A Quick Introduction to PyTorch

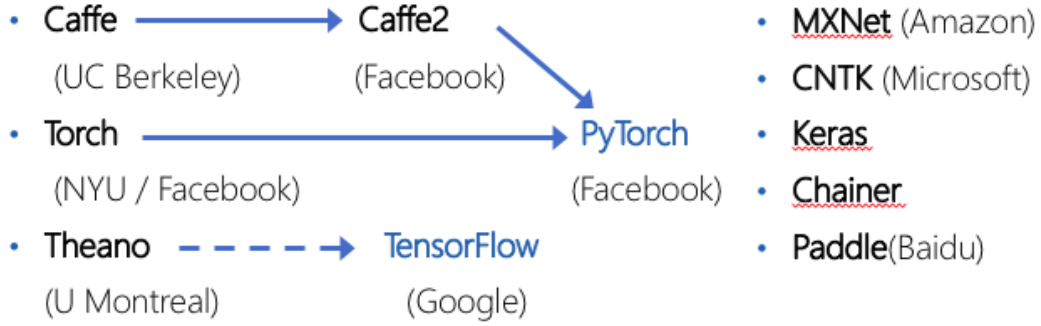
Instructor: Lei Wu [1]

Mathematical Introduction to Machine Learning

Peking University, Fall 2024

[1]School of Mathematical Sciences; **Center for Machine Learning Research**

Deep learning packages overview



Packages comparison

Criteria	TensorFlow	PyTorch	Keras	MxNet
Prototying	3	5	5	4
Ease of Use	3	5	5	4
Community Support	5	4	3.5	2.5
Efficiency	4	4	1	4
Ease of Learning	3	5	5	4
Industry	5	4 (After 1.0)	0	-

This table is slightly outdated. Take also a look at https://jax.readthedocs.io/en/latest/

About TensorFlow

TensorFlow™ is an open source software library for high performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices. Originally developed by researchers and engineers from the Google Brain team within Google's AI organization, it comes with strong support for machine learning and deep learning and the flexible numerical computation core is used across many other scientific domains.

- Tensor computation with strong GPU acceleration
- Automatic differentiation
- Lots of API for building deep neural networks.
- ...

PyTorch

PyTorch (website) is a python-based scientific computing package.

- NumPy-like tensor computations with strong GPU acceleration (A replacement of NumPy).
- Automatic differentiation.
- Lots of APIs for building and training neural networks.

JAX: High-Performance Array Computing

JAX is Autograd and XLA, brought together for high-performance numerical computing.

Familiar API

JAX provides a familiar NumPy-style API for ease of adoption by researchers and engineers.

Transformations

JAX includes composable function transformations for compilation, batching, automatic differentiation, and parallelization.

Run Anywhere

The same code executes on multiple backends, including CPU, GPU, & TPU

Core modules

Package	Description
torch	A Tensor library like Numpy, with strong GPU acceleration
torch.autograd	A dynamical automatic differentiation library that support all differentiable Tensor operations in torch
torch.nn	Neural networks library
torch.optim	Optimization package, providing SGD, Adam, L-BFGS, etc.
torch.utils	DataLoader

PyTorch tensor

- PyTorch tensors are very similar to NumPy ndarrays.
- But they have a device attribute: ‘cpu’, ‘cuda’, or ‘cuda:2’.
- They might require gradients for the automatic differentiation.

```
>>> t = torch.tensor([1,2,3], device='cpu',
...         requires_grad=False,dtype=torch.float32)
>>> print(t.dtype)
torch.float32
>>> print(t.device)
cpu
>>> print(t.requires_grad)
False
```

PyTorch tensor (Cont'd)

- The APIs of PyTorch tensor are almost the same as NumPy array, although there exist some differences.

```
# PyTorch
x = torch.randn(1000,1000)
y = torch.randn(1000,1000)

z = x * y
z = x * torch.tanh(z)
z = torch.matmul(x, y)

print(z.shape)
```

```
torch.Size([1000, 1000])
```

```
# NumPy
x = numpy.random.randn(1000,1000)
y = numpy.random.randn(1000,1000)

z = x * y
z = x * numpy.tanh(z)
z = numpy.matmul(x, y)

print(z.shape)
```

```
(1000, 1000)
```

- Refer to https://github.com/wkentaro/pytorch-for-numpy-users for detailed API comparisons of NumPy and PyTorch.

