Sequence-to-Sequence Models and Attention

CS 780/880 Natural Language Processing Lecture 18

Samuel Carton, University of New Hampshire

Last lecture

RNNs for language modeling

Generating text

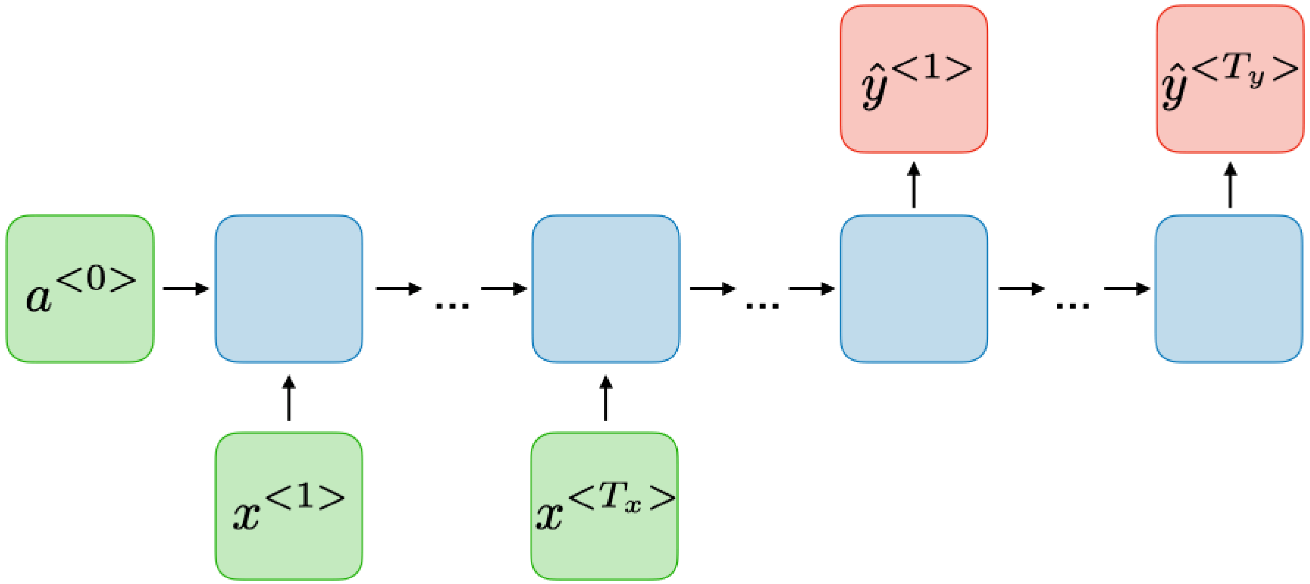
- Greedy decoding
- Random sampling
- Beam search decoding

