Pytorch

Natural Language Processing
Lecture 7

THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

Pytorch

- Open source machine learning library.
- Developed by Facebook's AI Research lab.
- It leverages the power of GPUs.
- Automatic computation of gradients.
- Makes it easier to test and develop new ideas.

Why PyTorch?

- It is pythonic - concise, close to Python conventions
- Strong GPU support
- Autograd - automatic differentiation
- Many algorithms and components are already implemented
- Similar to NumPy

Pytorch Advantages

- Easy to understand the code
- Has as many type of layers as Torch (Pool, CONV 1,2,3D, LSTM, MLP)
- Lot's of loss functions
- Very similar to numpy library.
- Faster compare to other frameworks.
- Allow to build networks which structure is dependent on the computation itself.

Pytorch Levels of Abstraction

- Tensor: Like numpy array, but runs on GPU
- Variable: Node in a computational graph; stores data and gradient
- Module: A neural network layer; may store state or learnable weights

- Create a tensor
- Math Operation
- https://pytorch.org/docs/stable/index.html

- NumPy's main object is the homogeneous multidimensional array.
- It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers.
- In NumPy dimensions are called axes. The number of axes is rank.
- For example, the coordinates of a point in 3D space [1, 2, 1] is an array of rank 1, because it has one axis. That axis has a length of 3.

Initializing a Tensor

- Directly from data: Tensors can be created from data. The data type is automatically inferred.

```
import torch
data = [[1,2,3,4],[4,5,6,7]]
data_tensor = torch.Tensor(data)
print(data_tensor)
```

- From a Numpy array: Tensors can be created from a numpy array

```
import numpy as np
data_numpy = np.array(data)
data_tensor = torch.Tensor(data_numpy)
print(data_tensor)
```

- From another tensors: The new tensor retains the properties (shape and datatype) of the argument tensor, unless explicitly overridden.

```
x = torch.ones_like(data_tensor)
print(x)
y = torch.randn_like(data_tensor)
print(y)
```

- Tensor attributes describe the shape, datatype, and the device on which they are sorted.

```
import torch
T = torch.randn(3,4)
print(T.dtype)
print(T.shape)
print(T.device)
```

- By default, tensors are created on the CPU. We need to explicitly move tensors to GPU. We can use a method **.to**. Note that copying data from cpu to gpu and viceversa can be expansive in terms of time and memory.

```
if torch.cuda.is_available():
        tensor_gpu = T.to('cuda')
else:
    print('Tensors are not in the GPU.')
```

- Standard numpy-like indexing and slicing.

```
T1 = torch.ones(4,4)
print(T1[:,0])
print(T1[:,-1])
print(T1[...,-1])
```

- Matrix multiplication (element wise and matrix multiplication)

```
T2 = torch.rand(4,1)
T3 = T1 @ T2 ; print(T3)
T4 = T1.matmul(T2); print(T4)

T5 = torch.randn(4,4)
T6 = T1 * T5; print(T6)
T7 = T1.mul(T5); print(T7)
```

- Tensor Operations

```
import torch

x = torch.Tensor(2, 3) ; print(x)
y = torch.rand(2, 3)
z2 = torch.add(x, y); print(z2)
print(torch.is_tensor(z2))
z1 = torch.Tensor(2, 3)
torch.add(x, y, out=z1)

print(x.size())
print(torch.numel(x))

k = x.view(6); print(k)
l = x.view(-1, 2); print(l)

x1 = torch.randn(5, 3).type(torch.FloatTensor); print(x1)
x2 = torch.randn(5, 3).type(torch.LongTensor); print(x2)

v = torch.arange(9); print(v)
r1 = torch.cat((x, x, x), 0); print(r1)
r2 = torch.stack((v, v)); print(r2)
r3 = torch.chunk(v, 3); print(r3)
```

