC) the arguments dim0 and dim1 must be both batch dimensions, or must both be sparse dimensions. The batch dimensions of a sparse tensor are the dimensions preceding the sparse dimensions.

7.Why is matrix multiplication written in plain Python very slow?

I try to find an explanation why my matrix multiplication with Numba is much slower than using NumPy's dot function. Although I am using the most basic code for writing a matrix multiplication function with Numba, I don't think that the significantly slower performance is due to the algorithm. For simplicity, I consider two k x k square matrices, A and B. My code reads

1 @njit('f8[:,:](f8[:,:], f8[:,:])')

2 def numba\_dot(A, B):

3

4 k=A.shape[1]

5 C = np.zeros((k, k))

6

7 for i in range(k):

8 for j in range(k):

9

10 tmp = 0.

11 for l in range(k):

12 tmp += A[i, l] \* B[l, j]

13

14 C[i, j] = tmp

15

16 return C

Running this code repeatedly with two random matrices 1000 x 1000 Matrices, it typically takes at least about 1.5 seconds to finish. On the other hand, if I don't update the matrix c.

8.In matmul, why is ac==br?

Matrix Multiplication

A great and interactive place to understand matrix multiplication as referenced by Jeremy is available at.

Essentially, as can be seen from the image, we take the transpose of the second matrix, multiply and add elements together to get the result. As an example, the very first item 15 in the resulting matrix comes from 1\*2 + 6\*2 + 1\*1 = 2 + 12 + 1 = 15.

While this explanation is visually pleasing, in my humble opinion, it is hard to convert it to code. So let’s use Excel and understand matrix multiplication in another way!

Matrix Multiplication in EXCEL

Consider two matrices A and B of size 4x3and 3x4.  
Note: I will be using bold notation A and B to refer to matrices in this article. It is common practice to reference vectors and matrices using bold notation.

From the image we saw before, if you remember, we took the transpose of B and multiplied rows of A with columns of B to get the resulting matrix. Rather, let’s skip this step of taking the transpose this time and multiply rows and columns straight away.

Therefore, Row 0 of A gets multiplied with column 0 of B element wise and resulting element wise products get added to get the first item in resulting matrix C at position [0][0].

9.In Jupyter Notebook, how do you measure the time taken for a single cell to execute?

Measure execution time in Jupyter Notebook: %timeit , %%timeit. In Jupyter Notebook (IPython), you can use the magic commands %timeit and %%timeit to measure the execution time of your code.

The nbextension offers a few options for how to display and interpret timestamps. Options are stored in the notebook section of the server’s nbconfig, under the key ExecuteTime. The easiest way to configure these is using the [jupyter\_nbextensions\_configurator](https://github.com/Jupyter-contrib/jupyter_nbextensions_configurator), which if you got this nbextension in the usual way from [jupyter\_contrib\_nbextensions](https://github.com/ipython-contrib/jupyter_contrib_nbextensions), should also have been installed.

Alternatively, you can also configure them directly with a few lines of python. For example, to alter the displayed message, use relative timestamps, and set them to update every 5 seconds, we can use the following python snippet:

from notebook.services.config import ConfigManager

ConfigManager().update('notebook', {'ExecuteTime': {

'display\_absolute\_timestamps': False,

'relative\_timing\_update\_period': 5,

'template': {

'executed': 'started ${start\_time}, finished in ${duration}',

}

}})

The available options are:

ExecuteTime.clear\_timings\_on\_clear\_output: When cells’ outputs are cleared, also clear their timing data, e.g. when using the Kernel > Restart & Clear Output menu item

ExecuteTime.clear\_timings\_on\_kernel\_restart: Clear all cells’ execution timing data on any kernel restart event

ExecuteTime.display\_absolute\_timings: Display absolute timings for the start/end time of execution. Setting this false will result in the display of a relative timestamp like ‘a few seconds ago’ (see the moment.js function [fromNow](https://momentjs.com/docs/" \l "/displaying/fromnow/) for details). Defaults to true.

ExecuteTime.display\_absolute\_format: The format to use when displaying absolute timings (see ExecuteTime.display\_absolute\_timings, above). See the moment.js function [format](https://momentjs.com/docs/#/displaying/format/) for details of the template tokens available. Defaults to 'YYYY-MM-DD HH:mm:ss'.