Training RNNs

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

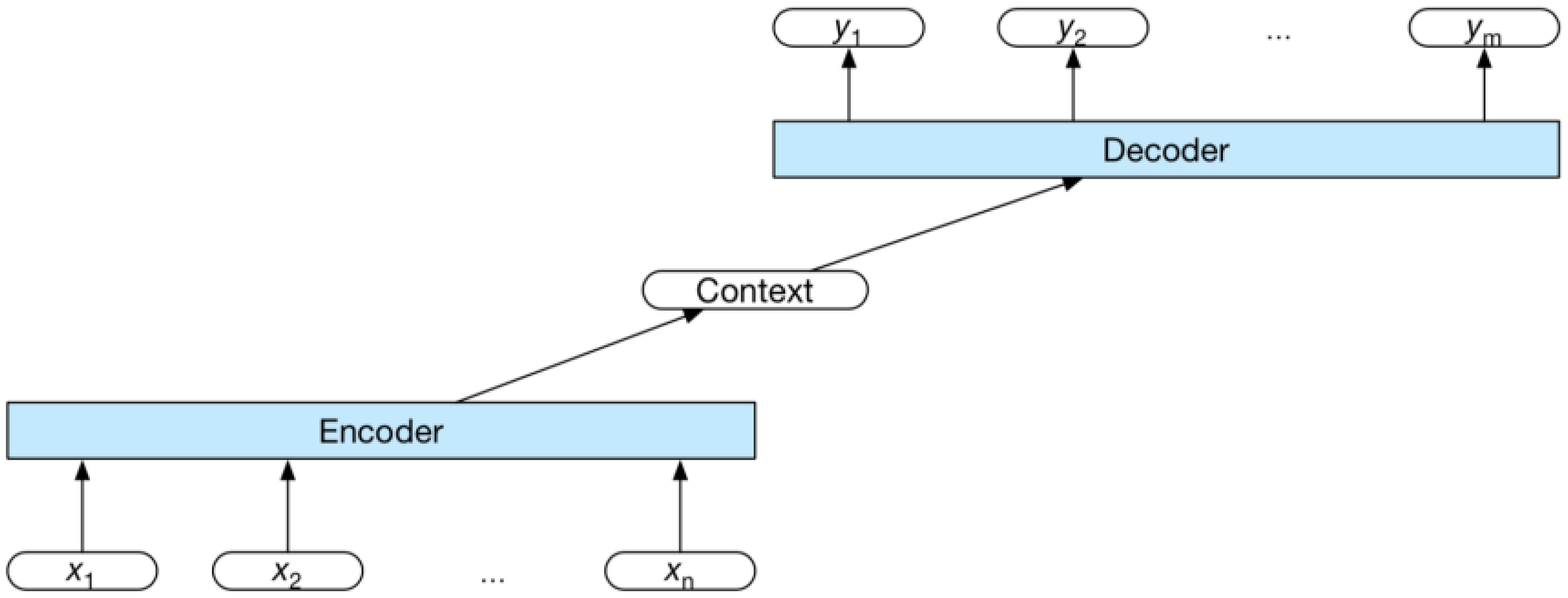
Sequence-to-sequence models

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

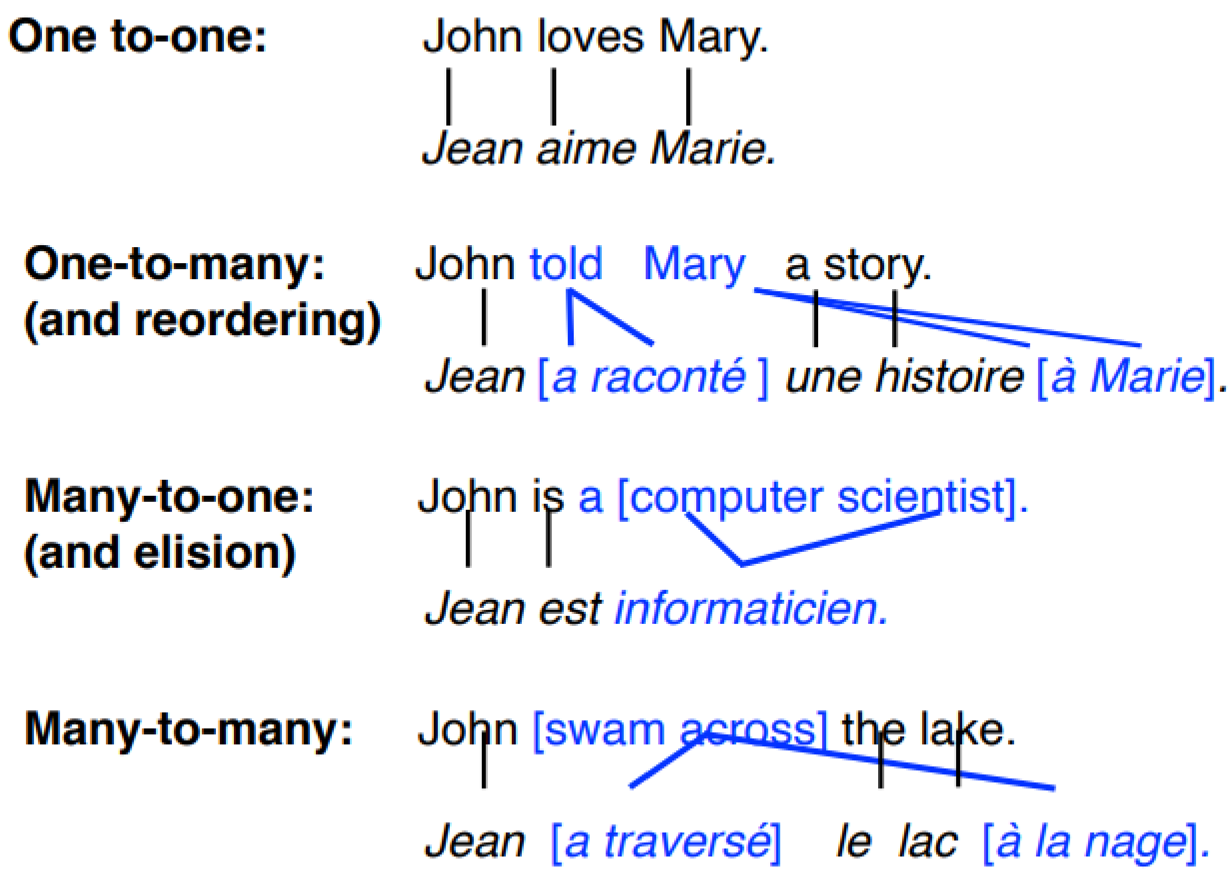
Machine translation

One to-one: John loves Mary.
Jean aime Marie.

