[SKT AI Course: Deep Learning Basics]

Practice #3: PyTorch Basics

Basic Concepts, Tensors, Data Processing, Autograd Mechanics, Feed-Forward NN

TA: Jun-Sik Choi & Jee-Seok Yoon

Instructor: Heung-Il Suk

hisuk@korea.ac.kr

http://www.ku-milab.org

Department of Brain and Cognitive Engineering,
Korea University

October 24, 2017

Contents

PyTorch vs. TensorFlow

Machine
Intelligence
Lab

What is PyTorch?

- **PyTorch**
  - Python based deep learning library for researching and developing deep learning models
  - Origin from lua-based deep learning library, Torch
  - Written by native python language
    (Not a simple set of wrapper to support Python)
  - Actively used at Facebook
  - Essentially a GPU enabled version for NumPy with higher-level functionality for building and training deep neural networks

What is TensorFlow?

- **TensorFlow**
  - Deep learning library for researching, developing, and distributing deep learning models
  - Developed by Google Brain and actively used at Google
  - Programming language embedded within Python (TensorFlow codes are complied into a graph by Python and then run by the TensorFlow execution engine)

Differences - Adoption

- **TensorFlow**
  - Well documented
  - Large user pool
  - Many tutorials are available
  - Hundreds of implemented and trained models on GitHub

- **PyTorch**
  - Quickly getting its momentum
  - Still in beta version (v. 0.2.0)
  - Nice documentation
  - Official tutorials
  - Several computer vision architectures available

Machine Intelligence Lab

Differences – Graph definition

- They are different in a way to define directed acyclic graph (DAG)

- **TensorFlow**
  - Use **Static graph**
    (Graph is defined before a model can run.)
    - Support limited dynamic inputs
  - All communications are performed via **tf.session** and **tf.Placeholder**

- **PyTorch**
  - Use **Dynamic graph**
    (Graphs can be defined, changed, and executed as model runs).
    - Dynamic neural net like RNNs can benefit from this dynamic approach
  - Being **tightly integrated with Python** language, give more native and free way to work with models

Differences – Data loading

- **TensorFlow**
  - Relatively not intuitive for data loading
  - Adding preprocessing code in parallel into TensorFlow graph is not straight-forward.

- **PyTorch**
  - APIs for data loading are well designed.
  - A data loader takes a dataset and produces an iterator over the dataset.
  - <u>Parallelizing</u> data loading is simple.

Differences – Debugging

- **TensorFlow**
  - Need to use a special debugging tool, **tfdbg**
  - **tfdbg** allows to evaluate TensorFlow expressions at runtime and browse all tensors and operations in session scope

- **PyTorch**
  - Graph is defined at runtime.
  - Can use your favorite Python debugging tools such as **pdb, ipdb, Pycharm debuggers**.

Differences – Visualization

- **TensorFlow**
  - has its own visualization tool **Tensorboard.**

  - **Tensorboard** can

    - display model graph
    - plot scalar variables
    - visualize distributions and histograms
    - visualize images
    - visualize embeddings
    - play audio

- **PyTorch**
  - Currently, no equivalent for Tensorboard.
  - Yet, Integrations to tensorboard do exist.
  - Standard plotting tools (matplotlib, seaborn) can be used.

Machine Intelligence Lab

Differences – Serialization

- Both frameworks provides simple way to saving and loading models

- **TensorFlow**
  - **Tf.Saver** object provides easy way to save models and check-points.
  - Entire graph can be saved as a protocol buffer.
  - Graph can be loaded in other supported languages (C++, JAVA)

- **PyTorch**
  - Provides simple API that can save all the weights of a model

Difference – Custom extensions

- Building or binding custom extensions written in C, C++ or CUDA is doable in both frameworks

- **TensorFlow**
  - Requires more boilerplate code for custom extensions

- **PyTorch**
  - Can make extensions by simply writing an <u>interface</u> and <u>corresponding implementation</u> for each of the CPU and GPU versions

Differences – Deployment

- **TensorFlow**
  - **TensorFlow Serving** provides a easy way to deploy models.
  - Models can be deployed into embeded system or mobile platforms.
  - Provides high performance server-side deployments
  - Supports **Distributed training**

- **PyTorch**
  - For a small-scale server-side deployments are easy to wrap using Flask web server.
  - Supports **Distributed training**

Summary

- Both frameworks provides useful abstractions to reduce repeated codes and speed up the model development.