GPU acceleration

Let `x` be a torch tensor.

- `y = x.cuda()` creates a GPU tensor from a CPU tensor.

GPU acceleration

Let `x` be a torch tensor.

- `y = x.cuda()` creates a GPU tensor from a CPU tensor.
- `z = y.cpu()` creates a CPU tensor from GPU tensor.

GPU acceleration

Let `x` be a torch tensor.

- `y = x.cuda()` creates a GPU tensor from a CPU tensor.
- `z = y.cpu()` creates a CPU tensor from GPU tensor.
- Let `y1,y2` be two GPU tensor. Then, `y1+y2` is evaluated on GPU.

GPU acceleration

Let `x` be a torch tensor.

- `y = x.cuda()` creates a GPU tensor from a CPU tensor.
- `z = y.cpu()` creates a CPU tensor from GPU tensor.
- Let `y1,y2` be two GPU tensor. Then, `y1+y2` is evaluated on GPU.
- **Acceleration:** An example of matrix multiplication. In this example, GPU is 50 times faster than CPU.

```
# CPU
X = torch.randn(10000, 10000)
Z = torch.matmul(X,X)

# GPU
X = X.cuda()
Z = torch.matmul(X,X)
```

```
CPU execution time: 0.1181325912475586 secs
GPU execution time: 0.0023763840198516846 secs
```

GPU acceleration

Let `x` be a torch tensor.

- `y = x.cuda()` creates a GPU tensor from a CPU tensor.
- `z = y.cpu()` creates a CPU tensor from GPU tensor.
- Let `y1,y2` be two GPU tensor. Then, `y1+y2` is evaluated on GPU.
- **Acceleration:** An example of matrix multiplication. In this example, GPU is 50 times faster than CPU.

```
# CPU
X = torch.randn(10000, 10000)
Z = torch.matmul(X,X)

# GPU
X = X.cuda()
Z = torch.matmul(X,X)
```

```
CPU execution time: 0.1181325912475586 secs
GPU execution time: 0.0023763840198516846 secs
```

GPU acceleration

Let `x` be a torch tensor.

- `y = x.cuda()` creates a GPU tensor from a CPU tensor.
- `z = y.cpu()` creates a CPU tensor from GPU tensor.
- Let `y1,y2` be two GPU tensor. Then, `y1+y2` is evaluated on GPU.
- **Acceleration:** An example of matrix multiplication. In this example, GPU is 50 times faster than CPU.

```
# CPU
X = torch.randn(10000, 10000)
Z = torch.matmul(X,X)

# GPU
X = X.cuda()
Z = torch.matmul(X,X)
```

```
CPU execution time: 0.1181325912475586 secs
GPU execution time: 0.0023763840198516846 secs
```

When there are several GPUs in your computer, you can use the following code.

- `device=torch.device('cuda:1')` # use the first gpu
- `y = x.to(device)`

Interacting with NumPy

- `torch.from_numpy` and `torch.numpy`.
- The conversion between numpy array and torch tensor does not allocate new memory.

```
x_np = numpy.random.rand(2,2)
x_th = torch.from_numpy(x_np)
x_np_new = x_th.numpy()

x_th.fill_(1)
print(x_np,'\n')
print(x_np_new)
```

```
[[1. 1.]
 [1. 1.]]

[[1. 1.]
 [1. 1.]]
```

Computational graphs: forward propagation

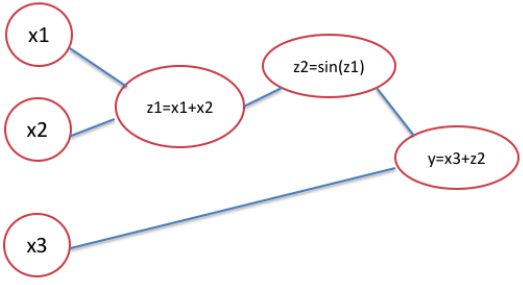
Before proceeding to the automatic differentiation, we need to understand an important concept: computational graph.

- A computational graph is a directed acyclic graph (DAG) that stores how the function is computed.

Computational graphs: forward propagation

Before proceeding to the automatic differentiation, we need to understand an important concept: computational graph.

- A computational graph is a directed acyclic graph (DAG) that stores how the function is computed.
- An example: $y = \sin(x_1 + x_2) + x_3$. The computational graph decomposes the computation into the compositions of a series of simple operations.