- Tensor Operations

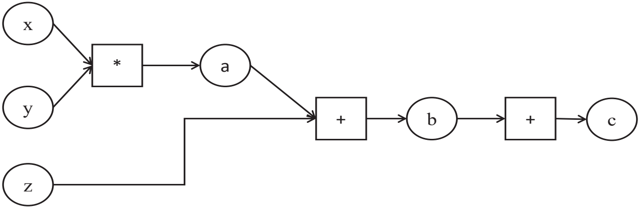
```python
import torch

t = torch.rand(2, 1, 2, 1); print(t)
r = torch.squeeze(t); print(r)
r = torch.squeeze(t, 1); print(r)


x = torch.rand([1, 2, 3]); print(x)
r = torch.unsqueeze(x, 0); print(r)
r = torch.unsqueeze(x, 1); print(r)

v = torch.arange(9).reshape(3,3)
# flatten a Tensor and return elements with given indexes
r = torch.take(v, torch.LongTensor([0, 4, 2]))
r = torch.transpose(v, 0, 1); print(r)

mat1 = torch.randn(2, 3)
mat2 = torch.randn(3, 4)
r = torch.mm(mat1, mat2)

v1 = torch.ones(3)
r = torch.diag(v1)
```

- A Pytorch Variable is a node in a computational graph.
- The autograd package provides automatic differentiation for all operations on Tensors.
- " autograd.Variable is the central class of the package. It wraps a Tensor, and supports nearly all of operations defined on it.
- Once you finish your computation you can call .backward() and have all the gradients computed automatically. "

Computational Graph - Numpy

- Lets use numpy and code the graph.

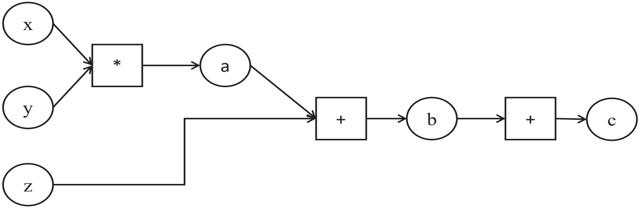
```python
import numpy as np
x = np.random.randn(3,4)
y = np.random.randn(3,4)
z = np.random.randn(3,4)
a = x * y
b = a + z
c = np.sum(b)
grad_c = 1.0
grad_b = grad_c * np.ones((3,4))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a *x
```



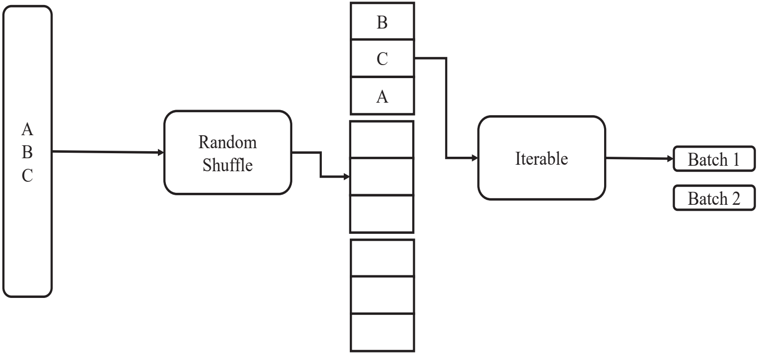
Computational Graph - PyTorch

- Lets use torch Variable and code the graph.

```python
import torch
from torch.autograd import Variable
x = Variable(torch.randn(3,4), requires_grad = True)
y = Variable(torch.randn(3,4), requires_grad = True)
z = Variable(torch.randn(3,4), requires_grad = True)
a = x * y
b = a + z
c = torch.sum(b)

c.backward()
print(x.grad.data)
print(y.grad.data)
print(z.grad.data)
```



- Shuffle and iterate the data.



Define the Class

- We define neural network by subclassing nn.Module and initialize the neural network layers in __init__. Every nn.Module subclass implements the operations on input data in forward methos.

```python
from torch import nn
class model(nn.Module):
    def __init__(self, hidden_dim):
        super(model, self).__init__()
        self.linear1 = nn.Linear(1, hidden_dim)
        self.act1 = torch.sigmoid
        self.linear2 = nn.Linear(hidden_dim, 1)
    def forward(self, x):
        return self.linear2(self.act1(self.linear1(x)))
```

Loass Function, Optimizers, Training Loop

- We pass our models output logits to loss function

```
import torch
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr = 0.001)
```

- Training Loop

```
def train_loop (dataloader, model, loss, optimizer):
    size = len(dataloader.dataset)
    for batch, (X,y) in enumerate(dataloader):
        pred = model(X)
        loss = criterion(pred,y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Sample Code

- To begin, load the required libraries. The first package you'll import is the torch library.

