Sequence-to-Sequence Models and Basic Attention

CS 780/880 Natural Language Processing Lecture 19

Samuel Carton, University of New Hampshire

Last lecture

RNNs for language modeling in PyTorch

Generating text

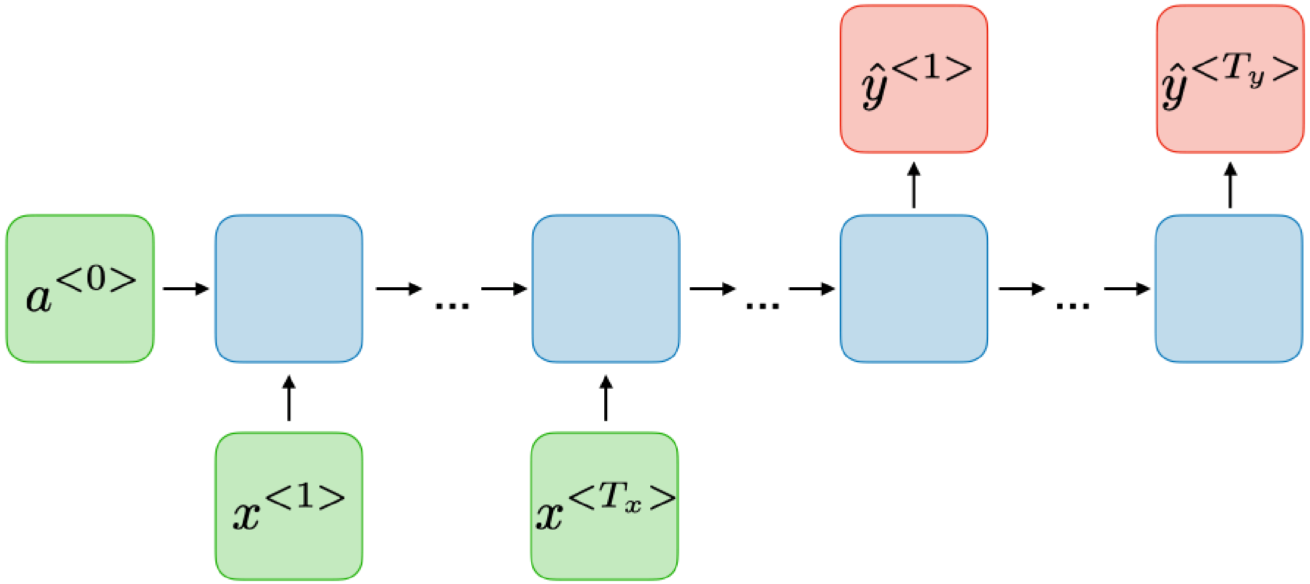
- **Greedy decoding**
- **Random sampling**
- Beam search decoding

Training RNNs

- **Teacher forcing**
    - Exposure bias
- Alternatives
    - Minimum risk, reinforcement learning, GANs

