

# Your first Deep Learning code

11-785 / Fall 2018 / Recitation 2

Shaden Shaar, Raphaël Olivier

# Recap

---

You have seen :

- What numpy is for and how to use it for general-purpose computations and algebra
- What a neural network is (a complicated functions with parameters)
- What it can model (everything)
- Some basics of how to train it

Today, we start learning how to **write deep learning code**

# Plan

---

- Why use deep learning frameworks/which ones
- The philosophy of pytorch
- Operations in pytorch
- Create and run a model
- Train a model
- Some common issues

Advanced data loading and optimization will be covered in detail next week !

# Logistics

---

## Material

On [the GitHub repository](#) you will find two notebooks.

*Tutorial-pytorch* : some example codes of what we will see today, often with more details. You can look at it in parallel or later.

*Pytorch-example* : a complete code example that we will use in the end.

# Logistics

---

## Content

Unfortunately we need to take some advance on the lectures so that you can do the homeworks.

In HW1 part 1 : you are asked to write your own version of some tools we see today.

In everything else : you will use these tools.

Conclusion : pay attention ;)

# Deep Learning Frameworks

---

What do they provide ?

- Computation (often with some numpy support/encapsulation)
- GPU support for parallel computations
- Some basic neural layers to combine in your models
- Tools to train your models
- Enforce a general way to code your models
- And most importantly, **automatic *backpropagation*.**

# Which one to choose

---

Just in python, are lots of available frameworks to use :  
Tensorflow, Pytorch, Keras, Theano, Caffe, DyNet...

They differ in philosophy, performance, user-friendliness,  
verbosity...

Some of them tackle specific problems (language, image,...) or  
contexts (Big Data...)

Let's review a few.

# Tensorflow

---

- Developed by Google, one of the most widely used
- Provides very efficient computations
- Approach a bit surprising when you are used to python or a similar language
- It's kind of hard to get used to it.

It is a *static framework* : you first define a computational graph that cannot change, and later feed it with some data. A bit like Fortran (supposedly). More on that in a later recitation.

# Pytorch

---

- Developed by Facebook, also **widely used**
- Reasonably **easy to use**, very python-friendly : you create and inherit classes
- Performance okay, although on some points below Tensorflow
- Can be **a bit verbose** (but that can be arranged by extra frameworks)

**Pytorch is the framework used in this course.** If you want to use something else, you can but you're on your own (Pytorch will be mandatory for HW2 part 1).

# Keras

---

- Very easy to use
- Performance below other frameworks, but it's possible you wouldn't see much difference
- Might be a bit limited (some complex things would be hard to do with it)

Keras is actually just a frontend, that can use either Tensorflow or Theano as its backend

# DyNet

---

- An example of more specialized framework
- Lots of tools very handy when *working with language*
- "Smart" framework that will optimize a lot of stuff in your place
- Developed by Graham Neubig (CMU) and his group.

# Back to Pytorch

---

We recommend Pytorch 0.4.1

You should all have access to an environment with it, and hopefully a GPU.

.....Let's start!

# Data and operations

Use the `torch.Tensor` class (~array)

```
# Create uninitialized tensor
x = torch.FloatTensor(2,3)
# from numpy
np_array = np.random.random((2,3)).astype(float)
x1 = torch.FloatTensor(np_array)
x2 = torch.randn(2,3)
# export to numpy array
x_np = x2.numpy()
# basic operation
x = torch.arange(4,dtype=torch.float).view(2,2)
s = torch.sum(x)
e = torch.exp(x)
# elementwise and matrix multiplication
z = s*e + torch.matmul(x1,x2.t()) # size 2*2
```

Looks a lot like numpy (and binded with it)

# Move Tensors to the GPU

For big computations, it is likely to make computations faster !

```
# create a tensor
x = torch.rand(3,2)
# copy to GPU
y = x.cuda() # type torch.cuda.FloatTensor
z = x ** 3
# copy back to CPU
t = z.cpu()
# get CPU tensor as numpy array
print(z.numpy())
```

**Warning : you cannot use an operation on two Tensors on different processing units !**

**You cannot export a gpu tensor on numpy !**

Please take a look at the provided tutorial's examples and error cases.

# Backpropagation

---

You haven't seen it yet (mentioned Wednesday)

Backpropagation in a nutshell :

You have seen gradient descent, and you know that to train a network you need to compute gradients, i.e. derivatives, of some loss (~divergence) over every parameter (weights, biases)

To compute them (with the chain rule), we first do a **forward pass** to compute the output, the loss and store *all intermediate results*.

Then in a **backward pass**, we compute all possible partial derivatives.

