pytorch-nlp-tutorial Documentation

Brian McMahan and Delip Rao

Sep 10, 2019

Extra Resources

CHAPTER 1

Getting the Data

In this training, there are two options of participating.

1.1 Option 1: Download and Setup things on your laptop

The first option is to download the data below, setup the environment, and download the notebooks when we make them available. If you choose this options but do not download the data before the first day, we will have several flash drives with the data on it.

Please visit this link to download the data.

1.2 Option 2: Use O'Reilly's online resource through your browser

The second option is to use an online resource provided by O'Reilly. On the first day of this training, you will be provided with a link to a JupyterHub instance where the environment will be pre-made and ready to go! If you choose this option, you do not have to do anything until you arrive on Sunday. You are still required to bring your laptop.

CHAPTER 2

Environment Setup

On this page, you will find not only the list of dependencies to install for the tutorial, but a description of how to install them. This tutorial assumes you have a laptop with OSX or Linux. If you use Windows, you might have to install a virtual machine to get a UNIX-like environment to continue with the rest of this instruction. A lot of this instruction is more verbose than needed to accomodate participants of different skill levels.

Please note that these are only optional. On the first day of this training, you will be provided with a link to a JupyterHub instance where the environment will be pre-made and ready to go!

2.1 0. Get Anaconda

Anaconda is a Python (and R) distribution that aims to provide everything needed for common scientific and machine learning situations out-of-the-box. We chose Anaconda for this tutorial as it significantly simplifies Python dependency management.

In practice, Anaconda can be used to manage different environment and packages. This setup document will assume that you have Anaconda installed as your default Python distribution.

You can download Anaconda here: https://www.continuum.io/downloads

After installing Anaconda, you can access its command-line interface with the `conda` command.

2.2 1. Create a new environment

Environments are a tool for sanitary software development. By this, we mean that you can install specific versions of packages without worrying that it breaks a dependency elsewhere.

Here is how you can create an environment with Anaconda

```
conda create -n dl4nlp python=3.6
```

2.3 2. Install Dependencies

2.3.1 2a. Activate the environment

After creating the environment, you need to **activate** the environment:

```
source activate dl4nlp
```

After an environment is activated, it might prepend/append itself to your console prompt to let you know it is active.

With the environment activated, any installation commands (whether it is `pip install X`, `python setup.py install` or using Anaconda's install command `conda install X`) will only install inside the environment.

2.3.2 2b. Install IPython and Jupyter

Two core dependencies are IPython and Jupyter. Let's install them first:

```
conda install ipython
conda install jupyter
```

To allow a jupyter notebooks to use this environment as their kernel, it needs to be linked:

```
python -m ipykernel install --user --name dl4nlp
```

2.3.3 2c. Installing CUDA (optional)

NOTE: CUDA is currently not supported out of the conda package control manager. Please refer to pytorch's github repository for compilation instructions.

If you have a CUDA compatible GPU, it is worthwhile to take advantage of it as it can significantly speedup training and make your PyTorch experimentation more enjoyable.

To install CUDA:

1. Download CUDA appropriate to your OS/Arch from here.
2. Follow installation steps for your architecture/OS. For Ubuntu/x86_64, see here.
3. Download and install CUDNN from here.

Make sure you have the latest CUDA and CUDNN.

2.3.4 2d. Clone (or Download) Repository

At this point, you may have already cloned the tutorial repository. But if you have not, you will need it for the next step.

```
git clone https://github.com/joosthub/pytorch-nlp-tutorial-sj2019
```

If you do not have git or do not want to use it, you can also download the repository as a zip file

2.3.5 2e. Install Dependencies from Repository

Assuming the you have cloned (or downloaded and unzipped) the repository, please navigate to the directory in your terminal. Then, you can do the following. This will also install PyTorch.

```
pip install -r requirements.txt
```

CHAPTER 3

Frequency Asked Questions

On this page, you will find a list of questions that we either anticipate people will ask or that we have been asked previously. They are intended to be the first stop for any confusion or trouble that might occur.

3.1 Do I Need to have a NVIDIA GPU enabled laptop?

Nope! While having a NVIDIA GPU enabled laptop will make the training run faster, we provide instructions for people who do not have one.

If you are plan on working on Natural Language Processing/Deep Learning in the future, a GPU enabled laptop might be a good investment.

