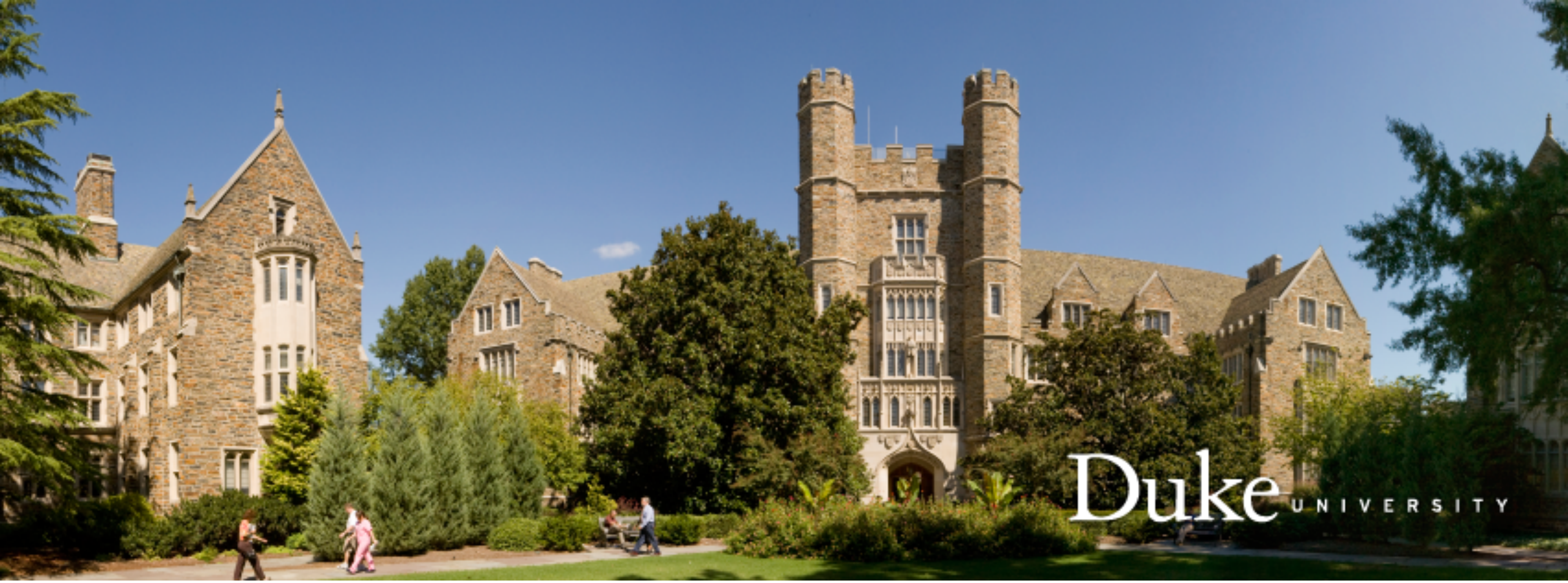

ECE 590-10/11

COMP ENG ML & DEEP NEURAL NETS

Lecture: NumPy/PyTorch Tutorial

Administrative: Lab #1

Lab #1 is released today.

- Deep Neural Network Visualization
- Warmup tutorial for PyTorch

Lab 1 is due on 11:59 pm, 09/11

You can either use the Jupyter Lab server or use your own PC to complete Lab #1.

Administrative: Jupyter Lab

- If you don’t have access to Jupyter Lab GPU container, contact a TA immediately.

Jupyter Lab Container Rule:

- Free GPU resource if you don’t need it.
- Shutdown the running notebook instance before you log out.
- Use **nvidia-smi** to monitor your GPU usage.
- Plan your work ahead.

Administrative: Prerequisites

- If you do not have object-oriented programming prerequisites or basic knowledge of Machine Learning, ask a TA for help immediately.

