Title: Abstractive Text Summarization

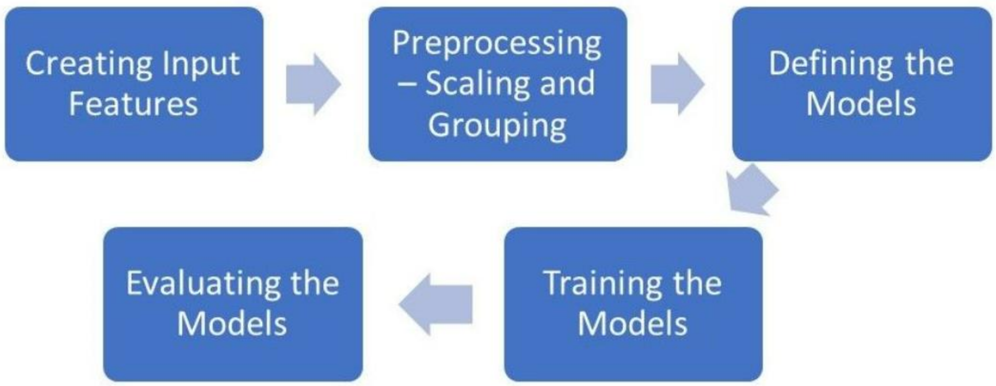
By: Ravina Ingole

Contents:

1.Introduction:

This project is regarding **Text summarization** which is the problem of reducing the number of sentences and words of the article without changing its meaning. There are different techniques to extract information from raw text data and use it for a summarization model, overall, they can be categorized as **Extractive** and **Abstractive.** Extractive methods select the most important sentences within a text (without necessarily understanding the meaning), therefore the result summary is just a subset of the full text. On the contrary, Abstractive models use advanced NLP (word embeddings) to understand the semantics of the text and generate a meaningful summary. Consequently.

In this project I have used **Abstractive method** (Sequence 2 Sequence) for text summarization, The Gated Recurrent Unit (GRU) is the younger sibling of the more popular Long Short-Term Memory (LSTM) network, and also a type of Recurrent Neural Network (RNN). Below steps are followed for training model.



Data is collected from **"The Charges Bulletin"**.
Link of data Source: https://chargerbulletin.com/
Label Preparation: Heading is considered as the target value, and context as text(input).

Mycharger Bulletien_updated.csv

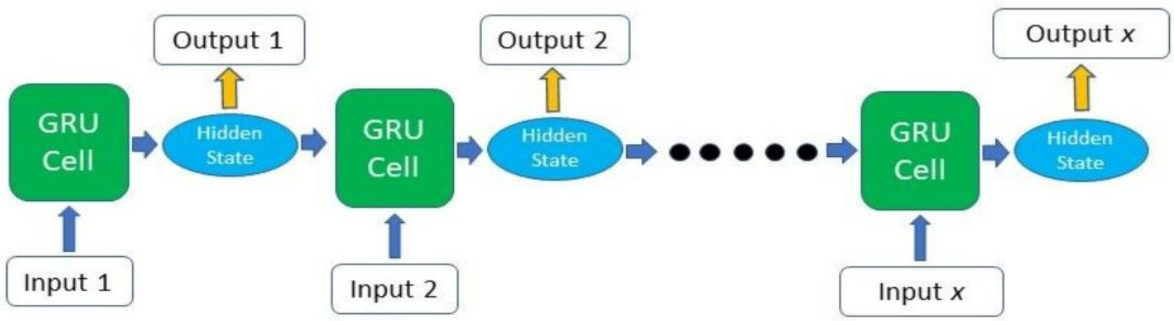
2.Methods:

For text summarization I have used PyTorch to build a **sequence 2 sequence (encoder-decoder)** model with simple dot product attention using **Gated Recurrent Unit GRU** and evaluate their attention scores.

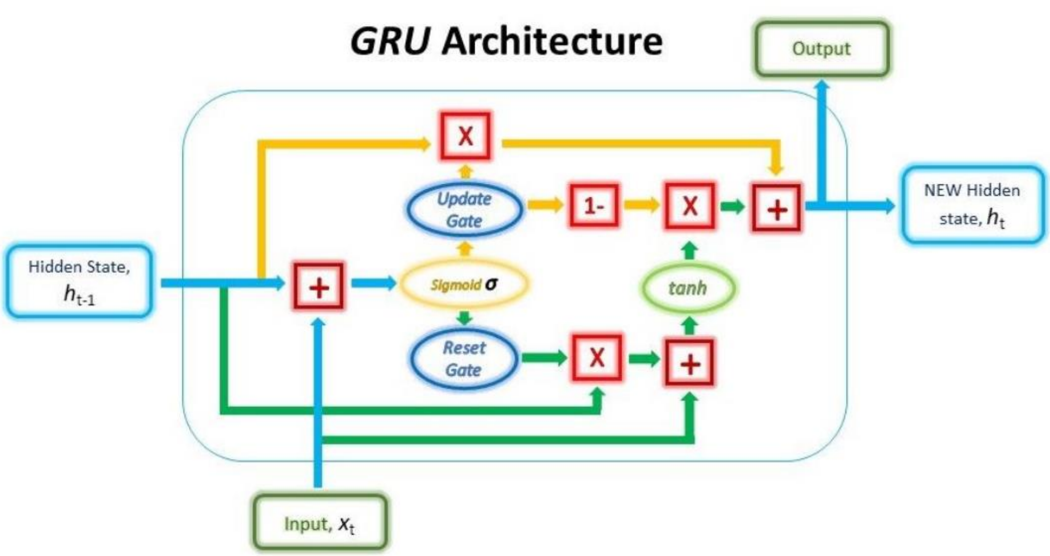
2.1 Algorithms:

The structure of the GRU allows it to adaptively capture dependencies from large sequences of data without discarding information from earlier parts of the sequence. This is achieved through its **gating** units, like the ones in LSTMs, which solve the vanishing/exploding gradient problem of traditional RNNs. These gates are responsible for regulating the information to be kept or discarded at each time step.

Other than its internal gating mechanisms, the GRU functions just like an RNN, where sequential input data is consumed by the GRU cell at each time step along with the memory, or otherwise known as the **hidden state**. The hidden state is then re-fed into the RNN cell together with the next input data in the sequence. This process continues like a relay system, producing the desired output.



The GRU cell contains only two gates: the **Update gate** and the **Reset gate**. These gates are essentially vectors containing values between *0* to *1* which will be multiplied with the input data and/or hidden state. A *0* value in the gate vectors indicates that the corresponding data in the input or hidden state is unimportant and will, therefore, return as a zero. On the other hand, a *1* value in the gate vector means that the corresponding data is important and will be used.



Encoder: The encoder layer of the seq2seq model extracts information from the input text and encodes it into a single vector, that is a context vector. I have used **GRU(Gated Recurrent Unit)** for the encoder layer in order to capture long term dependencies - mitigating the vanishing/exploding gradient problem encountered while working with vanilla RNNs. The GRU cell reads one word at a time and using the update and reset gate, computes the hidden state content andcell state.

Decoder: The decoder layer of a seq2seq model uses the last hidden state of the encoder i.e., the context vector and generates the output words. The decoding process starts once the sentence has been encoded and the decoder is given a hidden state and an input token at each step/time.

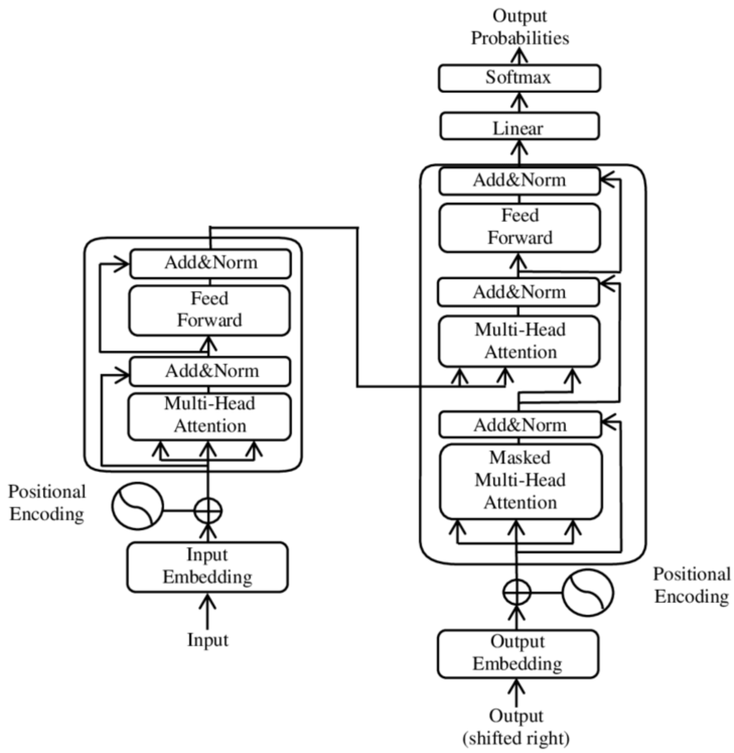
```
class Encoder(nn.Module):
   def __init__(self, input_dim, hidden_dim, embbed_dim, num_layers):
       super(Encoder, self).__init__()

       #set the encoder input dimesion , embbed dimesion, hidden dimesion, and number of layers
       self.input_dim = input_dim
       self.embbed_dim = embbed_dim
       self.hidden_dim = hidden_dim
       self.num_layers = num_layers

       #initialize the embedding layer with input and embbed dimention
       self.embedding = nn.Embedding(input_dim, self.embbed_dim)
      #change

       #intialize the GRU to take the input dimetion of embbed, and output dimention of hidden and
       #set the number of gru layers
       self.gru = nn.GRU(self.embbed_dim, self.hidden_dim, num_layers=self.num_layers)
   def forward(self, src):

       embedded = self.embedding(src).view(1,1,-1)
       outputs, hidden = self.gru(embedded)
       return outputs, hidden

class Decoder(nn.Module):
   def __init__(self, output_dim, hidden_dim, embbed_dim, num_layers):
       super(Decoder, self).__init__()

#set the encoder output dimension, embed dimension, hidden dimension, and number of layers
       self.embbed_dim = embbed_dim
       self.hidden_dim = hidden_dim
       self.output_dim = output_dim
       self.num_layers = num_layers

# initialize every layer with the appropriate dimension. For the decoder layer, it will consist of an embedding, GRU, a Linear layer and a Log softmax activation function.
       self.embedding = nn.Embedding(output_dim, self.embbed_dim)
       self.gru = nn.GRU(self.embbed_dim, self.hidden_dim, num_layers=self.num_layers)
       self.out = nn.Linear(self.hidden_dim, output_dim)
       self.softmax = nn.LogSoftmax(dim=1)

   def forward(self, input, hidden):

# reshape the input to (1, batch_size)
       input = input.view(1, -1)
       embedded = F.relu(self.embedding(input))
       output, hidden = self.gru(embedded, hidden)
       prediction = self.softmax(self.out(output[0]))

       return prediction, hidden
```

