Sequence-to-Sequence Models With Attention

CS 780/880 Natural Language Processing Lecture 20

Samuel Carton, University of New Hampshire

Last lecture

Sequence-to-sequence models

- Main application: translation

Attention

- Improves performance of sequence-to-sequence models
- **Improves interpretability of classifiers**

Model saving/loading

Correction to previous model

```
class EnglishFrenchSeqToSeqModel(pl.LightningModule):
  def __init__(self,
               english_word_vectors:np.ndarray,
               french_word_vectors:np.ndarray,
               english_vocab_size:int,
               french_vocab_size:int,
               learning_rate:float,
               english_padding_id:int,
               french_padding_id:int,
               french_eos_id:int,
               lstm_hidden_size:int=100, # how big the inner vectors of the LSTM will be,
               lstm_layers:int =2, # how many layers the LSTM will have
               dropout_prob:float=0.1,
               loss_print_interval=100,
               **kwargs):
    super().__init__( **kwargs)

    self.english_embeddings = torch.nn.Embedding.from_pretrained(embeddings=torch.tensor(english_word_vectors),
                                                                 freeze=True)
    self.french_embeddings = torch.nn.Embedding.from_pretrained(embeddings=torch.tensor(french_word_vectors),
                                                                freeze=True)
    self.lstm = torch.nn.LSTM(input_size = english_word_vectors.shape[1], # The LSTM will be taking in word vectors
                              hidden_size = lstm_hidden_size,
                              num_layers=lstm_layers,
                              bidirectional=False, # We can't count on being able to proceed both backward and forward
                              dropout=dropout_prob,
                              batch_first=True # This is important. Set to False by default for some reason.
                              )

    # Output layer has to produce one logit per potential word, so the output size is vocab_size
    self.output_layer  = torch.nn.Linear(lstm_hidden_size, french_vocab_size)
    self.lstm_layers = lstm_layers
    self.learning_rate = learning_rate
    self.english_padding_id = english_padding_id
    self.french_padding_id = french_padding_id
    self.french_eos_id = french_eos_id
```

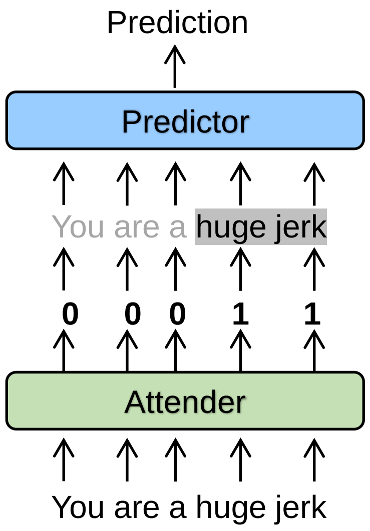
Correction to previous model

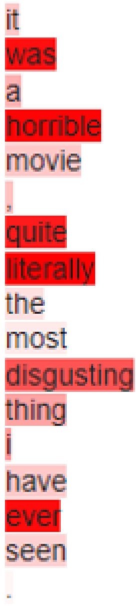
```python
class EnglishFrenchSeqToSeqModel(pl.LightningModule):
  def __init__(self,
               english_word_vectors:np.ndarray,
               french_word_vectors:np.ndarray,
               english_vocab_size:int,
               french_vocab_size:int,
               learning_rate:float,
               english_padding_id:int,
               french_padding_id:int,
               french_eos_id:int,
               lstm_hidden_size:int=100, # how big the inner vectors of the LSTM will be,
               lstm_layers:int =2, # how many layers the LSTM will have
               dropout_prob:float=0.1,
               loss_print_interval=100,
               **kwargs):
    super().__init__( **kwargs)

    self.english_embeddings = torch.nn.Embedding.from_pretrained(embeddings=torch.tensor(english_word_vectors),
                                                                  freeze=True)
    self.french_embeddings = torch.nn.Embedding.from_pretrained(embeddings=torch.tensor(french_word_vectors),
                                                                 freeze=True)
    self.encoder = torch.nn.LSTM(input_size = english_word_vectors.shape[1], # The LSTM will be taking in word vectors
                                 hidden_size = lstm_hidden_size,
                                 num_layers=lstm_layers,
                                 bidirectional=False, # We can't count on being able to proceed both backward and forward
                                 dropout=dropout_prob,
                                 batch_first=True # This is important. Set to False by default for some reason.
                                 )

    self.decoder = torch.nn.LSTM(input_size = french_word_vectors.shape[1], # The LSTM will be taking in word vectors
                                 hidden_size = lstm_hidden_size,
                                 num_layers=lstm_layers,
                                 bidirectional=False, # We can't count on being able to proceed both backward and forward
                                 dropout=dropout_prob,
                                 batch_first=True # This is important. Set to False by default for some reason.
                                 )
```

Reminder: attention

Last week, we learned a model that would incorporate **attention** into **classification**

But how do we apply that to the concept of sequence-to-sequence models?