Overview

- Environment setup
- NumPy tutorial
- PyTorch tutorial

Environment setup

- Anaconda installation
- PyTorch installation

Environment setup

- Installing Anaconda

Linux

```
wget https://repo.anaconda.com/archive/Anaconda3-2019.07-Linux-x86_64.sh
chmod +x Anaconda3-2019.07-Linux-x86_64.sh
./Anaconda3-2019.07-Linux-x86_64.sh
```

macOSX

```
wget https://repo.anaconda.com/archive/Anaconda3-2019.07-MacOSX-x86_64.sh
chmod +x Anaconda3-2019.07-MacOSX-x86_64.sh
./Anaconda3-2019.07-MacOSX-x86_64.sh
```

Proceed with the instructions inside.

Note:
When you completed the installation on **macOSX**, remember to source ~/.bash_profile to activate conda environment.
When you completed the installation on **Linux**, remember to source ~/.bashrc to activate conda environment.

Environment setup

- PyTorch installation

Create and activate a PyTorch environment

```
conda create -n pytorch anaconda python=3.6
conda activate pytorch
```

Install the PyTorch package

For CUDA 9.2,

```
conda install pytorch torchvision cudatoolkit=9.2 -c pytorch
```

For CUDA 10.0,

```
conda install pytorch torchvision cudatoolkit=10.0 -c pytorch
```

For CPU only,

```
conda install pytorch torchvision cpuonly -c pytorch
```

Environment setup

- Validate the installation of PyTorch

```
Python 3.6.8 |Anaconda, Inc.| (default, Dec 29 2018, 19:04:46)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> torch.__version__
'1.1.0'
```

Note:
PyTorch 1.1.0 and above works well in this course.

NumPy tutorial

- Array
- Indexing
- Math operations
- Broadcasting
- Frequently used functions

NumPy tutorial

- Array

A NumPy array is a grid of values which have the same type. It is indexed by a tuple of non-negative integers. The number of dimensions is the rank of the array; the shape of an array is a tuple of integers giving the size of the array along each dimension.

```
import numpy as np            # Import the numpy library
# a is a python list.
a = [2,3,4,5]
# b is a numpy array, which has the same values and shapes as a.
b = np.array([2,3,4,5])
# c is also a numpy array, which has the same values and shapes as a.
c = np.array(a)
# d is a 2×4 numpy array with all zeros.
d = np.zeros((2,4))
# e is a numpy array with all zeros with the same shape as a.
e = np.zeros_like(a)
```

NumPy tutorial

- Array shape

```
import numpy as np            # Import the numpy library
# a is a numpy array.
a = np.array([[2,3],[4,5]])
# Get the shape of a.
print(a.shape)
Output: (2,2)
# Reshape a to 1×4 array.
a=np.reshape(a, (1,4))
print(a)
Output: array([[2, 3, 4, 5]])
```

NumPy tutorial

- Array indexing

Unlike python list, NumPy arrays can be sliced multidimensionally.

```
import numpy as np            # Import the NumPy library
# a is a python list.
a = [[2,3],[4,5]]
# b is a NumPy array, which has the same values and shapes as a.
b = np.array([[2,3],[4,5]])
# Slicing a list multi-dimensionally will lead to error
a[:1, :1]
# Error: list indices must be integers or slices, not tuple
# However, NumPy array can be sliced multi-dimensionally.
b[:1,:1]
# Output: array([[2]])
```

Source:http://cs231n.github.io/python-numpy-tutorial/

NumPy tutorial

- Boolean indexing

Boolean array indexing lets you pick out arbitrary elements of an array with minimal time cost.

```
import numpy as np
a = np.array([[1,2], [3, 4], [5, 6]])
# Find the elements of a that are greater than 2 and
bool_idx = (a > 2)
# return the corresponding boolean mask.
print(bool_idx)
Output: array([[False False] [ True True] [ True True]])
print(a[bool_idx])
Output: array([3 4 5 6])
# We can do all of the above in a single concise statement:
print(a[a > 2])
Output: [3 4 5 6]
```

