# Sequence to Sequence Learning with Neural Networks

## Report

**Likhith Sai Chaitanya Vadlapatla, Bhuvaneswari Vallagachu, Varaprasad Bonthu**

### Introduction:

A Question Answering system is an artificial intelligence system that provides clear responses to their inquiries. It analyses user questions using natural language processing, machine learning, and other cutting-edge technologies before retrieving data from a database or the internet and presenting the solution in a way that is understandable to humans. By reducing time and effort needed to retrieve information, these systems employ variety of applications. Question-and-answer systems have evolved into a crucial tool for effective communication in the modern world, when we have access to a great amount of information.

### Methodology:

Based on the user's needs, this question-and-answer system will offer a means to extract useful information from a context. It functions as described in the following steps:

1. To begin with, our system has a question processing component that reads questions and extracts keywords from them.

2. After retrieving the text input, a corpus processing component searches for the answer in the context.

3. Finally, output is retrieved for the user-posted mandatory query from the context.

### Approach:

The steps that we are going to follow in our project are as mentioned below:

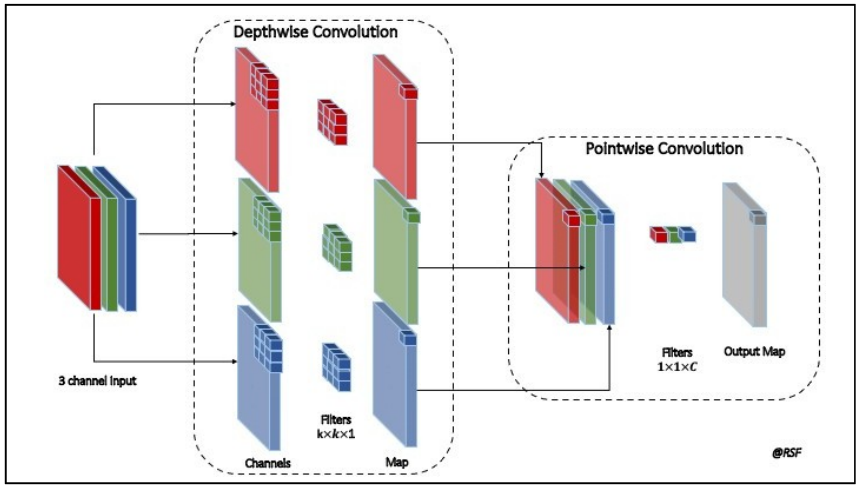
1. Web scrapping the data
2. Creating Custom Dataset and Dataloader
3. Token generation using Regular Expressions
4. Generating Vocabulary
5. Generating Dictionary
6. Creating a Mini Network from Scratch
7. Defining Hyper Parameters
8. Training the model
9. Model Evaluation
10. Transfer Learning

**Algorithms:**

***Depthwise separable convolution***

The main difference between depthwise separable convolutions and regular convolutions is that depthwise separable convolutions are quicker since they need fewer multiplication operations. To do this, the convolution procedure is split into the depthwise convolution and the pointwise convolution processes.

Contrary to normal CNNs, where convolution is performed to all M channels at once, depth-wise operation only applies convolution to one channel at a time. The filters/kernels used here will thus be Dk x Dk x 1. Given that the input data has M channels, M such filters are necessary. The output will be Dp x Dp x M in size.




***Highway Networks***

Highway networks were first developed to make deep neural network training simpler. Even though research had discovered the secret to improving shallow neural networks, training deep networks remained difficult because of issues like disappearing gradients, etc. The structure that has been covered so far is a common pattern in NLP systems. Even though big pretrained language models and transformers may have made this pattern obsolete, many NLP systems used it prior to the invention of transformers. The reasoning for this is that by including highway layers, the network can utilize character embeddings more effectively. A word will probably be initialized if it is not included in the pre-trained word vector vocabulary (OOV word).

***Embedding Layer***

This layer:

- converts word-level tokens into a 300-dim pre-trained glove embedding vector
- creates trainable character embeddings using 2-D convolutions
- concatenates character and word embeddings and passes them through a highway network

***Multi-Headed Self Attention***

The fundamental principle of attention is that we compute the similarity between two representations, transform them into an attention distribution, and then add the values in a weighted manner. But there are some specifics involved that must be addressed.

