Lab 3. PyTorch and ONNX

Objective

To install **PyTorch** in your host machine and use it as the **ML framework** to create models instead of creating it from scratch with numpy that you had learnt from the previous labs.

Then, install **ONNX** to export the model into the standard format which can be used in other frameworks and can be deployed with different types of hardware.

Experiment 1 : PyTorch Installation

Step 1.

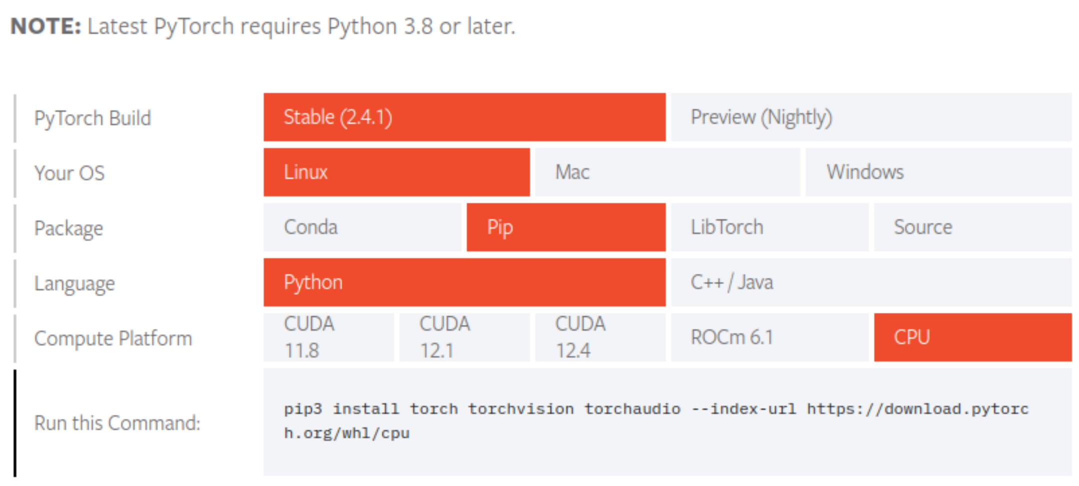
Go to this URL to get the installation command, based on your host machine's preferences.

https://pytorch.org/get-started/locally/

Select the OS of your machine (**Windows, Mac, Linux**).

Python and **PIP** will be used as the language and the package installer for pytorch. You can activate your python environment as you prefer.

For the Compute Platform, select **CPU** to reduce the GPU setup problem (CUDA, CUDNN, ROCm etc) during the lab.



NOTE: Latest PyTorch requires Python 3.8 or later.

PyTorch Build	[✓ Stable (2.4.1)] Preview (Nightly)
Your OS	[✓ Linux] Mac Windows
Package	Conda [✓ Pip] LibTorch Source
Language	[✓ Python] C++ / Java
Compute Platform	CUDA 11.8 CUDA 12.1 CUDA 12.4 ROCm 6.1 [✓ CPU]
Run this Command:	`pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cpu`

Windows
$ pip3 install torch torchvision torchaudio
Mac
$ pip3 install torch torchvision torchaudio
Linux
$ pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cpu

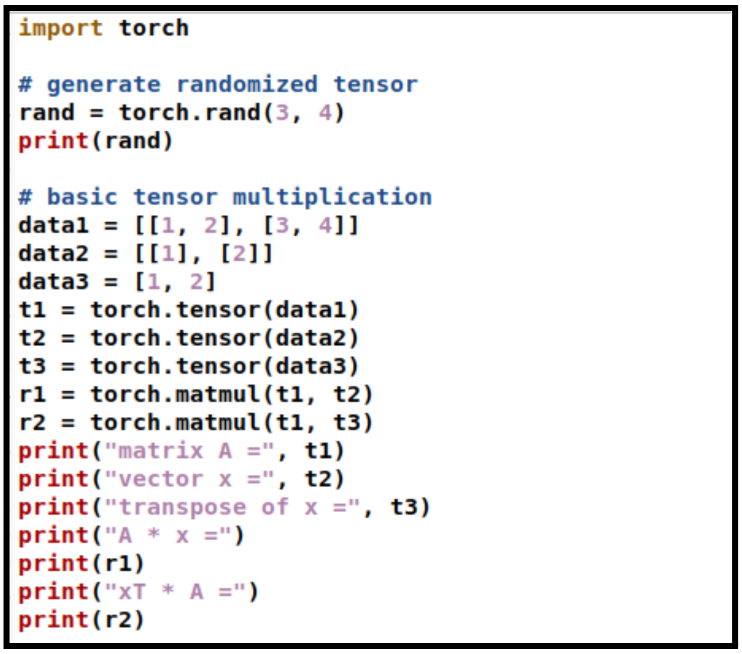
Warning
Please ensure that you have met the prerequisites depending on your package manager (We use PIP as our current package manager). **Python version >= 3.8** should be enough.

