Name1: Joshua Huang

NetID1: jth239

Name2: Chad Yu

NetID2: cky25

HW3: Semantic Role Labeling using LSTM and Encoder-Decoder Architectures

CS 4740 HW 3 Questions

Q3.1: What are the limitations of converting semantic role labeling task to Question & Answer task (Model 2) using an Encoder-Decoder model? (max 4 sentences)

We present a few points that are limiting factors of the Question-Answer format that we needed to adopt by using an Encoder-Decoder model. Firstly, we note that this Q&A format introduces extra complexity, as we now have to replace a possible semantic role with a longer Q&A instance, which makes the dataset we have to work with larger. Another inefficiency that we notice is that in a similar manner, now we have to work with multiple questions per sentence, and we need to run the model several times for each predicate-sentence combination, while other models may only need to run through the sentence once in a more classical role labeling situation.

Q3.2: In the initialization of your encoder model, you initialize a LSTM encoding layer and two linear projection layers. Give the dimensions of each of these layers and explain your reasoning. Please reference specific parts of your code in your answer. (max 6 sentences)

First, we let `e = embed_size`, `h = hidden_size`. The dimension of the LSTM layer is given by `(h, e)`; this is pretty straightforward, as the LSTM takes in an embedding and its output is a hidden layer.

```
self.encoder = nn.LSTM(
        input_size=embed_size,
        hidden_size=hidden_size,
        batch_first=True,
        bias=True,
        bidirectional=True,
    )
```

Then, the dimension of the `h_projection` is given by `(h, 2h)`. For each token, it takes in the concatenated hidden states from the forward and backward LSTM (which results in a `2h x 1` vector), and outputs the first hidden state of the decoder, which is in $\mathbb{R}^h$.

```
self.h_projection = nn.Linear(
        in_features=2 * hidden_size, out_features=hidden_size, bias=True
    )
```

Similarly, the dimension of the `c_projection` is also `(h, 2h)`, as the cell state is essentially another hidden layer for LSTMs, and we also concatenate the cells states of the forward and backward LSTM as we do for the hidden state.

```
self.c_projection = nn.Linear(
        in_features=2 * hidden_size, out_features=hidden_size, bias=True
    )
```

Q3.3: In the forward step of your encoder, you construct your tensor X by embedding the input. What should the resulting shape be?

If we let `b = batch_size` and `L` be the max length of a sentence, and since we are processing using `batch_first=True` with embedding dimension `e`, passing through the embedding layer will simply add the second dimension `e` to the input, giving a shape of `(b, L, e)`.

Q3.4: In the forward step of your encoder, why are last_hidden and last_cell_state of size (2, b = batch_size, h = hidden_size)? Why is the first dimension 2? Furthermore, why do we want to concatenate the forward and backwards tensors (and what is the shape of the concatenated output)? (3 sentences)

The `last_hidden` and `last_cell_state` are of size `(2, b, h)` because the `nn.LSTM` forward function with `bidirectional=True` outputs the hidden state/cell state tensors of the forward and backward LSTMs stacked on top of each other, and each output tensor of each LSTM has dimension `(b, h)` as the LSTM layer is dimension `(h, e)`, and we are only grabbing the final hidden/cell state for each element in the sequence. We want to concatenate the forward and backward tensor in dimension `1` instead of stacking so that we can match the dimensions of the projection as given by `h_projection` and `c_projection` layers. The ultimate shape of the concatenated output should be `(b, 2h)`.

Q3.5: In the forward step of your encoder, what is the dimension of the resulting tensor when we pass through the h_projection layer? (One sentence)

The dimension of the resulting tensor when passed through the `h_projection` layers is `(b, h)`.

Q3.6: In the forward step of your encoder, what is the dimension of the resulting tensor when we pass through the c_projection layer? (One sentence)

Similarly to `h_projection`, the dimension of the resulting tensor when passed through `c_projection` is `(b, h)`.

Q3.7: In the initialization of your decoder layer, you initialize your actual decoder, and two projection layers. What are the dimensions with which you initialize all of these variables, and why do these dimensions make sense? Please have your answer include no more than two sentences per variable, and please include your code in the answer as you discuss the initialization.