Source:http://cs231n.github.io/python-numpy-tutorial/

NumPy tutorial

- Math operations

Most of the math operations operate elementwise on arrays.

```
import numpy as np
# Initialize two arrays
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)
# Elementwise sum. '+' is overloaded.
print(x + y)
print(np.add(x, y))
Output: [[ 6.0 8.0]  [10.0 12.0]]
# Elementwise product; both produce the array
print(x * y)
print(np.multiply(x, y))
Output: [[ 5.0 12.0] [21.0 32.0]]
# Elementwise square root; produces the array
print(np.sqrt(x))
Output: [[ 1. 1.41421356] [ 1.73205081 2. ]]
```

Source:http://cs231n.github.io/python-numpy-tutorial/

NumPy tutorial

• Broadcasting

Broadcasting is a powerful mechanism that allows NumPy to work with arrays of different shapes when performing arithmetic operations.

```
import numpy as np
# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v                    # v is expanded to [[1,0,1],[1,0,1],[1,0,1],[1,0,1]]
# Add v to each row of x using broadcasting
print(y)
Output: [[ 2 2 4] [ 5 5 7] [ 8 8 10] [11 11 13]]
```

Use broadcasting instead of loops to carry on matrix operations.

Source:http://cs231n.github.io/python-numpy-tutorial/

NumPy tutorial

- Frequently used functions

```
import numpy as np
```

Function	Description
np.concatenate	Concatenate two arrays
np.random.random	Generate random arrays
np.random.permutation	Generate random sequence
np.sum/np.mean/np.std	Get sum/mean/variance of an array
np.argsort	Get the indices that would sort an array
np.random.choice	Randomly choose elements from an array
np.min/np.max	Get the max/min value of an array

PyTorch tutorial

- PyTorch basics
- Building DNN block
- Training setup
- Case study: Dynamic Net
- Frequently used functions

What is PyTorch?

- A replacement for NumPy to use the power of GPUs.
- A deep learning research platform that provides maximum flexibility and speed.

PyTorch vs TensorFlow

	PyTorch	**TensorFlow**
Graph Definition Method	Dynamic	Static (1.x) Dynamic (2.0)
Visualization	Use external tools, not good enough	Native Tensorboard, detailed view of deep learning deployment
Debugging difficulty	Easy	Hard
Data parallelism	Easy to implement	Requires more careful thought, but more efficient.

Imports

```
import torch.nn.functional as F  # functions such as activations are here.
import torch.nn as nn            # pytorch neural network modules
import torchvision               # pytorch computer vision model zoo
```

Tensors

- PyTorch uses **Tensors** to hold weights and activations during neural network computation.
- **Tensors** are similar to NumPy’s ndarrays, with the addition being that Tensors can also be used on a GPU to accelerate computing.

Tensors: Example

```
from __future__ import print_function
import torch
# Create a 5x3 matrix, uninitialized:
x = torch.empty(5, 3)
# Create a random initialized 5x3 matrix:
x = torch.rand(5, 3)
# Create a matrix filled of zeros with dtype long:
x = torch.zeros(5, 3, dtype=torch.long)
# Output for visualization
print(x)
```

Out:
Tensor([[0,0,0], [0,0,0],
[0,0,0], [0,0,0], [0,0,0]])

Operation

- Arithmetic operations are the same as NumPy operations.

For example, Tensor Addition

torch.add(x,y) and x+y are equivalent.