Step 2.

Verify the installation

$ pip3 show torch

After installation is completed. Do the experiment on PyTorch's tensor data with this python code:

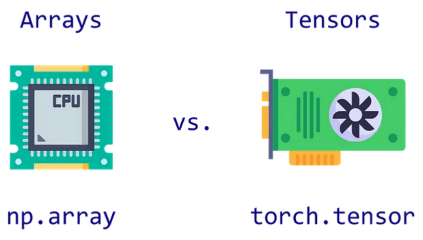


```python
import torch

# generate randomized tensor
rand = torch.rand(3, 4)
print(rand)

# basic tensor multiplication
data1 = [[1, 2], [3, 4]]
data2 = [[1], [2]]
data3 = [1, 2]
t1 = torch.tensor(data1)
t2 = torch.tensor(data2)
t3 = torch.tensor(data3)
r1 = torch.matmul(t1, t2)
r2 = torch.matmul(t1, t3)
print("matrix A =", t1)
print("vector x =", t2)
print("transpose of x =", t3)
print("A * x =")
print(r1)
print("xT * A =")
print(r2)
```

Output

A PyTorch Tensor is basically the same as a numpy array: it is just a generic n-dimensional array to be used for arbitrary numeric computation. We can do vector or matrix operations in linear algebra to the tensor data.

The biggest difference between a numpy array and a PyTorch Tensor is that a PyTorch **Tensor can run on either CPU or GPU.** The operation can be faster with GPU hardware acceleration.



Experiment 2 : ANN for Iris dataset using PyTorch

Step 1. Load Iris dataset and convert into PyTorch's tensor data type.

```python
# Load the Iris dataset from local .csv file
# dataset can be founded in here : https://gist.github.com/netj/8836201
import pandas as pd
import numpy as np

filename = 'iris.csv'
df = pd.read_csv(filename)

# Create feature set : X by drop label column in dataframe
X=df.drop('variety', axis=1)
X = X.values

# Label the output classes
# Since Setosa and Versicolor are linearly separable and between Setosa and Virginica too,
# We will do the binary classification between Setosa (0) and group of Versicolor + Virginica (1)
y=df['variety']
y = y.values.reshape(150, 1)

l_y = []
for i in range(len(y)):
    if y[i, 0] == 'Setosa':
        l_y.append([1, 0, 0])
    elif y[i, 0] == 'Versicolor':
        l_y.append([0, 1, 0])
    elif y[i, 0] == 'Virginica':
        l_y.append([0, 0, 1])
y = np.array(l_y)

# convert data into pytorch tensor
import torch
X = torch.Tensor(X)
y = torch.Tensor(y)
print(X)
print(y)
print(X.shape)
print(y.shape)
```

Step 2. Split the data into a training set and testing set.

```python
# Prepare the data
# Split the data into training set and testing set
from sklearn.model_selection import train_test_split
from torch.utils.data import TensorDataset, DataLoader

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=23)

# pack the feature X and output y together
train_ds = TensorDataset(X_train, y_train)

# create the training data, batch size.
batch_size = 1
train_dl = DataLoader(train_ds, batch_size, shuffle=True)
```

