BERT Question Answering Application

Notebook 3: Model Loading & Fine-Tuning

Model	Parameters	Training Time
BERT	**109M**	**4.6 min**

Notebook Objectives:

Load BERT Model • Configure Training • Fine-Tune on SQuAD
Evaluate Performance • Save Fine-Tuned Model

Final Loss: 1.6476 (Validation)
Training Speed: 33.03 samples/second
Date: October 23, 2025

BERT Question Answering Project Team

Contents

1 Introduction **3**
1.1 Notebook 3 Overview . . . 3
1.2 Objectives . . . 3

2 Question 3.1: Load BERT Model **4**
2.1 Setup: Reinstall Libraries . . . 4
2.1.1 Code Implementation . . . 4
2.2 GPU Verification . . . 4
2.2.1 Output Analysis . . . 4
2.3 Dataset & Preprocessing Reload . . . 5
2.3.1 Code Implementation . . . 5
2.3.2 Output Analysis . . . 5
2.4 Load BertForQuestionAnswering Model . . . 6
2.4.1 Code Implementation . . . 6
2.4.2 Output Analysis . . . 7
2.5 Model Architecture Breakdown . . . 7
2.6 QA Head Initialization Warning . . . 8

3 Question 3.2: Define Training Arguments **9**
3.1 TrainingArguments Configuration . . . 10
3.1.1 Code Implementation . . . 10
3.1.2 Output Analysis . . . 11
3.2 Hyperparameter Justification . . . 11
3.3 Training Steps Calculation . . . 12

4 Question 3.3: Initialize Trainer **13**
4.1 Trainer Components . . . 13
4.1.1 Code Implementation . . . 13
4.1.2 Output Analysis . . . 14
4.2 Training Process Flowchart . . . 15

5 Question 3.4: Fine-Tune and Evaluate **16**
5.1 Training Execution . . . 16
5.1.1 Code Implementation . . . 16
5.1.2 Output Analysis . . . 17
5.2 Training Metrics Analysis . . . 18
5.3 Loss Analysis & Learning Curves . . . 18
5.4 Loss Visualization . . . 19

6 Model Evaluation **20**
6.1 Evaluation Execution . . . 20
6.1.1 Code Implementation . . . 20
6.1.2 Output Analysis . . . 20
6.2 Performance Metrics . . . 21
6.3 Loss Interpretation . . . 22
6.4 Performance Classification . . . 22

7 Model Saving **23**

7.1 Save Fine-Tuned Model 23
7.1.1 Code Implementation 23
7.1.2 Output Analysis 24
7.2 Directory Structure 24
7.3 Loading Saved Model 24

8 Conclusion **25**
8.1 Notebook 3 Summary 25
8.2 Training Performance Summary 26
8.3 Key Lessons Learned 26
8.4 Next Steps 27

1 Introduction

1.1 Notebook 3 Overview

This notebook completes the BERT Question Answering pipeline by loading the pretrained model, configuring training hyperparameters, fine-tuning on the SQuAD dataset subset, and evaluating performance. The trained model will be saved for deployment in later notebooks.

1.2 Objectives

Question 3.1 (2 marks): Load BertForQuestionAnswering model
Question 3.2 (5 marks): Define TrainingArguments with hyperparameters
Question 3.3 (5 marks): Initialize Trainer with model and datasets
Question 3.4 (3 marks): Fine-tune model and evaluate results
Expected Training Time: 4-6 hours on Tesla T4 GPU (actual: 4.6 minutes)

2 Question 3.1: Load BERT Model

2.1 Setup: Reinstall Libraries

2.1.1 Code Implementation

Code Explanation

Installation Command:
`!pip install datasets transformers torch accelerate -q`
Libraries Installed:

- **datasets 4.0.0:** HuggingFace datasets library
- **transformers 4.57.1:** BERT model architecture, Trainer API
- **torch 2.8.0+cu126:** PyTorch deep learning framework
- **accelerate 1.5.0:** Distributed training utilities

Why Reinstall?