- Use torch.view to reshape a tensor.

```
x = torch.randn(4, 4)
y = x.view(16)
z = x.view(-1, 8) # the size -1 is inferred from other dimensions
print(x.size(), y.size(), z.size())
```

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

Autograd

- PyTorch provides automated differentiation for all operations on Tensors.

```
x = torch.ones(2, 2, requires_grad=True)
y = x + 2
z = y * y * 3
out = z.mean()
print(z, out)
# Use autograd to compute gradient
out.backward()
print(x.grad)
```

```
Out:
tensor([[27., 27.], [27., 27.]],
grad_fn=<MulBackward0>) tensor(27.,
grad_fn=<MeanBackward0>)

Out:
tensor([[4.5000, 4.5000], [4.5000,
4.5000]])
```

Building a block: Template

- **Custom PyTorch Block**

```
import torch.nn as nn
class Block(nn.Module):
    def __init__(self):
        super(Block, self).__init__()
        ...

    def forward(self, x):
        ...
```

- Each block must inherit parent class **nn.Module** to be recognized as a component of DNN in PyTorch.
- Variables are defined and initialized in the __init__ method.
- Each block must have a method called forward. Computational graph is constructed in the forward method.

Building a block: Example

- Building a LeNet-5 for MNIST/CIFAR-10

```
import torch.nn as nn
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 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):
        out = F.relu(self.conv1(x))
        out = F.max_pool2d(out, 2)
        out = F.relu(self.conv2(out))
        out = F.max_pool2d(out, 2)
        out = out.view(out.size(0), -1)
        out = F.relu(self.fc1(out))
        out = F.relu(self.fc2(out))
        out = self.fc3(out)
        return out
```

Layer definitions are defined in the __init__ function. Weights for convolutional/affine layers are initialized.

A neural network is constructed. That means connections between layers and the flow of tensors are defined here.

Building a block: Modularization

```
import torch.nn as nn
class Block(nn.Module):
   def __init__(self):
      super(Block, self).__init__()
      ...
      pass

   def forward(self, x):
      ...
      pass

class Net(nn.Module):
   def __init__(self):
      super(Net, self).__init__()
      self.layer = Block()
      ...
      pass

   def forward(self, x):
      output = self.layer()
      ...
      return output
```

Important: use super to initialize the parent class.

As long as defined blocks are following the template, we can construct larger modules using predefined blocks.

Note:
It is always a good practice to modularize as much as possible, especially for neural networks with replicated structures. (VGG-16, ResNet etc.) n addition, modularization relieves the trouble of debugging and makes the code more readable.

Data processing

- PyTorch has many built-in functions for data preprocessing.

First, we should import the essential modules for transformation:

```
import torchvision.transforms as transforms
```

Use functions from **torchvision.transforms** to do data preprocessing as well as data augmentation.

Function Name	Description
torchvision.transforms.ToTensor	Converting an NumPy array to a torch tensor. Also, normalize the given array to range [0,1].
torchvision.transforms.Normalize	Normalize the input with given mean and standard deviation.
torchvision.transforms.RandomHorizontalFlip	Randomly do a horizontal flip on the input image. This is for data augmentation.
torchvision.transforms.RandomCrop	Randomly crop an image to target size. This function will first add a given padding to the image, then randomly crop it to get the target image.

Data processing

- Example: Data preprocessing on CIFAR-10 dataset

```
import torchvision.transforms as transforms
# Data transformation for training dataset
transform_train = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])
# Data transformation for testing dataset
transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])
```

Note: preprocessing for training/testing dataset should be consistent. For example, use the same normalization value for both training and validation dataset.

Data loader

- PyTorch has native data loaders for most computer vision tasks.

TORCHVISION.DATASETS

All datasets are subclasses of `torch.utils.data.Dataset` i.e, they have `__getitem__` and `__len__` methods implemented. Hence, they can all be passed to a `torch.utils.data.DataLoader` which can load multiple samples parallelly using `torch.multiprocessing` workers. For example:

```
imagenet_data = torchvision.datasets.ImageNet('path/to/imagenet_root/')
data_loader = torch.utils.data.DataLoader(imagenet_data,
                                          batch_size=4,
                                          shuffle=True,
                                          num_workers=args.nThreads)
```

Data loader

- Native data loader for CIFAR-10 dataset

```
import torchvision

trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True,
transform=transform_train)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=args.batch_size,
shuffle=True, num_workers=16)

testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True,
transform=transform_test)
testloader = torch.utils.data.DataLoader(testset, batch_size=100, shuffle=False,
num_workers=16)
```

Note: We will use an alternative dataset loader in Lab 2.