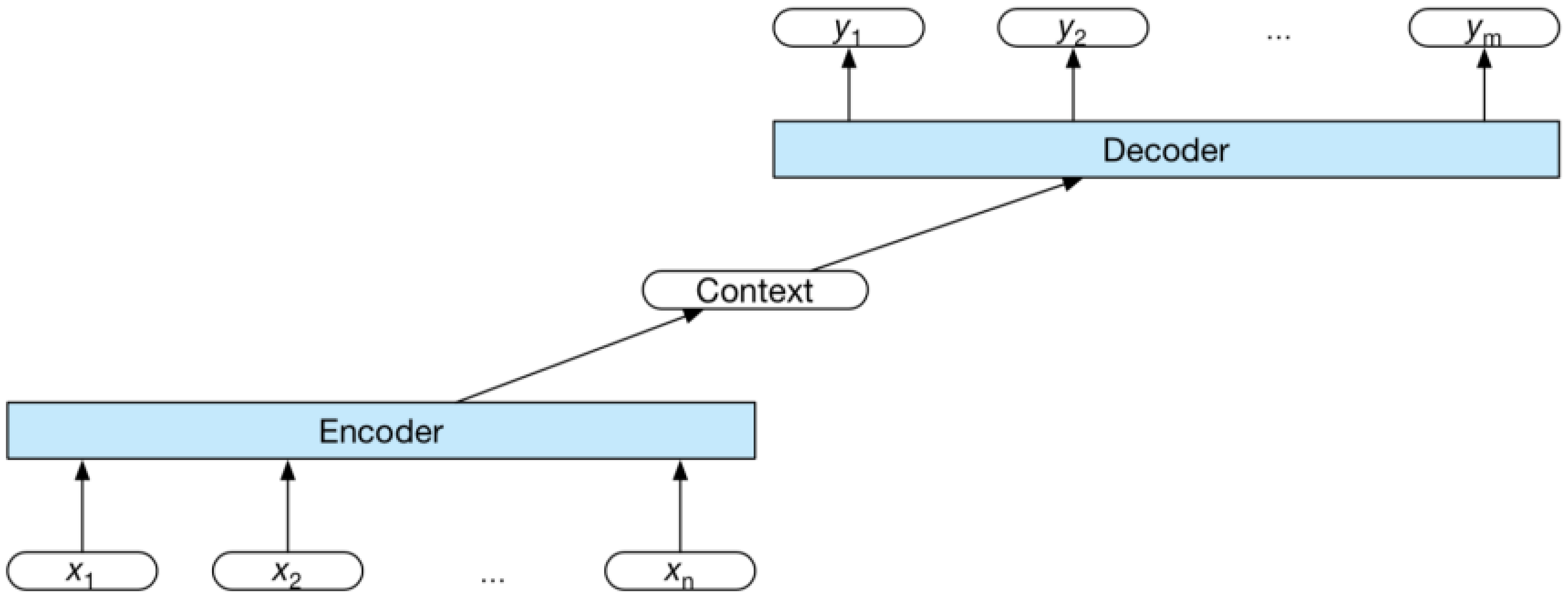
Sequence-to-sequence models

Basic idea: run an entire sequence through an RNN (the **encoder**), and then give the final vector it makes (the **context**) to another RNN (the **decoder**) to generate a new text sequence with



https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks

Sequence-to-sequence models



https://courses.engr.illinois.edu/cs447/fa2020/Slides/Lecture12.pdf

Machine translation

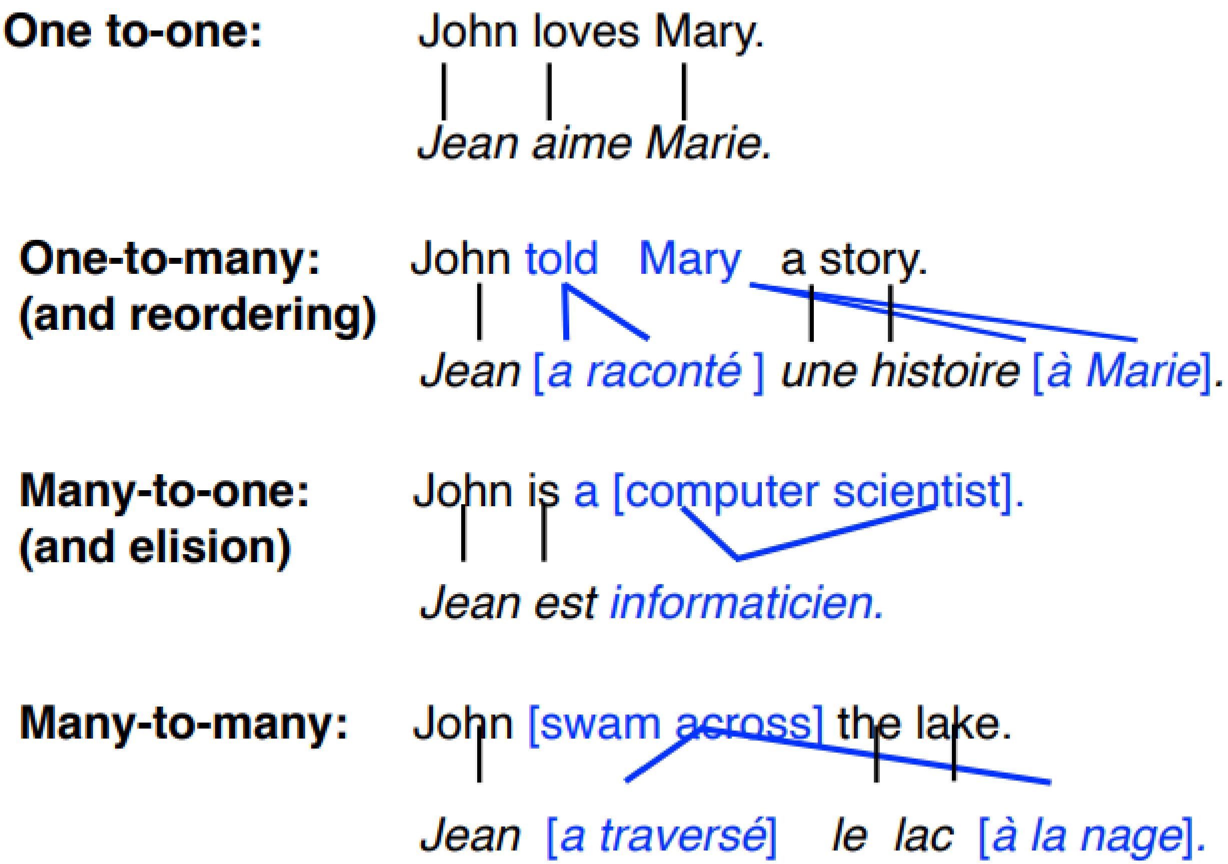
One to-one: John loves Mary.

