Attention

Tarushii Goel

April 2020

1 Overview

Recurrent Neural Networks (RNNs) provide outstanding results on a variety of Natural Language Processing tasks; however, they continue to struggle in capturing long-term dependencies in complex sentences. Several advancements, such as LSTMs & GRUs were designed to tackle the problem of learning long-term dependencies, but perhaps one of the most influential is Attention. This lecture will begin by discussing the origin of Attention as a improvement to Seq2Seq for Neural Machine Translation (NMT), discuss and compare a variety of attention mechanisms, and conclude by exploring self-attention multi-head attention, which serve as the groundwork for recent advancements in Transformers and BERT.

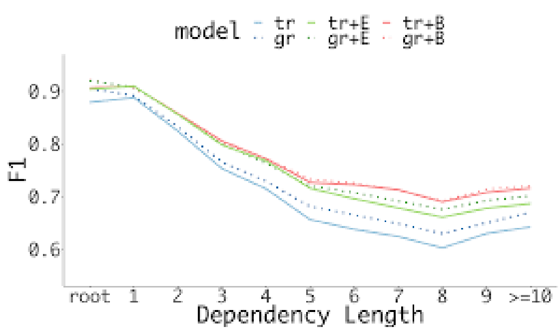

Figure 1: The Long-Term Dependency Problem (Graph of BERT variants, which are state-of-the-art)

2 Recap

To refresh you memory, I'll briefly review RNNs and Encoder-Decoders models. RNNs are recurrent; in each iteration they are fed the next input in a sequence and the previous hidden state, which carries accumulated information from other parts of the sentence, and produce a new hidden state.

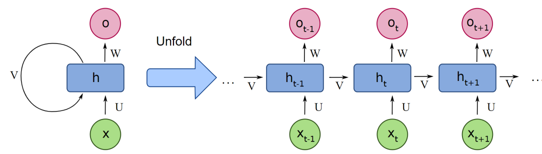

Figure 2: A Diagram of an RNN

Seq2Seq are a unique RNN architecture used for NMT. It is composed to two RNNs; an encoder, which 'encodes' the sentences into a 'context' vector (the last encoder hidden state, a latent representation of the entire sentence), and a decoder, which takes the 'context' vector and decodes it into a tranlated sentence.

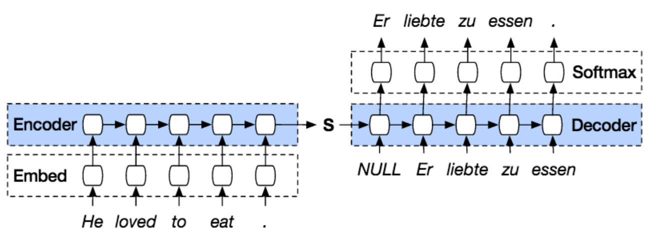

Figure 3: A Diagram of an RNN

3 What is Attention?

When we think about the English word "Attention", we know that it means directing your focus at something and taking greater notice. The Attention mechanism in Deep Learning is based off this concept of directing your focus, and it pays greater attention to certain factors when processing the data.

In broad terms, Attention is one component of a network's architecture, and is in charge of managing and quantifying the interdependence:

1. Between the input and output elements (General Attention)
2. Within the input elements (Self-Attention)

Say we have the sentence "How was your day", which we would like to translate to the French version - "Comment se passe ta journée". What the Attention component of the network will do for each word in the output sentence is map the important and relevant words from the input sentence and assign higher weights to these words, enhancing the accuracy of the output prediction.

Now, the question remains, how do we calculate attention weights?