Loss function

- For most of the problems here, we will use the cross-entropy loss function.

```
import torch.nn as nn
criterion = nn.CrossEntropyLoss()
```

- We recommend looking at the source code of PyTorch. This loss function takes two arguments as the input:

```
class CrossEntropyLoss(_WeightedLoss):
    def __init__(self, weight=None, size_average=None, ignore_index=-100,
            reduce=None, reduction='mean'):
        super(CrossEntropyLoss, self).__init__(weight, size_average, reduce, reduction)
        self.ignore_index = ignore_index

    def forward(self, input, target):
        return F.cross_entropy(input, target, weight=self.weight,
                        ignore_index=self.ignore_index, reduction=self.reduction)
```

- Therefore, the correct way to use cross entropy loss here is to call

```
loss = nn.CrossEntropyLoss(outputs, targets)
OR
loss = criterion(outputs, targets)
```

Loss function

- Now let's take a deeper look into the documentation.

CLASS `torch.nn.CrossEntropyLoss(weight=None, size_average=None, ignore_index=-100, reduce=None, reduction='mean')` [SOURCE]

This criterion combines `nn.LogSoftmax()` and `nn.NLLLoss()` in one single class.

It is useful when training a classification problem with C classes. If provided, the optional argument `weight` should be a 1D *Tensor* assigning weight to each of the classes. This is particularly useful when you have an unbalanced training set.

The *input* is expected to contain raw, unnormalized scores for each class.

input has to be a Tensor of size either $(minibatch, C)$ or $(minibatch, C, d_1, d_2, ..., d_K)$ with $K \geq 1$ for the *K*-dimensional case (described later).

This criterion expects a class index in the range $[0, C-1]$ as the *target* for each value of a 1D tensor of size *minibatch*; if *ignore_index* is specified, this criterion also accepts this class index (this index may not necessarily be in the class range).

The loss can be described as:

$$\text{loss}(x, class) = -\log\left(\frac{\exp(x[class])}{\sum_j \exp(x[j])}\right) = -x[class] + \log\left(\sum_j \exp(x[j])\right)$$

or in the case of the `weight` argument being specified:

$$\text{loss}(x, class) = weight[class]\left(-x[class] + \log\left(\sum_j \exp(x[j])\right)\right)$$

The losses are averaged across observations for each minibatch.

Can also be used for higher dimension inputs, such as 2D images, by providing an input of size $(minibatch, C, d_1, d_2, ..., d_K)$ with $K \geq 1$, where K is the number of dimensions, and a target of appropriate shape (see below).

DO NOT use softmax activation in the last layer. The softmax operation is fused into cross entropy loss in PyTorch.

Note:
In situation of any confusion, always look at source code/documentation of PyTorch.

Optimizer

Optimizer is what we use to train the neural network. The optimizers are defined in **torch.optim** package.

Optimizer Name	**Description**
torch.optim.Adadelta	Implements Adadelta algorithm.
torch.optim.Adagrad	Implements Adagrad algorithm.
torch.optim.Adam	Implements Adam algorithm.
torch.optim.ASGD	Implements Averaged Stochastic Gradient Descent.
torch.optim.RMSprop	Implements RMSprop algorithm.
torch.optim.SGD	Implements stochastic gradient descent (optionally with momentum).

We will use **torch.optim.SGD** under most of the cases.

Optimizer

Optimizer should be defined outside the computational graph before the training process.

Suppose we have defined and instantiated a neural network called **net**.