# Forward pass

Teaser for Monday's slides

- Input:  $D$  dimensional vector  $\mathbf{x} = [x_j, j = 1 \dots D]$

Set:

- $D_0 = D$ , is the width of the 0<sup>th</sup> (input) layer
- $y_j^{(0)} = x_j, j = 1 \dots D; y_0^{(k=1\dots N)} = x_0 = 1$

For layer  $k = 1 \dots N$

- For  $j = 1 \dots D_k$  D<sub>k</sub> is the size of the kth layer
  - $z_j^{(k)} = \sum_{i=0}^{N_k} w_{i,j}^{(k)} y_i^{(k-1)}$
  - $y_j^{(k)} = f_k(z_j^{(k)})$

Output:

- $Y = y_j^{(N)}, j = 1..D_N$

# Backward pass

Output layer ( $N$ ) :

– For  $i = 1 \dots D_N$

- $\frac{\partial Div}{\partial y_i} = \frac{\partial Div(Y, d)}{\partial y_i^{(N)}}$

- $\frac{\partial Div}{\partial z_i^{(N)}} = \frac{\partial Div}{\partial y_i^{(N)}} \frac{\partial y_i^{(N)}}{\partial z_i^{(N)}}$

For layer  $k = N - 1$  down to 0

– For  $i = 1 \dots D_k$

- $\frac{\partial Div}{\partial y_i^{(k)}} = \sum_j w_{ij}^{(k+1)} \frac{\partial Div}{\partial z_j^{(k+1)}}$

- $\frac{\partial Div}{\partial z_i^{(k)}} = \frac{\partial Div}{\partial y_i^{(k)}} \frac{\partial y_i^{(k)}}{\partial z_i^{(k)}}$

- $\frac{\partial Div}{\partial w_{ji}^{(k+1)}} = y_j^{(k)} \frac{\partial Div}{\partial z_i^{(k+1)}} \quad \text{for } j = 1 \dots D_{k+1}$

# Backpropagation in Pytorch

Pytorch can retro-compute gradients for any succession of operations, when you ask for it ! Use the `.backward()` method.

```
# Create differentiable tensor
x = torch.tensor(torch.arange(0,4), requires_grad=True)
z = x ** 2
b = torch.tensor(torch.zeros(4), requires_grad=True)
y = 5*z+x+b
# Calculate gradients (dy/dz=5,dz/dx=2x,dy/dx=10x+1,dy/db=1)
y.sum().backward()
print(y)
print(x.grad)
print(b.grad)
print(z.grad) # = None because that's an intermediate variable

tensor([ 0.,  6.,  22.,  48.])
tensor([ 1.,  11.,  21.,  31.])
tensor([ 1.,  1.,  1.,  1.])
None
```

For results, gradients are computed but not retained.

# Backpropagation in Pytorch

Warning : `.backward()` doesn't replace, but accumulate !

```
# Create a variable
x=torch.tensor(torch.arange(0,4), requires_grad=True)
# Differentiate
torch.sum(x**2).backward()
print(x.grad)
# Differentiate again (accumulates gradient)
torch.sum(x**2).backward()
print(x.grad)
# Zero gradient before differentiating
x.grad.data.zero_()
torch.sum(x**2).backward()
print(x.grad)
```

```
tensor([ 0.,  2.,  4.,  6.])
tensor([ 0.,  4.,  8., 12.])
tensor([ 0.,  2.,  4.,  6.])
```

# Neural networks in Pytorch

As you know, a neural network :

- Is a function connecting an input to an output
- Depends on (a lot of) parameters

In Pytorch, a neural network is a class that implements the base class `torch.nn.Module`.

You are provided with some pre-implemented networks, such as `torch.nn.Linear` which is a just a single-layer perceptron.

```
net = torch.nn.Linear(4,2)
```

# Neural networks in Pytorch

The `.forward()` method applies the function

```
x = torch.arange(0,4).float()
y = net.forward(x)
y = net(x) # Alternatively
print(y)
```

```
tensor([-0.4807, -0.7048])
```

The `.parameters()` method gives access to all of the network parameters

```
for param in net.parameters():
    print(param)
```

```
Parameter containing:
tensor([[-0.1506,  0.3700, -0.4565,  0.4557],
       [-0.4525, -0.0645, -0.3689,  0.4634]])
Parameter containing:
tensor([ 0.1931,  0.3287])
```

# Let's write a MLP

The worst way ever :

```
class MyNet0(nn.Module):
    def __init__(self,input_size, hidden_size, output_size):
        super(MyNetworkWithParams,self).__init__()
        self.layer1_weights = nn.Parameter(torch.randn(input_size,hidden_size))
        self.layer1_bias = nn.Parameter(torch.randn(hidden_size))
        self.layer2_weights = nn.Parameter(torch.randn(hidden_size,output_size))
        self.layer2_bias = nn.Parameter(torch.randn(output_size))

    def forward(self,x):
        h1 = torch.matmul(x,self.layer1_weights) + self.layer1_bias
        h1_act = torch.max(h1, torch.zeros(h1.size())) # ReLU
        output = torch.matmul(h1_act,self.layer2_weights) + self.layer2_bias
        return output
net=MyNet0(4,16,2)
```

All attributes of Parameter type become network parameters.

