

Deep Learning Report of Computer Assignment

Iman Alizadeh Fakouri
Student Number: 401102134
Instructor: Dr. Emad Fatemizadeh

December 28, 2024

Contents

1	Introduction	3
2	Part 1 & 2: Model Loading and Text Generation	3
2.1	Key Configuration Parameters in Text Generation	5
3	Part 3: Loading and Tokenizing SST-2 Dataset	6
3.1	Explanation	6
4	Part 4 & 5: Removing the Final Layer and Adding a Linear Layer for Classification	7
4.1	General Explanation	7
4.2	General Idea of the Code	7
4.3	Code Implementation	8
4.3.1	Code Breakdown	9
4.4	Metrics and Evaluation	9
4.4.1	Metrics Breakdown	10
4.5	Training Setup	10
4.5.1	Training Setup Breakdown	11
4.6	Confusion Matrix and Visualizations	12
4.6.1	Confusion Matrix Breakdown	12
4.7	Training Logs and Loss/Accuracy Plots	13
4.7.1	Loss/Accuracy Breakdown	13
5	Part 6: Bidirectional Attention Layer for Classification	16
5.1	Bidirectional Attention Layer	16
5.2	Custom GPT-2 Model	16
5.3	Training Configuration	17
6	Part 7: Using Left-to-Right and Right-to-Left Unidirectional Attention	20
6.1	General Explanation	20
6.2	Code Explanation	20
6.3	Explanation of Code Differences	22
6.4	Training	23
7	Part 8: Zero-shot Performance with Pre-trained BERT Model	25
7.1	General Explanation	25
7.2	Code Explanation	28
7.3	Explanation of the Code	29

1 Introduction

Large Language Models (LLMs) are a class of deep learning models designed for processing and generating natural language. These models are trained using large amounts of textual data and utilize architectures based on transformers. Applications of LLMs include text generation, machine translation, summarization, and sentiment analysis. This report explores the use of LLMs in a binary sentiment classification task. We modify a decoder-only generative model into an encoder and evaluate its performance. Additionally, we investigate configurations like temperature, top-k, top-p, and repetition penalties that affect text generation. The SST-2 dataset is employed for experiments, and results with models like T5 and GPT are analyzed.

2 Part 1 & 2: Model Loading and Text Generation

In this section, we demonstrate the process of importing a decoder-only model (GPT-2) and loading the weights of a pre-trained version of the model. We then generate several outputs from the model using different input texts and key configuration parameters.

First, we import the necessary libraries and load the GPT-2 model and tokenizer:

```
1 import torch
2 from transformers import GPT2LMHeadModel, GPT2Tokenizer
3
4 model_name = "gpt2"
5 model = GPT2LMHeadModel.from_pretrained(model_name)
6 tokenizer = GPT2Tokenizer.from_pretrained(model_name)
7 device = torch.device("cuda" if torch.cuda.is_available()
8                       else "cpu")
9 model.to(device)
```

Next, we define a function `generateText` that uses various configuration parameters to generate text:

```
1 def generate_text(input_text, max_length=50, temperature=1.0,
2                  top_k=50, top_p=0.95,
3                  repetition_penalty=1.0, num_beams=1,
4                  no_repeat_ngram_size=2):
5     input_ids = tokenizer.encode(input_text, return_tensors='pt').to(device)
6     attention_mask = torch.ones(input_ids.shape, device=device)
```

```

5     pad_token_id = tokenizer.eos_token_id
6     output = model.generate(
7         input_ids=input_ids,
8         attention_mask=attention_mask,
9         max_length=max_length,
10        temperature=temperature,
11        top_k=top_k,
12        top_p=top_p,
13        repetition_penalty=repetition_penalty,
14        num_beams=num_beams,
15        no_repeat_ngram_size=no_repeat_ngram_size,
16        pad_token_id=pad_token_id,
17        eos_token_id=tokenizer.eos_token_id
18    )
19
20    generated_text = tokenizer.decode(output[0],
21        skip_special_tokens=True)
22    return generated_text

```

We then generate outputs for a set of input texts and print the results:

```

1 inputs = [
2     "The future of artificial intelligence is",
3     "The economy is changing due to",
4     "In the year 2050, people will",
5     "Climate change is one of the most",
6     "The role of technology in education is",
7     "In a distant galaxy, there was",
8     "The internet of things is transforming",
9     "The impact of social media on society is",
10    "Self-driving cars are expected to",
11    "The rise of renewable energy is"
12 ]
13
14 generated_outputs = [generate_text(input_text=input_text,
15     max_length=50) for input_text in inputs]
16 for i, output in enumerate(generated_outputs):
17     print(f"Input {i+1}: {inputs[i]}")
18     print(f"Generated Output {i+1}: {output}\n")

```

Generated Outputs:

Here are some sample outputs generated from the input prompts:

- **Input 1:** The future of artificial intelligence is
Generated Output 1: The future of artificial intelligence is uncertain. "We're not sure what the future will look like," said Dr. Michael S. Schoenfeld, a professor of computer science at the University of California, Berkeley. "But we're very..."

- **Input 2:** The economy is changing due to
Generated Output 2: The economy is changing due to the rise of the internet, and the fact that it is becoming more and more difficult to find jobs. The government has been trying to get the economy back on track, but it has not been able to do...
- **Input 3:** In the year 2050, people will
Generated Output 3: In the year 2050, people will be able to buy a home in the United States, and the average home price will rise by 1,000. *The average American household will have a median income of 50,400,* according to...
- And so on for the other inputs.

2.1 Key Configuration Parameters in Text Generation

The following key configuration parameters were used in the text generation process:

- **Temperature:** Controls the randomness of the output. Higher values (e.g., 1.5) make the output more diverse, while lower values (e.g., 0.7) make the output more focused and deterministic.
- **topk:** Limits the number of candidate tokens to the top k most likely tokens. For instance, topk=50 restricts token selection to the top 50.
- **topp** (Nucleus Sampling): Selects the smallest set of tokens whose cumulative probability exceeds a threshold p. For example, topp=0.95 means the model selects the smallest set of tokens whose cumulative probability is greater than 95
- **Repetition Penalty:** Penalizes repetitive sequences by assigning a higher penalty for repeated tokens. Values greater than 1.0 discourage repetition.
- **numbeams:** Controls the beam search algorithm. A higher number of beams improves output coherence but reduces randomness.
- **norepeatngramsize:** Prevents the model from repeating n-grams (such as bigrams). For example, setting this value to 2 avoids repeating any bigrams in the generated output.

These parameters allow fine control over the generation process, enabling the user to balance between randomness and coherence, as well as prevent undesirable repetitions.

3 Part 3: Loading and Tokenizing SST-2 Dataset

In this part, we load the SST-2 dataset, which is part of the GLUE benchmark, for sentiment classification. We apply necessary padding to the dataset to ensure parallel execution of the model.