```
import torch
from torch import nn
from torchtext.data.utils import get_tokenizer
from torchtext.vocab import build_vocab_from_iterator
from torch.utils.data import DataLoader
from torchtext.datasets import AG_NEWS
import time
```

- For example, the AG_NEWS dataset iterators yield the raw data as a tuple of label and text.

```
train_iter = list(AG_NEWS(split='train'))
test_iter = list(AG_NEWS(split='test'))
print(train_iter[0])
```

- The first step is to build a vocabulary with the raw training dataset.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
tokenizer = get_tokenizer('basic_english')
def yield_tokens(data_iter):
    for _, text in data_iter:
        yield tokenizer(text)
```

- Here we use built in factory function build_vocab_from_iterator which accepts iterator that yield list or iterator of tokens.

```
vocab = build_vocab_from_iterator(yield_tokens(train_iter), specials=["<unk>"])
vocab.set_default_index(vocab["<unk>"])
print(vocab(['here', 'is', 'an', 'example']))
```

- Users can also pass any special symbols to be added to the vocabulary.

Prepare data processing pipelines

- Prepare the text processing pipeline with the tokenizer and vocabulary.

```
text_pipeline = lambda x: vocab(tokenizer(x))
label_pipeline = lambda x: int(x) - 1
```

- The text pipeline converts a text string into a list of integers based on the lookup table defined in the vocabulary. The label pipeline converts the label into integers.

```
print(text_pipeline('here is the an example'))
print(label_pipeline('10'))
```

Generate data batch and iterator

- **torch.utils.data.DataLoader** is recommended for PyTorch user.
- It works with a map-style dataset that implements the getitem() and len() protocols, and represents a map from indices/keys to data samples.
- Before sending to the model, collate_fn function works on a batch of samples generated from DataLoader.
- The input to collate_fn is a batch of data with the batch size in DataLoader.
- Pay attention here and make sure that collate_fn is declared as a top level def.
- This ensures that the function is available in each worker.

Custom Dataloader

- This is a general architecture of any dataloader.

```python
from torch.utils.data import DataLoader
import numpy as np
from torch.utils.data import Dataset
import torch
class Custom_Data_loader(Dataset):
    def __init__(self):
        y = np.loadtxt('data-diabetes.csv', delimiter=',', dtype=np.float32)
        self.len = y.shape[0]
        self.x_data = torch.from_numpy(y[:, 0:-1])
        self.y_data = torch.from_numpy(y[:, [-1]])

def __len__(self):
    return self.len

def __getitem__(self, index):
    return self.x_data[index], self.y_data[index]


dataset = Custom_Data_loader()
train_loader = DataLoader(dataset=dataset,batch_size=32,shuffle=True,num_workers=2)
```

