Pad pack sequences for Pytorch batch processing with DataLoader

Jul 1, 2019

Pytorch setup for batch sentence/sequence processing - minimal working example. The pipeline consists of the following:

1. Convert sentences to ix
2. `pad_sequence` to convert variable length sequence to same size (using dataloader)
3. Convert padded sequences to embeddings
4. `pack_padded_sequence` before feeding into RNN
5. `pad_packed_sequence` on our packed RNN output
6. Eval/reconstruct actual output

1. Convert sentences to ix

Construct word-to-index and index-to-word dictionaries, tokenize words and convert words to indexes. Note the special indexes that we need to reserve for `<pad>`, `EOS`, `<unk>`, `N` (digits). The indexes should correspond to the position of the word-embedding matrix.

2. `pad_sequence` to convert variable length sequences to same size

For the network to take in a batch of variable length sequences, we need to first pad each sequence with empty values (0). This makes every training sentence the same length, and the input to the model is now (N, M), where N is the batch size and M is the longest training instance.

```
from torch import nn
from torch.nn.utils.rnn import pad_sequence
# x_seq = [[5, 18, 29], [32, 100], [699, 6, 9, 17]]
x_padded = pad_sequence(x_seq, batch_first=True, padding_value=0)
# x_padded = [[5, 18, 29, 0], [32, 100, 0, 0], [699, 6, 9, 17]]
```

For batch processing, a typical pattern is to use this with Pytorch's DataLoader and Dataset:

```
from torch.utils.data import Dataset, DataLoader
## refer to pytorch tutorials on how to inherit from Dataset class
dataset = Dataset(...)
data_loader = DataLoader(dataset=dataset, batch_size=32, shuffle=True, collate_fn=p
ad_collate)

def pad_collate(batch):
  (xx, yy) = zip(*batch)
  x_lens = [len(x) for x in xx]
  y_lens = [len(y) for y in yy]

  xx_pad = pad_sequence(xx, batch_first=True, padding_value=0)
  yy_pad = pad_sequence(yy, batch_first=True, padding_value=0)

  return xx_pad, yy_pad, x_lens, y_lens
```

One instance from the traindataset returns (xx, yy) (unpadded), such that when used together with our custom collate function, we get tuples of xxs and yys, and can pad them by batch. Next, enumerate over the dataloader to get the padded sequences and lengths (before padding).

Note: Here we are assuming yy is a target sequence. If yy is just a categorical variable then they are already fixed length for all data instances and there is no need to pad.

3. Convert padded sequences to embeddings

`x_padded` is a (N, M) matrix, and subsequently becomes (N, E, M) where E is the embedding dimension. Note the `vocab_size` should include the special `<pad>`, `<EOS>`, etc characters.

```
embedding = nn.Embedding(vocab_size, embedding_dim)
for (x_padded, y_padded, x_lens, y_lens) in enumerate(data_loader):
  x_embed = embedding(x_padded)
```

4. `pack_padded_sequence` before feeding into RNN

Actually, pack the padded, *embedded* sequences. For pytorch to know how to pack and unpack properly, we feed in the length of the original sentence (before padding). Note we wont be able to pack before embedding. `rnn` can be GRU, LSTM etc.

```
from torch.nn.utils.rnn import pack_padded_sequence
rnn = nn.GRU(embedding_dim, h_dim, n_layers, batch_first=True)
x_packed = pack_padded_sequence(x_embed, x_lens, batch_first=True, enforce_sorted=F
alse)
output_packed, hidden = rnn(x_packed, hidden)
```

The `x_packed` and `output_packed` are formats that the pytorch rnns can read and ignore the padded inputs when calculating gradients for backprop. We can also `enforce_sorted=True`, which requires input sorted by decreasing length, just make sure the target y are also sorted accordingly.

Note: It is standard to initialise hidden states of the LSTM/GRU cell to 0 for each new sequence. There are of course other ways like random initialisation or learning the initial hidden state which is an active area of research

5. `pad_packed_sequence` on our packed RNN output

This returns our familiar padded output format, with (N, M_{out}, H) where M_{out} is the length of the longest sequence, and the length of each sentence is given by `output_lengths`. H is the RNN hidden dimension.

```
from torch.nn.utils.rnn import pad_packed_sequence
output_padded, output_lengths = pad_packed_sequence(output_packed, batch_first=True
)
```

6. Eval/reconstruct actual output

Push the padded output through the final output layer to get (unormalise) scores over the vocabulary space.

Finally we can (1) recover the actual output by taking the argmax and slicing with `output_lengths` and converting to words using our index-to-word dictionary, or (2) directly calculate loss with `cross_entropy` by ignoring index.

```
from torch.nn import functional as F
fc_out = nn.Linear(h_dim, vocab_size)
output_padded = fc_out(output_padded)

batch_ce_loss = 0.0
for i in range(output_padded.size(0)):
  ce_loss = F.cross_entropy(output_padded[i], y[i], reduction="sum", ignore_index=0
)
  batch_ce_loss += ce_loss
```

« Modes of Convergence

Clean TreeLSTMs implementation in PyTorch using NLTK treepositions and Easy-First Parsing »