Introduction to pytorch

Geoffroy Peeters
LTCI, Télécom Paris, IP-Paris

version: 0.2 (2021/03/02)
slides done using https://remarkjs.com/

Pytorch

- **PyTorch** is an open-source machine-learning library based on the **Torch** library
  - **Torch** was also an open-source machine-learning library based on the **Lua** programming language but its development is no longer active since 2018
  - **PyTorch** is actively developed since then
  - **PyTorch** can be considered as the python interface to Torch
- Used by:
  - Facebook AI Research Group (FAIR), IBM, Yandex and the Idiap Research Institute.
- Website and documentation
  - https://pytorch.org
  - https://pytorch.org/docs/stable/index.html
- Other Usefull resources
  - Deep Learing with pytorch: A 60 minute blitz:
    - https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html
  - PyData, Berlin 2008, "Deep Neural Networks with Pytorch":s
    - https://youtu.be/KoyUYUcEyT8

Pytorch

Importing pytorch package

```
import torch
torch.__version__
torch.__path__
```

```
'1.7.1'
['/anaconda3/lib/python3.8/site-packages/torch']
```

Torch tensors

Creating a tensor.

Official doc: https://pytorch.org/docs/stable/tensors.html

Create a tensor, get size

```
x = torch.tensor([[5.5, 3],[2, 1]], dtype=torch.float32) # --- or float
x = torch.tensor([[5.5, 3],[2, 1]], dtype=torch.float64) # --- or double
print(x)
print(x.size())
```

```
tensor([[5.5000, 3.0000],
        [2.0000, 1.0000]], dtype=torch.float64)
torch.Size([2, 2])
```

Change the shape of a tensor

```
x = torch.randn(4,4)
y = x.view(16)
z = x.view(-1, 8)   # --- ues '-1' when you want the relative shape to be infered automaticallly
print(x.size(), y.size(), z.size())
```

```
torch.Size([4, 4]) torch.Size([16]) torch.Size([2, 8])
```

Torch tensors

Various ways of creating tensors

```
%pylab
import torch.nn

x = torch.arange(-10, 10, 0.1)
plot(x.numpy(), torch.nn.Tanh()(x).numpy());
plot(x.numpy(), torch.nn.functional.tanh(x).numpy());

x = torch.Tensor(2, 3)
x = torch.rand(2, 3)
x = torch.randn(2, 3)
x = torch.eye(3)
x = torch.ones(10)
x = torch.ones(2,3)
x = torch.zeros(10)
x = torch.ones_like(x)
x = torch.arange(5)
x = torch.arange(0, 5, step=1)
x = torch.linspace(1, 10, steps=10)
x = torch.logspace(start=-10, end=10, steps=5)

x.size()
torch.numel(x)
```

Tensor Dimensions

Matrices are store row by row: [[row1],[row2],[row3]]

```
v = torch.arange(0,6).view(2,3)

v
tensor([[0, 1, 2],
        [3, 4, 5]])

v[0]    # --- return the first row
tensor([0, 1, 2])

v[0][2] # --- second elements in first row
tensor(2)

v[0,2]  # --- same as above
tensor(2)

v[0,:]    # --- return all elements of the first row
tensor([0, 1, 2])

v[:,0]    # --- return all elements of the first column
tensor([0, 3])

v[1].fill_(2)
tensor([[0, 1, 2],
        [2, 2, 2]])
```

From / To numpy

Torch to Numpy: .numpy()

```
a = torch.ones(2,5, dtype=torch.float64)
b = a.numpy()

print(type(b))
<class 'numpy.ndarray'>
print(b.dtype)
float64
print(b)
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
```

Numpy to Torch: .from_numpy( )

Warning: both ndarray and Tensor share the same memory storage.

```
a = np.ones((2,5)) # --- by default uses dtype=np.float64
a = np.ones((2,5), dtype=np.int32)
b = torch.from_numpy(a)
np.add(a, 1, out=a)

print(a)
[[2 2 2 2 2]
 [2 2 2 2 2]]
print(b)
tensor([[2, 2, 2, 2, 2]
```

Operators

see https://jhui.github.io/2018/02/09/PyTorch-Basic-operations/

Addition

```
x = torch.Tensor(2, 3)
y = torch.rand(2, 3)

z1 = x + y
z2 = torch.add(x, y)

r1 = torch.Tensor(2, 3)
torch.add(x, y, out=r1)
```