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

Sequence tagging will work

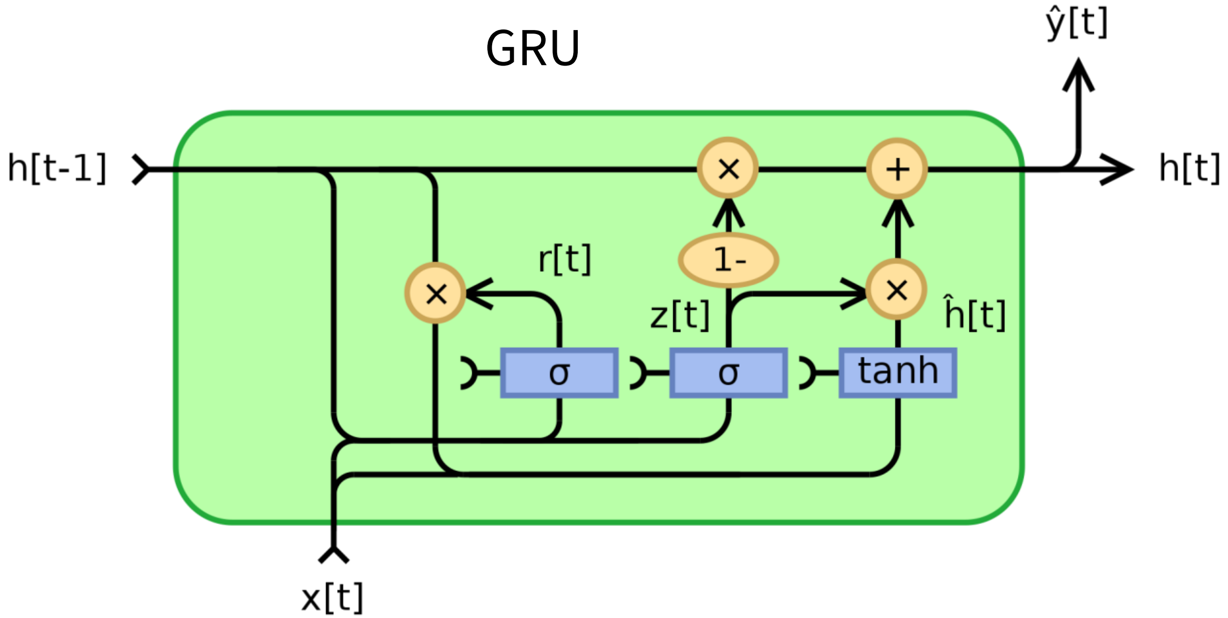
Machine translation



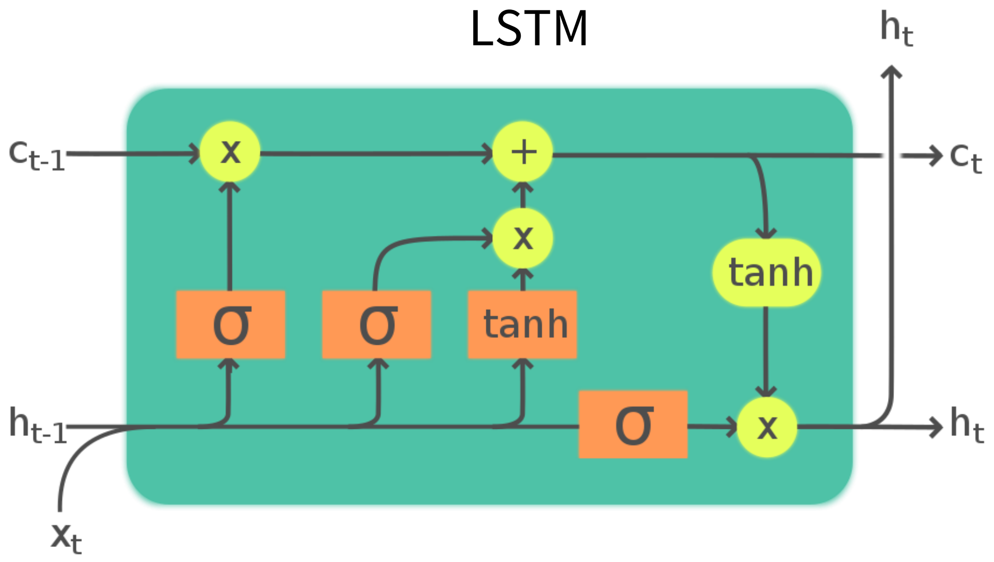
Sequence tagging won’t work!

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

Gated Recurrent Unit (GRU)



https://en.wikipedia.org/wiki/Gated_recurrent_unit



https://en.wikipedia.org/wiki/Long_short-term_memory

Downloading the translation dataset

```
data_url = 'https://download.pytorch.org/tutorial/data.zip'
```

```
!wget $data_url # this is a linux comand that will grab the file to the local directory
!unzip data.zip
```

```
--2023-04-04 16:17:57--  https://download.pytorch.org/tutorial/data.zip
Resolving download.pytorch.org (download.pytorch.org)... 52.222.139.109, 52.222.139.21, 52.222.139.90, ...
Connecting to download.pytorch.org (download.pytorch.org)|52.222.139.109|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2882130 (2.7M) [application/zip]
Saving to: ‘data.zip’

data.zip            100%[===================>]   2.75M  --.-KB/s    in 0.08s

2023-04-04 16:17:58 (34.2 MB/s) - ‘data.zip’ saved [2882130/2882130]

Archive:  data.zip
   creating: data/
  inflating: data/eng-fra.txt
```

Downloading the translation dataset

```
!ls # this shows us what files we've downloaded
```

```
data  data.zip  sample_data
```

```
!ls data #this shows us what is inside the file we just unzipped
```

```
eng-fra.txt  names
```

Seq2Seq tutorial

https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html

A little different from what I've been showing you

- GRU instead of LSTM
- Manual training loop
- Incorporates modeling decisions into training loop
    - I think this is terrible
- Very "from scratch"
    - No word embeddings
    - No NLTK

Text preprocessing

```python
SOS_token = 0
EOS_token = 1


class Lang:
    def __init__(self, name):
        self.name = name
        self.word2index = {}
        self.word2count = {}
        self.index2word = {0: "SOS", 1: "EOS"}
        self.n_words = 2  # Count SOS and EOS

    def addSentence(self, sentence):
        for word in sentence.split(' '):
            self.addWord(word)

    def addWord(self, word):
        if word not in self.word2index:
            self.word2index[word] = self.n_words
            self.word2count[word] = 1
            self.index2word[self.n_words] = word
            self.n_words += 1
        else:
            self.word2count[word] += 1
```

```python
# Turn a Unicode string to plain ASCII, thanks to
# https://stackoverflow.com/a/518232/2809427
def unicodeToAscii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
    )

# Lowercase, trim, and remove non-letter characters


def normalizeString(s):
    s = unicodeToAscii(s.lower().strip())
    s = re.sub(r"([.!?])", r" \1", s)
    s = re.sub(r"[^a-zA-Z.!?]+", r" ", s)
    return s
```