***Positional Embedding***

The model has no concept of how words are arranged in a phrase as of yet. Because we were utilizing RNNs or LSTMs to encode this information at some point in earlier models, this was handled. RNNs preserve hidden states at each place in the input sequence as they process input in a sequential manner. But in this case, we need a way to introduce the positional information of tokens into the model.

***Encoder Block***

The following steps are performed by this layer:

- A positional embedding is injected into the input.
- This is subsequently run through several convolutional layers. The layer that these encoder blocks are a part of determines how many of these levels there are. This value is 4 for the embedding encoder layer and 2 for the model encoder layer. Convolutional layer definitions.
- After that, the output is sent to a feedforward network, which is just a linear layer, and lastly to a multi-headed self-attention layer.
- An easy way to understand the residual connections in code would be to draw 2-3 iterations of the lower block (that involves convolution) and ensure that everything matches.

***Context-Query Attention Layer***

It divides attention into two halves. The most relevant query terms for each context word are revealed via context-query attention.

***Output Layer***

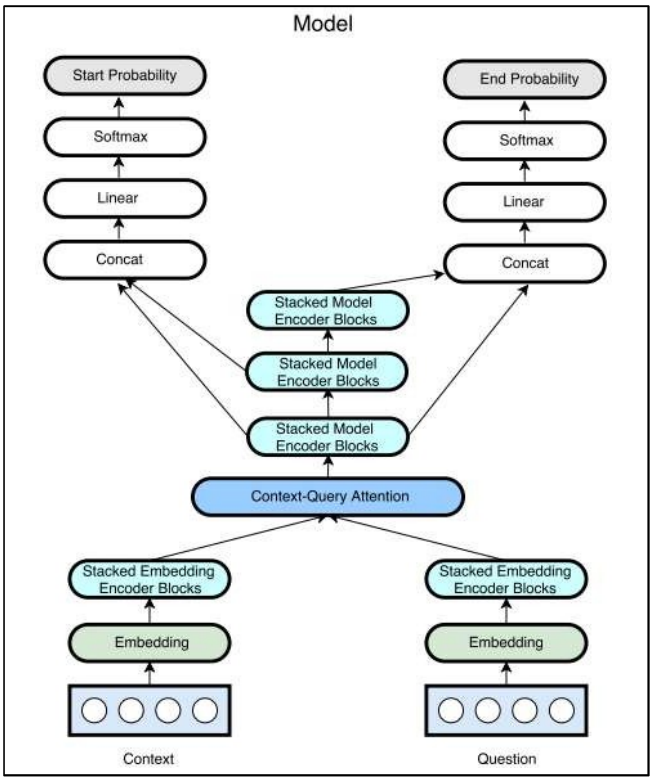
Predicting the start and end indices of the answer from the context is the responsibility of the output layer.

***QANet***

This brings all the components mentioned above together:

- The word-level and character-level tokens for the context and the question are the inputs to the forward procedure. Here, the embedding layer receives these tokens.
- The embedded encoding layer, which consists of a single encoder block with four convolution layers, receives the downsized tensors next. In the self-attention module, which is the same for all the encoder blocks in the model, 8 attention heads are employed.
- The Context-Query Attention Layer is then given the results of the preceding layer.
- The model encoder layer receives the encoded representation after that, and the output layer, which determines the start and end indexes of the response, receives the shared-weight matrices last.

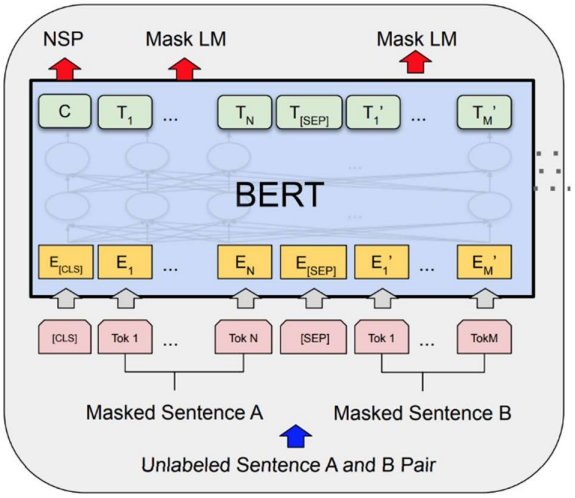
## Mini Network Architecture




The QANet architecture uses a combination of convolutional neural networks (CNNs) and self-attention mechanisms to extract information from the input passage and the question. The CNNs capture local features of the input, while the self-attention mechanisms allow the model to attend to different parts of the input at different levels of granularity.

