1. **Python NumPy Tutorial: Learn with Example**

**What is NumPy?**

NumPy is an open source library available in Python that aids in mathematical, scientific, engineering, and data science programming. NumPy is an incredible library to perform mathematical and statistical operations. It works perfectly well for multi-dimensional arrays and matrices multiplication

For any scientific project, NumPy is the tool to know. It has been built to work with the N-dimensional array, linear algebra, random number, Fourier transform, etc. It can be integrated to C/C++ and Fortran.

NumPy is a programming language that deals with multi-dimensional arrays and matrices. On top of the arrays and matrices, NumPy supports a large number of mathematical operations. In this part, we will review the essential functions that you need to know for the tutorial on 'TensorFlow.'

**Why use NumPy?**

NumPy is memory efficiency, meaning it can handle the vast amount of data more accessible than any other library. Besides, NumPy is very convenient to work with, especially for matrix multiplication and reshaping. On top of that, NumPy is fast. In fact, TensorFlow and Scikit learn to use NumPy array to compute the matrix multiplication in the back end.

# How to Install NumPy

## How to Install NumPy?

To install NumPy library, please refer our tutorial [How to install TensorFlow](https://www.guru99.com/download-install-tensorflow.html). NumPy is installed by default with Anaconda.

In remote case, NumPy not installed-

You can install NumPy using Anaconda:

conda install -c anaconda numpy

* In Jupyter Notebook :

import sys

!conda install --yes --prefix {sys.prefix} numpy

### Import NumPy and Check Version

The command to import numpy is

import numpy as np

Above code renames the Numpy namespace to np. This permits us to prefix Numpy function, methods, and attributes with " np " instead of typing " numpy." It is the standard shortcut you will find in the numpy literature

To check your installed version of Numpy use the command

print (np.\_\_version\_\_)

**Output**

1.18.0

# Python Numpy Array Tutorial

### What is Python Numpy Array?

NumPy arrays are a bit like Python lists, but still very much different at the same time. For those of you who are new to the topic, let’s clarify what it exactly is and what it’s good for.

As the name kind of gives away, a NumPy array is a central data structure of the numpy library. The library’s name is actually short for "Numeric Python" or "Numerical Python".

### Create a NumPy Array

Simplest way to create an array in Numpy is to use Python List

myPythonList = [1,9,8,3]

To convert python list to a numpy array by using the object np.array.

numpy\_array\_from\_list = np.array(myPythonList)

To display the contents of the list

numpy\_array\_from\_list

**Output**

array([1, 9, 8, 3])

In practice, there is no need to declare a Python List. The operation can be combined.

a = np.array([1,9,8,3])

**NOTE**: Numpy documentation states use of np.ndarray to create an array. However, this the recommended method

You can also create a numpy array from a Tuple

## Mathematical Operations on an Array

You could perform mathematical operations like additions, subtraction, division and multiplication on an array. The syntax is the array name followed by the operation (+.-,\*,/) followed by the operand

Example:

numpy\_array\_from\_list + 10

**Output:**

array([11, 19, 18, 13])

This operation adds 10 to each element of the numpy array.

## Shape of Array

You can check the shape of the array with the object shape preceded by the name of the array. In the same way, you can check the type with dtypes.

import numpy as np

a = np.array([1,2,3])

print(a.shape)

print(a.dtype)

(3,)

int64

An integer is a value without decimal. If you create an array with decimal, then the type will change to float.

#### Different type

b = np.array([1.1,2.0,3.2])

print(b.dtype)

float64

### 2 Dimension Array

You can add a dimension with a ","coma

Note that it has to be within the bracket []

### 2 dimension

c = np.array([(1,2,3),

(4,5,6)])

print(c.shape)

(2, 3)

### 3 Dimension Array

Higher dimension can be constructed as follow:

### 3 dimension

d = np.array([

[[1, 2,3],

[4, 5, 6]],

[[7, 8,9],

[10, 11, 12]]

])

print(d.shape)

(2, 2, 3)

### Summary

Below, a summary of the essential functions used with NumPy.

|  |  |
| --- | --- |
| **Objective** | **Code** |
| Create array | array([1,2,3]) |
| print the shape | array([.]).shape |

# numpy.zeros() and numpy.ones() in Python

#### What is np.zeros and np.ones?

You can create a matrix full of zeroes or ones using np.zeros and np.one commands respectively. It can be used when you initialized the weights during the first iteration in TensorFlow and other statistic tasks.

The syntax is

**Numpy Zero**

numpy.zeros(shape, dtype=float, order='C')

**Numpy Once**

numpy.ones(shape, dtype=float, order='C')

Here,

**Shape**: is the shape of the array

**Dtype**: is the datatype. It is optional. The default value is float64

**Order**: Default is C which is an essential row style.

**Example numpy zero**

import numpy as np

np.zeros((2,2))

Output:

array([[0., 0.],

[0., 0.]])

**Example numpy zero with datatype**

import numpy as np

np.zeros((2,2), dtype=np.int16)

Output:

array([[0, 0],

[0, 0]], dtype=int16)

**Example numpy one 2D Array with datatype**

import numpy as np

np.ones((1,2,3), dtype=np.int16)

array([[[1, 1, 1],

[1, 1, 1]]], dtype=int16)

# numpy.reshape() and numpy.flatten() in Python

#### Reshape Data

In some occasions, you need to reshape the data from wide to long. You can use the reshape function for this. The syntax is

numpy.reshape(a, newShape, order='C')

Here,

**a**: Array that you want to reshape

**newShape**: The new desires shape

**Order**: Default is C which is an essential row style.

**Exampe of Reshape**

import numpy as np

e = np.array([(1,2,3), (4,5,6)])

print(e)

e.reshape(3,2)

Output:

// Before reshape

[[1 2 3]

[4 5 6]]

//After Reshape

array([[1, 2],

[3, 4],

[5, 6]])

## Flatten Data

When you deal with some neural network like convnet, you need to flatten the array. You can use flatten(). The syntax is

numpy.flatten(order='C')

Here,

**Order**: Default is C which is an essential row style.

**Exampe of Flatten**

e.flatten()

Output:

array([1, 2, 3, 4, 5, 6])

# numpy.hstack() and numpy.vstack() in Python with Example

**What is hstack?**

With hstack you can appened data horizontally. This is a very convinient function in Numpy. Lets study it with an example:

## Horitzontal Stack

import numpy as np

f = np.array([1,2,3])

g = np.array([4,5,6])

print('Horizontal Append:', np.hstack((f, g)))

Output:

Horizontal Append: [1 2 3 4 5 6]

**What is vstack?**

With vstack you can appened data vertically. Lets study it with an example:

## Vertical Stack

import numpy as np

f = np.array([1,2,3])

g = np.array([4,5,6])

print('Vertical Append:', np.vstack((f, g)))

Output:

Vertical Append: [[1 2 3]

[4 5 6]]

**Generate Random Numbers**

To generate random numbers for Gaussian distribution use

numpy.random.normal(loc, scale, size)

Here

* Loc: the mean. The center of distribution
* scale: standard deviation.
* Size: number of returns

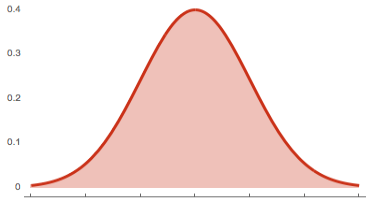
## Generate random nmber from normal distribution

normal\_array = np.random.normal(5, 0.5, 10)

print(normal\_array)

[5.56171852 4.84233558 4.65392767 4.946659 4.85165567 5.61211317 4.46704244 5.22675736 4.49888936 4.68731125]

If plotted the distribution will be similar to following plot



**Summary**

Below, a summary of the essential functions used with NumPy

|  |  |
| --- | --- |
| **Objective** | **Code** |
| append vertically | vstack |
| append horizontally | hstack |

# numpy.asarray() in Python with Example

### Asarray

The asarray()function is used when you want to convert an input to an array. The input could be a lists, tuple, ndarray, etc.