Text preprocessing

```python
def readLangs(lang1, lang2, reverse=False):
    print("Reading lines...")

    # Read the file and split into lines
    lines = open('data/%s-%s.txt' % (lang1, lang2), encoding='utf-8').\
        read().strip().split('\n')

    # Split every line into pairs and normalize
    pairs = [[normalizeString(s) for s in l.split('\t')] for l in lines]

    # Reverse pairs, make Lang instances
    if reverse:
        pairs = [list(reversed(p)) for p in pairs]
        input_lang = Lang(lang2)
        output_lang = Lang(lang1)
    else:
        input_lang = Lang(lang1)
        output_lang = Lang(lang2)

    return input_lang, output_lang, pairs
```

```python
MAX_LENGTH = 10

eng_prefixes = (
    "i am ", "i m ",
    "he is", "he s ",
    "she is", "she s ",
    "you are", "you re ",
    "we are", "we re ",
    "they are", "they re "
)


def filterPair(p):
    return len(p[0].split(' ')) < MAX_LENGTH and \
        len(p[1].split(' ')) < MAX_LENGTH and \
        p[1].startswith(eng_prefixes)


def filterPairs(pairs):
    return [pair for pair in pairs if filterPair(pair)]
```

Text preprocessing

```python
def prepareData(lang1, lang2, reverse=False):
    input_lang, output_lang, pairs = readLangs(lang1, lang2, reverse)
    print("Read %s sentence pairs" % len(pairs))
    pairs = filterPairs(pairs)
    print("Trimmed to %s sentence pairs" % len(pairs))
    print("Counting words...")
    for pair in pairs:
        input_lang.addSentence(pair[0])
        output_lang.addSentence(pair[1])
    print("Counted words:")
    print(input_lang.name, input_lang.n_words)
    print(output_lang.name, output_lang.n_words)
    return input_lang, output_lang, pairs


input_lang, output_lang, pairs = prepareData('eng', 'fra', True)
```

```
Reading lines...
Read 135842 sentence pairs
Trimmed to 10599 sentence pairs
Counting words...
Counted words:
fra 4345
eng 2803
```

```python
pprint(pairs[0:5])
```

```
[['j ai ans .', 'i m .'],
 ['je vais bien .', 'i m ok .'],
 ['ca va .', 'i m ok .'],
 ['je suis gras .', 'i m fat .'],
 ['je suis gros .', 'i m fat .']]
```

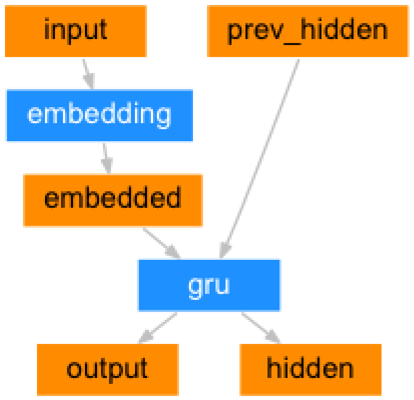
Model classes

```python
class EncoderRNN(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(EncoderRNN, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(input_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size)

    def forward(self, input, hidden):
        embedded = self.embedding(input).view(1, 1, -1)
        output = embedded
        output, hidden = self.gru(output, hidden)
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)
```

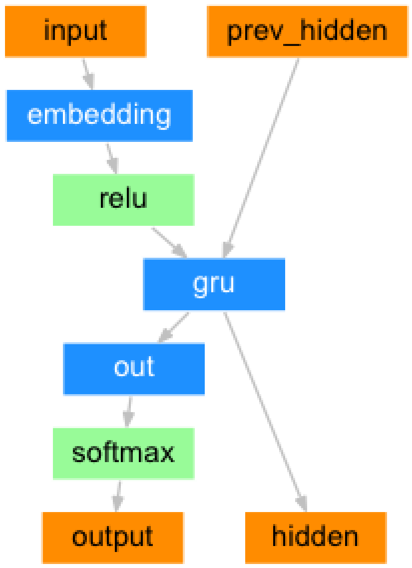
```python
class DecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size):
        super(DecoderRNN, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(output_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden, *args):
        output = self.embedding(input).view(1, 1, -1)
        output = F.relu(output)
        output, hidden = self.gru(output, hidden)
        output = self.softmax(self.out(output[0]))
        return output, hidden, None

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)
```

