pytorch-nlp-tutorial Documentation

Brian McMahan and Delip Rao

Oct 09, 2018

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. Install PyTorch

There are instructions on http://pytorch.org which detail how to install it. If you have been following along so far and have Anaconda installed with CUDA enabled, you can simply do:

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

2.3.5 2e. 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-eu2018.git
```

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

2.3.6 2f. 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:

```
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

Migrating to PyTorch 0.4.0

If you have used PyTorch before 0.4.0, some things have changed! To help you understand how to migrate, the PyTorch folks have a wonderful migration guide found here.

CHAPTER 5

Take-Home Exercises

5.1 Exercise 1

Implement Deep Continuous Bag-of-Words (CBOW). Here is a link to the paper!

5.2 Exercise 2

Implement a convnet classifier to classify surnames

At the end of class, we talked about how CNNs can be used to incrementally shrink an intermediate data tensor until a dimension of size 1 is left.

Here is a notebook that I pieced together for you to do this assignment with: https://gist.github.com/braingineer/1d7baecf2c99013d88d4d1db77449aec

Some other points that were made:

1. At first, the size of the data tensor is (batch, max_seq_len). Then, after using the embedding layer, it is (batch, max_seq_len, embeddin_dim). However, as was pointed out, convolutions expect the channel dimension (the features per position in the sequence) to be on the 1st position. So, a conv1d will expect: (batch, feature_dim, max_seq_len).
2. When a sequence/hierarchy of 1D convolutions are applied, they can eventually shrink the sequence dimension to size 1. This is a goal. Specifically, you want (batch, feature_dim, 1) so that use the "squeeze" operation to remove the 1-dimension and have a single feature vector per item in the batch.
3. Once you have the correct sequence of convolutions and/or pooling operations to create your feature vectors, then you can add a Linear layer which will map from the feature vector to a prediction vector. This can be modeled after the other examples.

CHAPTER 6

Solutions

6.1 Problem 1

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

x = torch.autograd.Variable(torch.FloatTensor([1]),
                            requires_grad=True)

y = f(x)
print(y)

y.backward()

x.grad

y.grad_fn
```

6.2 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)
```

(continues on next page)

(continued from previous page)

```
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")
# >> 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 7

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 8

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.

8.1 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 9

Tensor-Fu-1

9.1 Exercise 1

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

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

9.2 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)
```

9.3 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 10

Tensor-Fu-2

10.1 Exercise 1

Task: The code below is broken. How can we fix it?

Hint: The input data tensor (indices) might be the wrong shape.

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

emb = nn.Embedding(num_embeddings=100, embedding_dim=16)
emb(indices)
```

10.2 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.concat([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 11

Small Tidbits

11.1 Set Seed Everywhere

```
import numpy as np
import torch

def set_seed_everywhere(seed, cuda):(
        """Set the seed for numpy and pytorch

        Args:
            seed (int): the seed to set everything to
            cuda (bool): whether to set the cuda seed as well
    """
    np.random.seed(seed)
    torch.manual_seed(seed)
    if cuda:
        torch.cuda.manual_seed_all(seed)
```

CHAPTER 12

Exercise: Fast Lookups for Encoded Sequences

Let's suppose that you want to embed or encode something that you want to look up at a later date. For example, you could be embedded things that need to be identified (such as a song). Or maybe you want to just find the neighbors of a new data point.

In any case, using the approximate nearest neighbors libraries are wonderful for this. For this exercise, we will use Spotify's annoy library (we saw this on day 1, in the pretrained word vector notebook). You should aim to complete the following steps:

1. **Load the network from the Day 2, 01 notebook using the pre-trained weights.**
    - You could use the 02 notebook, but we want to get a single vector per each sequence.
    - So, to use 02, you would need to port the `column_gather` function.
    - One reason why you might be interested in doing this is because the 02 objective function learned a better final vector representation.
2. **Given a loaded network with pre-trained weights, write a function which does nearly exactly what the forward function d**
    - This is because we want the feature vector just before the fully connected.
    - it is common to assume that the penultimate layer has learned more generalizable features than the final layer (which is used in softmax computations and is this used to being normalize inducing a probability distribution).
    - The code for this shoud look something like:

```
def get_penultimate(net, x_in, x_lengths=None):
     x_in = net.emb(x_in)
     x_mid = net.conv(x_in.permute(0, 2, 1)).permute(0, 2, 1)
     y_out = net.rnn(x_in)

     if x_lengths is not None:
         y_out = column_gather(y_out, x_lengths)
     else:
         y_out = y_out[:, -1, :]
```

(continues on next page)

(continued from previous page)