# Let's write a MLP

A better way :

```
class MyNet1(torch.nn.Module):
    def __init__(self,input_size, hidden_size, output_size):
        super().__init__()
        self.layer1 = torch.nn.Linear(input_size,hidden_size)
        self.layer2 = torch.nn.Sigmoid()
        self.layer3 = torch.nn.Linear(hidden_size,output_size)

    def forward(self, input_val):
        h = input_val
        h = self.layer1(h)
        h = self.layer2(h)
        h = self.layer3(h)
        return h
net = MyNet1(4,16,2)
```

You can use small networks inside big networks. Parameters of sub-networks will be "absorbed".

# Let's write a MLP

Even better :

```
def generate_net(input_size,hidden_size, output_size):
    return nn.Sequential(nn.Linear(input_size,hidden_size),
                        nn.ReLU(),
                        nn.Linear(hidden_size,output_size))

net = generate_net(4,16,2)
```

This is a shortcut for simple feed-forward networks (so all you need in HW1 part 2, but probably not in later homeworks)

# Let's write a MLP

Your own classes can also be used in bigger networks !

```
def relu_mlp(size_list):
    layers = []
    for i in range(len(size_list)-2):
        layers.append(nn.Linear(size_list[i],size_list[i+1]))
        layers.append(nn.ReLU())
    layers.append(nn.Linear(size_list[-2],size_list[-1]))
    return nn.Sequential(*layers)

my_big_MLP = nn.Sequential(
    relu_mlp([1000,512,512,256]),
    nn.Sigmoid(),
    relu_mlp([256,128,64,32,10]))
```

Allows a sort of "tree structure"

# Final layers and losses

You know you need a differentiable divergence (aka loss) between your output and the desired output. `torch.nn` provides a lot. Let's focus on Cross-Entropy

$$z_i = \sum_j w_{ij}x_j + b_i \quad \text{Final layer output}$$

$$y_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \quad \text{Softmax activation}$$

$$Div(y_i, d_i) = - \sum_i d_i \log(y_i) \quad \text{Cross-entropy loss } (d \text{ is one-hot desired output})$$

Popular in multi-class classification

# Final layers and losses

`torch.nn.CrossEntropyLoss` includes both the softmax and the loss criterion, and is stable (uses the log-softmax).

```
x = torch.tensor([np.arange(4), np.zeros(4),np.ones(4)]).float()
y = torch.tensor([0,1,0])
criterion = nn.CrossEntropyLoss()

output = net(x)
loss = criterion(output,y)
print(loss)

tensor(2.4107)
```

Contrary to before, the input  $x$  is 2-dimensional : it is a *batch* of input vectors (that's usually the case).

# How to train ?

---

The parameters have correct gradients now, but we still have to apply gradient descent (or something else ! More next week).

You must use an optimizer, subclass of `torch.optim.Optimizer`.

For gradient descent, you should use `torch.optim.SGD` (stands for Stochastic Gradient Descent, but here we'll use it as regular gradient descent)

# Use the optimizer

The optimizer is initialized with the parameters that you want to update.

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

The `.step()` method will apply gradient descent on all these parameters, using the gradients they contain.

```
optimizer.step()
```

# Use the optimizer

Remember that backpropagation in pytorch accumulates !

If you want to apply several iterations of gradient descent, gradients must be set to zero before each optimization step.

```
n_iter = 100
for i in range(n_iter):
    optimizer.zero_grad() # equivalent to net.zero_grad()
    output = net(x)
    loss = criterion(output,y)
    loss.backward()
    optimizer.step()
```

# A final code

---

Let's apply all of this to a task !

**Open the notebook *pytorch\_example.ipynb***

# Issues to pay attention to

## Tensor operations

- GPU+CPU computations
- Size mismatch in vector multiplications
- **\* is not matrix multiplication**

```
x = 2* torch.ones(2,2)
y = 3* torch.ones(2,2)
print(x * y)
print(x.matmul(y))
```

```
tensor([[ 6.,  6.],
        [ 6.,  6.]])
tensor([[ 12.,  12.],
        [ 12.,  12.]])
```

# Issues to pay attention to

## Tensor operations

### .view() is not transposition

```
x = torch.tensor([[1,2,3],[4,5,6]])
print(x)
print(x.t())
print(x.view(3,2))

tensor([[ 1,  2,  3],
       [ 4,  5,  6]])
tensor([[ 1,  4],
       [ 2,  5],
       [ 3,  6]])
tensor([[ 1,  2],
       [ 3,  4],
       [ 5,  6]])
```