First, we import the necessary libraries for tokenization and dataset loading:

```
1 from transformers import GPT2Tokenizer, GPT2LMHeadModel,
   GPT2Model
2 import torch
3 from torch import nn
4 from datasets import load_dataset
5
6 device = torch.device("cpu")
7 tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
```

Next, we load the SST-2 dataset from the GLUE benchmark:

```
1 dataset = load_dataset("glue", "sst2")
```

We define a function to tokenize the dataset and pad it accordingly for parallel execution:

```
1 def tokenize_function(examples):
2     tokenizer.pad_token = tokenizer.eos_token
3     return tokenizer(examples["sentence"])
4
5 tokenized_datasets = dataset.map(tokenize_function, batched=
   True)
```

3.1 Explanation

In this section, we load the SST-2 dataset, which is used for sentiment classification. The tokenizer is applied to the dataset to ensure that each sentence is properly tokenized. The padding token is set to the 'eos_token' for GPT-2, ensuring consistency with the model's input requirements. After tokenization, the dataset is processed in batches to allow for parallel execution, which speeds up the training or evaluation process.

This prepares the dataset for efficient use in further processing or model training.

4 Part 4 & 5: Removing the Final Layer and Adding a Linear Layer for Classification

4.1 General Explanation

In these steps, we modify the GPT-2 model to perform sentiment classification on the SST-2 dataset (a binary classification task from the GLUE benchmark). The goal is to adapt GPT-2 for classification by removing the final layer and adding a custom linear layer for classification purposes. Here's a more detailed breakdown of the process:

- **Removing the final layer:** The last layer in GPT-2 outputs logits corresponding to the model's vocabulary. In a classification task, we don't need to predict the next token; instead, we need a class label (positive or negative). Hence, we remove the model's final layer.
- **Using the CLS token embedding:** The embedding vector of the first token (often called the CLS token) in the input sequence is typically used for classification tasks. This vector is a representation of the entire input sequence.
- **Adding a Linear Layer:** Since the CLS token embedding may not always provide the best representation of the entire input, we add a linear layer on top of the encoder's output. This additional layer helps aggregate information from multiple tokens and provides a more robust representation for classification.
- **Training and Evaluation:** After modifying the model, we set up the training pipeline, compute metrics, and evaluate the model using accuracy, precision, recall, and F1 score.

4.2 General Idea of the Code

The following steps describe how the code implements the changes for sentiment classification:

- **Custom Model Definition:** We define a custom class `gpt_model` that inherits from `nn.Module`. This class modifies the GPT-2 model by removing the final layer and adding a linear layer for classification.
- **Forward Pass:** During the forward pass, the model produces hidden states, which are the internal representations of the input sequence.

We use the hidden states of the final layer to generate logits for classification. These logits are then passed through a linear layer to make the final classification.

- **Metrics Calculation:** We calculate accuracy, precision, recall, and F1 score using the `compute_metrics` function.
- **Trainer Setup:** The Hugging Face `Trainer` class is used for managing the training and evaluation processes, including metrics computation and saving model checkpoints.

4.3 Code Implementation

```
1 import torch.nn as nn
2 from transformers import GPT2LMHeadModel
3 import torch
4
5 device = torch.device("cuda")
6
7 class gpt_model(nn.Module):
8     def __init__(self):
9         super(gpt_model, self).__init__()
10        # Load pre-trained GPT-2 model
11        self.gpt2 = GPT2LMHeadModel.from_pretrained("gpt2")
12        # Disable weight tying between word embeddings and
13        # output layer
14        self.gpt2.config.tie_word_embeddings = False
15        # Add a new linear layer for binary classification (
16        # positive/negative)
17        self.linear = nn.Linear(self.gpt2.config.hidden_size,
18                                2)
19
20    def forward(self, input_ids, attention_mask=None, labels=
21    None):
22        # Get the outputs from GPT-2, including hidden states
23        outputs = self.gpt2(input_ids=input_ids,
24                             attention_mask=attention_mask,
25                             output_hidden_states=True)
26        # Extract the last hidden states (final layer's
27        # output)
28        hidden_states = outputs.hidden_states[-1]
29        # Pass the hidden states through the linear layer for
30        # classification
31        logits = self.linear(hidden_states)
32        loss = None
33        # If labels are provided, calculate loss (Cross-
34        # Entropy Loss for classification)
```



```

26         if labels is not None:
27             logits = logits.mean(dim=1) # Average over the
                sequence length
28             loss_fn = nn.CrossEntropyLoss()
29             loss = loss_fn(logits, labels)
30             # Return the logits (and loss if labels are given)
31             return {"loss": loss, "logits": logits} if labels is
                not None else {"logits": logits}
32
33 model = gpt_model().to(device)

```

Listing 1: Custom GPT-2 Model for Sentiment Classification

4.3.1 Code Breakdown

- **Model Definition:**

- We first initialize the GPT-2 model using `GPT2LMHeadModel.from_pretrained("gpt2")`. This loads the pre-trained GPT-2 model.
- We disable the weight tying between the word embeddings and the output layer by setting `self.gpt2.config.tie_word_embeddings = False`. This ensures the embeddings and output layers are independent.
- A new `Linear` layer is added, which takes the output hidden size of the GPT-2 model and produces a 2-dimensional output for binary classification (positive/negative).

- **Forward Pass:**

- During the forward pass, the input IDs and attention masks are passed through the GPT-2 model. The model produces hidden states, which are internal representations of the input text.
- The final layer's hidden states are extracted, and the linear layer is applied to obtain the logits (the raw predictions for each class).
- If labels are provided (during training), we calculate the loss using Cross-Entropy Loss for binary classification. We use `logits.mean(dim=1)` to average the logits across the sequence tokens (since we are using the CLS token's representation).

4.4 Metrics and Evaluation

The evaluation metrics are crucial for understanding the model's performance. Here's the code for computing various metrics:

```

1 from sklearn.metrics import accuracy_score,
   precision_recall_fscore_support
2 import numpy as np
3
4 def compute_metrics(pred):
5     logits, labels = pred
6     predictions = np.argmax(logits, axis=-1) # Get the
       predicted class (0 or 1)
7     accuracy = accuracy_score(labels, predictions) # Compute
       accuracy
8     precision, recall, f1, _ =
       precision_recall_fscore_support(labels, predictions,
       average="binary")
9     return {
10         "accuracy": accuracy,
11         "precision": precision,
12         "recall": recall,
13         "f1": f1,
14     }

```

Listing 2: Evaluation Metrics Function

4.4.1 Metrics Breakdown

- The `compute_metrics` function calculates several evaluation metrics:
 - **accuracy**: The percentage of correct predictions.
 - **precision**: The proportion of positive predictions that are actually correct.
 - **recall**: The proportion of actual positives that were correctly predicted.
 - **f1**: The harmonic mean of precision and recall, providing a balanced metric.
- `pred` is the output from the model, which contains both logits (predictions) and labels (true values). We use `np.argmax(logits, axis=-1)` to convert logits into predicted class labels.