Model Architecture:

```
... Encoder(
      (embedding): Embedding(5487, 256)
      (gru): GRU(256, 512, num_layers=7)
    )
    Decoder(
      (embedding): Embedding(340, 256)
      (gru): GRU(256, 512, num_layers=7)
      (out): Linear(in_features=512, out_features=340, bias=True)
      (softmax): LogSoftmax(dim=1)
    )
    10
```

2.2 objective function and network architectures for transfer learning:

A transformer is a deep learning model that adopts the mechanism of self-attention, differentially weighting the significance of each part of the input data. It is used primarily in the fields of natural language processing (NLP) and computer vision. Below is the architecture for T5 Model.



2.3 Mini Network:

For min Network reduced GRU layer from 7 to 4. Also did some changes in the code. Below is Model on which dataset is trained on.

Model Architecture:

```
Encodermini(
  (embedding): Embedding(5487, 256)
  (gru): GRU(256, 512, num_layers=4)
)
Decodermini(
  (embedding): Embedding(340, 256)
  (gru): GRU(256, 512, num_layers=4)
  (out): Linear(in_features=512, out_features=340, bias=True)
  (softmax): LogSoftmax(dim=1)
)
6
```

2.4 Evaluation:

Used Rouge to evaluate training performance. ROUGE, or Recall-Oriented Understudy for Evaluation, is a set of metrics and a software package used for evaluating automatic summarization and machine translation software in natural language processing. The metrics compare an automatically produced summary or translation against a reference or a set of references (human-produced) summary or translation.

Note: As of now, ROUGE score is coming 0.021739129702859194 after 15000 iterations, it will improve when number of datasets is increased.

3. Contribution:

Note: Everything is done by me it’s an individual project.

Dataset Creation and cleaning dataset:

Dataset was created using University of New Haven charger Bulletin.

```
stop_words = set(stopwords.words('english'))

def text_cleaner(text,num):
  str = text.lower()
  str = BeautifulSoup(str, "lxml").text
  str = re.sub(r'\([^)]*\)', '', str)
  str = re.sub('"','', str)
  str = ' '.join([contraction_mapping[t] if t in contraction_mapping else t for t in str.split(" ")])
  str = re.sub(r"'s\b","",str)
  str = re.sub("[^a-zA-Z]", " ", str)
  str = re.sub('[m]{2,}', 'mm', str)
  if(num==0):
    str = re.sub(r'\.',' . ',str)
  if(num==0):
      tokens = [w for w in str.split() if not w in stop_words]

  else:
      tokens=str.split()
  long_words=[]
  for i in tokens:
      if len(i)>1:                                   #removing short words
          long_words.append(i)
  return (" ".join(long_words)).strip()
```

Customized dataset:

Created different functions to customize dataset that can fit according to model.

```
def prepareData(lang1, lang2, reverse=False):
    input_lang, output_lang, pairs = readLangs(lang1, lang2, reverse)
    print("Read %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
```

```
#call the function
clean_text = []
for t in df['text']:
    clean_text.append(text_cleaner(t,0))
```

```
#call the function
clean_summary = []
for t in df['headline']:
    clean_summary.append(text_cleaner(t,0))
```

```
df['text']=clean_text
df['headline']=clean_summary

df.replace('', np.nan, inplace=True)
df.dropna(axis=0,inplace=True)
```