ExecuteTime.relative\_timing\_update\_period: Seconds to wait between updating the relative timestamps, if using them (see ExecuteTime.display\_absolute\_timings, above). Defaults to 10.

ExecuteTime.display\_in\_utc: Display times in UTC, rather than in the local timezone set by the browser. Defaults to false.

ExecuteTime.default\_kernel\_to\_utc: For kernel timestamps which do not specify a timezone, assume UTC. Defaults to true.

ExecuteTime.display\_right\_aligned: Right-align the text in the timing area under each cell. Defaults to false.

ExecuteTime.highlight.use: Highlight the displayed execution time on completion of execution. Defaults to true.

ExecuteTime.highlight.color: Color to use for highlighting the displayed execution time. Defaults to '#00BB00'.

ExecuteTime.template.executed: Template for the timing message for executed cells. See readme for replacement tokens. Defaults to 'executed in ${duration}, finished ${end\_time}'.

ExecuteTime.template.queued: Template for the timing message for queued cells. The template uses an ES2015-like syntax, but replaces only the exact strings ${start\_time}, plus (if defined) ${end\_time} and ${duration}. Defaults to 'execution queued ${start\_time}'.

Limitations

timezones

As discussed in [ipython-contrib/jupyter\_contrib\_nbextensions#549](https://github.com/ipython-contrib/jupyter_contrib_nbextensions/issues/549), [ipython-contrib/jupyter\_contrib\_nbextensions#904](https://github.com/ipython-contrib/jupyter_contrib_nbextensions/issues/904), and [jupyter/jupyter\_client#143](https://github.com/jupyter/jupyter_client/issues/143), although they are (now) supposed to, Jupyter kernels don’t always specify a timezone for their timestamps, which can cause problems when the [moment.js](https://momentjs.com/) library assumes the local timezone, rather than UTC, which is what most kernels are actually using. To help to address this, see the [options](https://jupyter-contrib-nbextensions.readthedocs.io/en/latest/nbextensions/execute_time/readme.html#Options) above, which can be used to assume UTC for unzoned timestamps.

execution queues

For a reason I don’t understand, when multiple cells are queued for execution, the kernel doesn’t send a reply immediately after finishing executing each cell. Some replies are delayed, and sent at the same time as later replies, meaning that the output of a cell can be updated with its finished value, before the notebook recieves the kernel execution reply. For the same reason, you can see this in the fact that the star for an executing cell can remain next to two cells at once, if several are queued to execute together. Since this extension uses the times in the kernel message (see internals, below), and these remain correct, the timings displayed are still accurate, but they may get displayed later due to this kernel issue.

Installation

Install the master version of the jupyter\_contrib\_nbextensions repository as explained in the docs at [jupyter-contrib-nbextensions.readthedocs.io](https://jupyter-contrib-nbextensions.readthedocs.io/en/latest/install.html).

Then you can use the [jupyter\_nbextensions\_configurator](https://github.com/Jupyter-contrib/jupyter_nbextensions_configurator) to enable/disable this extension for all notebooks.

Internals

The execution start and end times are stored in the cell metadata as ISO8601 strings, for example:

{

"ExecuteTime": {

"start\_time": "2016-02-11T18:51:18.536796",

"end\_time": "2016-02-11T18:51:35.806119"

}

}

The times in the timing areas are formatted using the [moment.js](https://momentjs.com/) library (already included as part of Jupyter), but the durations use a custom formatting function, as I ([@jcb91](https://github.com/jcb91)) couldn’t find an existing one that I liked.

The event execute.CodeCell is caught in order to create a start time, and add the timing area with its ‘Execution queued at’ message. The extension again uses [moment.js](https://momentjs.com/) for formatting this as an ISO string time.

To determine the execution time, the extension patches the Jupyter class prototype CodeCell.prototype.get\_callbacks from notebook/js/codecell.js. This patch then patches the callbacks.shell.reply function returned by the original CodeCell.prototype.get\_callbacks, wrapping it in a function which reads the msg.header.date value from the kernel message, to provide the execution end time. This is more accurate than creating a new time, which can be affected by client-side variability. In addition, for accurate timings, the start time is also revised using the msg.metadata.started value supplied in the callback, which can be very different from the time the cell was queued for execution (as a result of other cells already being executed). The kernel reply message times are already ISO8601 strings, so no conversion is necessary, although again, [moment.js](https://momentjs.com/) is used for parsing and diff’ing them.