4.5 Training Setup

The training setup defines the training arguments and the configuration for the training loop:

```

1 from transformers import Trainer, TrainingArguments,
   DataCollatorWithPadding
2
3 training_args = TrainingArguments(
4     output_dir='./results',
5     num_train_epochs=3,
6     per_device_train_batch_size=16,
7     per_device_eval_batch_size=64,
8     warmup_steps=500,
9     weight_decay=0.01,
10    logging_dir='./logs',
11    save_strategy="no",
12    save_total_limit=0,
13    evaluation_strategy="epoch"
14 )
15
16 data_collator = DataCollatorWithPadding(tokenizer=tokenizer,
17    pad_to_multiple_of=8)
18
19 trainer = Trainer(
20     model=model,
21     args=training_args,
22     train_dataset=tokenized_datasets['train'],
23     eval_dataset=tokenized_datasets['validation'],
24     data_collator=data_collator,
25     compute_metrics=compute_metrics
26 )
27 trainer.train()

```

Listing 3: Training Setup

4.5.1 Training Setup Breakdown

- **TrainingArguments:** This class defines various training hyperparameters, such as the number of epochs, batch sizes, and evaluation strategy. Some key arguments:
 - `num_train_epochs=3`: The model will train for 3 epochs.
 - `per_device_train_batch_size=16`: The training batch size is 16 per device.
 - `evaluation_strategy="epoch"`: The evaluation will happen at the end of each epoch.
 - `save_strategy="no"`: No model checkpoints will be saved during training.

- **DataCollatorWithPadding:** This collator is used to pad sequences in the dataset to a fixed length. The padding ensures that all input sequences are of the same length, which is required for batch processing.
- **Trainer:** The Hugging Face **Trainer** class is responsible for managing the training process. It takes the model, training arguments, datasets, and metrics function and handles the training and evaluation loops.

4.6 Confusion Matrix and Visualizations

After training, we can evaluate the model and visualize the results, such as plotting a confusion matrix, accuracy over epochs, and loss over epochs.

```

1 import matplotlib.pyplot as plt
2 from sklearn.metrics import confusion_matrix,
   ConfusionMatrixDisplay, accuracy_score
3
4 predictions = trainer.predict(test_dataset=tokenized_datasets
   ['validation'])
5 logits = predictions.predictions
6 predicted_classes = np.argmax(logits, axis=-1)
7 true_labels = predictions.label_ids
8 accuracy = accuracy_score(true_labels, predicted_classes)
9
10 print(f"Test Accuracy: {accuracy:.2f}")
11 conf_matrix = confusion_matrix(true_labels, predicted_classes
   )
12 disp = ConfusionMatrixDisplay(conf_matrix, display_labels=[0,
   1])
13 plt.figure(figsize=(8, 8))
14 disp.plot(cmap=plt.cm.Blues, colorbar=True)
15 plt.title("Confusion Matrix")
16 plt.show()

```

Listing 4: Confusion Matrix and Evaluation

4.6.1 Confusion Matrix Breakdown

- After predicting the classes on the test set, we calculate the accuracy using `accuracy_score`.
- We generate the confusion matrix using `confusion_matrix`, which shows the count of true positives, false positives, true negatives, and false negatives.
- The confusion matrix is then displayed using `ConfusionMatrixDisplay`.

4.7 Training Logs and Loss/Accuracy Plots

The training logs provide information about the model's performance during training. The following code generates training loss and accuracy plots.

```
1 training_logs = trainer.state.log_history
2 print(training_logs)
3 losses = [log["eval_loss"] for log in training_logs if "
4     eval_loss" in log]
5 accuracy_logs = [log["eval_accuracy"] for log in
6     training_logs if "eval_accuracy" in log]
7 epochs = range(1, len(losses) + 1)
8
9 # Plotting training loss
10 plt.figure(figsize=(8, 6))
11 plt.plot(epochs, losses, marker='o', label='Training Loss')
12 plt.xlabel("Epoch")
13 plt.ylabel("Loss")
14 plt.title("Training Loss Over Epochs")
15 plt.legend()
16 plt.grid()
17 plt.show()
18
19 # Plotting training accuracy
20 accuracy_logs = [log["eval_accuracy"] for log in trainer.
21     state.log_history if "eval_accuracy" in log]
22 epochs = range(1, len(accuracy_logs) + 1)
23 if accuracy_logs:
24     plt.figure(figsize=(8, 6))
25     plt.plot(epochs, accuracy_logs, marker='o', label='
26         Training Accuracy')
27     plt.xlabel("Epoch")
28     plt.ylabel("Accuracy")
29     plt.title("Training Accuracy Over Epochs")
30     plt.legend()
31     plt.grid()
32     plt.show()
33 else:
34     print("No accuracy logs available.")
```

Listing 5: Training Logs and Plots

4.7.1 Loss/Accuracy Breakdown

- We extract the loss and accuracy logs from the trainer's history.
- We plot the training loss and accuracy over epochs using `matplotlib` to visualize the training process.

trainings-warn [12630/12630 34.57, Epoch 3/3]

Epoch	Training Loss	Validation Loss	Accuracy	Precision	Recall	F1
1	0.124600	0.595689	0.903670	0.930622	0.876126	0.902552
2	0.153900	0.490250	0.910550	0.896663	0.927928	0.913525
3	0.082200	0.557137	0.911697	0.908686	0.918919	0.913774

TrainOutput(global_step=12630, training_loss=0.12773739881569258, metrics={'train_runtime': 2098.0638, 'train_samples_per_second': 96.302, 'train_steps_per_second': 6.02, 'total_flos': 0.0, 'train_loss': 0.12773739881569258, 'epoch': 3.0})

