



PyTorch Tutorial

I-Hung Hsu
Sep 14th, 2021

Outline



- Introduction
- Autograd
- Network
 - nn Package
 - Optimizer
- Dataset and DataLoader
- Tips to write codes for NLP research?



Introduction



PYTORCH

- It's a python-based scientific computing package
- A replacement for NumPy to use the power of GPUs
- A deep learning research platform that provides maximum flexibility and speed



Installation

- Follow instructions in <https://pytorch.org/get-started/locally/>
-

PyTorch Build	Stable (1.9.0)		Preview (Nightly)		LTS (1.8.2)	
Your OS	Linux		Mac		Windows	
Package	Conda	Pip		LibTorch		Source
Language	Python			C++ / Java		
Compute Platform	CUDA 10.2	CUDA 11.1		ROCm 4.2 (beta)		CPU
Run this Command:	conda install pytorch torchvision torchaudio cudatoolkit=10.2 -c pytorch					



Three Levels of Abstraction

- Tensor: Imperative ndarray but runs on GPU or other hardware accelerators.
- (Trainable) Tensor: Node in a computational graph; stores data and gradient
- Module: A neural network layer; may store state or learnable weights

Tensors

PyTorch Tensors are just like numpy arrays, but they can run on GPU.

Here we fit a two-layer net using PyTorch Tensors.

```
import torch
```

```
dtype = torch.cuda.FloatTensor
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)
```

```
lr = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred-y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred-y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= lr*grad_w1
    w2 -= lr*grad_w2
```

Tensors

Create random tensor for data and weight

Tensor can also be loaded by:

1. Load from data (list)

```
data = [[1, 2],[3, 4]]  
x_data = torch.tensor(data)
```

2. From Numpy array

```
np_array = np.array(data)  
x_np = torch.from_numpy(np_array)
```

3. From another tensor

```
np_array = np.array(data)  
x_np = torch.from_numpy(np_array)
```

```
import torch
```

```
dtype = torch.cuda.FloatTensor
```

```
N, D_in, H, D_out = 64, 1000, 100, 10  
x = torch.randn(N, D_in).type(dtype)  
y = torch.randn(N, D_out).type(dtype)  
w1 = torch.randn(D_in, H).type(dtype)  
w2 = torch.randn(H, D_out).type(dtype)
```

```
lr = 1e-6
```

```
for t in range(500):
```

```
    h = x.mm(w1)
```

```
    h_relu = h.clamp(min=0)
```

```
    y_pred = h_relu.mm(w2)
```

```
    loss = (y_pred-y).pow(2).sum()
```

```
    grad_y_pred = 2.0 * (y_pred-y)
```

```
    grad_w2 = h_relu.t().mm(grad_y_pred)
```

```
    grad_h_relu = grad_y_pred.mm(w2.t())
```

```
    grad_h = grad_h_relu.clone()
```

```
    grad_h[h < 0] = 0
```

```
    grad_w1 = x.t().mm(grad_h)
```

```
    w1 -= lr*grad_w1
```

```
    w2 -= lr*grad_w2
```


Tensors

Forward pass: compute predictions and loss

```
import torch

dtype = torch.cuda.FloatTensor
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)
```

```
lr = 1e-6
```

```
for t in range(500):
```

```
    h = x.mm(w1)
```

```
    h_relu = h.clamp(min=0)
```

```
    y_pred = h_relu.mm(w2)
```

```
    loss = (y_pred-y).pow(2).sum()
```

```
    grad_y_pred = 2.0 * (y_pred-y)
```

```
    grad_w2 = h_relu.t().mm(grad_y_pred)
```

```
    grad_h_relu = grad_y_pred.mm(w2.t())
```

```
    grad_h = grad_h_relu.clone()
```

```
    grad_h[h < 0] = 0
```


```
    grad_w1 = x.t().mm(grad_h)
```

```
    w1 -= lr*grad_w1
```

```
    w2 -= lr*grad_w2
```

