

Your First Deep Learning Code

Recitation 2

11-785 Fall 2019

Recap

You have seen:

- You have seen the foundations of Python, NumPy and basics of Linear Algebra.
- You have been given the definition of what a neural network is and what it can model.

and

Today, we start learning how to **write Deep Learning code!**

Overview

1. Deep Learning Frameworks
2. Review Tensors (Math, Linear Algebra, indexing and slicing)
3. CPU and GPU Operations
4. Backpropagation
5. Neural Network Modules
6. Optimization and Loss
7. Saving and Loading
8. Common issues to look out for
9. Full NN Example in code

Logistics

You should be able to download the notebooks from the [course page](#).

- **Tutorial-pytorch**: Some code examples of what we will see today, with more details. You can look at it in parallel or later.
- **MNIST-example**: A complete pytorch example that we will walk-through at the end of this recitation.
- **Pytorch_example**: Another complete pytorch example for reference.

Logistics

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 be using these tools.

Conclusion: Pay Attention ;)

Overview

1. Deep Learning Frameworks

2. Review Tensors (Math, Linear Algebra, indexing and slicing)

3. CPU and GPU Operations

4. Backpropagation

5. Neural Network Modules

6. Optimization and Loss

7. Saving and Loading

8. Common issues to look out for

9. Full NN Example in code

Let's start with 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?

Framework		Main Purpose	User-Friendliness	Performance
	Developed by Google	Tensor board for effective data visualization, and static computation graphs.	Hard to get use to it Lots of online learning resources	Provides very efficient computations
	Developed by a Google Engineer	Used mainly as a front-end framework and to enable fast experimentation.	Easiest framework to Learn	High Performance as it Uses TensorFlow or Theano as the backend
	Developed by Graham Neubig (CMU) and his group	A specialized framework, and very handy when working NLP.	The framework provides detailed documentation	Very high performance in optimization

Framework		Main Usage	User-Friendliness	Performance
	Developed by Berkeley AI Research (BAIR)	Its main usage is in image processing , and deploying models for smart devices	Fair amount of online learning resources	Highly efficient when processing images, not so much for NLP and RNNs
	Developed by Adam Gibson, Skymind	Takes advantage of distributed frameworks (spark/Hadoop).	Guides and tutorials available. Might present issues in debugging	Can process huge data sets without reducing speed. Performance like Caffe
	Developed by University of Montreal*	Mainly used for computing CNNs, and RNNs, and as a backend for Keras.	Not so easy to use, but there are tutorials available	Moderate computation speed
	Develop by Facebook AI research group	Mainly used in scientific research , although use has increase due to its dynamic computational graphs and backprop.	Easier than TensorFlow and lots of support by the dev. Community.	Provides very efficient computations

Pytorch

We recommend Pytorch 0.4 or 1.0

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

LET'S START!

Overview

1. Deep Learning Frameworks
- 2. Review Tensors (Math, Linear Algebra, indexing and slicing)**
3. CPU and GPU Operations
4. Backpropagation
5. Neural Network Modules
6. Optimization and Loss
7. Saving and Loading
8. Common issues to look out for
9. Full NN Example in code

Data Operations

- Use the torch.Tensor class (~np.ndarray)

```
# 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 binds with it!
check [Recitation 0A](#)

Overview

1. Deep Learning Frameworks
2. Review Tensors (Math, Linear Algebra, indexing and slicing)
- 3. CPU and GPU Operations**
4. Backpropagation
5. Neural Network Modules
6. Optimization and Loss
7. Saving and Loading
8. Common issues to look out for
9. Full NN Example in code

Move Tensors to the GPU

For big computations, GPUs offer huge speedups!

```
1 # create a tensor
2 x = torch.rand(3,2)
3 # copy to GPU
4 y = x.cuda()
5 # copy back to CPU
6 z = y.cpu()
7 # get CPU tensor as numpy array
8 # cannot get GPU tensor as numpy array directly
9 try:
10     y.numpy()
11 except RuntimeError as e:
12     print(e)
```

Tensors can be copied between CPU and GPU. It is important that everything involved in a calculation is on the same device.

This portion of the tutorial may not work for you if you do not have a GPU available.

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-10-ad31a5261faa> in <module>
      9 # cannot get GPU tensor as numpy array directly
     10 try:
----> 11     y.numpy()
     12 except RuntimeError as e:
     13     print(e)
```

TypeError: can't convert CUDA tensor to numpy. Use Tensor.cpu() to copy the tensor to host memory first.