Jean aime Marie.

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

Sequence tagging will work

Machine translation



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

Sequence tagging won’t work!

Language modeling batch loss

When we do teacher forcing for language modeling, we judge the model's output for each input token, against the next consecutive output token.

But that won't work directly for translation. So what do we do?

~~the~~	movie	was	very	good	.	<pad>	<pad>	
	↑	↑	↑	↑	↑			
	the	movie	was	very	good	.	<pad>	~~<pad>~~

Sequence-to-sequence batch loss

Instead, we pass in all the English (or whichever) tokens, and then do teacher forcing loss on the French (or whatever) tokens only.

So for:

my	name	is	samuel	<eos>

to

mon	nom	est	samuel	<eos>

The loss would look like:

				mon	nom	est	samuel	<eos>	
my	name	is	samuel	<eos>	mon	nom	est	samuel	<eos>

Preliminaries

Similar preliminaries:

1. Download translation dataset:
   - https://download.pytorch.org/tutorial/data.zip
2. Load 2 vector models-one for English, one for French
   - Add special tokens to both: <pad>, <unk>, <sos>, <eos>
3. Preprocess translation dataset and map to vector model tokens
4. Create dataset & dataloader
5. Install PyTorch Lightning

	english	french
0	Go.	Va !
1	Run!	Cours !
2	Run!	Courez !
3	Wow!	Ça alors !
4	Fire!	Au feu !
5	Help!	À l'aide !
6	Jump.	Saute.
7	Stop!	Ça suffit !
8	Stop!	Stop !
9	Stop!	Arrête-toi !

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]

LSTM seq-to-seq model: __init__()

```
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
```

LSTM seq-to-seq model: forward()

```python
def forward(self,
            english_input_ids:torch.Tensor, #(batch size x max sequence length),
            french_input_ids:torch.Tensor
            ):

    # First we need to pass the English tokens into the model to generate a final hidden vector and cell state
    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.lstm.forward(english_packed_embeddings)

    # We don't need to bother unpacking the English outputs since we're not using them
    # english_hiddens, _ = pad_packed_sequence(english_packed_output, batch_first=True, padding_value=0.0, total_length=english_input_ids.shape[1])

    # Then we start with that final hidden and final state, and do teacher-forcing using the french sequence vs. itself
    french_embeds = self.french_embeddings(french_input_ids) #(batch size x max french sequence length x embedding size)
    french_padding_mask = (french_input_ids != self.french_padding_id).int() #(batch size x max french sequence length)
    french_input_lengths = french_padding_mask.sum(dim=1).detach().cpu() #(batch size)
    french_packed_embeddings = pack_padded_sequence(french_embeds, french_input_lengths, batch_first=True, enforce_sorted=False)
    french_packed_output, (final_french_hidden, final_french_state) = self.lstm.forward(french_packed_embeddings, (final_english_hidden, final_english_state))
    french_hiddens, _ = pad_packed_sequence(french_packed_output, batch_first=True, padding_value=0.0, total_length=french_input_ids.shape[1])

    french_output_logits = self.output_layer(french_hiddens) #(batch size x max french sequence length x vocab size)

    losses = torch.nn.functional.cross_entropy(french_output_logits[:,:-1].transpose(1,2), french_input_ids[:,1:], reduction='none')

    # Then the final thing we need to do is zero out the losses whenever the target token is a padding token
    padded_losses = losses * french_padding_mask[:,1:] # (batch size x max french sequence length)
    loss = padded_losses.mean() #(1)

    return {'loss':loss}
```

