



**JHANVI PAREKH**

**60009210033**

### **Experiment No 9**

**Aim:** To fine-tune a BERT model on custom data for performing a question-answering task.

#### **Introduction:**

Question Answering (QA) systems aim to provide precise answers to questions posed in natural language. Fine-tuning BERT (Bidirectional Encoder Representations from Transformers) on custom datasets allows for developing QA models tailored to specific domains. By training on relevant question-answer pairs, the model learns to locate and provide precise answers based on context.

BERT is especially effective for this task due to its ability to understand contextual embedding's, making it well-suited for handling the complexities of language. Fine-tuning involves training the model on a labeled dataset where each entry includes a passage, a question, and the corresponding answer span within the passage.

#### **Fine-Tuning BERT for Question Answering**

1. **Model Selection:** Use the pre-trained BERT model as the base, available through Hugging Face's Transformers library. Models such as bert-large-uncased are commonly used for QA tasks.
2. **Data Preparation:** Format the custom data into question-passage-answer triplets. The passage contains the context, and the answer is a span within this context.
3. **Tokenization:** Tokenize the data using the BERT tokenizer, which splits text into tokens compatible with BERT's vocabulary. Additionally, align each token with its position in the original text to aid in identifying the answer span.
4. **Fine-Tuning Process:** Train the BERT model on the formatted dataset, optimizing it to locate the start and end tokens of the answer span within the context passage.

#### **Lab Experiment**

**Step 1:** Install required libraries.

**Step 2:** Load a pre-trained BERT model for QA and tokenizer.

**Step 3:** Prepare the custom data in question-passage-answer format, ensuring each answer is marked with its start and end positions in the passage.

**Step 4:** Tokenize the data and format it for BERT.



Department of Computer Science and Engineering (Data Science)  
Lab Manual

Sub: Advanced Computational Linguistics

Year/Sem: BTech/VII

**Step 5:** Fine-tune the model.

**Step 6:** Test the model with new questions and passages.

Link:

<https://colab.research.google.com/drive/17BYqWIPkU82cjDPiBc4IvQqOXtnBVsr?usp=sharing>

The image displays two screenshots of a Google Colab notebook interface. The top screenshot shows the initial code setup, including imports for torch, transformers, and torch.optim, followed by the definition of a custom QADataset class. The bottom screenshot shows the continuation of the code, where the dataset is iterated over, and the question and passage are tokenized using a BertTokenizer, with start and end positions for the question being recorded.

```
import torch
from torch.utils.data import Dataset, DataLoader
from transformers import BertTokenizer, BertForQuestionAnswering
from torch.optim import AdamW

[ ]

# Define the custom Dataset class
class QADataset(Dataset):
    def __init__(self, data, tokenizer, max_length=512):
        self.data = data
        self.tokenizer = tokenizer
        self.max_length = max_length

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        item = self.data[idx]
        question = item['question']
        passage = item['passage']
        start_position = item['start_position']
        end_position = item['end_position']

question = item['question']
passage = item['passage']
start_position = item['start_position']
end_position = item['end_position']

# Tokenize question and passage
inputs = self.tokenizer(
    question,
    passage,
    max_length=self.max_length,
    padding='max_length',
    truncation=True,
    return_tensors='pt'
)

# Add start and end positions
inputs['start_positions'] = torch.tensor(start_position)
inputs['end_positions'] = torch.tensor(end_position)

return {
    'input_ids': inputs['input_ids'].squeeze(),
    'attention_mask': inputs['attention_mask'].squeeze(),
    'start_positions': inputs['start_positions'],
    'end_positions': inputs['end_positions']
}
```



Department of Computer Science and Engineering (Data Science)  
Lab Manual

Sub: Advanced Computational Linguistics

Year/Sem: BTech/VII

```
[ ] # Initialize the tokenizer
tokenizer = BertTokenizer.from_pretrained('bert-large-uncased-whole-word-masking-finetuned-squad')

[ ] # Example data (replace this with your actual dataset)
data = [
    {
        'question': "What is the capital of France?",
        'passage': "France's capital is Paris, which is known for its art, fashion, and culture.",
        'answer_text': "Paris",
        'start_position': 16,
        'end_position': 20
    },
    # Add more examples here
]

# Create the dataset and data loader
qa_dataset = QADataset(data, tokenizer)
data_loader = DataLoader(qa_dataset, batch_size=8, shuffle=True)

[ ] # Initialize the model for question answering
model = BertForQuestionAnswering.from_pretrained('bert-large-uncased-whole-word-masking-finetuned-squad')
```

