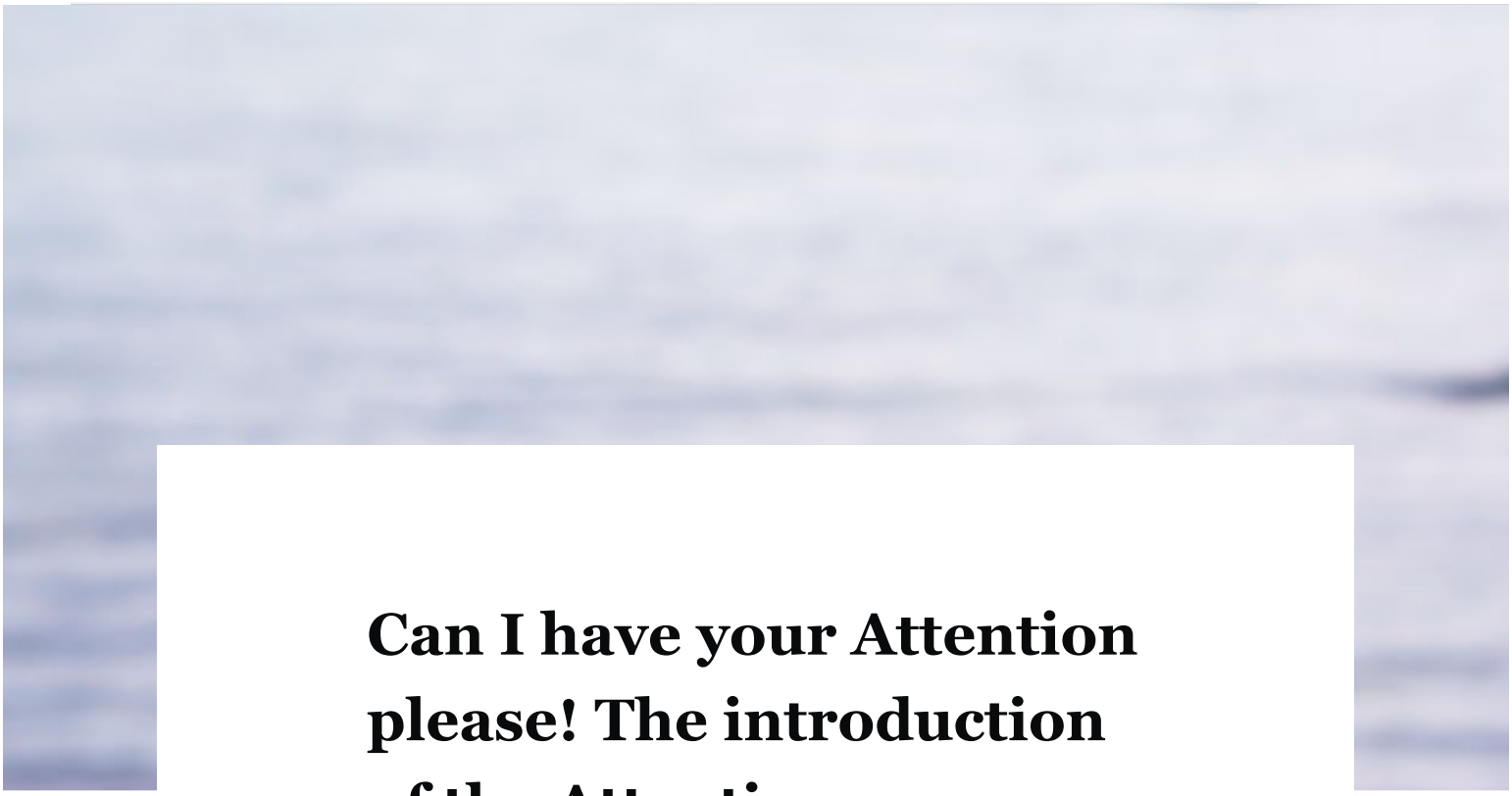
15 SEPTEMBER 2019 / DEEP LEARNING

Attention Mechanism



Can I have your Attention please! The introduction of the Attention Mechanism in deep learning has improved

the success of various models in recent years, and continues to be an omnipresent component in state-of-the-art models. Therefore, it is vital that we pay Attention to Attention and how it goes about achieving its effectiveness.

In this article, I will be covering the main concepts behind Attention, including an implementation of a sequence-to-sequence Attention model, followed by the application of Attention in Transformers and how they can be used for state-of-the-art results. It is advised that you have some knowledge of Recurrent Neural Networks (RNNs) and their variants, or an understanding of how sequence-to-sequence models work.

What is Attention?

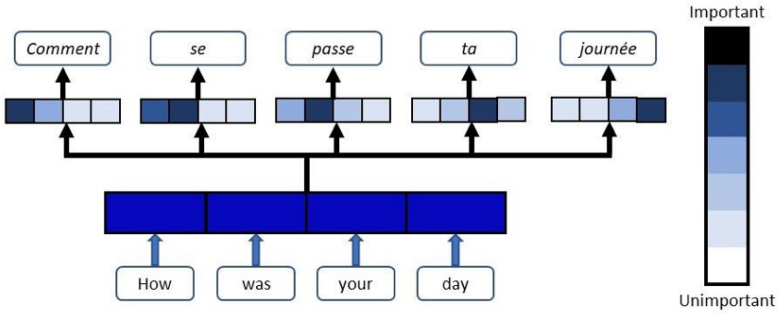
When we think about the English word “Attention”, we know that it means directing your focus at something and taking greater notice. The Attention mechanism in Deep Learning is based off this concept of directing your focus, and it pays greater attention to certain factors when processing the data.

