Comment A:

Describe what happens when self.bidirectional is set to True.

Self.bidirectional is a Boolean attribute, defaulting to TRUE, modified by the ‘bidirectional argument’ in the constructor of the LSTMEncoder class (line 93).

If set to TRUE, line 145 runs: defining a function ‘combine\_directions’ with the obligatory argument ‘outs’.

Line 146: Calls the torch.cat function which (according to pytorch.org) ‘concatenates the given sequence of **seq** tensors in the given dimension.’ Here this means that it concatenates the tensor input as ‘outs’ read forwards and backwards **[check this with Ameer send him the line and ask about the slice]**. The tensor object produced has a dimension of 2 (‘dim = 2’)

line 147 calls the function combine\_directions() on the final\_hidden\_states

line 148 calls the function combine\_directions() on the final\_cell\_states

final\_hidden\_states and final\_cell\_states are defined in line 103-107 as the outputs of the pythorch function torch.nn.LSTM (where nn = neural network)

What is the difference between final\_hidden\_states and final\_cell\_states?

final\_hidden\_states is the variable corresponding to the output h\_n of the function self.nn.LSTM (of shape (num\_layers \* num\_directions, batch, hidden\_size): tensor containing the hidden state for t = seq\_len. From pytorch.org).

final\_cell\_states is the varaible correspodning to the output c\_n of the function self.nn.LSTM (of shape (num\_layers \* num\_directions, batch, hidden\_size): tensor containing the cell state for t = seq\_len.)

The hidden state is the ‘output of the cell’ whereas the cell state is the ‘memory of the cell’ (<https://www.reddit.com/r/MLQuestions/comments/65ltu9/lstm_c_and_h_states_whats_the_difference/>).

The cell states are the ‘long-term memory of the model (unique to LSTM)’, and the hidden state is the ‘working memory (also part of RNN models)’. The hidden state is a linear combination of the previous step and the current input, so every hidden step is considered in here during back propagation. Whereas the cell state provides better memory of past events (avoiding vanishing gradient etc), so that they can be brought right the the current step without being altered (‘loaded into working memory’). The cell state runs down the entire chain, with only minor linear interactions, with info added or removed, controlled by gates.

Comment B.

Describe how the attention context vector is calculated.

The attention context vector is generated at line 188:

attn\_context = torch.bmm(attn\_weights, encoder\_out).squeeze(dim=1)

Here the ‘torch.bmm() function ‘Performs a batch matrix-matrix product’ of two matrices, here ‘attn\_weights’ and ‘encoder\_out’.

attn\_weights is generated at line 187:

attn\_weights = F.softmax(attn\_scores, dim=-1)

So attn\_weights is the vector produced by applying a softmax function to the attn\_scores vector **[check that these are actually vectors!]**

In line 188:

The .squeeze method removes all dimensions from the input tensor of size one if input is of shape: (A×1×B×C×1×D) then the out tensor will be of shape: (A×B×C×D)) Because the arg ‘dim=1’ is given: The ‘squeeze’ operation is only conducted in dimension 1, so that (A×1×B) would be shaped to (A×B).

(from <https://pytorch.org/docs/stable/generated/torch.squeeze.html>)

Why do we need to apply a mask to the attention scores?

[from my lecture understanding] The mask is applied because the LSTM is bidirectional. If it were not applied the model could simply ‘cheat’ in predicting the next word, by simply ‘looking’ at that position in the backwards sweep. By masking those positions, the model is forced to actually generate the words at those positions.

Comment C.

How are attention scores calculated?

What role does matrix multiplication (i.e. torch.bmm()) play in aligning encoder and decoder representations?

Comment D.

Describe how the decoder state is initialized.

Assuming that ‘decoder’ state means tgt\_hidden\_states and tgt\_cell\_states:

(line 281) If a ‘cached state’ already exists, i.e. the tgt\_hidden\_states and tgt\_cell\_states have already been generated and a key/value pair corresponding to them exists in the incremental state dictionary, then tgt\_hidden\_states, tgt\_cell\_states, input\_feed are retrieved (via utils.get\_incremental\_state() ) from the instrumental state dictionary.

If the ‘cached state’ does not already exist (‘else’ line 284, i.e. ‘cached-state==None’) then the decoder states (tgt\_hidden\_states, and tgt\_cell\_states) are initialized (line 284 and 285 resp) as tensors with scalar value 0, with a size defined by the first argument: here this is tgt\_inputs.size()[0] in both cases. Each torch.zeros object (tensor) has the ‘width’ (**[not sure about this term?]** self.hidden\_size, so that the tensors created are ‘(tgt\_inputs.size()[0]) x (self.hidden\_size)’. This process is then repeated to create one of these tensor objects for each of the hidden layers in the instance object: for i in range(len(self.layers).

When is cached\_state == None?

cached\_state == None when that state has not already been filled/created by a previous run, and therefore there is no corresponding key-value pair in the incremental state dictionary.

What role does input\_feed play?

Comment E.

Comment F.

Describe what the following lines of code do

In general: (lines 124-131) are implementing (aspects of) (stochastic/batch?)gradient descent:

(line 124): output, \_ = model(sample['src\_tokens'], sample['src\_lengths'], sample['tgt\_inputs'])

Here the output (‘y-hat’) is sampled.

(from line 82:) model = models.build\_model(args, src\_dict, tgt\_dict)

So really line 124 is running models.build\_model(args, src\_dict, tgt\_dict) with sample['src\_tokens'], sample['src\_lengths'], sample['tgt\_inputs'] as the three arguments.

(line126/127) output.view(-1, output.size(-1)), sample['tgt\_tokens'].view(-1)) / len(sample['src\_lengths']

generates the cross-entropy loss on the output via the ‘criterion’ function.

(line 129) grad\_norm = torch.nn.utils.clip\_grad\_norm\_(model.parameters(), args.clip\_norm)

line 129 is performing ‘gradient clipping’. This is a technique to prevent an ‘exploding gradient’. It sets a threshold, above which no gradient can rise.

Line 130: optimizer.step()

optimiser is set up at line 87: optimizer = torch.optim.Adam(model.parameters(), args.lr)

So at line 130, one ‘step’ of the pytorch optimiser method is used to optimize the model parameters, specifically the Adam optimisation tool is used to optimize the learning rate.

Line 131: optimizer.zero\_grad()

In line 131, the gradients are reset to zero, since pytorch accumulates gradients on each ‘pass’ of backpropgation. In order to find the correct loss for each run, it’s necessary to ‘start from zero’ each time.