In-place operation: all operations end with “_” is in place operations

```
x.add_(y)   # Same as x = x + y
```

Matrix operators

Dot product of 2 tensors: torch.dot

```
vec1 = torch.ones(10)
vec2 = 2*torch.ones(10)
r = torch.dot(vec1,vec2)
r = vec1 @ vec2

print(vec1.size())
torch.Size([10])

print(vec2.size())
torch.Size([10])

print(r)
tensor(10.)
```

Matrix operators

Matrix, vector products: torch.mv

```
mat = torch.randn(2, 4)
vec = torch.randn(4)
r = torch.mv(mat, vec)
r = mat @ vec

print(r.size())
torch.Size([2])
```

Matrix operators

Matrix, Matrix products: torch.mm

```
mat1 = torch.ones(5,10)
mat2 = torch.ones(10,20)
r = torch.mm(mat1, mat2)
r = mat1 @ mat2

print(mat1.size())
torch.Size([5, 10])

print(mat2.size())
torch.Size([10, 20])

print(r.size())
torch.Size([5, 20])
```

Matrix operators

Element-wise mutiplication: torch.mul

```
mat1 = torch.ones(5,10)
mat2 = 2*torch.ones(5,10)
r = torch.mul(mat1,mat2)
r = mat1 * mat2

print(r.size())
torch.Size([5, 10])
```

Matrix operators

torch.matmul

matmul(tensor1, tensor2, out=None) -> Tensor

Matrix product of two tensors.

The behavior depends on the dimensionality of the tensors as follows:

- If both tensors are 1-dimensional, the dot product (scalar) is returned.
- If both arguments are 2-dimensional, the matrix-matrix product is returned.
- If the first argument is 1-dimensional and the second argument is 2-dimensional, a 1 is prepended to its dimension for the purpose of the matrix multiply. After the matrix multiply, the prepended dimension is removed.
- If the first argument is 2-dimensional and the second argument is 1-dimensional, the matrix-vector product is returned.
- If both arguments are at least 1-dimensional and at least one argument is N-dimensional (where N > 2), then a batched matrix multiply is returned. If the first argument is 1-dimensional, a 1 is prepended to its dimension for the purpose of the batched matrix multiply and removed after. If the second argument is 1-dimensional, a 1 is appended to its dimension for the purpose of the batched matrix multiple and removed after. The non-matrix (i.e. batch) dimensions are :ref:broadcasted <broadcasting-semantics> (and thus must be broadcastable). For example, if :attr:tensor1 is a :math:(j \times 1 \times n \times m) tensor and :attr:tensor2 is a :math:(k \times m \times p) tensor, :attr:out will be an :math:(j \times k \times n \times p) tensor.

Matrix operators

Broadcasting

- refers to how pytorch treats arrays with different dimension during arithmetic operations which lead to certain constraints,
- the smaller array is broadcast across the larger array so that they have compatible shapes.

```
matA = torch.ones(5,10)
matB = torch.ones(5,1)
print( (matA + matB).size() )
torch.Size([5, 10])

matA = torch.ones(5,10)
vecB = torch.ones(5)
print( (matA + vecB).size() )
RuntimeError: The size of tensor a (10) must match the size of tensor b (5) at non-singleton dimension 1

matA = torch.ones(5,10)
matC = torch.ones(1,10)
print( (matA + matC).size() )
torch.Size([5, 10])

matA = torch.ones(5,10)
vecC = torch.ones(10)
print( (matA + vecC).size() )
torch.Size([5, 10])
```

Autograd

Autograd provides automatic differentiation for all operations on Tensors.

Introduction: https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html

Autograd

Example:

- implement the following **computation graph** from x to z: $z = \sum_i 2x_i^3$.
- declare that we will need a gradient w.r.t. to x: x = ... requires_grad=True.
- get $\frac{dz}{dx_i}$ we perform the backward propagation from z: z.backward.
- result is stored in x.grad: x.grad contains $\frac{dz}{dx_i} = 6$.

```
x = torch.ones(2, 2, requires_grad=True)
y = 2 * (x**3)
z = y.sum()
print(z.grad_fn)
<SumBackward0 object at 0x1254df940>

z.backward(retain_graph=True)
print(x.grad)
tensor([[6., 6.],
        [6., 6.]])

#print(x.grad.data)
z.backward(retain_graph=True)
print(x.grad)
tensor([[12., 12.],
        [12., 12.]])

x.grad.zero_();
print(x.grad)
tensor([[0., 0.],
```

package torch.NN

NN is the main package for neural networks in pytorch. It contains many subpackage to MLP, CNN, RNN, activation functions, loss functions.