Tensors

Backward pass: manually compute
gradients *if you don't have autograd.*



```
import torch
```

```
dtype = torch.cuda.FloatTensor
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)
```

```
lr = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred-y).pow(2).sum()
```

```
grad_y_pred = 2.0 * (y_pred-y)
grad_w2 = h_relu.t().mm(grad_y_pred)
grad_h_relu = grad_y_pred.mm(w2.t())
grad_h = grad_h_relu.clone()
grad_h[h < 0] = 0
grad_w1 = x.t().mm(grad_h)
```

```
w1 -= lr*grad_w1
w2 -= lr*grad_w2
```

Tensors

Optimization: Gradient descent step on weights

```
import torch

dtype = torch.cuda.FloatTensor
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

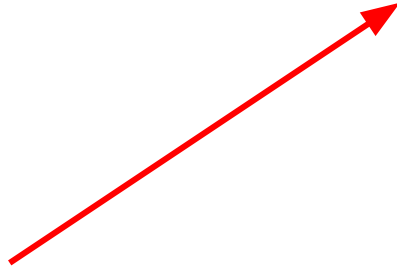
lr = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred-y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred-y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= lr*grad_w1
    w2 -= lr*grad_w2
```

Tensors

To run on GPU, just cast tensors to a cuda datatype.



```
import torch

dtype = torch.cuda.FloatTensor
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

lr = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred-y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred-y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= lr*grad_w1
    w2 -= lr*grad_w2
```



Autograd

Autograd



The previous process:

- Slow
- Gradient is hard to compute when model becomes more complex

=> That's why we need PyTorch

Key:

Set “required_grad” to True to enable torch.autograd

Autograd



We set the weights' "requires_grad" to be True

```
dtype = torch.cuda.FloatTensor
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

w3 = w1.detach().clone()
w3.requires_grad=True
# To initialize, you can use:
# w3 = torch.randn(D_in, H, requires_grad=True).type(dtype)

w4 = w2.detach().clone()
w4.requires_grad=True
```



Autograd

Forward pass looks exactly the same as the Tensor/Numpy version.

```
lr = 1e-6
for t in range(500):
    y_pred2 = x.mm(w3).clamp(min=0).mm(w4)
    loss = (y_pred2-y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w3 -= lr * w3.grad
        w4 -= lr * w4.grad

    w3.grad.zero_()
    w4.grad.zero_()
```




Autograd

But the gradient of loss with respect to w_3 and w_4 can be done by a simple one-line code.

```
lr = 1e-6
for t in range(500):
    y_pred2 = x.mm(w3).clamp(min=0).mm(w4)
    loss = (y_pred2-y).pow(2).sum()
    loss.backward()

    with torch.no_grad():
        w3 -= lr * w3.grad
        w4 -= lr * w4.grad

    w3.grad.zero_()
    w4.grad.zero_()
```



Autograd

Make gradient step on weights.

What's `torch.no_grad()`?

- We need to use `NO_GRAD` to keep the update out of the gradient computation
- Why is that? It boils down to the DYNAMIC GRAPH that PyTorch uses.

```
lr = 1e-6
for t in range(500):
    y_pred2 = x.mm(w3).clamp(min=0).mm(w4)
    loss = (y_pred2-y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w3 -= lr * w3.grad
        w4 -= lr * w4.grad

    w3.grad.zero_()
    w4.grad.zero_()
```



New Autograd Functions

We can define our own autograd functions by writing forward and backward for Tensors

```
class ReLU(torch.autograd.Function):  
  
    @staticmethod  
    def forward(ctx, x):  
        ctx.save_for_backward(x)  
        return x.clamp(min=0)  
  
    @staticmethod  
    def backward(ctx, grad_y):  
        x, = ctx.saved_tensors  
        grad_input = grad_y.clone()  
        grad_input[x < 0] = 0  
        return grad_input
```



New Autograd Functions

```
class ReLU(torch.autograd.Function):
```

```
    @staticmethod
    def forward(ctx, x):
        ctx.save_for_backward(x)
        return x.clamp(min=0)
```

```
    @staticmethod
    def backward(ctx, grad_y):
        x, = ctx.saved_tensors
        grad_input = grad_y.clone()
        grad_input[x < 0] = 0
        return grad_input
```

```
lr = 1e-6
for t in range(500):
    y_pred3 = ReLU.apply(x.mm(w5)).mm(w6)
    loss = (y_pred3 - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w5 -= lr * w5.grad
        w6 -= lr * w6.grad

    w5.grad.zero_()
    w6.grad.zero_()
```

We then apply our new function in the forward pass.



