PyTorch Tutorials

PyTorch

0.4.0

Search docs

BEGINNER TUTORIALS

- Deep Learning with PyTorch: A 60 Minute Blitz
- PyTorch for former Torch users
- Learning PyTorch with Examples
- Transfer Learning tutorial
- Data Loading and Processing Tutorial
- Deep Learning for NLP with Pytorch
  - Introduction to PyTorch
  - Deep Learning with PyTorch
  - Word Embeddings: Encoding Lexical Semantics
    - Getting Dense Word Embeddings
    - Word Embeddings in Pytorch
    - An Example: N-Gram Language Modeling
    - Exercise: Computing Word Embeddings: Continuous Bag-of-Words
  - Sequence Models and Long-Short Term Memory Networks
  - Advanced: Making Dynamic Decisions and the Bi-LSTM CRF

INTERMEDIATE TUTORIALS

- Classifying Names with a Character-Level RNN
- Generating Names with a Character-Level RNN
- Translation with a Sequence to

Word Embeddings: Encoding Lexical Semantics

Word embeddings are dense vectors of real numbers, one per word in your vocabulary. In NLP, it is almost always the case that your features are words! But how should you represent a word in a computer? You could store its ascii character representation, but that only tells you what the word *is*, it doesn't say much about what it *means* (you might be able to derive its part of speech from its affixes, or properties from its capitalization, but not much). Even more, in what sense could you combine these representations? We often want dense outputs from our neural networks, where the inputs are $|V|$ dimensional, where V is our vocabulary, but often the outputs are only a few dimensional (if we are only predicting a handful of labels, for instance). How do we get from a massive dimensional space to a smaller dimensional space?

How about instead of ascii representations, we use a one-hot encoding? That is, we represent the word w by

$$\overbrace{\left[0, 0, \dots, 1, \dots, 0, 0 \right]}^\text{|V| elements}$$

where the 1 is in a location unique to w. Any other word will have a 1 in some other location, and a 0 everywhere else.

There is an enormous drawback to this representation, besides just how huge it is. It basically treats all words as independent entities with no relation to each other. What we really want is some notion of *similarity* between words. Why? Let's see an example.

Suppose we are building a language model. Suppose we have seen the sentences

- The mathematician ran to the store.
- The physicist ran to the store.
- The mathematician solved the open problem.

in our training data. Now suppose we get a new sentence never before seen in our training data:

- The physicist solved the open problem.

Our language model might do OK on this sentence, but wouldn't it be much better if we could use the following two facts:

- We have seen mathematician and physicist in the same role in a sentence. Somehow they have a semantic relation.
- We have seen mathematician in the same role in this new unseen sentence as we are now seeing physicist.

and then infer that physicist is actually a good fit in the new unseen sentence? This is what we mean by a notion of similarity: we mean *semantic similarity*, not simply having similar orthographic representations. It is a technique to combat the sparsity of linguistic data, by connecting the dots between what we have seen and what we haven't. This example of course relies on a fundamental linguistic assumption: that words appearing in similar contexts are related to each other semantically. This is called the distributional hypothesis.

Getting Dense Word Embeddings

How can we solve this problem? That is, how could we actually encode semantic similarity in words? Maybe we think up some semantic attributes. For example, we see that both mathematicians and physicists can run, so maybe we give these words a high score for the "is able to run" semantic attribute. Think of some other attributes, and imagine what you might score some common words on those attributes.

If each attribute is a dimension, then we might give each word a vector, like this:

$$q_\text{mathematician} = \left[\overbrace{2.3}^\text{can run}, \overbrace{9.4}^\text{likes coffee}, \overbrace{-5.5}^\text{majored in Physics}, \dots \right]$$

$$q_\text{physicist} = \left[\overbrace{2.5}^\text{can run}, \overbrace{9.1}^\text{likes coffee}, \overbrace{6.4}^\text{majored in Physics}, \dots \right]$$

Then we can get a measure of similarity between these words by doing:

$$\text{Similarity}(\text{physicist}, \text{mathematician}) = q_\text{physicist} \cdot q_\text{mathematician}$$

Although it is more common to normalize by the lengths:

$$\text{Similarity}(\text{physicist}, \text{mathematician}) = \frac{q_\text{physicist} \cdot q_\text{mathematician}}{\| q_{\backslash\text{physicist}} \| \| q_\text{mathematician} \|} = \cos(\phi)$$

Where ϕ is the angle between the two vectors. That way, extremely similar words (words whose embeddings point in the same direction) will have similarity 1. Extremely dissimilar words should have similarity -1.