Sequence-to-sequence with attention

Basic idea: we'll have an additional module in the model whose forward function will take in:

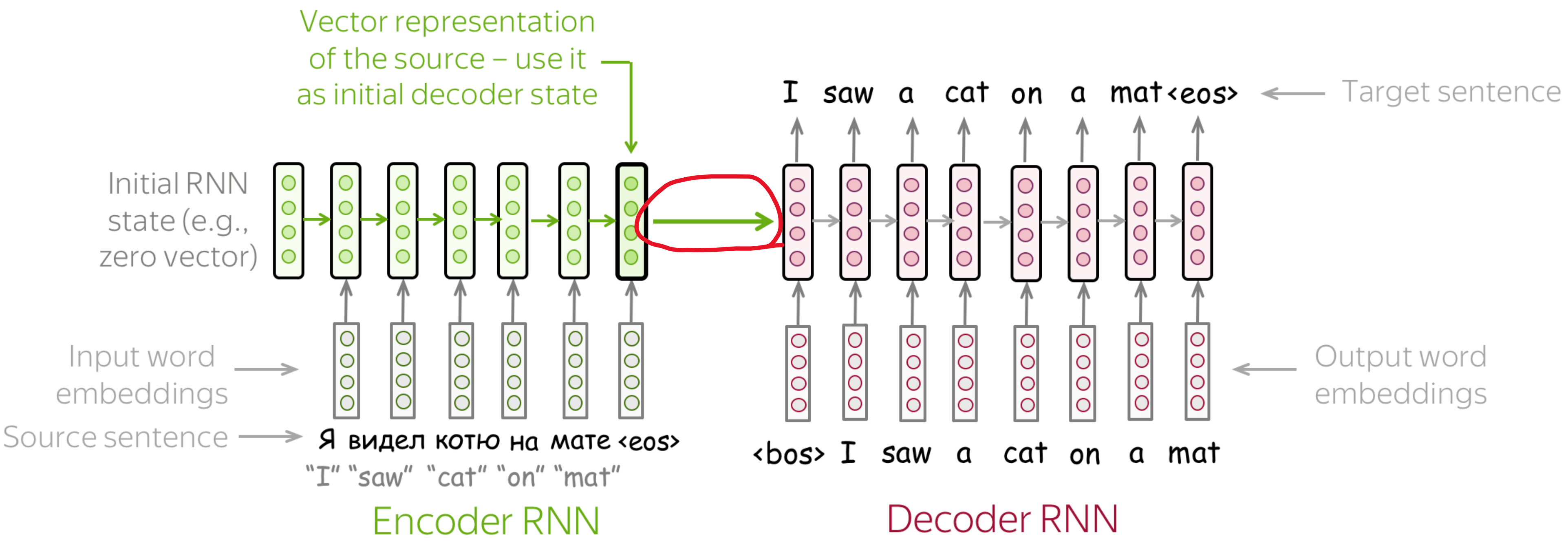
1. A single output hidden-state vector from the decoder
2. All of the output hidden state vectors from the encoder

And then it will:

1. Decide how important each encoder hidden state vector was to this particular decoder hidden state vector
2. Use those importance weights to created a weighted sum of encoder hidden state vectors, called a **context vector**

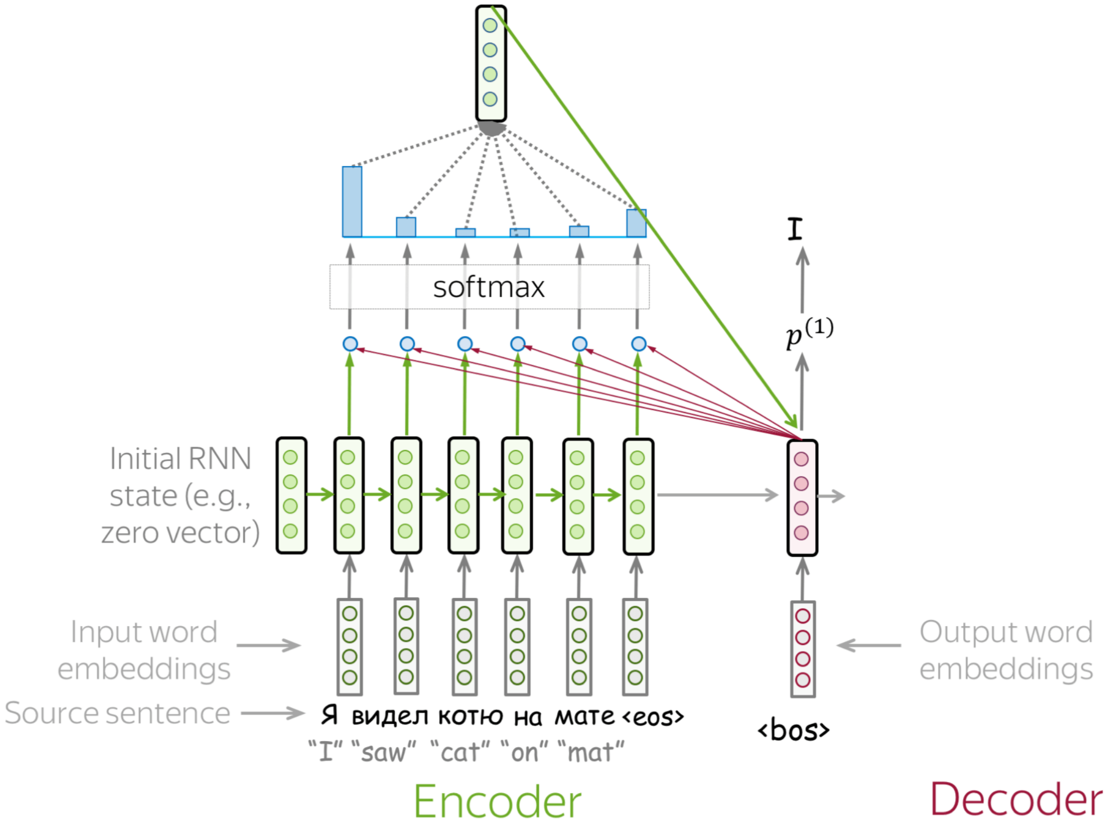
And finally: the next input to the decoder will be concatenated [previous hidden state, context vector]