Network

torch.nn

- Higher-level wrapper for working with neural nets
- The most commonly used in my own research

```
import torch

dtype = torch.cuda.FloatTensor
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out)
)
model.cuda()

loss_fn = torch.nn.MSELoss(reduction='sum')

def weights_init(m):
    if isinstance(m, torch.nn.Linear):
        torch.nn.init.zeros_(m.weight)
        torch.nn.init.ones_(m.bias)

model.apply(weights_init)
print(model)

lr = 1e-6

for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()
    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param.data -= lr * param.grad.data
```





torch.nn

Define our model as a
sequence of layers

Nn also defines common
loss functions

```
import torch

dtype = torch.cuda.FloatTensor
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out)
)
model.cuda()

loss_fn = torch.nn.MSELoss(reduction='sum')

def weights_init(m):
    if isinstance(m, torch.nn.Linear):
        torch.nn.init.zeros_(m.weight)
        torch.nn.init.ones_(m.bias)

model.apply(weights_init)
print(model)

lr = 1e-6

for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

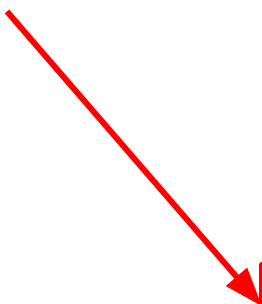
    model.zero_grad()
    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param.data -= lr * param.grad.data
```

torch.nn

Forward pass:

- Feed data to model
- Use prediction to and ground truth to get loss function



```
import torch

dtype = torch.cuda.FloatTensor
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out)
)
model.cuda()

loss_fn = torch.nn.MSELoss(reduction='sum')

def weights_init(m):
    if isinstance(m, torch.nn.Linear):
        torch.nn.init.zeros_(m.weight)
        torch.nn.init.ones_(m.bias)

model.apply(weights_init)
print(model)

lr = 1e-6

for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()
    loss.backward()

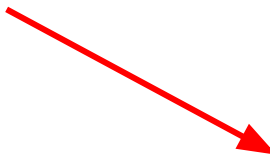
    with torch.no_grad():
        for param in model.parameters():
            param.data -= lr * param.grad.data
```



torch.nn

Pytorch handles autograd for us!

And now, we can simply use
`model.zero_grad()` to clear all
gradients for the parameters in the
model



```
import torch

dtype = torch.cuda.FloatTensor
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out)
)
model.cuda()

loss_fn = torch.nn.MSELoss(reduction='sum')

def weights_init(m):
    if isinstance(m, torch.nn.Linear):
        torch.nn.init.zeros_(m.weight)
        torch.nn.init.ones_(m.bias)

model.apply(weights_init)
print(model)

lr = 1e-6

for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()
    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param.data -= lr * param.grad.data
```



Optimizer

Make gradient step on each model parameter.

Question:

- How can we apply more advanced rules for updating
- Easier way?

```
import torch

dtype = torch.cuda.FloatTensor
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out)
)
model.cuda()

loss_fn = torch.nn.MSELoss(reduction='sum')

def weights_init(m):
    if isinstance(m, torch.nn.Linear):
        torch.nn.init.zeros_(m.weight)
        torch.nn.init.ones_(m.bias)

model.apply(weights_init)
print(model)

lr = 1e-6

for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()
    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param.data -= lr * param.grad.data
```





Optimizer

Call nn.optim package, which contains various advanced optimizer other than SGD.