Figure 1: Training the model

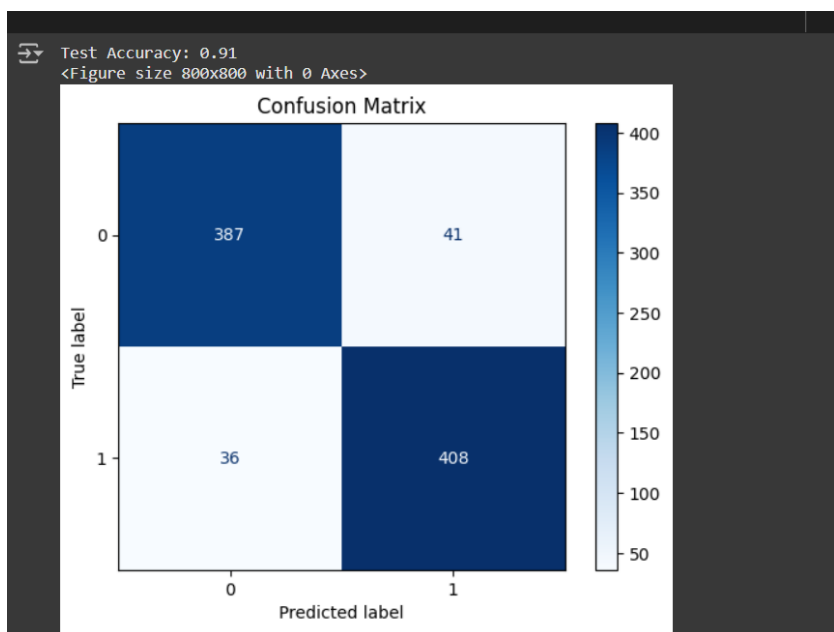


Figure 2: Confusion Matrix

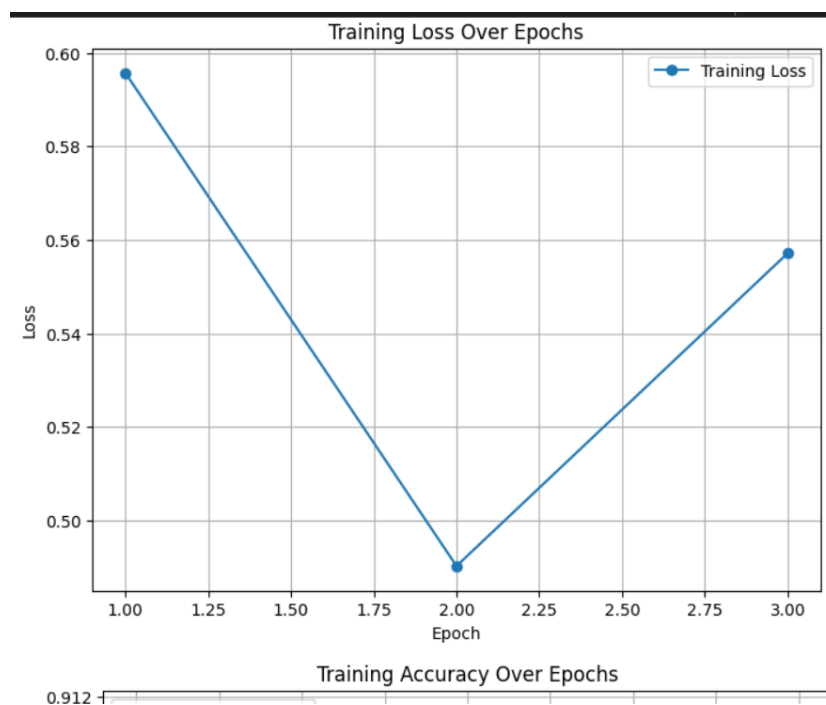


Figure 3: Training Loss Over Epochs

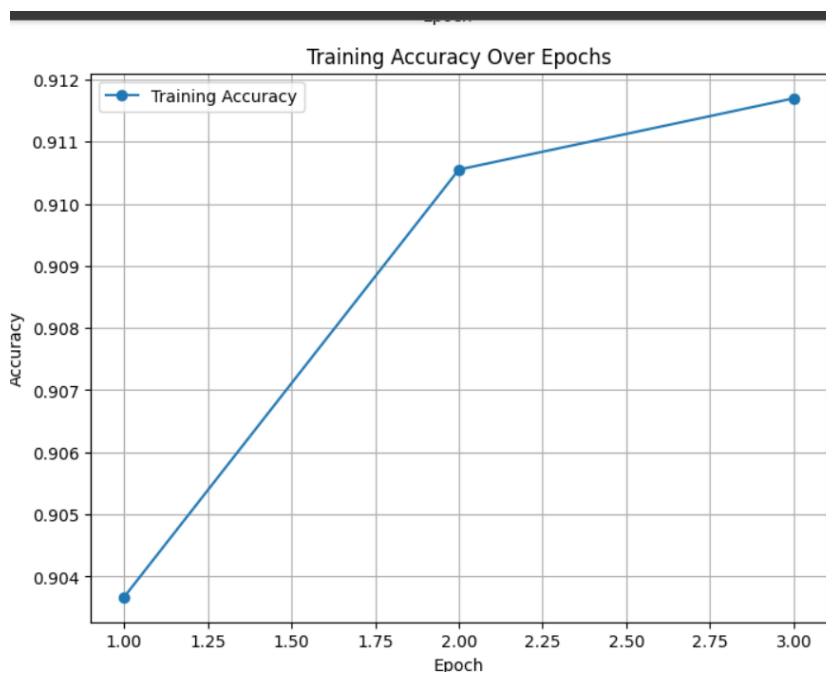


Figure 4: Training Accuracies Over epochs

5 Part 6: Bidirectional Attention Layer for Classification

In this part, we modify the model further by replacing the linear classification layer with a custom bidirectional attention layer. The attention layer uses a custom number of attention heads (12 in this case) to capture richer contextual information for sentiment classification. The primary objective is to improve the model's ability to understand the relationships between tokens across the entire sequence.

5.1 Bidirectional Attention Layer

A custom bidirectional attention layer is implemented using PyTorch's `MultiheadAttention` module. This layer allows the model to attend to all positions in the input sequence in both directions (past and future tokens), providing more comprehensive context for classification tasks.

The bidirectional attention layer is defined as follows:

```
1 import torch.nn as nn
2 from transformers import GPT2Config
3
4 class BidirectionalAttentionLayer(nn.Module):
5     def __init__(self, hidden_size, num_attention_heads):
6         super(BidirectionalAttentionLayer, self).__init__()
7         self.attention = nn.MultiheadAttention(embed_dim=
8             hidden_size, num_heads=num_attention_heads)
9
10    def forward(self, hidden_states):
11        return self.attention(hidden_states, hidden_states,
12                                hidden_states)[0]
```

5.2 Custom GPT-2 Model

We define a new class `CustomGPT2` that integrates the bidirectional attention layer into the GPT-2 architecture. The model first passes the input through the GPT-2 encoder and then applies the bidirectional attention layer on the hidden states.

The model is defined as:

```
1 class CustomGPT2(nn.Module):
2     def __init__(self, model_name='gpt2', num_attention_heads
3         =12):
4         super(CustomGPT2, self).__init__()
```



```

4         self.gpt2_config = GPT2Config.from_pretrained(
5             model_name)
6         self.gpt2 = GPT2Model.from_pretrained(model_name,
7             config=self.gpt2_config)
8         self.bidirectional_attention =
9             BidirectionalAttentionLayer(
10                 hidden_size=self.gpt2.config.n_embd,
11                 num_attention_heads=num_attention_heads
12             )
13
14     def forward(self, input_ids, attention_mask=None, labels=
15         None):
16         outputs = self.gpt2(input_ids=input_ids,
17             attention_mask=attention_mask)
18         hidden_states = outputs.last_hidden_state
19         attention_output = self.bidirectional_attention(
20             hidden_states)
21
22         loss = None
23         logits = attention_output.mean(dim=1)
24
25         if labels is not None:
26             loss_fn = nn.CrossEntropyLoss()
27             loss = loss_fn(logits, labels)
28
29         return {"loss": loss, "logits": logits} if labels is
30             not None else {"logits": logits}

```

5.3 Training Configuration

The training process is similar to previous parts, with the addition of the new model class `CustomGPT2`. The training loop is set up using the `Trainer` class from the `transformers` library, and the training arguments are defined as follows:

```

1 device = torch.device("cuda" if torch.cuda.is_available()
2     else "cpu")
3
4 model = CustomGPT2(num_attention_heads=12).to(device)
5
6 training_args = TrainingArguments(
7     output_dir='./results',
8     num_train_epochs=3,
9     per_device_train_batch_size=16,
10    per_device_eval_batch_size=64,
11    warmup_steps=500,
12    weight_decay=0.01,
13    logging_dir='./logs',