CHAPTER 4

Solutions

4.1 Problem 1

For when x is a scalar or a vector of length 1:

```
def f(x):
    if x > 0:
        return torch.sin(x)
    else:
        return torch.cos(x)

x = torch.tensor([1.0, 0.5], requires_grad=True)

y = f(x)
print(y)
y.backward()
print(x.grad)
```

For when x is a vector, the conditional becomes ambiguous. To handle this, we can use the python *all* function. Computing the backward pass requires that the error signal be a scalar. Since there are now multiple outputs of *f*, we can turn *y* into a scalar just by summing the outputs.

```
def f(x):
    if all(x > 0):
        return torch.sin(x)
    else:
        return torch.cos(x)

x = torch.tensor([1.0, 0.5], requires_grad=True)

y = f(x)
print(y)
y.sum().backward()
print(x.grad)
```

There is one last catch to this: we are forcing the fate of the entire vector on a strong "and" condition (all items must be above 0 or they will all be considered below 0). To handle things in a more granular level, there are two different methods.

Method 1: use a for a loop

```
def f2(x):
    output = []
    for x_i in x:
        if x_i > 0:
            output.append(torch.sin(x_i))
        else:
            output.append(torch.cos(x_i))
    return torch.stack(output)

x = torch.tensor([1.0, -1.0], requires_grad=True)
y = f2(x)
print(y)
y.sum().backward()
print(x.grad)
```

Method 2: use a mask

```
def f3(x):
    mask = (x > 0).float()
    # alternatively, mask = torch.gt(x, 0).float()
    return mask * torch.sin(x) + (1 - mask) * torch.cos(x)

x = torch.tensor([1.0, -1.0], requires_grad=True)
y = f3(x)
print(y)
y.sum().backward()
print(x.grad)
```

4.2 Problem 2

```
def cbow(phrase):
    words = phrase.split(" ")
    embeddings = []
    for word in words:
        if word in glove.word_to_index:
            embeddings.append(glove.get_embedding(word))
    embeddings = np.stack(embeddings)
    return np.mean(embeddings, axis=0)

cbow("the dog flew over the moon").shape

# >> (100,)

def cbow_sim(phrase1, phrase2):
    vec1 = cbow(phrase1)
    vec2 = cbow(phrase2)
    return np.dot(vec1, vec2) / (np.linalg.norm(vec1) * np.linalg.norm(vec2))

cbow_sim("green apple", "green apple")
```

(continues on next page)

(continued from previous page)

```
# >> 1.0

cbow_sim("green apple", "apple green")
# >> 1.0

cbow_sim("green apple", "red potato")
# >> 0.749

cbow_sim("green apple", "green alien")
# >> 0.683

cbow_sim("green apple", "blue alien")
# >> 0.5799815958114477

cbow_sim("eat an apple", "ingest an apple")
# >> 0.9304712574359718
```

CHAPTER 5

Warm Up Exercise

To get you back into the PyTorch groove, let's do some easy exercises. You will have 10 minutes. See how far you can get.

1. Use `torch.randn` to create two tensors of size (29, 30, 32) and and (32, 100).
2. Use `torch.matmul` to matrix multiply the two tensors.
3. Use `torch.sum` on the resulting tensor, passing the optional argument of `dim=1` to sum across the 1st dimension. Before you run this, can you predict the size?
4. Create a new long tensor of size (3, 10) from the `np.random.randint` method.
5. Use this new long tensor to index into the tensor from step 3.
6. Use `torch.mean` to average across the last dimension in the tensor from step 5.

CHAPTER 6

Fail Fast Prototype Mode

When building neural networks, you want things to either work or fail fast. Long iteration loops are the truest enemy of the machine learning practitioner.

To that end, the following techniques will help you out.

```
import torch
import torch.nn as nn

# 2dim tensor.. aka a matrix
x = torch.randn(4, 5)

# this is the same as:
batch_size = 4
feature_size = 5
x = torch.randn(batch_size, feature_size)

# now let's try out some NN layer
output_size = 10
fc = nn.Linaer(feature_size, output_size)
print(fc(x).shape)
```

You can construct whatever prototype variables you want doing this.