LSTM seq-to-seq model: generate()

```python
def generate(self,
             english_input_ids:torch.Tensor, # shape (1,sequence length) or (sequence length)
             max_output_length:int, # How many tokens to generate past the input sequence
             temperature:float=0.5, # How loosely to sample from the output distribution
             ):

  # If the input shape is (1, sequence length), make it (sequence length)
  if english_input_ids.ndim == 2: english_input_ids = english_input_ids.squeeze(0)

  # Remove padding tokens if they are present
  english_padding_mask = (english_input_ids != self.english_padding_id).int() #(batch size x max sequence length)
  english_input_length = english_padding_mask.sum().detach().cpu() #(batch size)
  english_input_ids = english_input_ids[0:english_input_length]
  english_inputs_embeds = self.english_embeddings(english_input_ids) #(sequence length x embedding size)

  # First we run the given sequence through the LSTM
  # Because we aren't using a batch of variable-length sequences, we don't have to bother with a packed padded sequence like above
  english_hiddens, (final_english_hidden, final_english_state) = self.lstm.forward(english_inputs_embeds) # (sequence length x lstm hidden size),
   #( (lstm layers x lstm hidden size), (lstm layers x lstm hidden size) )

  output_tokens = []
  # 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
  last_logits = self.output_layer(last_hidden[-1])
  last_token_id = self.sample_token_id_from_logits(last_logits, temperature)
  output_tokens.append(last_token_id)
  for i in range(english_input_length, max_output_length):
    last_embeds = self.french_embeddings(last_token_id).unsqueeze(0)
    last_output, (last_hidden, last_state) = self.lstm.forward(last_embeds, (last_hidden, last_state))
    last_logits = self.output_layer(last_hidden[-1])
    last_token_id = self.sample_token_id_from_logits(last_logits, temperature)
    output_tokens.append(last_token_id)
    # Uncomment this if you want the model to stop on <eos>
    # if last_token_id == self.french_eos_id:
    #   break

  output_ids = torch.stack(output_tokens)

  return {'french_output_ids':output_ids}
```

Train model

```python
seqtoseq_model = EnglishFrenchSeqToSeqModel(
    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>'],
    lstm_hidden_size=100,
    lstm_layers=2,
    dropout_prob=0.1)

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

trainer = Trainer(
    accelerator="auto",
    devices=1 if torch.cuda.is_available() else None,
    max_epochs=5,
    callbacks=[TQDMProgressBar(refresh_rate=20)],
    val_check_interval = 0.1,
    )
trainer.fit(model=seqtoseq_model,
            train_dataloaders=train_dataloader,
            val_dataloaders=val_dataloader)
```

```
Mean training loss (steps -100-0): nan
Mean training loss (steps 0-100): 4.271
Mean training loss (steps 100-200): 3.373
Epoch 0 step 250 validation loss: tensor(3.2704, device='cuda:0')
Mean training loss (steps 200-300): 3.250
Mean training loss (steps 300-400): 3.109
Epoch 0 step 500 validation loss: tensor(2.9881, device='cuda:0')
Mean training loss (steps 400-500): 3.076
Mean training loss (steps 500-600): 2.968
Mean training loss (steps 600-700): 2.906
Epoch 0 step 750 validation loss: tensor(2.8339, device='cuda:0')
Mean training loss (steps 700-800): 2.901
Mean training loss (steps 800-900): 2.844
Epoch 0 step 1000 validation loss: tensor(2.7491, device='cuda:0')
```