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)

Let me give you an example of how Attention works in a translation task.

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.



Weights are assigned to input words at each step of the translation

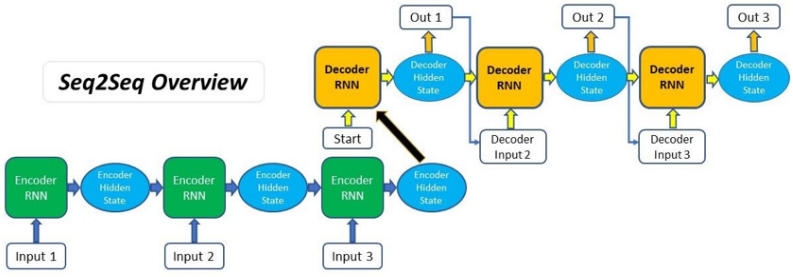
The above explanation of Attention is very broad and vague due to the various types of Attention mechanisms available. But fret not, you'll gain a clearer picture of how

you'll gain a clearer picture of how Attention works and achieves its objectives further in the article. As the Attention mechanism has undergone multiple adaptations over the years to suit various tasks, there are many different versions of Attention that are applied. We will only cover the more popular adaptations here, which are its usage in sequence-to-sequence models and the more recent Self-Attention.

While Attention does have its application in other fields of deep learning such as Computer Vision, its main breakthrough and success comes from its application in Natural Language Processing (NLP) tasks. This is due to the fact that Attention was introduced to address the problem of long sequences in Machine Translation, which is also a problem for most other NLP tasks as well.

Attention in Sequence-to-Sequence Models

Most articles on the Attention Mechanism will use the example of sequence-to-sequence (seq2seq) models to explain how it works. This is because Attention was originally introduced as a solution to address the main issue surrounding seq2seq models, and to great success. If you are unfamiliar with seq2seq models, also known as the Encoder-Decoder model, I recommend having a read through this article to get you up to speed.

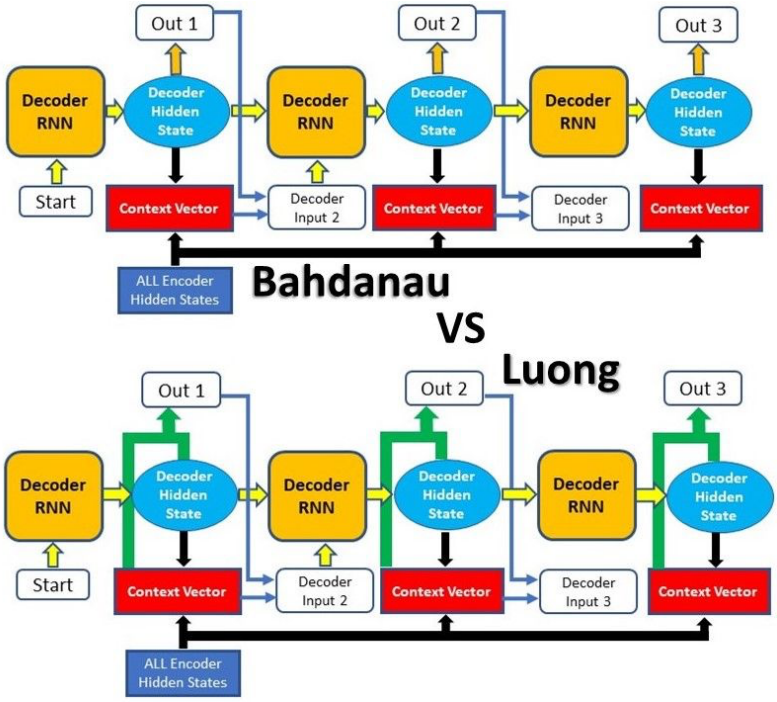


Overall process of a Sequence-to-sequence model

The standard seq2seq model is generally unable to accurately process long input sequences, since only the last hidden state of the encoder RNN is used as the context vector for the decoder. On the other hand, the Attention Mechanism directly addresses this issue as it retains and utilises all the hidden states of the input sequence during the decoding process. It does this by creating a unique mapping between each time step of the decoder output to all the encoder hidden states. This means that for each output that the decoder makes, it has access to the entire input sequence and can selectively pick out specific elements from that sequence to produce the output.

Therefore, the mechanism allows the model to focus and place more "Attention" on the relevant parts of the input sequence as needed.

Types of Attention



Comparing Bahdanau Attention with Luong Attention

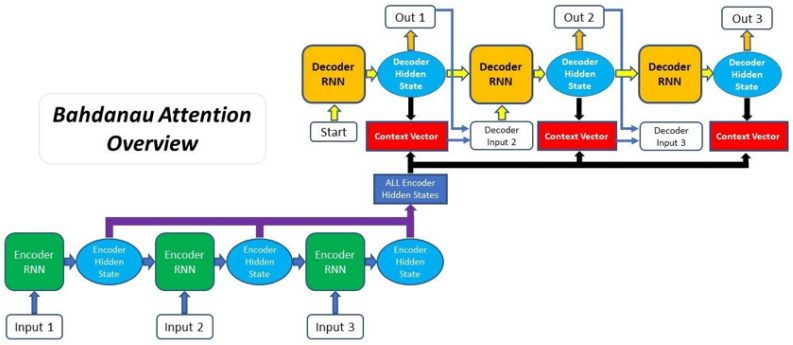
Before we delve into the specific mechanics behind Attention, we must note that there are 2 different major types of Attention:

- Bahdanau Attention
- Luong Attention

While the underlying principles of Attention are the same in these 2 types, their differences lie mainly in

their architectures and computations.