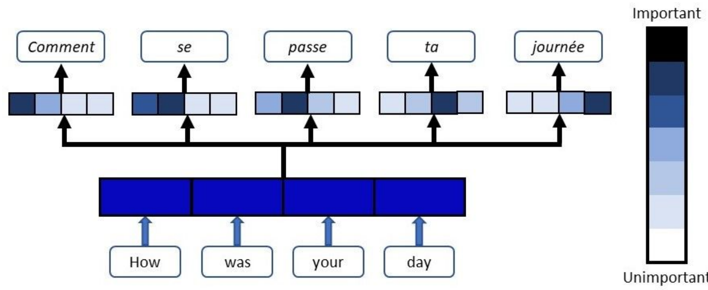

Figure 4: Example of Attention

4 Types of Attention Calculation

4.1 General Attention

General Attention is used to improve Seq2Seq models. Traditionally, the only information carried over from the encoder to the decoder is the last encoder hidden state, but with Attention, a weighted sum of the encoder hidden states of every time-step is appended to the input. Given a query q and a set of key-

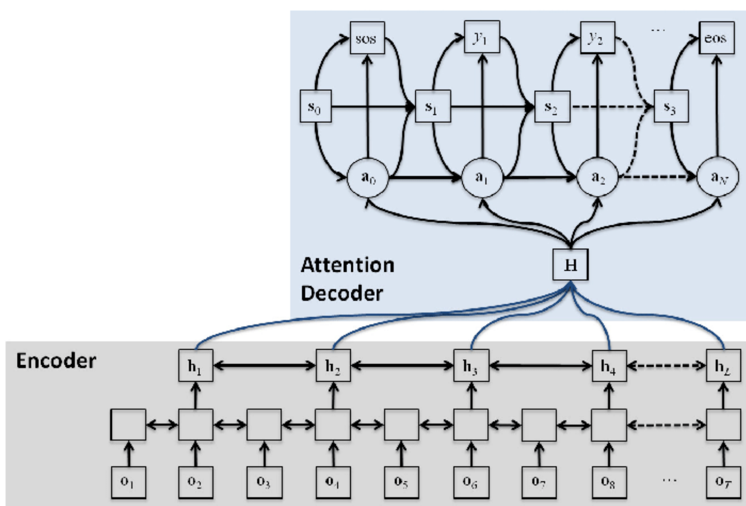

Figure 5: General Attention

value pairs (K, V), attention can be generalised to compute a weighted sum of the values dependent on the query and the corresponding keys. The query determines which values to focus on; we can say that the query 'attends' to the values. In the Seq2Seq, the query is the previous decoder hidden state $s_i - 1$ while the set of encoder hidden states h_0 to h_n represented both the keys and the values. The alignment model, is a row of weights applied to the decoder hidden state and the encoder hidden states. To calculate an 'alignment score' for any, calculate the dot-product $s_i^T * h_i$. Once the 'alignment scores are calculated for $h_0 - n$, apply a softmax to get the weights.

4.2 Self-Attention

With self-attention, each hidden state attends to the previous hidden states of the same RNN. Here s_t is the query while the decoder hidden states s_0 to $s_t - 1$ represent both the keys and the values.

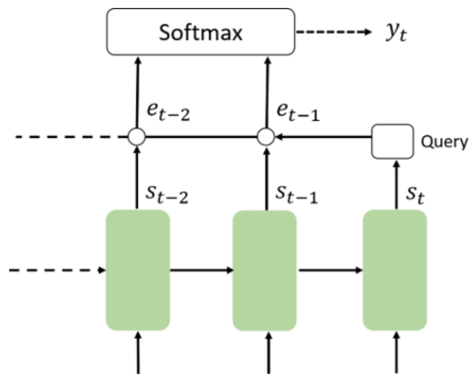

Figure 6: Self-Attention

4.3 Multi-Head Attention

When we have multiple queries q, we can stack them in a matrix Q. Each query can be thought of as a different 'head.' In the figure below, two attention heads are displayed (biege and green). The query word is "it". The first attention head is focusing more on the "animal" while the second in green is focusing on "tired". These multi-head mechanisms apparently help in translation tasks.