Ordinary sequence-to-sequence model



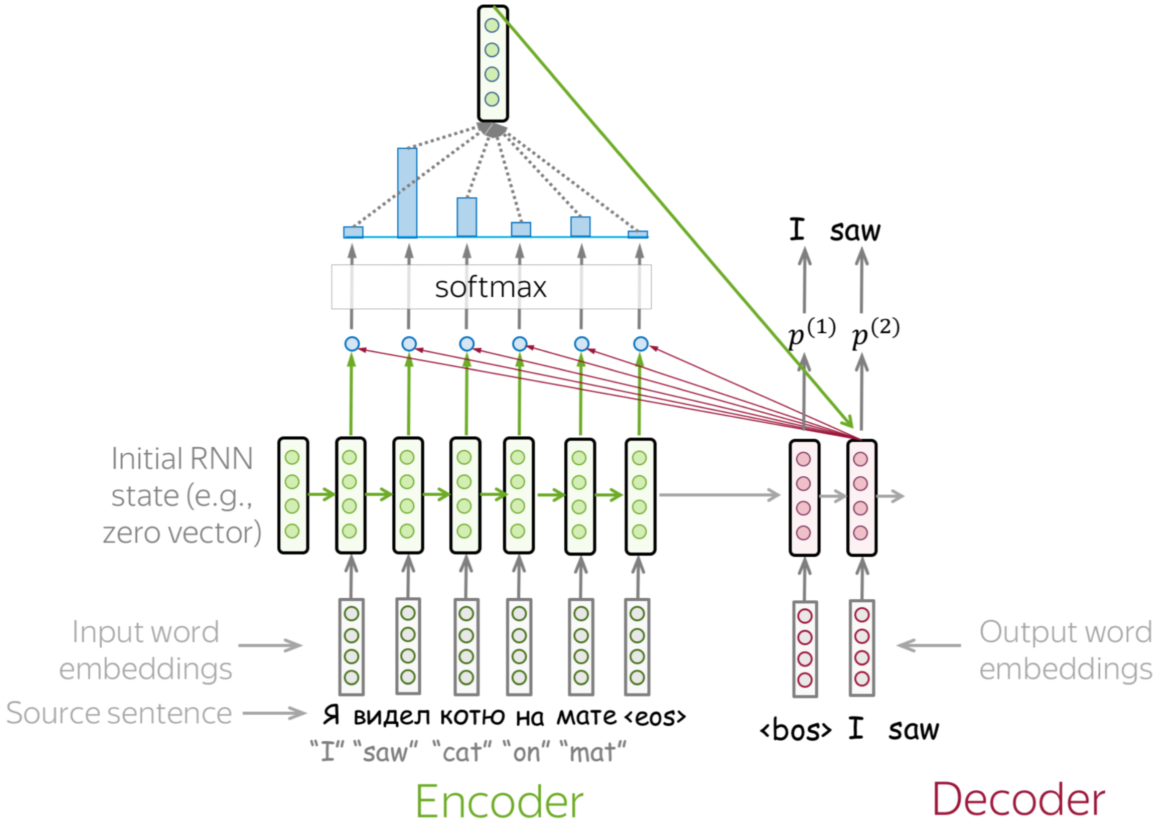
https://lena-voita.github.io/nlp_course/seq2seq_and_attention.html