Computational graph: backward propagation

The computation graph is used to calculate the gradient.

- How to compute the gradients of those simple functions have been implemented in PyTorch.

Computational graph: backward propagation

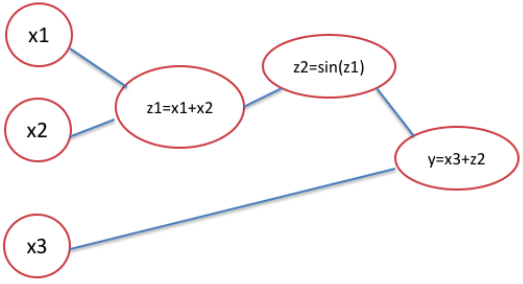
The computation graph is used to calculate the gradient.

- How to compute the gradients of those simple functions have been implemented in PyTorch.
- Then, by the chain rule, we can compute the gradients of a general function through the back-propogation of the computational graph.

Computational graph: backward propagation

The computation graph is used to calculate the gradient.

- How to compute the gradients of those simple functions have been implemented in PyTorch.
- Then, by the chain rule, we can compute the gradients of a general function through the back-propogation of the computational graph.
- Example: $\frac{\partial y}{\partial x_1} = \frac{\partial y}{\partial z_2}\frac{\partial z_2}{\partial z_1}\frac{\partial z_1}{\partial x_1} = 1 \times \cos(z_1) \times 1$. The value of z_1 is stored during the forward propagation, when the computational graph is constructed.



Autograd

- There are two ways to compute the gradients in PyTorch.

```
x1 = torch.tensor(2., requires_grad=True)
x2 = torch.tensor(3., requires_grad=True)
x3 = torch.tensor(5., requires_grad=True)

y = torch.sin(x1+x2) + x3

dy_dx = torch.autograd.grad(y, [x1,x2,x3])
print(dy_dx)
```

```
(tensor(0.2837), tensor(0.2837), tensor(1.))
```

```
x1 = torch.tensor(2., requires_grad=True)
x2 = torch.tensor(3., requires_grad=True)
x3 = torch.tensor(5., requires_grad=False)

y = torch.sin(x1+x2) + x3

y.backward()
print(x1.grad, x2.grad, x3.grad)
```

```
tensor(0.2837) tensor(0.2837) None
```

Autograd

- There are two ways to compute the gradients in PyTorch.

```
x1 = torch.tensor(2., requires_grad=True)
x2 = torch.tensor(3., requires_grad=True)
x3 = torch.tensor(5., requires_grad=True)

y = torch.sin(x1+x2) + x3

dy_dx = torch.autograd.grad(y, [x1,x2,x3])
print(dy_dx)
```

```
(tensor(0.2837), tensor(0.2837), tensor(1.))
```

```
x1 = torch.tensor(2., requires_grad=True)
x2 = torch.tensor(3., requires_grad=True)
x3 = torch.tensor(5., requires_grad=False)

y = torch.sin(x1+x2) + x3

y.backward()
print(x1.grad, x2.grad, x3.grad)
```

```
tensor(0.2837) tensor(0.2837) None
```

- The "`backward()`" function is more convenient since you don't need to specify the tensors. When calling `backward()` function, the chain rule is applied back to all the leaf tensors with `requires_grad=True`.

Autograd

- There are two ways to compute the gradients in PyTorch.

```
x1 = torch.tensor(2., requires_grad=True)
x2 = torch.tensor(3., requires_grad=True)
x3 = torch.tensor(5., requires_grad=True)

y = torch.sin(x1+x2) + x3

dy_dx = torch.autograd.grad(y, [x1,x2,x3])
print(dy_dx)
```

```
(tensor(0.2837), tensor(0.2837), tensor(1.))
```

```
x1 = torch.tensor(2., requires_grad=True)
x2 = torch.tensor(3., requires_grad=True)
x3 = torch.tensor(5., requires_grad=False)

y = torch.sin(x1+x2) + x3

y.backward()
print(x1.grad, x2.grad, x3.grad)
```

```
tensor(0.2837) tensor(0.2837) None
```

- The "`backward()`" function is more convenient since you don't need to specify the tensors. When calling `backward()` function, the chain rule is applied back to all the leaf tensors with `requires_grad=True`.
- A leaf tensor is a tensor you created directly, not a result of an operation. x_1, x_2, x_3 are the leaf tensors in the above example.