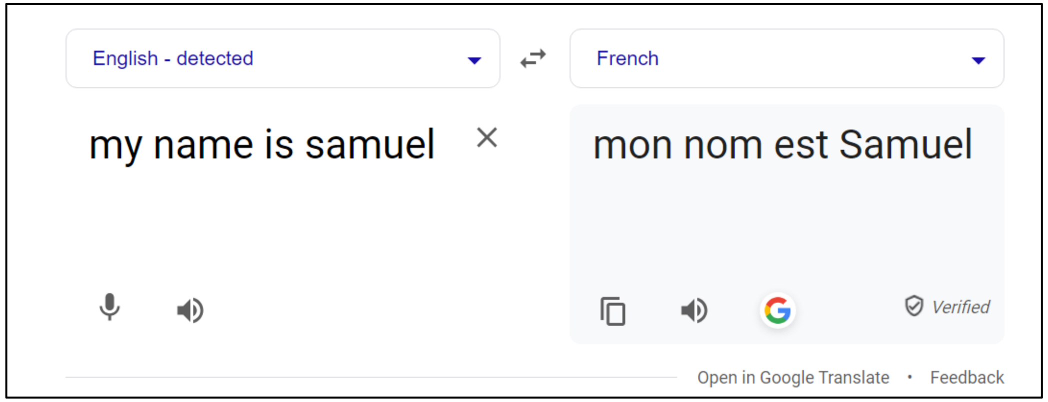
```
Epoch 4 step 11750 validation loss: tensor(1.7015, device='cuda:0')
Mean training loss (steps 1700-1800): 1.540
Mean training loss (steps 1800-1900): 1.558
Epoch 4 step 12000 validation loss: tensor(1.6946, device='cuda:0')
Mean training loss (steps 1900-2000): 1.612
Mean training loss (steps 2000-2100): 1.505
Mean training loss (steps 2100-2200): 1.572
Epoch 4 step 12250 validation loss: tensor(1.6841, device='cuda:0')
Mean training loss (steps 2200-2300): 1.564
Mean training loss (steps 2300-2400): 1.562
Epoch 4 step 12500 validation loss: tensor(1.6682, device='cuda:0')
INFO:pytorch_lightning.utilities.rank_zero:`Trainer.fit` stopped: `max_epochs=
```

Generate text

…and it didn't really work. RIP.

```
english_sequence = text_to_id_vector("<sos> My name is Samuel. <eos>", language='english', vector_model=english_vector_model)
print(english_sequence)
french_tokens = seqtoseq_model.generate(english_sequence,
                                        max_output_length = 20,
                                        temperature=0.5)
print(french_tokens)
french_text = id_vector_to_text(french_tokens['french_output_ids'], vector_model=french_vector_model)
print(french_text)
```

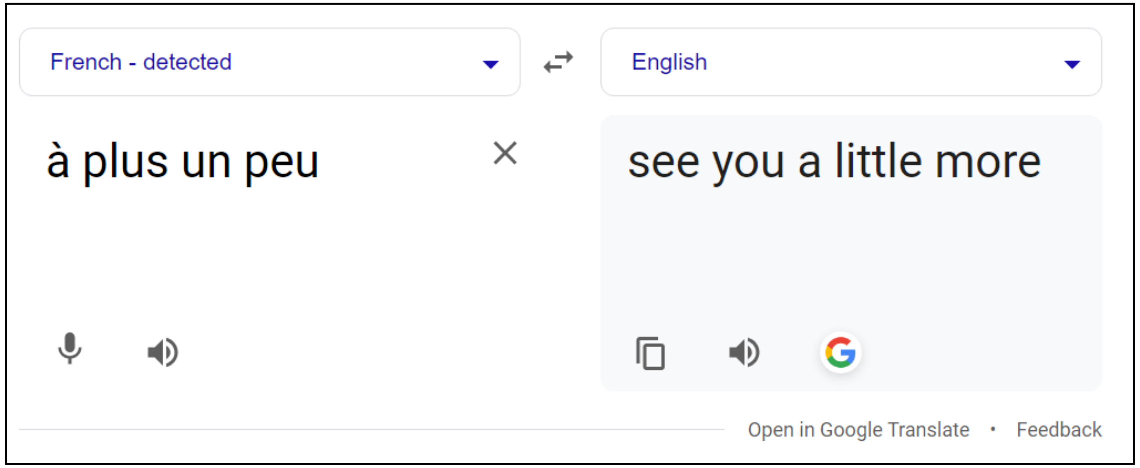
```
tensor([19795, 28069, 12257,   192,   311,    14,  4858,     2, 19795, 26828,
        12257])
{'french_output_ids': tensor([155562,      7,     90,    739, 155562, 155565, 155562, 155565, 155562,
        155565])}
<unk> à mon père <unk> <eos> <unk> <eos> <unk> <eos>
```



Generate text

```
english_sequence = text_to_id_vector("<sos> I am a person. <eos>", language='english', vector_model=english_vector_model)
print(english_sequence)
french_tokens = seqtoseq_model.generate(english_sequence,
                                        max_output_length = 20,
                                        temperature=0.5)
print(french_tokens)
french_text = id_vector_to_text(french_tokens['french_output_ids'], vector_model=french_vector_model)
print(french_text)
```

```
tensor([19795, 28069, 12257,    41,   913,     7,   899,     2, 19795, 26828,
        12257])
{'french_output_ids': tensor([     7,     28,     12,    308, 155562,     12, 155562, 155565, 155562,
        155565])}
à plus un peu <unk> un <unk> <eos> <unk> <eos>
```



Improving naïve seq2seq

Big problem here: we're expecting a **lot** out of that final encoder context vector.

