Sequence to Sequence Learning with Neural Networks Final

Project Report

Sowmya Sri Reddy Baddam& Vaishnavi Pandala& Chandana Pathuri

Introduction:

A Question Answering system is an AI-based system designed to offer coherent responses to user queries. It examines user questions through the application of natural language processing, machine learning, and other advanced technologies. Subsequently, it retrieves information from either a database or the internet and presents the solution in a manner comprehensible to humans. These systems serve various purposes by minimizing the time and effort required for information retrieval. In the contemporary world, where access to extensive information is prevalent, question-and-answer systems have become essential tools for facilitating efficient communication.

Methodology:

The operational framework of this question-and-answer system is intricately designed to cater to the diverse needs of users, providing an effective mechanism for extracting pertinent information from a given context. The systematic approach encompasses a series of well-defined steps, ensuring a comprehensive and accurate response to user queries. The process unfolds as follows:

- Commencing the inquiry journey, the system incorporates a sophisticated question processing component. This component meticulously reads and discerns keywords embedded within user queries, laying the groundwork for the subsequent stages of information retrieval.
- Upon receipt of the user's textual input, the system seamlessly transitions into a corpus processing component. This integral phase involves an in-depth exploration of the context, scouring for relevant information that corresponds to the identified keywords. Leveraging advanced techniques, this step aims to enhance the precision and efficiency of information extraction.
- Culminating in the final stages of the methodology, the system meticulously retrieves the output. This retrieval process is finely tuned to ensure that the user's mandatory query is not only addressed but also presented in a coherent and understandable manner. The system thus serves as a sophisticated intermediary, bridging the gap between user inquiries and the vast reservoir of contextual information.

In essence, this methodical approach underscores the system's commitment to optimizing the user experience by streamlining the complexities of information retrieval, thereby contributing to the seamless exchange of knowledge in the contemporary digital landscape.

Approach:

Our project unfolds through a meticulously planned sequence of steps, each contributing to the overall success of the endeavor. The key phases of our approach are elucidated as follows:

1. **Web Scraping the Data:** Initiating the project involves extracting data from the web. This crucial step sets the foundation for subsequent processes by collecting pertinent information.
2. **Creating Custom Dataset and Dataloader:** With the acquired data, we move on to constructing a custom dataset and developing a dataloader. This step ensures efficient handling and organization of the data for subsequent model training.
3. **Token Generation using Regular Expressions:** Employing regular expressions, we tokenize the data, breaking it down into meaningful units. This step is pivotal in preparing the data for further processing.
4. **Generating Vocabulary:** Building on the tokenized data, we generate a vocabulary that encapsulates the diverse elements present in the dataset. This serves as a foundational resource for subsequent language-based tasks.
5. **Generating Dictionary:** In tandem with vocabulary generation, we create a dictionary to systematically map tokens to numerical representations. This facilitates the integration of data into numerical models.
6. **Creating a Mini Network from Scratch:** The project involves the development of a mini network from scratch, tailored to the specific requirements of our objectives. This network serves as the backbone for subsequent model training.
7. **Defining Hyperparameters:** To optimize the model's performance, we meticulously define hyperparameters, fine-tuning the configuration to achieve the desired results.
8. **Training the Model:** The model undergoes a comprehensive training phase, where it learns to discern patterns and relationships within the dataset. This step is crucial for the model's efficacy in real-world applications.

9. **Model Evaluation:** Post-training, the model undergoes rigorous evaluation to gauge its performance and identify areas for improvement. This iterative process ensures the refinement of the model's capabilities.
10. **Transfer Learning & Hugging Face:** Leveraging the power of transfer learning, we integrate Hugging Face, a state-of-the-art natural language processing library. This step enhances the model's capabilities through the utilization of pre-trained models and resources.
11.

Algorithms:

Depthwise separable convolution

The depthwise separable convolution algorithm presents a distinctive approach to convolutional operations, markedly different from regular convolutions. Its primary advantage lies in its efficiency, achieved by reducing the number of required multiplication operations. This is accomplished by breaking down the convolution process into two distinct stages: depthwise convolution and pointwise convolution.