Define an optimizer using SGD with momentum algorithm

```
import torch.optim as optim

optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.9,
weight_decay=1e-4)
```

To achieve the best performance, it is recommended to leave all of the parameters in their default settings. We will talk about hyperparameter tuning in the next few lectures.

Optimizer

- Step 1: Zero the gradients.

```
optimizer.zero_grad()
```

- Step 2: Backward propagation

```
loss.backward()
```

- Step 3: Take the optimization step

```
optimizer.step()
```

Optimizer

- Schedule the learning rate in the optimizer

```
import torch.optim as optim
optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.9,
weight_decay=1e-4)
new_lr = 0.1
for param_group in optimizer.param_groups:
   param_group['lr'] = new_lr
```

- Or, use the learning rate scheduler in torch.optim.lr_scheduler.

```
import torch.optim as optim
optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.9,
weight_decay=1e-4)
# Apply 0.1 learning rate decay for every 30 epochs.
optimizer = optim.lr_scheduler.StepLR(optimizer,step_size=30,
gamma=0.1)
```

Case study: Dynamic Net

- We are going to create a neural network with dynamic depth. That means, we will randomly choose 0-3 hidden layers for forward propagation. Note that weights for hidden layers are shared despite of the number of hidden layers chosen in forward/backward propagation.

Case study: Dynamic Net

- Import essentials

```
import torch
import random
```

- For more complicated neural architecture design, it is recommended to import the following packages:

```
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import torch.backends.cudnn as cudnn
import torchvision
import torchvision.transforms as transforms
```

Case study: Dynamic Net

- Create the dynamic net module

```
class DynamicNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(DynamicNet, self).__init__()
        self.input_linear = torch.nn.Linear(D_in, H)
        self.middle_linear = torch.nn.Linear(H, H)
        self.output_linear = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h_relu = self.input_linear(x).clamp(min=0)
        for _ in range(random.randint(0, 3)):
            h_relu = self.middle_linear(h_relu).clamp(min=0)
        y_pred = self.output_linear(h_relu)
        return y_pred
```

Important: initialize the parent class.

Initialize layer/weight configuration

Specify the connection relationship.

Randomly choose 0-3 hidden layers.

Case study: Dynamic Net

- Generate toy data

```
# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold inputs and outputs
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
```

Case study: Dynamic Net

- Instantiate model, create loss function and optimization op.

```
# Construct our model by instantiating the class defined above
model = DynamicNet(D_in, H, D_out)

# Construct our loss function and an Optimizer. Training this strange model
with vanilla stochastic gradient descent is tough, so we use momentum
criterion = torch.nn.MSELoss(reduction='sum')
#Use mean squared error as loss function.
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4, momentum=0.9)
```

Since the data is not normalized, we use a smaller learning rate 1e-4 to prevent gradient explosion. Usually, if we use a normalized data, default learning rate parameter for momentum optimizer should be set to 1e-2.

Case study: Dynamic Net

- Begin the forward/backward pass

```
for t in range(500):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x)

    # Compute and print loss
    loss = criterion(y_pred, y)
    print(t, loss.item())

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Advanced PyTorch topics

- Train/Eval mode
- Training on GPU
- Model load/save
- Data parallel
- Learning rate scheduler

Train/Evaluation mode

- Some neural network layers (e.g. dropout, batch normalization) have completely different behavior during training and evaluation. It is important to set the correct mode for both training and evaluation.

```
import torch
...
# Construct our model by instantiating the class defined above
model = DynamicNet(D_in, H, D_out)
...
# Set to train mode before running the training process
model.train()
... # Training code
# Set to eval mode before running the evaluation process
model.eval()
... # Evaluation code
```

Training on GPU

- GPU gives a considerable acceleration on training speed compared to CPUs.

Deploy models on GPU

```
import torch
# Find if GPU device is available
device = 'cuda' if torch.cuda.is_available() else 'cpu‘
# Construct our model by instantiating the class defined above
model = DynamicNet(D_in, H, D_out)
# Copy to CUDA device. This is very important.
model.to(device)
```

Training on GPU

- Don’t forget to copy the inputs to GPU devices during training!