- **PyTorch**
  - Provides flexible dynamic graph definition
  - More 'pythonic' way for developing and debugging
  - An object-oriented approach
  - Better for rapid prototyping in research, and small scale projects.
- **TensorFlow**
  - Provides great visualization and deployment tools.
  - A good choice when developing a model for production and deploying on mobile platforms
  - Better for large-scale deployments especially when cross-platform and embedded deployment is a consideration

PyTorch Tensor

Machine
Intelligence
Lab

Tensor

• torch.Tensor

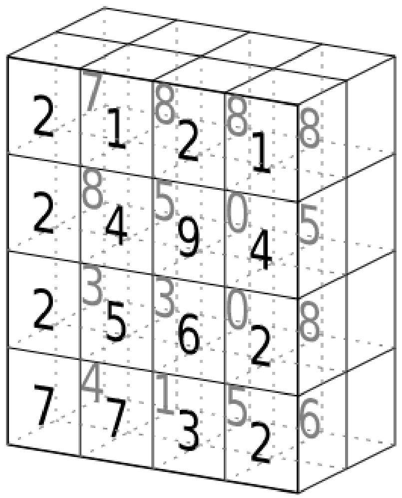
- A <u>multi-dimensional matrix</u> containing elements of a single data
- Similar to numpy's ndarray, except **torch.Tensors** can also be used on a <u>GPU to accelerate</u> computing.

't'
'e'
'n'
's'
'o'
'r'

tensor of dimensions [6]
(vector of dimension 6)

3	1	4	1
5	9	2	6
5	3	5	8
9	7	9	3
2	3	8	4
6	2	6	4

tensor of dimensions [6,4]
(matrix 6 by 4)



tensor of dimensions [4,4,2]

Create Tensor

PyTorch provides various ways to create Tensors (from list, ndarray,…)

```python
# 파이썬 리스트에서 초기화
tensor_from_list = torch.FloatTensor([[1,2,3],[-4,-5,-6]])
print("tensor_from_list : ",tensor_from_list)

# 비어있는 텐서 1
zero_tensor = torch.zeros(2,3)
print("zero_tensor : ",zero_tensor)

# 비어있는 텐서 2
empty_tensor = torch.IntTensor(2,3).zero_()
print("empty tensor: ", empty_tensor)

# 끝에 _(under score)가 붙은 method는 텐서 자체를 변화(mutate)시킨다.
tensor_from_list.abs_()
print("Apply .abs_(): ", tensor_from_list)

# 초기값이 주어지지 않은 텐서는 임의로 초기화된다.
uninitialized_tensor = torch.Tensor(2,3)
print("uninitialized_tensor : ",uninitialized_tensor)
```

```python
# [0,1) uniform distribution에서 초기화
random_tensor = torch.rand(2,3)
print("random_tensor : ",random_tensor)

# N(0,1) Normal distribution에서 초기화
normal_tensor = torch.randn(2,3)
print("normal_tensor : ",normal_tensor)

# ndarray에서 텐서 생성
ndarr = np.array([[1,2,3],[6,5,4]])
from_numpy_tensor = torch.from_numpy(ndarr)
print("from_numpy_tensor : ", from_numpy_tensor)

# Tensor에서 ndarray 생성
from_tensor_ndarray = from_numpy_tensor.numpy()
print("from_tensor_ndarray : ", from_tensor_ndarray)
```

Task #1

- Uncomment *examples_create_tensor()*, run it, and observe the results

- Refer to http://PyTorch.org/docs/master/torch.html?highlight=tensor#creation-ops and Create Torch.Tensor in various way by yourself.

- Check the GPU is available by *torch.cuda.is_available()* and if available, move tensors to GPU by *.cuda()* method.