Sequence-to-sequence with attention



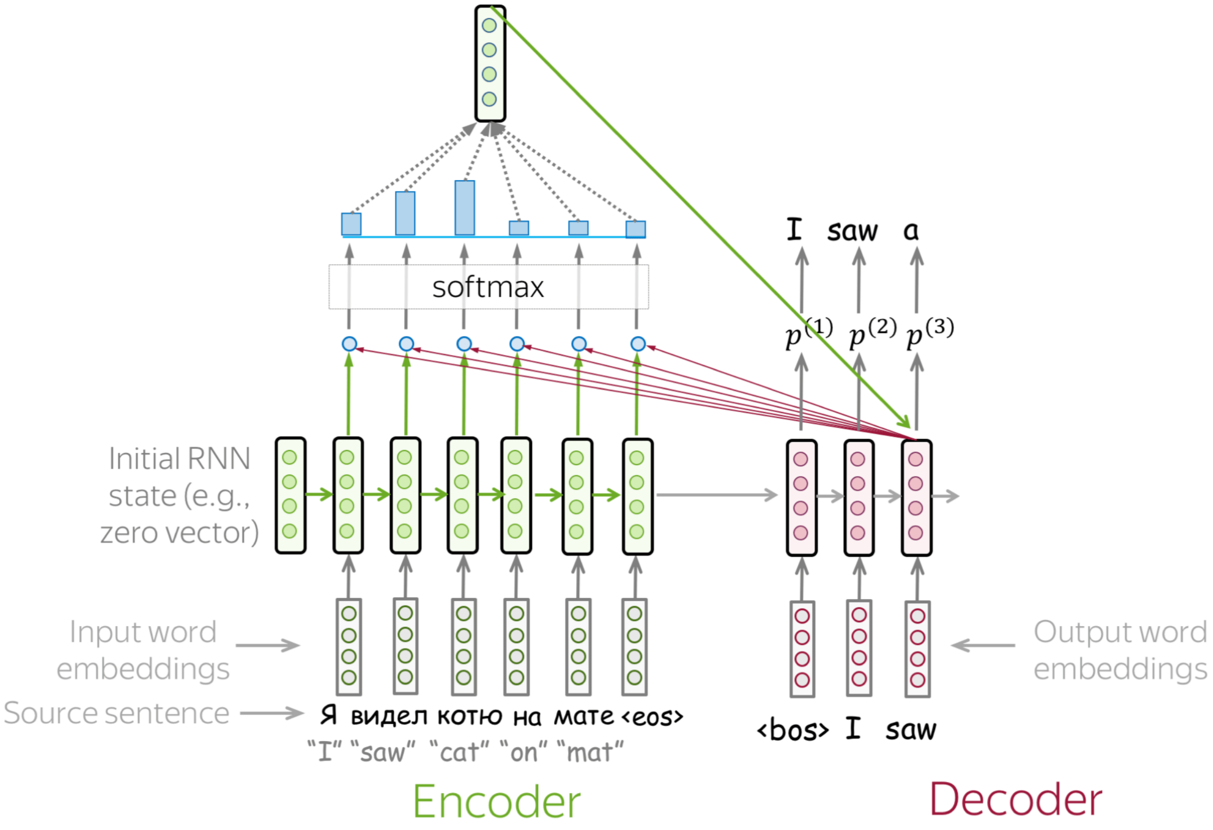
https://lena-voita.github.io/nlp_course/seq2seq_and_attention.html

Sequence-to-sequence with attention



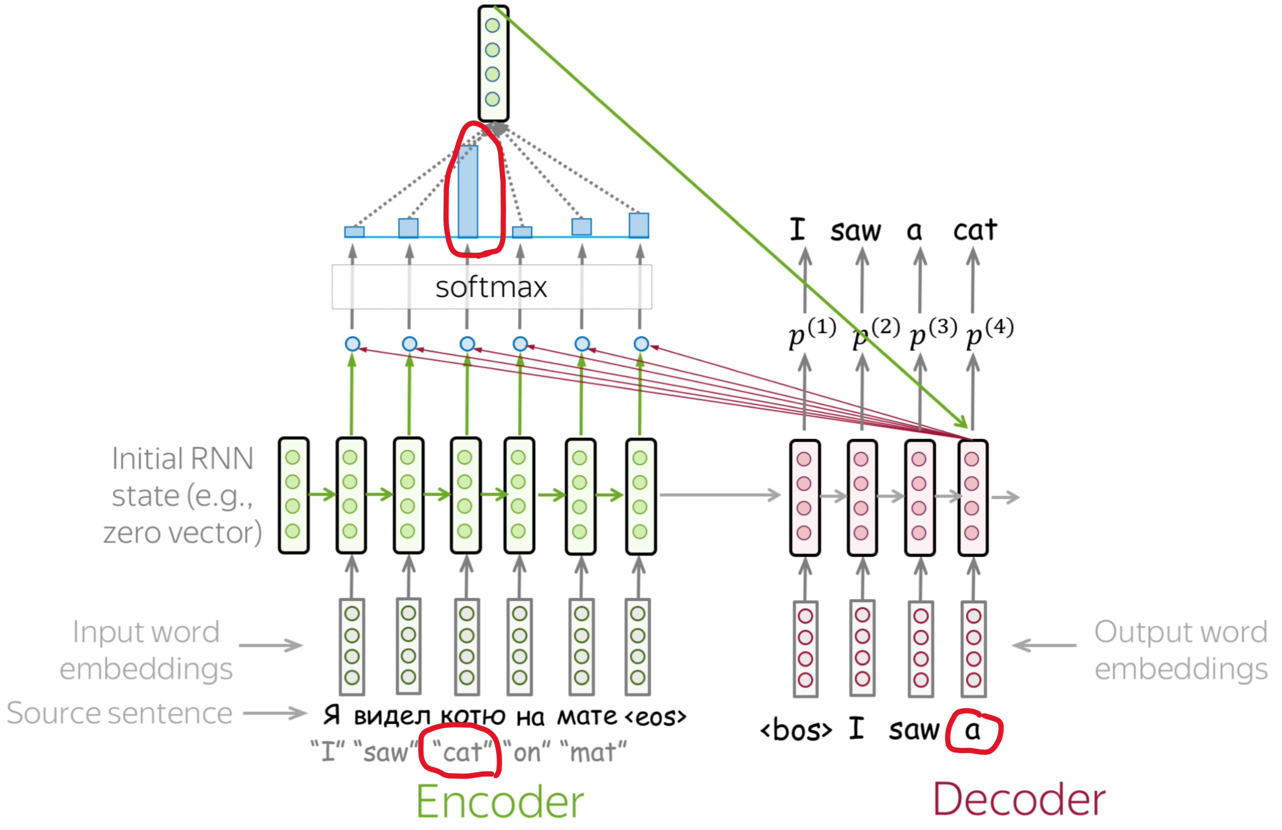
https://lena-voita.github.io/nlp_course/seq2seq_and_attention.html