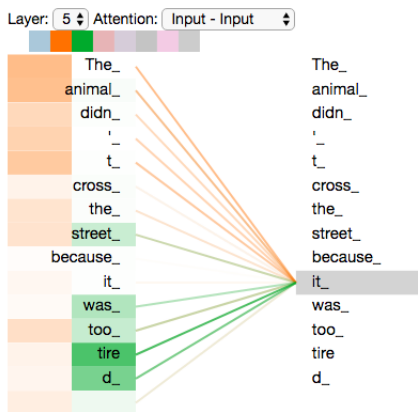

Figure 7: Multi-Head Attention

5 Code

I coded a standard Seq2Seq model with attention in PyTorch and put the code below. I highly recommend implementing the techniques I discussed in this lecture on your own and using this as a reference. Let me know if you have any questions!

```python
#data is from European Parliament proceedings, download here: http://www.statmt.org/europarl/
PATH = 'YOUR_PATH' #set to the path to your data
en_data = open(PATH + 'europarl-v7.fr-en.en', encoding='utf-8').read().split('\n')
fr_data = open(PATH + 'europarl-v7.fr-en.fr', encoding='utf-8').read().split('\n')

import spacy
import torch
import torchtext
from torch import nn
import torch.nn.functional as F

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

en_field = torchtext.data.Field(sequential=True, use_vocab=True, init_token='<sos>', eos_token='<eos>', tokenize='spacy', tokenizer_language='en')
fr_field = torchtext.data.Field(sequential=True, use_vocab=True, init_token = '<sos>', eos_token = '<eos>', tokenize='spacy', tokenizer_language='fr')

#Now, covert the data to a csv file to take advantage of torchtext's versatile TabularDataset class
import pandas as pd
from sklearn.model_selection import train_test_split