Now, all the parameters can be updated via one-line code.

```
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    #model.zero_grad()
    optimizer.zero_grad()
    loss.backward()

    optimizer.step()

    # with torch.no_grad():
    #     for param in model.parameters():
    #         param.data -= lr * param.grad.data
```



Define new modules

Pytorch **Module** is a neural network layer, it can contain weights or other modules.

It provides a more systematic way to structure our code.

```
class TwoLayerMLP(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super().__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)
        self.weight_init()

    def weight_init(self):
        torch.nn.init.zeros_(self.linear1.weight)
        torch.nn.init.zeros_(self.linear2.weight)
        torch.nn.init.ones_(self.linear1.bias)
        torch.nn.init.ones_(self.linear2.bias)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

model = TwoLayerMLP(D_in, H, D_out)
model.cuda()
print(model)
loss_fn = torch.nn.MSELoss(reduction='sum')
lr = 1e-6

optimizer = torch.optim.Adam(model.parameters(), lr=lr)
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```



Dataset & DataLoaders

DataLoaders

A **DataLoader** wraps a **Dataset** and provides minibatching, shuffling, multithreading, for you

When you need to load custom data, just **write your own Dataset class**

```
import torch
from torch.utils import data

class Dataset(data.Dataset):
    'Characterizes a dataset for PyTorch'
    def __init__(self, list_IDs, labels):
        'Initialization'
        self.labels = labels
        self.list_IDs = list_IDs

    def __len__(self):
        'Denotes the total number of samples'
        return len(self.list_IDs)

    def __getitem__(self, index):
        'Generates one sample of data'
        # Select sample
        ID = self.list_IDs[index]

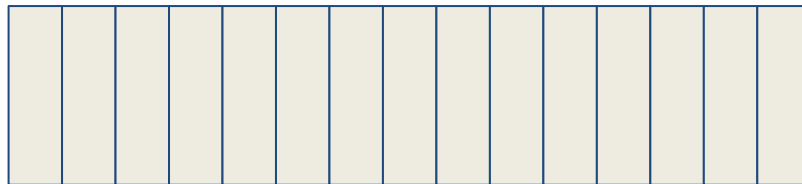
        # Load data and get label
        X = torch.load('data/' + ID + '.pt')
        y = self.labels[ID]

        return X, y
```

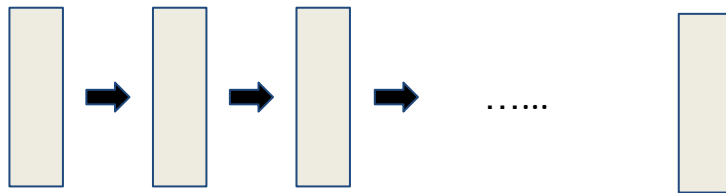


What is (mini)batching?

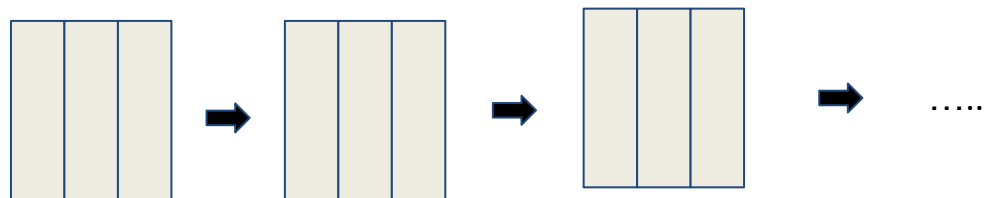
All your training data:



Standard for loop:



(Mini) batching:





Adapt Dataset to DataLoaders

```
# Parameters
params = {'batch_size': 64,
          'shuffle': True,
          'num_workers': 6}
max_epochs = 100

# Datasets
partition = # IDs
labels = # Labels

# Generators
training_set = Dataset(partition['train'], labels)
training_generator = data.DataLoader(training_set, **params)

validation_set = Dataset(partition['validation'], labels)
validation_generator = data.DataLoader(validation_set, **params)

# Loop over epochs
for epoch in range(max_epochs):
    # Training
    for local_batch, local_labels in training_generator:
        # Transfer to GPU
        local_batch, local_labels = local_batch.to(device), local_labels.to(device)