Bahdanau Attention



Overall process for Bahdanau Attention seq2seq model

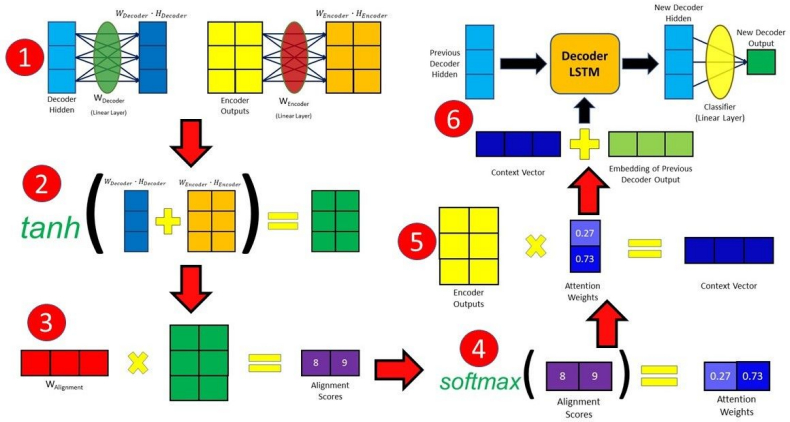
The first type of Attention, commonly referred to as Additive Attention, came from a paper by Dzmitry Bahdanau, which explains the less-descriptive original name. The paper aimed to improve the sequence-to-sequence model in machine translation by aligning the decoder with the relevant input sentences and implementing Attention. The entire step-by-step process of applying

Attention in Bahdanau's paper is as

follows:

1. Producing the Encoder Hidden States - Encoder produces hidden states of **each** element in the input sequence
2. Calculating Alignment Scores between the previous decoder hidden state and each of the encoder's hidden states are calculated *(Note: The last encoder hidden state can be used as the first hidden state in the decoder)*
3. Softmaxing the Alignment Scores - the alignment scores for each encoder hidden state are combined and represented in a single **vector** and subsequently **softmaxed**
4. Calculating the Context Vector - the encoder hidden states and their respective alignment scores are *multiplied* to form the **context vector**

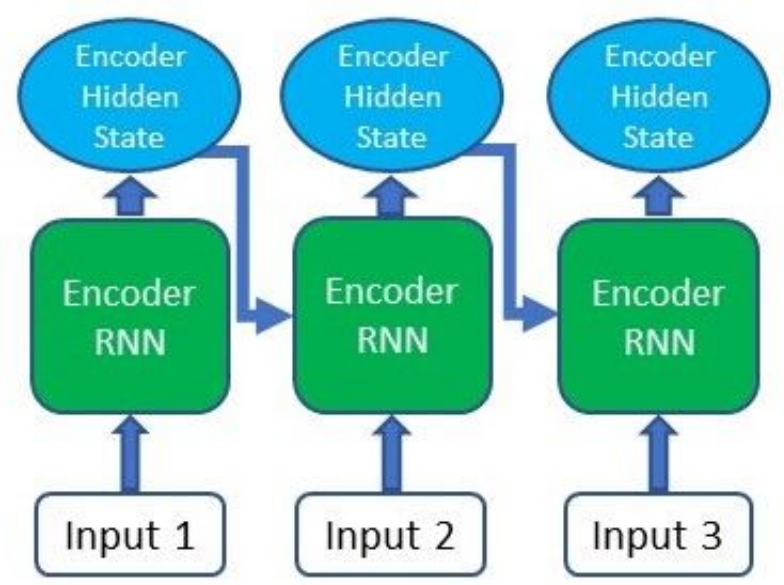
5. Decoding the Output - the context vector is *concatenated* with the previous decoder output and fed into the **Decoder RNN** for that time step along with the previous decoder hidden state to produce a **new output**
6. The process (steps 2-5) **repeats** itself for each time step of the decoder until an token is produced or output is past the specified maximum length



Flow of calculating Attention weights in Bahdanau Attention

Now that we have a high-level understanding of the flow of the Attention mechanism for Bahdanau, let's take a look at the inner workings and computations involved, together with some code implementation of a language seq2seq model with Attention in PyTorch.

1. Producing the Encoder Hidden States



Encoder RNNs will produce a hidden state for each input

For our first step, we'll be using an RNN or any of its variants (e.g. LSTM,

GRU) to encode the input sequence. After passing the input sequence through the encoder RNN, a hidden state/output will be produced for each input passed in. Instead of using only the hidden state at the final time step, we'll be carrying forward all the hidden states produced by the encoder to the next step.

```
class EncoderLSTM(nn.Module):
  def __init__(self, input_size, hidden_si
    super(EncoderLSTM, self).__init__()
    self.hidden_size = hidden_size
    self.n_layers = n_layers

    self.embedding = nn.Embedding(input_si
    self.lstm = nn.LSTM(hidden_size, hidde

  def forward(self, inputs, hidden):
    # Embed input words
    embedded = self.embedding(inputs)
    # Pass the embedded word vectors into
    output, hidden = self.lstm(embedded, h
    return output, hidden

  def init_hidden(self, batch_size=1):
    return (torch.zeros(self.n_layers, bat
            torch.zeros(self.n_layers, bat
```

In the code implementation of the encoder above, we're first embedding the input words into word vectors

(assuming that it's a language task) and then passing it through an LSTM. The encoder over here is exactly the same as a normal encoder-decoder structure without Attention.

