Encoder

file named fairseq/models/simple\_lstm.py:

**import** **torch.nn** **as** **nn**

**from** **fairseq** **import** utils

**from** **fairseq.models** **import** FairseqEncoder

**class** **SimpleLSTMEncoder**(FairseqEncoder):

**def** \_\_init\_\_(

self, args, dictionary, embed\_dim=128, hidden\_dim=128, dropout=0.1,

):

super().\_\_init\_\_(dictionary)

self.args = args

self.embed\_tokens = nn.Embedding(

num\_embeddings=len(dictionary),

embedding\_dim=embed\_dim,

padding\_idx=dictionary.pad(),

)

self.dropout = nn.Dropout(p=dropout)

self.lstm = nn.LSTM(

input\_size=embed\_dim,

hidden\_size=hidden\_dim,

num\_layers=1,

bidirectional=**False**,

batch\_first=**True**,

)

**def** forward(self, src\_tokens, src\_lengths):

**if** self.args.left\_pad\_source:

*# Convert left-padding to right-padding.*

src\_tokens = utils.convert\_padding\_direction(

src\_tokens,

padding\_idx=self.dictionary.pad(),

left\_to\_right=**True**

)

x = self.embed\_tokens(src\_tokens)

*.*

x = self.dropout(x)

x = nn.utils.rnn.pack\_padded\_sequence(x, src\_lengths, batch\_first=**True**)

\_outputs, (final\_hidden, \_final\_cell) = self.lstm(x)

*.*

**return** {

'final\_hidden': final\_hidden.squeeze(0),

}

**def** reorder\_encoder\_out(self, encoder\_out, new\_order):

*"""*

*Reorder encoder output according to `new\_order`.*

*Args:*

*encoder\_out: output from the ``forward()`` method*

*new\_order (LongTensor): desired order*

*Returns:*

*`encoder\_out` rearranged according to `new\_order`*

*"""*

final\_hidden = encoder\_out['final\_hidden']

**return** {

'final\_hidden': final\_hidden.index\_select(0, new\_order),

}

DECODER

**import** **torch**

**from** **fairseq.models** **import** FairseqDecoder

**class** **SimpleLSTMDecoder**(FairseqDecoder):

**def** \_\_init\_\_(

self, dictionary, encoder\_hidden\_dim=128, embed\_dim=128, hidden\_dim=128,

dropout=0.1,

):

super().\_\_init\_\_(dictionary)

self.embed\_tokens = nn.Embedding(

num\_embeddings=len(dictionary),

embedding\_dim=embed\_dim,

padding\_idx=dictionary.pad(),

)

self.dropout = nn.Dropout(p=dropout)

*.*

self.lstm = nn.LSTM(

input\_size=encoder\_hidden\_dim + embed\_dim,

hidden\_size=hidden\_dim,

num\_layers=1,

bidirectional=**False**,

)

*.*

self.output\_projection = nn.Linear(hidden\_dim, len(dictionary))

**def** forward(self, prev\_output\_tokens, encoder\_out):

*"""*

*Args:*

*prev\_output\_tokens (LongTensor): previous decoder outputs of shape*

*`(batch, tgt\_len)`, for teacher forcing*

*encoder\_out (Tensor, optional): output from the encoder, used for*

*encoder-side attention*

*Returns:*

*tuple:*

*- the last decoder layer's output of shape*

*`(batch, tgt\_len, vocab)`*

*- the last decoder layer's attention weights of shape*

*`(batch, tgt\_len, src\_len)`*

*"""*

bsz, tgt\_len = prev\_output\_tokens.size()

*# Extract the final hidden state from the Encoder.*

final\_encoder\_hidden = encoder\_out['final\_hidden']

x = self.embed\_tokens(prev\_output\_tokens)

*.*

x = self.dropout(x)

x = torch.cat(

[x, final\_encoder\_hidden.unsqueeze(1).expand(bsz, tgt\_len, -1)],

dim=2,

)

initial\_state = (

final\_encoder\_hidden.unsqueeze(0), *# hidden*

torch.zeros\_like(final\_encoder\_hidden).unsqueeze(0), *# cell*

)

output, \_ = self.lstm(

x.transpose(0, 1), *# convert to shape `(tgt\_len, bsz, dim)`*

initial\_state,

)

x = output.transpose(0, 1) *# convert to shape `(bsz, tgt\_len, hidden)`*

*# Project the outputs to the size of the vocabulary.*

x = self.output\_projection(x)

*# Return the logits and ``None`` for the attention weights*

**return** x, **None**