    # Model computations
    [...]
```




Adapt Dataset to DataLoaders

```
# Parameters
params = {'batch_size': 64,
          'shuffle': True,
          'num_workers': 6}
max_epochs = 100
```

```
# Datasets
partition = # IDs
labels = # Labels
```

```
# Generators
training_set = Dataset(partition['train'], labels)
training_generator = data.DataLoader(training_set, **params)

validation_set = Dataset(partition['validation'], labels)
validation_generator = data.DataLoader(validation_set, **params)
```

```
# Loop over epochs
for epoch in range(max_epochs):
    # Training
    for local_batch, local_labels in training_generator:
        # Transfer to GPU
        local_batch, local_labels = local_batch.to(device), local_labels.to(device)

    # Model computations
    [...]
```

**DataLoader perform
batching automatically**

```
import torch
from torch.utils import data

class Dataset(data.Dataset):
    'Characterizes a dataset for PyTorch'
    def __init__(self, list_IDS, labels):
        'Initialization'
        self.labels = labels
        self.list_IDS = list_IDS

    def __len__(self):
        'Denotes the total number of samples'
        return len(self.list_IDS)

    def __getitem__(self, index):
        'Generates one sample of data'
        # Select sample
        ID = self.list_IDS[index]

        # Load data and get label
        X = torch.load('data/' + ID + '.pt')
        y = self.labels[ID]

        return X, y
```

However, what if your data structure is more complex?



```
data = {  
    'tokens': ["This", "is", "a", "scientific", "book", "."],  
    'pieces': ["This", "is", "a", "sci", "enti", "fic", "book", "."],  
    'triggers': ['is'],  
    ...  
}
```

- The automatic batching for DataLoader will only concatenate all “data” in the batch into a list.
- But, a list of data structure is not a tensor model can directly use.

=> collate_fn



DataLoader

Several parameters that can be adjusted, for data that are structured in extremely complex case, `collate_fn` is suggested to be use.

- You can organize your data in a map/dict style
- Given a batch, we reorganize and repack them.

Parameters

- **dataset** (*Dataset*) – dataset from which to load the data.
- **batch_size** (*int, optional*) – how many samples per batch to load (default: 1).
- **shuffle** (*bool, optional*) – set to `True` to have the data reshuffled at every epoch (default: `False`).
- **sampler** (*Sampler or Iterable, optional*) – defines the strategy to draw samples from the dataset. Can be any `Iterable` with `__len__` implemented. If specified, `shuffle` must not be specified.
- **batch_sampler** (*Sampler or Iterable, optional*) – like `sampler`, but returns a batch of indices at a time. Mutually exclusive with `batch_size`, `shuffle`, `sampler`, and `drop_last`.
- **num_workers** (*int, optional*) – how many subprocesses to use for data loading. 0 means that the data will be loaded in the main process. (default: 0)
- **collate_fn** (*callable, optional*) – merges a list of samples to form a mini-batch of Tensor(s). Used when using batched loading from a map-style dataset.

```
def collate_fn(self, batch):
    tokens = [inst.tokens for inst in batch]
    pieces = [inst.pieces for inst in batch]
    piece_idx = [inst.piece_idx for inst in batch]
    token_lens = [inst.token_lens for inst in batch]
    token_start_idx = [inst.token_start_idx for inst in batch]
    triggers = [inst.triggers for inst in batch]
    roles = [inst.roles for inst in batch]
    wnd_ids = [inst.wnd_id for inst in batch]