If `e = embed_size` and `h = hidden_size`, our actual decoder, an `nn.LSTMCell`, is `(h, e+h)`. Again, this is pretty easily explainable, as the input to the t^{th} step of the decoder is a concatenation of the embedding for the t^{th} token and a tuple of the previous h^{dec} and c^{dec} states, which are separately handled by the `nn.LSTMCell` module.

```
self.decoder = nn.LSTMCell(
        input_size=embed_size + hidden_size, hidden_size=hidden_size, bias=True
    )
```

The `combined_output_projection` layer has dimension `(h, 3h)`. This is because the combined-output vector is of dimension `(h, 1)`, but the input u_t vector is a concatenation of the decoder hidden state, of dimension `(h, 1)`, and the attention vector, of dimension `(2h, 1)`.

```
self.combined_output_projection = nn.Linear(
        in_features=3 * hidden_size, out_features=hidden_size, bias=False
    )
```

Finally, the `target_vocab_projection` layer has dimension `(v, h)`, where `v = output_vocab_size`. The final output is the probability distribution P_t, which is a softmax over a `(v, 1)` vector, so that from the combined-output vector of shape `(h, 1)`, a `(v, h)` transformation is sufficient.

Q3.8: In the forward step of your decoder you should construct tensor Y by embedding the input. What should the resulting shape of Y be? (1 sentence)

Each index in `source_padded` in decoder's `forward` should generate an embedding vector, so the shape of Y should be `(b, L, e)`, where `L` is the max length of a sentence (the number of time steps).

Q3.9: In the forward step of your decoder, what is the dimension of the resulting tensor when passed through self.att_projection? (One sentence)

As the dimension of `enc_hiddens` passed from the encoder is `(b, L, 2h)`, we have that passing it through `self.att_projection` results in a tensor of size `(b, L, h)`.

Q3.10: In the forward step of your decoder, over what dimension do we iterate, and what should the resulting shape of the new tensor be in relation to b and e where b is batch size and e is embedding size? (One sentence)

We iterate over the second dimension which is the time step, or the max sentence length, so that after squeezing along this dimension, the shape of the new tensor should be `(b, e)`.

Q3.11: In the forward step of your decoder, what is the dimension of Ybar_t? (One sentence)

The dimension of `Ybar_t` should be `(b, e+h)`, as `o_prev` is initialized with size `(b, h)`, and from the last response, `Y_t` has size `(b, e)` after squeezing, and we concatenate over dimension `1`.

Q3.12: At the end of the forward step of your decoder, what is the shape of the single tensor made from stacking combined_outputs? (One sentence)

As mentioned in the documentation of the code, at each time step, the size of `combined_outputs` is `(b, h)`, so that stacking them across all time steps will give us a tensor of size `(L, b, h)`.

Q3.13: Discuss the manipulations to tensor dimensions that you performed in the attention step of the decoder step function (TODO 2), and why these were necessary. Please include direct references to the code you wrote, as well as the code you are referencing. (Maximum 4 sentences)

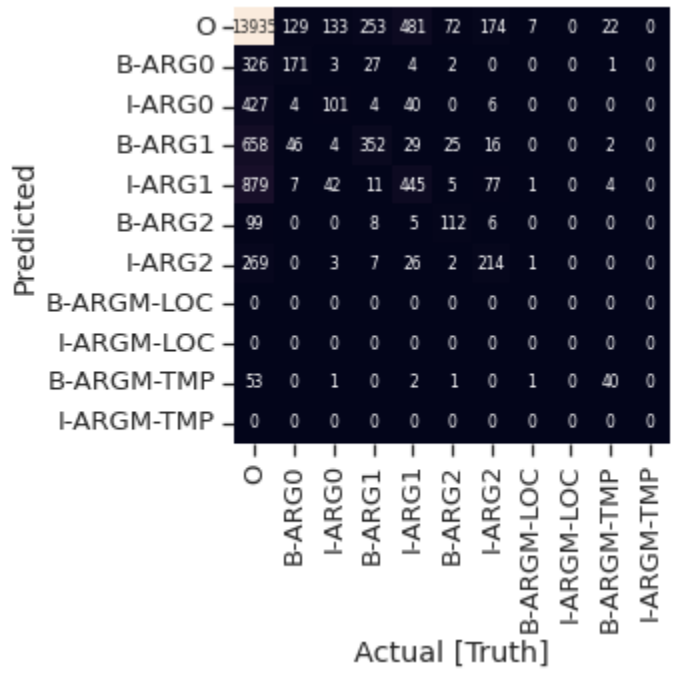
```
alpha_t = F.softmax(torch.matmul(enc_hiddens_proj, dec_hidden.unsqueeze(dim=2)), dim=1)
#alpha_t = (b, L, 1)
a_t = torch.matmul(torch.transpose(enc_hiddens, dim0=1, dim1=2), alpha_t) #a_t = (b, 2h, 1)
a_t = a_t.squeeze(dim=2)
U_t = torch.cat((dec_hidden, a_t), dim=1)
```

