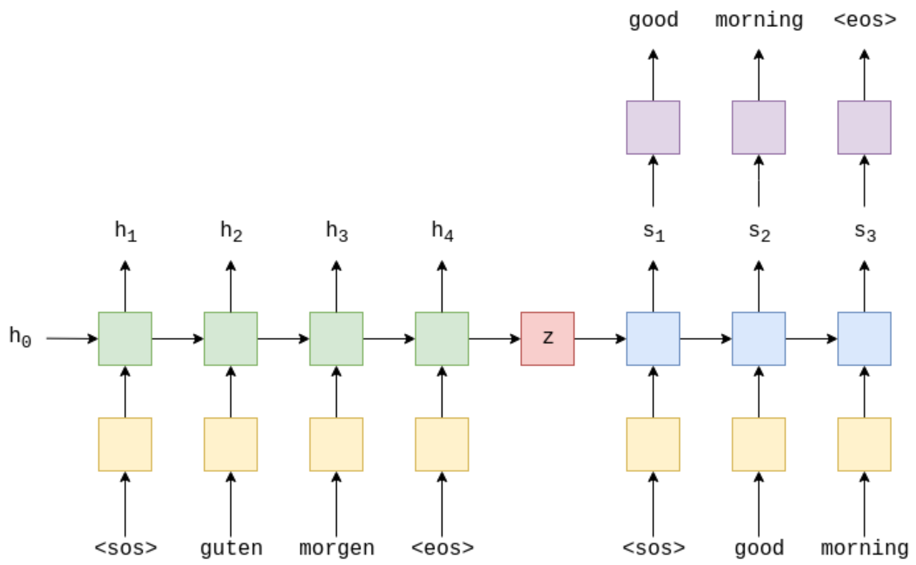
Auto-Encoder ChatBot on WikiQADataset

In this series we'll be building a machine learning model to go from once sequence to another, using PyTorch and torchtext. This will be done on German to English translations, but the models can be applied to any problem that involves going from one sequence to another, such as summarization, i.e. going from a sequence to a shorter sequence in the same language.

In this first notebook, we'll start simple to understand the general concepts by implementing the model from the Sequence to Sequence Learning with Neural Networks paper.

Introduction

The most common sequence-to-sequence (seq2seq) models are *encoder-decoder* models, which commonly use a *recurrent neural network* (RNN) to *encode* the source (input) sentence into a single vector. In this notebook, we'll refer to this single vector as a *context vector*. We can think of the context vector as being an abstract representation of the entire input sentence. This vector is then *decoded* by a second RNN which learns to output the target (output) sentence by generating it one word at a time.



The above image shows an example translation. The input/source sentence, "guten morgen", is passed through the embedding layer (yellow) and then input into the encoder (green). At each time-step, the input to the encoder RNN is both the embedding, e, of the current word, $e(x_t)$, as well as the hidden state from the previous time-step, h_{t-1}, and the encoder RNN outputs a new hidden state h_t. We can think of the hidden state as a vector representation of the sentence so far. The RNN can be represented as a function of both of $e(x_t)$ and h_{t-1}:

$$h_t = \text{EncoderRNN}(e(x_t), h_{t-1})$$

We're using the term RNN generally here, it could be any recurrent architecture, such as an *LSTM* (Long Short-Term Memory) or a *GRU* (Gated Recurrent Unit).

Here, we have $X = \{x_1, x_2, ..., x_T\}$, x_1 = guten$, etc. The initial hidden state, h_0, is usually either initialized to zeros or a learned parameter.

Once the final word, x_T, has been passed into the RNN via the embedding layer, we use the final hidden state, h_T, as the context vector, i.e. $h_T = z$. This is a vector representation of the entire source sentence.

Now we have our context vector, z, we can start decoding it to get the output/target sentence, "good morning". Again, we append start and end of sequence tokens to the target sentence. At each time-step, the input to the decoder RNN (blue) is the embedding, d, of current word, $d(y_t)$, as well as the hidden state from the previous time-step, s_{t-1}, where the initial decoder hidden state, s_0, is the context vector, $s_0 = z = h_T$, i.e. the initial decoder hidden state is the final encoder hidden state. Thus, similar to the encoder, we can represent the decoder as:

$$s_t = \text{DecoderRNN}(d(y_t), s_{t-1})$$

Although the input/source embedding layer, e, and the output/target embedding layer, d, are both shown in yellow in the diagram they are two different embedding layers with their own parameters.