Autograd: backward function

- Notice that "`backward()`" function accumulate sthe gradients.

```
x = torch.ones(2, requires_grad=True)
y = x.sum()
y.backward()


z = x.pow(2).sum()
z.backward()

# the gradient is dy/dx + dz/dx
print(x.grad)
```

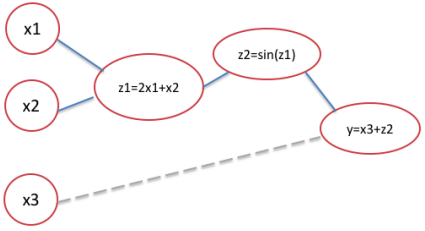
```
tensor([3., 3.])
```

Autograd (Cont'd)

- When `requires_grad=False`, the computational graph is not constructed.

```
x1 = torch.tensor(2., requires_grad=True)
x2 = torch.tensor(3., requires_grad=True)
x3 = torch.tensor(5., requires_grad=False)

y = torch.sin(x1+x2) + x3

y.backward()
print(x1.grad, x2.grad, x3.grad)
```

```
tensor(0.2837) tensor(0.2837) None
```

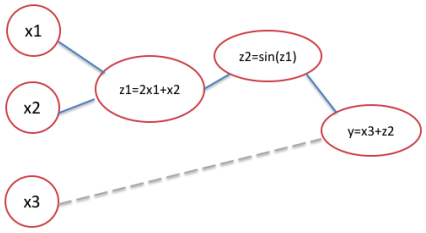


Autograd (Cont'd)

- When `requires_grad=False`, the computational graph is not constructed.

```
x1 = torch.tensor(2., requires_grad=True)
x2 = torch.tensor(3., requires_grad=True)
x3 = torch.tensor(5., requires_grad=False)

y = torch.sin(x1+x2) + x3

y.backward()
print(x1.grad, x2.grad, x3.grad)
```

```
tensor(0.2837) tensor(0.2837) None
```



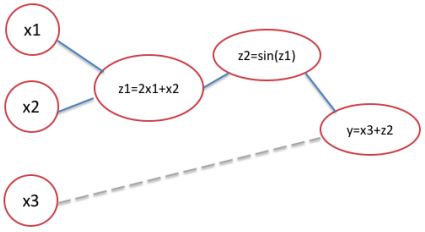
- If you want to detach a tensor from the graph, you can use `detach()`.

Autograd (Cont'd)

- When `requires_grad=False`, the computational graph is not constructed.

```
x1 = torch.tensor(2., requires_grad=True)
x2 = torch.tensor(3., requires_grad=True)
x3 = torch.tensor(5., requires_grad=False)

y = torch.sin(x1+x2) + x3

y.backward()
print(x1.grad, x2.grad, x3.grad)
```

```
tensor(0.2837) tensor(0.2837) None
```



- If you want to detach a tensor from the graph, you can use `detach()`.
- If you want to get a python number from a tensor, you can use `item()`.

```
print(x1)
print(x1.detach())
print(x1.item())
```

```
tensor(2., requires_grad=True)
tensor(2.)
2.0
```

Higher-order gradients

- PyTorch does no provide the direct APIs to compute higher-order gradients, such as Hessian. Since for a high dimensional function, the storage and computation of full higher-order gradients are extremely expensive.

Higher-order gradients

- PyTorch does no provide the direct APIs to compute higher-order gradients, such as Hessian. Since for a high dimensional function, the storage and computation of full higher-order gradients are extremely expensive.
- In most cases, we should try to avoid it using some tricks, e.g., Hessian-vector product:

$$\nabla^2 f(x)v = \nabla(v^\top \nabla f(x)).$$

We refer to https://iclr-blogposts.github.io/2024/blog/bench-hvp/ for more details.

Higher-order gradients

- PyTorch does no provide the direct APIs to compute higher-order gradients, such as Hessian. Since for a high dimensional function, the storage and computation of full higher-order gradients are extremely expensive.
- In most cases, we should try to avoid it using some tricks, e.g., Hessian-vector product:

$$\nabla^2 f(x)v = \nabla(v^\top \nabla f(x)).$$

  We refer to https://iclr-blogposts.github.io/2024/blog/bench-hvp/ for more details.
- **Remark:** The Hessian-vector product has many applications, e.g. computing the eigenvalues of the Hessian with power iteration.