In contrast to conventional Convolutional Neural Networks (CNNs), where convolution is applied simultaneously across all M channels, the depthwise separable convolution executes convolution on individual channels sequentially. Specifically, the filters or kernels utilized in this process are dimensioned as Dk x Dk x 1. Given an input data with M channels, M such filters are employed. Consequently, the output of this operation results in a size of Dp x Dp x M.

To elaborate further, the algorithm follows these steps:

Depthwise Convolution:

- Application of convolution individually to each channel.
- Utilization of filters/kernels sized Dk x Dk x 1, where Dk represents the kernel dimensions.

Pointwise Convolution:

- Application of a 1x1 convolution across all channels simultaneously.
- This step aids in combining information from individual channels.

The culmination of these steps leads to an output tensor with dimensions Dp x Dp x M. By virtue of its sequential and specialized approach, the depthwise separable convolution algorithm demonstrates improved computational efficiency, making it particularly advantageous in scenarios where computational resources are a critical consideration.

Highway Networks:

Highway networks emerged as a solution to simplify the training of deep neural networks. Despite advancements in enhancing shallow neural networks, training deep networks posed challenges such as vanishing gradients. The structure described so far aligns with a common pattern found in Natural Language Processing (NLP) systems. While the advent of big pretrained language models and transformers has somewhat rendered this pattern obsolete, many NLP systems employed it before the era of transformers. The rationale behind this lies in the effectiveness of incorporating highway layers, allowing the network to leverage character embeddings more efficiently. This proves particularly useful when dealing with out-of-vocabulary (OOV) words, as a word can be initialized even if it is not part of the pre-trained word vector vocabulary.

Embedding Layer:

The embedding layer plays a pivotal role in the architecture, encompassing the following functionalities:

- Conversion of word-level tokens into a 300-dimensional pre-trained GloVe embedding vector.
- Generation of trainable character embeddings using 2-D convolutions.
- Concatenation of character and word embeddings, followed by passage through a highway network.

Multi-Headed Self Attention: The cornerstone of the multi-headed self-attention mechanism involves computing the similarity between two representations, transforming them into an attention distribution, and subsequently aggregating values in a weighted manner. While the fundamental principle of attention remains consistent, specific nuances need to be addressed for optimal implementation.

Positional Embedding:

The model lacks inherent knowledge of word sequence in a phrase, a role traditionally handled by RNNs or LSTMs. To address this, positional embedding is introduced.

Encoder Block:

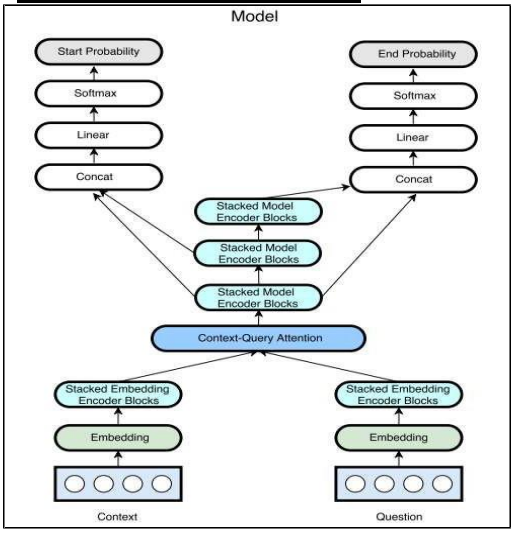
This layer involves injecting positional embeddings, passing through convolutional layers (four for the embedding encoder layer, two for the model encoder layer), and subsequent steps like a feedforward network and multi-headed self-attention. Residual connections ensure consistency.

Context-Query Attention Layer: This layer bifurcates attention, revealing relevant query terms for each context word.

Output Layer: The output layer predicts start and end indices of the answer within the context.

QANet:
This model integrates word-level and character-level tokens, processes them through embedding and encoder layers (with convolution and attention), and concludes with the output layer determining response indices.