In the decoder, we need to go from the hidden state to an actual word, therefore at each time-step we use s_t to predict (by passing it through a `Linear` layer, shown in purple) what we think is the next word in the sequence, $\hat{y}_t$.

$$\hat{y}_t = f(s_t)$$

The words in the decoder are always generated one after another, with one per time-step. For subsequent inputs $y_{t>1}$, sometimes use the actual, ground truth next word in the sequence, y_t and sometimes use the word predicted by our decoder, $\hat{y}_{t-1}$. This is called *teacher forcing*, see a bit more info about it here.

When training/testing our model, we always know how many words are in our target sentence, so we stop generating words once we hit that many.

Once we have our predicted target sentence, $\hat{Y} = \{\hat{y}_1, \hat{y}_2, ..., \hat{y}_T\}$, we compare it against our actual target sentence, $Y = \{y_1, y_2, ..., y_T\}$, to calculate our loss. We then use this loss to update all of the parameters in our model.

Preparing Data

We'll be coding up the models in PyTorch and using torchtext to help us do all of the pre-processing required. We'll also be using spaCy to assist in the tokenization of the data.

Building the Seq2Seq Model

We'll be building our model in three parts. The encoder, the decoder and a seq2seq model that encapsulates the encoder and decoder and will provide a way to interface with each.

Model Architecture:

Encoder

First, the encoder, a 2 layer LSTM. The paper we are implementing uses a 4-layer LSTM, but in the interest of training time we cut this down to 2-layers. The concept of multi-layer RNNs is easy to expand from 2 to 4 layers.

For a multi-layer RNN, the input sentence, X, after being embedded goes into the first (bottom) layer of the RNN and hidden states, $H = \{h_1, h_2, ..., h_T\}$, output by this layer are used as inputs to the RNN in the layer above. Thus, representing each layer with a superscript, the hidden states in the first layer are given by:

$$h_t^1 = \text{EncoderRNN}^1(e(x_t), h_{t-1}^1)$$

The hidden states in the second layer are given by:

$$h_t^2 = \text{EncoderRNN}^2(h_t^1, h_{t-1}^2)$$

Using a multi-layer RNN also means we'll also need an initial hidden state as input per layer, h_0^l, and we will also output a context vector per layer, z^l.

Without going into too much detail about LSTMs (see this blog post to learn more about them), all we need to know is that they're a type of RNN which instead of just taking in a hidden state and returning a new hidden state per time-step, also take in and return a *cell state*, c_t, per time-step.

$$\begin{aligned} h_t &= \text{RNN}(e(x_t), h_{t-1}) \\ (h_t, c_t) &= \text{LSTM}(e(x_t), h_{t-1}, c_{t-1}) \end{aligned}$$

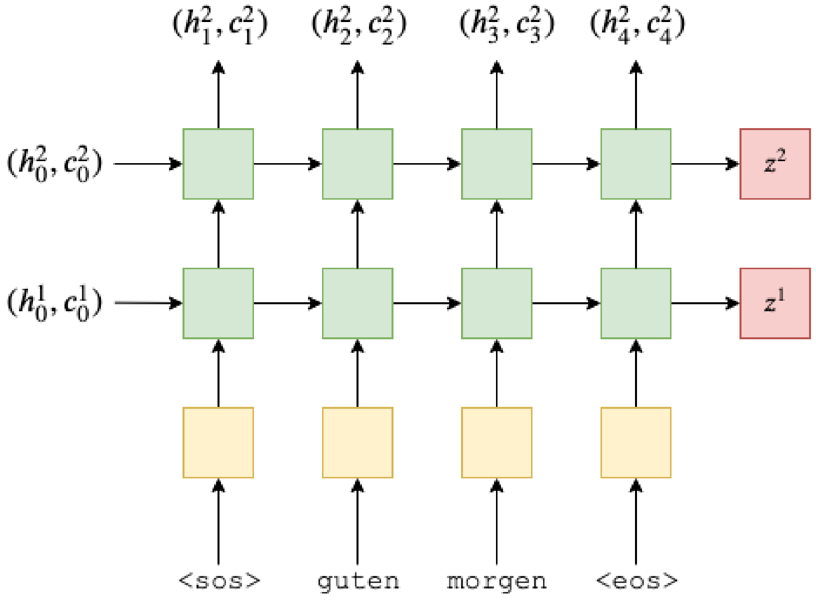
We can just think of c_t as another type of hidden state. Similar to h_0^l, c_0^l will be initialized to a tensor of all zeros. Also, our context vector will now be both the final hidden state and the final cell state, i.e. $z^l = (h_T^l, c_T^l)$.