2. Calculating Alignment Scores

For these next 3 steps, we will be going through the processes that happen in the Attention Decoder and discuss how the Attention mechanism is utilised. The class BahdanauDecoderLSTM defined below encompasses these 3 steps in the forward function.

```
class BahdanauDecoder(nn.Module):
  def __init__(self, hidden_size, output_s
    super(BahdanauDecoder, self).__init__()
    self.hidden_size = hidden_size
    self.output_size = output_size
    self.n_layers = n_layers
    self.drop_prob = drop_prob

    self.embedding = nn.Embedding(self.outp

    self.fc_hidden = nn.Linear(self.hidden_
    self.fc_encoder = nn.Linear(self.hidde
    self.weight = nn.Parameter(torch.Float
    self.attn_combine = nn.Linear(self.hid
```

```
        self.dropout = nn.Dropout(self.drop_pro
        self.lstm = nn.LSTM(self.hidden_size*2
        self.classifier = nn.Linear(self.hidde

    def forward(self, inputs, hidden, encode
        encoder_outputs = encoder_outputs.squee
        # Embed input words
        embedded = self.embedding(inputs).view(
        embedded = self.dropout(embedded)

        # Calculating Alignment Scores
        x = torch.tanh(self.fc_hidden(hidden[0]
        alignment_scores = x.bmm(self.weight.un

        # Softmaxing alignment scores to get A
        attn_weights = F.softmax(alignment_scor

        # Multiplying the Attention weights wi
        context_vector = torch.bmm(attn_weights
                                   encoder_output

        # Concatenating context vector with em
        output = torch.cat((embedded, context_v
        # Passing the concatenated vector as i
        output, hidden = self.lstm(output, hidd
        # Passing the LSTM output through a Li
        output = F.log_softmax(self.classifier(
        return output, hidden, attn_weights
```

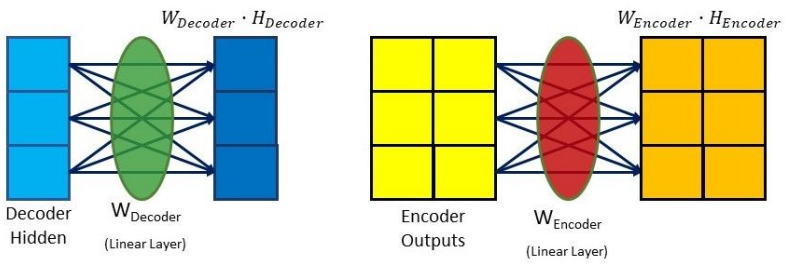
After obtaining all of our encoder outputs, we can start using the decoder to produce outputs. At each time step of the decoder, we have to calculate the alignment score of each encoder output with respect to the decoder input and hidden state at that

time step. The alignment score is the

time step. The alignment score is the essence of the Attention mechanism, as it quantifies the amount of "Attention" the decoder will place on each of the encoder outputs when producing the next output.

The alignment scores for Bahdanau Attention are calculated using the hidden state produced by the decoder in the previous time step and the encoder outputs with the following equation:

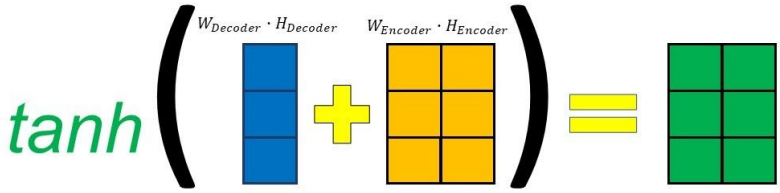
The decoder hidden state and encoder outputs will be passed through their individual Linear layer and have their own individual trainable weights.



Linear layers for encoder outputs and decoder hidden states

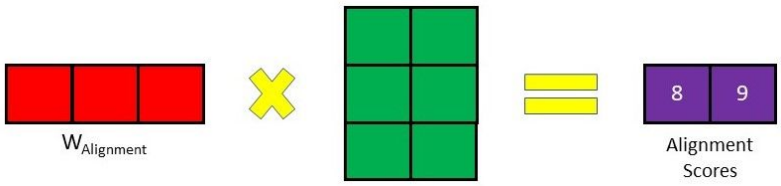
In the illustration above, the hidden size is 3 and the number of encoder outputs is 2.

Thereafter, they will be added together before being passed through a *tanh* activation function. The decoder hidden state is added to each encoder output in this case.



Above outputs combined and *tanh* applied

Lastly, the resultant vector from the previous few steps will undergo matrix multiplication with a trainable vector, obtaining a final alignment score vector which holds a score for each encoder output.



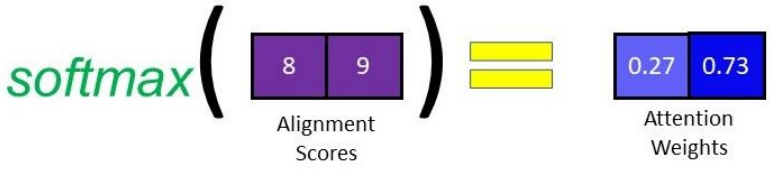
Matrix Multiplication to obtain Alignment score

Note: As there is no previous hidden state or output for the first decoder step, the last encoder hidden state and a Start Of String (<SOS>) token can be used to replace these two, respectively.