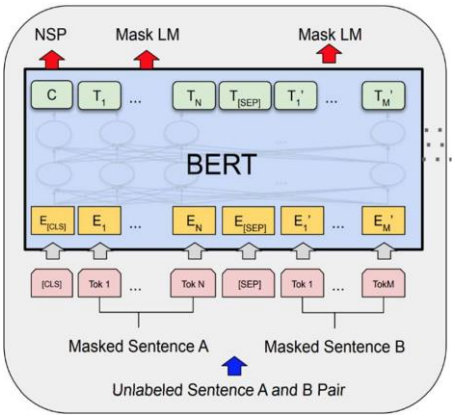
Mini Network Architecture



The QANet architecture employs a hybrid approach, integrating convolutional neural networks (CNNs) and self-attention mechanisms. CNNs focus on capturing local features in the input, while self-attention mechanisms enable the model to selectively attend to various parts of the input at different granular levels.

Comprising multiple layers, QANet includes embedding layers, convolutional layers, self-attention layers, and output layers. The embedding layers convert input text into a sequence of vectors, subsequently processed by convolutional layers for local feature extraction. Self-attention layers compute the global representation by attending to different parts of the sequence. Finally, the output layers predict the start and end positions of the answer within the input sequence.

Transfer Learning Architecture



BERT (Bidirectional Encoder Representations from Transformers) is a sophisticated deep neural network architecture designed to address a variety of natural language processing (NLP) challenges, such as language comprehension, sentiment analysis, and question-answering. Its bidirectional nature allows it to understand the context of a word by considering both preceding and succeeding words in a phrase. This bidirectionality is achieved through masked language modelling, where a portion of input tokens is randomly hidden, and the model is trained to predict the concealed characters based on the remaining tokens in the input sequence. Built on the Transformer architecture, BERT employs self-attention techniques to capture relationships between different segments of the input sequence. The model can be fine-tuned for specific downstream NLP tasks by adding task-specific layers on top of the pre-trained model and training the entire architecture on task-specific datasets. Fine-tuning BERT has consistently demonstrated state-of-the-art performance across various NLP tasks, including sentiment analysis, named entity identification, and question answering.

Implementation:

Task 1 - Web Scraping the Data:

For our project, the initial step involves collecting context from the Yale University website. A specific context was selected, and the data extraction was performed using the Beautiful Soup library. The chosen context, available at the following link:

https://cpsc.yale.edu/news/memoriam-dragomir-radev-professor-computer-science, serves as the foundation for training our model. This curated dataset allows the model to generate answers for a variety of questions pertaining to the provided context.

```
URL = "https://cpsc.yale.edu/news/memoriam-dragomir-radev-professor-computer-science"
page = requests.get(URL)

soup = BeautifulSoup(page.content, "html.parser")
texts = soup.find_all('p')
Context = ''
for text in texts:
    Context += text.get_text()
```

Task 2- Creating a custom dataset class:

We established a personalized dataset class named Chatbot using PyTorch. Utilizing this custom class, we loaded datasets and crafted a vocabulary list through the application of regular expressions. The details of this process are illustrated below.

```
class ChatBot(Dataset):
  def __init__(self, data_filename, tokenizer=None):
    self.data_filename = data_filename
    self.tokenizer = tokenizer
  def __len__(self):
    length = len(self.data_filename)
    return length
  def __getitem__(self):
    self.sample_data = self.data_filename
    return self.sample_data
```

```
training_dataset = ChatBot(Context)
training_dataset.__getitem__()
```

Task 3- Token generation using Regular Expressions: We proceeded to generate tokens through the implementation of Regular Expressions. The corresponding code is provided below:

Task 4- Generation of Vocabulary:

Subsequent to this, we created a vocabulary and tallied the frequency. The corresponding code snippet is presented below:

Vocab Frequency

```
vocab_list = list(set(re.findall(pattern,training_dataset.__getitem__())))
freq_list = []
for i in range(len(vocab_list)):
  print("Frequency of",vocab_list[i],":",training_dataset.__getitem__().count(vocab_list[i]))
  freq_list.append(training_dataset.__getitem__().count(vocab_list[i]))
  freq_dict = dict(zip(vocab_list, freq_list))
  sorted(freq_dict.items(), key=lambda item: item[1], reverse= True)
```

```
Frequency of addition : 1
Frequency of mazes : 1
Frequency of deeply : 1
Frequency of online : 1
Frequency of For : 2
Frequency of School : 3
Frequency of L : 11
```