Manipulate Tensor

```
X = torch.randn(3,5)
print("Original : ",X)

# Concatenation
concat_tensor_0 = torch.cat(seq=(X,X,X), dim=0)
print("Concat through axis 0 :", concat_tensor_0)
concat_tensor_1 = torch.cat((X,X,X),1)
print("Concat through axis 1 : ", concat_tensor_1)

# Chunking
chunk_tensor = torch.chunk(tensor=X, chunks=3, dim=0)
print("chunk_tensor : ", chunk_tensor)

# Non-zero
eye_tensor = torch.eye(3,3)
nonzero_index = torch.nonzero(eye_tensor)
print("nonzero_index : ", nonzero_index)

# Transpose
trans_tensor = torch.t(X)
print("trans_tensor", trans_tensor)
```

Machine Intelligence Lab

Task #2

- Uncomment *examples_manipulate_tensor()*, run it, and observe the results

- Refer to http://PyTorch.org/docs/master/torch.html?highlight=tensor#indexing-slicing-joining-mutating-ops and manipulate Torch.Tensor in various way by yourself.

Machine Intelligence Lab

Tensor operation

```python
A = torch.randn(2,2)
B = torch.randn(2,2)

print("Original A : ", A)
print("Original B : ", B)

# element-wise tensor addition
added_tensor = torch.add(A,B)
# or,
added_tensor_2 = A+B
print("Added tensor : ",added_tensor)

# Clamping tensor
clamp_tensor = torch.clamp(A, min=-0.5, max=0.5)
print("clamp_tensor : ", clamp_tensor)

# Divide
divide_by_const_tensor = torch.div(A,2)
divide_by_tensor = torch.div(A,B)
# or,
devied_by_tensor_2 = A/B
print("divide_by_const_tensor : ",divide_by_const_tensor)
print("divide_by_tensor : ",divide_by_tensor)
```

```python
# Element-wise multiplication
mul_by_const_tensor = torch.mul(A,10)
mul_by_tensor = torch.mul(A,B)
# or,
mul_by_tensor = A*B
print("mul_by_const_tensor : ",mul_by_const_tensor)
print("mul_by_tensor : ",mul_by_tensor)

# Matrix multiplication
matrix_mul_tensor = torch.mm(A,B)
print("matrix_multiplication : ", matrix_mul_tensor)

# Sigmoid
sigmoid_tensor = torch.sigmoid(A)
print("sigmoid_tensor : ",sigmoid_tensor)

# Summation
sum_tensor = torch.sum(A)
print("sum_tensor : ",sum_tensor)

# Mean, standard diviation
print("Mean : ",torch.mean(A), "std :", torch.std(A))
```

Task #3

- Uncomment *examples_operate_tensor()*, run it, and observe the results

- Refer to http://PyTorch.org/docs/master/torch.html?highlight=tensor#math-operations and test various Torch.Tensor operations in various way by yourself.

Quiz #1

- Implement Logistic Sigmoid function with torch.Tensor

$$h(X) = \frac{1}{1 + \exp(-X^T W)},$$

$$\textit{where}\ \text{input}\ X\ \in\ \mathbb{R}^d\ \textit{and model parameter}\ W \in\ \mathbb{R}^d$$

```
def logistic_regression(X, W):
    # X ; input tensor
    # W ; model weight tensor

    # transpose X
    X_transpose =

    # X.T * W
    X_txW =

    # exp(X.T * W)
    exp =

    # 1/(1+exp(X.T * W))
    h_x =

    print("Result of Logistic Regression : ",h_x)

    return h_x

```

Machine Intelligence Lab

Answer

```
def logistic_regression(X, W):
    # X ; input tensor
    # W ; model weight tensor

    # transpose X
    X_transpose = torch.t(X)

    # X.T * W
    X_txW = torch.mm(X_transpose,W)

    # exp(X.T * W)
    exp = torch.exp(X_txW)

    # 1/(1+exp(X.T * W))
    h_x = 1/(1+exp)

    print("Result of Logistic Regression : ",h_x)

    return h_x

