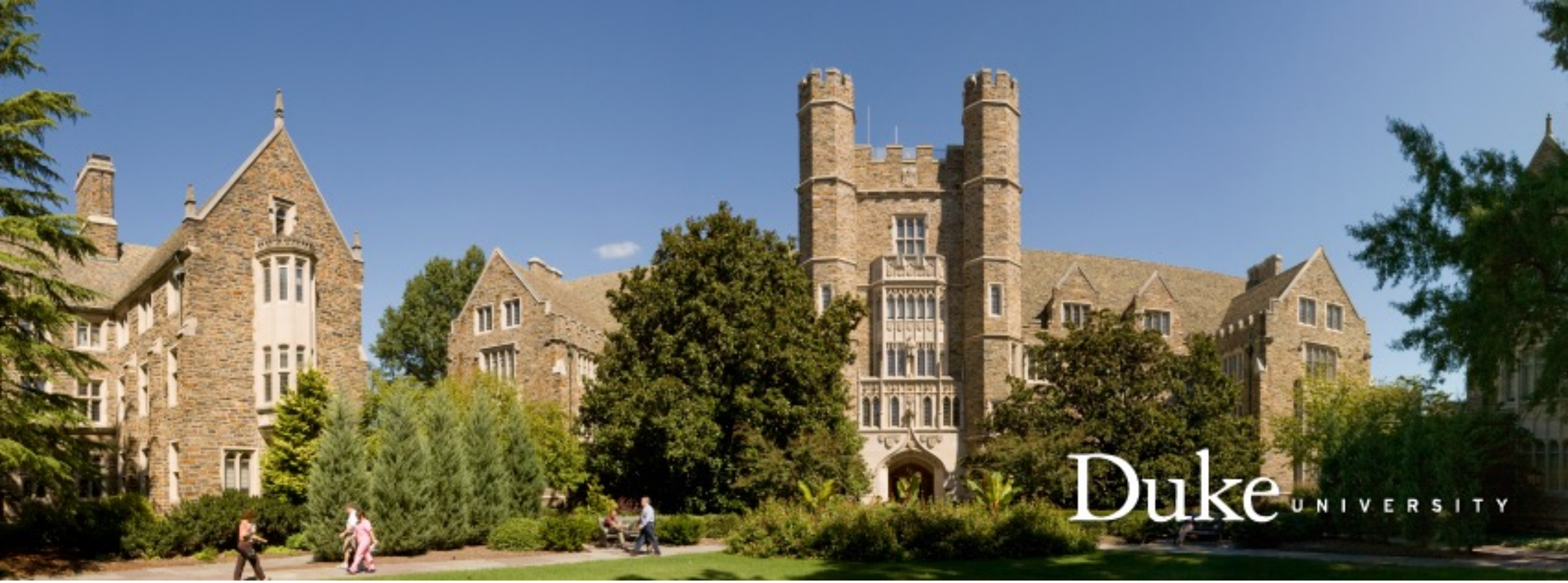

ECE 661

COMP ENG ML & DEEP NEURAL NETS

TUTORIAL: NUMPY/PYTORCH

YIRAN CHEN, FALL 2022

COLAB ENVIRONMENT

Overview

- Google Colaboratory is an online interactive python IDE that provides free GPU resources
  - Already has most scientific packages (e.g., NumPy, PyTorch) installed
- You can think of it as a Jupyter Notebook running in your Google Drive
- Workflow:
  - Download .ipynb file from Sakai
  - Upload it to a folder in your Google Drive
  - From within Google Drive, open the file with Colab

Local install

- Alternatively, you could install PyTorch on your local machine (macOS or Linux)
  - https://pytorch.org/get-started/locally/
  - If you do not have a GPU w/ CUDA, your training will be very slow
  - We recommend students use Colab to leverage GPUs when training

NUMPY TUTORIAL

What is Numpy?

- NumPy is a fundamental Python package for scientific computing on CPU
- Provides us a user-friendly interface to perform fast matrix computations via vectorized backend code
- The core is the ndarray object, which is an n-dimensional array

NumPy tutorial

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

NumPy tutorial

- NumPy Array

A NumPy array is a grid of values which have the same type. A NumPy array is indexed by a tuple of non-negative integers.

```
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)
```

The **rank** of the array is the number of dimensions.