**Syntax:**

numpy.asarray(data, dtype=None, order=None)[source]

Here,

**data**: Data that you want to convert to an array

**dtype**: This is an optional argument. If not specified, the data type is inferred from the input data

**Order**: Default is C which is an essential row style. Other option is F (Fortan-style)

**Example:**

Consider the following 2-D matrix with four rows and four columns filled by 1

import numpy as np

A = np.matrix(np.ones((4,4)))

If you want to change the value of the matrix, you cannot. The reason is, it is not possible to change a copy.

np.array(A)[2]=2

print(A)

[[1. 1. 1. 1.]

[1. 1. 1. 1.]

[1. 1. 1. 1.]

[1. 1. 1. 1.]]

Matrix is immutable. You can use asarray if you want to add modification in the original array. Let's see if any change occurs when you want to change the value of the third rows with the value 2

np.asarray(A)[2]=2

print(A)

**Code Explanation:**

np.asarray(A): converts the matrix A to an array

[2]: select the third rows

**Output:**

[[1. 1. 1. 1.]

[1. 1. 1. 1.]

[2. 2. 2. 2.] # new value

[1. 1. 1. 1.]]

# numpy.arange() in Python with Example

**Whay is Arrange?**

Sometimes, you want to create values that are evenly spaced within a defined interval. For instance, you want to create values from 1 to 10; you can use numpy.arange() function

Syntax:

numpy.arange(start, stop,step)

* Start: Start of interval
* Stop: End of interval
* Step: Spacing between values. Default step is 1

**Example:**

import numpy np

np.arange(1, 11)

Output:

array([ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

**Example:**

If you want to change the step, you can add a third number in the parenthesis. It will change the step.

import numpy np

np.arange(1, 14, 4)

Output:

array([ 1, 5, 9, 13])

# numpy.linspace() and numpy.logspace() in Python with Example

## Linspace

Linspace gives evenly spaced samples.

Syntax:

numpy.linspace(start, stop, num, endpoint)

Here,

* Start: Starting value of the sequence
* Stop: End value of the sequence
* Num: Number of samples to generate. Default is 50
* Endpoint: If True (default), stop is the last value. If False, stop value is not included.

**Example:**

For instance, it can be used to create 10 values from 1 to 5 evenly spaced.

import numpy as np

np.linspace(1.0, 5.0, num=10)

Output:

array([1. , 1.44444444, 1.88888889, 2.33333333, 2.77777778, 3.22222222, 3.66666667, 4.11111111, 4.55555556, 5. ])

If you do not want to include the last digit in the interval, you can set endpoint to false

np.linspace(1.0, 5.0, num=5, endpoint=False)

Output:

array([1. , 1.8, 2.6, 3.4, 4.2])

### LogSpace

LogSpace returns even spaced numbers on a log scale. Logspace has the same parameters as np.linspace.

Syntax:

numpy.logspace(start, stop, num, endpoint)

**Example:**

np.logspace(3.0, 4.0, num=4)

Output:

array([ 1000. , 2154.43469003, 4641.58883361, 10000. ])

Finaly, if you want to check the size of an array, you can use itemsize

x = np.array([1,2,3], dtype=np.complex128)

x.itemsize

Output:

16

The x element has 16 bytes.

## Summary

Below, a summary of the essential functions used with NumPy

|  |  |
| --- | --- |
| **Objective** | **Code** |
| Create a linear space | linspace |
| Create a log space | logspace |

# Indexing and Slicing NumPy Arrays in Python with Example

### Indexing and slicing

Slicing data is trivial with numpy. We will slice the matrice "e". Note that, in Python, you need to use the brackets to return the rows or columns

## Slice

import numpy as np

e = np.array([(1,2,3), (4,5,6)])

print(e)

[[1 2 3]

[4 5 6]]

Remember with numpy the first array/column starts at 0.

## First column

print('First row:', e[0])

## Second col

print('Second row:', e[1])

Output:

First row: [1 2 3]

Second row: [4 5 6]

In Python, like many other languages,

* The values before the comma stand for the rows
* The value on the rights stands for the columns.
* If you want to select a column, you need to add : before the column index.
* : means you want all the rows from the selected column.

print('Second column:', e[:,1])

Second column: [2 5]

To return the first two values of the second row. You use : to select all columns up to the second

## Second Row, two values

print(e[1, :2])

[4 5]

# NumPy Statistical Functions with Example

1. NumPy has quite a few useful statistical functions for finding minimum, maximum, percentile standard deviation and variance, etc from the given elements in the array. The functions are explained as follows −

### Statistical function

1. Numpy is equipped with the robust statistical function as listed below

|  |  |
| --- | --- |
| Function | Numpy |
| Min | np.min() |
| Max | np.max() |
| Mean | np.mean() |
| Median | np.median() |
| Standard deviation | np.std() |

1. Consider the following Array
2. import numpy as np
3. normal\_array = np.random.normal(5, 0.5, 10)
4. print(normal\_array)
5. Output:
6. [5.56171852 4.84233558 4.65392767 4.946659 4.85165567 5.61211317 4.46704244 5.22675736 4.49888936 4.68731125]
7. **Example:Statistical function**
8. ### Min
9. print(np.min(normal\_array))
10. ### Max
11. print(np.max(normal\_array))
12. ### Mean
13. print(np.mean(normal\_array))
14. ### Median
15. print(np.median(normal\_array))
16. ### Sd
17. print(np.std(normal\_array))
18. Output:
19. 4.467042435266913
20. 5.612113171990201
21. 4.934841002270593
22. 4.846995625786663
23. 0.3875019367395316

# 11. numpy.dot(): Dot Product in Python using Numpy

### Dot Product

Numpy is powerful library for matrices computation. For instance, you can compute the dot product with np.dot

Syntax

numpy.dot(x, y, out=None)

Here,

**x,y**: Input arrays. x and y both should be 1-D or 2-D for the function to work

**out**: This is the output argument.For 1-D array scalar is returned. Other wise ndarray

**Example:**

## Linear algebra

### Dot product: product of two arrays

f = np.array([1,2])

g = np.array([4,5])

### 1\*4+2\*5

np.dot(f, g)

### Output

14

# 12. NumPy Matrix Multiplication with np.matmul() Example

#### Matrix Multiplication

The Numpu matmul() function is used to return the matrix product of 2 arrays. Here is how it works

1) 2-D arrays, it returns normal product

2) Dimensions > 2, the product is treated as a stack of matrix

3) 1-D array is first promoted to a matrix, and then the product is calculated

numpy.matmul(x, y, out=None)

Here,

**x,y**: Input arrays. scalars not allowed

**out**: This is optional parameter. Usually output is stored in ndarray

**Example:**

In the same way, you can compute matrices multiplication with np.matmul

### Matmul: matruc product of two arrays

h = [[1,2],[3,4]]

i = [[5,6],[7,8]]

### 1\*5+2\*7 = 19

np.matmul(h, i)

Output:

array([[19, 22],

[43, 50]])

### Determinant

Last but not least, if you need to compute the determinant, you can use np.linalg.det(). Note that numpy takes care of the dimension.