3. Softmaxing the Alignment Scores

After generating the alignment scores vector in the previous step, we can then apply a softmax on this vector to obtain the attention weights. The softmax function will cause the values in the vector to sum up to 1 and each individual value will lie between 0 and 1, therefore representing the

weightage each input holds at that time step.



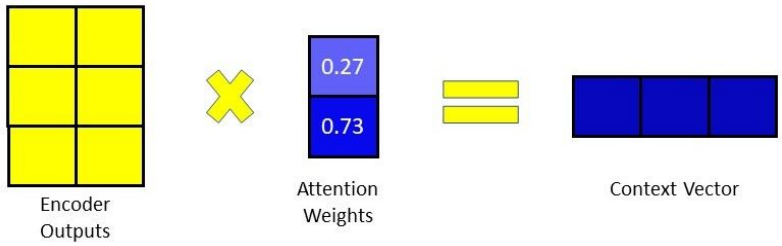
Alignment scores are softmaxed

4. Calculating the Context Vector

After computing the attention weights in the previous step, we can now generate the context vector by doing an element-wise multiplication of the attention weights with the encoder outputs.

Due to the softmax function in the previous step, if the score of a specific input element is closer to 1 its effect and influence on the decoder output is amplified, whereas if the score is close to 0, its influence is drowned out

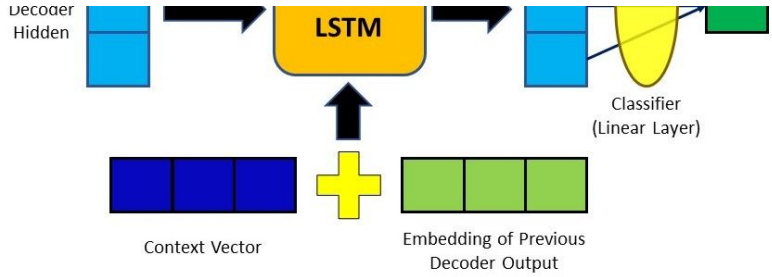
and nullified.



Context Vector is derived from the weights and encoder outputs

5. Decoding the Output

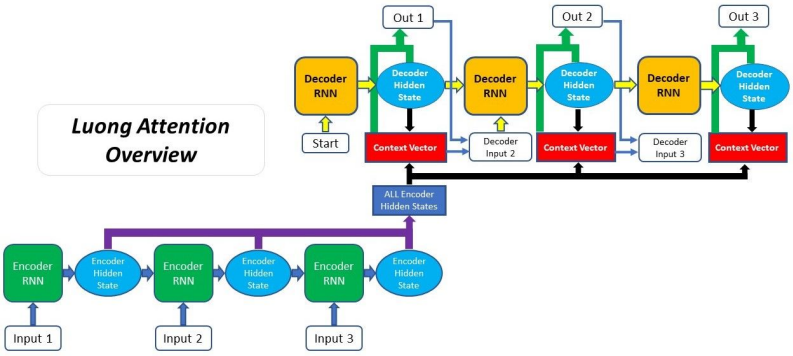
The context vector we produced will then be concatenated with the previous decoder output. It is then fed into the decoder RNN cell to produce a new hidden state and the process repeats itself from step 2. The final output for the time step is obtained by passing the new hidden state through a Linear layer, which acts as a classifier to give the probability scores of the next predicted word.



Context vector and previous output will give new decoder hidden state

Steps 2 to 4 are repeated until the decoder generates an End Of Sentence token or the output length exceeds a specified maximum length.

Luong Attention



Overall process for Luong Attention seq2seq model

The second type of Attention was proposed by Thang Luong in this

paper. It is often referred to as Multiplicative Attention and was built on top of the Attention mechanism proposed by Bahdanau. The two main differences between Luong Attention and Bahdanau Attention are:

1. The way that the alignment score is calculated
2. The position at which the Attention mechanism is being introduced in the decoder

There are three types of alignment scoring functions proposed in Luong's paper compared to Bahdanau's one type. Also, the general structure of the Attention Decoder is different for Luong Attention, as the context vector is only utilised after the RNN produced the output for that time step. We will explore these differences in greater detail as we go through the Luong Attention process, which is:

1. Producing the Encoder Hidden

1. Producing the Encoder Hidden States - Encoder produces hidden states of **each** element in the input sequence
2. Decoder RNN - the previous decoder hidden state and decoder output is passed through the **Decoder RNN** to generate a **new hidden state** for that time step
3. Calculating Alignment Scores - using the new decoder hidden state and the encoder hidden states, **alignment scores** are calculated
4. Softmaxing the Alignment Scores - the alignment scores for each encoder hidden state are combined and represented in a single **vector** and subsequently **softmaxed**
5. Calculating the Context Vector - the encoder hidden states and their respective alignment

scores are multiplied to form

scores are multiplied to form the **context vector**

6. Producing the Final Output - the context vector is *concatenated* with the decoder hidden state generated in step 2 as passed through a fully connected layer to produce a **new output**
7. The process (steps 2-6) **repeats** itself for each time step of the decoder until an token is produced or output is past the specified maximum length

As we can already see above, the order of steps in Luong Attention is different from Bahdanau Attention. The code implementation and some calculations in this process is different as well, which we will go through now.

1. Producing the Encoder Hidden States

Just as in Bahdanau Attention, the

Just as in Bahdanau Attention, the encoder produces a hidden state for each element in the input sequence.