QANet is composed of several layers, including embedding layers, convolutional layers, self-attention layers, and output layers. The embedding layers transform the input text into a sequence of vectors, which are then fed into the convolutional layers to extract local features. The self-attention layers are used to compute the global representation of the input 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



A deep neural network architecture called BERT (Bidirectional Encoder Representations from Transformers) is intended to handle a range of natural language processing (NLP) challenges, including language comprehension, sentiment analysis, and question-answering. Being a bidirectional model, it can comprehend the context of a word depending on both the words that come before and after it in a phrase. This is accomplished via a method known as masked language modeling, where part of the input tokens are arbitrarily hidden, and the model is trained to anticipate the hidden characters based on the input sequence's remaining tokens.

BERT is built on a neural network known as the Transformer architecture, which employs self-attention techniques to record the relationships between various portions of the input sequence. By adding task-specific layers on top of the pre-trained model and then training the entire model on the task-specific dataset, the pre-trained BERT model may be fine-tuned on a particular downstream NLP job. On a number of NLP tasks, including sentiment analysis, named entity identification, and question answering, fine-tuning BERT has demonstrated to achieve state-of-the-art performance.

## Implementation

### *Task1- Web scrapping the data*

The context for our project is collected from Yale university website. A context was chosen initially and was web scrapped using Beautiful Soup Library. This context is later used to train our model from which all the questions can be asked and the answers would be generated.

Context link: https://cpsc.yale.edu/news/memoriam-dragomir-radev-professor-computer-science

```
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 created custom dataset class called ChatBot using PyTorch. Datasets are loaded using this custom dataset class and generated a vocabulary list using the regular expressions. This is shown 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__()
```

```
training_dataset.__len__()
```

```
4120
```

```
testing_dataset = ChatBot(test_Context)
testing_dataset.__getitem__()
```

```
"Yale offers advanced degrees through its Graduate School of Arts & Sciences and 13 professional schools. \nBrowse the organiza
tions below for information on programs of study, academic requirements, and faculty research.\nYale's Graduate School of Arts
& Sciences offers programs leading to M.A., M.S., M.Phil., and Ph.D. degrees in 73 departments and programs.\nIn graduate schoo
l, you will transition from being your professors' student to becoming their colleague. If this prospect excites you, please co
nsider applying to Yale. \nFew places in the world can serve as a better incubator for your continued learning, growth, and pot
ential as a scholar, independent researcher, and leader.\n"
```

```
testing_dataset.__len__()
```

```
687
```

```
training_data = DataLoader(training_dataset, batch_size = 8)
testing_data = DataLoader(testing_dataset, batch_size = 8)
```

***Task 3- Token generation using Regular Expressions.***

Next, we generated tokens using the Regular Expressions. The code is mentioned below as

```
pattern = r'''(?x)
\d+\,\d+
| [A-Z]\.[A-Z]\.
| [A-Z][a-z]*\.\s\d?\d\,\s\d+\s\d?\d\:\d+\s[ap]m\sET
| \$?\d+\%?
| Model[s]?\s\d?[A-Z]?
| \w+\-\w+
| \w+
| [n]?'\w+
| [][.,;"?():-_]'''
re.findall(pattern, training_dataset.__getitem__())
```

***Task 4- Generation of Vocabulary.***

Next, we generated vocabulary and count the frequency. The code is mentiond below as

```
list(set(re.findall(pattern,training_dataset.__getitem__())))
```

```
['taught',
 'varied',
 'celebrated',
 'popular',
 'survived',
 'Accessibility',
 'Medicine',
 'numerous',
 '2022',
 'North',
 'with',
 'addition',
 'mazes',
 'deeply',
 'online',
 'For',
 'School',
 'L',
 'was',
```

**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.*

Next, we generate dictionary from the obtained 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),
```

### *Task 6- Creating a Mini Network from Scratch*