Simple example: a linear regression in pytorch

```
import torch.nn

# --- define the model (it is not yet trained)
model = torch.nn.Linear(in_features=64, out_features=1, bias=True)

# --- feed 100 data in the model and get outputs
x = torch.rand(100, 64)
hat_y = model(x)

print(hat_y.size())
torch.Size([100, 1])
```

package torch.NN

```
# model contains all the parameters of the model (weight matrices, bias vectors, ...)
print(model.__dict__)
{'training': True, '_parameters': OrderedDict([('weight', Parameter containing:
tensor([[ 0.0796,  0.0870,  0.0066,  0.0872, -0.0659, -0.0428, -0.0776,  0.0235,
          0.0558,  0.0627,  0.0015, -0.0697,  0.0526,  0.0009, -0.1122,  0.0895,
          0.0680,  0.0905, -0.0038, -0.0039,  0.0035,  0.0418,  0.1073, -0.0257,
          0.0051,  0.0752,  0.1111, -0.1130, -0.0794, -0.0911,  0.1096, -0.1208,
         -0.0234,  0.0403, -0.0664,  0.0177, -0.1109,  0.0644,  0.1243,  0.1025,
         -0.0103, -0.0477,  0.0397, -0.1160,  0.0140, -0.1133, -0.0942, -0.0603,
          0.1001,  0.0752,  0.0282, -0.0561,  0.0528, -0.0917,  0.0540,  0.0511,
          0.0630, -0.1248, -0.0035, -0.0327,  0.0693, -0.1036,  0.1133, -0.1162]],
       requires_grad=True)), ('bias', Parameter containing:
tensor([-0.0388], requires_grad=True))]), '_buffers': OrderedDict(), '_non_persistent_buffers_set': set(), '_backward

print(model.weight)
...

print(model.bias)
Parameter containing:
tensor([-0.0388], requires_grad=True)
```

package torch.NN

```
# various ways to access the parameters of the model without knowing their names
#   param: are of types torch.nn.parameter.Parameter
#   param.data contains the value (without telling it is a trainable parameter)
for param in model.parameters():
  print(param)
...

for name, param in model.named_parameters():
  print('{}:{}'.format(name, param.data))
weight:tensor([[ 0.0796,  0.0870,  0.0066,  0.0872, -0.0659, -0.0428, -0.0776,  0.0235,
          0.0558,  0.0627,  0.0015, -0.0697,  0.0526,  0.0009, -0.1122,  0.0895,
          0.0680,  0.0905, -0.0038, -0.0039,  0.0035,  0.0418,  0.1073, -0.0257,
          0.0051,  0.0752,  0.1111, -0.1130, -0.0794, -0.0911,  0.1096, -0.1208,
         -0.0234,  0.0403, -0.0664,  0.0177, -0.1109,  0.0644,  0.1243,  0.1025,
         -0.0103, -0.0477,  0.0397, -0.1160,  0.0140, -0.1133, -0.0942, -0.0603,
          0.1001,  0.0752,  0.0282, -0.0561,  0.0528, -0.0917,  0.0540,  0.0511,
          0.0630, -0.1248, -0.0035, -0.0327,  0.0693, -0.1036,  0.1133, -0.1162]])
bias:tensor([-0.0388])

for name, param in model.state_dict().items():
  print('{}:{}'.format(name, param.data))
...
```

Defining a network

There are several ways to define a neural network in pytorch depending on how much you want to use pre-defined functionnalities of pytorch.