You can think of the sparse one-hot vectors from the beginning of this section as a special case of these new vectors we have defined, where each word basically has similarity 0, and we gave each word some unique semantic attribute. These new vectors are *dense*, which is to say their entries are (typically) non-zero.

But these new vectors are a big pain: you could think of thousands of different semantic attributes that might be relevant to determining similarity, and how on earth would you set the values of the different attributes? Central to the idea of deep learning is that the neural network learns representations of the features, rather than requiring the programmer to design them herself. So why not just let the word embeddings be parameters in our model, and then be updated during training? This is exactly what we will do. We will have some *latent semantic attributes* that the network can, in principle, learn. Note that the word embeddings will probably not be interpretable. That is, although with our hand-crafted vectors above we can see that mathematicians and physicists are similar in that they both like coffee, if we allow a neural network to learn the embeddings and see that both mathematicians and physicists have a large value in the second dimension, it is not clear what that means. They are similar in some latent semantic dimension, but this probably has no interpretation to us.

In summary, **word embeddings are a representation of the *semantics* of a word, efficiently encoding semantic information that might be relevant to the task at hand**. You can embed other things too: part of speech tags, parse trees, anything! The idea of feature embeddings is central to the field.

Word Embeddings in Pytorch

Before we get to a worked example and an exercise, a few quick notes about how to use embeddings in Pytorch and in deep learning programming in general. Similar to how we defined a unique index for each word when making one-hot vectors, we also need to define an index for each word when using embeddings. These will be keys into a lookup table. That is, embeddings are stored as a $|V| \times D$ matrix, where D is the dimensionality of the embeddings, such that the word assigned index i has its embedding stored in the i'th row of the matrix. In all of my code, the mapping from words to indices is a dictionary named word_to_ix.

The module that allows you to use embeddings is torch.nn.Embedding, which takes two arguments: the vocabulary size, and the dimensionality of the embeddings.

To index into this table, you must use torch.LongTensor (since the indices are integers, not floats).

```
# Author: Robert Guthrie

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

torch.manual_seed(1)
```

```
word_to_ix = {"hello": 0, "world": 1}
embeds = nn.Embedding(2, 5)  # 2 words in vocab, 5 dimensional embeddings
lookup_tensor = torch.tensor([word_to_ix["hello"]], dtype=torch.long)
hello_embed = embeds(lookup_tensor)
print(hello_embed)
```

Out:

```
tensor([[ 0.6614,  0.2669,  0.0617,  0.6213, -0.4519]])
```

An Example: N-Gram Language Modeling

Recall that in an n-gram language model, given a sequence of words w, we want to compute

$$P(w_i | w_{i-1}, w_{i-2}, \dots, w_{i-n+1})$$

Where w_i is the ith word of the sequence.

In this example, we will compute the loss function on some training examples and update the parameters with backpropagation.

```
CONTEXT_SIZE = 2
EMBEDDING_DIM = 10
# We will use Shakespeare Sonnet 2
test_sentence = """When forty winters shall besiege thy brow,
And dig deep trenches in thy beauty's field,
Thy youth's proud livery so gazed on now,
Will be a totter'd weed of small worth held:
Then being asked, where all thy beauty lies,
Where all the treasure of thy lusty days;
To say, within thine own deep sunken eyes,
Were an all-eating shame, and thriftless praise.
How much more praise deserv'd thy beauty's use,
If thou couldst answer 'This fair child of mine
Shall sum my count, and make my old excuse,'
Proving his beauty by succession thine!
This were to be new made when thou art old,
And see thy blood warm when thou feel'st it cold.""".split()
# we should tokenize the input, but we will ignore that for now
# build a list of tuples.  Each tuple is ([ word_i-2, word_i-1 ], target word)
trigrams = [([test_sentence[i], test_sentence[i + 1]], test_sentence[i + 2])
            for i in range(len(test_sentence) - 2)]
# print the first 3, just so you can see what they look like
print(trigrams[:3])

vocab = set(test_sentence)
word_to_ix = {word: i for i, word in enumerate(vocab)}


class NGramLanguageModeler(nn.Module):

    def __init__(self, vocab_size, embedding_dim, context_size):
        super(NGramLanguageModeler, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.linear1 = nn.Linear(context_size * embedding_dim, 128)
        self.linear2 = nn.Linear(128, vocab_size)

    def forward(self, inputs):
        embeds = self.embeddings(inputs).view((1, -1))
        out = F.relu(self.linear1(embeds))
        out = self.linear2(out)
        log_probs = F.log_softmax(out, dim=1)
        return log_probs


losses = []
loss_function = nn.NLLLoss()
model = NGramLanguageModeler(len(vocab), EMBEDDING_DIM, CONTEXT_SIZE)
optimizer = optim.SGD(model.parameters(), lr=0.001)

for epoch in range(10):
    total_loss = 0
    for context, target in trigrams:

        # Step 1. Prepare the inputs to be passed to the model (i.e, turn the words
        # into integer indices and wrap them in tensors)
        context_idxs = torch.tensor([word_to_ix[w] for w in context], dtype=torch.long)

        # Step 2. Recall that torch *accumulates* gradients. Before passing in a
        # new instance, you need to zero out the gradients from the old
        # instance
        model.zero_grad()

        # Step 3. Run the forward pass, getting log probabilities over next
        # words
        log_probs = model(context_idxs)

        # Step 4. Compute your loss function. (Again, Torch wants the target
        # word wrapped in a tensor)
        loss = loss_function(log_probs, torch.tensor([word_to_ix[target]], dtype=torch.long))

        # Step 5. Do the backward pass and update the gradient
        loss.backward()
        optimizer.step()

        # Get the Python number from a 1-element Tensor by calling tensor.item()
        total_loss += loss.item()
    losses.append(total_loss)
print(losses)  # The loss decreased every iteration over the training data!
```