6.1 Prototyping an embedding

```
import torch
import torch.nn as nn

batch_size = 4
sequence_size = 5
integer_range = 100
embedding_size = 25
```

(continues on next page)

(continued from previous page)

```
# notice rand vs randn.  rand is uniform (0,1), and randn is normal (-1,1)
random_numbers = (torch.rand(batch_size, sequence_size) * integer_range).long()

embedder = nn.Embedding(num_embeddings=integer_range,
                        embedding_dim=embedding_size)

print(embedder(x).shape)
```

CHAPTER 7

Tensor-Fu-1

7.1 Exercise 1

Task: create a tensor for prototyping using **'torch.randn'_**.

```
import torch
import torch.nn as nn
```

7.2 Exercise 2

Task: Create a linear layer which works wih x2dim

```
import torch
import torch.nn as nn

x2dim = torch.randn(9, 10)

# required and default parameters:
# fc = nn.Linear(in_features, out_features)
```

7.3 Exercise 3

Task: Create a convolution which works on x3dim

```
import torch
import torch.nn as nn

x3dim = torch.randn(9, 10, 11)
```

(continues on next page)

(continued from previous page)

```
# required and default parameters:
# conv1 = nn.Conv1d(in_channels, out_channels, kernel_size, stride=1, padding=0)
```

CHAPTER 8

Tensor-Fu-2

8.1 Exercise 1

Task: The code below is not quite right for prototyping purposes. Fix it so that the indices is more like an actual batched data point and has batch_size=30 and seq_length=10.

```
indices = torch.from_numpy(np.random.randint(0, 100, size=(10,)))

emb = nn.Embedding(num_embeddings=100, embedding_dim=16)
assert emb(indices).shape == (30, 10, 16)
```

8.2 Exercise 2

Task: Create a MultiEmbedding class which can input two sets of indices, embed them, and concat the results!

```
class MultiEmbedding(nn.Module):
    def __init__(self, num_embeddings1, num_embeddings2, embedding_dim1, embedding_
↪dim2):
        pass

    def forward(self, indices1, indices2):
        # use something like
        # z = torch.cat([x, y], dim=1)

        pass


# testing

# use indices method from above
# the batch dimensions should agree
```

(continues on next page)

(continued from previous page)

```
# indices1 =
# indices2 =
# multiemb = MutliEmbedding(num_emb1, num_emb2, size_emb1, size_emb2)
# output = multiemb(indices1, indices2)
# print(output.shape) # should be (batch, size_emb1 + size_emb2)
```

CHAPTER 9

Choose Your Own Adventure

The Choose Your Own Adventures have been structured to allow for pure model exploration without worrying as much about the dataset or the training routine. In each notebook, you will find an implemented dataset loading routine as well as an implemented training routine. These two parts should seem familiar to you given the last 2 days of content. What is not implemented is a model definition nor its instantiation.

There are kinds of NLP Task you could solve:

- **Classification**
  - (day_2/CYOA-Amazon-Reviews) Sentiment Analysis with Amazon Reviews
  - (day_2/CYOA-Movie-Reviews) Sentiment Analysis with Movie Reviews
  - (day_2/CYOA-Surname-Classification) Surname Classication
  - (day_2/CYOA-Twitter-Language-ID) Twitter Language ID
  - (day_2/CYOA-CFPB-Classification) CFPB Classification
  - News categorization with NLTK's Brown corpus
  - Question Classification with NLTK's question dataset
- **Sequence Prediction**
  - Language Model with NLTK's Shakespeare dataset
- **Sequence Tagging**
  - NER with NLTK's CoNLL NER dataset
  - Chunking with NLTK's CoNLL Chunking dataset

The tasks that have notebooks are indicated. We did not include notebooks for every task to narrow the scope of the in-class work.

9.1 Strategies for Model Exploration

9.1.1 Identifying the I/O

A good place to start when doing model exploration is by defining the input-output program that the model is intended to solve.

If you look at the previous models, you will notice the following pattern:

- Each model starts with embedding the inputs
- Each model ends with applying a Linear layer to create the correct output size

These are the input and output of the models.

9.1.2 Fail Fast Prototyping

Use the dataset to get a single batch and the input data from that batch. You can use that sample input data to prototype an approach to solving the I/O problem.

```
batch = next(iter(DataLoader(dataset, batch_size=4)))
print(module_to_test(batch['x_data']).shape)
```