In the following we looks at implementations of a network:

- 1) from scratch: writting all forward, loss, backward and update ourself
- 2) using autograd: we do not need to write anymore the backward
- 3) using prefined nn.Sequential, predifined losses and model.parameters
- 4) using optim to perform the update
- 5) using a class to define the network

1) Defining a network from scratch

```python
dtype = torch.float
device = torch.device("cpu")

N, D_in, H, D_out = 64, 1000, 100, 10

x = torch.randn(N, D_in, device=device, dtype=dtype)
y = torch.randn(N, D_out, device=device, dtype=dtype)
w1 = torch.randn(D_in, H, device=device, dtype=dtype)
w2 = torch.randn(H, D_out, device=device, dtype=dtype)

learning_rate = 1e-6
for t in range(500):
    # --- forward
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)

    # --- loss
    loss = (y_pred - y).pow(2).sum().item()

    # --- backward
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    # --- update
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

2) Defining a network using autograd

```
dtype = torch.float
device = torch.device("cpu")

N, D_in, H, D_out = 64, 1000, 100, 10

x = torch.randn(N, D_in, device=device, dtype=dtype)
y = torch.randn(N, D_out, device=device, dtype=dtype)
w1 = torch.randn(D_in, H, device=device, dtype=dtype, requires_grad=True)
w2 = torch.randn(H, D_out, device=device, dtype=dtype, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    # --- forward
    y_pred = x.mm(w1).clamp(min=0).mm(w2)

    # --- loss
    loss = (y_pred - y).pow(2).sum()

    # --- backward
    loss.backward()

    # --- update
    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

package torch.NN

Generic modules

```
torch.nn.Sequential
torch.nn.BatchNorm1d / torch.nn.BatchNorm2d
torch.nn.DropOut
```

Activation functions

```
torch.nn.Tanh
torch.nn.ReLU       # same as torch.nn.functional.relu
torch.nn.Sigmoid
torch.nn.LogSigmoid # same as torch.nn.functional.logsi
torch.nn.Softmax
torch.nn.LogSoftmax
```

Loss

```
torch.nn.MSELoss / torch.nn.functional.mse_loss
torch.nn.BCELoss
torch.nn.NLLLoss
torch.nn.CrossEntropyLoss  # it combines nn.LogSoftmax
torch.nn.BCEWithLogitsLoss # it combines a Sigmoid la
```

For Convolutional Neural Networks

```
torch.nn.Conv1d/ torch.nn.Conv2d
torch.nn.MaxPool1d/ torch.nn.MaxPool2d
```

For Recurrent Neural Networks

```
torch.nn.Embedding
torch.nn.RNN / torch.nn.RNNCell
torch.nn.LSTM / torch.nn.LSTMCell
torch.nn.GRU / torch.nn.GRUCell
```

3) Defining a network using nn.Sequential and nn.MSELoss and model.parameters()

```
N, D_in, H, D_out = 64, 1000, 100, 10

x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out),
)

criterion = torch.nn.MSELoss(reduction='sum')

learning_rate = 1e-4

for t in range(500):
    y_pred = model(x)
    loss = criterion(y_pred, y)

    model.zero_grad()
    loss.backward()
    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
```

package torch.optim

```python
import torch.optim

# --- We define one optimize (for example Stochastic Gradient Descent)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-5, momentum=0.9)
optimizer = torch.optim.Adam([var1, var2], lr=0.0001)

for ... # loop over data

  # --- First set to zero the .grad field in all parameters of the network (otherwise they will be accumulated)
  optimizer.zero_grad()
  # --- this is equivalent to model.zero_grad()

  # --- Then perform one step of the related optimizer algorithm (for example one step of gradient descent)
  optimizer.step()
```

4) Defining a network using ... and optim

```
N, D_in, H, D_out = 64, 1000, 100, 10

x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out),
)

criterion = torch.nn.MSELoss(reduction='sum')

learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
for t in range(500):
    y_pred = model(x)
    loss = criterion(y_pred, y)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

5) Defining a network using ... classes (instead of nn.Sequential)

Your class should be a subclass the class nn.Module (which is the base class for all neural network modules).

```
N, D_in, H, D_out = 64, 1000, 100, 10

x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

model = TwoLayerNet(D_in, H, D_out)

criterion = torch.nn.MSELoss(reduction='sum')
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = criterion(y_pred, y)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

package torch.utils

Dataset/ DataLoader

Who to make the following easy ?

```
train_dataset = Dataset(...)
train_dataloader = DataLoader(train_dataset, ...)

for epoch in range(nb_epoch):
  for i_batch, sample_batched in enumerate(train_dataloader):
    x, y = sample_batched[0], sample_batched[0]
    y_pred = model(x)
    loss = criterion(y_pred, y)
    ...
```