Higher-order gradients (Cont'd)

- Let us consider the example to regularize the square norm of model's gradient: $G(f, x) = \|\nabla f(x)\|_2^2$. The objective function $cost = \hat{R}_n + \frac{\lambda}{n} \sum_{i=1}^{n} G(f, x_i)$.

Higher-order gradients (Cont'd)

- Let us consider the example to regularize the square norm of model's gradient: $G(f, x) = \|\nabla f(x)\|_2^2$. The objective function $cost = \hat{R}_n + \frac{\lambda}{n} \sum_{i=1}^{n} G(f, x_i)$.
- To optimize the above objective function, we need to compute the gradient of G:

$$\nabla G(f, x) = 2\nabla^2 f(x) \nabla f(x).$$

Higher-order gradients (Cont'd)

- Let us consider the example to regularize the square norm of model's gradient: $G(f,x) = \|\nabla f(x)\|_2^2$. The objective function $cost = \hat{R}_n + \frac{\lambda}{n}\sum_{i=1}^{n} G(f, x_i)$.
- To optimize the above objective function, we need to compute the gradient of G:

$$\nabla G(f,x) = 2\nabla^2 f(x)\nabla f(x).$$

- The right code is for $f(x) = x_1^2 + x_2^2 + x_3^2$.
- Call `autograd.grad` twice. Set `create_grad=True` at the first time, which indicates that the computational graph of computing gradients will be constructed. This graph will be used for the second the back-propogation to compute the second-order gradients.

```
x = torch.tensor([1.0,2.0,3.0],requires_grad=True)
f = (x*x).sum()

df = torch.autograd.grad(f, x, create_graph=True)
print(df[0])


G = df[0].pow(2).sum()
dG = torch.autograd.grad(G,x)
print(dG[0])
```

```
tensor([2., 4., 6.], grad_fn=<AddBackward0>)
tensor([ 8., 16., 24.])
```

Building neural network models

- The following codes build a two-layer ReLU network. The left define the network manually, while the right uses the APIs provided in PyTorch.

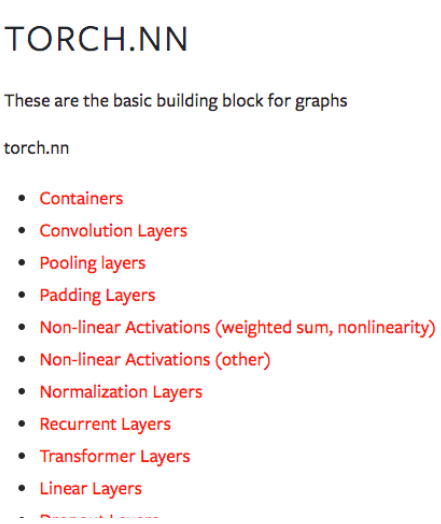
```python
import torch.nn as nn

class Net(nn.Module):
    def __init__(self, input_d, width, ouput_d):
        super(Net, self).__init__()
        self.B = torch.randn(input_d, width)
        self.c = torch.zeros(1, width)
        self.A = torch.randn(width, output_d)

        self.B = nn.Parameters(self.B)
        self.C = nn.Parameters(self.C)
        self.A = nn.Parameters(self.A)

    def forward(self, x):
        h = x.matmul(self.B) + self.C
        h = torch.relu(h)
        h = h.matmul(self.A)

        return h
```

```python
import torch.nn as nn

class Net(nn.Module):
    def __init__(self, input_d, width, ouput_d):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(input_d, width)
        self.fc2 = nn.Linear(width, output_d)

    def forward(self, x):
        h = self.fc1(x)
        h = torch.relu(h)
        h = self.fc2(x)

        return h
```

- `nn.Module` is a class to help decouple the process of input data and learnable parameters. The weights of a `nn.Module` are "`nn.Parameters`", which is similar to tensor but with "`requires_grad=True`".

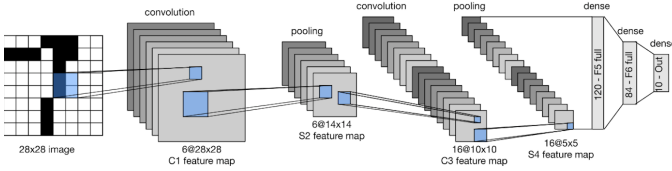
APIs for deep learning

- `torch.nn` contains many APIs for building neural network models, such as linear layer, convolutional layer, batch normalization layer, lots of loss functions, activation functions, and initializations.