- Essentially we're asking it to save up everything it needs to know to then go ahead and spit out the text we want.
- That's a lot of info to squeeze into a 100-element vector

Idea: What if we also let the decoder look at the original input while it is decoding the context?

- But it would need to be able to learn which parts of the original input were pertinent to what it was trying to do at any given point

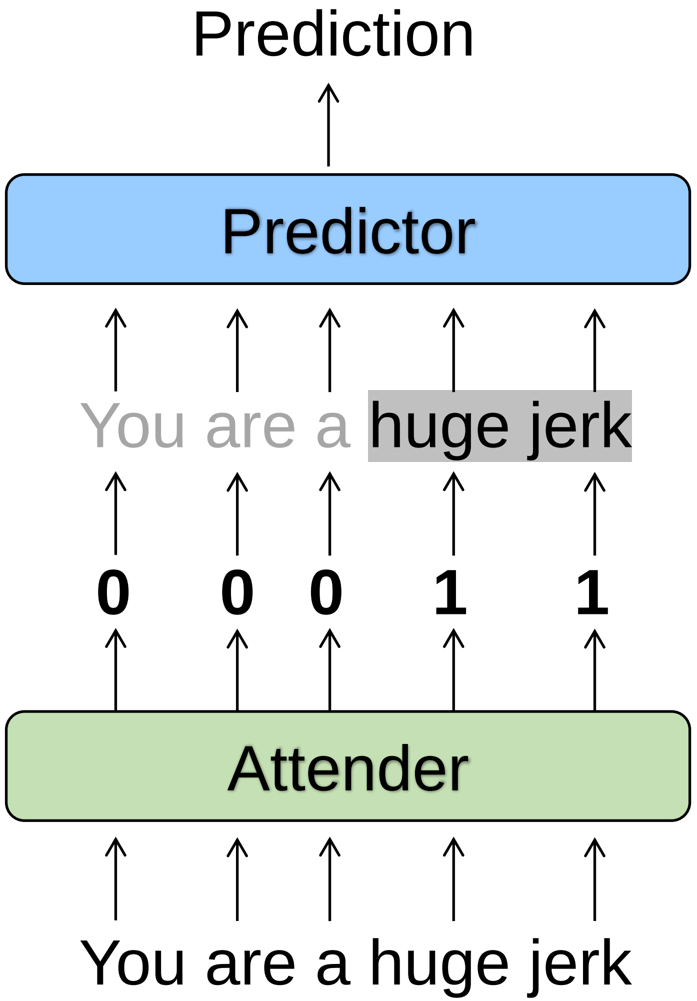
Solution: **Attention**

Classification with attention

Basic idea: Use one RNN (attender) to generate attention weights over a sequence, then a second RNN (predictor) to make predictions from the attention-weighted sequence

Dual training objective which encourages attention weights to be sparse, but predictor to be accurate.

In theory, leads to only important information (stuff needed for prediction) to be attended to.



Attention classification model

```python
class AttentionClassifier(pl.LightningModule):
  def __init__(self,
               word_vectors:np.ndarray,
               num_classes:int,
               learning_rate:float,
               padding_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,
               sparsity_loss_weight:float= 0.15,
               **kwargs):
    super().__init__( **kwargs)

    # We'll use the same PyTorch Embedding layer as before
    self.word_embeddings = torch.nn.Embedding.from_pretrained(embeddings=torch.tensor(word_vectors),
                                                              freeze=True)


    self.attender = torch.nn.LSTM(input_size = word_vectors.shape[1],
                                  hidden_size = lstm_hidden_size,
                                  num_layers=lstm_layers,
                                  bidirectional=True,
                                  dropout=dropout_prob,
                                  batch_first=True)
    self.attender_output_layer  = torch.nn.Linear(2*lstm_hidden_size, 1)

    self.predictor = torch.nn.LSTM(input_size = word_vectors.shape[1],
                                   hidden_size = lstm_hidden_size,
                                   num_layers=lstm_layers,
                                   bidirectional=True,
                                   dropout=dropout_prob,
                                   batch_first=True)
    self.predictor_output_layer  = torch.nn.Linear(2*lstm_hidden_size, num_classes)


    # Output layer input size has to be doubled because the LSTM is bidirectional
    self.lstm_layers = lstm_layers
    self.learning_rate = learning_rate
    self.padding_id = padding_id # we'll need this later
    self.sparsity_loss_weight = sparsity_loss_weight
    self.train_accuracy = Accuracy(task='multiclass', num_classes=num_classes)
    self.val_accuracy = Accuracy(task='multiclass', num_classes=num_classes)
```