Extending our multi-layer equations to LSTMs, we get:

$$(h_t^1, c_t^1) = \text{EncoderLSTM}^1(e(x_t), (h_{t-1}^1, c_{t-1}^1))$$
$$(h_t^2, c_t^2) = \text{EncoderLSTM}^2(h_t^1, (h_{t-1}^2, c_{t-1}^2))$$

Note how only our hidden state from the first layer is passed as input to the second layer, and not the cell state.

So our encoder looks something like this:



We create this in code by making an `Encoder` module, which requires we inherit from `torch.nn.Module` and use the `super().__init__()` as some boilerplate code. The encoder takes the following arguments: - `input_dim` is the size/dimensionality of the one-hot vectors that will be input to the encoder. This is equal to the input (source) vocabulary size. - `emb_dim` is the dimensionality of the embedding layer. This layer converts the one-hot vectors into dense vectors with `emb_dim` dimensions. - `hid_dim` is the dimensionality of the hidden and cell states. - `n_layers` is the number of layers in the RNN. - `dropout` is the amount of dropout to use. This is a regularization parameter to prevent overfitting. Check out this for more details about dropout.

We aren't going to discuss the embedding layer in detail during these tutorials. All we need to know is that there is a step before the words - technically, the indexes of the words - are passed into the RNN, where the words are transformed into vectors. To read more about word embeddings, check these articles: 1, 2, 3, 4.

The embedding layer is created using `nn.Embedding`, the LSTM with `nn.LSTM` and a dropout layer with `nn.Dropout`. Check the PyTorch documentation for more about these.

One thing to note is that the `dropout` argument to the LSTM is how much dropout to apply between the layers of a multi-layer RNN, i.e. between the hidden states output from layer l and those same hidden states being used for the input of layer $l+1$.

In the `forward` method, we pass in the source sentence, X, which is converted into dense vectors using the `embedding` layer, and then dropout is applied. These embeddings are then passed into the RNN. As we pass a whole sequence to the RNN, it will automatically do the recurrent calculation of the hidden states over the whole sequence for us! Notice that we do not pass an initial hidden or cell state to the RNN. This is because, as noted in the documentation, that if no hidden/cell state is passed to the RNN, it will automatically create an initial hidden/cell state as a tensor of all zeros.