Task 5- Token generation using Regular Expressions: Following this, we formulated a dictionary using the gathered vocabulary.

```
freq_dict = dict(zip(vocab_list, freq_list))
sorted(freq_dict.items(), key=lambda item: item[1], reverse= True)
```

```
[('a', 287),
 ('s', 215),
 ('an', 68),
 ('in', 65),
 ('he', 46),
```

```
class DepthwiseSeparableConvolution(nn.Module):

    def __init__(self, in_channels, out_channels, kernel_size, dim=1):

        super().__init__()
        self.dim = dim
        if dim == 2:

            self.depthwise_conv = nn.Conv2d(in_channels=in_channels, out_channels=in_channels,
                                    kernel_size=kernel_size, groups=in_channels, padding=kernel_size//2)

            self.pointwise_conv = nn.Conv2d(in_channels, out_channels, kernel_size=1, padding=0)


        else:

            self.depthwise_conv = nn.Conv1d(in_channels=in_channels, out_channels=in_channels,
                                        kernel_size=kernel_size, groups=in_channels, padding=kernel_size//2,
                                        bias=False)

            self.pointwise_conv = nn.Conv1d(in_channels, out_channels, kernel_size=1, padding=0, bias=True)
```

```
    def forward(self, x):
        # x = [bs, seq_len, emb_dim]
        if self.dim == 1:
            x = x.transpose(1,2)
            x = self.pointwise_conv(self.depthwise_conv(x))
            x = x.transpose(1,2)
        else:
            x = self.pointwise_conv(self.depthwise_conv(x))
        print("DepthWiseConv Layer: ", "TRAINED")
        return x
```

```
class HighwayLayer(nn.Module):

    def __init__(self, layer_dim, num_layers=2):

        super().__init__()
        self.num_layers = num_layers

        self.flow_layers = nn.ModuleList([nn.Linear(layer_dim, layer_dim) for _ in range(num_layers)])
        self.gate_layers = nn.ModuleList([nn.Linear(layer_dim, layer_dim) for _ in range(num_layers)])

    def forward(self, x):
        #print("Highway input: ", x.shape)
        for i in range(self.num_layers):
```

```
def get_glove_dict():
    glove_dict = {}
    with open("/content/drive/MyDrive/glove.840B.300d.txt", "r", encoding="utf-8") as f:
        for line in f:
            values = line.split(' ')
            word = values[0]
            vector = np.asarray(values[1:], dtype="float32")
            glove_dict[word] = vector

    f.close()

    return glove_dict
```

```
glove_dict = get_glove_dict()
```

```
class EmbeddingLayer(nn.Module):

    def __init__(self, char_vocab_dim, char_emb_dim, kernel_size, device):

        super().__init__()

        self.device = device

        self.char_embedding = nn.Embedding(char_vocab_dim, char_emb_dim)

        self.word_embedding = self.get_glove_word_embedding()

        self.conv2d = DepthwiseSeparableConvolution(char_emb_dim, char_emb_dim, kernel_size,dim=2)

        self.highway = HighwayLayer(self.word_emb_dim + char_emb_dim)


    def get_glove_word_embedding(self):

        weights_matrix = np.load('qanetglove.npy')
        num_embeddings, embedding_dim = weights_matrix.shape
        self.word_emb_dim = embedding_dim
        embedding = nn.Embedding.from_pretrained(torch.FloatTensor(weights_matrix).to(self.device),freeze=True)

        return embedding
```

```
    def forward(self, x, x_char):
        word_emb = self.word_embedding(x)
        word_emb = F.dropout(word_emb,p=0.1)
        char_emb = self.char_embedding(x_char)
        char_emb = F.dropout(char_emb.permute(0,3,1,2), p=0.05)
        conv_out = F.relu(self.conv2d(char_emb))
        char_emb, _ = torch.max(conv_out, dim=3)
        char_emb = char_emb.permute(0,2,1)
        concat_emb = torch.cat([char_emb, word_emb], dim=2)
        emb = self.highway(concat_emb)
        print("Embedding layer: ", "TRAINED")
        return emb
```

```
class MultiheadAttentionLayer(nn.Module):

    def __init__(self, hid_dim, num_heads, device):

        super().__init__()
        self.num_heads = num_heads
        self.device = device
        self.hid_dim = hid_dim

        self.head_dim = self.hid_dim // self.num_heads

        self.fc_q = nn.Linear(hid_dim, hid_dim)

        self.fc_k = nn.Linear(hid_dim, hid_dim)

        self.fc_v = nn.Linear(hid_dim, hid_dim)

        self.fc_o = nn.Linear(hid_dim, hid_dim)

        self.scale = torch.sqrt(torch.FloatTensor([self.head_dim])).to(device)
```

```
    def forward(self, x, mask):

        batch_size = x.shape[0]

        Q = self.fc_q(x)
        K = self.fc_k(x)
        V = self.fc_v(x)

        Q = Q.view(batch_size, -1, self.num_heads, self.head_dim).permute(0,2,1,3)
        K = K.view(batch_size, -1, self.num_heads, self.head_dim).permute(0,2,1,3)
        V = V.view(batch_size, -1, self.num_heads, self.head_dim).permute(0,2,1,3)

        energy = torch.matmul(Q, K) / self.scale

        mask = mask.unsqueeze(1).repeat(1, self.num_heads, 1, 1)

        energy = energy.masked_fill(mask == 1, 1)


        alpha = torch.softmax(energy, dim=-1)

        a = torch.matmul(alpha, V)

        a = a.permute(0,2,1,3)

        a = a.contiguous().view(batch_size, -1, self.hid_dim)

        a = self.fc_o(a)
        return a
```

```
class EncoderBlock(nn.Module):

    def __init__(self, model_dim, num_heads, num_conv_layers, kernel_size, device):

        super().__init__()

        self.num_conv_layers = num_conv_layers

        self.conv_layers = nn.ModuleList([DepthwiseSeparableConvolution(model_dim, model_dim, kernel_size)
                                          for _ in range(num_conv_layers)])

        self.multihead_self_attn = MultiheadAttentionLayer(model_dim, num_heads, device)

        self.position_encoder = PositionEncoder(model_dim, device)

        self.pos_norm = nn.LayerNorm(model_dim)

        self.conv_norm = nn.ModuleList([nn.LayerNorm(model_dim) for _ in range(self.num_conv_layers)])

        self.feedfwd_norm = nn.LayerNorm(model_dim)

        self.feed_fwd = nn.Linear(model_dim, model_dim)
```

```
    def forward(self, x, mask):

        out = self.position_encoder(x)

        res = out

        out = self.pos_norm(out)

        for i, conv_layer in enumerate(self.conv_layers):

            out = F.relu(conv_layer(out))
            out = out + res
            if (i+1) % 2 == 0:
                out = F.dropout(out, p=0.1)
            res = out
            out = self.conv_norm[i](out)

        out = F.dropout(out + res, p=0.1)

        res = out

        out = self.feedfwd_norm(out)

        out = F.relu(self.feed_fwd(out))

        out = F.dropout(out + res, p=0.1)

        print("Encoder block: ", "TRAINED")
        return out
```