Out:

```
[(['When', 'forty'], 'winters'), (['forty', 'winters'], 'shall'), (['winters', 'shall'], 'bes
[521.4627077579498, 518.9053602218628, 516.3665678501129, 513.8453557491302, 511.3382725715
```

Exercise: Computing Word Embeddings: Continuous Bag-of-Words

The Continuous Bag-of-Words model (CBOW) is frequently used in NLP deep learning. It is a model that tries to predict words given the context of a few words before and a few words after the target word. This is distinct from language modeling, since CBOW is not sequential and does not have to be probabilistic. Typcially, CBOW is used to quickly train word embeddings, and these embeddings are used to initialize the embeddings of some more complicated model. Usually, this is referred to as *pretraining embeddings*. It almost always helps performance a couple of percent.

The CBOW model is as follows. Given a target word w_i and an N context window on each side, $w_{i-1}, \dots, w_{i-N}$ and $w_{i+1}, \dots, w_{i+N}$, referring to all context words collectively as C, CBOW tries to minimize

$$-\log p(w_i | C) = -\log \text{Softmax}(A(\sum_{w \in C} q_w) + b)$$

where q_w is the embedding for word w.

Implement this model in Pytorch by filling in the class below. Some tips:

- Think about which parameters you need to define.
- Make sure you know what shape each operation expects. Use .view() if you need to reshape.

```
CONTEXT_SIZE = 2  # 2 words to the left, 2 to the right
raw_text = """We are about to study the idea of a computational process.
Computational processes are abstract beings that inhabit computers.
As they evolve, processes manipulate other abstract things called data.
The evolution of a process is directed by a pattern of rules
called a program. People create programs to direct processes. In effect,
we conjure the spirits of the computer with our spells.""".split()

# By deriving a set from `raw_text`, we deduplicate the array
vocab = set(raw_text)
vocab_size = len(vocab)

word_to_ix = {word: i for i, word in enumerate(vocab)}
data = []
for i in range(2, len(raw_text) - 2):
    context = [raw_text[i - 2], raw_text[i - 1],
               raw_text[i + 1], raw_text[i + 2]]
    target = raw_text[i]
    data.append((context, target))
print(data[:5])


class CBOW(nn.Module):

    def __init__(self):
        pass

    def forward(self, inputs):
        pass

# create your model and train.  here are some functions to help you make
# the data ready for use by your module


def make_context_vector(context, word_to_ix):
    idxs = [word_to_ix[w] for w in context]
    return torch.tensor(idxs, dtype=torch.long)


make_context_vector(data[0][0], word_to_ix)  # example
```

Out:

```
[(['We', 'are', 'to', 'study'], 'about'), (['are', 'about', 'study', 'the'], 'to'), (['about
```

Total running time of the script: (0 minutes 0.538 seconds)

Download Python source code: word_embeddings_tutorial.py

Download Jupyter notebook: word_embeddings_tutorial.ipynb

Gallery generated by Sphinx-Gallery

Previous

Next