## Determinant 2\*2 matrix

### 5\*8-7\*6np.linalg.det(i)

Output:

-2.000000000000005

# 2. PyTorch Tutorial: Regression, Image Classification Example

**What is PyTorch?**

PyTorch is a Torch based machine learning library for Python. It's similar to numpy but with powerful GPU support. It was developed by Facebook's AI Research Group in 2016. PyTorch offers Dynamic Computational Graph such that you can modify the graph on the go with the help of autograd. Pytorch is also faster in some cases than other frameworks, but you will discuss this later in the other section.

**PyTorch Advantages and Weakness**

Advantages

1. Simple Library

PyTorch code is simple. It is easy to understand, and you use the library instantly. For example, take a look at the code snippet below:

class Net(torch.nn.Module):

def \_\_init\_\_(self):

super(Net, self).\_\_init\_\_()

self.layer = torch.nn.Linear(1, 1)

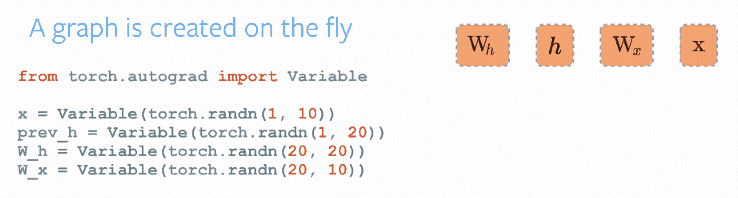
def forward(self, x):

x = self.layer(x)

return x

As above, you can define the network model easily, and you can understand the code quickly without much training.

1. Dynamic Computational Graph



[Image Source: Exploring Deep Learning with PyTorch](https://blog.algorithmia.com/exploring-the-deep-learning-framework-pytorch/)

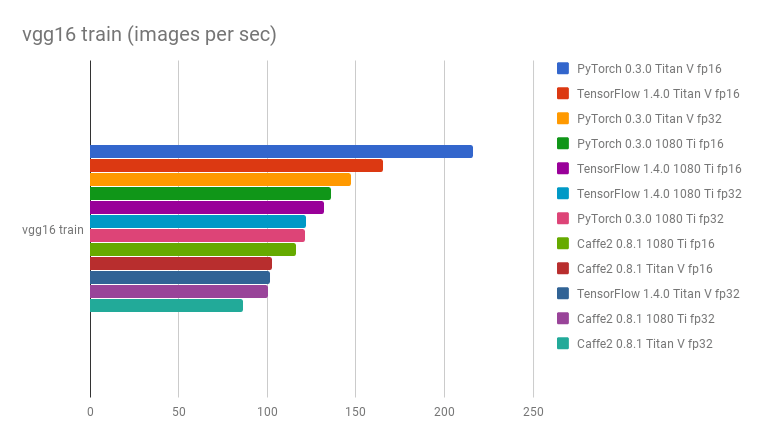
Pytorch offers Dynamic Computational Graph (DAG). Computational graphs is a way to express mathematical expressions in graph models or theories such as nodes and edges. The node will do the mathematical operation, and the edge is a Tensor that will be fed into the nodes and carries the output of the node in Tensor.

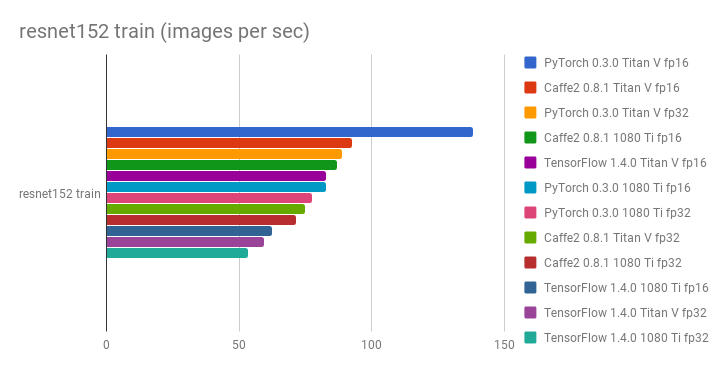
DAG is a graph that holds arbitrary shape and able to do operations between different input graphs. Every iteration, a new graph is created. So, it is possible to have the same graph structure or create a new graph with a different operation, or we can call it a dynamic graph.

1. Better Performance

Communities and researchers, benchmark and compare frameworks to see which one is faster. A GitHub repo [Benchmark on Deep Learning Frameworks and GPUs](https://github.com/u39kun/deep-learning-benchmark) reported that PyTorch is faster than the other framework in term of images processed per second.

As you can see below, the comparison graphs with vgg16 and resnet152





1. Native Python

PyTorch is more python based. For example, if you want to train a model, you can use native control flow such as looping and recursions without the need to add more special variables or sessions to be able to run them. This is very helpful for the training process.

Pytorch also implements Imperative Programming, and it's definitely more flexible. So, it's possible to print out the tensor value in the middle of a computation process.

### Weakness

PyTorch is not yet officially ready, because it is still being developed into version 1. So, further development and research is needed to achieve a stable version.

## PyTorch Vs. TensorFlow



The most popular deep learning framework is [Tensorflow](https://www.guru99.com/tensorflow-tutorial.html). Developed by Google's Brain Team, it's the foremost common deep learning tool.

## PyTorch vs. Tensorflow

|  |  |  |
| --- | --- | --- |
| Parameters | PyTorch | Tensorflow |
| Model Definition | The model is defined in a subclass and offers easy to use package | The model is defined with many, and you need to understand the syntax |
| GPU Support | Yes | Yes |
| Graph Type | Dynamic | Static |
| Tools | No visualization tool | You can use Tensorboard visualization tool |
| Community | The community still growing | Large active communities |