2. Decoder RNN

Unlike in Bahdanau Attention, the decoder in Luong Attention uses the RNN in the first step of the decoding process rather than the last. The RNN will take the hidden state produced in the previous time step and the word embedding of the final output from the previous time step to produce a new hidden state which will be used in the subsequent steps

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

    self.n_layers = n_layers
    self.drop_prob = drop_prob

    # The Attention Mechanism is defined
    self.attention = attention

    self.embedding = nn.Embedding(self.ou
    self.dropout = nn.Dropout(self.drop_p
    self.lstm = nn.LSTM(self.hidden_size,
    self.classifier = nn.Linear(self.hidc
```

```
  def forward(self, inputs, hidden, encoc
    # Embed input words
    embedded = self.embedding(inputs).vie
    embedded = self.dropout(embedded)

    # Passing previous output word (embec
    lstm_out, hidden = self.lstm(embeddec

    # Calculating Alignment Scores - see
    alignment_scores = self.attention(lst
    # Softmaxing alignment scores to obtc
    attn_weights = F.softmax(alignment_sc

    # Multiplying Attention weights with
    context_vector = torch.bmm(attn_weigh

    # Concatenating output from LSTM with
    output = torch.cat((lstm_out, context
    # Pass concatenated vector through Li
    output = F.log_softmax(self.classifie
    return output, hidden, attn_weights

class Attention(nn.Module):
  def __init__(self, hidden_size, method=
    super(Attention, self).__init__()
    self.method = method
    self.hidden_size = hidden_size

    # Defining the layers/weights require
    if method == "general":
      self.fc = nn.Linear(hidden_size, hi


    elif method == "concat":
      self.fc = nn.Linear(hidden_size, hi
      self.weight = nn.Parameter(torch.Fl

  def forward(self, decoder_hidden, encoc
    if self.method == "dot":
      # For the dot scoring method, no we
      return encoder_outputs.bmm(decoder_

    elif self.method == "general":
```

```
        # For general scoring, decoder hidd
        out = self.fc(decoder_hidden)
        return encoder_outputs.bmm(out.view

    elif self.method == "concat":
        # For concat scoring, decoder hidde
        out = torch.tanh(self.fc(decoder_hi
        return out.bmm(self.weight.unsqueez
```

3. Calculating Alignment Scores

In Luong Attention, there are three different ways that the alignment scoring function is defined- dot, general and concat. These scoring functions make use of the encoder outputs and the decoder hidden state produced in the previous step to calculate the alignment scores.

- **Dot**

  The first one is the dot scoring function. This is the simplest of the functions; to produce the alignment score we only need to take the hidden states of the encoder and multiply them by the hidden state of the

  decoder

decoder.

- **General**
  The second type is called general and is similar to the dot function, except that a weight matrix is added into the equation as well.

- **Concat**
  The last function is slightly similar to the way that alignment scores are calculated in Bahdanau Attention, whereby the decoder hidden state is added to the encoder hidden states.

However, the difference lies in the

fact that the decoder hidden state and encoder hidden states are added together first before being passed through a Linear layer. This means that the decoder hidden state and encoder hidden state will not have their individual weight matrix, but a shared one instead, unlike in Bahdanau Attention.After being passed through the Linear layer, a *tanh* activation function will be applied on the output before being multiplied by a weight matrix to produce the alignment score.

4. Softmaxing the Alignment Scores

Similar to Bahdanau Attention, the alignment scores are softmaxed so that the weights will be between 0 to 1.

5. Calculating the Context Vector

Again, this step is the same as the one

in Bahdanau Attention where the attention weights are multiplied with the encoder outputs.

6. Producing the Final Output

In the last step, the context vector we just produced is concatenated with the decoder hidden state we generated in step 2. This combined vector is then passed through a Linear layer which acts as a classifier for us to obtain the probability scores of the next predicted word.

Testing The Model

Since we've defined the structure of the Attention encoder-decoder model and understood how it works, let's see how we can use it for an NLP task - Machine Translation.

We will be using English to German sentence pairs obtained from the Tatoeba Project, and the compiled sentences pairs can be found at this link. You can run the code implementation in this article on FloydHub using their GPUs on the cloud by clicking the following link and using the `main.ipynb` notebook.

This will speed up the training process significantly. Alternatively, the link to the GitHub repository can be found here.

The goal of this implementation is not to develop a complete English to German translator, but rather just as a sanity check to ensure that our model is able to learn and fit to a set of training data. I will briefly go through the data preprocessing steps before running through the training procedure.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import numpy as np
import pandas
import spacy

from spacy.lang.en import English
from spacy.lang.de import German
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
from tqdm import tqdm_notebook
import random
from collections import Counter

if torch.cuda.is_available:
    device = torch.device("cuda")
```

```
else:
    device = torch.device("cpu")
```

We start by importing the relevant libraries and defining the device we are running our training on (GPU/CPU). If you're using FloydHub with GPU to run this code, the training time will be significantly reduced. In the next code block, we'll be doing our data preprocessing steps:

1. Tokenizing the sentences and creating our vocabulary dictionaries
2. Assigning each word in our vocabulary to integer indexes
3. Converting our sentences into their word token indexes

```
# Reading the English-German sentences pai
with open("deu.txt","r+") as file:
  deu = [x[:-1] for x in file.readlines()]
en = []
de = []
for line in deu:

  en.append(line.split("\t")[0])
  de.append(line.split("\t")[1])
```

```
# Setting the number of training sentences
training_examples = 10000
# We'll be using the spaCy's English and G
spacy_en = English()
spacy_de = German()

en_words = Counter()
de_words = Counter()
en_inputs = []
de_inputs = []

# Tokenizing the English and German senten
for i in tqdm_notebook(range(training_examp
    en_tokens = spacy_en(en[i])
    de_tokens = spacy_de(de[i])
    if len(en_tokens)==0 or len(de_tokens)=
        continue
    for token in en_tokens:
        en_words.update([token.text.lower()
    en_inputs.append([token.text.lower() fo
    for token in de_tokens:
        de_words.update([token.text.lower()
    de_inputs.append([token.text.lower() fo

# Assigning an index to each word token, i
en_words = ['_SOS','_EOS','_UNK'] + sorted(
en_w2i = {o:i for i,o in enumerate(en_words
en_i2w = {i:o for i,o in enumerate(en_words
de_words = ['_SOS','_EOS','_UNK'] + sorted(
de_w2i = {o:i for i,o in enumerate(de_words
de_i2w = {i:o for i,o in enumerate(de_words

# Converting our English and German senten
for i in range(len(en_inputs)):
    en_sentence = en_inputs[i]
    de_sentence = de_inputs[i]
    en_inputs[i] = [en_w2i[word] for word
    de_inputs[i] = [de_w2i[word] for word
```

Since we've already defined our Encoder and Attention Decoder

model classes earlier, we can now instantiate the models.

```
hidden_size = 256
encoder = EncoderLSTM(len(en_words), hidde
attn = Attention(hidden_size,"concat")
decoder = LuongDecoder(hidden_size,len(de_

lr = 0.001
encoder_optimizer = optim.Adam(encoder.par
decoder_optimizer = optim.Adam(decoder.par
```

We'll be testing the **LuongDecoder** model with the scoring function set as *concat*. During our training cycle, we'll be using a method called **teacher forcing** for 50% of the training inputs, which uses the real target outputs as the input to the next step of the decoder instead of our decoder output for the previous time step. This allows the model to converge faster, although there are some drawbacks involved (e.g. instability of trained model).

```
EPOCHS = 10
teacher_forcing_prob = 0.5
encoder.train()
```

```
decoder.train()
tk0 = tqdm_notebook(range(1,EPOCHS+1),tota
for epoch in tk0:
    avg_loss = 0.
    tk1 = tqdm_notebook(enumerate(en_inputs
    for i, sentence in tk1:
        loss = 0.
        h = encoder.init_hidden()
        encoder_optimizer.zero_grad()
        decoder_optimizer.zero_grad()
        inp = torch.tensor(sentence).unsque
        encoder_outputs, h = encoder(inp,h)

        #First decoder input will be the S
        decoder_input = torch.tensor([en_w2
        #First decoder hidden state will be
        decoder_hidden = h
        output = []
        teacher_forcing = True if random.r

        for ii in range(len(de_inputs[i])):
          decoder_output, decoder_hidden, d
          # Get the index value of the word
          top_value, top_index = decoder_ou
          if teacher_forcing:
            decoder_input = torch.tensor([d
          else:
            decoder_input = torch.tensor([t
          output.append(top_index.item())
          # Calculate the loss of the predi
          loss += F.nll_loss(decoder_output
        loss.backward()
        encoder_optimizer.step()
        decoder_optimizer.step()
        avg_loss += loss.item()/len(en_inpu
    tk0.set_postfix(loss=avg_loss)
  # Save model after every epoch (Optional)
torch.save({"encoder":encoder.state_dict()
```

Using our trained model, let's visualise some of the outputs that the model produces and the attention

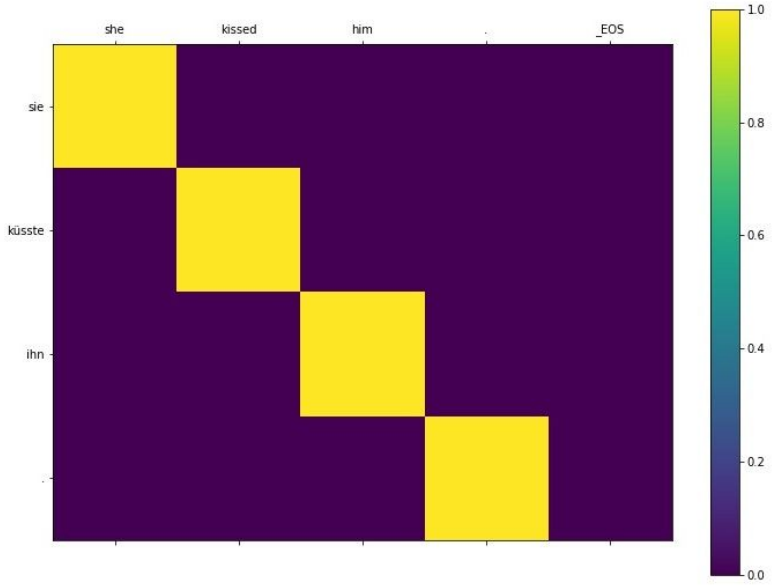
model produces and the attention weights the model assigns to each input element.

```
encoder.eval()
decoder.eval()
# Choose a random sentences
i = random.randint(0,len(en_inputs)-1)
h = encoder.init_hidden()
inp = torch.tensor(en_inputs[i]).unsqueeze(
encoder_outputs, h = encoder(inp,h)

decoder_input = torch.tensor([en_w2i['_SOS
decoder_hidden = h
output = []
attentions = []
while True:
  decoder_output, decoder_hidden, attn_weig
  _, top_index = decoder_output.topk(1)
  decoder_input = torch.tensor([top_index.
  # If the decoder output is the End Of Se
  if top_index.item() == de_w2i["_EOS"]:
    break
  output.append(top_index.item())
  attentions.append(attn_weights.squeeze()
print("English: "+ " ".join([en_i2w[x] for
print("Predicted: " + " ".join([de_i2w[x]
print("Actual: " + " ".join([de_i2w[x] for

# Plotting the heatmap for the Attention w
fig = plt.figure(figsize=(12,9))
ax = fig.add_subplot(111)
cax = ax.matshow(np.array(attentions))
fig.colorbar(cax)
ax.set_xticklabels(['']+[en_i2w[x] for x i
ax.set_yticklabels(['']+[de_i2w[x] for x i
ax.xaxis.set_major_locator(ticker.Multiple
ax.yaxis.set_major_locator(ticker.Multiple
plt.show()
```

```
[Out]: English: she kissed him . _EOS
       Predicted: sie küsste ihn .
       Actual: sie küsste ihn . _EOS
```