10.What is elementwise arithmetic?

So, addition is an element-wise operation, and in fact, all the arithmetic operations, add, subtract, multiply, and divide are element-wise operations.

Matrix and Element-wise Operations

Some operations are intended for matrices in particular. These include the conjugate and non-conjugate transpose operators ' and .', the matrix multiplication operator tex2html_wrap_inline2423 , and the left and right matrix ``division'' operators tex2html_wrap_inline2561 and /. For instance, if A is a matrix and x and b are vectors, then the lines

.1ex>> A'  
.1ex>> A2  
.1ex>> A \* x = b  
.1ex>> x = A tex2html_wrap_inline2561 b  
.1ex>> x \* A = b  
.1ex>> x = A/b

respectively take the conjugate transpose of A, take the square of A, give a typical matrix equation involving matrix multiplication, give the solution for that equation, give another matrix equation, and give the solution for the second equation.

Such solutions to matrix equations are solved exactly (with Gaussian elimination) if the matrix is square; others are solved in a least-squares sense (with Householder orthogonalization). Also see help slash.

To get inner and outer products of vectors, remember their formal definitions. The inner product is given by x' \* y = y' \* x, and the outer products by x \* y' and y \* x' = (x \* y')'.

MATLAB understands multiplication and division between a matrix and a scalar in the normal sense;

.1ex>> 10 \* [1 2; 3 4]  
ans =  
  
If you want to take two matrices (or vectors) and multiply or divide them element by element, or if you want to exponentiate each element of a matrix, place a period before the operator. These are array operations as opposed to matrix operations.

.1ex>> [1 2; 3 4].2  
ans =  
  
Exponentiation also has both matrix and array forms. If x and y are scalars and A and B are matrices, y tex2html_wrap_inline2591 x, A tex2html_wrap_inline2591 x, and x tex2html_wrap_inline2591 A have their usual mathematical meanings. Array exponentiation is available with A. tex2html_wrap_inline2591 x to raise each element to a power, and A. tex2html_wrap_inline2591 B to raise each element of A to the power of the corresponding element of B.

MATLAB also has a large number of matrix functions to implement common mathematical operations, such as finding eigenvalues and eigenvectors. For instance, kron will give the (Kronecker) tensor product. See help matfun.

If you apply a function that operates on scalars to a matrix or vector, or if you apply a function that operates on vectors to a matrix, MATLAB performs the operation element-wise. Scalar functions will be applied to each element of the matrix, and the result will be a matrix of the same size. Vector functions will be applied to each column of the matrix, and the result will be a row vector of the same width. (Use the transpose operators to effect row-by-row application.) See help funm if you want to use the matrix and not the array version of a function. Lastly, functions defined strictly on the real line are applied separately to the real and imaginary parts of a complex number.

.1ex>> sin([0 (pi/6) (pi/2) pi])  
ans =  
  
.1ex>> max([1 10; 20 2])  
ans =  
  
.1ex>> max(max([1 10; 20 2]))  
ans =  
  
.1ex>> round(1.7+3.2i)  
ans =  
+ 3i Certain functions are particularly useful for this. The functions max, min, median, mean, std, sum, and prod take a vector and return its maximum, minimum, median, arithmetic mean, standard deviation, element sum, and element product (respectively). Applied to a matrix, they return a row vector of the result on each column. sort sorts a vector (or each column of a matrix) in ascending order

11.Write the PyTorch code to test whether every element of a is greater than the corresponding element of b.

Working on Datasets

If you are working on a real-time project involving Deep Learning, it's common that most of your time goes into handling data, rather than the neural network that you would build. This is because data is like fuel for your network: the more appropriate it is, the faster and the more accurate the results are! One of the main reasons for your neural network to underperform might be due to bad, or poorly understood data. Hence it is important to understand, preprocess, and load your data into the network in a more intuitive way.