An example of LeNet

```
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=3, padding=1)
        self.pool1 = nn.AvgPool2d(kernel_size=2)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
        self.pool2 = nn.AvgPool2d(kernel_size=2)

        self.classifier = nn.Sequential(
                        nn.Linear(16*5*5, 120), nn.ReLU(),
                        nn.Linear(120, 84), nn.ReLU(),
                        nn.Linear(84, 10))

    def forward(self, x):
        o1 = self.pool1(torch.relu(self.conv1(x)))
        o2 = self.pool2(torch.relu(self.conv2(o1)))
        o3 = o2.view(-1, 16*5*5)

        o4 = self.classifier(o3)
        return o4
```



- "`nn.sequential()`" provides an easy way to build up a sequential network (without any skip connections).
- The dimensions: x: Bx1x28x28, o1: Bx6x14x14, o2: Bx16x5x5, o3: Bx400, o4: Bx10.

Train a model

torch.optim implements many optimizers (e.g., SGD (+momentum), Adam, RMSprop, Rprop, L-BFGS) and learning rate schedulers.

- How do we use it? See the codes in the red rectangles.
- "`net.parameters()`" returns the parameters to be optimized.
- "`optimizer.zero_grad()`" sets the parameters' gradients to zero, since "`backward()`" will accumulate the gradients.
- "`optimizer.step()`" performs one-step update using the gradients stored in parameter tensors.

```
d = 2
nsamples = 1000
batchsize = 50
m = 100

X = torch.rand(nsamples, d)
y = torch.sin(X.sum(dim=1))

net = Net(d, m, 1)
optimizer = torch.optim.SGD(net.parameters(), lr=0.01, momentum=0.9)

loss_curve = []
for epoch in range(100):
    for i in range(0, nsamples, batchsize):
        x_batch, y_batch = X[i:(i+batchsize),:], y[i:(i+batchsize)]

        optimizer.zero_grad()
        y_pred = net(x_batch).squeeze()
        loss = (y_pred-y_batch).pow(2).mean()
        loss.backward()
        optimizer.step()

        loss_curve.append(loss.item())
```

Source code of “torch.optim”

The code of torch.optim is quite neat, you can take a look at it.
https://github.com/pytorch/pytorch/tree/master/torch/optim

Save and load a model

PyTorch uses Python's pickle utility to serialize the data.

- Save a model:

```
state = {'model_state': net.state_dict(),
    'optimizer_state': optimizer.state_dict()}
torch.save(state, 'lenet.pt')
```

Save and load a model

PyTorch uses Python's pickle utility to serialize the data.

- Save a model:

```
state = {'model_state': net.state_dict(),
    'optimizer_state': optimizer.state_dict()}
torch.save(state, 'lenet.pt')
```

- Restore a model:

```
net = LeNet()
optimizer = torch.optim.SGD(net.parameters(), lr=0.1)
checkpoint = torch.load('lenet.pt')

net.load_state_dict(checkpoint['model_state'])
optimizer.load_state_dict(checkpoint['optimizer_state'])
```

Save and load a model

PyTorch uses Python's pickle utility to serialize the data.

- Save a model:

```
state = {'model_state': net.state_dict(),
    'optimizer_state': optimizer.state_dict()}
torch.save(state, 'lenet.pt')
```

- Restore a model:

```
net = LeNet()
optimizer = torch.optim.SGD(net.parameters(), lr=0.1)
checkpoint = torch.load('lenet.pt')

net.load_state_dict(checkpoint['model_state'])
optimizer.load_state_dict(checkpoint['optimizer_state'])
```

- Important functions: "`torch.save()`, `torch.load()`, `load_state_dict()`, `state_dict()` ".

Save and load a model

PyTorch uses Python's pickle utility to serialize the data.

- Save a model:

```
state = {'model_state': net.state_dict(),
    'optimizer_state': optimizer.state_dict()}
torch.save(state, 'lenet.pt')
```

- Restore a model:

```
net = LeNet()
optimizer = torch.optim.SGD(net.parameters(), lr=0.1)
checkpoint = torch.load('lenet.pt')

net.load_state_dict(checkpoint['model_state'])
optimizer.load_state_dict(checkpoint['optimizer_state'])
```

- Important functions: "`torch.save()`, `torch.load()`, `load_state_dict()`, `state_dict()` ".
- Refer to https://pytorch.org/tutorials/beginner/saving_loading_models.html for more details.