## Installing PyTorch

### Linux

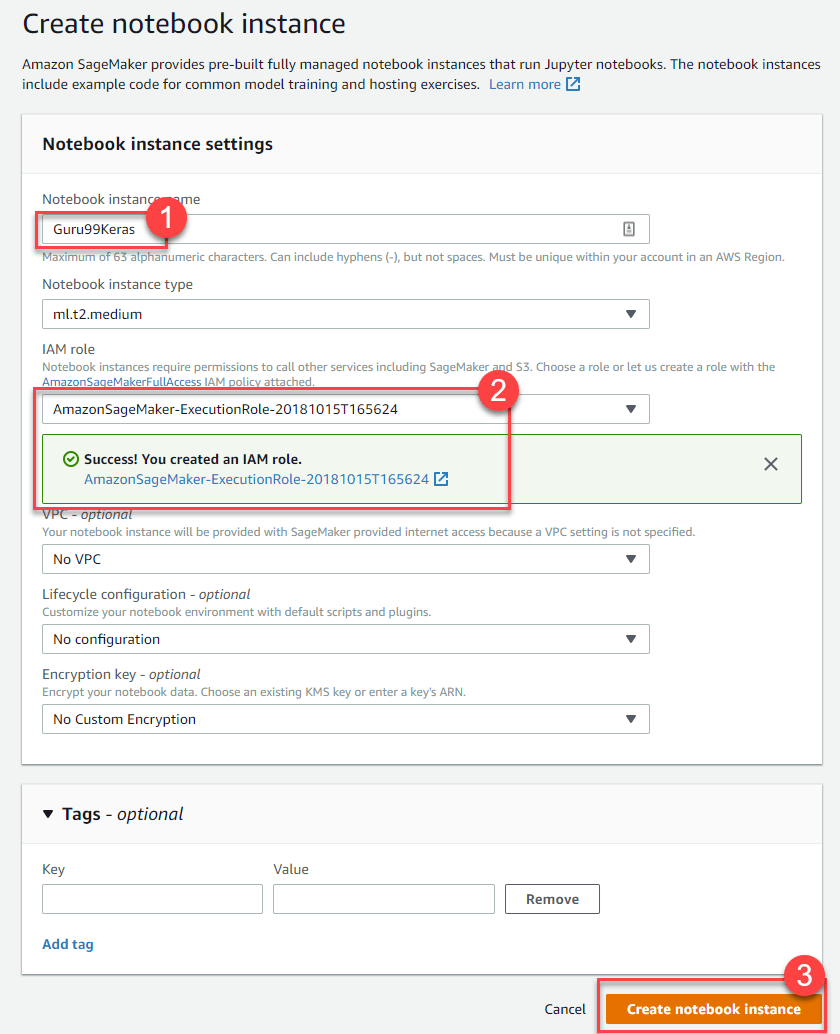
It's straightforward to install it in Linux. You can choose to use a virtual environment or install it directly with root access. Type this command in the terminal

pip3 install --upgrade torch torchvision

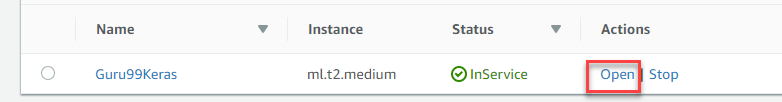
### AWS Sagemaker

Sagemaker is one of the platforms in Amazon Web Service that offers a powerful Machine Learning engine with pre-installed deep learning configurations for data scientist or developers to build, train, and deploy models at any scale.

First Open the [Amazon Sagemaker](https://console.aws.amazon.com/sagemaker/) console and click on Create notebook instance and fill all the details for your notebook.



Next Step, Click on Open to launch your notebook instance.



Finally, In Jupyter, Click on New and choose conda\_pytorch\_p36 and you are ready to use your notebook instance with Pytorch installed.

## PyTorch Framework Basics

Let's learn the basic concepts of PyTorch before we deep dive. PyTorch uses Tensor for every variable similar to numpy's ndarray but with GPU computation support. Here we will explain the network model, loss function, Backprop, and Optimizer.

### Network Model

The network can be constructed by subclassing the torch.nn. There are 2 main parts,

1. The first part is to define the parameters and layers that you will use
2. The second part is the main task called the forward process that will take an input and predict the output.

Import torch

import torch.nn as nn

import torch.nn.functional as F

class Model(nn.Module):

def \_\_init\_\_(self):

super(Model, self).\_\_init\_\_()

self.conv1 = nn.Conv2d(3, 20, 5)

self.conv2 = nn.Conv2d(20, 40, 5)

self.fc1 = nn.Linear(320, 10)

def forward(self, x):

x = F.relu(self.conv1(x))

x = F.relu(self.conv2(x))

x = x.view(-1, 320)

x = F.relu(self.fc1(x))

return F.log\_softmax(x)

net = Model()

As you can see above, you create a class of nn.Module called Model. It contains 2 Conv2d layers and a Linear layer. The first conv2d layer takes an input of 3 and the output shape of 20. The second layer will take an input of 20 and will produce an output shape of 40. The last layer is a fully connected layer in the shape of 320 and will produce an output of 10.

The forward process will take an input of X and feed it to the conv1 layer and perform ReLU function,

Similarly, it will also feed the conv2 layer. After that, the x will be reshaped into (-1, 320) and feed into the final FC layer. Before you send the output, you will use the softmax activation function.

The backward process is automatically defined by autograd, so you only need to define the forward process.

### Loss Function

The loss function is used to measure how well the prediction model is able to predict the expected results. PyTorch already has many standard loss functions in the torch.nn module. For example, you can use the Cross-Entropy Loss to solve a multi-class classification problem. It's easy to define the loss function and compute the losses:

loss\_fn = nn.CrossEntropyLoss()

#training process

loss = loss\_fn(out, target)

It's easy to use your own loss function calculation with PyTorch.

### Backprop

To perform the backpropagation, you simply call the los.backward(). The error will be computed but remember to clear the existing gradient with zero\_grad()

net.zero\_grad() # to clear the existing gradient

loss.backward() # to perform backpropragation

### Optimizer

The torch.optim provides common optimization algorithms. You can define an optimizer with a simple step:

optimizer = torch.optim.SGD(net.parameters(), lr = 0.01, momentum=0.9)

You need to pass the network model parameters and the learning rate so that at every iteration the parameters will be updated after the backprop process.

## Simple Regression with PyTorch

**Step 1)** Creating our network model

Our network model is a simple Linear layer with an input and an output shape of 1.

from \_\_future\_\_ import print\_function

import torch

import torch.nn as nn

import torch.nn.functional as F

from torch.autograd import Variable

class Net(nn.Module):

def \_\_init\_\_(self):

super(Net, self).\_\_init\_\_()

self.layer = torch.nn.Linear(1, 1)

def forward(self, x):

x = self.layer(x)

return x

net = Net()

print(net)

And the network output should be like this

Net(

(hidden): Linear(in\_features=1, out\_features=1, bias=True)

)

**Step 2)** Test Data

Before you start the training process, you need to know our data. You make a random function to test our model. Y = x3 sin(x)+ 3x+0.8 rand(100)

# Visualize our data

import matplotlib.pyplot as plt

import numpy as np

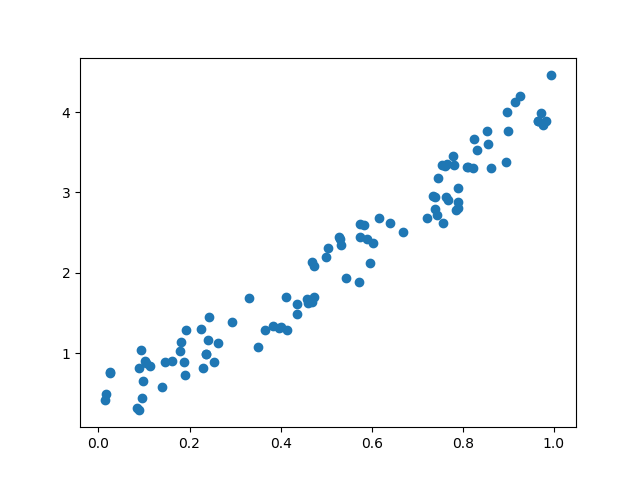
x = np.random.rand(100)

y = np.sin(x) \* np.power(x,3) + 3\*x + np.random.rand(100)\*0.8

plt.scatter(x, y)

plt.show()

Here is the scatter plot of our function:



Before you start the training process, you need to convert the numpy array to Variables that supported by Torch and autograd

# convert numpy array to tensor in shape of input size

x = torch.from\_numpy(x.reshape(-1,1)).float()

y = torch.from\_numpy(y.reshape(-1,1)).float()

print(x, y)

**Step 3)** Optimizer and Loss

Next, you should define the Optimizer and the Loss Function for our training process.