Some weights of the model checkpoint at bert-large-uncased-whole-word-masking-finetuned-squad were not used when initializing BertForQuestionAnswering - This IS expected if you are initializing BertForQuestionAnswering from the checkpoint of a model trained on another task or with another architecture - This IS NOT expected if you are initializing BertForQuestionAnswering from the checkpoint of a model that you expect to be exactly identical (initialization)

```
[ ] # Set up the optimizer
optimizer = AdamW(model.parameters(), lr=5e-5)

# Check for GPU availability
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

BertForQuestionAnswering(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(30522, 1024, padding_idx=0)
      (position_embeddings): Embedding(512, 1024)
      (token_type_embeddings): Embedding(2, 1024)
      (LayerNorm): LayerNorm((1024,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
```



Department of Computer Science and Engineering (Data Science)  
Lab Manual

Sub: Advanced Computational Linguistics

Year/Sem: BTech/VII

```
(dropout): Dropout(p=0.1, inplace=False)
)
(encoder): BertEncoder(
  (layer): ModuleList(
    (0-23): 24 x BertLayer(
      (attention): BertAttention(
        (self): BertSdpaSelfAttention(
          (query): Linear(in_features=1024, out_features=1024, bias=True)
          (key): Linear(in_features=1024, out_features=1024, bias=True)
          (value): Linear(in_features=1024, out_features=1024, bias=True)
          (dropout): Dropout(p=0.1, inplace=False)
        )
        (output): BertSelfOutput(
          (dense): Linear(in_features=1024, out_features=1024, bias=True)
          (LayerNorm): LayerNorm((1024,), eps=1e-12, elementwise_affine=True)
          (dropout): Dropout(p=0.1, inplace=False)
        )
      )
      (intermediate): BertIntermediate(
        (dense): Linear(in_features=1024, out_features=4096, bias=True)
        (intermediate_act_fn): GELUActivation()
      )
      (output): BertOutput(
        (dense): Linear(in_features=4096, out_features=1024, bias=True)
        (LayerNorm): LayerNorm((1024,), eps=1e-12, elementwise_affine=True)
        (dropout): Dropout(p=0.1, inplace=False)
      )
    )
  )
)

# Training loop
model.train()
for epoch in range(3): # number of epochs
    for batch in data_loader:
        # Extract inputs and labels from batch
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        start_positions = batch['start_positions'].to(device)
        end_positions = batch['end_positions'].to(device)

        # Forward pass
        outputs = model(input_ids=input_ids,
                        attention_mask=attention_mask,
                        start_positions=start_positions,
                        end_positions=end_positions)
        loss = outputs.loss

        # Backward pass and optimization
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

    print(f"Epoch {epoch + 1} loss: {loss.item()}")
```



Department of Computer Science and Engineering (Data Science)  
Lab Manual

Sub: Advanced Computational Linguistics

Year/Sem: BTech/VII

```
Epoch 1 loss: 11.197107315063477
Epoch 2 loss: 3.319542646408081
Epoch 3 loss: 2.353126049041748

[ ] # After training, save the fine-tuned model
model.save_pretrained('./fine_tuned_bert_qa')
tokenizer.save_pretrained('./fine_tuned_bert_qa')

('./fine_tuned_bert_qa/tokenizer_config.json',
 './fine_tuned_bert_qa/special_tokens_map.json',
 './fine_tuned_bert_qa/vocab.txt',
 './fine_tuned_bert_qa/added_tokens.json')

[ ] # Optionally: Evaluate the model on some test questions
# Test the model with a question-passage pair
test_question = "What is the capital of France?"
test_passage = "France's capital is Paris, which is known for its art, fashion, and culture."

[ ] # Tokenize the test question and passage
inputs = tokenizer(test_question, test_passage, return_tensors='pt').to(device)

[ ] # Get the model's output

# Tokenize the test question and passage
inputs = tokenizer(test_question, test_passage, return_tensors='pt').to(device)

[ ] # Get the model's output
outputs = model(**inputs)

[ ] # Get the predicted answer's start and end positions
start_idx = torch.argmax(outputs.start_logits)
end_idx = torch.argmax(outputs.end_logits)

[ ] # Convert token indices to text
answer_tokens = inputs['input_ids'][0][start_idx:end_idx+1]
answer = tokenizer.decode(answer_tokens, skip_special_tokens=True)

print(f"Predicted answer: {answer}")

Predicted answer: which is known for its
```