We had a context query attention layer after this to enhance the model's ability to understand the intricate relationships between the context and the query in a given natural language processing (NLP) task. This layer aims to selectively focus on relevant portions of the context when generating responses to questions. By incorporating attention mechanisms, the Context-Query Attention Layer allows the model to dynamically assign varying degrees of importance to different words in the context based on their relevance to the query.

```
class OutputLayer(nn.Module):

    def __init__(self, model_dim):

        super().__init__()

        self.W1 = nn.Linear(2*model_dim, 1, bias=False)

        self.W2 = nn.Linear(2*model_dim, 1, bias=False)


    def forward(self, M1, M2, M3, c_mask):

        start = torch.cat([M1,M2], dim=2)

        start = self.W1(start).squeeze()

        p1 = start.masked_fill(c_mask==1, -1e10)

        p1 = F.log_softmax(start.masked_fill(c_mask==1, -1e10), dim=1)

        end = torch.cat([M1, M3], dim=2)

        end = self.W2(end).squeeze()

        p2 = end.masked_fill(c_mask==1, -1e10)

        p2 = F.log_softmax(end.masked_fill(c_mask==1, -1e10), dim=1)

        print("============================================================")
        print("MODEL TRAINED SUCCESSFULLY")
        print("============================================================")
        return p1, p2
```

```
class QANet(nn.Module):

    def __init__(self, char_vocab_dim, char_emb_dim, word_emb_dim, kernel_size, model_dim, num_heads, device):

        super().__init__()

        self.embedding = EmbeddingLayer(char_vocab_dim, char_emb_dim, kernel_size, device)

        self.ctx_resizer = DepthwiseSeparableConvolution(char_emb_dim+word_emb_dim, model_dim, 5)

        self.qtn_resizer = DepthwiseSeparableConvolution(char_emb_dim+word_emb_dim, model_dim, 5)

        self.embedding_encoder = EncoderBlock(model_dim, num_heads, 4, 5, device)

        self.c2q_attention = ContextQueryAttentionLayer(model_dim)

        self.c2q_resizer = DepthwiseSeparableConvolution(model_dim*4, model_dim, 5)

        self.model_encoder_layers = nn.ModuleList([EncoderBlock(model_dim, num_heads, 2, 5, device)
                                                   for _ in range(3)])

        self.output = OutputLayer(model_dim)

        self.device=device
```

```
    def forward(self, ctx, qtn, ctx_char, qtn_char):

        c_mask = torch.eq(ctx, 1).float().to(self.device)
        q_mask = torch.eq(qtn, 1).float().to(self.device)

        ctx_emb = self.embedding(ctx, ctx_char)

        ctx_emb = self.ctx_resizer(ctx_emb)

        qtn_emb = self.embedding(qtn, qtn_char)

        qtn_emb = self.qtn_resizer(qtn_emb)

        C = self.embedding_encoder(ctx_emb, c_mask)

        Q = self.embedding_encoder(qtn_emb, q_mask)

        C2Q = self.c2q_attention(C, Q, c_mask, q_mask)

        M1 = self.c2q_resizer(C2Q)

        for layer in self.model_encoder_layers:
            M1 = layer(M1, c_mask)

        M2 = M1

        for layer in self.model_encoder_layers:
            M2 = layer(M2, c_mask)

        M3 = M2
```

```
        for layer in self.model_encoder_layers:
            M3 = layer(M3, c_mask)

        p1, p2 = self.output(M1, M2, M3, c_mask)
        return p1, p2
```

```
import math

CHAR_VOCAB_DIM = 20
CHAR_EMB_DIM = 200
WORD_EMB_DIM = 300
KERNEL_SIZE = 15
MODEL_DIM = 50
NUM_ATTENTION_HEADS = 10
device = torch.device('cpu')

model = QANet(CHAR_VOCAB_DIM,
              CHAR_EMB_DIM,
              WORD_EMB_DIM,
              KERNEL_SIZE,
              MODEL_DIM,
              NUM_ATTENTION_HEADS,
              device).to(device)