Step 3. Create ANN model for Iris classifier. We choose the 4-8-3 structure as the default.

```python
# Create NN model using PyTorch
import torch
from torch import nn

class MLP(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.layer1 = nn.Linear(input_size, hidden_size)
        self.layer_out = nn.Linear(hidden_size, output_size)
        self.activation_fn = nn.Sigmoid()

    def forward(self, x):
        z1 = self.layer1(x)
        a1 = self.activation_fn(z1)
        z_out = self.layer_out(a1)
        y = nn.Softmax(dim=1)(z_out)
        return y

model_iris = MLP(4, 8, 3)
print(model_iris)
```

```
MLP(
  (layer1): Linear(in_features=4, out_features=8, bias=True)
  (layer_out): Linear(in_features=8, out_features=3, bias=True)
  (activation_fn): Sigmoid()
)
```

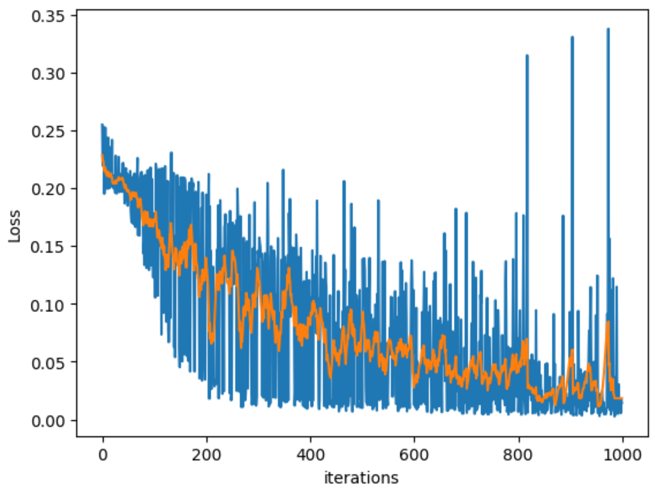
Step 4. Train the model.

```python
# Training
from torch.optim import SGD

# hyperparameters
learning_rate = 0.002
momentum = 0.1

# Loss function
loss_fn = nn.MSELoss(reduction='mean')
# Stochastic Gradient Descent (SGD) as optimizer in training procedure
#
# v = momentum * v - lr * grad(w)
# w = w + v
optimizer = SGD(model_iris.parameters(), lr=learning_rate, momentum=momentum)

num_epochs = 1000
batch_size = 1

# Start iterate for model training
loss_arr = []
for epoch in range(num_epochs):
    for x_batch, y_batch in train_dl:
        # 1. Generate predictions
        predict = model_iris(x_batch)
        # 2. Calculate loss
        loss = loss_fn(predict, y_batch)
        # 3. Compute gradients
        loss.backward()
        # 4. Update parameters using gradients
        optimizer.step()
        # 5. Reset the gradients to zero
        optimizer.zero_grad()
    if epoch % (num_epochs/10) == 0:
        print(f'Epoch {epoch} Loss {loss.item():.4f}')
    loss_arr.append(loss.item())
```

```
Epoch 0 Loss 0.2550
Epoch 100 Loss 0.1654
Epoch 200 Loss 0.1945
Epoch 300 Loss 0.1419
Epoch 400 Loss 0.0165
Epoch 500 Loss 0.0099
Epoch 600 Loss 0.0171
Epoch 700 Loss 0.1787
Epoch 800 Loss 0.0066
Epoch 900 Loss 0.0166
```

Step 5. Plot the loss function vs epoch

```python
# Plot the epoch vs loss
import matplotlib.pyplot as plt

# calculate moving average of loss
def wma(data, window_size):
    weights = np.arange(1, window_size + 1)
    return list(np.convolve(data, weights, mode='valid') / weights.sum())
loss_avg_arr = []
window_size=15
loss_avg_arr = wma(loss_arr, window_size)
for i in range(window_size - 1):
    loss_avg_arr.append(loss_avg_arr[-1])
plt.plot(list(range(num_epochs)), loss_arr)
plt.plot(list(range(num_epochs)), loss_avg_arr)
plt.xlabel('iterations')
plt.ylabel('Loss')
plt.show()
```