NumPy tutorial

- Shape of NumPy 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 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

- Indexing of NumPy array

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 of NumPy array

Boolean array indexing allows us to select arbitrary elements of an array with maximal efficiency.

```
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 of NumPy arrays

Most math operations operate **element-wise** on NumPy 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)
# Element-wise sum. '+' is overloaded.
print(x + y)
print(np.add(x, y))
Output: [[ 6.0 8.0]  [10.0 12.0]]
# Element-wise product; both produce the array
print(x * y)
print(np.multiply(x, y))
Output: [[ 5.0 12.0] [21.0 32.0]]
# Element-wise 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 on NumPy array

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]]
```

Note: We recommend you use broadcasting carry on matrix operations. Avoid using loops as it is quite inefficient.

NumPy tutorial

- Always remember to import NumPy:

```
import numpy as np
```

Frequently used functions:

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

Refer to NumPy documentation for more details:
https://numpy.org/doc/1.19/

PYTORCH TUTORIAL

PyTorch tutorial

- PyTorch basics
- Setting up training pipelines
- Case study: Dynamic Net
- Advanced PyTorch functions

What is PYTORCH?

- A GPU-version of the NumPy library. The NumPy way of processing arrays can be reused.
- A deep learning framework that provides maximum flexibility and speed on training deep neural networks on various tasks.
- Deep learning models are represented as computation graphs in PyTorch.

Imports

- Essential imports for PyTorch utilities.

```
import torch.nn.functional as F
import torch.nn as nn
import torchvision
```

torch.nn.functional: Contains all functions such as non-linear activation functions.
torch.nn: Contains all neural network modules.
torchvision: PyTorch computer vision utilities.

- Other utilities

torch.optim: Contains all PyTorch-supported optimizers.
torch.utils.data: Data loader.
torch.backends: PyTorch backend.

Tensors

- PyTorch uses **Tensors** to hold weights and activations during neural network computation.
- **Tensors** are similar to NumPy’s ndarrays.
- **Tensors** can also be processed and executed on a GPU to accelerate computing.

Generating PyTorch Tensors

- A toy 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]])
```

Tensors are placed on **CPU** by default. Use **x = x.cuda()** to place tensor x on GPUs.

PyTorch vs. NumPy Tensors

How to convert PyTorch and NumPy tensors from one to another?

- Convert NumPy Tensor to PyTorch Tensor

```
a = np.ones(5)
b = torch.from_numpy(a)
```

- Convert PyTorch Tensor to NumPy Tensor

```
a = torch.ones(5)
b = a.numpy()
```

- If the designated torch tensor is on **GPU**, you may use:

```
a = torch.ones(5)
b = a.cpu().numpy()
```

Best practices on using NumPy and PyTorch

- Always refer to the document when you have confusion about using library functions.
- Avoid using loops. Instead, try vectorized representations to make your code compact and efficient.
- You may 'Google' the questions (e.g., error message) you encountered when using NumPy/PyTorch.
  - However, please **DO NOT** copy the code without reference, as we hope you learn from this process.

PyTorch operations

- In PyTorch, you can simply use arithmetic operations the same as NumPy operations.

```
# For example, If you want to add two tensors:
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])
```

This is useful when you want to feed the output of final convolution layer to fully connected layers.

Autograd

- PyTorch provides automated differentiation for all operations on Tensors. Thus, you don't need to rewrite the backpropagation parts.

Example: Differentiating $\mathbf{Z} = \frac{3}{4}\left\|(\boldsymbol{X}+\mathbf{2}) \odot (\boldsymbol{X}+\mathbf{2})\right\|_{\mathbf{1}}, \boldsymbol{X} = \mathbf{1}$

```
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]])
```

If you see something like <MulBackward0>, your tensor has a backward gradient computed by the Autograd engine.

Setting up training pipeline

- Step 0: Write the DNN model
- Step 1: Setup transformation (Preprocessing)
- Step 2: Setup Data Loader (I/O)
- Step 3: Setup Dataset (I/O)
- Step 4: Setup Loss Function
- Step 5: Setup Optimizer

How to write your own DNN model?

- Custom PyTorch Block: Template

Please follow this template. Otherwise, your PyTorch code cannot run normally.

```
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. You must super the parent class.

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.

The above block can be instantiated by:

```
net = Block()
```

How to write your own DNN model?

- **Example:** Building a LeNet-5 for MNIST

```
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
```

Super your class to initialize from parent class in __init__ function. Layer definitions are defined in the **__init__** method. Weights for convolutional/linear layers are initialized.

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

How to write your own DNN model?

- Modularize your DNN

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

  def forward(self, x):
     …
     return output

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

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

Larger blocks can be constructed by reusing smaller blocks.