Sequence-to-sequence with attention



https://lena-voita.github.io/nlp_course/seq2seq_and_attention.html

Sequence-to-sequence with attention



https://lena-voita.github.io/nlp_course/seq2seq_and_attention.html

Preprocessed dataset

```
eng_fra_df.head(10)
```

	english	french	english_tokens	french_tokens	english_token_ids	french_token_ids
0	Go.	Va !	[<sos>, go, ., <eos>]	[<sos>, va, !, <eos>]	[400002, 242, 2, 400003]	[155564, 158, 155562, 155565]
1	Run!	Cours !	[<sos>, run, !, <eos>]	[<sos>, cours, !, <eos>]	[400002, 307, 805, 400003]	[155564, 239, 155562, 155565]
2	Run!	Courez !	[<sos>, run, !, <eos>]	[<sos>, courez, !, <eos>]	[400002, 307, 805, 400003]	[155564, 38881, 155562, 155565]
3	Wow!	Ça alors !	[<sos>, wow, !, <eos>]	[<sos>, ça, alors, !, <eos>]	[400002, 14397, 805, 400003]	[155564, 110, 140, 155562, 155565]
4	Fire!	Au feu !	[<sos>, fire, !, <eos>]	[<sos>, au, feu, !, <eos>]	[400002, 484, 805, 400003]	[155564, 22, 1092, 155562, 155565]
5	Help!	À l'aide !	[<sos>, help, !, <eos>]	[<sos>, à, l'aide, !, <eos>]	[400002, 275, 805, 400003]	[155564, 7, 16685, 155562, 155565]
6	Jump.	Saute.	[<sos>, jump, ., <eos>]	[<sos>, saute, ., <eos>]	[400002, 3106, 2, 400003]	[155564, 11775, 155562, 155565]
7	Stop!	Ça suffit !	[<sos>, stop, !, <eos>]	[<sos>, ça, suffit, !, <eos>]	[400002, 837, 805, 400003]	[155564, 110, 1292, 155562, 155565]
8	Stop!	Stop !	[<sos>, stop, !, <eos>]	[<sos>, stop, !, <eos>]	[400002, 837, 805, 400003]	[155564, 6517, 155562, 155565]
9	Stop!	Arrête-toi !	[<sos>, stop, !, <eos>]	[<sos>, arrête-toi, !, <eos>]	[400002, 837, 805, 400003]	[155564, 155562, 155562, 155565]

Attention module

```python
class BahdanauAttention(nn.Module):
    def __init__(self, hidden_size):
        super(BahdanauAttention, self).__init__()
        self.Wa = nn.Linear(hidden_size, hidden_size)
        self.Ua = nn.Linear(hidden_size, hidden_size)
        self.Va = nn.Linear(hidden_size, 1)

    def forward(self,
                query, # (batch size x 1 x hidden size)
                keys): # (batch size x sequence length x hidden size
        scores = self.Va(torch.tanh(self.Wa(query) + self.Ua(keys)))
        scores = scores.squeeze(2).unsqueeze(1)

        weights = F.softmax(scores, dim=-1)
        context = torch.bmm(weights, keys)

        return context, weights
```

Neural machine translation by jointly learning to align and translate
D Bahdanau, K Cho, Y Bengio - arXiv preprint arXiv:1409.0473, 2014 - arxiv.org
… By letting the decoder have an **attention** mechanism, we relieve the encoder from the burden of having to encode all information in the source sentence into a fixedlength vector. With …
☆ Save 99 Cite Cited by 33746 Related articles All 25 versions