```
for t in range(500):
    # Copy inputs to GPU. This is very important.
    x, y = x.to(device), y.to(device)
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x)

    # Compute and print loss
    loss = criterion(y_pred, y)
    print(t, loss.item())

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Model load/save

- Save/Load the whole model

```
import torch
# Construct our model by instantiating the class defined above
model = DynamicNet(D_in, H, D_out)
# Configure the optimizer and training
...
# Save model
torch.save(model, “dynamic_net.pth”)
```

Note: the model is serialized in a pickle object. The disadvantage of this approach is that the serialized data is bound to the specific classes and the exact directory structure used when the model is saved.

Model load/save

- Load the whole model

```
import torch
# Load model
model = torch.load("dynamic_net.pth")
```

Note: the model is serialized in a pickle object. The disadvantage of this approach is that the serialized data is bound to the specific classes and the exact directory structure used when the model is loaded.

Model load/save

- Save the weight parameters of a model

```
import torch
# Construct our model by instantiating the class defined above
model = DynamicNet(D_in, H, D_out)
# Configure the optimizer and training
...
# Save weight parameters
torch.save(model.state_dict(), “dynamic_net.pt”)
```

Note: This approach is better because weight parameters do not rely on specific classes or code structures during the saving process.

Model load/save

- Load the weight parameters of a model

```
import torch
# Construct our model by instantiating the class defined above
model = DynamicNet(D_in, H, D_out)
# Configure the optimizer and training
...
# Load weight parameters
model.load_state_dict(torch.load("dynamic_net.pt"))
```

Note: This approach is better because weight parameters do not rely on specific classes or code structures during the saving process.

Data parallel

- Much more accelerations can be achieved using Multiple GPU cards.

```
import torch
# Find if GPU device is available
device = 'cuda' if torch.cuda.is_available() else 'cpu‘
# Construct our model by instantiating the class defined above
model = DynamicNet(D_in, H, D_out)
# Copy to CUDA device. This is very important.
model.to(device)
# Apply the data parallelization semantics.
model = torch.nn.DataParallel(model)
```

Note: Due to limited GPU resources we have for this class, using Data Parallel is prohibited on the JupyerLab server.

Learning rate schedule

- Use the learning rate scheduler in **torch.optim.lr_scheduler** package.

Example: Schedule an exponential learning rate decay

CLASS `torch.optim.lr_scheduler.StepLR(optimizer, step_size, gamma=0.1, last_epoch=-1)` [SOURCE]

Sets the learning rate of each parameter group to the initial lr decayed by gamma every step_size epochs. When last_epoch=-1, sets initial lr as lr.

Parameters

- **optimizer** (*Optimizer*) – Wrapped optimizer.
- **step_size** (*int*) – Period of learning rate decay.
- **gamma** (*float*) – Multiplicative factor of learning rate decay. Default: 0.1.
- **last_epoch** (*int*) – The index of last epoch. Default: -1.

Example

```
>>> # Assuming optimizer uses lr = 0.05 for all groups
>>> # lr = 0.05     if epoch < 30
>>> # lr = 0.005    if 30 <= epoch < 60
>>> # lr = 0.0005   if 60 <= epoch < 90
>>> # ...
>>> scheduler = StepLR(optimizer, step_size=30, gamma=0.1)
>>> for epoch in range(100):
>>>     train(...)
>>>     validate(...)
>>>     scheduler.step()
```

We will see the power of learning rate schedule in the next a few lectures.

Reference

- **NumPy tutorial**

http://cs231n.github.io/python-numpy-tutorial/

- **PyTorch master documentation**

https://pytorch.org/docs/stable/index.html