In many cases, we train neural networks on default or well-known datasets like MNIST or CIFAR. While working on these, we can easily achieve accuracy greater than 90% for prediction- and classification-type problems. The reason being, these datasets are neatly organized and easy to preprocess. But when you are working on a dataset of your own, it’s quite tricky and challenging to achieve high accuracy. We’ll learn about working on custom datasets in the next sections. Before that, we’ll have a quick look at the datasets that are included in the PyTorch library.

PyTorch comes with several built-in datasets, all of which are pre-loaded in the class torch.datasets. Does that ring any bells? In the previous example, when we were classifying MNIST images, we used the same class to download our images. What’s in the package torch and torchvision? The package torch consists of all the core classes and methods required to implement neural networks, while torchvision is a supporting package consisting of popular datasets, model architectures, and common image transformations for computer vision. There is one more package named torchtext which has all the basic utilities of PyTorch Natural Language Processing. This package consists of datasets that are related to text.

Here’s a quick overview of datasets that are included in the classes torchvision and torchtext.

Datasets in Torchvision

MNIST: MNIST is a dataset consisting of handwritten images that are normalized and center-cropped. It has over 60,000 training images and 10,000 test images. This is one of the most-used datasets for learning and experimenting purposes. To load and use the dataset you can import using the below syntax after the torchvision package is installed.

torchvision.datasets.MNIST()

Fashion MNIST: This dataset is similar to MNIST, but instead of handwritten digits, this dataset includes clothing items like T-shirts, trousers, bags, etc. The number of training and testing samples is 60,000 and 10,000 respectively. Below is the location of FMNIST class.

torchvision.datasets.FashionMNIST()

CIFAR: The CIFAR dataset has two versions, CIFAR10 and CIFAR100. CIFAR10 consists of images of 10 different labels, while CIFAR100 has 100 different classes. These include common images like trucks, frogs, boats, cars, deer, and others. This dataset is recommended for building CNNs.

torchvision.datasets.CIFAR10()

torchvision.datasets.CIFAR100()

COCO: This dataset consists of over 100,000 everyday objects like people, bottles, stationery, books, etc. This dataset of images is widely used for object detection and image captioning applications. Below is the location from which COCO can be loaded:

torchvision.datasets.CocoCaptions()

EMNIST: This dataset is an advanced version of the MNIST dataset. It consists of images including both numbers and alphabets. If you are working on a problem that is based on recognizing text from images, this is the right dataset to train with. Below is the class:

torchvision.datasets.EMNIST()

IMAGE-NET: ImageNet is one of the flagship datasets that is used to train high-end neural networks. It consists of over 1.2 million images spread across 10,000 classes. Usually, this dataset is loaded on a high-end hardware system as a CPU alone cannot handle datasets this big in size. Below is the class to load the ImageNet dataset:

torchvision.datasets.ImageNet()

These are a few datasets that are the most frequently used while building neural networks in PyTorch. A few others include KMNIST, QMNIST, LSUN, STL10, SVHN, PhotoTour, SBU, Cityscapes, SBD, USPS, Kinetics-400. You can learn more about these from the [PyTorch official documentation](https://pytorch.org/docs/stable/torchvision/datasets.html).

Datasets in Torchtext

As discussed previously, torchtext is a supporting package that consists of all the basic utilities for Natural Language Processing. If you are new to NLP, it is a subfield of Artificial Intelligence that processes and analyzes large amounts of natural language data (mostly relating to text).

Now let's take a look at a few popular text datasets to experiment and work with.

IMDB: This is a dataset for sentiment classification that contains a set of 25,000 highly polar movie reviews for training, and another 25,000 for testing. We can load this data by using the following class from torchtext:

torchtext.datasets.IMDB()

WikiText2: This language modelling dataset is a collection of over 100 million tokens. It is extracted from Wikipedia and retains the punctuation and the actual letter case. It is widely used in applications that involve long-term dependencies. This data can be loaded from torchtext as follows:

torchtext.datasets.WikiText2()

Besides the above two popular datasets, there are still many more available in the torchtext library, such as SST, TREC, SNLI, MultiNLI, WikiText-2, WikiText103, PennTreebank, Multi30k, etc.

So far, we’ve seen datasets that are based on a predefined set of images and text. But what if you have your own? How do you load it? For now let's learn the ImageFolder class, which you can use to load your own image datasets.