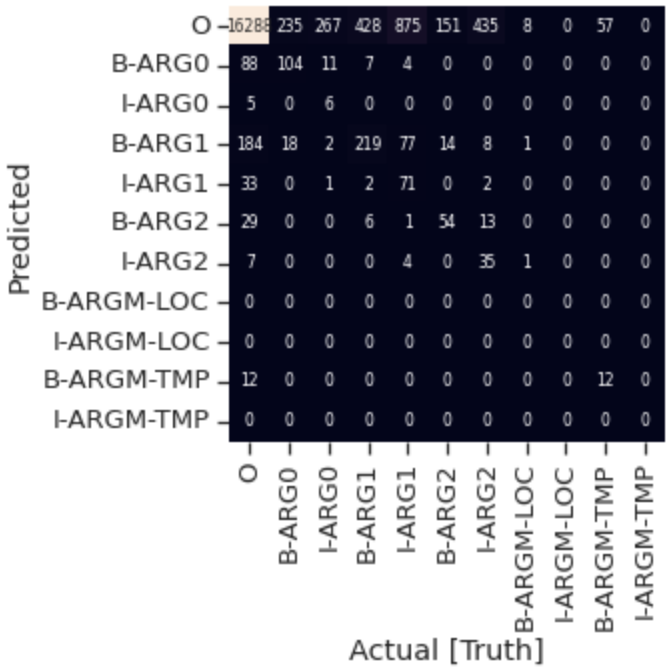
The code from the attention step of the decoder step function is above. We first need to add a dimension to `dec_hidden` because we want to find the dot product between it and `enc_hiddens_proj`, so we need to matrix multiply them. We see that `enc_hiddens_proj` has dimension `(b, L, h)` and `dec_hidden` has dimension `(b, h)`, so we add a dimension to make it

`(b, h, 1)`, then we get `alpha_t` with dimension `(b, L, 1)`. Next, in order to matrix multiply `alpha_t` with `enc_hiddens` where the dimension is `(b, L, 2h)`, we must transpose the second and third dimensions `enc_hiddens` to get a tensor of dimension `(b, 2h, L)`, which now lets us multiply it with `alpha_t`. Now, `a_t` is dimension `(b, 2h, 1)` so we squeeze out the extra dimension to get `a_t` with dimension `(b, 2h)`.

Q3.14: In the decoder step function, what is the dimension of U_t? (One sentence)
The dimension of `U_t` is `(3h, b)`.

Q4.1: Compare two models above either using quantitative or qualitative analysis.





In our experiments, we observed that the LSTM encoder model exhibited an F1 entity score of 0.2377 on the validation set and 0.2427 on the test set. Meanwhile, the encoder-decoder achieved an accuracy of 0.2701 on the validation set and 0.3072 on the test set. Overall, it became evident that the LSTM encoder model generally underperforms in comparison to our encoder-decoder.

Both models encountered errors related to 'O' labels, wherein the model either incorrectly assigned roles as 'O' or labeled true 'O' entities as something else. The LSTM model, in particular, struggled with missing many true 'O' labels, substituting them with different labels. It demonstrated a macro precision score of 0.37 and a recall score of 0.4617. Out of 16,646 true 'O' labels, the model predicted 'O' 15,206 times, correctly labeling 13,935 of them. However, within the 15,206 'O' predictions, 1,271 were incorrect.

On the other hand, the encoder-decoder faced a primary issue of frequently predicting 'O' incorrectly when the true role was different, resulting in a total of 18,744 'O' labels predicted—over 2,000 more than the actual count. It achieved a macro precision score of 0.53 but a recall score of 0.24. Notably, across all categories of true labels, the model tended to predict 'O' the majority of the time. This trend was particularly evident in its poor performance with I-ARG0, where it attained an F1-score of 0.04. Out of 287 true I-ARG0 labels, it only predicted I-ARG0 six times, opting for 'O' in 267 instances and B-ARG0 in 11 instances.