Attention seq-to-seq model: __init__()

```python
class AttentionSeqToSeqModel(pl.LightningModule):
  def __init__(self,
               english_word_vectors:np.ndarray,
               french_word_vectors:np.ndarray,
               english_vocab_size:int,
               french_vocab_size:int,
               learning_rate:float,
               english_padding_id:int,
               french_padding_id:int,
               french_eos_id:int,
               french_sos_id:int,
               lstm_hidden_size:int=100, # how big the inner vectors of the LSTM will be,
               lstm_layers:int =2, # how many layers the LSTM will have
               dropout_prob:float=0.1,
               loss_print_interval=100,
               **kwargs):
    super().__init__( **kwargs)

    self.attention = BahdanauAttention(hidden_size = lstm_hidden_size)

    self.english_embeddings = torch.nn.Embedding.from_pretrained(embeddings=torch.tensor(english_word_vectors),
                                                                 freeze=True)
    self.french_embeddings = torch.nn.Embedding.from_pretrained(embeddings=torch.tensor(french_word_vectors),
                                                                freeze=True)
    self.encoder = torch.nn.LSTM(input_size = english_word_vectors.shape[1], # The LSTM will be taking in word vectors
                                 hidden_size = lstm_hidden_size,
                                 num_layers=lstm_layers,
                                 bidirectional=False, # We can't count on being able to proceed both backward and forward
                                 dropout=dropout_prob,
                                 batch_first=True # This is important. Set to False by default for some reason.
                                 )

    self.decoder = torch.nn.LSTM(input_size = french_word_vectors.shape[1] + lstm_hidden_size, # The LSTM will be taking in word vectors
                             hidden_size = lstm_hidden_size,
                             num_layers=lstm_layers,
                             bidirectional=False, # We can't count on being able to proceed both backward and forward
                             dropout=dropout_prob,
                             batch_first=True # This is important. Set to False by default for some reason.
                             )
```

Attention seq-to-seq model: encode()

```python
def encode(self,
           english_input_ids:torch.Tensor
           ):
  english_embeds = self.english_embeddings(english_input_ids) #(batch size x max english sequence length x embedding size)
  english_padding_mask = (english_input_ids != self.english_padding_id).int() #(batch size x max english sequence length)
  english_input_lengths = english_padding_mask.sum(dim=1).detach().cpu() #(batch size)
  english_packed_embeddings = pack_padded_sequence(english_embeds, english_input_lengths, batch_first=True, enforce_sorted=False)
  english_packed_output, (final_english_hidden, final_english_state) = self.encoder.forward(english_packed_embeddings)
  english_hiddens, _ = pad_packed_sequence(english_packed_output, batch_first=True, padding_value=0.0, total_length=english_input_ids.shape[1])


  return {
    'english_hiddens':english_hiddens, # (batch size x max sequence length x lstm hidden size )
    'final_english_hidden':final_english_hidden, # (lstm layers x batch size x lstm hidden size)
      'final_english_state':final_english_state} # (lstm layers x batch size x lstm hidden size)
```

Attention seq-to-seq model: decode()

```
def decode(self,
           english_hiddens:torch.Tensor=None,
           final_english_hidden:torch.Tensor=None,
           final_english_state:torch.Tensor=None,
           french_input_ids:torch.Tensor=None,
           do_teacher_forcing:bool=True,
           max_output_length:int=None, # How many tokens to generate
           temperature:float=None):

  # Then, for the rest of the desired output length, we generate one token at a time, conditioned on the previous generated token
  last_hidden, last_state = final_english_hidden, final_english_state # both (lstm layers x batch size x lstm hidden size)

  # Figure out what the first input to the decoder will be.
  if do_teacher_forcing:
    current_input_id = french_input_ids[:,0:1] # (batch size x 1)
    output_logits = []
    # current_input_ids = [current_input_id]
    max_output_length = french_input_ids.shape[1]
  else:
    current_input_id = torch.empty(final_english_hidden.shape[1], 1,
                                   dtype=torch.long,
                                   device=final_english_hidden.device).fill_(self.french_sos_id) #(batch size x 1)
    output_ids = [current_input_id]
```

Attention seq-to-seq model: decode()

```
for i in range(1, max_output_length):
  current_embeds = self.french_embeddings(current_input_id) # (batch size x 1 x embedding size)

  query = last_hidden[-1].unsqueeze(1) #.permute(2,0,1) # ()
  # print('query:', query.shape)
  # print('english hiddens',english_hiddens.shape)
  current_context, current_attn_weights = self.attention(query, english_hiddens)
  currrent_decoder_input = torch.cat((current_embeds, current_context), dim=2)

  current_output, (current_hidden, current_state) = self.decoder.forward(currrent_decoder_input, (last_hidden, last_state))
  current_logits = self.output_layer(current_output) #(batch size x 1 x vocab size)

  if do_teacher_forcing:
    output_logits.append(current_logits)
    current_input_id = french_input_ids[:, i:i+1] # (batch size x 1)
    # current_input_ids.append(current_input_id)
  else:
    current_output_id = self.sample_token_id_from_logits(current_logits, temperature=temperature) # (batch size x 1)
    output_ids.append(current_output_id)
    current_input_id = current_output_id

  last_hidden, last_state = current_hidden, current_state
```