```

Machine Intelligence Lab

PyTorch Autograd Mechanics

Machine
Intelligence
Lab

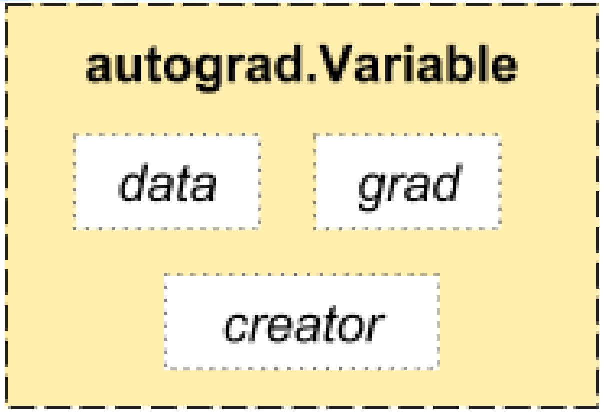
Automatic differentiation package

• Autograd package

- A reverse automatic differentiation system
- Records every operations in a DAG
- Provides classes and functions implementing automatic differentiation of arbitrary scalar valued functions

→ To apply autograd package to a **torch.Tensor**,
All tensors need to be wrapped into **autograd.Variable** objects.

Automatic differentiation package



- **class torch.autograd.Variable**
  - Wraps a tensor and records the operation applied to it
  - Supports nearly all of operations defined to the tensor
  - Can access the raw tensor through *.data*
  - Gradient w.r.t variable is accumulated into *.grad*
  - **grad_fn** references a **autograd.Function** that created the Variable

→ **autograd.Variable** + **autograd.Function** => **acyclic graph**

Example 1 : What is a autograd.Variable?

```python
def example_1():
    # Create a Variable that wraps a simple tensor
    V_no_grad = Variable(torch.ones(2,2))
    V = Variable(torch.ones(2,2),requires_grad=True)

    print(type(V))
    print(V_no_grad.requires_grad) # default = False

    print(V.data) # a raw Tensor wrapped by Variable
    print(V.grad) # no gradient accumulated yet
    print(V.grad_fn)
    # This Variable is created by User, not a autograd.Function

    # A Variable operation creates another Variable
    child = V + 2 # Add Constant

    print(child.data)
    print(child.grad)
    print(child.grad_fn)
    # You can see autograd.Function class that create this Variable

    return
```

```
type <class 'torch.autograd.variable.Variable'>
Default requires_grad? False
raw data :
 1  1
 1  1
[torch.FloatTensor of size 2x2]

initial gradient :  None
Creator? :  None
```

```
raw data :
 3  3
 3  3
[torch.FloatTensor of size 2x2]

initial gradient :  None
Creator? :  <torch.autograd.function.AddConstantBackward object at 0x1149a1a98>
```

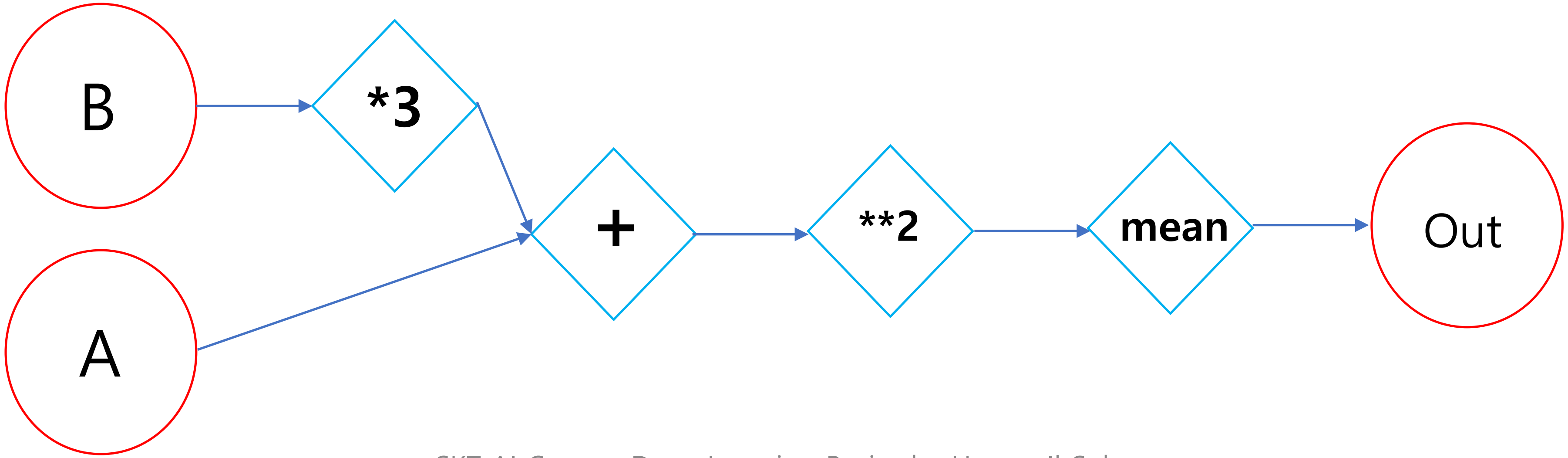
Machine Intelligence Lab

How to compute gradient?