Attention classification model

```python
def forward(self, y:torch.Tensor, input_ids:torch.Tensor, verbose=False):
  inputs_embeds = self.word_embeddings(input_ids) #(batch size x sequence length x embedding size)
  input_lengths = (input_ids != self.padding_id).sum(dim=1).detach().cpu()

  packed_embeddings = pack_padded_sequence(inputs_embeds, input_lengths, batch_first=True, enforce_sorted=False)
  packed_attender_output, _ = self.attender.forward(packed_embeddings)
  attender_output, _ = pad_packed_sequence(packed_attender_output, batch_first=True, padding_value=0.0, total_length=input_ids.shape[1])
  attention_logits = self.attender_output_layer(attender_output) #(batch size x sequence length x 1)
  attention_mask = torch.nn.functional.sigmoid(attention_logits)
  attention_masked_inputs_embeds = attention_mask * inputs_embeds
  attention_mask = attention_mask.squeeze(-1)
  sparsity_loss = masked_mean(attention_mask, (input_ids == self.padding_id)).mean()

  packed_masked_embeddings = pack_padded_sequence(attention_masked_inputs_embeds, input_lengths, batch_first=True, enforce_sorted=False)
  _, (final_predictor_hidden, final_predictor_state) = self.attender.forward(packed_masked_embeddings)
  last_layer_idx = self.lstm_layers-1
  last_layer_final_forward_hiddens = final_predictor_hidden[2*last_layer_idx]
  last_layer_final_reverse_hiddens = final_predictor_hidden[2*last_layer_idx+1]
  combined_last_layer_hiddens = torch.cat([last_layer_final_forward_hiddens, last_layer_final_reverse_hiddens], dim=1)
  py_logits = self.predictor_output_layer(combined_last_layer_hiddens)
  py = torch.argmax(py_logits, dim=1)
  py_loss = torch.nn.functional.cross_entropy(py_logits, y, reduction='mean')

  loss = py_loss + self.sparsity_loss_weight * sparsity_loss
  return {'py':py,
          'sparsity_loss':sparsity_loss,
          'py_loss':py_loss,
          'attention_mask':attention_mask,
          'loss':loss}
```

Trainer

```python
from pytorch_lightning import Trainer
from pytorch_lightning.callbacks.progress import TQDMProgressBar
from pytorch_lightning.callbacks import ModelCheckpoint

checkpoint_callback = ModelCheckpoint(dirpath=".", save_top_k=1, monitor="val_loss")

trainer = Trainer(
    accelerator="auto",
    devices=1 if torch.cuda.is_available() else None,
    max_epochs=3,
    callbacks=[TQDMProgressBar(refresh_rate=20), checkpoint_callback],
    val_check_interval = 0.5,
    default_root_dir='.' # This tells Pytorch Lightning to save checkpoints in the current working directory
    )
```

```
INFO:pytorch_lightning.utilities.rank_zero:GPU available: True (cuda), used: True
INFO:pytorch_lightning.utilities.rank_zero:TPU available: False, using: 0 TPU cores
INFO:pytorch_lightning.utilities.rank_zero:IPU available: False, using: 0 IPUs
INFO:pytorch_lightning.utilities.rank_zero:HPU available: False, using: 0 HPUs
```

Trainer

```
trainer.fit(model=model,
            train_dataloaders=train_dataloader,
            val_dataloaders=dev_dataloader)
```

```
/usr/local/lib/python3.9/dist-packages/pytorch_lightning/callbacks/model_checkpoint.py:613: UserWarning: Checkpoint directory /content exists and is not empty.
  rank_zero_warn(f"Checkpoint directory {dirpath} exists and is not empty.")
INFO:pytorch_lightning.accelerators.cuda:LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
INFO:pytorch_lightning.callbacks.model_summary:
  | Name                  | Type               | Params
-------------------------------------------------------------
0 | word_embeddings       | Embedding          | 40.0 M
1 | attender              | LSTM               | 403 K
2 | attender_output_layer | Linear             | 201
3 | predictor             | LSTM               | 403 K
4 | predictor_output_layer | Linear            | 402
5 | train_accuracy        | MulticlassAccuracy | 0
6 | val_accuracy          | MulticlassAccuracy | 0
-------------------------------------------------------------
807 K     Trainable params
40.0 M    Non-trainable params
40.8 M    Total params
163.229   Total estimated model params size (MB)
Validation accuracy: tensor(0.5234, device='cuda:0')
```