```
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):

            flow = self.flow_layers[i](x)
            gate = torch.sigmoid(self.gate_layers[i](x))

            x = gate * flow + (1 - gate) * x

        print("Highway layer: ", "TRAINED")
        return x
```

```
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()
```

```
def create_weights_matrix(glove_dict):
    weights_matrix = np.zeros((len(vocab_list), 300))
    words_found = 0
    for i, word in enumerate(vocab_list):
        try:
            weights_matrix[i] = glove_dict[word]
            words_found += 1
        except:
            pass

    return weights_matrix, words_found
```

```
weights_matrix, words_found = create_weights_matrix(glove_dict)
print("Words found in the vocab: " ,words_found)
```

```
Words found in the vocab:  342
```

```
np.save('qanetglove.npy', weights_matrix)
```

```
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
```

```
from torch.autograd import Variable
import math
class PositionEncoder(nn.Module):

    def __init__(self, model_dim, device, max_length=2592):

        super().__init__()

        self.device = device

        self.model_dim = model_dim

        pos_encoding = torch.zeros(max_length, model_dim)

        for pos in range(max_length):

            for i in range(0, model_dim, 2):

                pos_encoding[pos, i] = math.sin(pos / (10000 ** ((2*i)/model_dim)))
                pos_encoding[pos, i+1] = math.cos(pos / (10000 ** ((2*(i+1))/model_dim)))


        pos_encoding = pos_encoding.unsqueeze(0).to(device)
        self.register_buffer('pos_encoding', pos_encoding)


    def forward(self, x):
        x = x + Variable(self.pos_encoding[:, :x.shape[1]], requires_grad=False)
        return x
```

```
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
```

```
class ContextQueryAttentionLayer(nn.Module):

    def __init__(self, model_dim):

        super().__init__()

        self.W0 = nn.Linear(3*model_dim, 1, bias=False)

    def forward(self, C, Q, c_mask, q_mask):

        c_mask = c_mask.unsqueeze(2)

        q_mask = q_mask.unsqueeze(1)

        ctx_len = C.shape[1]
        qtn_len = Q.shape[1]

        C_ = C.unsqueeze(2).repeat(1,1,qtn_len,1)

        Q_ = Q.unsqueeze(1).repeat(1,ctx_len,1,1)

        C_elemwise_Q = torch.mul(C_, Q_)

        S = torch.cat([C_, Q_, C_elemwise_Q], dim=3)

        S = self.W0(S).squeeze()

        S_row = S.masked_fill(q_mask==1, -1e10)
        S_row = F.softmax(S_row, dim=2)

        S_col = S.masked_fill(c_mask==1, -1e10)
        S_col = F.softmax(S_col, dim=1)
```

```
        A = torch.bmm(S_row, Q)

        B = torch.bmm(torch.bmm(S_row,S_col.transpose(1,2)), C)

        model_out = torch.cat([C, A, torch.mul(C,A), torch.mul(C,B)], dim=2)

        print("C2Q layer: ","TRAINED")
        return F.dropout(model_out, p=0.1)
```

```
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
```

From the above codes, we have defined different classes for **DepthwiseSeparableConvolution, HighwayLayer, get_glove_dict, create_weights_matrix,EmbeddingLayer, MultiheadAttentionLayer, PositionEncoder, EncoderBlock, ContextQueryAttentionLayer, OutputLayer**. All these classes are later integrated into the **QANET** class to get the final results.

***Task 7- Defining Hyper Parameters***

```
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)

```

Next, we define few hyperparameters 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)
```

Next, we train the model with all the parameters and hyperparameters defined.