Attention seq-to-seq model: decode()

```
if do_teacher_forcing:
  output_logits = torch.concatenate(output_logits,dim=1)
  losses = torch.nn.functional.cross_entropy(output_logits.transpose(1,2), french_input_ids[:,1:], reduction='none') #(batch size x max french

  # current_input_ids = torch.concatenate(current_input_ids,dim=1)

  # Then the final thing we need to do is zero out the losses whenever the target token is a padding token
  french_padding_mask = (french_input_ids != self.french_padding_id).int() #(batch size x max french sequence length)
  padded_losses = losses * french_padding_mask[:,1:] # (batch size x max french sequence length-1)
  loss = padded_losses.mean() #(1)
  output_dict.update({
      #  'current_input_ids':current_input_ids,
      # 'output_logits':output_logits,
      #               'losses':losses,
                    'loss':loss
                    })
else:
  output_ids = torch.concatenate(output_ids,dim=1)
  output_dict.update({
      # 'current_output_id':current_output_id,
                    'output_ids':output_ids})


return output_dict
```

Attention seq-to-seq model: forward()

```python
def forward(self,
            english_input_ids:torch.Tensor,
            french_input_ids:torch.Tensor):
    '''
    Once we have encode() and decode() defined, forward() is just a matter of
    calling both of them (with teacher forcing on for decoding)
    '''

    encoded = self.encode(english_input_ids = english_input_ids)
    decoded = self.decode(**encoded, french_input_ids=french_input_ids, do_teacher_forcing=True)

    return decoded
```

Attention seq-to-seq model: generate()

```python
def generate(self,
             english_input_ids:torch.Tensor,
             **kwargs):
    '''
    And then generation is a matter of calling encode(), and then decode()
    with teacher forcing turned off (and whatever other parameters need to be passed)
    '''

    encoded = self.encode(english_input_ids = english_input_ids)
    decoded = self.decode(**encoded, do_teacher_forcing=False, **kwargs)

    return decoded
```

Training the model

```python
attention_seqtoseq_model = AttentionSeqToSeqModel(
    english_word_vectors=english_vector_model.vectors,
    english_vocab_size = english_vector_model.vectors.shape[0],
    french_word_vectors=french_vector_model.vectors,
    french_vocab_size = french_vector_model.vectors.shape[0],
    learning_rate = 0.001, #I'll typically start with something like 1e-3 for LSTMs
    english_padding_id = english_vector_model.key_to_index['<pad>'],
    french_padding_id = french_vector_model.key_to_index['<pad>'],
    french_eos_id = french_vector_model.key_to_index['<eos>'],
    french_sos_id = french_vector_model.key_to_index['<sos>'],
    lstm_hidden_size=100,
    lstm_layers=2,
    dropout_prob=0.1)

from pytorch_lightning import Trainer
from pytorch_lightning.callbacks.progress import TQDMProgressBar

# Training our manual decoding model will be a little slower because the pytorch
# LSTM has optimizations for being run across the length of a batch
attention_trainer = Trainer(
    accelerator="auto",
    devices=1 if torch.cuda.is_available() else None,
    max_epochs=2,
    callbacks=[TQDMProgressBar(refresh_rate=20)],
    val_check_interval = 0.2,
    )
attention_trainer.fit(model=attention_seqtoseq_model,
                      train_dataloaders=train_dataloader,
                      val_dataloaders=val_dataloader)
```

```
Mean training loss (steps 300-400): 3.106
Epoch 0 step 500 validation loss: tensor(2.9043, device='cuda:0')
Mean training loss (steps 400-500): 2.981
Mean training loss (steps 500-600): 2.828
Mean training loss (steps 600-700): 2.828
Mean training loss (steps 700-800): 2.709
Mean training loss (steps 800-900): 2.620
Epoch 0 step 1000 validation loss: tensor(2.5956, device='cuda:0')
Mean training loss (steps 900-1000): 2.666
Mean training loss (steps 1000-1100): 2.679
Mean training loss (steps 1100-1200): 2.585
Mean training loss (steps 1200-1300): 2.515
Mean training loss (steps 1300-1400): 2.461
Epoch 0 step 1500 validation loss: tensor(2.3929, device='cuda:0')
```

```
Epoch 1 step 4000 validation loss: tensor(1.9521, device='cuda:0')
Mean training loss (steps 1400-1500): 1.899
Mean training loss (steps 1500-1600): 1.927
Mean training loss (steps 1600-1700): 1.908
Mean training loss (steps 1700-1800): 1.855
Mean training loss (steps 1800-1900): 1.890
Epoch 1 step 4500 validation loss: tensor(1.8937, device='cuda:0')
Mean training loss (steps 1900-2000): 1.833
Mean training loss (steps 2000-2100): 1.837
Mean training loss (steps 2100-2200): 1.908
Mean training loss (steps 2200-2300): 1.885
Mean training loss (steps 2300-2400): 1.901
Epoch 1 step 5000 validation loss: tensor(1.8491, device='cuda:0')
```