```

```
raw_data = {'English' : [line for line in en_data], 'French': [line for line in fr_data]}
df = pd.DataFrame(raw_data, columns=["English", "French"])
# create train and validation set
train, val = train_test_split(df, test_size=0.1)
train.to_csv(PATH + "train.csv", index=False)
val.to_csv(PATH + "val.csv", index=False)

train_set,val_set = torchtext.data.TabularDataset.splits(path=PATH, train='train.csv', validation='val.csv', format='csv', fields=[('English', en_field), ('French', fr_field)])

#loads 100-dimensional pre-trained glove word embeddings
en_field.build_vocab(train_set, val_set, vectors='glove.6B.100d')
fr_field.build_vocab(train_set, val_set, vectors='glove.6B.100d')

train_iter, val_iter = torchtext.data.BucketIterator.splits((train_set, val_set), batch_size=32, sort_key=lambda x: len(x.French), shuffle=True)

class Embedder(nn.Module):
    def __init__(self, embedding):
        super().__init__()
        self.embed = nn.Embedding(embedding.shape[0], embedding.shape[1])
        self.embed.weight.data.copy_(embedding)
        self.embed.weight.requires_grad = False
    def forward(self, input_sequence):
        return self.embed(input_sequence)

class Embedder(nn.Module):
    def __init__(self, embedding):
        super().__init__()
        self.embed = nn.Embedding(embedding.shape[0], embedding.shape[1])
        self.embed.weight.data.copy_(embedding)
        self.embed.weight.requires_grad = False
    def forward(self, input_sequence):
        return self.embed(input_sequence)

class Encoder(nn.Module):
    def __init__(self, hidden_size, embedding, num_layers=1, dropout=0.0):
        super(Encoder, self).__init__()
        self.hidden_size = hidden_size
        self.embedding = Embedder(embedding)
        self.gru = nn.GRU(embedding.shape[1], hidden_size, num_layers=num_layers, dropout=dropout, bidirectional=True, batch_first=True)
    def forward(self, input_sequence):
        embedded = self.embedding(input_sequence)
      #  x = nn.utils.rnn.pack_padded_sequence(x, lens) # unpad
        outputs, hidden_state = self.gru(embedded) # gru returns hidden state of all timesteps as well as hidden state at last timestep
        # pad the sequence to the max length in the batch
     #    output, _ = nn.utils.rnn.pad_packed_sequence(output)
```

```python
        # The ouput of a GRU has shape (seq_len, batch, hidden_size * num_directions)
        # Because the Encoder is bidirectional, combine the results from the
        # forward and reversed sequence by simply adding them together.
        outputs = outputs[:, :, :self.hidden_size] + outputs[:, :, self.hidden_size:]
        return outputs, hidden_state

class Decoder(nn.Module):
    def __init__(self, batch_size, hidden_size, embedding, num_layers=1, drop_prob=0.1):
        super(Decoder, self).__init__()
        self.batch_size = batch_size
        self.embedding_size = embedding.shape[1]
        self.embedding = Embedder(embedding)
        self.attn = nn.Linear(hidden_size * 2, 1)
        self.gru = nn.GRU(embedding.shape[1]+hidden_size, hidden_size, num_layers=num_layers, dropout=drop_prob, batch_first=True)
        self.classifier = nn.Linear(hidden_size, embedding.shape[0])

    def forward(self, decoder_hidden, encoder_outputs, inputs):
        # Embed input words
        embedded = self.embedding(inputs)
        #Assumed size of decoder_hidden -> (num_layers, batch_size, embedding_size) size of encoder_outputs -> (batch_size, sentence_len, embedding_size)
        #need to convert length of decoder_hidden to (batch_size, sentence_len, embedding_size)
        self.sequence_len = encoder_outputs.shape[1]
        decoder_hidden = torch.sum(decoder_hidden, axis=0)
        attn_inp = decoder_hidden.unsqueeze(1).repeat(1, self.sequence_len,1)
        weights = self.attn(torch.cat((attn_inp, encoder_outputs), dim = 2)).squeeze()
        normalized_weights = F.softmax(weights)
        attn_applied = torch.bmm(normalized_weights.unsqueeze(1), encoder_outputs)
        cat_input = torch.cat((embedded, attn_applied), axis=2)
        output, hidden_state = self.gru(cat_input, decoder_hidden.unsqueeze(0))
        output = self.classifier(output).squeeze()
        return output, hidden_state

def train(encoder, decoder, encoder_opt, decoder_opt, criterion, input, target):
    #set both to train moode
    encoder.train()
    decoder.train()
    #pass through encoder
    enc_output, enc_hidden = encoder(target)
    #initialize input to '<sos>' tokends anddecoder hidden state to final encoder hidden state
    dec_input, dec_hidden = target[:, 0].unsqueeze(1), enc_hidden
```

```
    loss = 0
    for i in range(1, target.shape[1]):
        dec_input, dec_hidden = decoder(dec_hidden, enc_output, dec_input)
        loss += criterion(dec_input, target[:, i])
        topv, topi = dec_input.topk(1)
        dec_input = topi.detach()  # detach from history as input
    loss.backward()
    encoder_opt.step()
    decoder_opt.step()
    return loss

hidden_size = 200
batch_size = 2
epochs = 1
encoder = Encoder(hidden_size, en_field.vocab.vectors).to(device)
decoder = Decoder(batch_size, hidden_size, fr_field.vocab.vectors).to(device)
encoder_opt = torch.optim.Adam([param for param in encoder.parameters() if param.requires_grad == True], lr=1.0e-4)
decoder_opt = torch.optim.Adam([param for param in decoder.parameters() if param.requires_grad == True], lr=1.0e-4)
criterion = nn.CrossEntropyLoss(ignore_index=1)

for i in range(epochs):
    losses = []
    print('Epoch %x:' % i, end='')
    c = 0
    for batch in train_iter:
        input = batch.English.t().to(device)
        target = batch.French.t().to(device)
        if (torch.max(input) > 116730):
            continue
        losses.append(train(encoder, decoder, encoder_opt, decoder_opt, criterion, input, target).item())
        if (c % 10 == 0):
            print('.', end='')
        c+=1
    print(' loss: ', sum(losses)/len(losses))
```

6 References

I got a lot of the explanation and graphics for Attention from this blog post: https://blog.floydhub.com/attention-mechanism/.