- **torch.autograd.backward**
  - Computes the <u>sum of gradients</u> of given variables w.r.t. graph leaves
  - Differentiated using the <u>chain rule</u>
  - Can call *Variable.backward()*

  →The leaf node's gradient is computed automatically.

- **Example 2 : Gradient computation toy example**



Example 2

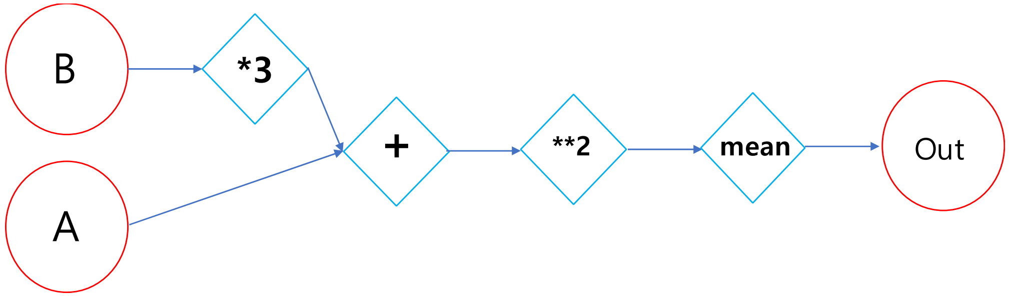
```python
def example_2():
    A = Variable(torch.eye(2,2),requires_grad=True)
    B = Variable(torch.ones(2,2),reqires_grad=True)

    inter_1 = A+3*B
    inter_2 = inter_1**2
    out = inter_2.mean()

    out.backward()

    # Check that B.grad = A.grad * 3
    print("Gradient of A : ", A.grad)
    print("Gradient of B : ", B.grad)

    return
```




```
In [15]: A.grad
Out[15]:
Variable containing:
 2.0000  1.5000
 1.5000  2.0000
[torch.FloatTensor of size 2x2]

In [16]: B.grad
Out[16]:
Variable containing:
 6.0000  4.5000
 4.5000  6.0000
[torch.FloatTensor of size 2x2]
```

```
inter_1.grad_fn
<torch.autograd.function.AddBackward at 0x10eecb4f8>

inter_2.grad_fn
<torch.autograd.function.PowConstantBackward at 0x10eecb228>

out.grad_fn
<torch.autograd.function.MeanBackward at 0x10eecb318>
```

$$\frac{\partial Out}{\partial a_i} = \frac{(a_i + b_i)}{2}, \qquad \frac{\partial Out}{\partial b_i} = \frac{3(a_i + b_i)}{2},$$

Excluding subgraphs from backward path

- **requires_grad**
  - As you saw before, if input's **requires_grad=True**, the output of operation has **requires_grad=True** too, vice versa.
  - You can **freeze part of your model** with this property. (ex. finetuning model)

- **volatile**
  - **volatile** is useful when you use a model as inference mode.
    (You won't be calling a *.backward()*)
  - if a Variable has a flag "**volatile=True**", then **requires_grad** is also False.
  - Single **volatile** leaf → **volatile** output.
    (You don't need to make every leaves have requires_grad=False)

```python
def excluding_subgraph():

    # Output has requires_grad=True if one of its leaves has requires_grad=True
    A = Variable(torch.randn(2,2),requires_grad=False)
    B = Variable(torch.randn(2,2),requires_grad=True)
    C = A+B
    print("C.requires_grad : ",C.requires_grad)

    # If an input has flag "Volatile=True", then the output has requires_grad=False
    AA = Variable(torch.randn(2,2), Volatile=True)
    BB = Variable(torch.randn(2,2), requires_grad=True)
    CC = AA+BB
    print("CC.requires_grad : ",CC.requires_grad)

    return