Generating text

```
english_sequence = text_to_id_vector("My name is Samuel.", language='english', vector_model=english_vector_model)
print(english_sequence)
print(id_vector_to_text(english_sequence, vector_model=english_vector_model))
with torch.no_grad():
  french_tokens = attention_seqtoseq_model.generate(english_input_ids=english_sequence.unsqueeze(0),
                                                    max_output_length = 25,
                                                    temperature=0.1)
print(french_tokens['output_ids'])
french_text = id_vector_to_text(french_tokens['output_ids'].squeeze(0), vector_model=french_vector_model)
print(french_text)
```

```
tensor([400002,    192,    311,     14,   4858,      2, 400003])
<sos> my name is samuel . <eos>
tensor([[155564,    151,   1085,     13,     12, 155562, 155565, 155562, 155565,
         155562, 155565, 155562, 155565, 155562, 155565, 155562, 155565, 155562,
         155565, 155562, 155565, 155562, 155565, 155562, 155565]])
<sos> ma mère est un <unk> <eos> <unk> <eos> <unk> <eos> <unk> <eos> <unk> <eos> <unk> <eos> <unk> <eos> <unk> <eos> <unk> <eos> <unk> <eos>
```

Evaluating machine translation

Difficult because there can be **multiple valid target translations** of the same input text

Booz endormi (Original French - 1859-83 Victor Hugo)	**Boaz Asleep** (Translation circa late 1800s, various publishers)	**Boaz Asleep** (Translation - 2001 EH and AM Blackmore)	**Boaz Asleep** (translation - 2002 Brooks Haxton)
Booz s'était couché de fatigue accablé; Il avait tout le jour travaillé dans son aire; Puis avait fait son lit à sa place ordinaire; Booz dormait auprès des boisseaux pleins de blé. Ce vieillard possédait des champs de blés et d'orge; Il était, quoique riche, à la justice enclin; Il n'avit pas de fange en l'eau de son moulin; Il n'avit pas d'enfer dans le feu de sa forge.	At work within his barn since very early, Fairly tired out with toiling all the day, Upon the small bed where he always lay Boaz was sleeping by his sacks of barley. Barley and wheat fields he possessed, and well, Though rich, loved justice; wherfore all the flood That turned his mill-wheels was unstained with mud, And in his smithy blazed no fire of hell.	There Boaz lay, overcome and worn out. All day he'd labored at his threshing floor; Now, bedded in his usual place once more, He slept, with grain bagged everywhere about. Boaz owned fields of barleycorn and wheat-- A rich old man, but righteous, even so. There was no foulness in his millstream's flow, There was no hellfire in his forge's heat.	Boaz, overcome with weariness, by torchlight made his pallet on the thresing floor where all day he had worked, and now he slept among the bushels of threshed wheat. The old man owned wheatfields and barley, and though he was rich, he was still fair-minded. No filth soured the sweetness of his well. No hot iron of torture whitened his forge.

http://www.gavroche.org/vhugo/vhpoetry/comparison.gav

BLEU score

Basic idea: Provide several reference target texts, and measure how well the model matched any/all of them

- Not idea, but works pretty well in practice

Based on **N-gram precision**: how many n-grams in the candidate translation occur also in one of the reference translations?

C1: It is a guide to action which ensures that the military always obeys the commands of the party.
C2: It is to insure the troops forever hearing the activity guidebook that party direct

R1: It is a guide to action that ensures that the military will forever heed Party commands.
R2: It is the guiding principle which guarantees the military forces always being under the command of the Party.
R3: It is the practical guide for the army always to heed the directions of the party.

https://courses.engr.illinois.edu/cs447/fa2020/Slides/Lecture13.pdf

BLEU details

For $n \in \{1,\ldots,4\}$, compute the (modified) **precision of all *n*-grams**:

$$Prec_n = \frac{\sum_{c \in C} \sum_{n\text{-gram} \in c} \text{MaxFreq}_{\text{ref}}(n\text{-gram})}{\sum_{c \in C} \sum_{\text{-gram} \in c} \text{Freq}_{\text{c}}(n\text{-gram})}$$

$\text{MaxFreq}_{\text{ref}}$(‘*the party*’) = max. count of ‘*the party*’ in **one** reference translation.

Freq_{c}(‘*the party*’) = count of ‘*the party*’ in candidate translation c.

Penalize short candidate translations by a **brevity penalty** BP

c = length (number of words) of the whole candidate translation corpus

r = Pick for each candidate the reference translation that is closest in length; sum up these lengths.

Brevity penalty $\text{BP} = \exp(1 - c/r)$ for $c \leq r$; $\text{BP} = 1$ for $c > r$

(BP ranges from e for $c=0$ to 1 for $c=r$)

https://courses.engr.illinois.edu/cs447/fa2020/Slides/Lecture13.pdf

Concluding thoughts

Sequence-to-sequence models

- Main application: translation

Attention

- **Improves performance of sequence-to-sequence models**
- Improves interpretability of classifiers