# Issues to pay attention to

## Broadcasting

```
x = torch.ones(4,5)
y = torch.arange(5)
print(x+y)
y = torch.arange(4).view(-1,1)
print(x+y)
y = torch.arange(4)
print(x+y) # exception

tensor([[ 1.,  2.,  3.,  4.,  5.],
       [ 1.,  2.,  3.,  4.,  5.],
       [ 1.,  2.,  3.,  4.,  5.],
       [ 1.,  2.,  3.,  4.,  5.]])
tensor([[ 1.,  1.,  1.,  1.,  1.],
       [ 2.,  2.,  2.,  2.,  2.],
       [ 3.,  3.,  3.,  3.,  3.],
       [ 4.,  4.,  4.,  4.,  4.]])
```

---

```
RuntimeError                                     Traceback (most recent call last)
<ipython-input-47-8799a16e988f> in <module>()
      6 print(x+y)
      7 y = torch.arange(4)
----> 8 print(x+y) # exception
```

RuntimeError: The size of tensor a (5) must match the size of tensor b (4) at non-singleton dimension 1

# Issues to pay attention to

## GPU memory error

```
net = nn.Sequential(nn.Linear(2048,2048),nn.ReLU(),
                    nn.Linear(2048,2048),nn.ReLU(),
                    nn.Linear(2048,2048),nn.ReLU(),
                    nn.Linear(2048,2048),nn.ReLU(),
                    nn.Linear(2048,2048),nn.ReLU(),
                    nn.Linear(2048,2048),nn.ReLU(),
                    nn.Linear(2048,120))

x = torch.ones(256,2048)
y = torch.zeros(256).long()
net.cuda()
x.cuda()
crit=nn.CrossEntropyLoss()
out = net(x)
loss = crit(out,y)
loss.backward()
```

# Issues to pay attention to

```
net = nn.Linear(4,2)
x = torch.tensor([1,2,3,4])
y = net(x)
print(y)
```

What's the problem ?

# Issues to pay attention to

## Type error

```
net = nn.Linear(4,2)
x = torch.tensor([1,2,3,4])
y = net(x)
print(y)
```

RuntimeError: Expected object of type torch.LongTensor but found type torch.FloatTensor

```
x = x.float()
x = torch.tensor([1.,2.,3.,4.])
```

# Issues to pay attention to

```
class MyNet(nn.Module):
    def __init__(self,n_hidden_layers):
        super(MyNet,self).__init__()
        self.n_hidden_layers=n_hidden_layers
        self.final_layer = nn.Linear(128,10)
        self.act = nn.ReLU()
        self.hidden = []
        for i in range(n_hidden_layers):
            self.hidden.append(nn.Linear(128,128))

    def forward(self,x):
        h = x
        for i in range(self.n_hidden_layers):
            h = self.hidden[i](h)
            h = self.act(h)
        out = self.final_layer(h)
        return out
```

What's the problem ?

# Issues to pay attention to

## Parameter issue

```
class MyNet(nn.Module):
    def __init__(self,n_hidden_layers):
        super(MyNet,self).__init__()
        self.n_hidden_layers=n_hidden_layers
        self.final_layer = nn.Linear(128,10)
        self.act = nn.ReLU()
        self.hidden = []
        for i in range(n_hidden_layers):
            self.hidden.append(nn.Linear(128,128))

    def forward(self,x):
        h = x
        for i in range(self.n_hidden_layers):
            h = self.hidden[i](h)
            h = self.act(h)
        out = self.final_layer(h)
        return out
```

Hidden layers are not module parameters !

They will not be optimized.

# Issues to pay attention to

## Solution

```
class MyNet(nn.Module):
    def __init__(self,n_hidden_layers):
        super(MyNet,self).__init__()
        self.n_hidden_layers=n_hidden_layers
        self.final_layer = nn.Linear(128,10)
        self.act = nn.ReLU()
        self.hidden = []
        for i in range(n_hidden_layers):
            self.hidden.append(nn.Linear(128,128))
        self.hidden = nn.ModuleList(self.hidden)

    def forward(self,x):
        h = x
        for i in range(self.n_hidden_layers):
            h = self.hidden[i](h)
            h = self.act(h)
        out = self.final_layer(h)
        return out
```

# Pytorch debugging in one slide

If you have an error/bug in your code, or a question about pytorch :

- Always try to figure it out by yourself, that's how you learn the most ! For a strange behavior in your code, try printing outputs/inputs/parameters/errors
- Tons of online resources : great pytorch documentation, and basically every error is somewhere on StackOverflow
- Use piazza ! Check if someone else had your error, if not ask us
- Come to office hours !