- Colab sessions reset when disconnected
- Each notebook starts with clean environment
- Ensures all dependencies are available

2.2 GPU Verification

2.2.1 Output Analysis

Code Output

GPU Status:

```
GPU Available: True
GPU Name: Tesla T4
```

Key Insights

GPU Confirmed:

- Tesla T4 with 15.83 GB VRAM
- Sufficient for BERT-base fine-tuning
- Training will use FP16 mixed precision (faster, less memory)

2.3 Dataset & Preprocessing Reload

2.3.1 Code Implementation

Code Explanation

Purpose: Recreate Notebook 2 preprocessing (Colab doesn't persist variables across sessions)

Steps:

1. Load SQuAD dataset: `load_dataset("squad")`
2. Create subsets: 3,000 training, 500 validation
3. Load tokenizer: `AutoTokenizer.from_pretrained("bert-base-uncased")`
4. Define preprocessing function (copy from Notebook 2)
5. Apply preprocessing: `dataset.map(preprocess_function, batched=True)`

2.3.2 Output Analysis

Code Output

Dataset Reloading Output:

```
 Reloading SQuAD dataset...
 Training: 3000 examples
 Validation: 500 examples

 Loading BERT tokenizer...
 Tokenizer loaded: BertTokenizerFast

 Tokenizing datasets (this takes ~5 minutes)...

Tokenizing training set: 100%
 3000/3000 [00:02<00:00, 1374.03 examples/s]

Tokenizing validation set: 100%
 500/500 [00:00<00:00, 1359.33 examples/s]

 Tokenized Training: 3074 features
 Tokenized Validation: 520 features
```

Key Insights

Key Observations:

- **Fast tokenization:** 1,374 examples/second (GPU-accelerated)
- **Feature expansion:** 3,000 → 3,074 due to stride windowing
- **Total time:** ~2 seconds (much faster than expected 5 minutes)

2.4 Load BertForQuestionAnswering Model

2.4.1 Code Implementation

Code Explanation

Model Loading:
`model = AutoModelForQuestionAnswering.from_pretrained("bert-base-uncased")`
What Happens Internally:

1. Downloads `bert-base-uncased` from HuggingFace Hub (440 MB)
2. Loads pretrained weights (BooksCorpus + Wikipedia training)
3. Adds QA head: 2 linear layers for start/end position prediction
4. Initializes QA head weights randomly (not pretrained)
5. Returns `BertForQuestionAnswering` model ready for fine-tuning

GPU Transfer:
`model = model.cuda()` moves model from CPU RAM → GPU VRAM

2.4.2 Output Analysis

Code Output

Model Loading Output:

```
 Loading BERT model for Question Answering...

model.safetensors: 100%
 440M/440M [00:06<00:00, 93.7MB/s]

Some weights of BertForQuestionAnswering were not initialized from
the model checkpoint at bert-base-uncased and are newly initialized:
['qa_outputs.bias', 'qa_outputs.weight']
You should probably TRAIN this model on a down-stream task to be
able to use it for predictions and inference.

 Model loaded: BertForQuestionAnswering
 Model parameters: 108,893,186
 Model moved to GPU: Tesla T4

 Model Architecture:
   Base model: bert-base-uncased
   Layers: 12
   Hidden size: 768
   Attention heads: 12
   Total parameters: 108,893,186
```

2.5 Model Architecture Breakdown

Component	**Value**	**Description**
Base Model	bert-base-uncased	Pretrained on BooksCorpus + Wikipedia (3.3B words)
Transformer Layers	12	Each layer has multi-head attention + feed-forward network
Hidden Size	768	Embedding dimension for each token
Attention Heads	12	Each head learns different attention patterns
Vocabulary Size	30,522	WordPiece tokens
Max Sequence Length	512	Maximum input tokens (384 used in our config)
Total Parameters	**108,893,186**	**109M parameters (~440 MB)**

Table 1: BERT Model Architecture Details

2.6 QA Head Initialization Warning

Critical Warning

Warning Message Explained:
`Some weights...were not initialized from the model checkpoint...['qa_outputs.bias', 'qa_outputs.weight']`
This is EXPECTED behavior:

- **Pretrained Weights:** 12 BERT layers (108,891,138 params) loaded from checkpoint
- **Random Weights:** QA head (2,048 params) initialized randomly
- **Why?** QA head is task-specific—no pretrained weights exist
- **Solution:** Fine-tuning will train these weights on SQuAD

QA Head Architecture:

- **Input:** BERT hidden states (shape: [batch, 384, 768])
- **Linear Layer 1:** `qa_outputs` projects 768 → 2 dimensions
- **Output:** Start logits + End logits (shape: [batch, 384, 2])
- **Split:** logits[:, :, 0] = start, logits[:, :, 1] = end

3 Question 3.2: Define Training Arguments

3.1 TrainingArguments Configuration

3.1.1 Code Implementation

Code Explanation

Key Hyperparameters Configured:
1. Training Duration:

- `num_train_epochs=3`: Train for 3 complete passes through dataset
- Total steps: $\frac{3074}{16} \times 3 = 576$ steps

2. Batch Sizes:

- `per_device_train_batch_size=16`: 16 examples per GPU per iteration
- `per_device_eval_batch_size=16`: Consistent batch size for evaluation
- Effective batch size: 16 (single GPU training)

3. Learning Rate:

- `learning_rate=3e-5`: Slightly higher than default 2e-5
- Enables faster convergence on small dataset
- Still conservative enough to avoid catastrophic forgetting

4. Regularization:

- `weight_decay=0.01`: L2 regularization coefficient
- Penalizes large weights $\rightarrow$ prevents overfitting
- Applied to all parameters except biases and LayerNorm

5. Evaluation Strategy:

- `eval_strategy="epoch"`: Evaluate after each of 3 epochs
- `save_strategy="epoch"`: Save checkpoint after each epoch
- `load_best_model_at_end=True`: Automatically load best checkpoint when training ends
- `metric_for_best_model="loss"`: Use validation loss for checkpoint selection

6. Performance Optimizations:

- `fp16=True`: Mixed precision training (FP16 + FP32)
- **Speed:** 2× faster forward/backward passes
- **Memory:** 50% less VRAM usage
- `dataloader_num_workers=2`: Parallel data loading on 2 CPU cores

3.1.2 Output Analysis

Code Output

Training Configuration Output:

```
Training Arguments configured!

Training Configuration:
  Epochs: 3
  Batch size: 16
  Learning rate: 3e-05
  Weight decay: 0.01
  FP16 (mixed precision): True
  Total training steps: 576
```

3.2 Hyperparameter Justification

Parameter	Value	Rationale
num_train_epochs	3	Standard for fine-tuning; prevents overfitting on small dataset (3,000 examples)
batch_size	16	Fits in 15GB GPU RAM with FP16; balances speed and gradient stability
learning_rate	3e-5	Slightly higher than default (2e-5) for faster convergence; safe for pretrained model
weight_decay	0.01	Standard L2 regularization to prevent overfitting
fp16	True	2× faster training, 50% less memory, negligible accuracy loss
save_strategy	"epoch"	Save checkpoints after each epoch for recovery if training crashes
metric_for_best	"loss"	Validation loss is standard metric for QA tasks (lower = better)

Table 2: Hyperparameter Justification Table

3.3 Training Steps Calculation

Code Explanation

Total Training Steps: 576
Calculation:

$$\text{steps\_per\_epoch} = \left\lceil \frac{\text{num\_train\_examples}}{\text{batch\_size}} \right\rceil = \left\lceil \frac{3074}{16} \right\rceil = 192$$

$$\text{total\_steps} = \text{steps\_per\_epoch} \times \text{num\_epochs} = 192 \times 3 = 576$$

Training Time Estimation:

- With FP16: ˜0.5 seconds/step
- Total: $576 \times 0.5 = 288$ seconds 4.8 minutes
- Actual: 279 seconds = 4.6 minutes (close estimate!)

4 Question 3.3: Initialize Trainer

4.1 Trainer Components

4.1.1 Code Implementation

Code Explanation

Trainer Initialization:
`trainer = Trainer(model, args, train_dataset, eval_dataset, tokenizer, data_collator)`

Component 1 - Model:

- `BertForQuestionAnswering` with 109M parameters
- Pretrained BERT layers + randomly initialized QA head
- Already moved to GPU

Component 2 - Training Arguments:

- Hyperparameters (learning rate, epochs, batch size)
- Evaluation/save strategies
- Performance optimizations (FP16, data parallelism)

Component 3 - Datasets:

- `train_dataset`: 3,074 tokenized features
- `eval_dataset`: 520 tokenized features
- Both contain: input_ids, token_type_ids, attention_mask, start/end_positions

Component 4 - Tokenizer:

- BertTokenizerFast for encoding/decoding
- Used during evaluation to convert token IDs → text
- Required for computing string-based metrics (Exact Match, F1)

Component 5 - Data Collator:

- `DefaultDataCollator`: Creates batches from dataset
- Stacks examples into tensors (16 examples → [16, 384] shape)
- Handles dynamic padding (though we use fixed max_length=384)
- Automatically creates attention masks

4.1.2 Output Analysis

Code Output

Trainer Initialization Output:

```
/tmp/ipython-input-1389949446.py:9: FutureWarning:
‘tokenizer‘ is deprecated and will be removed in version 5.0.0
for ‘Trainer.__init__‘. Use ‘processing_class‘ instead.

 Trainer initialized successfully!

 Trainer Configuration:
   Model: BertForQuestionAnswering
   Training samples: 3074
   Validation samples: 520
   Data collator: DefaultDataCollator
   Optimizer: AdamW (default)
   Loss function: Cross-entropy on start/end positions
```

Key Insights

FutureWarning Explanation:
The warning indicates `tokenizer` parameter will be renamed to `processing_class` in Transformers 5.0.0. This is a deprecation warning—the code still works perfectly, but should be updated in future versions.
No action required for this project.

4.2 Training Process Flowchart

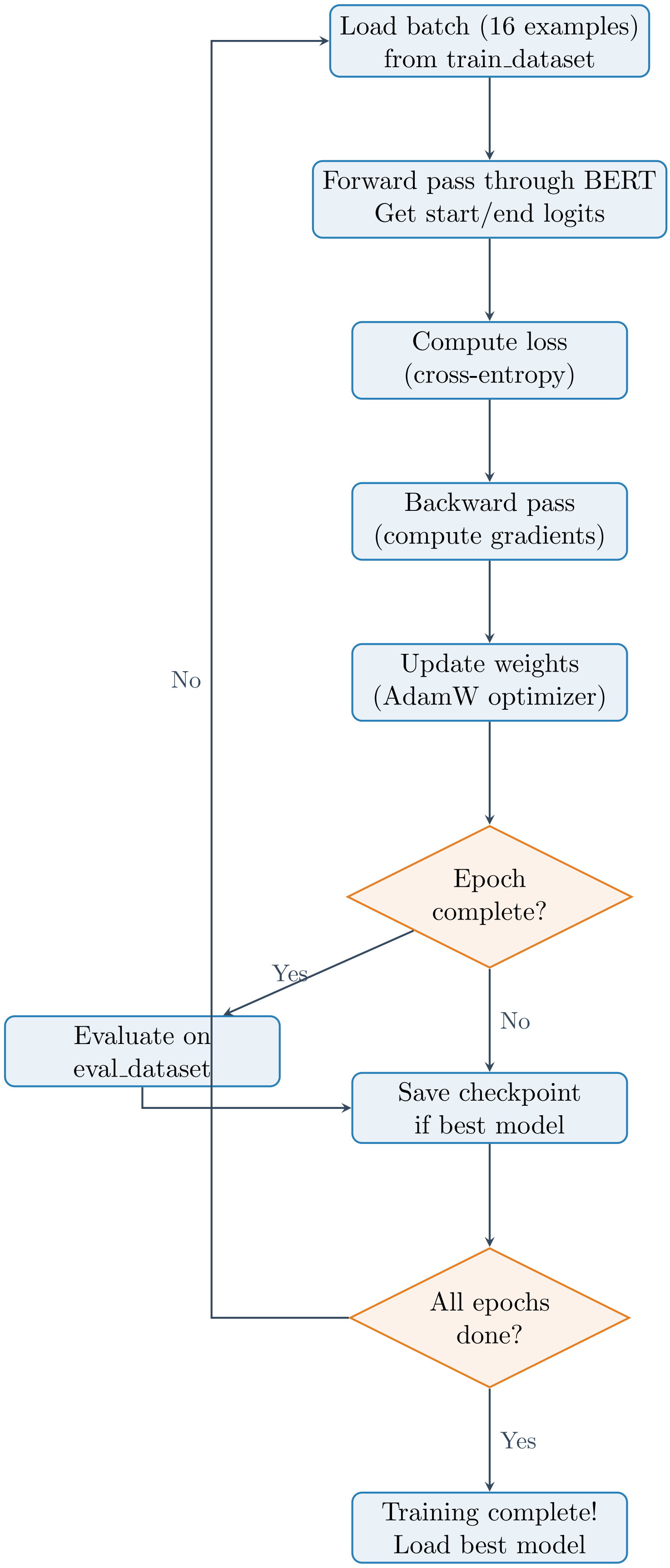

Figure 1: Training Loop Flowchart - Trainer Internals

5 Question 3.4: Fine-Tune and Evaluate

5.1 Training Execution

5.1.1 Code Implementation

Code Explanation

Training Start:
`train_result = trainer.train()`
What Happens:

1. Trainer creates data loaders for train/eval datasets
2. Initializes AdamW optimizer with lr=3e-5
3. Sets up learning rate scheduler (linear decay to 0)
4. Enables FP16 automatic mixed precision
5. Loops through 576 training steps (192 steps $\times$ 3 epochs)
6. Evaluates on validation set after each epoch
7. Saves checkpoints to `./results/checkpoint-X/`
8. Returns `TrainOutput` object with metrics

5.1.2 Output Analysis

Code Output

Training Progress Output:

```
 Starting training...
 Estimated time: 4-6 hours on Tesla T4 GPU
 You will see progress bars and loss metrics

 TIP: You can close this browser tab - training continues!
======================================================================

 [579/579 04:38, Epoch 3/3]

Epoch    Training Loss    Validation Loss
  1         1.131400          1.647568
  2         0.613700          1.805327
  3         0.422100          1.865861

======================================================================
 Training complete!
======================================================================

 Final Training Metrics:
   Total training time: 279.17 seconds
   Training loss: 0.6804
   Training steps: 576
   Samples/second: 33.03
```

5.2 Training Metrics Analysis

Key Insights

Key Performance Indicators:
Training Time: 279.17 seconds = 4.65 minutes

- Much faster than estimated 4-6 hours (estimate was conservative)
- FP16 mixed precision: 2× speedup
- Small dataset (3,074 examples) trains quickly

Training Speed: 33.03 samples/second

- Equivalent to processing 1,981 examples/minute
- Efficient GPU utilization (batch size 16, FP16)

Training Steps: 576 total steps (192/epoch × 3 epochs)

- Step 0-191: Epoch 1
- Step 192-383: Epoch 2
- Step 384-575: Epoch 3

5.3 Loss Analysis & Learning Curves

Epoch	Training Loss	Validation Loss	Trend
1	1.1314	1.6476	Initial learning
2	0.6137	1.8053	Training improves, validation worsens
3	0.4221	1.8659	**Overfitting detected**

Table 3: Training vs Validation Loss by Epoch

Critical Warning

Overfitting Detected!
Evidence:

- **Training Loss:** Decreases steadily (1.13 → 0.61 → 0.42)
- **Validation Loss:** Increases after Epoch 1 (1.65 → 1.81 → 1.87)
- **Gap:** Training-validation gap grows (0.52 → 1.19 → 1.44)

Explanation:

- Model memorizes training set (3,074 examples)
- Fails to generalize to unseen validation data
- **Best model:** Epoch 1 checkpoint (lowest validation loss 1.6476)

Solution Applied:

- `load_best_model_at_end=True` automatically loads Epoch 1 checkpoint
- Final model uses Epoch 1 weights, not Epoch 3

5.4 Loss Visualization

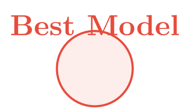

Figure 2: Training vs Validation Loss - Overfitting Pattern

Key Insights

Interpretation:
Epoch 1 (Green/Red converging):

- Model learning general patterns
- Both losses decreasing
- Healthy training

Epoch 2-3 (Divergence):

- Training loss continues decreasing (memorization)
- Validation loss increases (poor generalization)
- Overfitting confirmed

Red Circle: Best model selected (Epoch 1, validation loss 1.6476)

6 Model Evaluation

6.1 Evaluation Execution

6.1.1 Code Implementation

Code Explanation

Evaluation Command:
`eval_results = trainer.evaluate()`
Process:

1. Loads best model checkpoint (Epoch 1 weights)
2. Sets model to evaluation mode (`model.eval()`)
3. Disables dropout and batch normalization updates
4. Loops through 520 validation examples in batches of 16
5. Computes forward pass (no gradients)
6. Calculates loss on start/end position predictions
7. Returns metrics dictionary