package torch.utils

Data/ Dataset/ from arrays

https://pytorch.org/docs/stable/data.html#torch.utils.data.Dataset

```
import numpy as np
import torch.utils.data

inputs = np.random.randn(1000,64,64)
targets = np.random.randn(1000,1)
# --- Convert numpy to torch-tensors
inputs = torch.from_numpy(inputs)
targets = torch.from_numpy(targets)

# --- Create a train set
train_dataset = torch.utils.data.TensorDataset(inputs, targets)

x, y = train_dataset[0]
print(x.size(), y.size())
torch.Size([64, 64]) torch.Size([1])

for x, y in iter(train_dataset):
  print(x.size(),y)
torch.Size([64, 64]) tensor([-0.4926], dtype=torch.float64)
torch.Size([64, 64]) tensor([1.6715], dtype=torch.float64)
torch.Size([64, 64]) tensor([0.3556], dtype=torch.float64)
...
```

package torch.utils

Data/ Dataloader

Creating a dataloader

```
train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=8, shuffle=True)

x, y = next(iter(train_dataloader))
print(x.size(), y)
torch.Size([8, 64, 64]) tensor([[-0.7249],
        [ 0.4788],
        [-0.6805],
        [ 1.1882],
        [ 0.1682],
        [ 0.2114],
        [-0.2210],
        [ 1.1738]], dtype=torch.float64)

for i_batch, sample_batched in enumerate(train_dataloader):
    x, y = sample_batched[0], sample_batched[0]
    print(i_batch, x.size(), y.size())
    ...
0 torch.Size([8, 64, 64]) torch.Size([8, 64, 64])
1 torch.Size([8, 64, 64]) torch.Size([8, 64, 64])
2 torch.Size([8, 64, 64]) torch.Size([8, 64, 64])
...
```

package torch.utils

Data/ Dataset/ by class

https://pytorch.org/tutorials/beginner/data_loading_tutorial.html

```python
class FaceLandmarksDataset(Dataset):
    """Face Landmarks dataset."""
    def __init__(self, csv_file, root_dir, transform=None):
        """
        Args:
            csv_file (string): Path to the csv file with annotations.
            root_dir (string): Directory with all the images.
            transform (callable, optional): Optional transform to be applied
                on a sample.
        """
        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):
        if torch.is_tensor(idx):
            idx = idx.tolist()

        img_name = os.path.join(self.root_dir,
                                self.landmarks_frame.iloc[idx, 0])
        image = io.imread(img_name)
        landmarks = self.landmarks_frame.iloc[idx, 1:]
        landmarks = np.array([landmarks])
        landmarks = landmarks.astype('float').reshape(-1, 2)
        sample = {'image': image, 'landmarks': landmarks}
        if self.transform:
            sample = self.transform(sample)
        return sample
```

package torch.utils

Data/ Dataloader

Using it as iterator:

```
dataloader = torch.utils.data.DataLoader(face_dataset, batch_size=4, shuffle=True)

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

Dimensions

https://pytorch.org/docs/stable/generated/torch.nn.Linear.html
https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html
https://pytorch.org/docs/stable/generated/torch.nn.RNN.html

```
data (batch or N, in_features)
data (seq_len, batch or R, in_features)

torch.nn.Linear(in_features, out_features, bias=True)
  Input: (N, *, H_in)
  Output: (N, *, H_out)
  Y (N, n_out) = X (N, n_in) W (n_out, n_int)'

torch.nn.Conv2d(in_channels, out_channels, kernel_size=(,), stride=(,)), padding=(,), dilation=1, groups=1, bias=Tr
  Input: (N, C_in, H_in, W_in)
  Output: (N, C_out, H_out, W_out)

torch.nn.RNN(*args, **kwargs)
  Input: (seq_len, batch, input_size)
  h0: (num_layers * num_directions, batch, hidden_size)

  Output: (seq_len, batch, num_directions * hidden_size)
    tensor containing the output features (h_t) from the last layer of the RNN, for each t
  h_n: (num_layers * num_directions, batch, hidden_size)
    tensor containing the hidden state for t = seq_len
```

ConvNet

```
### Without padding

# --- input (N, C_in, H_in, W_in)
input = torch.randn(64, 3, 16, 16)
# --- nn.Conv2d(in_channels, out_channels, kernel_size=(h,w))
model = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=(7,5), padding=(0,0), stride=(1,1))
# --- weight (out_channels, in_channels, h, w)
# --- output (N, C_out, H_out, W_out)
output = model(input)

print('input.size(): ', input.size())
print('model.weight.size(): ', model.weight.size())
print('model.bias.size(): ', model.bias.size())
print('output.size(): ', output.size())

input.size():  torch.Size([64, 3, 16, 16])
model.weight.size():  torch.Size([32, 3, 7, 5])
model.bias.size():  torch.Size([32])
output.size():  torch.Size([64, 32, 10, 12])

### With padding

model = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=(7,5), padding=(3,2), stride=(1,1))
output = model(input)

print('output.size(): ', output.size())

output.size():  torch.Size([64, 32, 16, 16])
```

ConvNet (cont.)