9.1.3 Three simple models to try

1. **Continuous Bag of Words (CBOW)**
    - CBOW is a model which has the following structure: embed the tokens, pool their embeddings in some way, classify the resulting vector. One way of pooling is to just average them. Others could include taking the max or summing. A Linear layer is then used to compute the classification vector.
2. **Text Convolutional Neural Network (CNN)**
    - CNN will learn spatially invariant patterns because it applies its weights as a sliding window over the input. You can keep applying more and more CNNs (as in the Chinese Document example), or you could apply one or two and then pool in the same way as the CBOW. Once you have a single vector for each data point, a final Linear layer is used to compute the classification vector.
3. **Recurrent Neural Network (RNN)**
    - Whether the character or word variants, an RNN learns a sequence model of its inputs. In doing classification, the final vector of the sequence is used to represent the entire sequence. Then, this vector is optionally passed through a couple Linear layers (which themselves can be grouped and described as a Multilayer Perceptron). Finally, whether the final RNN vector is passed through a Multilayer Perceptron or not, a final Linear layer is used to compute the classification output.

9.1.4 More complicated models to try

1. **Better RNNs (Gated Recurrent Unit (GRU), Long Short Term Memory (LSTM))**
    - Instead of using the simple RNN provided, use an RNN variant that has gating like GRU or LSTM
2. **BiRNN**
    - Use the bi-directional RNN model from the day_2 notebook
2. **CNN + RNN**

- One thing you could try is to apply a CNN one or more times to create sequences of vectors that are informed by their neighboring vectors. Then, apply the RNN to learn a sequence model over these vectors. You can use the same method to pull out the final vector for each sequence, but with one caveat. If you apply the CNN in a way that shrinks the sequence dimension, then the indices of the final positions won't quite be right. One way to get around this is to have the CNN keep the sequence dimension the same size. This is done by setting the *padding* argument to be *kernel_size//2*. For example, if *kernel_size=3*, then it should be that *padding=1*. Similarly with *kernel_size=5*, then *padding=2*. The padding is added onto both sides of the sequence dimension.

4. **Using Attention**

   - If you're feeling ambitious, try implementing attention!
   - **One way to do attention is use a Linear layer which maps feature vectors to scalars**
     - We begin with a sequence tensor, x_data, that is embedded, x_embedded_sequence = emb(x_data)
     - The shape here is the similar as the embedded sequence tensor: (batch, sequence, 1)
     - You can use the apply_across_sequence_loop or apply_across_sequence_reshape
   - **A softmax is then used on the scalar to produce a probability vector**
     - attention_weights = F.softmax(attention_weights, dim=2)
   - **The probability vector is broadcast (multiplied) across the sequences, so that it weights each sequence vector**
     - weighted_sequence = attention_weights * x_embedded_sequence
   - **The sequences are the summed over**
     - weighted_sequence.sum(dim=1)

CHAPTER 10

Recipes and PyTorch patterns

In this section, you will find a set of recipes for doing various things with PyTorch.

10.1 Loading Pretrained Vectors

It can be extremely useful to make a model which had as advantageous starting point.

To do this, we can set the values of the embedding matrix.

```
def get_pretrained_embeddings(filename, dim_size, token_vocab):
    embedding_matrix = torch.zeros(len(token_vocab), dim_size)
    all_words = set(token_vocab.keys())

    with open(filename) as fp:
        for line in tqdm_notebook(fp.readlines(), leave=False):
            line = line.split(" ")
            word = line[0]
            if word not in token_vocab:
                continue
            all_words.remove(word)
            row_index = token_vocab[word]
            embedding_matrix[row_index] = torch.FloatTensor([float(x) for x in␣
↪line[1:]])
    for remaining_word in all_words:
        row_index = token_vocab[remaining_word]
        embedding_matrix[row_index] = torch.nn.init.kaiming_normal_(torch.zeros(1,␣
↪dim_size))

    return embedding_matrix
```

Then, we can load that embedding matrix:

```
load_pretrained = True
pretrained_embeddings = None

if load_pretrained:
    pretrained_embeddings = get_pretrained_embeddings("../data/glove.6B.100d.txt",
                                                       dim_size=100,
                                                       token_vocab=dataset.vectorizer.
↪token_vocab)
    embedding_size = pretrained_embeddings.shape[1]
```

And we can use it in an embedding layer:

```
emb = nn.Embedding.from_pretrained(embeddings=pretrained_embeddings, freeze=False,␣
↪padding_idx=0)
```

10.2 Compute Convolution Sizes

```
import math

def conv_shape_helper_1d(input_seq_len, kernel_size, stride=1, padding=0, dilation=1):
    kernel_width = dilation * (kernel_size - 1) + 1
    tensor_size = input_seq_len + 2 * padding
    return math.floor((tensor_size - kernel_width) / stride + 1)
```