```

```

12     save_strategy="no",
13     save_total_limit=0,
14     evaluation_strategy="epoch"
15 )
16
17 data_collator = DataCollatorWithPadding(tokenizer=tokenizer,
18     pad_to_multiple_of=8)
19
20 trainer = Trainer(
21     model=model,
22     args=training_args,
23     train_dataset=tokenized_datasets['train'],
24     eval_dataset=tokenized_datasets['validation'],
25     data_collator=data_collator,
26     compute_metrics=compute_metrics
27 )
28
29 trainer.train()

```

[[2650/12630/3241, epoch 3/3]]

Epoch	Training Loss	Validation Loss	Accuracy	Precision	Recall	F1
1	0.229000	0.264501	0.903670	0.941178	0.864865	0.901408
2	0.173800	0.310741	0.901378	0.903153	0.903153	0.903153
3	0.105100	0.372378	0.915138	0.920455	0.912162	0.916290

TrainOutput(global_step=12638, training_loss=0.24692074941859755, metrics={'train_runtime': 1984.1169, 'train_samples_per_second': 101.832, 'train_steps_per_second': 6.366, 'total_flos': 0.0, 'train_loss': 0.24692074941859755, 'epoch': 3.0})

Figure 5: Training the model

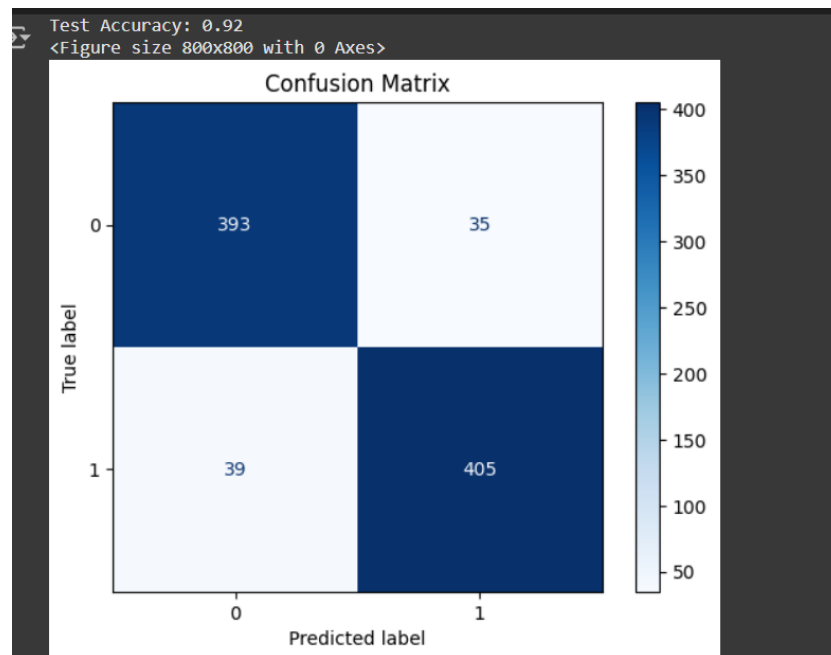


Figure 6: Confusion Matrix

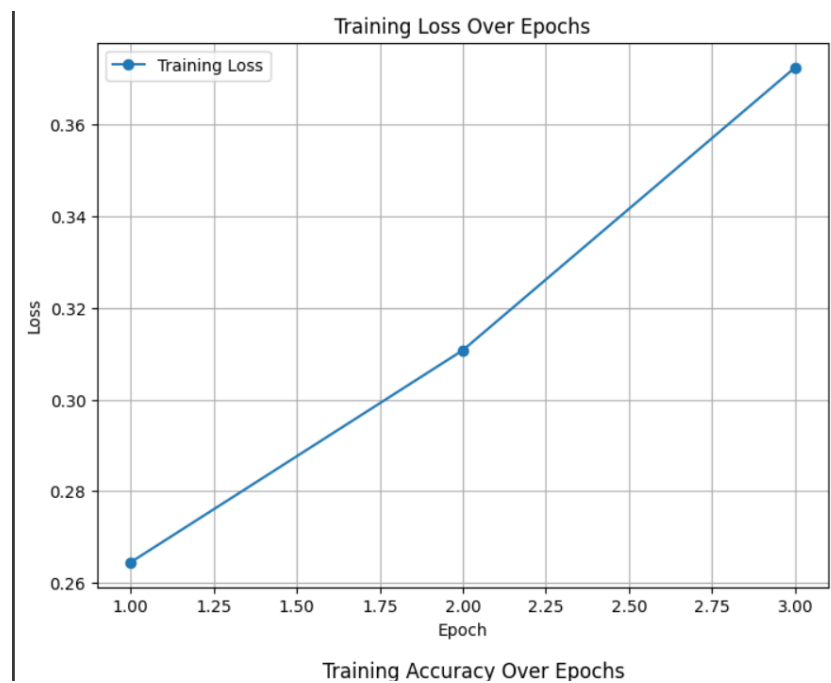


Figure 7: Training Loss Over Epochs



Figure 8: Training Accuracies Over epochs

6 Part 7: Using Left-to-Right and Right-to-Left Unidirectional Attention

6.1 General Explanation

In this part, the task is to experiment with **left-to-right** (LTR) and **right-to-left** (RTL) unidirectional attention mechanisms. Instead of using bidirectional attention or the previous bidirectional attention layer, we will explore two separate models:

- A model that processes the input sequence from left to right (LTR).
- A model that processes the input sequence from right to left (RTL).

This setup allows us to evaluate the performance of unidirectional attention mechanisms on sentiment classification. The key difference between this part and the previous part is the direction of attention, which is controlled by flipping the input sequence for RTL.

6.2 Code Explanation

The following code defines two models using GPT-2 with unidirectional attention:

```

1 from transformers import GPT2Config
2 import torch.nn as nn
3 import torch
4 from transformers import GPT2LMHeadModel
5 from transformers import Trainer, TrainingArguments,
   DataCollatorWithPadding
6 from sklearn.metrics import accuracy_score,
   precision_recall_fscore_support
7
8 device = torch.device("cuda")
9
10 class gpt_model(nn.Module):
11     def __init__(self, direction='ltr'):
12         super(gpt_model, self).__init__()
13         self.gpt2 = GPT2LMHeadModel.from_pretrained("gpt2")
14         self.gpt2.config.tie_word_embeddings = False
15         self.linear = nn.Linear(self.gpt2.config.hidden_size,
16                                 2)
17         self.direction = direction
18
19     def forward(self, input_ids, attention_mask=None, labels=
20 None):
21         # Flip the input for right-to-left attention
22         if self.direction == 'rtl':
23             input_ids = torch.flip(input_ids, dims=[1])
24
25         # Pass through the model and get hidden states
26         outputs = self.gpt2(input_ids=input_ids,
27                             attention_mask=attention_mask,
28                             output_hidden_states=True)
29         hidden_states = outputs.hidden_states[-1]
30         logits = self.linear(hidden_states)
31
32         # Compute loss if labels are provided
33         loss = None
34         if labels is not None:
35             logits = logits.mean(dim=1)
36             loss_fn = nn.CrossEntropyLoss()
37             loss = loss_fn(logits, labels)
38
39         return {"loss": loss, "logits": logits} if labels is
40 not None else {"logits": logits}
41
42 # Initialize models for left-to-right and right-to-left
43 processing
44 ltr_model = gpt_model(direction='ltr').to(device)
45 rtl_model = gpt_model(direction='rtl').to(device)