ImageFolder Class

ImageFolder is a generic data loader class in torchvision that helps you load your own image dataset. Let’s imagine you are working on a classification problem and building a neural network to identify if a given image is an apple or an orange. To do this in PyTorch, the first step is to arrange images in a default folder structure as shown below:

root

├── orange

│ ├── orange\_image1.png

│ └── orange\_image1.png

├── apple

│ └── apple\_image1.png

│ └── apple\_image2.png

│ └── apple\_image3.png

After you arrange your dataset as shown, you can use the ImageLoader class to load all these images. Below is the code snippet you would use to do so:

torchvision.datasets.ImageFolder(root, transform)

In the next section, let’s see how to load data into our programs.

Data Loading in PyTorch

Data loading is one of the first steps in building a Deep Learning pipeline, or training a model. This task becomes more challenging when the complexity of the data increases. In this section, we will learn about the DataLoader class in PyTorch that helps us to load and iterate over elements in a dataset. This class is available as DataLoader in the torch.utils.data module. DataLoader can be imported as follows:

from torch.utils.data import DataLoader

Let’s now discuss in detail the parameters that the DataLoader class accepts, shown below.

from torch.utils.data import DataLoader

DataLoader(

dataset,

batch\_size=1,

shuffle=False,

num\_workers=0,

collate\_fn=None,

pin\_memory=False,

)

COPY

1. Dataset: The first parameter in the DataLoader class is the dataset. This is where we load the data from.

2. Batching the data: batch\_size refers to the number of training samples used in one iteration. Usually we split our data into training and testing sets, and we may have different batch sizes for each.

3. Shuffling the data: shuffle is another argument passed to the DataLoader class. The argument takes in a Boolean value (True/False). If shuffle is set to True, then all the samples are shuffled and loaded in batches. Otherwise they are sent one-by-one without any shuffling.

4. Allowing multi-processing: As deep learning involves training models with a lot of data, running only single processes ends up taking a lot of time. In PyTorch, you can increase the number of processes running simultaneously by allowing multiprocessing with the argument num\_workers. This also depends on the batch size, but I wouldn’t set num\_workers to the same number because each worker loads a single batch, and returns it only once it’s ready.

num\_workers=0 means that it’s the main process that does the data loading when needed.

num\_workers=1 means you only have a single worker, so it might be slow.

5. Merging datasets: The collate\_fn argument is used if we want to merge datasets. This argument is optional, and mostly used when batches are loaded from map-styled datasets.

6. Loading data on CUDA tensors: You can directly load datasets as CUDA tensors using the pin\_memory argument. It is an optional parameter that takes in a Boolean value; if set to True, the DataLoader class copies Tensors into CUDA-pinned memory before returning them.

12.What is a rank-0 tensor? How do you convert it to a plain Python data type?

The tensor() method

This method returns a tensor when data is passed to it. data can be a scalar, tuple, a list or a NumPy array.

In the above example, a NumPy array that was created using np.arange() was passed to the tensor() method, resulting in a 1-D tensor.

We can create a multi-dimensional tensor by passing a tuple of tuples, a list of lists, or a multi-dimensional NumPy array.

When an empty tuple or list is passed into tensor(), it creates an empty tensor.

The zeros() method

This method returns a tensor where all elements are zeros, of specified size (shape). The size can be given as a tuple or a list or neither.

We could have passed 3, 2 inside a tuple or a list as well. It is self-explainable that passing negative numbers or a float would result in a run time error.

Passing an empty tuple or an empty list gives a tensor of size (dimension) 0, having 0 as its only element, whose data type is float.

The ones() method

Similar to zeros(), ones() returns a tensor where all elements are 1, of specified size (shape). The size can be given as a tuple or a list or neither.

Like zeros(), passing an empty tuple or list gives a tensor of 0 dimension, having 1 as the sole element, whose data type is float.

The full() method

What if you want all the elements of a tensor to be equal to some value but not only 0 and 1? Maybe 2.9?

full() returns a tensor of a shape given by the size argument, with all its elements equal to the fill\_value.

Here, we have created a tensor of shape 3, 2 with the fill\_value as 3. Here again, passing an empty tuple or list creates a scalar tensor of zero dimension.

While using full, it is necessary to give size as a tuple or a list.