# Define Optimizer and Loss Function

optimizer = torch.optim.SGD(net.parameters(), lr=0.2)

loss\_func = torch.nn.MSELoss()

**Step 4)**Training

Now let's start our training process. With an epoch of 250, you will iterate our data to find the best value for our hyperparameters.

inputs = Variable(x)

outputs = Variable(y)

for i in range(250):

prediction = net(inputs)

loss = loss\_func(prediction, outputs)

optimizer.zero\_grad()

loss.backward()

optimizer.step()

if i % 10 == 0:

# plot and show learning process

plt.cla()

plt.scatter(x.data.numpy(), y.data.numpy())

plt.plot(x.data.numpy(), prediction.data.numpy(), 'r-', lw=2)

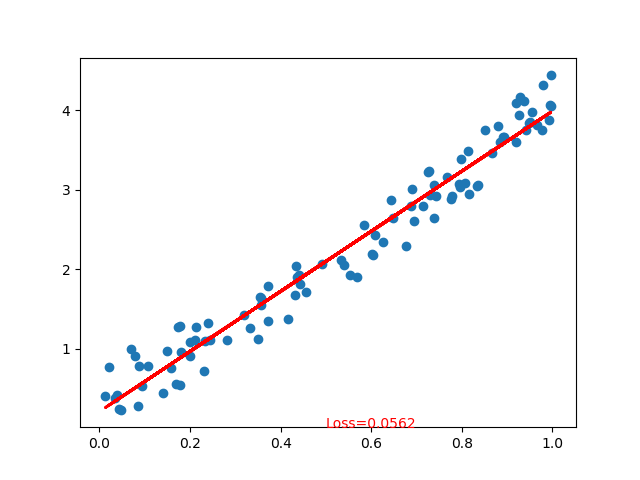
plt.text(0.5, 0, 'Loss=%.4f' % loss.data.numpy(), fontdict={'size': 10, 'color': 'red'})

plt.pause(0.1)

plt.show()

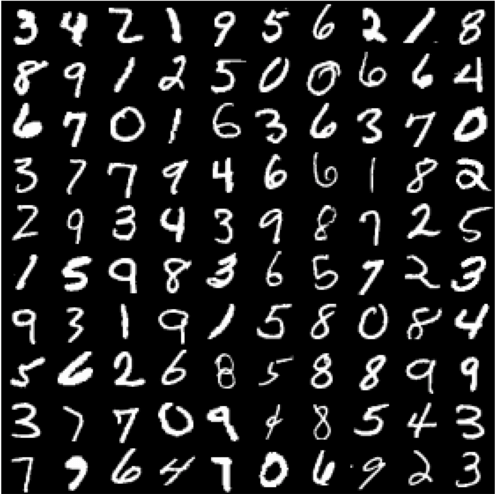
**Step 5)** Result

As you can see below, you successfully performed regression with a neural network. Actually, on every iteration, the red line in the plot will update and change its position to fit the data. But in this picture, you only show you the final result



## Image Classification with PyTorch

One of the popular methods to learn the basics of deep learning is with the MNIST dataset. It is the "Hello World" in deep learning. The dataset contains handwritten numbers from 0 - 9 with the total of 60,000 training samples and 10,000 test samples that are already labeled with the size of 28x28 pixels.



**Step 1)** Preprocess the Data

Before you start the training process, you need to understand the data. In the first step, you will load the dataset using torchvision module. Torchvision will load the dataset and transform the images with the appropriate requirement for the network such as the shape and normalizing the images.

import torch

import torchvision

import numpy as np

from torchvision import datasets, models, transforms

# This is used to transform the images to Tensor and normalize it

transform = transforms.Compose(

[transforms.ToTensor(),

transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

training = torchvision.datasets.MNIST(root='./data', train=True,

download=True, transform=transform)

train\_loader = torch.utils.data.DataLoader(training, batch\_size=4,

shuffle=True, num\_workers=2)

testing = torchvision.datasets.MNIST(root='./data', train=False,

download=True, transform=transform)

test\_loader = torch.utils.data.DataLoader(testing, batch\_size=4,

shuffle=False, num\_workers=2)

classes = ('0', '1', '2', '3',

'4', '5', '6', '7', '8', '9')

import matplotlib.pyplot as plt

import numpy as np

#create an iterator for train\_loader

# get random training images

data\_iterator = iter(train\_loader)

images, labels = data\_iterator.next()

#plot 4 images to visualize the data

rows = 2

columns = 2

fig=plt.figure()

for i in range(4):

fig.add\_subplot(rows, columns, i+1)

plt.title(classes[labels[i]])

img = images[i] / 2 + 0.5 # this is for unnormalize the image

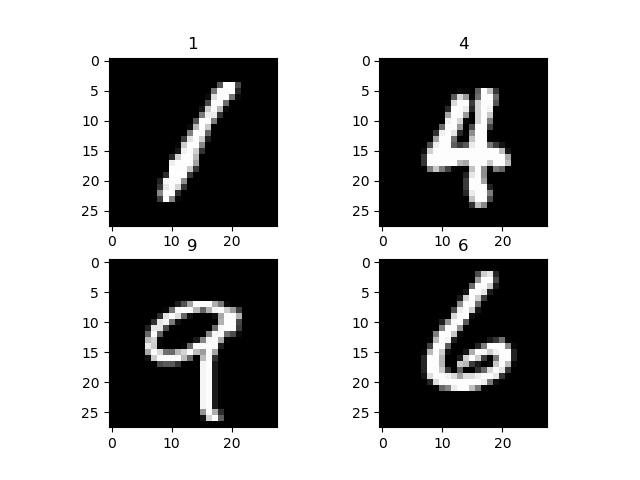
img = torchvision.transforms.ToPILImage()(img)

plt.imshow(img)

plt.show()

The transform function converts the images into tensor and normalizes the value. The function torchvision.transforms.MNIST, will download the dataset (if it's not available) in the directory, set the dataset for training if necessary and do the transformation process.

To visualize the dataset, you use the data\_iterator to get the next batch of images and labels. You use matplot to plot these images and their appropriate label. As you can see below our images and their labels.