```
### Max-pooling

input = torch.randn(64, 3, 16, 16)
# --- nn.Conv2d(in_channels, out_channels, kernel_size(H,W))
model = torch.nn.Sequential(
  nn.Conv2d(in_channels=3, out_channels=32, kernel_size=(7,5), padding=(0,0), stride=(1,1)),
  nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2))
  )
output = model(input)

print('output.size(): ', output.size())

output.size():  torch.Size([64, 32, 5, 6])
```

RNN

```
input = [torch.tensor([6,7,8,9]), torch.tensor([1,2,3]), torch.tensor([3,4])]
print(input)
[tensor([6, 7, 8, 9]), tensor([1, 2, 3]), tensor([3, 4])]

seq_lens = [len(inp) for inp in input]
print(seq_lens)
[4, 3, 2]
```

Padding

Conversion to matrix with padding. The default format is [seq_len, batch_len]

```
input_padded = torch.nn.utils.rnn.pad_sequence(input, padding_value=999)

print(input_padded)
tensor([[  6,   1,   3],
        [  7,   2,   4],
        [  8,   3, 999],
        [  9, 999, 999]])
```

RNN

Pack/Unpack

```
input_pack_padded = torch.nn.utils.rnn.pack_padded_sequence(input_padded, lengths=seq_lens)

print(input_pack_padded)
PackedSequence(data=tensor([6, 1, 3, 7, 2, 4, 8, 3, 9]), batch_sizes=tensor([3, 3, 2, 1]))
```

```
input_padded = torch.nn.utils.rnn.pad_packed_sequence(input_pack_padded, padding_value=999)

print(input_padded)
(tensor([[  6,   1,   3],
        [  7,   2,   4],
        [  8,   3, 999],
        [  9, 999, 999]]), tensor([4, 3, 2]))
```

using GPU

Running your pytorch code in GPU (considering that it has been correctly configured) is quiet easy in pytorch. You need to send your model to the GPU (model.to("cuda:0")) and your data to the GPU (data.to("cuda:0")). You can then check GPU suage using nvidia-smi.

model.cuda() will send your model to the "current device", which can be set with torch.cuda.set_device(device). An alternative way to send the model to a specific device is model.to(torch.device('cuda:0')).

```
class MyNetwork(nn.Module):
  def __init__(self):
    super(MyNetwork, self).__init__()
    self.conv1 = nn.Conv2d(3, 6, (5, 5))
    self.pool = nn.MaxPool2d(2, 2)
    self.conv2 = nn.Conv2d(6, 16, (5, 5))
    self.fc1 = nn.Linear(16 * 5 * 5, 120)
    self.fc2 = nn.Linear(120, 84)
    self.fc3 = nn.Linear(84, 10)

  def forward(self, x):
    x = self.pool(F.relu(self.conv1(x)))
    x = self.pool(F.relu(self.conv2(x)))
    x = x.view(-1, 16 * 5 * 5)
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = self.fc3(x)
    return x

model = MyNetwork(D_in, H, D_out)
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

using Tensorboard

Tensorboard allows you to monitor the training of your model through the vizualization of its parameters (loss, accuracy, ...) within a web-browser.

see https://pytorch.org/docs/stable/tensorboard.html

Lauch the tensorboard server using directory 'runs' (the place where the files will be written)

```
pip install tensorboard
tensorboard --logdir=runs
```

From whithin pytorch add the functions to write probe variables

```
# Writer will output to ./runs/ directory by default
from torch.utils.tensorboard import SummaryWriter
import numpy as np

writer = SummaryWriter()

for n_iter in range(100):
    writer.add_scalar('Loss/train', np.random.random(), n_iter)
    writer.add_scalar('Loss/test', np.random.random(), n_iter)
    writer.add_scalar('Accuracy/train', np.random.random(), n_iter)
    writer.add_scalar('Accuracy/test', np.random.random(), n_iter)
```