```

Machine Intelligence Lab

Two-layer Neural Network Toy example

```
def Two_layer_NN():
    #1. CPU vs GPU
    dtype = torch.FloatTensor
    #2. Graph specification
    N, D_in, H, D_out = 64, 1000, 100, 10
    #3. Creat dummy data
    x = Variable(torch.randn(N, D_in).type(dtype), requires_grad=False)
    y = Variable(torch.randn(N, D_out).type(dtype), requires_grad=False)
    #4. Initialize weight
    w1 = Variable(torch.randn(D_in, H).type(dtype), requires_grad=True)
    w2 = Variable(torch.randn(H, D_out).type(dtype), requires_grad=True)

    #5. Set learning rate and epoch
    learning_rate = 1e-6

    for t in range(500):
        #6. Forward Path
        y_pred = x.mm(w1).clamp(min=0).mm(w2)
        #7. Define the loss function
        loss = (y_pred - y).pow(2).sum()
        #8. Back Propagate
        loss.backward()
        #9. Update weights
        w1.data -= learning_rate * w1.grad.data
        w2.data -= learning_rate * w2.grad.data
        #10. Zero the gradients for next epoch
        w1.grad.data.zero_()
        w2.grad.data.zero_()

```

Machine Intelligence Lab

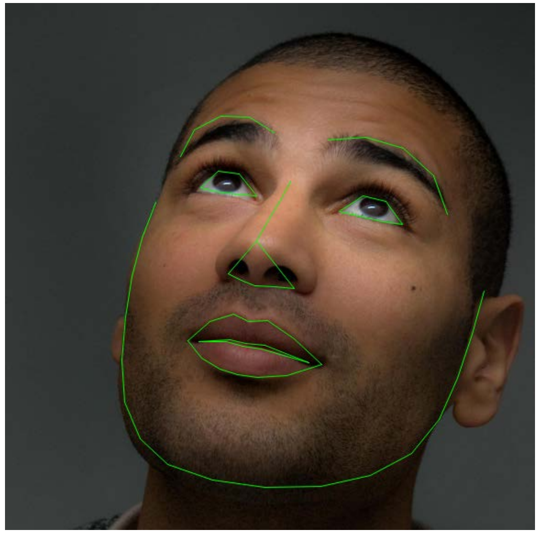
Data Loading and Processing

Data Loading and processing

- Preparing the data for a model requires considerable time
- **PyTorch** provides many tools to make data loading easy
- In this tutorial, we will see how to load and **preprocess/augment** data from a non-trivial dataset.

Class Dataset

- **torch.utils.data.Dataset**
  - An abstract class representing a dataset
  - Custom dataset should inherit Dataset and override the following methods
    - *__len__* : returns the size of the dataset.; *len(dataset)*
    - *__getitem__* : support the indexing such that ***dataset[i]*** can be used to get *ith* sample.



dataset.py

```python
class FaceLandmarksDataset(Dataset):
    """Face Landmarks dataset."""
    def __init__(self, csv_file, root_dir, transform=None):
        self.landmarks_frame = pd.read_csv(csv_file)
        self.root_dir = root_dir
        self.transform = transform

    def __len__(self):
        return len(self.landmarks_frame)

    def __getitem__(self, idx):
        img_name = os.path.join(self.root_dir, self.landmarks_frame.ix[idx, 0])
        image = io.imread(img_name)
        landmarks = self.landmarks_frame.ix[idx, 1:].as_matrix().astype('float')
        landmarks = landmarks.reshape(-1, 2)
        sample = {'image': image, 'landmarks': landmarks}

        if self.transform:
            sample = self.transform(sample)

        return sample

```

Transform

- **torchvision** package consists of popular datasets, model architectures, and common image transformations for computer vision.
- **torchvision.transforms**
  - provides common image transforms
  - transforms can be chained together using **torchvision.transforms.Compose**
  - [Various torchvision transformations]

Transform

- Let's create three transforms:
  - **Rescale**: to scale the image
  - **RandomCrop**: to crop from image randomly. This is data augmentation.
  - **ToTensor**: to convert the numpy images to torch images (we need to swap axes)
  - We will implement ***__call__()*** method so the transform need not be passed everytime it's called.