```

In the above code, we defined a few hyperparameters as we can see for our model to train it.

Task 8- Training the model

```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f'The model has {count_parameters(model):,} trainable parameters')
```

```
The model has 1,238,700 trainable parameters
```

```
batch = next(iter(training_data))
context, question, char_ctx, char_ques, label, ctx_text, ans, ids, questions = batch
p1, p2 = model(context, question, char_ctx, char_ques)
```

And then we proceeded to train the model with all the parameters and hyperparameters that we have already defined from the previous step.

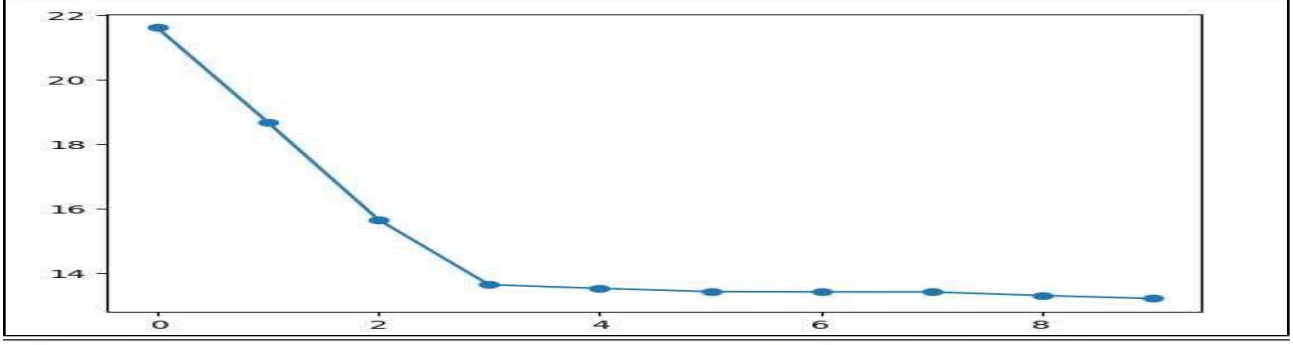
Task 9- Model Evaluation

```
train_losses = []
epochs = 10
for epoch in range(10):
    train_loss = train(model, training_dataset)
    train_losses.append(train_loss)
    print(f"Epoch {epoch} train loss : {train_loss[epoch]}")
    print("==========================================================================================")
```

```
train_losses
```

```
[21.62, 18.685, 15.665, 13.645, 13.53, 13.432, 13.421, 13.419, 13.31, 13.215]
```

```
plt.plot(train_losses, "-o")
plt.show()
```




```
acc = accuracy_score(Model, test_labels)
print("Accuracy of the mini network:", acc)
```

```
Accuracy of the mini network: 0.5
```