```

```

41 # Training configuration
42 training_args = TrainingArguments(
43     output_dir='./results',
44     num_train_epochs=3,
45     per_device_train_batch_size=16,
46     per_device_eval_batch_size=64,
47     warmup_steps=500,
48     weight_decay=0.01,
49     logging_dir='./logs',
50     evaluation_strategy="epoch",
51     save_strategy="no",
52     save_total_limit=0,
53 )
54
55 data_collator = DataCollatorWithPadding(tokenizer=tokenizer,
56     pad_to_multiple_of=8)
57
58 # Trainer for left-to-right model
59 trainer_left_to_right = Trainer(
60     model=ltr_model,
61     args=training_args,
62     train_dataset=tokenized_datasets['train'],
63     eval_dataset=tokenized_datasets['validation'],
64     data_collator=data_collator,
65     compute_metrics=compute_metrics,
66 )
67
68 # Trainer for right-to-left model
69 trainer_right_to_left = Trainer(
70     model=rtl_model,
71     args=training_args,
72     train_dataset=tokenized_datasets['train'],
73     eval_dataset=tokenized_datasets['validation'],
74     data_collator=data_collator,
75     compute_metrics=compute_metrics,
76 )
77
78 trainer_right_to_left.train()

```

6.3 Explanation of Code Differences

- The `gpt_model` class now accepts a `direction` argument which can be either `'ltr'` (left-to-right) or `'rtl'` (right-to-left).
- If the `direction` is set to `'rtl'`, the input sequence is flipped using `torch.flip(input_ids, dims=[1])` to process the sequence from

right to left.

- The model architecture and training setup remain similar to the previous sections. However, two separate models are instantiated, one for left-to-right and one for right-to-left processing.

6.4 Training

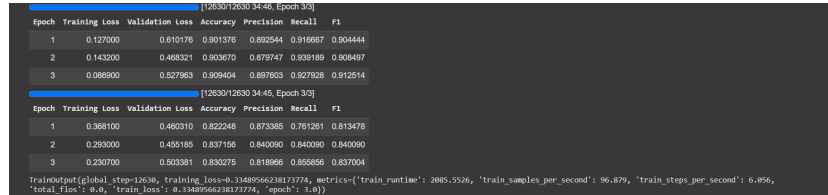
The models are trained separately using the **Trainer** class. The training arguments are set up with the same parameters as in previous sections, with the training and evaluation datasets being provided.

Left-to-Right Model Training:

- The model processes the input sequence from left to right.
- The `trainer_left_to_right.train()` function trains the left-to-right model.

Right-to-Left Model Training:

- The model processes the input sequence from right to left.
- The `trainer_right_to_left.train()` function trains the right-to-left model.



```
[[12630/12630 34.46, Epoch 3/3]]
Epoch Training Loss Validation Loss Accuracy Precision Recall F1
1 0.127000 0.610176 0.901376 0.892544 0.916667 0.904444
2 0.143200 0.468321 0.903670 0.879747 0.909189 0.908487
3 0.066900 0.527963 0.909404 0.897603 0.927928 0.912514

[[12630/12630 34.45, Epoch 3/3]]
Epoch Training Loss Validation Loss Accuracy Precision Recall F1
1 0.368100 0.460310 0.822248 0.873388 0.761261 0.813478
2 0.293000 0.455185 0.837156 0.840090 0.840090 0.840090
3 0.230700 0.503381 0.830275 0.818966 0.855856 0.837004

TrainOutput(global_stop=12638, training_loss=0.33489566238173774, metrics={'train_runtime': 2085.5526, 'train_samples_per_second': 96.879, 'train_steps_per_second': 6.496,
'total_flos': 8.6, 'train_loss': 0.33489566238173774, 'epoch': 3.0})
```