The arange() method

This method returns a 1-D tensor, with elements from start (inclusive) to end (exclusive) with a common difference step. The default value for start is 0 while that for step is 1.

The elements of the tensor can be said to be in Arithmetic Progression, with step as common difference.

Here, we created a tensor which starts from 2 and goes until 20 with a step (common difference) of 2.

All the three parameters, start, end and step can be positive, negative or float.

While choosing start, end, and step, we need to ensure that start and end are consistent with the step sign.

Since step is set as -2, there is no way -42 can reach -22 (exclusive). Hence, it gives an error.

The linspace() method

This method returns a 1-D dimensional tensor, with elements from start (inclusive) to end (inclusive). However, unlike arange(), here, steps isn't the common difference but the number of elements to be in the tensor.

PyTorch automatically decides the common difference based on the steps given.

Not providing a value for steps is deprecated. For backwards compatibility, not providing a value for steps creates a tensor with 100 elements. According to the official documentation, in a future PyTorch release, failing to provide a value for steps will throw a runtime error.

Unlike arange(), linspace can have a start greater than end since the common difference is automatically calculated.

Since steps here is not a common difference, but the number of elements, it can only be a non-negative integer.

The rand() method

This method returns a tensor filled with random numbers from a uniform distribution on the interval 0 (inclusive) to 1 (exclusive). The shape is given by the size argument. The size argument can be given as a tuple or list or neither.

Passing an empty tuple or list creates a scalar tensor of zero dimension.

The randint() method

This method returns a tensor filled with random integers generated uniformly between low (inclusive) and high (exclusive). The shape is given by the size argument. The default value for low is 0.

When only one int argument is passed, low gets the value 0, by default, and high gets the passed value.

The size argument only takes a tuple or a list. An empty tuple or list creates a tensor with zero dimension.

The eye() method

This method returns a 2-D tensor with ones on the diagonal and zeros elsewhere. The number of rows is given by n and columns is given by m.

The default value for m is the value of n. When only n is passed, it creates a tensor in the form of an identity matrix. An identity matrix has its diagonal elements as 1 and all others as 0.

The complex() method

This method returns a complex tensor with its real part equal to real and its imaginary part equal to imag. Both real and imag are tensors.

The data type of both the real and imag tensors should be either float or double.

Also, the size of both tensors, real and imag, should be the same, since the corresponding elements of the two matrices form a complex number.

13.How does elementwise arithmetic help us speed up matmul?

1. NumPy Matrix Multiplication Element Wise

If you want element-wise matrix multiplication, you can use multiply() function.

import numpy as np

arr1 = np.array([[1, 2],

[3, 4]])

arr2 = np.array([[5, 6],

[7, 8]])

arr\_result = np.multiply(arr1, arr2)

print(arr\_result)

Output:

[[ 5 12]

[21 32]]

The below image shows the multiplication operation performed to get the result matrix.

Numpy Matrix multiply()

2. Matrix Product of Two NumPy Arrays

If you want the matrix product of two arrays, use matmul() function.

import numpy as np

arr1 = np.array([[1, 2],

[3, 4]])

arr2 = np.array([[5, 6],

[7, 8]])

arr\_result = np.matmul(arr1, arr2)

print(f'Matrix Product of arr1 and arr2 is:\n{arr\_result}')

arr\_result = np.matmul(arr2, arr1)

print(f'Matrix Product of arr2 and arr1 is:\n{arr\_result}')

Output:

Matrix Product of arr1 and arr2 is:

[[19 22]

[43 50]]

Matrix Product of arr2 and arr1 is:

[[23 34]

[31 46]]

The below diagram explains the matrix product operations for every index in the result array. For simplicity, take the row from the first array and the column from the second array for each index. Then multiply the corresponding elements and then add them to reach the matrix product value.

Numpy Matrix Product

The matrix product of two arrays depends on the argument position. So matmul(A, B) might be different from matmul(B, A).