Plot showing the weights assigned to each input word for each output

From the example above, we can see that for each output word from the decoder, the weights assigned to the input words are different and we can see the relationship between the inputs and outputs that the model is able to draw. You can try this on a few more examples to test the results of

the translator.

In our training, we have clearly overfitted our model to the training sentences. If we were to test the trained model on sentences it has never seen before, it is unlikely to produce decent results. Nevertheless, this process acts as a sanity check to ensure that our model works and is able to function end-to-end and learn.

The challenge of training an effective model can be attributed largely to the lack of training data and training time. Due to the complex nature of the different languages involved and a large number of vocabulary and grammatical permutations, an effective model will require tons of data and training time before any results can be seen on evaluation data.

Conclusion

The Attention mechanism has

The Attention mechanism has revolutionised the way we create NLP models and is currently a standard fixture in most state-of-the-art NLP models. This is because it enables the model to "remember" all the words in the input and focus on specific words when formulating a response.

We covered the early implementations of Attention in seq2seq models with RNNs in this article. However, the more recent adaptations of Attention has seen models move beyond RNNs to Self-Attention and the realm of Transformer models. Google's BERT, OpenAI's GPT and the more recent XLNet are the more popular NLP models today and are largely based on self-attention and the Transformer architecture.

I'll be covering the workings of these models and how you can implement and fine-tune them for your own downstream tasks in my next article.

downstream tasks in my next article. Stay tuned!

About Gabriel Loye

Gabriel is an Artificial Intelligence enthusiast and web developer. He's currently exploring various fields of deep learning from Natural Language Processing to Computer Vision. He's always open to learning new things and implementing or researching on novel ideas and technologies. He'll soon start his undergraduate studies in Business Analytics at the NUS School of Computing and is currently an intern at Fintech start-up PinAlpha. Gabriel is also a FloydHub AI Writer. You can connect with Gabriel on LinkedIn and GitHub.

Subscribe to FloydHub Blog

Get the latest posts delivered right to your inbox

youremail@example.c Subscribe

Gabriel Loye

An Artificial Intelligence enthusiast, web developer and student exploring various fields of deep learning

Read More

— FloydHub Blog —

Deep Learning

NLP Datasets: How good is your deep learning model?

Tokenizers: How machines read

Distilling knowledge from Neural Networks to build smaller and faster models

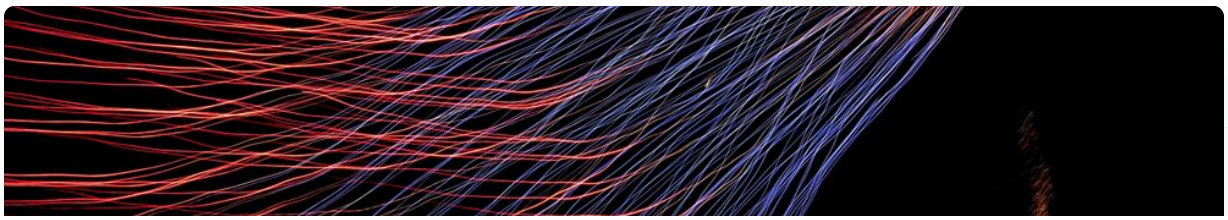
See all 44 posts →

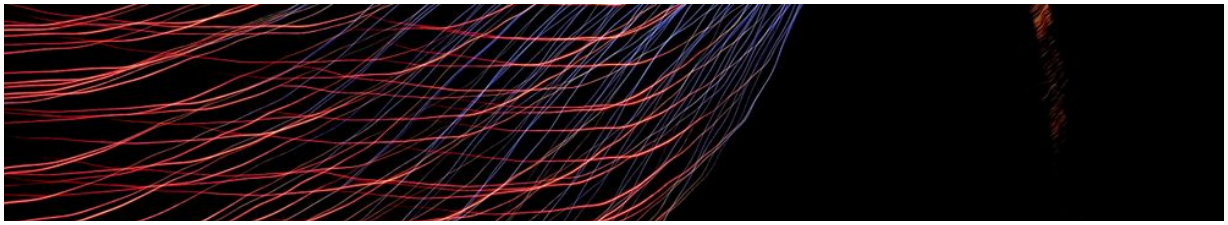
DEEP LEARNING

Training Neural Nets: a Hacker's Perspective

This deep dive is all about neural networks - training them using best practices, debugging them and maximizing their performance using cutting edge research.

26 MIN READ





DATA SCIENCE

Multiprocessing vs. Threading in Python: What Every Data Scientist Needs to Know

This deep dive on Python parallelization libraries - multiprocessing and threading - will explain which to use when for different data scientist problem sets.

14 MIN READ