10.3 Notes for Convolution Models

Implementing a convolutional model can be tricky. Here are some notes to help get you along:

1. **Convolutions in PyTorch expect the channels to be on the 1st dimension**
    - We treat the feature vectors (typically from an embedding layer) as the channel / kernel dimension
    - If the shape of your tensor is (batch, seq, feature), then this means a permutation is needed to move the (batch, feature, seq)
    - To get an intuition as why, imagine an image. A minibatch of images is (batch, 3, width, height) because an image exists as RGB coordinates for each pixel.
    - This can also be thought of as 3 feature maps of the image
    - In the same way, the feature dimension of a sequence can be separate feature maps for the entire sequence
    - **The consequence of all of this is a required permute operation post-embedding but pre-convolution: x_embedded.permute(0, 2, 1)**
2. **It is a modeling choice on how to go from the embedded tensor to a final vector, but the goal is to end up with a single final**
    - This means the channel dimension will be our final vector and we want to apply operations to shrink the sequence dimension until it is size=1
    - You could create enough convolutions that eventually it will shrink to size 1
    - This becomes dependent on the max_seq_len (if this changes, the number of convolutions to shrink to size=1 also changes)

- Some sort of pooling or lenght-variation operation is recommended for the final reduction.

10.4 Design Pattern: Attention

Attention is a useful pattern for when you want to take a collection of vectors—whether it be a sequence of vectors representing a sequence of words, or an unordered collections of vectors representing a collection of attributes—and summarize them into a single vector. This has similar analogs to the CBOW examples we saw on Day 1, but instead of just averaging or using max pooling, we are learning a function which learns to compute the weights for each of the vectors before summing them together.

Importantly, the weights that the attention module is learning is a valid probability distribution. This means that weighting the vectors by the value the attention module learns can additionally be seen as computing the Expection. Or, it could as interpolating. In any case, attention's main use is to select 'softly' amongst a set of vectors.

The attention vector has several different published forms. The one below is very simple and just learns a single vector as the attention mechanism.

Using the `new_parameter` function we have been using for the RNN notebooks:

```
def new_parameter(*size):
    out = Parameter(FloatTensor(*size))
    torch.nn.init.xavier_normal(out)
    return out
```

We can then do:

```
class Attention(nn.Module):
    def __init__(self, attention_size):
        super(Attention, self).__init__()
        self.attention = new_parameter(attention_size, 1)

    def forward(self, x_in):
        # after this, we have (batch, dim1) with a diff weight per each cell
        attention_score = torch.matmul(x_in, self.attention).squeeze()
        attention_score = F.softmax(attention_score).view(x_in.size(0), x_in.size(1),␣
↪1)
        scored_x = x_in * attention_score

        # now, sum across dim 1 to get the expected feature vector
        condensed_x = torch.sum(scored_x, dim=1)

        return condensed_x



attn = Attention(100)
x = Variable(torch.randn(16,30,100))
attn(x).size() == (16,100)
```

Hello! This is a directory of resources for a training tutorial to be given at the O'Reilly AI Conference in San Jose on Monday, September 9th, and Tuesday, September 10th.

Please read below for general information. There are 2 repositories you can use for this tutorial: our github repository or the OReilly gitlab repository. Please note that there are two ways to engage in this training (described below).

More information will be added to this site as the training progresses.

CHAPTER 11

General Information

11.1 Prerequisites:

- A working knowledge of Python and the command line
- Familiarity with precalc math (multiply matrices, dot products of vectors, etc.) and derivatives of simple functions.
- A general understanding of machine learning (setting up experiments, evaluation, etc.) (useful but not required)

11.2 Hardware and/or installation requirements:

- **There are two options:**
    1. **Using O'Reilly's online resources**. For this, you only needs a laptop; on the first day, we will provide a URL to use an online computing resource (a JupyterHub instance) provided by O'Reilly. If you have a Safari account, you can use that to log on. Otherwise, you can create a free trial (the free trial button is on the URL we provide). You will be able to access Jupyter notebooks through this and they will persist until the end of the second day of training. This option is not limited by what operating system you have. You will need to have a browser installed.
    2. **Setting everything up locally**. For this, you need a laptop with the PyTorch environment set up. This is only recommended if you want to have the environment locally or have a laptop with a GPU. (If you have trouble following the provided instructions or if you find any mistakes, please file an issue here.)