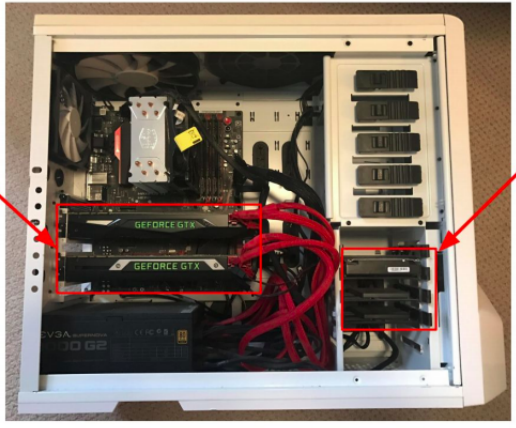
Reproducity

To keep our results reproducible, we need to fix the random seeds.

```
import numpy as np
np.random.seed(42)

import torch
torch.manual_seed(42)

# disable optimizations
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

- CUDA may use certain randomized methods for accelerations. So we must set it to use deterministic methods.

CPU / GPU Communication



If you aren't careful, training can bottleneck on reading data and transferring to GPU!

Solutions:

- Read all data into RAM
- Use SSD instead of HDD
- Use multiple CPU threads to prefetch data

Taken from standford cs231n.

DataLoader (Cont'd)

```python
class MyDataset:
    def __init__(self,X,y):
        self.X, self.y = X, y

    def __getitem__(self,idx):
        xi, yi = self.X[idx], self.y[idx] # return the data on the index
        return xi, yi

    def __len__(self):
        return self.X.size(0)     # return the data size
```

Define dataset

```python
dataset = MyDataset(torch.rand(100,2,30,30), torch.rand(100))
dataloader = torch.utils.data.DataLoader(
                            dataset=dataset,
                            batch_size = 50, # batch size
                            shuffle = False,
                            num_workers = 2  # use 2 cpu cores to parallel the dataloading
)
```

Define dataloader

```python
for X_batch, y_batch in dataloader:
    # use the batch data to do something
```

use the dataloader

"`torch.utils.data.DataLoader`" can be used to mini-batch data, shuffle data, and parallelize the loading process.

Other important related packages

- The **torchvision:** package consists of popular datasets, model architectures, and common image transformations for computer vision.
  - MNIST, CIFAR10/100, Fashion-MNIST, ImageNet.
  - Many large-scale pretrained models.
  - Image transformations (used for data argumentation).

Other important related packages

- The **torchvision:** package consists of popular datasets, model architectures, and common image transformations for computer vision.
  - MNIST, CIFAR10/100, Fashion-MNIST, ImageNet.
  - Many large-scale pretrained models.
  - Image transformations (used for data argumentation).
- The **torchaudio** package consists of I/O, popular datasets and common audio transformations.

Other important related packages

- The **torchvision:** package consists of popular datasets, model architectures, and common image transformations for computer vision.
  - MNIST, CIFAR10/100, Fashion-MNIST, ImageNet.
  - Many large-scale pretrained models.
  - Image transformations (used for data argumentation).
- The **torchaudio** package consists of I/O, popular datasets and common audio transformations.
- The **torchtext** package consists of data processing utilities and popular datasets for natural language.

Other important related packages

- The **torchvision:** package consists of popular datasets, model architectures, and common image transformations for computer vision.
  - MNIST, CIFAR10/100, Fashion-MNIST, ImageNet.
  - Many large-scale pretrained models.
  - Image transformations (used for data argumentation).
- The **torchaudio** package consists of I/O, popular datasets and common audio transformations.
- The **torchtext** package consists of data processing utilities and popular datasets for natural language.
- The **PyTorch Geometric** package consists of many methods for deep learning on graphs and other irregular structures, also known as geometric deep learning.

References

- **Tutorial:**
  - https://github.com/yunjey/pytorch-tutorial
  - https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html
- **Example:** https://github.com/pytorch/examples
- **Document:** https://pytorch.org/docs/stable/index.html
- **Source code:** https://github.com/pytorch/pytorch