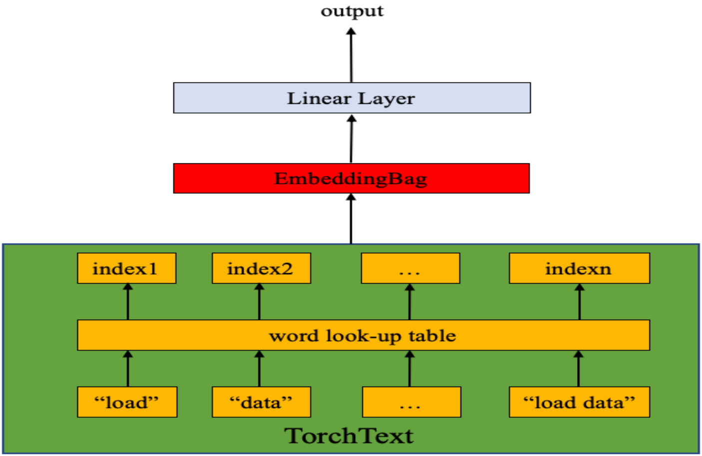
Generate data batch and iterator

- In this example, the text entries in the original data batch input are packed into a list and concatenated as a single tensor for the input of nn.EmbeddingBag.

```python
def collate_batch(batch):
    label_list, text_list, offsets = [], [], [0]
    for (_label, _text) in batch:
         label_list.append(label_pipeline(_label))
         processed_text = torch.tensor(text_pipeline(_text), dtype=torch.int64)
         text_list.append(processed_text)
         offsets.append(processed_text.size(0))
    label_list = torch.tensor(label_list, dtype=torch.int64)
    offsets = torch.tensor(offsets[:-1]).cumsum(dim=0)
    text_list = torch.cat(text_list)
    return label_list.to(device), text_list.to(device), offsets.to(device)
```

- The offset is a tensor of delimiters to represent the beginning index of the individual sequence in the text tensor. Label is a tensor saving the labels of individual text entries.

- Model Architecture



- The model is composed of the nn.EmbeddingBag layer plus a linear layer for the classification purpose.

```python
class TextClassificationModel(nn.Module):
    def __init__(self, vocab_size, embed_dim, num_class):
        super(TextClassificationModel, self).__init__()
        self.embedding = nn.EmbeddingBag(vocab_size, embed_dim)
        self.fc = nn.Linear(embed_dim, num_class)
        self.init_weights()
    def init_weights(self):
        initrange = 0.5
        self.embedding.weight.data.uniform_(-initrange, initrange)
        self.fc.weight.data.uniform_(-initrange, initrange)
        self.fc.bias.data.zero_()
    def forward(self, text, offsets):
        embedded = self.embedding(text, offsets)
        return self.fc(embedded)
```

- nn.EmbeddingBag with the default mode of “mean” computes the mean value of a “bag” of embeddings. Although the text entries here have different lengths, nn.EmbeddingBag module requires no padding here since the text lengths are saved in offsets.

Initiate an instance

- We build a model with the embedding dimension of 64. The vocab size is equal to the length of the vocabulary instance. The number of classes is equal to the number of labels,

```
num_class = len(set([label for (label, text) in train_iter]))
vocab_size = len(vocab)
emsize = 64
model = TextClassificationModel(vocab_size, emsize, num_class).to(device)
```

- Model training

```python
def train(dataloader):
    model.train()
    total_acc, total_count = 0, 0
    log_interval = 500
    start_time = time.time()

    for idx, (label, text, offsets) in enumerate(dataloader):
        optimizer.zero_grad()
        predicted_label = model(text, offsets)
        loss = criterion(predicted_label, label)
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), 0.1)
        optimizer.step()
        total_acc += (predicted_label.argmax(1) == label).sum().item()
        total_count += label.size(0)
        if idx % log_interval == 0 and idx > 0:
            elapsed = time.time() - start_time
            print('| epoch {:3d} | {:5d}/{:5d} batches '
                  '| accuracy {:8.3f}'.format(epoch, idx, len(dataloader),
                                              total_acc/total_count))
            total_acc, total_count = 0, 0
            start_time = time.time()
```

- Model training

```python
def train(dataloader):
    model.train()
    total_acc, total_count = 0, 0
    log_interval = 500
    start_time = time.time()

    for idx, (label, text, offsets) in enumerate(dataloader):
        optimizer.zero_grad()
        predicted_label = model(text, offsets)
        loss = criterion(predicted_label, label)
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), 0.1)
        optimizer.step()
        total_acc += (predicted_label.argmax(1) == label).sum().item()
        total_count += label.size(0)
        if idx % log_interval == 0 and idx > 0:
            elapsed = time.time() - start_time
            print('| epoch {:3d} | {:5d}/{:5d} batches '
                  '| accuracy {:8.3f}'.format(epoch, idx, len(dataloader),
                                              total_acc/total_count))
            total_acc, total_count = 0, 0
            start_time = time.time()
```