transforms.py - Rescale

```python
class Rescale(object):
    def __init__(self, output_size):
        assert isinstance(output_size, (int, tuple))
        self.output_size = output_size

    def __call__(self, sample):
        image, landmarks = sample['image'], sample['landmarks']

        h, w = image.shape[:2]
        if isinstance(self.output_size, int):
            if h > w:
                new_h, new_w = self.output_size * h / w, self.output_size
            else:
                new_h, new_w = self.output_size, self.output_size * w / h
        else:
            new_h, new_w = self.output_size

        new_h, new_w = int(new_h), int(new_w)

        img = transform.resize(image, (new_h, new_w))

        # h and w are swapped for landmarks because for images,
        # x and y axes are axis 1 and 0 respectively
        landmarks = landmarks * [new_w / w, new_h / h]

        return {'image': img, 'landmarks': landmarks}
```

Machine
Intelligence
Lab

transforms.py - Rescale

```python
class RandomCrop(object):
    def __init__(self, output_size):
        assert isinstance(output_size, (int, tuple))
        if isinstance(output_size, int):
            self.output_size = (output_size, output_size)
        else:
            assert len(output_size) == 2
            self.output_size = output_size

    def __call__(self, sample):
        image, landmarks = sample['image'], sample['landmarks']

        h, w = image.shape[:2]
        new_h, new_w = self.output_size

        top = np.random.randint(0, h - new_h)
        left = np.random.randint(0, w - new_w)

        image = image[top: top + new_h,
                      left: left + new_w]

        landmarks = landmarks - [left, top]

        return {'image': image, 'landmarks': landmarks}
```

```python
class ToTensor(object):
    def __call__(self, sample):
        image, landmarks = sample['image'], sample['landmarks']
        # swap color axis because
        # numpy image: H x W x C
        # torch image: C X H X W
        image = image.transpose((2, 0, 1))
        return {'image': torch.from_numpy(image),
                'landmarks': torch.from_numpy(landmarks)}
```

Apply transforms

```python
''' 03. Transforms '''

scale = Rescale(256)
crop = RandomCrop(128)
composed = transforms.Compose([Rescale(256),
                               RandomCrop(224)])

# Apply each of the above transforms on sample.
fig = plt.figure()
sample = face_dataset[65]
for i, tsfrm in enumerate([scale, crop, composed]):
    transformed_sample = tsfrm(sample)

    ax = plt.subplot(1, 3, i + 1)
    plt.tight_layout()
    ax.set_title(type(tsfrm).__name__)
    show_landmarks(**transformed_sample)

plt.savefig('figure.png')
elice_utils.send_image('figure.png')
plt.clf()

```

Iterate through dataset

```python
''' 04. Iterating through the dataset'''

transformed_dataset = FaceLandmarksDataset(csv_file='faces/face_landmarks.csv',
                                           root_dir='faces/',
                                           transform=transforms.Compose([
                                               Rescale(256),
                                               RandomCrop(224),
                                               ToTensor()
                                           ]))

for i in range(len(transformed_dataset)):
    sample = transformed_dataset[i]

    print(i, sample['image'].size(), sample['landmarks'].size())

    if i == 3:
        break

```

Dataloader

- Simple for loop can not …
  - **Batch** the data
  - **Shuffle** the data
  - Load the data in parallel using **multiprocessing** workers
- **Torch.utils.data.Dataloader**
  - An Iterator.
  - Combines a dataset and a sampler, and provides single- or multi-process iterators over the dataset.

Using Dataloader

```python
''' 05. Using Dataloader '''

dataloader = DataLoader(transformed_dataset, batch_size=4,
                        shuffle=True, num_workers=4)

for i_batch, sample_batched in enumerate(dataloader):
    print(i_batch, sample_batched['image'].size(),sample_batched['landmarks'].size())

    # observe 4th batch and stop.
    if i_batch == 3:
        plt.figure()
        show_landmarks_batch(sample_batched)

        break

```

Machine Intelligence Lab

Summary & Afterword

- We have seen how to write and use datasets, transforms and dataloader.
- Torchvision package provides some common datasets and transforms.
- You can use popular datasets (MNIST, COCO, LSUN, ImageNet-12, etc.) in **torchvision.datasets**
- You can load images from folder using **torchvision.datasets.ImageFolder**