**Step 2)** Network Model Configuration

Now you will make a simple neural network for image classification. Here, we introduce you another way to create the Network model in PyTorch. We will use nn.Sequential to make a sequence model instead of making a subclass of nn.Module.

import torch.nn as nn

# flatten the tensor into

class Flatten(nn.Module):

def forward(self, input):

return input.view(input.size(0), -1)

#sequential based model

seq\_model = nn.Sequential(

nn.Conv2d(1, 10, kernel\_size=5),

nn.MaxPool2d(2),

nn.ReLU(),

nn.Dropout2d(),

nn.Conv2d(10, 20, kernel\_size=5),

nn.MaxPool2d(2),

nn.ReLU(),

Flatten(),

nn.Linear(320, 50),

nn.ReLU(),

nn.Linear(50, 10),

nn.Softmax(),

)

net = seq\_model

print(net)

Here is the output of our network model

Sequential(

(0): Conv2d(1, 10, kernel\_size=(5, 5), stride=(1, 1))

(1): MaxPool2d(kernel\_size=2, stride=2, padding=0, dilation=1, ceil\_mode=False)

(2): ReLU()

(3): Dropout2d(p=0.5)

(4): Conv2d(10, 20, kernel\_size=(5, 5), stride=(1, 1))

(5): MaxPool2d(kernel\_size=2, stride=2, padding=0, dilation=1, ceil\_mode=False)

(6): ReLU()

(7): Flatten()

(8): Linear(in\_features=320, out\_features=50, bias=True)

(9): ReLU()

(10): Linear(in\_features=50, out\_features=10, bias=True)

(11): Softmax()

)

Network Explanation

1. The sequence is that the first layer is a Conv2D layer with an input shape of 1 and output shape of 10 with a kernel size of 5
2. Next, you have a MaxPool2D layer
3. A ReLU activation function
4. a Dropout layer to drop low probability values.
5. Then a second Conv2d with the input shape of 10 from the last layer and the output shape of 20 with a kernel size of 5
6. Next a MaxPool2d layer
7. ReLU activation function.
8. After that, you will flatten the tensor before you feed it into the Linear layer
9. Linear Layer will map our output at the second Linear layer with softmax activation function

**Step 3)**Train the Model

Before you start the training process, it is required to set up the criterion and optimizer function. For the criterion, you will use the CrossEntropyLoss. For the Optimizer, you will use the SGD with a learning rate of 0.001 and a momentum of 0.9.

import torch.optim as optim

criterion = nn.CrossEntropyLoss()

optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

The forward process will take the input shape and pass it to the first conv2d layer. Then from there, it will be feed into the maxpool2d and finally put into the ReLU activation function. The same process will occur in the second conv2d layer. After that, the input will be reshaped into (-1,320) and feed into the fc layer to predict the output.

Now, you will start the training process. You will iterate through our dataset 2 times or with an epoch of 2 and print out the current loss at every 2000 batch.

for epoch in range(2):

#set the running loss at each epoch to zero

running\_loss = 0.0

# we will enumerate the train loader with starting index of 0

# for each iteration (i) and the data (tuple of input and labels)

for i, data in enumerate(train\_loader, 0):

inputs, labels = data

# clear the gradient

optimizer.zero\_grad()

#feed the input and acquire the output from network

outputs = net(inputs)

#calculating the predicted and the expected loss

loss = criterion(outputs, labels)

#compute the gradient

loss.backward()

#update the parameters

optimizer.step()

# print statistics

running\_loss += loss.item()

if i % 1000 == 0:

print('[%d, %5d] loss: %.3f' %

(epoch + 1, i + 1, running\_loss / 1000))

running\_loss = 0.0

At each epoch, the enumerator will get the next tuple of input and corresponding labels. Before we feed the input to our network model, we need to clear the previous gradient. This is required because after the backward process (backpropagation process), the gradient will be accumulated instead of being replaced. Then, we will calculate the losses from the predicted output from the expected output. After that, we will do a backpropagation to calculate the gradient, and finally, we will update the parameters.

Here's the output of the training process

[1, 1] loss: 0.002

[1, 1001] loss: 2.302

[1, 2001] loss: 2.295

[1, 3001] loss: 2.204

[1, 4001] loss: 1.930

[1, 5001] loss: 1.791

[1, 6001] loss: 1.756

[1, 7001] loss: 1.744

[1, 8001] loss: 1.696

[1, 9001] loss: 1.650

[1, 10001] loss: 1.640

[1, 11001] loss: 1.631

[1, 12001] loss: 1.631

[1, 13001] loss: 1.624

[1, 14001] loss: 1.616

[2, 1] loss: 0.001

[2, 1001] loss: 1.604

[2, 2001] loss: 1.607

[2, 3001] loss: 1.602

[2, 4001] loss: 1.596

[2, 5001] loss: 1.608

[2, 6001] loss: 1.589

[2, 7001] loss: 1.610

[2, 8001] loss: 1.596

[2, 9001] loss: 1.598

[2, 10001] loss: 1.603

[2, 11001] loss: 1.596

[2, 12001] loss: 1.587

[2, 13001] loss: 1.596

[2, 14001] loss: 1.603

**Step 4)** Test the Model

After you train our model, you need to test or evaluate with other sets of images. We will use an iterator for the test\_loader, and it will generate a batch of images and labels that will be passed to the trained model. The predicted output will be displayed and compared with the expected output.

#make an iterator from test\_loader

#Get a batch of training images

test\_iterator = iter(test\_loader)

images, labels = test\_iterator.next()

results = net(images)

\_, predicted = torch.max(results, 1)

print('Predicted: ', ' '.join('%5s' % classes[predicted[j]] for j in range(4)))

fig2 = plt.figure()

for i in range(4):

fig2.add\_subplot(rows, columns, i+1)

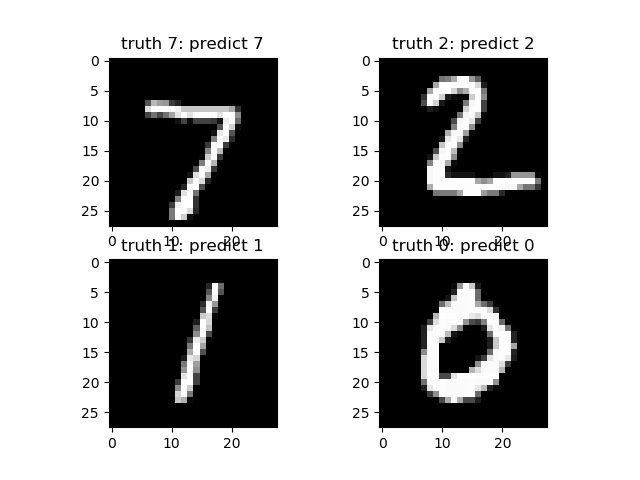
plt.title('truth ' + classes[labels[i]] + ': predict ' + classes[predicted[i]])

img = images[i] / 2 + 0.5 # this is to unnormalize the image

img = torchvision.transforms.ToPILImage()(img)

plt.imshow(img)

plt.show()

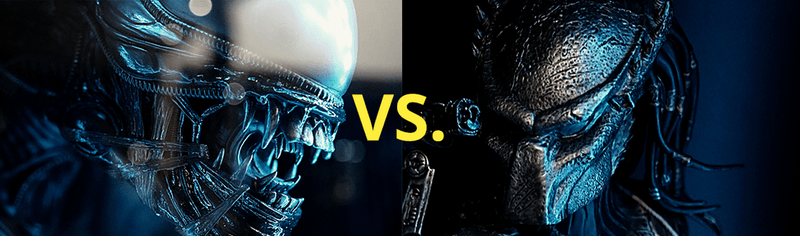


# Transfer Learning for Deep Learning with PyTorch

## What is Transfer Learning?

Transfer learning is a technique of using a trained model to solve another related task. It's popular to use other network model weight to reduce your training time because you need a lot of data to train a network model. To reduce the training time, you use other network and its weight and modify the last layer to solve our problem. The advantage is you can use a small dataset to train the last layer.