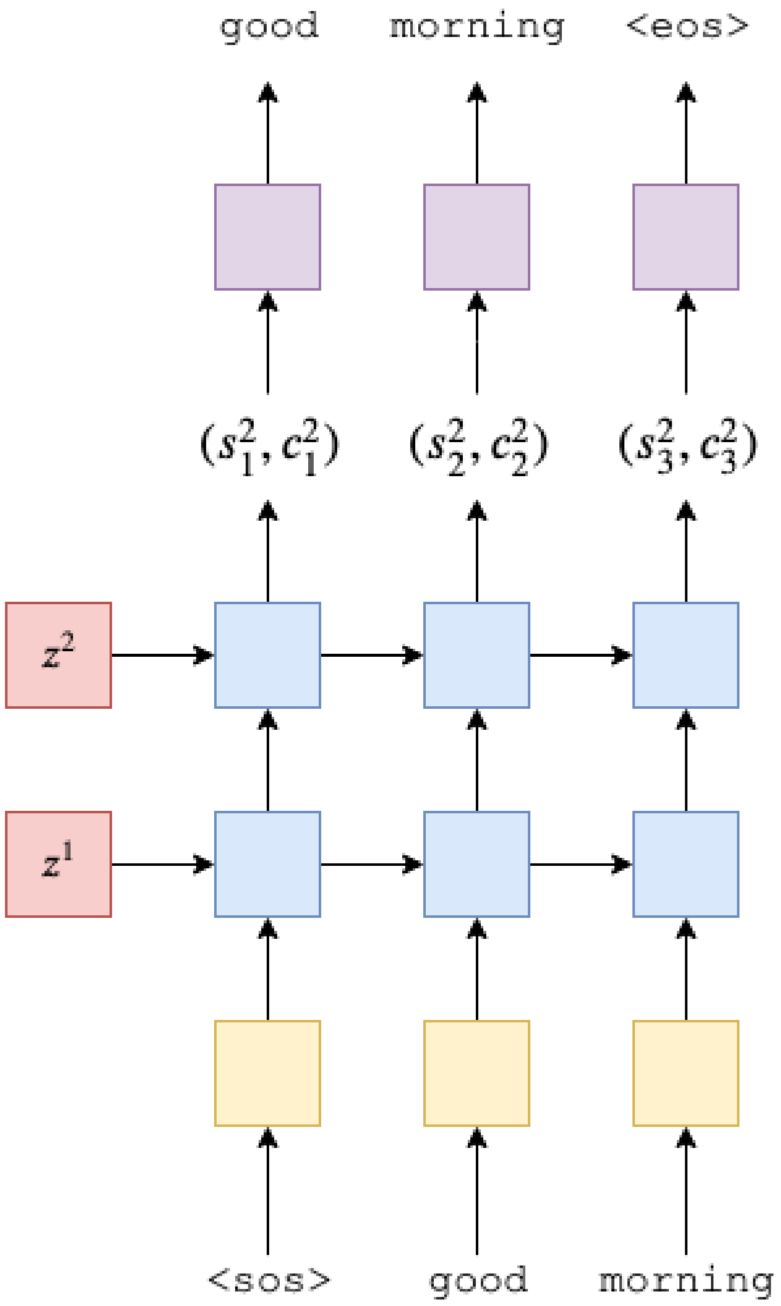
The RNN returns: `outputs` (the top-layer hidden state for each time-step), `hidden` (the final hidden state for each layer, h_T, stacked on top of each other) and `cell` (the final cell state for each layer, c_T, stacked on top of each other).

As we only need the final hidden and cell states (to make our context vector), `forward` only returns `hidden` and `cell`.

The sizes of each of the tensors is left as comments in the code. In this implementation `n_directions` will always be 1, however note that bidirectional RNNs (covered in tutorial 3) will have `n_directions` as 2.

Decoder

Next, we'll build our decoder, which will also be a 2-layer (4 in the paper) LSTM.



The `Decoder` class does a single step of decoding, i.e. it ouputs single token per time-step. The first layer will receive a hidden and cell state from the previous time-step, (s_{t-1}^1, c_{t-1}^1), and feeds it through the LSTM with the current embedded token, y_t, to produce a new hidden and cell state, (s_t^1, c_t^1). The subsequent layers will use the hidden state from the layer below, s_t^{l-1}, and the previous hidden and cell states from their layer, (s_{t-1}^l, c_{t-1}^l). This provides equations very similar to those in the encoder.

$$(s_t^1, c_t^1) = \text{DecoderLSTM}^1(d(y_t), (s_{t-1}^1, c_{t-1}^1))$$
$$(s_t^2, c_t^2) = \text{DecoderLSTM}^2(s_t^1, (s_{t-1}^2, c_{t-1}^2))$$

Remember that the initial hidden and cell states to our decoder are our context vectors, which are the final hidden and cell states of our encoder from the same layer, i.e. $(s_0^l, c_0^l) = z^l = (h_T^l, c_T^l)$.

We then pass the hidden state from the top layer of the RNN, s_t^L, through a linear layer, f, to make a prediction of what the next token in the target (output) sequence should be, $\hat{y}_{t+1}$.

$$\hat{y}_{t+1} = f(s_t^L)$$

The arguments and initialization are similar to the `Encoder` class, except we now have an `output_dim` which is the size of the vocabulary for the output/target. There is also the addition of the `Linear` layer, used to make the predictions from the top layer hidden state.

Within the `forward` method, we accept a batch of input tokens, previous hidden states and previous cell states. As we are only decoding one token at a time, the input tokens will always have a sequence length of 1. We `unsqueeze` the input tokens to add a sentence length dimension of 1. Then, similar to the encoder, we pass through an embedding layer and apply dropout. This batch of embedded tokens is then passed into the RNN with the previous hidden and cell states. This produces an `output` (hidden state from the top layer of the RNN), a new `hidden` state (one for each layer, stacked on top of each other) and a new `cell` state (also one per layer, stacked on top of each other). We then pass the `output` (after getting rid of the sentence length dimension) through the linear layer to receive our `prediction`. We then return the `prediction`, the new `hidden` state and the new `cell` state.