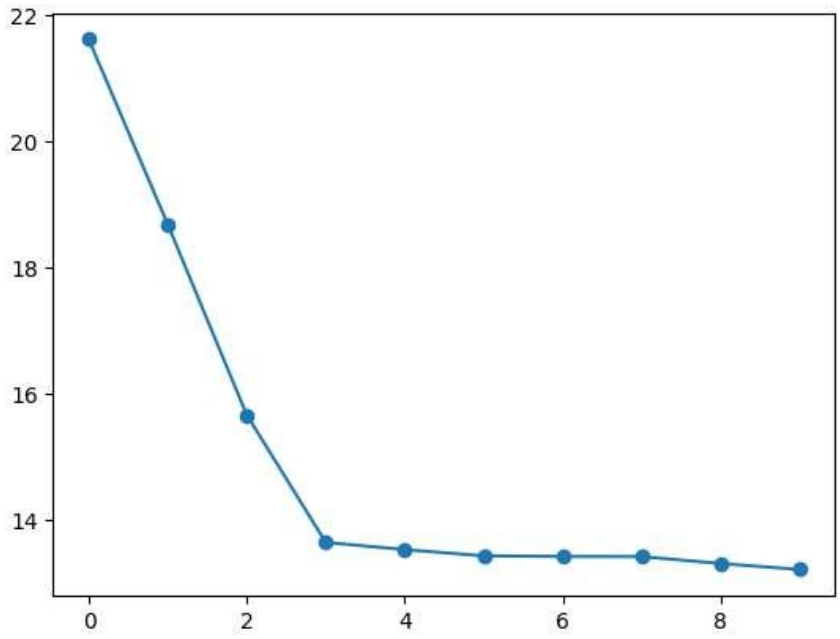
***Task 9- Model Evaluation***

```
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
```

When we evaluate our model with the test dataset, we get an accuracy of **50%.** When trained for **10** epochs, we get the minimum loss and the model converges after **3** epochs.

***Task 10- Transfer Learning***

```
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__())
```

```
answers=[]
print("Hi I'm ChatBot I'm Here to answer your Questions \n")
Question = input("Enter your question for me (Enter quit to exit): ")
Counter = 0
while Counter < 10:
  if Question == "quit":
    print("\nThanks for chatting with me. See you soon with more Questions!!!")
    break
  else:
    _, _ , _ , _, ans  = bert_ChatBot(Question, training_dataset.__getitem__())
    print('\nChatBot Answer: ', ans ,  '\n')
    Question = input("Enter your question for me (Enter quit to exit): ")
    answers.append(ans)
```

We performed transfer learning to train the chatbot. The model used here is '***BertForQuestionAnswering***' and tokenized using ***'tokenizer_for_bert'***. We are also giving few predefined syntaxes for the chatbot to show up when the user begins to ask questions.

## Conclusion & Results:

```
Hi I'm ChatBot I'm Here to answer your Questions

Enter your question for me (Enter quit to exit): What is the name of the university?

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:  university of michigan

Enter your question for me (Enter quit to exit): Name of the professor?

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:  dragomir radev

Enter your question for me (Enter quit to exit): Which department was handled by dragoman Radev?

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:  computer science

Enter your question for me (Enter quit to exit): Breaking news in India?

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!!!
```

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

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

We observe that the accuracy of BERT is **75 %** and that of our mininetwork is 50%. This gap can be covered by changing the hyperparameters of the mini newtwork. This model answers most of the questions that is defined in the context. Similarly, multiple questions can be asked and the answers can be retrieved.

## References:


1. Yu, A.W., Dohan, D., Luong, M.T., Zhao, R., Chen, K., Norouzi, M. and Le, Q.V., 2018. Qanet: Combining local convolution with global self-attention for reading comprehension. *arXiv preprint arXiv:1804.09541*.
2. Sutskever, I., Vinyals, O. and Le, Q.V., 2014. Sequence to sequence learning with neural networks. Advances in neural information processing systems, 27
3. Khin, N.N. and Soe, K.M., 2020, November. Question Answering based University Chatbot using Sequence to Sequence Model. In 2020 23rd Conference of the Oriental COCOSDA International Committee for the Co-ordination and Standardisation of Speech Databases and Assessment Techniques (O-COCOSDA) (pp. 55-59). IEEE.
4. Le, A.C. and Huynh, V.N., 2022, October. Towards a Human-like Chatbot using Deep Adversarial Learning. In 2022 14th International Conference on Knowledge and Systems Engineering (KSE) (pp. 1-5). IEEE
5. N. Kalchbrenner and P. Blunsom. Recurrent continuous translation models. In EMNLP, 2013.
6. K. Cho, B. Merrienboer, C. Gulcehre, F. Bougares, H.Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In Arxiv preprint arXiv:1406.1078,2014.
7. D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. In arXiv preprint arXiv:1409.0473, 2014.