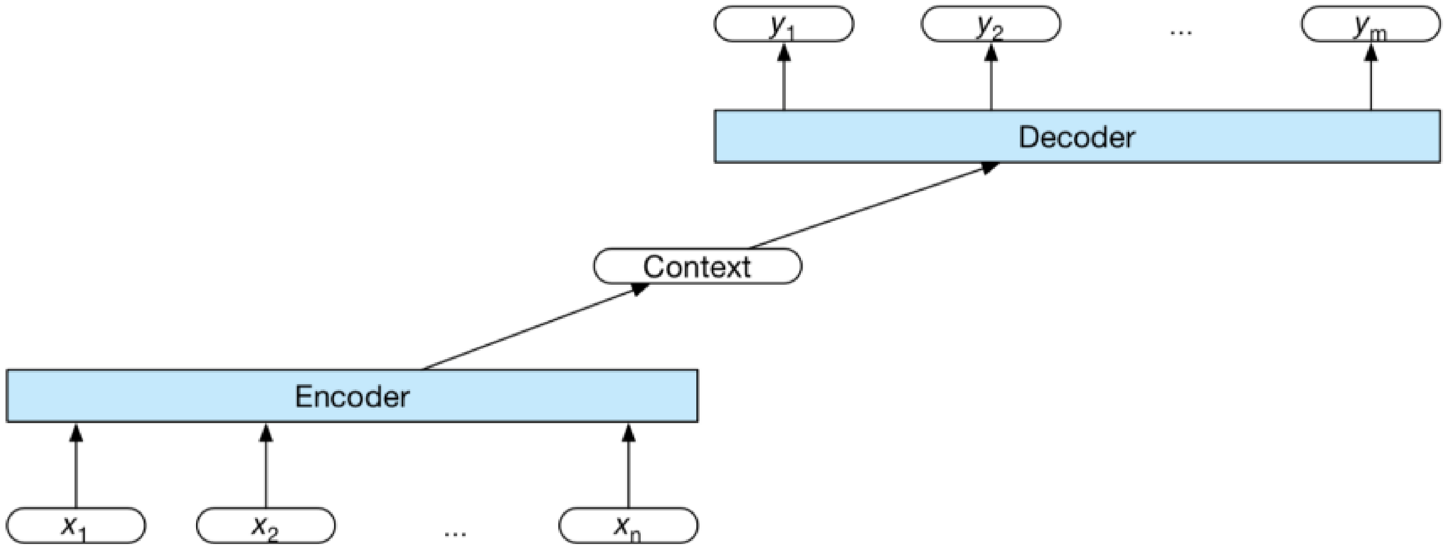
Sequence-to-sequence components

Encoder



Decoder





Training code

```
def indexesFromSentence(lang, sentence):
    return [lang.word2index[word] for word in sentence.split(' ')]


def tensorFromSentence(lang, sentence):
    indexes = indexesFromSentence(lang, sentence)
    indexes.append(EOS_token)
    return torch.tensor(indexes, dtype=torch.long, device=device).view(-1, 1)


def tensorsFromPair(pair):
    input_tensor = tensorFromSentence(input_lang, pair[0])
    target_tensor = tensorFromSentence(output_lang, pair[1])
    return (input_tensor, target_tensor)
```

Training code

```python
teacher_forcing_ratio = 0.5
def train(input_tensor, target_tensor, encoder, decoder,
          encoder_optimizer, decoder_optimizer, criterion, max_length=MAX_LENGTH):
    encoder_hidden = encoder.initHidden()

    encoder_optimizer.zero_grad()
    decoder_optimizer.zero_grad()

    input_length = input_tensor.size(0)
    target_length = target_tensor.size(0)
    encoder_outputs = torch.zeros(max_length, encoder.hidden_size, device=device)
    loss = 0
    for ei in range(input_length):
        encoder_output, encoder_hidden = encoder(
            input_tensor[ei], encoder_hidden)
        encoder_outputs[ei] = encoder_output[0, 0]

    decoder_input = torch.tensor([[SOS_token]], device=device)
    decoder_hidden = encoder_hidden
```

```python
    use_teacher_forcing = True if random.random() < teacher_forcing_ratio else False
    if use_teacher_forcing:
        # Teacher forcing: Feed the target as the next input
        for di in range(target_length):
            decoder_output, decoder_hidden, decoder_attention = decoder(
                decoder_input, decoder_hidden, encoder_outputs)
            loss += criterion(decoder_output, target_tensor[di])
            decoder_input = target_tensor[di]  # Teacher forcing

    else:
        # Without teacher forcing: use its own predictions as the next input
        for di in range(target_length):
            decoder_output, decoder_hidden, decoder_attention = decoder(
                decoder_input, decoder_hidden, encoder_outputs)
            topv, topi = decoder_output.topk(1)
            decoder_input = topi.squeeze().detach()  # detach from history as input

            loss += criterion(decoder_output, target_tensor[di])
            if decoder_input.item() == EOS_token:
                break

    loss.backward()
    encoder_optimizer.step()
    decoder_optimizer.step()
    return loss.item() / target_length
```

Training code

```python
teacher_forcing_ratio = 0.5
def train(input_tensor, target_tensor, encoder, decoder,
          encoder_optimizer, decoder_optimizer, criterion, max_length=MAX_LENGTH):
    encoder_hidden = encoder.initHidden()

    encoder_optimizer.zero_grad()
    decoder_optimizer.zero_grad()

    input_length = input_tensor.size(0)
    target_length = target_tensor.size(0)
    encoder_outputs = torch.zeros(max_length, encoder.hidden_size, device=device)
    loss = 0
    for ei in range(input_length):
        encoder_output, encoder_hidden = encoder(
            input_tensor[ei], encoder_hidden)
        encoder_outputs[ei] = encoder_output[0, 0]

    decoder_input = torch.tensor([[SOS_token]], device=device)
    decoder_hidden = encoder_hidden
```

```python
    use_teacher_forcing = True if random.random() < teacher_forcing_ratio else False
    if use_teacher_forcing:
        # Teacher forcing: Feed the target as the next input
        for di in range(target_length):
            decoder_output, decoder_hidden, decoder_attention = decoder(
                decoder_input, decoder_hidden, encoder_outputs)
            loss += criterion(decoder_output, target_tensor[di])
            decoder_input = target_tensor[di]  # Teacher forcing

    else:
        # Without teacher forcing: use its own predictions as the next input
        for di in range(target_length):
            decoder_output, decoder_hidden, decoder_attention = decoder(
                decoder_input, decoder_hidden, encoder_outputs)
            topv, topi = decoder_output.topk(1)
            decoder_input = topi.squeeze().detach()  # detach from history as input

            loss += criterion(decoder_output, target_tensor[di])
            if decoder_input.item() == EOS_token:
                break

    loss.backward()
    encoder_optimizer.step()
    decoder_optimizer.step()
    return loss.item() / target_length
```