```
    return y_out
```

3. As you get penultimate vectors for each datapoint, store them in spotify's annoy. This requires specifying some label for the vector. Using `vectorizer.surname_vocab.lookup` is how you can retrieve the character for each index value in the network inputs. There are some 'decode' functions in the day 2 02 and 03 notebooks.
4. Once everything is added to spotify's annoy, you can then look up any surname and find the set of nearest neighbors! Kind of cool! this is one way to do the k nearest neighbor classification rule.

CHAPTER 13

Exercise: Interpolating Between Vectors

One fun option for the conditional generation code is to interpolate between the learned hidden vectors.

To do this, first look at the code for sampling given a specific nationality:

```
def sample_n_for_nationality(nationality, n=10, temp=0.8):
    assert nationality in vectorizer.nationality_vocab.keys(), 'not a nationality we␣
↪trained on'
    keys = [nationality] * n
    init_vector = torch.tensor([vectorizer.nationality_vocab[key] for key in keys],␣
↪dtype=torch.int64)
    init_vector = net.conditional_emb(init_vector)
    samples = decode_matrix(vectorizer,
                            sample(net.emb, net.rnn, net.fc,
                               init_vector,
                               make_initial_x(n, vectorizer),
                               temp=temp))
    return list(zip(keys, samples))
```

As you can see, we create a list of keys that is the length of the number of samples we want (n). And we use that list to retrieve the correct index from the vocabulary. Finally, we use that index in the conditional embedding inside the network to get the initial hidden state for the sampler.

To do this exercise, write a function that has the following signature:

```
def interpolate_n_samples_from_two_nationalities(nationality1, nationality2, weight,␣
↪n=10, temp=0.8):
    print('awesome stuff here')
```

This should retrieve the `init_vectors` for two different nationalities. Then, using the weight, combine the init vectors as `weight * init_vector1 + (1 - weight) * init_vector2`.

For fun, after you finish this function, write a for loop which loops over the weight from 0.1 to 0.9 to see how it affects the generation.

CHAPTER 14

Exercise: Sampling from an RNN

The goal of sampling from an RNN is to initialize the sequence in some way, feed it into the recurrent computation, and retrieve the next prediction.

To start, we create the initial vectors:

```
start_index = vectorizer.surname_vocab.start_index
batch_size = 2
# hidden_size = whatever hidden size the model is set to

initial_h = torch.ones(batch_size, hidden_size)
initial_x_index = torch.ones(batch_size).long() * start_index
```

Then, we need to use these vectors to retrieve the next prediction:

```
# model is stored in variable called `net`

x_t = net.emb(initial_x_index)
print(x_t.shape)
h_t = net.rnn._compute_next_hidden(x_t, initial_h)

y_t = net.fc(h_t)
```

Now that we have a prediction vector, we can create a probability distribution and sample from it. Note we include a temperature hyper parameter for controlling how strongly we sample from the distribution (at high temperatures, everything is uniform, at low temperatures below 1, small differences are magnified). The temperature is always greater than 0.

```
temperature = 1.0
y_t = F.softmax(y_t / temperature, dim=1)
x_index_t = torch.multinomial(y_t, 1)[:, 0]
```

Now we can start the cycle over again:

```
x_t = net.emb(x_index_t)
h_t = net.rnn._compute_next_hidden(x_t, h_t)

y_t = net.fc(h_t)
```

Write a for loop which repeats this sequence and appends the x_t variable to a list.

Then, we can do the following:

```
final_x_indices = torch.stack(x_indices).squeeze().permute(1, 0)

# stop here if you don't know what cpu, data, and numpy do. Ask away!
final_x_indices = final_x_indices.cpu().detach().numpy()

# loop over the items in the batch
results = []
for i in range(len(final_x_indices)):
    tokens = []
    index_vector = final_x_indices[i]
    for x_index in index_vector:
        if vectorizer.surname_vocab.start_index == x_index:
            continue
        elif vectorizer.surname_vocab.end_index == x_index:
            break
        else:
            token = vectorizer.surname_vocab.lookup(x_index)
            tokens.append(token)

sampled_surname = "".join(tokens)
results.append(sampled_surname)
tokens = []
```

Hello! This is a directory of resources for a training tutorial to be given at the O'Reilly AI Conference in London on Monday, October 8th, and Tuesday, October 9th.

Please read below for general information. You can find the github repository at this link. 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. Specifically, we will be adding a 'recipes' section, 'errata' section, and a 'bonus exercise' section as the training progresses!

CHAPTER 15

General Information

15.1 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 (If you are new to linear algebra, this video course is handy.)
- A general understanding of machine learning (setting up experiments, evaluation, etc.) (useful but not required)

15.2 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 you with credentials and a URL to use an online computing resource (a JupyterHub instance) provided by O'Reilly. 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.)