Implement Feed-Forward Neural Network with PyTorch

Machine
Intelligence
Lab

Neural Networks

- Neural networks can be constructed using the **torch.nn** package.
- An **nn.Module** contains layers, and a method *forward(input)* that returns the output → torch.nn modules
- A typical training procedure for a neural network is as follows:
  1. **Define the neural network** that has some learnable parameters (or weights)
  2. **Iterate over a dataset** of inputs
  3. **Process** input through the network
  4. Compute the **loss** (how far is the output from being correct)
  5. **Propagate gradients back** into the network's parameters
  6. **Update** the weights of the network, typically using a simple update rule: **weight = weight - learning_rate * gradient**
-

Linear Regression

```
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
from torch.autograd import Variable

def LR():

    # Hyper Parameters
    input_size = 1
    output_size = 1
    num_epochs = 60
    learning_rate = 0.001

    # Toy Dataset
    x_train = np.array([[3.3], [4.4], [5.5], [6.71], [6.93], [4.168],
                        [9.779], [6.182], [7.59], [2.167], [7.042],
                        [10.791], [5.313], [7.997], [3.1]], dtype=np.float32)

    y_train = np.array([[1.7], [2.76], [2.09], [3.19], [1.694], [1.573],
                        [3.366], [2.596], [2.53], [1.221], [2.827],
                        [3.465], [1.65], [2.904], [1.3]], dtype=np.float32)

    # Linear Regression Model
    class LinearRegression(nn.Module):
        def __init__(self, input_size, output_size):
            super(LinearRegression, self).__init__()
            self.linear = nn.Lin

        def forward(self, x):
```

Feed Forward Neural Network (w/o optimizer)

```
import torchvision.datasets as dsets
import torchvision.transforms as transforms
from torch.autograd import Variable


# Hyper Parameters
input_size = 784
hidden_size = 500
num_classes = 10
num_epochs = 5
batch_size = 100
learning_rate = 0.001

# MNIST Dataset
train_dataset = dsets.MNIST(root='./data',
                            train=True,
                            transform=transforms.ToTensor(),
                            download=True)

test_dataset = dsets.MNIST(root='./data',
                           train=False,
                           transform=transforms.ToTensor())

# Data Loader (Input Pipeline)
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)

# Neural Network Model (1 hidden layer)
class Net(nn.Module):
    def __init__(self, input_size, hidden_size, n
```

Feed Forward Neural Network (w/ optimizer)

```python
import torchvision.datasets as dsets
import torchvision.transforms as transforms
from torch.autograd import Variable

def FFNN():
    # Hyper Parameters
    input_size = 784
    hidden_size = 500
    num_classes = 10
    num_epochs = 5
    batch_size = 100
    learning_rate = 0.001

''' 0. Dataset setting '''

    # MNIST Dataset
    train_dataset = dsets.MNIST(root='./data',
                                train=True,
                                transform=transforms.ToTensor(),
                                download=True)

    test_dataset = dsets.MNIST(root='./data',
                               train=False,
                               transform=transforms.ToTensor())

    # Data Loader (Input Pipeline)
    train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                               batch_size=batch_size,
                                               shuffle=True)

    test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                              batch_size=batch_size,
                                              shuffle=False)

''' 1. Define Neural Network '''
```

Summary

- we have seen how to define model, loss, optimizer, and train.
- PyTorch provides various options to train and define neural networks.
- More advance networks (CNN, RNN, GAN, … etc.) can be implemented in relatively readable and compact way in PyTorch

Reference

[1] https://medium.com/towards-data-science/PyTorch-vs-TensorFlow-spotting-the-difference-25c75777377b

[2] https://awni.github.io/PyTorch-TensorFlow/?imm_mid=0f5a8c&cmp=em-data-na-na-newsltr_ai_20170828

[3] https://github.com/yunjey/PyTorch-tutorial

[4] https://chsasank.github.io/

[5] https://www.youtube.com/watch?v=nbJ-2G2GXL0&t=113s

[6] https://github.com/llSourcell/PyTorch_in_5_minutes/blob/master/demo.py

Thank you

For your attention!!!

(Q & A)

hisuk (AT) korea.ac.kr

http://www.ku-milab.org

Machine
Intelligence
Lab