Training code

```python
teacher_forcing_ratio = 0.5
def train(input_tensor, target_tensor, encoder, decoder,
          encoder_optimizer, decoder_optimizer, criterion, max_length=MAX_LENGTH):
    encoder_hidden = encoder.initHidden()

    encoder_optimizer.zero_grad()
    decoder_optimizer.zero_grad()

    input_length = input_tensor.size(0)
    target_length = target_tensor.size(0)
    encoder_outputs = torch.zeros(max_length, encoder.hidden_size, device=device)
    loss = 0
    for ei in range(input_length):
        encoder_output, encoder_hidden = encoder(
            input_tensor[ei], encoder_hidden)
        encoder_outputs[ei] = encoder_output[0, 0]

    decoder_input = torch.tensor([[SOS_token]], device=device)
    decoder_hidden = encoder_hidden

    use_teacher_forcing = True if random.random() < teacher_forcing_ratio else False
    if use_teacher_forcing:
        # Teacher forcing: Feed the target as the next input
        for di in range(target_length):
            decoder_output, decoder_hidden, decoder_attention = decoder(
                decoder_input, decoder_hidden, encoder_outputs)
            loss += criterion(decoder_output, target_tensor[di])
            decoder_input = target_tensor[di]  # Teacher forcing

    else:
        # Without teacher forcing: use its own predictions as the next input
        for di in range(target_length):
            decoder_output, decoder_hidden, decoder_attention = decoder(
                decoder_input, decoder_hidden, encoder_outputs)
            topv, topi = decoder_output.topk(1)
            decoder_input = topi.squeeze().detach()  # detach from history as input

            loss += criterion(decoder_output, target_tensor[di])
            if decoder_input.item() == EOS_token:
                break

    loss.backward()
    encoder_optimizer.step()
    decoder_optimizer.step()
    return loss.item() / target_length
```

Training code

```python
teacher_forcing_ratio = 0.5
def train(input_tensor, target_tensor, encoder, decoder,
          encoder_optimizer, decoder_optimizer, criterion, max_length=MAX_LENGTH):
    encoder_hidden = encoder.initHidden()

    encoder_optimizer.zero_grad()
    decoder_optimizer.zero_grad()

    input_length = input_tensor.size(0)
    target_length = target_tensor.size(0)
    encoder_outputs = torch.zeros(max_length, encoder.hidden_size, device=device)
    loss = 0
    for ei in range(input_length):
        encoder_output, encoder_hidden = encoder(
            input_tensor[ei], encoder_hidden)
        encoder_outputs[ei] = encoder_output[0, 0]

    decoder_input = torch.tensor([[SOS_token]], device=device)
    decoder_hidden = encoder_hidden
```

```python
    use_teacher_forcing = True if random.random() < teacher_forcing_ratio else False
    if use_teacher_forcing:
        # Teacher forcing: Feed the target as the next input
        for di in range(target_length):
            decoder_output, decoder_hidden, decoder_attention = decoder(
                decoder_input, decoder_hidden, encoder_outputs)
            loss += criterion(decoder_output, target_tensor[di])
            decoder_input = target_tensor[di]  # Teacher forcing

    else:
        # Without teacher forcing: use its own predictions as the next input
        for di in range(target_length):
            decoder_output, decoder_hidden, decoder_attention = decoder(
                decoder_input, decoder_hidden, encoder_outputs)
            topv, topi = decoder_output.topk(1)
            decoder_input = topi.squeeze().detach()  # detach from history as input

            loss += criterion(decoder_output, target_tensor[di])
            if decoder_input.item() == EOS_token:
                break

    loss.backward()
    encoder_optimizer.step()
    decoder_optimizer.step()
    return loss.item() / target_length
```

Training code

```python
def trainIters(encoder, decoder, n_iters, print_every=1000, plot_every=100, learning_rate=0.01):
    start = time.time()
    plot_losses = []
    print_loss_total = 0  # Reset every print_every
    plot_loss_total = 0  # Reset every plot_every

    encoder_optimizer = optim.SGD(encoder.parameters(), lr=learning_rate)
    decoder_optimizer = optim.SGD(decoder.parameters(), lr=learning_rate)
    training_pairs = [tensorsFromPair(random.choice(pairs))
                      for i in range(n_iters)]
    criterion = nn.NLLLoss()

    for iter in range(1, n_iters + 1):
        training_pair = training_pairs[iter - 1]
        input_tensor = training_pair[0]
        target_tensor = training_pair[1]

        loss = train(input_tensor, target_tensor, encoder,
                     decoder, encoder_optimizer, decoder_optimizer, criterion)
        print_loss_total += loss
        plot_loss_total += loss

        if iter % print_every == 0:
            print_loss_avg = print_loss_total / print_every
            print_loss_total = 0
            print('%s (%d %d%%) %.4f' % (timeSince(start, iter / n_iters),
                                         iter, iter / n_iters * 100, print_loss_avg))

        if iter % plot_every == 0:
            plot_loss_avg = plot_loss_total / plot_every
            plot_losses.append(plot_loss_avg)
            plot_loss_total = 0

    showPlot(plot_losses)
```