Note: as we always have a sequence length of 1, we could use `nn.LSTMCell`, instead of `nn.LSTM`, as it is designed to handle a batch of inputs that aren't necessarily in a sequence. `nn.LSTMCell` is just a single cell and `nn.LSTM` is a wrapper around potentially multiple cells. Using the `nn.LSTMCell` in this case would mean we don't have to `unsqueeze` to add a fake sequence length dimension, but we would need one `nn.LSTMCell` per layer in the decoder and to ensure each `nn.LSTMCell` receives the correct initial hidden state from the encoder. All of this makes the code less concise - hence the decision to stick with the regular `nn.LSTM`.

Seq2Seq

For the final part of the implemenetation, we'll implement the seq2seq model. This will handle: - receiving the input/source sentence - using the encoder to produce the context vectors - using the decoder to produce the predicted output/target sentence

Our full model will look like this:

The `Seq2Seq` model takes in an `Encoder`, `Decoder`, and a `device` (used to place tensors on the GPU, if it exists).

For this implementation, we have to ensure that the number of layers and the hidden (and cell) dimensions are equal in the `Encoder` and `Decoder`. This is not always the case, we do not necessarily need the same number of layers or the same hidden dimension sizes in a sequence-to-sequence model. However, if we did something like having a different number of layers then we would need to make decisions about how this is handled. For example, if our encoder has 2 layers and our decoder only has 1, how is this handled? Do we average the two context vectors output by the decoder? Do we pass both through a linear layer? Do we only use the context vector from the highest layer? Etc.

Our `forward` method takes the source sentence, target sentence and a teacher-forcing ratio. The teacher forcing ratio is used when training our model. When decoding, at each time-step we will predict what the next token in the target sequence will be from the previous tokens decoded, $\hat{y}_{t+1} = f(s_t^L)$. With probability equal to the teaching forcing ratio (`teacher_forcing_ratio`) we will use the actual ground-truth next token in the sequence as the input to the decoder during the next time-step. However, with probability `1 - teacher_forcing_ratio`, we will use the token that the model predicted as the next input to the model, even if it doesn't match the actual next token in the sequence.

The first thing we do in the `forward` method is to create an `outputs` tensor that will store all of our predictions, $\hat{Y}$.

We then feed the input/source sentence, `src`, into the encoder and receive out final hidden and cell states.

We know how long our target sentences should be (`max_len`), so we loop that many times.

During each iteration of the loop, we: - pass the input, previous hidden and previous cell states (y_t, s_{t-1}, c_{t-1}) into the decoder - receive a prediction, next hidden state and next cell state ($\hat{y}_{t+1}, s_t, c_t$) from the decoder - place our prediction, $\hat{y}_{t+1}$/`output` in our tensor of predictions, $\hat{Y}$/`outputs` - decide if we are going to "teacher force" or not - if we do, the next `input` is the ground-truth next token in the sequence, y_{t+1}/`trg[t]` - if we don't, the next `input` is the predicted next token in the sequence, $\hat{y}_{t+1}$/`top1`, which we get by doing an `argmax` over the output tensor

Once we've made all of our predictions, we return our tensor full of predictions, $\hat{Y}$/`outputs`.

Note: our decoder loop starts at 1, not 0. This means the 0th element of our `outputs` tensor remains all zeros. So our `trg` and `outputs` look something like:

$$\begin{aligned}\text{trg} &= [y_1, y_2, y_3] \\ \text{outputs} &= [0, \hat{y}_1, \hat{y}_2, \hat{y}_3]\end{aligned}$$

Later on when we calculate the loss, we cut off the first element of each tensor to get:

$$\begin{aligned}\text{trg} &= [y_1, y_2, y_3] \\ \text{outputs} &= [\hat{y}_1, \hat{y}_2, \hat{y}_3]\end{aligned}$$

Training the Seq2Seq Model

Now we have our model implemented, we can begin training it.

First, we'll initialize our model. As mentioned before, the input and output dimensions are defined by the size of the vocabulary. The embedding dimesions and dropout for the encoder and decoder can be different, but the number of layers and the size of the hidden/cell states must be the same.

We then define the encoder, decoder and then our Seq2Seq model, which we place on the `device`.