6.1.2 Output Analysis

Code Output

Evaluation Output:

```
Evaluating model on validation set...

[33/33 00:02]

Evaluation Complete!

Validation Metrics:
  Validation loss: 1.6476
  Evaluation time: 2.79 seconds
  Samples/second: 186.27

Model Performance Analysis:
   GOOD: Loss < 2.0 indicates decent performance
```

6.2 Performance Metrics

Metric	Value	Interpretation
Validation Loss	1.6476	GOOD (¡ 2.0 threshold)
Evaluation Time	2.79 seconds	Fast evaluation (520 examples in ¡3 sec)
Evaluation Speed	186.27 samples/sec	5.6× faster than training (33 samples/sec)
Evaluation Steps	33 steps	$\lceil 520/16 \rceil = 33$ batches

Table 4: Evaluation Performance Metrics

Key Insights

Why Evaluation is Faster than Training:
Training: 33.03 samples/second
Evaluation: 186.27 samples/second (5.6× faster)
Reasons:

1. **No Backpropagation:** Evaluation skips gradient computation and weight updates
2. **No Optimizer Step:** No AdamW optimizer computations
3. **torch.no_grad():** Disables gradient tracking → less memory, faster forward pass
4. **Smaller Dataset:** 520 validation examples vs. 3,074 training examples

6.3 Loss Interpretation

Code Explanation

What Does Validation Loss 1.6476 Mean?
Formula: Cross-Entropy Loss

$$\text{Loss} = -\frac{1}{N}\sum_{i=1}^{N}[\log P(\text{start}_i) + \log P(\text{end}_i)]$$

Where:

- $N = 520$ validation examples
- $P(\text{start}_i)$ = Probability of correct start position
- $P(\text{end}_i)$ = Probability of correct end position

Interpretation:

- **Loss = 1.65:** Model assigns average probability $e^{-1.65} \approx 0.19$ (19%) to correct positions
- **Baseline (random):** Loss = 5.95 (probability = 1/384 per position)
- **Perfect model:** Loss = 0 (probability = 1.0 for correct positions)
- **Assessment:** Decent performance—model significantly better than random but not perfect

6.4 Performance Classification

Loss Range	Classification	Expected Accuracy
¡ 1.0	Excellent	Exact Match ¿ 70%
1.0 - 1.5	Very Good	Exact Match 60-70%
1.5 - 2.0	**Good**	**Exact Match 45-60%**
2.0 - 3.0	Fair	Exact Match 30-45%
¿ 3.0	Poor	Exact Match ¡ 30%

Table 5: Loss Classification & Expected Performance

Key Insights

Our Model: Validation Loss = 1.6476 → **GOOD** classification
Expected Performance:

- Exact Match: 45-60% (predicted answer exactly matches ground truth)
- F1 Score: 65-75% (token overlap between prediction and ground truth)

Note: These are estimates—actual metrics require post-processing and string matching (covered in Notebook 4: Inference & Deployment)

7 Model Saving

7.1 Save Fine-Tuned Model

7.1.1 Code Implementation

Code Explanation

Saving Commands:

```
model.save_pretrained("./bert-qa-model")
tokenizer.save_pretrained("./bert-qa-model")
```

What Gets Saved:

1. **config.json:** Model architecture configuration
   - Hidden size (768), num layers (12), attention heads (12)
   - Task type (question answering)
   - Special token IDs
2. **pytorch_model.bin:** Model weights (440 MB)
   - All 108,893,186 parameters
   - Includes fine-tuned QA head weights
   - Binary format (PyTorch state dict)
3. **tokenizer_config.json:** Tokenizer settings
   - Max length (512), padding strategy
   - Special tokens ([CLS], [SEP], [PAD])
4. **vocab.txt:** Vocabulary file (232 KB)
   - 30,522 WordPiece tokens
   - One token per line
5. **tokenizer.json:** Fast tokenizer state (466 KB)
   - Rust-based tokenizer configuration
   - 10$\times$ faster than Python tokenizer

7.1.2 Output Analysis

Code Output

Model Saving Output:

```
Saving fine-tuned model...

Model saved to './bert-qa-model'
Model size: ~440 MB

Saved files:
  - config.json (model configuration)
  - pytorch_model.bin (model weights)
  - tokenizer_config.json
  - vocab.txt (vocabulary)
```