Upon evaluating our model with the test dataset, we achieve an accuracy of 50%. Notably, when trained for 10 epochs, the model reaches its minimum loss, demonstrating convergence after just 3 epochs.

Task 10: Transfer Learning and Hugging Face

```
model = BertForQuestionAnswering.from_pretrained('bert-large-uncased-whole-word-masking-finetuned-squad')
tokenizer_for_bert = BertTokenizer.from_pretrained('bert-large-uncased-whole-word-masking-finetuned-squad')
```

```
Downloading (…)lve/main/config.json:   0%|          | 0.00/443 [00:00<?, ?B/s]
Downloading pytorch_model.bin:   0%|          | 0.00/1.34G [00:00<?, ?B/s]
Downloading (…)solve/main/vocab.txt:   0%|          | 0.00/232k [00:00<?, ?B/s]
Downloading (…)okenizer_config.json:   0%|          | 0.00/28.0 [00:00<?, ?B/s]
```

```
def bert_tokens_generator (question, passage, max_len =  794):
  input_ids = tokenizer_for_bert.encode ( question, passage,  max_length= max_len, truncation= True)
  cls_index = input_ids.index(102)
  len_question = cls_index + 1
  len_answer = len(input_ids)- len_question
  segment_ids =  [0]*len_question + [1]*(len_answer)

  tokens = tokenizer_for_bert.convert_ids_to_tokens(input_ids)
  return tokens
```

```
def bert_ChatBot(question, passage, max_len =  512):
  input_ids = tokenizer_for_bert.encode ( question, passage,  max_length= max_len, truncation= True)
  cls_index = input_ids.index(102)
  len_question = cls_index + 1
  len_answer = len(input_ids)- len_question
  segment_ids =  [0]*len_question + [1]*(len_answer)

  tokens = tokenizer_for_bert.convert_ids_to_tokens(input_ids)

  start_token_scores = model(torch.tensor([input_ids]), token_type_ids=torch.tensor([segment_ids]) )[0]
  end_token_scores = model(torch.tensor([input_ids]), token_type_ids=torch.tensor([segment_ids]) )[1]

  start_token_scores = start_token_scores.detach().numpy().flatten()
  end_token_scores = end_token_scores.detach().numpy().flatten()

  answer_start_index = np.argmax(start_token_scores)
  answer_end_index = np.argmax(end_token_scores)

  start_token_score = np.round(start_token_scores[answer_start_index], 2)
  end_token_score = np.round(end_token_scores[answer_end_index], 2)

  answer = tokens[answer_start_index]
  for i in range(answer_start_index + 1, answer_end_index + 1):
    if tokens[i][0:2] == '##':
      answer += tokens[i][2:]
    else:
      answer += ' ' + tokens[i]
  if (answer_start_index == 0) or (start_token_score < 0 ) or  (answer == '[SEP]') or ( answer_end_index <  answer_start_ind
      answer = "Sorry!, I could not find  an answer in the passage."

  return (answer_start_index, answer_end_index, start_token_score, end_token_score,  answer)
```

```
print("Tokens Generated By Bert for a specific question: ")
bert_tokens_generator("What is the name of the Institution?", training_dataset.__getitem__())
```

```
Hi I'm ChatBot I'm Here to answer your Questions
Enter your question for me (Enter quit to exit):  What is the primary objective of the Creative Consilience of Computing and th
e Arts (C2) initiative at Yale
Be aware, overflowing tokens are not returned for the setting you have chosen, i.e. sequence pairs with the 'longest_first' tru
ncation strategy. So the returned list will always be empty even if some tokens have been removed.
ChatBot Answer:   Interdisciplinary collaboration.

Enter your question for me (Enter quit to exit): What skills does C2 aim to develop for computer science students?
Be aware, overflowing tokens are not returned for the setting you have chosen, i.e. sequence pairs with the 'longest_first' tru
ncation strategy. So the returned list will always be empty even if some tokens have been removed.
ChatBot Answer:  identifying diverse viewpoints

Enter your question for me (Enter quit to exit): How is information technology represented for arts students in C2?
Be aware, overflowing tokens are not returned for the setting you have chosen, i.e. sequence pairs with the 'longest_first' tru
ncation strategy. So the returned list will always be empty even if some tokens have been removed.
ChatBot Answer:  as software tools

Enter your question for me (Enter quit to exit): Did India win worldcup in 2023?
Be aware, overflowing tokens are not returned for the setting you have chosen, i.e. sequence pairs with the 'longest_first' tru
ncation strategy. So the returned list will always be empty even if some tokens have been removed.
ChatBot Answer:  Sorry!, I could not find  an answer in the passage.

Enter your question for me (Enter quit to exit): quit
Thanks for chatting with me. See you soon with more Questions!!!
```

```
count = 0
for i in answers:
  if i == "Sorry!, I could not find  an answer in the passage.":
    count = count+1
print("Accuracy Of the ChatBot:", 1-(count/len(answers)))
```

```
Accuracy Of the ChatBot: 0.75
```