Move Tensors to the GPU

Operations between GPU and CPU tensors will fail. Operations require all arguments to be on the same device.

```
x = torch.rand(3,5) # CPU tensor
y = torch.rand(5,4).cuda() # GPU tensor
try:
    torch.mm(x,y) # Operation between CPU and GPU fails
except TypeError as e:
    print(e)
```

```
torch.mm received an invalid combination of arguments - got (torch.FloatTensor, torch.cuda.FloatTensor), but expected one of:
* (torch.FloatTensor source, torch.FloatTensor mat2)
  didn't match because some of the arguments have invalid types: (torch.FloatTensor or, torch.cuda.FloatTensor)
* (torch.SparseFloatTensor source, torch.FloatTensor mat2)
  didn't match because some of the arguments have invalid types: (torch.FloatTensor or, torch.cuda.FloatTensor)
```

Move Tensors to the GPU

Typical code should be compatible with both CPU & GPU. Include `if` statements or utilize helper functions so it can operate with or without the GPU.

```
1 # Put tensor on CUDA if available
2 x = torch.rand(3,2)
3 if torch.cuda.is_available():
4     x = x.cuda()
5     print(x, x.dtype)
6
7 # Do some calculations
8 y = x ** 2
9 print(y)
10
11 # Copy to CPU if on GPU
12 if y.is_cuda:
13     y = y.cpu()
14     print(y, y.dtype)
```



```
tensor([[0.1084, 0.5432],
        [0.2185, 0.3834],
        [0.3720, 0.5374]], device='cuda:0') torch.float32
tensor([[0.0117, 0.2951],
        [0.0477, 0.1470],
        [0.1383, 0.2888]], device='cuda:0')
tensor([[0.0117, 0.2951],
        [0.0477, 0.1470],
        [0.1383, 0.2888]]) torch.float32
```

Overview

1. Deep Learning Frameworks
2. Review Tensors (Math, Linear Algebra, indexing and slicing)
3. CPU and GPU Operations
- 4. Backpropagation**
5. Neural Network Modules
6. Optimization and Loss
7. Saving and Loading
8. Common issues to look out for
9. Full NN Example in code

Backpropagation

- You haven't seen it yet (mentioned last 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 the **backward pass** we compute a possible partial derivatives

Backpropagation in Pytorch

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

```
1 # Create differentiable tensor
2 x = torch.tensor(torch.arange(0,4), requires_grad=True)
3 print(x.dtype)
```

```
C:\Users\Wendy\Anaconda3\envs\DL\lib\site-packages\ipykernel_launcher.py:2: UserWarning: To copy construct from a tensor, it is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().detach().requires_grad_(True), rather than torch.tensor(sourceTensor).
```

```
-----
RuntimeError                                Traceback (most recent call last)
<ipython-input-29-e52532f7b323> in <module>
      1 # Create differentiable tensor
----> 2 x = torch.tensor(torch.arange(0,4), requires_grad=True)
      3 print(x.dtype)
```

```
RuntimeError: Only Tensors of floating point dtype can require gradients
```

For results, gradients are computed but not retained

Backpropagation in Pytorch

```
1 # Create differentiable tensor
2 x = torch.tensor(torch.arange(0,4), requires_grad=False)
3 print(x.dtype)
4 # Calculate y=sum(x**2)
5 y = x**2
6 # Calculate gradient (dy/dx=2x)
7 y.sum().backward()
8 # Print values
9 print(x)
10 print(y)
11 print(x.grad)
```

C:\Users\Wendy\Anaconda3\envs\DL\lib\site-packages\ipykernel_launcher.py:2: UserWarning: To copy construct from a tensor, it is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().detach().requires_grad_(True), rather than torch.tensor(sourceTensor).

torch.int64

RuntimeError Traceback (most recent call last)

<ipython-input-31-aabc06498b51> in <module>

```
5 y = x**2
6 # Calculate gradient (dy/dx=2x)
----> 7 y.sum().backward()
8 # Print values
9 print(x)
```

~\Anaconda3\envs\DL\lib\site-packages\torch\tensor.py in backward(self, gradient, retain_graph, create_graph)

```
116         products, Defaults to ``False``.
117         """
--> 118         torch.autograd.backward(self, gradient, retain_graph, create_graph)
119
120     def register_hook(self, hook):
```

~\Anaconda3\envs\DL\lib\site-packages\torch\autograd__init__.py in backward(tensors, grad_tensors, retain_graph, create_graph, grad_variables)

```
91     Variable._execution_engine.run_backward(
92         tensors, grad_tensors, retain_graph, create_graph,
--> 93         allow_unreachable=True) # allow_unreachable flag
94
95
```

RuntimeError: element 0 of tensors does not require grad and does not have a grad_fn