Figure 9: Training the model

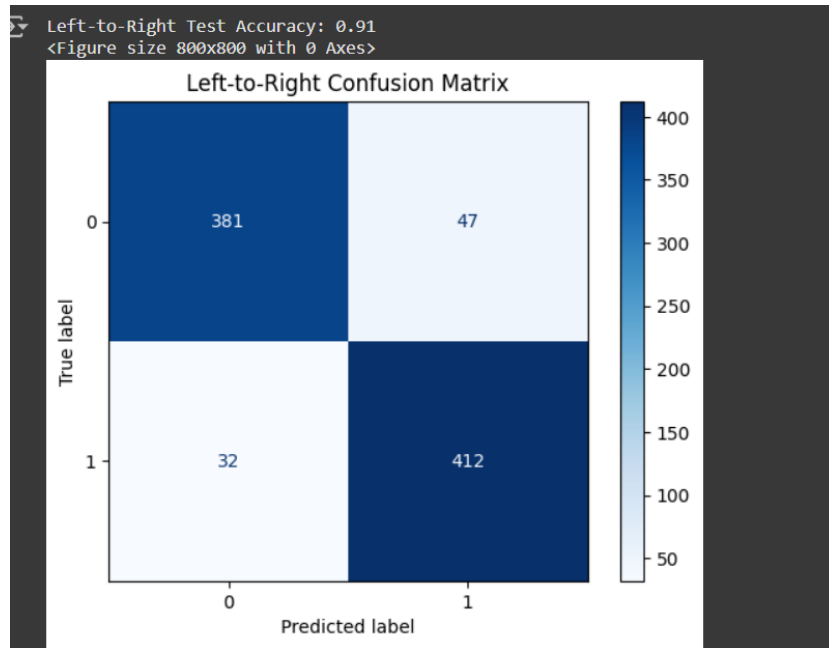


Figure 10: Confusion Matrix

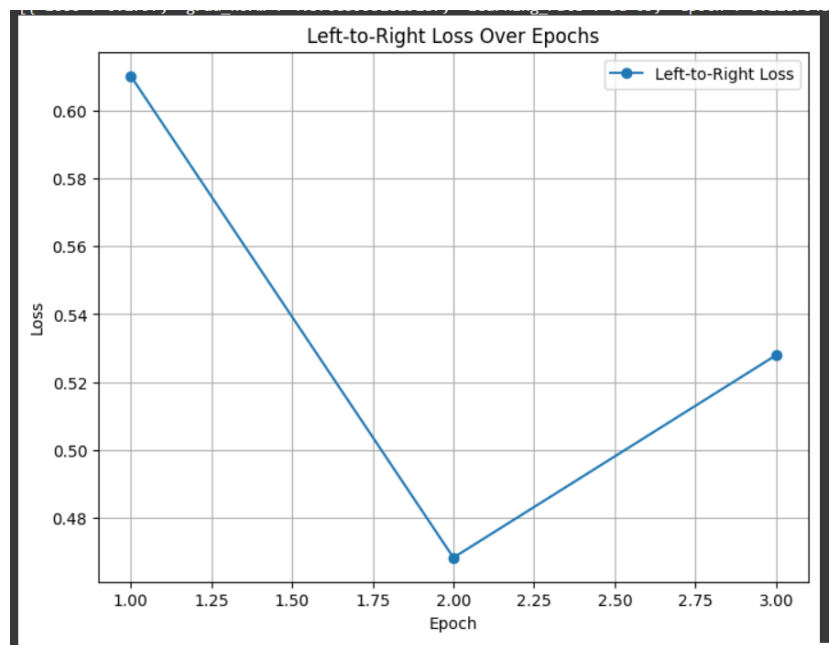


Figure 11: Training Loss Over Epochs

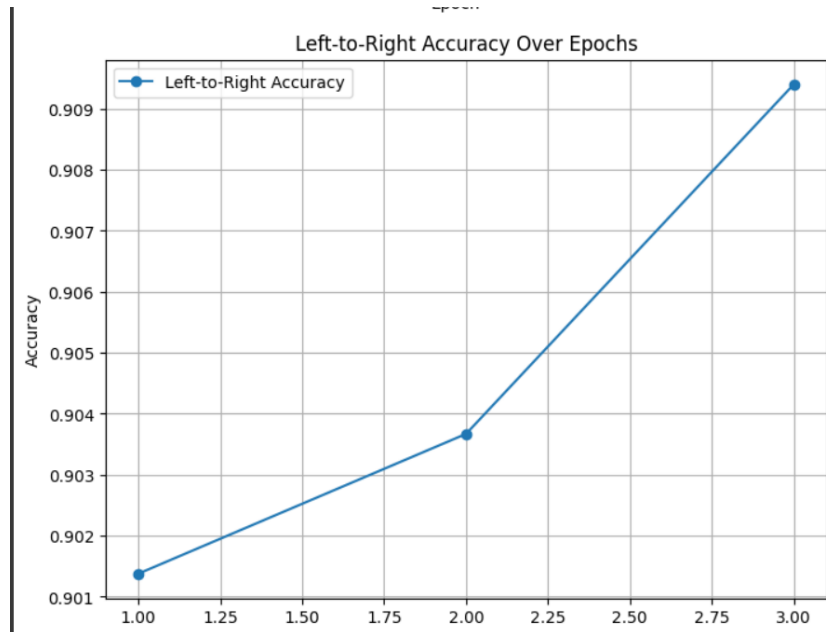


Figure 12: Training Accuracies Over epochs

7 Part 8: Zero-shot Performance with Pre-trained BERT Model

7.1 General Explanation

In this part, the objective is to evaluate the performance of a pre-trained BERT model on a sentiment classification task without any additional fine-tuning. This approach is known as **zero-shot learning**. In zero-shot learning, the pre-trained model is used directly on the test data, bypassing the training phase, and does not require any further optimization or adaptation to the specific dataset.

The idea behind zero-shot learning is that the pre-trained model, such as BERT, has learned useful representations of language from massive corpora of text (e.g., Wikipedia and BookCorpus). These learned representations allow the model to generalize well to various tasks, even those it hasn't explicitly been trained on.

In this specific task, we use a pre-trained BERT model (BERT-base-uncased) from the Hugging Face model hub. This model has already been trained on large-scale datasets for a variety of language tasks. We load this model and apply it to the sentiment classification task on the SST-2 dataset, where the goal is to classify sentences as positive or negative. The model