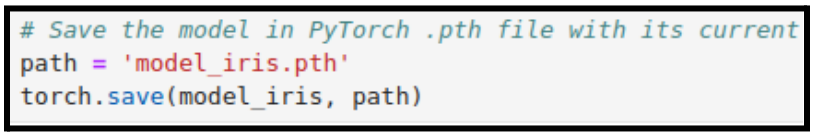


Step 6. Test the resulted model with test data

```python
# Testing resulted model by measuring the accuracy
# accuracy = num of correction / num of total prediction

correct = 0
total = y_test.shape[0]
y_predict = model_iris.forward(X_test)
for i in range(total):
    # compare the predicted output with the actual output
    # by using the indices of the maximum value in the output layer
    val_predict, indices_predict = torch.max(y_predict[i, :], 0)
    val_actual, indices_actual = torch.max(y_test[i, :], 0)
    if indices_predict == indices_actual:
        correct += 1
print("number of correction=", correct, " out of ", total, ", accuracy=", correct/total)
```

```
number of correction= 30  out of  30 , accuracy= 1.0
```

Step 7. We can save the model in .pth file to be used later.

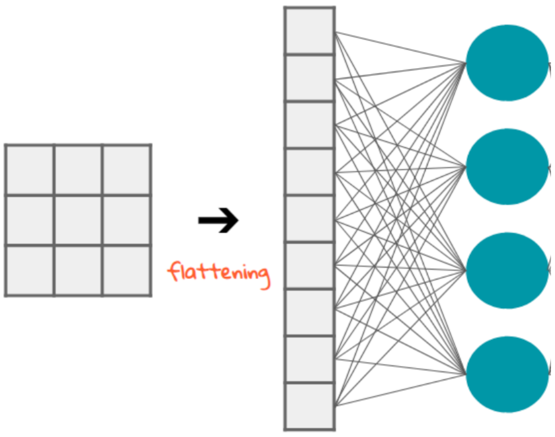


```
# Save the model in PyTorch .pth file with its current
path = 'model_iris.pth'
torch.save(model_iris, path)
```

Experiment 3 : ANN for MNIST handwritten digit using PyTorch

MNIST contains 70,000 images of handwritten digits: 60,000 for training and 10,000 for testing. The images are grayscale, 28x28 pixels, and centered to reduce preprocessing and get started quicker.

Every pixel in a 28x28 image will be the input of ANN. We need to rearrange it by using the **Flattening** technique first.



Please, look at these example .ipynb files in the MSTeam for how to read / manipulate the MNIST dataset.

resize_image.ipynb

load_mnist.ipynb

You need to create the classifier for the MNIST dataset with ANN by using the knowledge from Iris classifier in experiment 2. The structure is 784-100-100-10 (2 hidden layers, 10 outputs) as the default. The size of the hidden layer can be adjusted only.

Experiment 4 : ONNX Installation

What is ONNX?

The **O**pen **N**eural **N**etwork e**X**change (ONNX) is an open format designed to represent any type of machine learning model. The model can be NN-based or non-NN-based (Decision Tree, SVM, Rule-based etc.) These models can be trained using various frameworks including PyTorch, Keras/TensorFlow, Scikit-learn etc. Each of these frameworks generates trained models in different formats making them non-portable for use.

ONNX format allows for framework interoperability by providing a **uniform format** that acts as an **intermediate** between machine learning frameworks. This interoperability allows trained models to be easily deployed in **different platforms** or run the model on **ONNX runtime.**