Training code

```
def trainIters(encoder, decoder, n_iters, print_every=1000, plot_every=100, learning_rate=0.01):
    start = time.time()
    plot_losses = []
    print_loss_total = 0  # Reset every print_every
    plot_loss_total = 0  # Reset every plot_every

    encoder_optimizer = optim.SGD(encoder.parameters(), lr=learning_rate)
    decoder_optimizer = optim.SGD(decoder.parameters(), lr=learning_rate)
    training_pairs = [tensorsFromPair(random.choice(pairs))
                      for i in range(n_iters)]
    criterion = nn.NLLLoss()

    for iter in range(1, n_iters + 1):
        training_pair = training_pairs[iter - 1]
        input_tensor = training_pair[0]
        target_tensor = training_pair[1]

        loss = train(input_tensor, target_tensor, encoder,
                     decoder, encoder_optimizer, decoder_optimizer, criterion)
        print_loss_total += loss
        plot_loss_total += loss

        if iter % print_every == 0:
            print_loss_avg = print_loss_total / print_every
            print_loss_total = 0
            print('%s (%d %d%%) %.4f' % (timeSince(start, iter / n_iters),
                                         iter, iter / n_iters * 100, print_loss_avg))

        if iter % plot_every == 0:
            plot_loss_avg = plot_loss_total / plot_every
            plot_losses.append(plot_loss_avg)
            plot_loss_total = 0

    showPlot(plot_losses)
```

Model training

```
# First we'll try training the version that doesn't use attention
hidden_size = 256
encoder1 = EncoderRNN(input_lang.n_words, hidden_size).to(device)
nonattention_decoder = DecoderRNN(hidden_size, output_lang.n_words).to(device)

trainIters(encoder1, nonattention_decoder, 25000, print_every=5000)
```

```
0m 56s (- 3m 44s) (5000 20%) 2.9316
1m 49s (- 2m 43s) (10000 40%) 2.3708
2m 42s (- 1m 48s) (15000 60%) 2.0766
3m 36s (- 0m 54s) (20000 80%) 1.8224
4m 30s (- 0m 0s) (25000 100%) 1.6010
```

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?

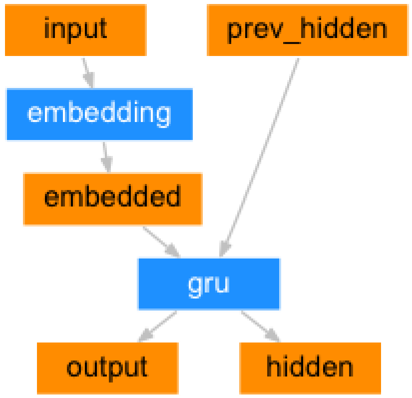
- But it would need to be able to learn which parts of the original input were pertinent to what it was trying to do at any given point

Solution: **Attention**

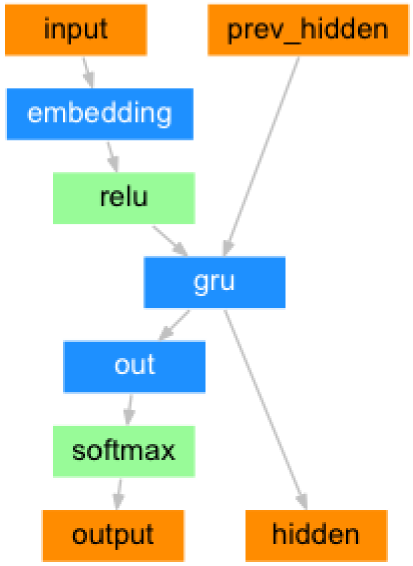
Attention in sequence-to-sequence

Basic idea: The decoder will have access to all the output from the encoder (not just the final output), but will learn a **weighting function** for how important any individual output is at a given timestep.

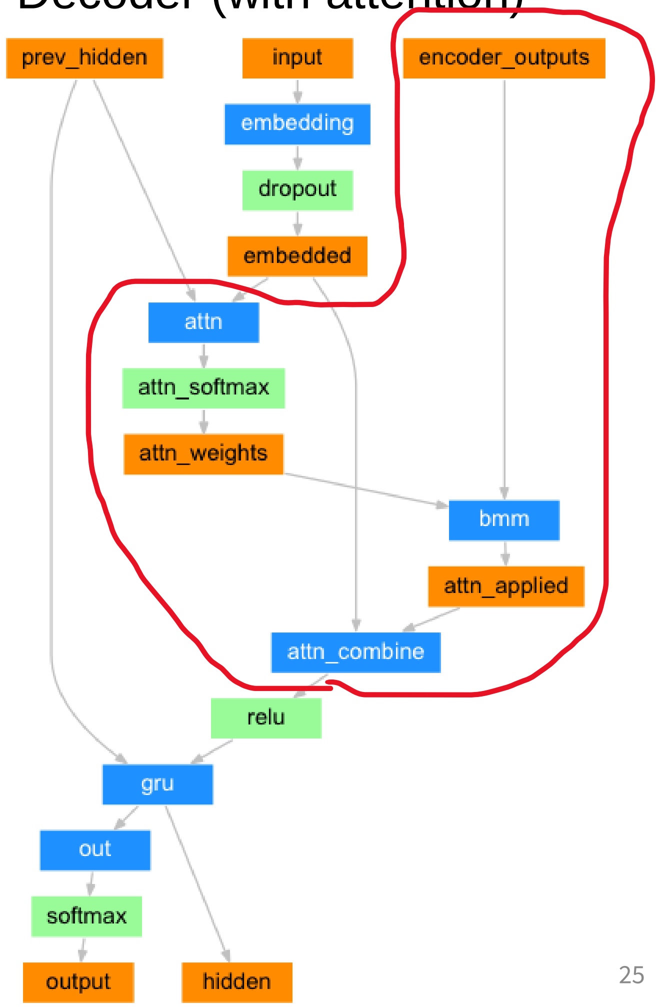
Encoder



Decoder (no attention)