    return EEBatch(
        tokens=tokens,
        pieces=pieces,
        piece_idx=piece_idx,
        token_lens=token_lens,
        token_start_idx=token_start_idx,
        triggers=triggers,
        roles=roles,
        wnd_ids=wnd_ids,
    )
```

Summary



1. Prepare you data
 - a. Write your own Dataset (inherit `torch.nn.util.dataset`)
 - b. (Write `collate_fn`)
2. Create your model
 - a. A sequential module if your model is super easy and will not be reused.
 - b. A `nn.Module` module
3. Write the loop (how many epoch/steps) to train your model:
 - a. Create a `DataLoader` that wraps the Dataset you provide
 - b. Set an optimizer
 - c. Set a loss for optimization
 - d. For loop....
 - i. Forward pass
 - ii. Zero-grad
 - iii. Backward pass => Get gradient
 - iv. Optimizer step to update your models' weight.



Other Tips



Prepare Different Version of Data

- Tiny-size: for debugging syntactic bug (especially for dynamic language, such as python)
- Small-size: check the behavior of the model
- Mid-size: for understanding model behavior and fast development
- Full-size: conduct final experiments



Tensorboard

- Installation:

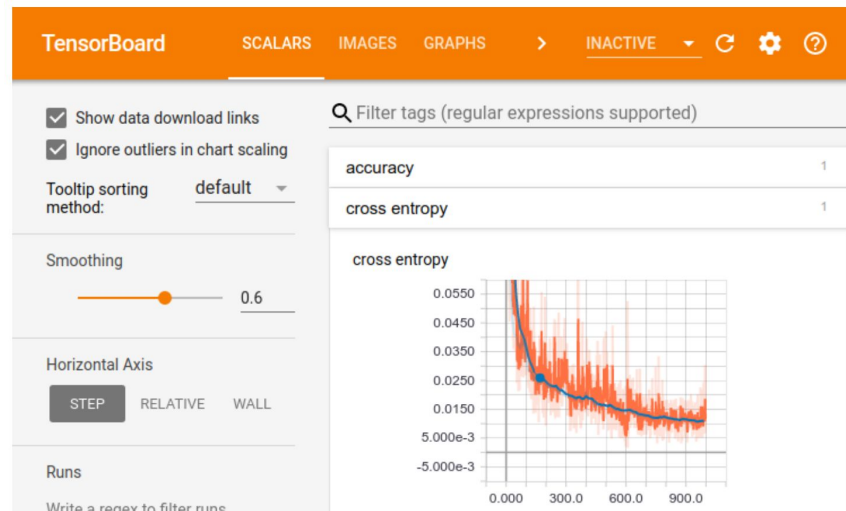
https://pytorch.org/tutorials/recipes/recipes/tensorboard_with_pytorch.html

- Create a Summary Writer

```
from torch.utils.tensorboard import SummaryWriter  
writer = SummaryWriter()
```

- During Training/Dev/Test

```
writer.add_scalar("Loss/train", loss, epoch)
```





Padding

- In NLP, we usually need to face cases that requires “padding” so as to batch your data
- Useful tool: `torch.nn.utils.rnn.pad_sequence`

```
>>> from torch.nn.utils.rnn import pad_sequence
>>> a = torch.ones(25, 300)
>>> b = torch.ones(22, 300)
>>> c = torch.ones(15, 300)
>>> pad_sequence([a, b, c]).size()
torch.Size([25, 3, 300])
```

- For unpack, use: `torch.nn.utils.rnn.pack_padded_sequence`

Parameters

- **input** (*Tensor*) – padded batch of variable length sequences.
- **lengths** (*Tensor or list(int)*) – list of sequence lengths of each batch element (must be on the CPU if provided as a tensor).
- **batch_first** (*bool, optional*) – if `True`, the input is expected in `B x T x *` format.
- **enforce_sorted** (*bool, optional*) – if `True`, the input is expected to contain sequences sorted by length in a decreasing order. If `False`, the input will get sorted unconditionally. Default: `True`.

Extension -- End to end Examples



- <https://www.analyticsvidhya.com/blog/2020/01/first-text-classification-in-pytorch/>
- <https://towardsdatascience.com/lstm-text-classification-using-pytorch-2c6c657f8fc0>
-



Acknowledgement

- Several materials from Pytorch Official Materials:
<https://pytorch.org/tutorials/>
- An easy to read blog:
<https://towardsdatascience.com/understanding-pytorch-with-an-example-a-step-by-step-tutorial-81fc5f8c4e8e#ea0d>
- Special thanks to the materials shared from the friends in UCLA NLP group.