Loading Dataset



[Source: Alien vs. Predator Kaggle](https://www.kaggle.com/pmigdal/alien-vs-predator-images/home)

Before you start, you need to understand the dataset that you are going to use. In this part, you will classify an Alien and a Predator from nearly 700 images. For this technique, you don't really need a big amount of data to train. You can download the dataset from [Kaggle: Alien vs. Predator.](https://www.kaggle.com/pmigdal/alien-vs-predator-images/home)

**Step 1)** Load the Data

The first step is to load our data and do some transformation to images so that it matched the network requirements. You will load the data from a folder with torchvision.dataset. The module will iterate in the folder to split the data for train and validation. The transformation process will crop the images from the center, perform a horizontal flip, normalize, and finally convert it to tensor.

from \_\_future\_\_ import print\_function, division

import os

import time

import torch

import torchvision

from torchvision import datasets, models, transforms

import torch.optim as optim

import numpy as np

import matplotlib.pyplot as plt

data\_dir = "alien\_pred"

input\_shape = 224

mean = [0.5, 0.5, 0.5]

std = [0.5, 0.5, 0.5]

#data transformation

data\_transforms = {

'train': transforms.Compose([

transforms.CenterCrop(input\_shape),

transforms.ToTensor(),

transforms.Normalize(mean, std)

]),

'validation': transforms.Compose([

transforms.CenterCrop(input\_shape),

transforms.ToTensor(),

transforms.Normalize(mean, std)

]),

}

image\_datasets = {

x: datasets.ImageFolder(

os.path.join(data\_dir, x),

transform=data\_transforms[x]

)

for x in ['train', 'validation']

}

dataloaders = {

x: torch.utils.data.DataLoader(

image\_datasets[x], batch\_size=32,

shuffle=True, num\_workers=4

)

for x in ['train', 'validation']

}

dataset\_sizes = {x: len(image\_datasets[x]) for x in ['train', 'validation']}

print(dataset\_sizes)

class\_names = image\_datasets['train'].classes

device = torch.device("cuda:0" if torch.cuda.is\_available() else "cpu")

Let's visualize our dataset. The visualization process will get the next batch of images from the train data-loaders and labels and display it with matplot.

images, labels = next(iter(dataloaders['train']))

rows = 4

columns = 4

fig=plt.figure()

for i in range(16):

fig.add\_subplot(rows, columns, i+1)

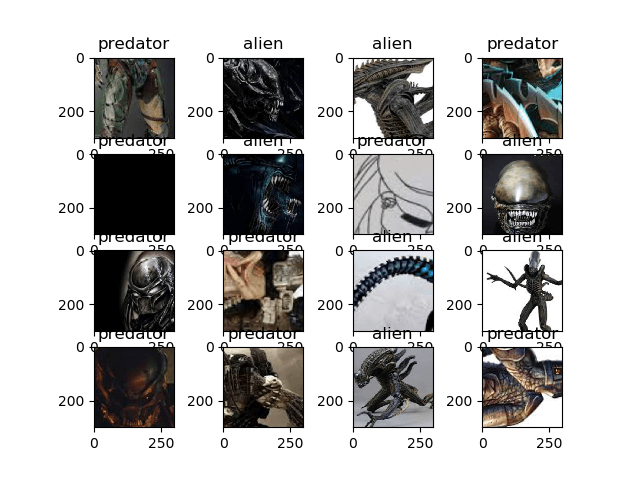
plt.title(class\_names[labels[i]])

img = images[i].numpy().transpose((1, 2, 0))

img = std \* img + mean

plt.imshow(img)

plt.show()



**Step 2)** Define Model

In this process, you will use ResNet18 from torchvision module. You will use torchvision.models to load resnet18 with the pre-trained weight set to be True. After that, you will freeze the layers so that these layers are not trainable. You also modify the last layer with a Linear layer to fit with our needs that is 2 classes. You also use CrossEntropyLoss for multi-class loss function and for the optimizer you will use SGD with the learning rate of 0.0001 and a momentum of 0.9.

## Load the model based on VGG19

vgg\_based = torchvision.models.vgg19(pretrained=True)

## freeze the layers

for param in vgg\_based.parameters():

param.requires\_grad = False

# Modify the last layer

number\_features = vgg\_based.classifier[6].in\_features

features = list(vgg\_based.classifier.children())[:-1] # Remove last layer

features.extend([torch.nn.Linear(number\_features, len(class\_names))])

vgg\_based.classifier = torch.nn.Sequential(\*features)

vgg\_based = vgg\_based.to(device)

print(vgg\_based)

criterion = torch.nn.CrossEntropyLoss()

optimizer\_ft = optim.SGD(vgg\_based.parameters(), lr=0.001, momentum=0.9)

The output model structure

VGG(

(features): Sequential(

(0): Conv2d(3, 64, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(1): ReLU(inplace)

(2): Conv2d(64, 64, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(3): ReLU(inplace)

(4): MaxPool2d(kernel\_size=2, stride=2, padding=0, dilation=1, ceil\_mode=False)

(5): Conv2d(64, 128, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(6): ReLU(inplace)

(7): Conv2d(128, 128, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(8): ReLU(inplace)

(9): MaxPool2d(kernel\_size=2, stride=2, padding=0, dilation=1, ceil\_mode=False)

(10): Conv2d(128, 256, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(11): ReLU(inplace)

(12): Conv2d(256, 256, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(13): ReLU(inplace)

(14): Conv2d(256, 256, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(15): ReLU(inplace)

(16): Conv2d(256, 256, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(17): ReLU(inplace)

(18): MaxPool2d(kernel\_size=2, stride=2, padding=0, dilation=1, ceil\_mode=False)

(19): Conv2d(256, 512, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(20): ReLU(inplace)

(21): Conv2d(512, 512, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(22): ReLU(inplace)

(23): Conv2d(512, 512, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(24): ReLU(inplace)

(25): Conv2d(512, 512, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(26): ReLU(inplace)

(27): MaxPool2d(kernel\_size=2, stride=2, padding=0, dilation=1, ceil\_mode=False)

(28): Conv2d(512, 512, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(29): ReLU(inplace)

(30): Conv2d(512, 512, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(31): ReLU(inplace)

(32): Conv2d(512, 512, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(33): ReLU(inplace)

(34): Conv2d(512, 512, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(35): ReLU(inplace)

(36): MaxPool2d(kernel\_size=2, stride=2, padding=0, dilation=1, ceil\_mode=False)

)

(classifier): Sequential(

(0): Linear(in\_features=25088, out\_features=4096, bias=True)

(1): ReLU(inplace)

(2): Dropout(p=0.5)

(3): Linear(in\_features=4096, out\_features=4096, bias=True)

(4): ReLU(inplace)

(5): Dropout(p=0.5)

(6): Linear(in\_features=4096, out\_features=2, bias=True)

)

)

**Step 3)** Train and Test Model

We will use some of the function from PyTorch Tutorial to help us train and evaluate our model.

def train\_model(model, criterion, optimizer, num\_epochs=25):

since = time.time()

for epoch in range(num\_epochs):

print('Epoch {}/{}'.format(epoch, num\_epochs - 1))

print('-' \* 10)