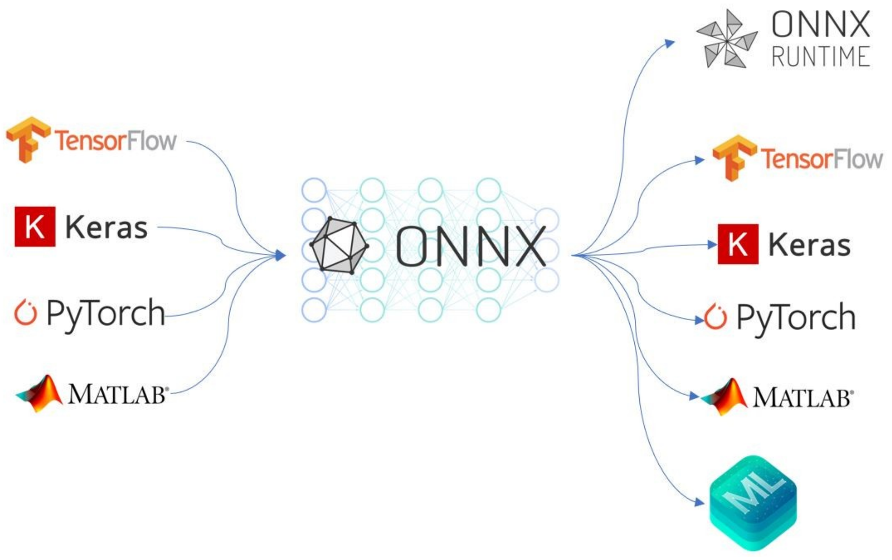


Fig. How ONNX is the intermediate between ML frameworks

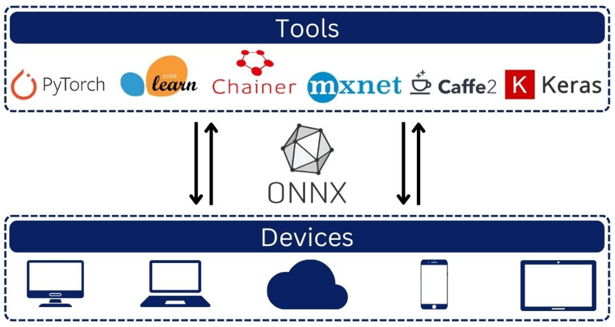


Fig. Deploying the ML model into the different device with ONNX

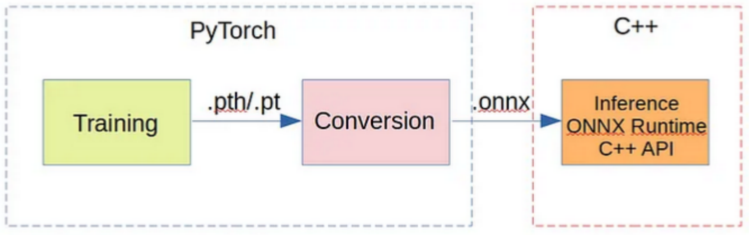


Fig. ONNX conversion process and deployment using ONNIX runtime

Step 1.

Install ONNX and its dependencies by using PIP.

$ pip install onnx

$ pip install onnxscript

Step 2.

Create a NN model with pytorch like in the experiment 2. Then, export the model as ONNX format file .onnx by **torch.onnx.export()**.

```
# Export ONNX formatted model from .pth model
import torch
input_size_iris = 4 # input for iris model
dummy_input = torch.rand(1, input_size_iris)
input_names = ["input"]
output_names = ["output"]
torch.onnx.export(loaded_model,
                  dummy_input,
                  "model_iris.onnx",
                  verbose=False,
                  input_names=input_names,
                  output_names=output_names,
                  export_params=True,
                  )
```

Save the .onnx file in the current directory by running:

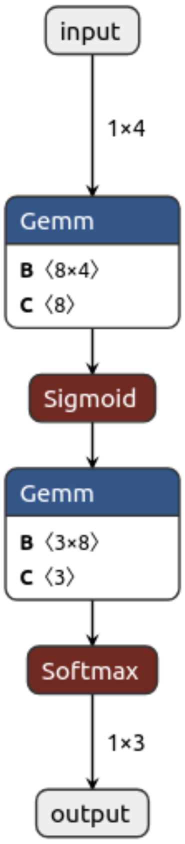
```
onnx_program.save("my_image_classifier.onnx")
```

Step 3.

Visualize the architecture of our exported ONNX model using **Netron.** Netron can either be installed locally, or run directly from the web browser. Let's try the web version by opening the following link: https://netron.app/. It will ask the location of our .onnx file.

Check the model structure in the graph. It should be the same structure as the PyTorch model.

Example graph. (It depends on your model)



Step 4.

Load the ONNX file back into memory and check if it is well formed with the following code:

```
import onnx
onnx_model = onnx.load("my_image_classifier.onnx")
onnx.checker.check_model(onnx_model)
```

Experiment 5 : Run .onnx file with ONNX runtime

Install ONNX runtime (ref : https://onnxruntime.ai/docs/install/)

For only CPU mode

$ pip install onnxruntime

(for Raspberry Pi only, this is for the last lab using Pi)

https://onnxruntime.ai/docs/tutorials/iot-edge/rasp-pi-cv.html

We will continue in the next lab by using Raspberry Pi5