Epoch 2: 100% 1081/1081 [00:24<00:00, 43.77it/s, loss=0.256, v_num=6]

```
Validation accuracy: tensor(0.8016, device='cuda:0')
Validation accuracy: tensor(0.8062, device='cuda:0')
Training accuracy: tensor(0.8196, device='cuda:0')
Validation accuracy: tensor(0.8417, device='cuda:0')
Validation accuracy: tensor(0.8394, device='cuda:0')
Training accuracy: tensor(0.8719, device='cuda:0')
Validation accuracy: tensor(0.8326, device='cuda:0')
INFO:pytorch_lightning.utilities.rank_zero:`Trainer.fit` stopped: `max_epochs=3` reached.
Validation accuracy: tensor(0.8532, device='cuda:0')
Training accuracy: tensor(0.8981, device='cuda:0')
```

Visualizing model output

```
sentence = "It was a horrible movie, quite literally the most disgusting thing I have ever seen."
sentence_label = 0

tokens = tokenize(sentence)
word_ids = tokens_to_ids(tokens)

input_ids = torch.tensor([word_ids])
print(input_ids)

y = torch.tensor([sentence_label])
print(y)
```

```
tensor([[   20,    15,     7, 10230,  1005,     1,  1689,  5917,     0,    96,
         23967,   873,    41,    33,   661,   541,     2]])
tensor([0])
```

```
with torch.no_grad():
  model_output = model.forward(input_ids=input_ids, y=y)
pprint(model_output)
```

```
{'attention_mask': tensor([[0.2119, 0.9403, 0.3098, 0.9849, 0.2164, 0.2423, 0.8949, 0.9950, 0.1231,
         0.0810, 0.6699, 0.3581, 0.4085, 0.2120, 0.9551, 0.1834, 0.0361]]),
 'loss': tensor(0.0012),
 'py': tensor([0]),
 'py_loss': tensor(0.0012),
 'sparsity_loss': tensor(0.)}
```

Visualizing model output

```python
from IPython.core.display import HTML

for token, attention_weight in zip(tokens, model_output['attention_mask'][0]):
  # print(token, attention_weight)
  token_html = HTML(f'<span style="background-color: rgba(255,0,0, {attention_weight});">{token}</span>')
  display(token_html,)
```

it
was
a
horrible
movie
,
quite
literally
the
most
disgusting
thing
i
have
ever
seen
.

Saving and loading the model

```
# We can see the best checkpoint that Pytorch lightning saved for us
!ls
```

```
data    data.zip  'epoch=1-step=2105.ckpt'   lightning_logs   sample_data
```

```
# But we can also manually save the model in its current state
torch.save(model.state_dict(), 'manually_saved_model.ckpt')
```

```
!ls
```

```
data       'epoch=1-step=2105.ckpt'   manually_saved_model.ckpt
data.zip    lightning_logs             sample_data
```

Saving and loading the model

```python
loaded_model = AttentionClassifier(word_vectors=vector_model.vectors,
                                   num_classes = 2,
                                   learning_rate = 0.001, #I'll typically start with something like 1e-3 for LSTMs
                                   padding_id = vector_model.key_to_index['<pad>'],
                                   lstm_hidden_size=100,
                                   lstm_layers=2,
                                   dropout_prob=0.1,
                                   sparsity_loss_weight=0.15)
loaded_model.load_state_dict(torch.load('manually_saved_model.ckpt'))
display(loaded_model)
```

```
AttentionClassifier(
  (word_embeddings): Embedding(400002, 100)
  (attender): LSTM(100, 100, num_layers=2, batch_first=True, dropout=0.1, bidirectional=True)
  (attender_output_layer): Linear(in_features=200, out_features=1, bias=True)
  (predictor): LSTM(100, 100, num_layers=2, batch_first=True, dropout=0.1, bidirectional=True)
  (predictor_output_layer): Linear(in_features=200, out_features=2, bias=True)
  (train_accuracy): MulticlassAccuracy()
  (val_accuracy): MulticlassAccuracy()
)
```

Concluding thoughts

Sequence-to-sequence models

- Main application: translation

Attention

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

Model saving/loading