Backpropagation in Pytorch

Solution

```
1 # Create differentiable tensor
2 x = torch.tensor(torch.arange(0,4)).float().requires_grad_(True)
3 print(x.dtype)
4 # Calculate y=sum(x**2)
5 y = x**2
6 # Calculate gradient (dy/dx=2x)
7 y.sum().backward()
8 # Print values
9 print(x)
10 print(y)
11 print(x.grad)
```

C:\Users\Wendy\Anaconda3\envs\DL\lib\site-packages\ipykernel_launcher.py:2: UserWarning: To copy construct from a tensor, it is recommended to use `sourceTensor.clone().detach()` or `sourceTensor.clone().detach().requires_grad_(True)`, rather than `torch.tensor(sourceTensor)`.

```
torch.float32
tensor([0., 1., 2., 3.], requires_grad=True)
tensor([0., 1., 4., 9.], grad_fn=<PowBackward0>)
tensor([0., 2., 4., 6.]
```

Overview

1. Deep Learning Frameworks
2. Review Tensors (Math, Linear Algebra, indexing and slicing)
3. CPU and GPU Operations
4. Backpropagation
- 5. Neural Network Modules**
6. Optimization and Loss
7. Saving and Loading
8. Common issues to look out for
9. Full NN Example in code

Neural Networks in Pytorch

As you know a neural network:

- Is a function connecting an input to an output
- Depends on (a lots 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 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 an 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 an 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 subnetworks will be “absorbed”

Let's write an 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 feedforward networks.

So all you need in HW1 P2,, but probably not in later homeworks

Let's write an MLP

Your own classes can also be use 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”

Overview

1. Deep Learning Frameworks
2. Review Tensors (Math, Linear Algebra, indexing and slicing)
3. CPU and GPU Operations
4. Backpropagation
5. Neural Network Modules
- 6. Optimization and Loss**
7. Saving and Loading
8. Common issues to look out for
9. Full NN Example in code

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 (which is usually the case)

Use the optimizer

You must use an optimizer, subclass of **torch.nn.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()
```

Overview

1. Deep Learning Frameworks
2. Review Tensors (Math, Linear Algebra, indexing and slicing)
3. CPU and GPU Operations
4. Backpropagation
5. Neural Network Modules
6. Optimization and Loss
- 7. Saving and Loading**
8. Common issues to look out for
9. Full NN Example in code

Saving and Loading

```
1 # get dictionary of keys to weights using `state_dict`  
2 net = torch.nn.Sequential(  
3     torch.nn.Linear(28*28,256),  
4     torch.nn.Sigmoid(),  
5     torch.nn.Linear(256,10))  
6 print(net.state_dict().keys())
```

```
odict_keys(['0.weight', '0.bias', '2.weight', '2.bias'])
```

```
1 # save a dictionary  
2 torch.save(net.state_dict(),'test.t7')  
3 # load a dictionary  
4 net.load_state_dict(torch.load('test.t7'))
```

```
<All keys matched successfully>
```

Overview

1. Deep Learning Frameworks
2. Review Tensors (Math, Linear Algebra, indexing and slicing)
3. CPU and GPU Operations
4. Backpropagation
5. Neural Network Modules
6. Optimization and Loss
7. Saving and Loading
- 8. Common issues to look out for**
9. Full NN Example in code

Common Issues to Look Out For

Tensor Operations

- GPU + CPU
- 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.]])
```

Common Issues to Look Out For

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]])
```

Common Issues to Look Out For

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
```

Common Issues to Look Out For

GPU Memory Error

```
net = nn.Sequential(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()
```

Common Issues to Look Out For

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

Is there a problem?

What is it?...

Common Issues to Look Out For

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.])
```

Common Issues to Look Out For

```
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?

Common Issues to Look Out For

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

Common Issues to Look Out For

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 question about Pytorch:

- **Always try to figure it out by yourself first**, that's how you learn the most, for a strange behavior in your code, try printing the outputs/inputs/parameters/errors
- **Use the debugger:** `import pdb; pdb.set_trace()`
- **Tons of online resources**, great pytorch documentation, and basically every error is somewhere on stackoverflow.
- **Use Piazza.**- First check if someone else have encountered the same error, if not ask us!
- **Come to office hours.**

Overview

1. Deep Learning Frameworks
2. Review Tensors (Math, Linear Algebra, indexing and slicing)
3. CPU and GPU Operations
4. Backpropagation
5. Neural Network Modules
6. Optimization and Loss
7. Saving and Loading
8. Common issues to look out for
- 9. Full NN Example in code**

Pytorch Example

Open the notebook `MNIST_example.ipynb`