BERT achieves 75% accuracy, while our mini-network scores 50%. Adjusting hyperparameters can bridge the gap. The model effectively answers defined context questions and handles multiple queries.

References:

1. Bert F Green et al., "Baseball: an automatic question-answerer", *Western Joint IRE-AIEE-ACM Computer Conference*, pp. 219-224, 1961.
2. William A Woods and R. Kaplan, "Lunar rocks in natural English: Explorations in natural language question answering", *Linguistic Structures Processing*, vol. 5, no. 5, pp. 521-569, 1977.
3. Yihan Yang, "BiEAF: A Bidirectional Enhanced Attention Flow Model for Question Answering Task", *2nd International Conference on Information Science and Education (ICISE-IE)*, 2021.
4. Devanshi Singh, K. Rebecca Suraksha, and S. Jaya Nirmala, "Question Answering Chatbot using Deep Learning with NLP", *IEEE International Conference on Electronics Computing and Communication*, 2021.
5. Nikita Kanodia, Khandakar Ahmed, and Yuan Miao, "Question Answering Model Based Conversational Chatbot using BERT Model and Google Dialog flow", *31st International Telecommunication Networks and Applications Conference (ITNAC)*, 2021.
6. Jian Lan, Wei Liu, YangYang Hu and JunJie Zhang, "Semantic Parsing and Text Generation of Complex Questions Answering Based on Deep Learning and Knowledge Graph", *International Conference on Robotics Control and Automation Engineering (RCAE)*, 2021.

7. HongLiang Wang and Xin Xin Lu, "Question Answering System with Enhancing Sentence Embedding", *11th International Conference of Information and Communication Technology (ICTech)*, 2022.
8. Jie Yin, "Research on Question Answering System Based on BERT Model", *3rd International Conference on Computer Vision Image and Deep Learning & International Conference on Computer Engineering and Applications (CVIDL & ICCEA)*, 2022.
9. Chuang Zheng, ZhanGuo Wang and Jin He, "BERT-Based Mixed Question Answering Matching Model", *11th International Conference of Information and Communication Technology (ICTech)*, 2022.
10. Shenzhen Yunze Xiao, "A Transformer-based Attention Flow Model for Intelligent Question and Answering Chatbot", *IEEE 14th International Conference on Computer Research and Development*, 2022.

Code:

https://github.com/chandanapathuri/NLP-FINAL-PROJECT