3. Dot Product of Two NumPy Arrays

The numpy dot() function returns the dot product of two arrays. The result is the same as the matmul() function for one-dimensional and two-dimensional arrays.

import numpy as np

arr1 = np.array([[1, 2],

[3, 4]])

arr2 = np.array([[5, 6],

[7, 8]])

arr\_result = np.dot(arr1, arr2)

print(f'Dot Product of arr1 and arr2 is:\n{arr\_result}')

arr\_result = np.dot(arr2, arr1)

print(f'Dot Product of arr2 and arr1 is:\n{arr\_result}')

arr\_result = np.dot([1, 2], [5, 6])

print(f'Dot Product of two 1-D arrays is:\n{arr\_result}')

Output:

Dot Product of arr1 and arr2 is:

[[19 22]

[43 50]]

Dot Product of arr2 and arr1 is:

[[23 34]

[31 46]]

Dot Product of two 1-D arrays is:

17

14.What are the broadcasting rules?

Broadcasting Rules:

If the arrays don't have the same rank then prepend the shape of the lower rank array with 1s until both shapes have the same length. The two arrays are compatible in a dimension if they have the same size in the dimension or if one of the arrays has size 1 in that dimension.

15.What is expand\_as? Show an example of how it can be used to match the results of broadcasting.

Outside radio broadcasts have been taking place since the early 1920s[[1]](https://en.wikipedia.org/wiki/Outside_broadcasting#cite_note-1) and television ones since the late 1920s.[[2]](https://en.wikipedia.org/wiki/Outside_broadcasting#cite_note-2) The first large-scale outside broadcast was the televising of the [Coronation of George VI and Elizabeth](https://en.wikipedia.org/wiki/Coronation_of_George_VI_and_Elizabeth) in May 1937, done by the [BBC](https://en.wikipedia.org/wiki/BBC)'s first Outside Broadcast truck, MCR 1 (short for Mobile Control Room).[[3]](https://en.wikipedia.org/wiki/Outside_broadcasting#cite_note-3)

After the Second World War, the first notable outside broadcast was of the [1948 Summer Olympics](https://en.wikipedia.org/wiki/1948_Summer_Olympics).[[4]](https://en.wikipedia.org/wiki/Outside_broadcasting#cite_note-mcrbw-4)[[5]](https://en.wikipedia.org/wiki/Outside_broadcasting#cite_note-5) The [Coronation of Elizabeth II](https://en.wikipedia.org/wiki/Coronation_of_Elizabeth_II) followed in 1953, with 21 cameras being used to cover the event.[[6]](https://en.wikipedia.org/wiki/Outside_broadcasting#cite_note-6)[[7]](https://en.wikipedia.org/wiki/Outside_broadcasting#cite_note-7)

[Television South (TVS)](https://en.wikipedia.org/wiki/Television_South) OB Unit 1 as seen in 1991

In December 1963 [instant replays](https://en.wikipedia.org/wiki/Instant_replays) were used for the first time. Director [Tony Verna](https://en.wikipedia.org/wiki/Tony_Verna) used the technique on the [Army-Navy game](https://en.wikipedia.org/wiki/Army-Navy_game) which aired on [CBS Sports](https://en.wikipedia.org/wiki/CBS_Sports) on December 7, 1963.[[8]](https://en.wikipedia.org/wiki/Outside_broadcasting#cite_note-8)

The [1968 Summer Olympics](https://en.wikipedia.org/wiki/1968_Summer_Olympics) was the first with competitions televised in colour.[[9]](https://en.wikipedia.org/wiki/Outside_broadcasting#cite_note-9) The [1972 Olympic Games](https://en.wikipedia.org/wiki/1972_Summer_Olympics) were the first where all competitions were captured by outside broadcast cameras.[[10]](https://en.wikipedia.org/wiki/Outside_broadcasting#cite_note-72olympics-10)[[11]](https://en.wikipedia.org/wiki/Outside_broadcasting#cite_note-11)

During the 1970s, ITV franchise holder [Southern Television](https://en.wikipedia.org/wiki/Southern_Television) was unique in having an outside broadcast boat, named Southener.[[12]](https://en.wikipedia.org/wiki/Outside_broadcasting#cite_note-12)

The [wedding of Prince Charles and Lady Diana Spencer](https://en.wikipedia.org/wiki/Wedding_of_Prince_Charles_and_Lady_Diana_Spencer) in July 1981 was the biggest outside broadcast at the time, with an estimated 750 million viewers.[[13]](https://en.wikipedia.org/wiki/Outside_broadcasting#cite_note-bbcCharlesDiana-13)