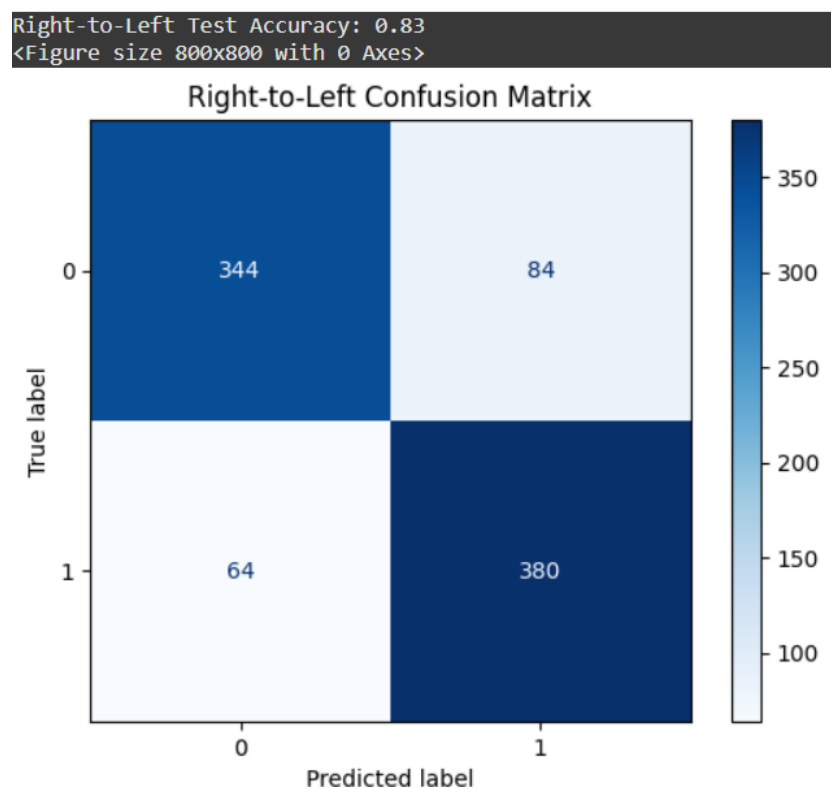


Figure 13: Confusion Matrix

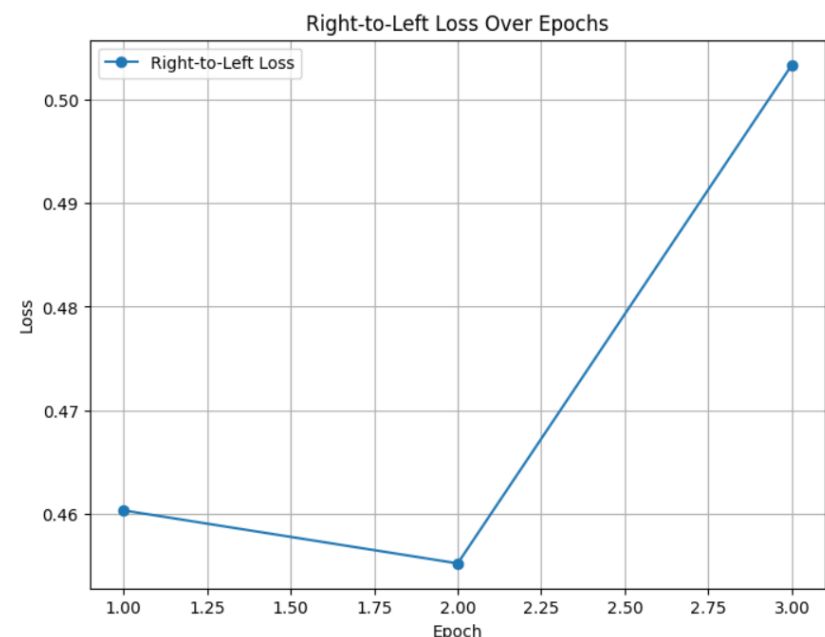


Figure 14: Training Loss Over Epochs

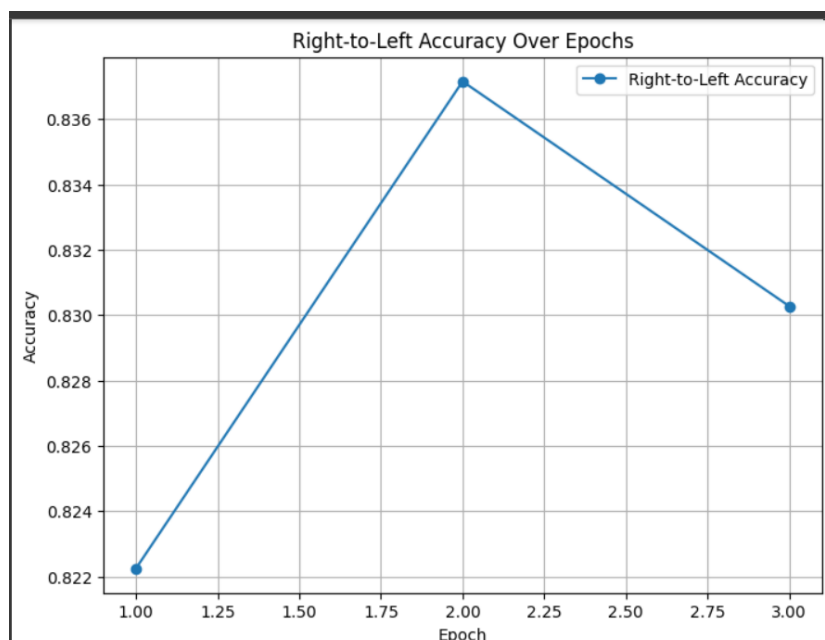


Figure 15: Training Accuracies Over epochs

processes the input text, and we compute the performance metrics without further training or modification.

Since the BERT model is a transformer-based model, it has been pre-trained using a language modeling objective that helps it learn contextual relationships between words. For sentiment analysis, we leverage this ability to understand context and apply it directly to predict sentiment, without any fine-tuning or additional task-specific training.

The following code demonstrates how we load the pre-trained BERT model and tokenizer, tokenize the test dataset, and make predictions on the test data:

7.2 Code Explanation

The code below demonstrates how to load a pre-trained BERT model and tokenizer, process the test dataset, and make predictions on the test data:

```
1 from transformers import BertTokenizer,
   BertForSequenceClassification
2 import torch
3
4 device = torch.device("cuda" if torch.cuda.is_available()
   else "cpu")
5
6 # Load pre-trained BERT tokenizer and model
7 tokenizer = BertTokenizer.from_pretrained("bert-base-uncased"
   )
8 model = BertForSequenceClassification.from_pretrained("bert-
   base-uncased", num_labels=2).to(device)
9
10 # Tokenize the test dataset
11 def tokenize_function(examples):
12     return tokenizer(examples["sentence"], padding=True,
   truncation=True, max_length=512)
13
14 tokenized_test_dataset = dataset["validation"].map(
   tokenize_function, batched=True)
15
16 # Prepare input tensors for the test data
17 test_inputs = torch.tensor(tokenized_test_dataset["input_ids"]
   ).to(device)
18 test_masks = torch.tensor(tokenized_test_dataset["
   attention_mask"]).to(device)
19 test_labels = torch.tensor(tokenized_test_dataset["label"]
   ).to(device)
```

7.3 Explanation of the Code

- `BertTokenizer` and `BertForSequenceClassification` are used to load the pre-trained BERT tokenizer and model. We use the `bert-base-uncased` version, which is a widely used version of BERT that is trained on lowercased English text.
- The `tokenize_function` processes the input sentences from the test dataset. It tokenizes the sentences, ensuring that they are padded to the same length, truncated to a maximum of 512 tokens, and converted into the input format required by the BERT model.
- The test dataset is tokenized in a batched manner using the `map` function to efficiently process all the data at once.
- The tokenized data is then converted into PyTorch tensors for the model to process, and these tensors are moved to the GPU (if available) for faster inference.

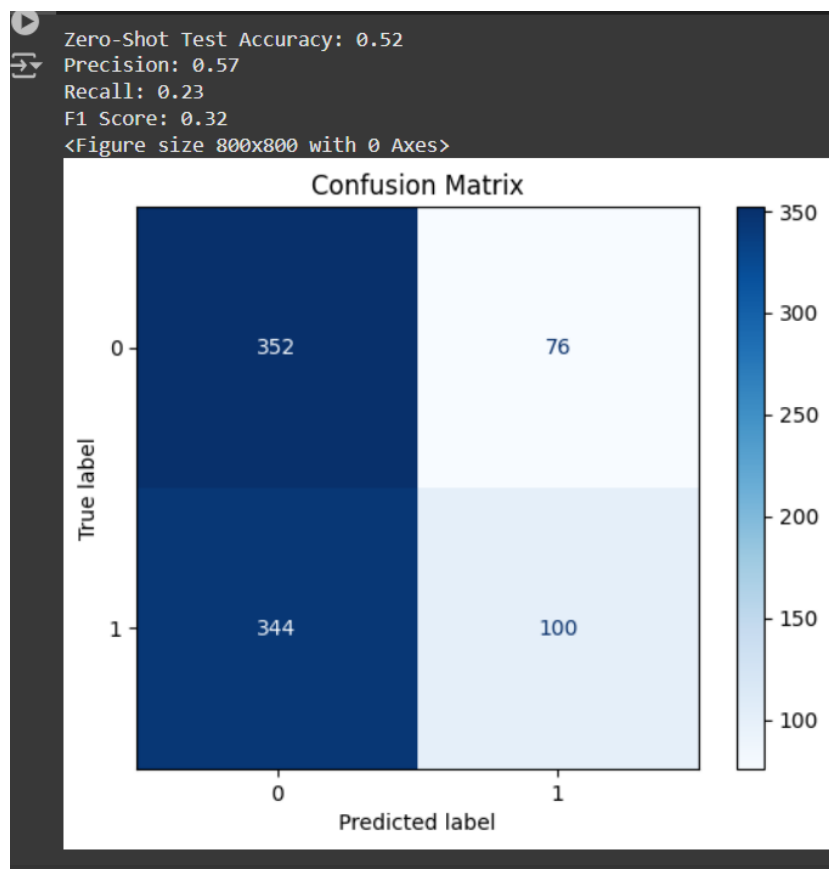


Figure 16: Training Accuracies Over epochs