Vocabulary Creation:

```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
```

Hyperparameter Tuning Changed:

1. **Training hyperparameters:**

   Num_epochs

   Learning_rate

   Batch_size

2. **Model Hyperparameters:**

   Load_model=False

   Device

   Input_size_encoder

   Input_size_decoder

   Output_size

   Encoder_embedding_size

   Decoder_embedding_size

   Hidden_size

   Num_layers

Customized Model for Mini-Network:

Created functions for Encoder, Decoder, and model. Edited layers, changes input, embedding and output dimensions to fit in model.

```
embed_size = 256
hidden_size = 512
num_layers = 4
num_iteration = 20000
output_size = output_lang.n_words
#create encoder-decoder model
encoder = Encodermini(input_lang.n_words, hidden_size, embed_size, num_layers)
decoder = Decodermini(output_size, hidden_size, embed_size, num_layers)

model = Seq2Seqmin(encoder, decoder, device).to(device)
#print model
print(encoder)
print(decoder)

model = trainModel(model, input_lang, output_lang, pairs, num_iteration)
evaluateRandomly(model, input_lang, output_lang, pairs,  n=1)
```

```
class Seq2Seqmin(nn.Module):
   def __init__(self, encoder, decoder, device, MAX_LENGTH=MAX_LENGTH):
       super().__init__()

#initialize the encoder and decoder
       self.encoder = encoder
       self.decoder = decoder
       self.device = device

   def forward(self, source, target, teacher_forcing_ratio=0.5):

       input_length = source.size(0) #get the input length (number of words in sentence)
       batch_size = target.shape[1]
       target_length = target.shape[0]
       vocab_size = self.decoder.output_dim

#initialize a variable to hold the predicted outputs
       outputs = torch.zeros(target_length, batch_size, vocab_size).to(self.device)

#encode every word in a sentence
       for i in range(input_length):
           encoder_output, encoder_hidden = self.encoder(source[i])

#use the encoder’s hidden layer as the decoder hidden
       decoder_hidden = encoder_hidden.to(device)

#add a token before the first predicted word
       decoder_input = torch.tensor([SOS_token], device=device)  # SOS

#topk is used to get the top K value over a list
#predict the output word from the current target word. If we enable the teaching force,  then the #next decoder input is the next word, else, use the decoder output highest value.

       for t in range(target_length):
           decoder_output, decoder_hidden = self.decoder(decoder_input, decoder_hidden)
           outputs[t] = decoder_output
           teacher_force = random.random() < teacher_forcing_ratio
           topv, topi = decoder_output.topk(1)
           input = (target[t] if teacher_force else topi)
           if(teacher_force == False and input.item() == EOS_token):
               break

       return outputs
```

Evaluation and Training Functions:

Changes code for Evaluation and training model.

```
[ ]  def evaluate(model, input_lang, output_lang, sentences, max_length=MAX_LENGTH):
         with torch.no_grad():
             input_tensor = tensorFromSentence(input_lang, sentences[0])
             output_tensor = tensorFromSentence(output_lang, sentences[1])

             decoded_words = []

             output = model(input_tensor, output_tensor)
             # print(output_tensor)

             for ot in range(output.size(0)):
                 topv, topi = output[ot].topk(1)
                 # print(topi)

                 if topi[0].item() == EOS_token:
                     decoded_words.append('<EOS>')
                     break
                 else:
                     decoded_words.append(output_lang.index2word[topi[0].item()])
         return decoded_words
     def evaluateRandomly(model, source, target, pairs, n=10):
         for i in range(n):
             output_sentence=""
             pair = random.choice(pairs)
             text = pair[0]
             summary = pair[1]
             output_words = evaluate(model, source, target, pair)
             output_sentence = ' '.join(output_words)
             print("Summary is:  ", pair[1])
             print("Predicted Summary is:",output_sentence)
             score = calculate_rogue(pair[1], output_sentence)
             print(score)
```

4. Results:

Used Rouge to evaluate training performance. ROUGE, or Recall-Oriented Understudy for Evaluation, is a set of metrics and a software package used for evaluating automatic summarization and machine translation software in natural language processing. The metrics compare an automatically produced summary or translation against a reference or a set of references (human-produced) summary or translation.

Note: As of now, ROUGE score is coming 0.021739129702859194 after 15000 iterations, it will improve when number of datasets is increased.

Discussion:

For this project Rouge score we very low, but it can get better by increasing dataset. GRU is good model for text summarization. GRU is less complex than LSTM because it has a smaller number of gates. If the dataset is small, then GRU is preferred otherwise LSTM for the larger dataset. GRU exposes the complete memory and hidden layers, but LSTM doesn't.