New technology[[edit](https://en.wikipedia.org/w/index.php?title=Outside_broadcasting&action=edit&section=2)]

In 2008, the first [3D](https://en.wikipedia.org/wiki/3D_television) outside broadcast took place with the transmission of a [Calcutta Cup](https://en.wikipedia.org/wiki/Calcutta_Cup) rugby match, but only to an audience of industry professionals who had been invited by [BBC Sport](https://en.wikipedia.org/wiki/BBC_Sport).[[14]](https://en.wikipedia.org/wiki/Outside_broadcasting#cite_note-14)

In March 2010, the first public 3D outside broadcast took place with a [NHL](https://en.wikipedia.org/wiki/National_Hockey_League) game between the [New York Rangers](https://en.wikipedia.org/wiki/New_York_Rangers) and [New York Islanders](https://en.wikipedia.org/wiki/New_York_Islanders).[[15]](https://en.wikipedia.org/wiki/Outside_broadcasting#cite_note-15)

The first commercial [ultra-high definition](https://en.wikipedia.org/wiki/Ultra-high-definition_television) outside broadcast was a [Premier League](https://en.wikipedia.org/wiki/Premier_League) game between [Stoke City](https://en.wikipedia.org/wiki/Stoke_City_F.C.) v [West Ham](https://en.wikipedia.org/wiki/West_Ham_United_F.C.), televised by [Sky Sports](https://en.wikipedia.org/wiki/Sky_Sports) in August 2013.[[16]](https://en.wikipedia.org/wiki/Outside_broadcasting#cite_note-SkyUHD-16)

Tests in [8K resolution](https://en.wikipedia.org/wiki/8K_resolution) outside broadcasts began to take place during the 2010s, including tests by [NHK](https://en.wikipedia.org/wiki/NHK)[[17]](https://en.wikipedia.org/wiki/Outside_broadcasting#cite_note-17) and [BT Sport](https://en.wikipedia.org/wiki/BT_Sport).[[18]](https://en.wikipedia.org/wiki/Outside_broadcasting#cite_note-18) The first public 8K outside broadcast in the UK took place in February 2020.[[19]](https://en.wikipedia.org/wiki/Outside_broadcasting#cite_note-19)[[20]](https://en.wikipedia.org/wiki/Outside_broadcasting#cite_note-20)

Modern applications[[edit](https://en.wikipedia.org/w/index.php?title=Outside_broadcasting&action=edit&section=3)]

Main article: [Production truck](https://en.wikipedia.org/wiki/Production_truck)

Modern outside broadcasts now use specially designed OB vehicles, many of which are now built based around [IP](https://en.wikipedia.org/wiki/Internet_Protocol) technology rather than relying on [coaxial cable](https://en.wikipedia.org/wiki/Coaxial_cable).[[21]](https://en.wikipedia.org/wiki/Outside_broadcasting#cite_note-21)

There has been an increasing rise in the use of flyaway or flypack Portable Production Units, which allow for an increased level of customisation and can be rigged in a larger variety of venues.[[22]](https://en.wikipedia.org/wiki/Outside_broadcasting#cite_note-22)

In the past many outside broadcasting applications have relied on using [satellite](https://en.wikipedia.org/wiki/Satellite) uplinks to broadcast live audio and video back to the studio. While this has its advantages such as the ability to set up anywhere covered by the respective geostationary satellite, satellite uplinking is relatively expensive and the round trip [latency](https://en.wikipedia.org/wiki/Latency_(engineering)) is in the range of 240 to 280 milliseconds.[[23]](https://en.wikipedia.org/wiki/Outside_broadcasting#cite_note-23)

As more venues install [fiber optic cable](https://en.wikipedia.org/wiki/Optical_fiber_cable" \o "Optical fiber cable), this is increasingly used.[[24]](https://en.wikipedia.org/wiki/Outside_broadcasting#cite_note-24) For news gathering, contribution over public internet is also now used. Modern applications such as hardware and software IP codecs have allowed the use of public 3G/4G networks to broadcast video and audio. The latency of 3G is around 100–500 ms, while 4G is less than 100 ms.[[](https://en.wikipedia.org/wiki/Outside_broadcasting#cite_note-25)