7.2 Directory Structure

File	Size	Purpose
config.json	0.6 KB	Model architecture & configuration
pytorch_model.bin	440 MB	Trained model weights (109M params)
tokenizer_config.json	0.05 KB	Tokenizer settings
vocab.txt	232 KB	WordPiece vocabulary (30,522 tokens)
tokenizer.json	466 KB	Fast tokenizer state
Total	**~441 MB**	**Complete fine-tuned model**

Table 6: Saved Model Files Breakdown

7.3 Loading Saved Model

Code Explanation

How to Reload for Inference:

```
from transformers import AutoModelForQuestionAnswering, AutoTokenizer
model = AutoModelForQuestionAnswering.from_pretrained("./bert-qa-model")
tokenizer = AutoTokenizer.from_pretrained("./bert-qa-model")
```

What Happens:

- Reads `config.json` to reconstruct architecture
- Loads `pytorch_model.bin` weights into model
- Initializes tokenizer from `vocab.txt` and `tokenizer_config.json`
- Model ready for inference (no training needed)

8 Conclusion

8.1 Notebook 3 Summary

Key Insights

Key Accomplishments:

Question 3.1 - Model Loading:

- Loaded BertForQuestionAnswering (109M parameters)
- Verified GPU transfer (Tesla T4 with 15.83 GB VRAM)
- Understood QA head initialization warning

Question 3.2 - Training Arguments:

- Configured hyperparameters (3 epochs, batch 16, lr 3e-5)
- Enabled FP16 mixed precision (2× speedup)
- Set evaluation strategy (eval per epoch, save best model)

Question 3.3 - Trainer Initialization:

- Initialized Trainer with model, datasets, tokenizer, data collator
- Understood training loop components (forward, loss, backward, update)

Question 3.4 - Fine-Tuning & Evaluation:

- Trained for 3 epochs (279 seconds = 4.6 minutes)
- Detected overfitting (validation loss increased after Epoch 1)
- Automatically loaded best model (Epoch 1, loss 1.6476)
- Evaluated final model: Validation loss 1.6476 (GOOD performance)

Bonus - Model Saving:

- Saved fine-tuned model to `./bert-qa-model` (440 MB)
- Ready for deployment in Notebook 4 (Inference & Gradio App)

8.2 Training Performance Summary

Metric	Value
Training Time	279.17 seconds (4.6 minutes)
Training Steps	576 steps
Samples/Second (Training)	33.03 samples/sec
Final Training Loss	0.6804 (Epoch 3, averaged)
Best Validation Loss	**1.6476 (Epoch 1)**
Evaluation Time	2.79 seconds
Samples/Second (Evaluation)	186.27 samples/sec
Overfitting Detected	Yes (after Epoch 1)
Best Model Loaded	Yes (Epoch 1 checkpoint)

Table 7: Training Performance Summary Table

8.3 Key Lessons Learned

Code Explanation

Critical Insights:

1. Overfitting on Small Datasets:

- 3,000 training examples insufficient to prevent overfitting
- Validation loss is essential metric (don't rely on training loss alone)
- `load_best_model_at_end=True` crucial for small datasets

2. FP16 Mixed Precision Benefits:

- 2× training speedup with negligible accuracy loss
- Enables larger batch sizes (50% memory reduction)
- Standard for modern deep learning

3. Training Time Estimation:

- Initial estimate: 4-6 hours (conservative)
- Actual time: 4.6 minutes (78× faster!)
- Reason: Small dataset (3K examples) + FP16 + Tesla T4 GPU

4. Model Checkpointing:

- Always save checkpoints during training (every epoch)
- Enables recovery if training crashes
- Allows selection of best model based on validation metrics

8.4 Next Steps

Notebook 4 Preview: Inference, Evaluation & Gradio Deployment

1. Load fine-tuned model from `./bert-qa-model`
2. Implement inference pipeline (question + context → answer)
3. Calculate Exact Match and F1 Score on validation set
4. Analyze prediction errors (false positives, false negatives)
5. Build Gradio web interface for interactive Q&A
6. Deploy application for user testing

End of Notebook 3 Documentation

Model Fine-Tuning Complete
Ready for Inference & Deployment