Decoder (with attention)



Model classes

```python
class DecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size):
        super(DecoderRNN, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(output_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden, *args):
        output = self.embedding(input).view(1, 1, -1)
        output = F.relu(output)
        output, hidden = self.gru(output, hidden)
        output = self.softmax(self.out(output[0]))
        return output, hidden, None

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)
```

```python
class AttnDecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size, dropout_p=0.1, max_length=MAX_LENGTH):
        super(AttnDecoderRNN, self).__init__()
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.dropout_p = dropout_p
        self.max_length = max_length

        self.embedding = nn.Embedding(self.output_size, self.hidden_size)
        self.attn = nn.Linear(self.hidden_size * 2, self.max_length)
        self.attn_combine = nn.Linear(self.hidden_size * 2, self.hidden_size)
        self.dropout = nn.Dropout(self.dropout_p)
        self.gru = nn.GRU(self.hidden_size, self.hidden_size)
        self.out = nn.Linear(self.hidden_size, self.output_size)

    def forward(self, input, hidden, encoder_outputs):
        embedded = self.embedding(input).view(1, 1, -1)
        embedded = self.dropout(embedded)

        attn_weights = F.softmax(
            self.attn(torch.cat((embedded[0], hidden[0]), 1)), dim=1)
        attn_applied = torch.bmm(attn_weights.unsqueeze(0),
                                 encoder_outputs.unsqueeze(0))
        output = torch.cat((embedded[0], attn_applied[0]), 1)
        output = self.attn_combine(output).unsqueeze(0)
        output = F.relu(output)
        output, hidden = self.gru(output, hidden)
        output = F.log_softmax(self.out(output[0]), dim=1)
        return output, hidden, attn_weights

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)
```

Model training

Without attention

```
hidden_size = 256
encoder1 = EncoderRNN(input_lang.n_words, hidden_size).to(device)
nonattention_decoder = DecoderRNN(hidden_size, output_lang.n_words).to(device)

trainIters(encoder1, nonattention_decoder, 25000, print_every=5000)
```

```
0m 56s (- 3m 44s) (5000 20%) 2.9316
1m 49s (- 2m 43s) (10000 40%) 2.3708
2m 42s (- 1m 48s) (15000 60%) 2.0766
3m 36s (- 0m 54s) (20000 80%) 1.8224
4m 30s (- 0m 0s) (25000 100%) 1.6010
```

With attention

```
hidden_size = 256
encoder2 = EncoderRNN(input_lang.n_words, hidden_size).to(device)
attention_decoder = AttnDecoderRNN(hidden_size, output_lang.n_words,
                                   dropout_p=0.1).to(device)

trainIters(encoder2, attention_decoder, 25000, print_every=5000)
```

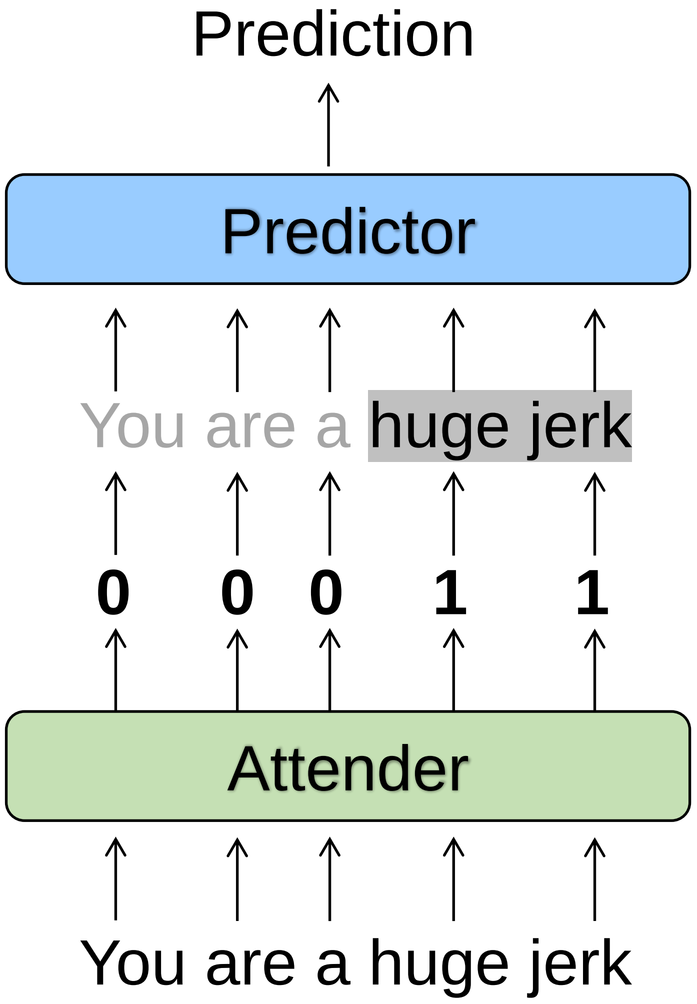
```
1m 23s (- 5m 32s) (5000 20%) 2.8722
2m 48s (- 4m 13s) (10000 40%) 2.2846
4m 15s (- 2m 50s) (15000 60%) 1.9739
5m 38s (- 1m 24s) (20000 80%) 1.7127
7m 1s (- 0m 0s) (25000 100%) 1.5260
```

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

```
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

```
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