#set model to trainable

# model.train()

train\_loss = 0

# Iterate over data.

for i, data in enumerate(dataloaders['train']):

inputs , labels = data

inputs = inputs.to(device)

labels = labels.to(device)

optimizer.zero\_grad()

with torch.set\_grad\_enabled(True):

outputs = model(inputs)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

train\_loss += loss.item() \* inputs.size(0)

print('{} Loss: {:.4f}'.format(

'train', train\_loss / dataset\_sizes['train']))

time\_elapsed = time.time() - since

print('Training complete in {:.0f}m {:.0f}s'.format(

time\_elapsed // 60, time\_elapsed % 60))

return model

def visualize\_model(model, num\_images=6):

was\_training = model.training

model.eval()

images\_so\_far = 0

fig = plt.figure()

with torch.no\_grad():

for i, (inputs, labels) in enumerate(dataloaders['validation']):

inputs = inputs.to(device)

labels = labels.to(device)

outputs = model(inputs)

\_, preds = torch.max(outputs, 1)

for j in range(inputs.size()[0]):

images\_so\_far += 1

ax = plt.subplot(num\_images//2, 2, images\_so\_far)

ax.axis('off')

ax.set\_title('predicted: {} truth: {}'.format(class\_names[preds[j]], class\_names[labels[j]]))

img = inputs.cpu().data[j].numpy().transpose((1, 2, 0))

img = std \* img + mean

ax.imshow(img)

if images\_so\_far == num\_images:

model.train(mode=was\_training)

return

model.train(mode=was\_training)

Finally, let's start our training process with the number of epochs set to 25 and evaluate after the training process. At each training step, the model will take the input and predict the output. After that, the predicted output will be passed to the criterion to calculate the losses. Then the losses will perform a backprop calculation to calculate the gradient and finally calculating the weights and optimize the parameters with autograd.

At the visualize model, the trained network will be tested with a batch of images to predict the labels. Then it will be visualized with the help of matplotlib.

vgg\_based = train\_model(vgg\_based, criterion, optimizer\_ft, num\_epochs=25)

visualize\_model(vgg\_based)

plt.show()

Results

The final result is that you achieved an accuracy of 92%.

Epoch 23/24

----------

train Loss: 0.0044

train Loss: 0.0078

train Loss: 0.0141

train Loss: 0.0221

train Loss: 0.0306

train Loss: 0.0336

train Loss: 0.0442

train Loss: 0.0482

train Loss: 0.0557

train Loss: 0.0643

train Loss: 0.0763

train Loss: 0.0779

train Loss: 0.0843

train Loss: 0.0910

train Loss: 0.0990

train Loss: 0.1063

train Loss: 0.1133

train Loss: 0.1220

train Loss: 0.1344

train Loss: 0.1382

train Loss: 0.1429

train Loss: 0.1500

Epoch 24/24

----------

train Loss: 0.0076

train Loss: 0.0115

train Loss: 0.0185

train Loss: 0.0277

train Loss: 0.0345

train Loss: 0.0420

train Loss: 0.0450

train Loss: 0.0490

train Loss: 0.0644

train Loss: 0.0755

train Loss: 0.0813

train Loss: 0.0868

train Loss: 0.0916

train Loss: 0.0980

train Loss: 0.1008

train Loss: 0.1101

train Loss: 0.1176

train Loss: 0.1282

train Loss: 0.1323

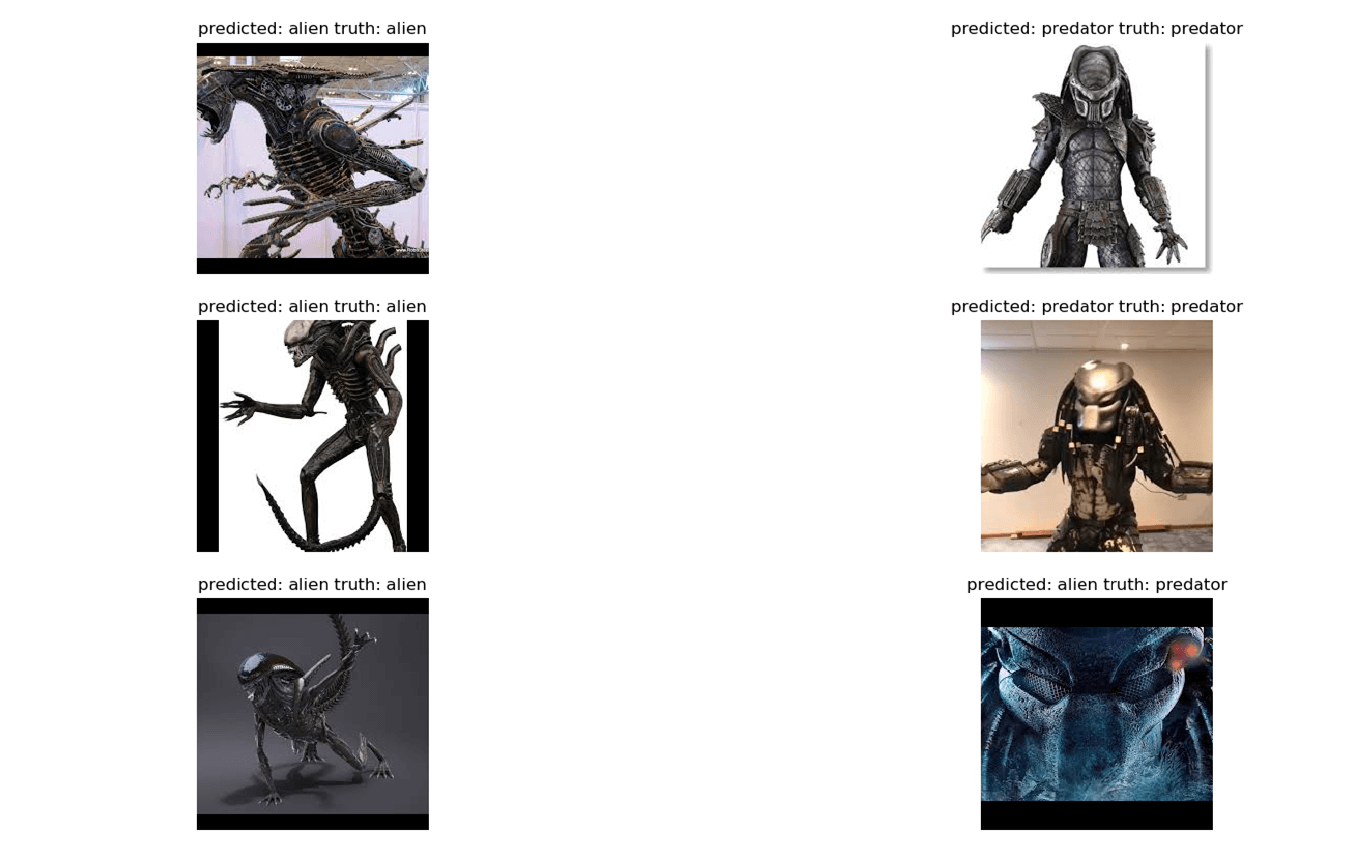
train Loss: 0.1397

train Loss: 0.1436

train Loss: 0.1467

Training complete in 2m 47s

End then the output of our model will be visualized with matplot below:



## Summary

So, let's summarize everything! The first factor is PyTorch is a growing deep learning framework for beginners or for research purpose. It offers high computation time, Dynamic Graph, GPUs support and it's totally written in Python. You are able to define our own network module with ease and do the training process with an easy iteration. It's clear that PyTorch is ideal for beginners to find out deep learning and for professional researchers it's very useful with faster computation time and also the very helpful autograd function to assist dynamic graph.