Modularization relieves the trouble of debugging and makes the code more readable.

Important: remember to use super to initialize each of your custom module from parent class.

Setting up pipeline I: Transformation

Data transformation is a preprocessing step for your target dataset.

```
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.

Setting up pipeline I: Transformation

- Use **transforms.Compose** to compose different transforms to formulate a pipeline.

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

Setting up pipeline II: Data loader

- Data loader sets the I/O pipeline and process your data efficiently.
- Data loaders can be imported from **torch.utils.data.DataLoader**.

```
CLASS torch.utils.data.DataLoader(dataset, batch_size=1, shuffle=False, sampler=None,
    batch_sampler=None, num_workers=0, collate_fn=None, pin_memory=False, drop_last=False,    [SOURCE]
    timeout=0, worker_init_fn=None, multiprocessing_context=None, generator=None)
```

Data loader. Combines a dataset and a sampler, and provides an iterable over the given dataset.

- You should specify some training settings in the data loader, including **batch_size**, **shuffle etc.**
- **num_workers** is usually configured to be the number of available CPU cores on your system. Please use it sparingly.

Setting up pipeline III: Dataset

- **Torchvision.datasets** prepares the dataset for you. Pass it to the data loader finishes the I/O pipeline.
- **Torchvision.datasets** covers a various of public vision datasets, including MNIST, CIFAR-10 etc.

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)
```

Setting up pipeline III: Dataset

- Example: 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=128,
shuffle=True, num_workers=4)

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=4)
```

Note: This is for general use. We will use an alternative dataset loader as a part of requirements in Lab 2.

Setting up pipeline IV: 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 :

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

Setting up pipeline IV: 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.

$\boxed{\text{The } \textit{input} \text{ is expected to contain raw, unnormalized scores for each class.}}$

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 or documentation of PyTorch.

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

Setting up pipeline V: Optimizer

- Optimizer is what we use to optimize the loss function during training. The optimizers are defined in **torch.optim** package.
- Some popular optimizers are:

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 cases.

Setting up pipeline V: Optimizer

Optimizer should be defined after the computational graph (your custom module) is finalized.

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

Example: Define an optimizer using SGD with momentum

```
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 parameters in their default settings. We will talk about hyperparameter tuning in the next few lectures.

Setting up pipeline V: Optimizer

- How to compute gradients and apply it to update the weights?
- **Step 1: Zero the gradients.** This ensures that the gradient computation is correct.

```
optimizer.zero_grad()
```

- Step 2: Compute gradients use back propagation

```
loss.backward()
```

- Step 3: Take the optimization step to apply gradients

```
optimizer.step()
```

Repeat this 3-step loop and you can train your neural network model gradually.

Setting up pipeline V: Optimizer

- Learning rate can be scheduled in the optimizer to achieve maximum performance.

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

- You can also use existing 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

- Putting the 5-stage pipeline all-together, you are able to train a real neural network smoothly!
- Let’s have a more concrete example. 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.
- This toy example mainly focuses on setting up the model and launching training. We will see more complicated examples in our labs.

Case study: Dynamic Net

- Import essentials

```
import torch
import random
```

- For more complicated neural architecture design, it is recommended to import all of 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.

The module can be instantiated by:

```
model = DynamicNet(D_in, H, D_out)
```

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)
```

- As we are using toy data here, we may skip step 2 and step 3 in this example. You may expect to see these steps on a real dataset in Lab 2 and beyond.
- **Exercise:** Try to change some of these parameters. What will you observe?

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 0.01.

Case study: Dynamic Net

- Up to now, the training pipeline is all set!
- Let's try to start it by running the forward/backward pass

```
for t in range(500):
    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()

    # 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())

    # Backward pass: Compute predicted y by passing x to the model
    loss.backward()
    optimizer.step()
```

Note: remember to copy inputs and labels to GPU device if you are using the GPU version of PyTorch.

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
```

- **Examples:** Dropout, BatchNorm, …

Training on GPU

- GPU gives a considerable acceleration on training speed compared to CPUs. As computation graph are placed on CPU by default, you have to manually deploy it on GPU.

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)
```

Note: this must be run on an instance with GPU configuration and CUDA compatibility.

Training on GPU

- Don't forget to copy **all** 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

The model is serialized in a pickle object.

```
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 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

Make sure the structure of your code is not broken before loading.

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

Note: 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 **(Recommended)**

Only the weight parameters are saved as a state dictionary.

```
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 (Recommended)

Construct the model, then load the state dictionary.

```
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