Define Function to Evaluate the Model

- Model evaluation

```
def evaluate(dataloader):
    model.eval()
    total_acc, total_count = 0, 0

    with torch.no_grad():
        for idx, (label, text, offsets) in enumerate(dataloader):
            predicted_label = model(text, offsets)
            loss = criterion(predicted_label, label)
            total_acc += (predicted_label.argmax(1) == label).sum().item()
            total_count += label.size(0)
    return total_acc/total_count
```

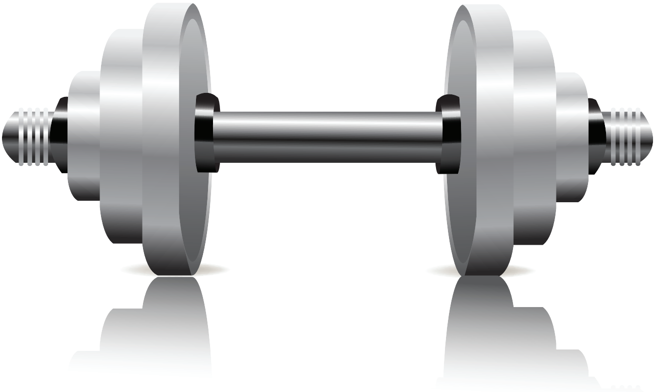
- Hyperparameters

```
EPOCHS = 10
LR = 0.001
BATCH_SIZE = 64
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=LR)
```

- Train the model with dataloader

```
train_dataloader = DataLoader(train_iter, batch_size=BATCH_SIZE,
                              shuffle=True, collate_fn=collate_batch)
test_dataloader = DataLoader(test_iter, batch_size=BATCH_SIZE,
                             shuffle=True, collate_fn=collate_batch)


for epoch in range(1, EPOCHS + 1):
    epoch_start_time = time.time()
    train(train_dataloader)
    accu_val = evaluate(test_dataloader)
    print('-' * 59)
    print('| end of epoch {:3d} | time: {:5.2f}s | '
          'valid accuracy {:8.3f} '.format(epoch,
                                           time.time() - epoch_start_time, accu_val))
    print('-' * 59)
```

- Exercise 1 to 1
- Class-Ex-Lecture7.py



Word2vec

- Instead of capturing co occurrence counts directly.
- Predict surrounding words of every word.
- Predict surrounding words in a window of length m of every word.
- Objec4ve func4on: Maximize the log probability of any context word given the current center word:

$$J(\theta) = \frac{1}{T} \sum_{T}^{t=1} \sum_{-m \geq j \leq m} \log p(w_{t+j} | w_t)$$

- Where θ represents all variables we optimize.

- There are two architectures used by Word2vec.

1- Continuous Bag of words (CBOW)

2- Skip gram

- In CBOW, the current word is predicted using the window of surrounding context windows.
- Skip-Gram performs opposite of CBOW which implies that it predicts the given sequence or context from the word.
- Word2vec provides an option to choose between CBOW (continuous Bag of words) and skim-gram. Such parameters are provided during training of the model.

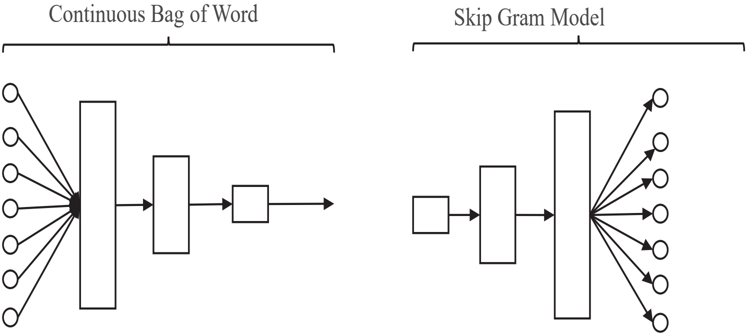
Windowing and Target Word

- Consider a local window of a target word

  Window 2 :Where there is a **will**there is a way

  Window 3 :Where there is a **will**there is a way

  Window 4 : Where there is a **will**there is a way


Continuous Bag of Word
Skip Gram Model

Exercise - Lecture 7

- Exercise 2 to 2
- Class-Ex-Lecture7.py