```
!pip install torch transformers datasets evaluate peft accelerate pandas -U -q
print("Libraries installed in fresh environment.")
```

```
 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 89.9/89.9 kB 6.6 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 363.4/363.4 MB 3.1 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 13.8/13.8 MB 111.5 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 24.6/24.6 MB 87.5 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 883.7/883.7 kB 51.0 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 664.8/664.8 MB 2.1 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 211.5/211.5 MB 4.7 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 56.3/56.3 MB 39.2 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 127.9/127.9 MB 18.6 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 207.5/207.5 MB 5.8 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 21.1/21.1 MB 93.2 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 10.4/10.4 MB 120.2 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 491.2/491.2 kB 31.7 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 84.0/84.0 kB 6.9 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 411.1/411.1 kB 27.9 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 354.7/354.7 kB 26.3 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 13.1/13.1 MB 100.7 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 116.3/116.3 kB 9.9 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 183.9/183.9 kB 15.3 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 143.5/143.5 kB 11.9 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 194.8/194.8 kB 15.2 MB/s eta 0:00:00
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This behaviour is the source
google-colab 1.0.0 requires pandas==2.2.2, but you have pandas 2.2.3 which is incompatible.
gcsfs 2025.3.2 requires fsspec==2025.3.2, but you have fsspec 2024.12.0 which is incompatible.
Libraries installed in fresh environment.
```

```
# Step 1: Imports
import os
import torch
import numpy as np
import pandas as pd
import evaluate
from datasets import load_dataset, Dataset, DatasetDict
from transformers import (
    AutoTokenizer,
    AutoModelForSequenceClassification,
    DataCollatorWithPadding,
    TrainingArguments,
    Trainer,
    get_linear_schedule_with_warmup # Explicitly import scheduler if customizing
)
from peft import LoraConfig, TaskType, get_peft_model, PeftModel # Ensure PeftModel is imported for loading
from google.colab import drive
#from transformers.optimization import AdamW #  AdamW from transformers.optimization does not seem to work also
```

```
# Step 2: Mount Google Drive
try:
    drive.mount('/content/drive')
    DRIVE_MOUNTED = True
except Exception as e:
    print(f"Error mounting Google Drive: {e}")
    print("Proceeding without Google Drive. Models and results will not be saved persistently.")
    DRIVE_MOUNTED = False
```

```
Mounted at /content/drive
```

```
# Step 3: Configuration (Reduced for faster testing)
MODEL_CHECKPOINT = "roberta-base"
MAX_LENGTH = 256
BATCH_SIZE = 16 # Adjust based on GPU memory
NUM_LABELS = 4 # AGNEWS

# --- Configuration for Hyperparameter Sweep (Reduced for speed) ---
# Expand these lists to explore more configurations
sweep_ranks = [4, 8] # Example: Test rank 4
sweep_alphas = [8, 16] # Example: Test alpha 8 (often r*2)
sweep_target_modules = [["query", "value"], ["query", "key", "value"]] # Example: Test targeting query and value
sweep_learning_rates = [5e-5, 1e-4] # Example: Test one learning rate
sweep_lora_dropout = [0.1]
sweep_epochs = [1, 3] # Reduced for faster experimentation
```

```
print("--- Hyperparameter Sweep Configuration ---")
print(f"Ranks: {sweep_ranks}")
print(f"Alphas: {sweep_alphas}")
print(f"Target Modules: {sweep_target_modules}")
print(f"Learning Rates: {sweep_learning_rates}")
print(f"Dropout Values: {sweep_lora_dropout}")
print(f"Epochs: {sweep_epochs}")
print("-" * 40)

# --- Output Directory on Google Drive ---
# Define a base directory on Google Drive IF mounted
if DRIVE_MOUNTED:
    DRIVE_BASE_DIR = "/content/drive/MyDrive/Colab_Checkpoints/AGNEWS_LORA_Project"
    os.makedirs(DRIVE_BASE_DIR, exist_ok=True)
else:
    DRIVE_BASE_DIR = "./AGNEWS_LORA_Project_Local" # Save locally if Drive not mounted
    os.makedirs(DRIVE_BASE_DIR, exist_ok=True)

print(f"Checkpoints and results will be saved under: {DRIVE_BASE_DIR}")
```

```
--- Hyperparameter Sweep Configuration ---
Ranks: [4, 8]
Alphas: [8, 16]
Target Modules: [['query', 'value'], ['query', 'key', 'value']]
Learning Rates: [5e-05, 0.0001]
Dropout Values: [0.1]
Epochs: [1, 3]
----------------------------------------
Checkpoints and results will be saved under: /content/drive/MyDrive/Colab_Checkpoints/AGNEWS_LORA_Project
```

```
# Step 4: Load and Prepare Dataset (Using Hugging Face Hub version)
print("Loading AGNEWS dataset...")
try:
    # Attempt to load directly from Hugging Face Hub
    agnews_dataset = load_dataset("ag_news")
    # Optional: Create a validation split if desired (useful for seeing validation performance during training)
    train_splits = agnews_dataset['train'].train_test_split(test_size=0.1, seed=42)
    prepared_datasets = DatasetDict({
         'train': train_splits['train'],
         'validation': train_splits['test'], # Use this for eval during training
         'test': agnews_dataset['test']      # Keep original test set separate
     })
    # Simpler: Just use train/test split provided
    #prepared_datasets = agnews_dataset

except Exception as e:
    print(f"Failed to load dataset from Hugging Face Hub: {e}")
    print("Please ensure internet connection is available or provide data files manually.")
    # Add fallback to load from local files (e.g., CSV) if needed, similar to Kaggle example
    # For now, exit if dataset loading fails
    raise SystemExit("Dataset loading failed.")
```

```
Loading AGNEWS dataset...
```

```
# Step 5: Load Tokenizer
print("Loading tokenizer...")
tokenizer = AutoTokenizer.from_pretrained(MODEL_CHECKPOINT)
```

```
Loading tokenizer...
```

```
# Step 6: Define Tokenization Function
def tokenize_function(examples):
    # Handles text column possibly named 'text' or 'Description' etc.
    text_column = "text" # Default for HF ag_news
    if text_column not in examples:
        # Attempt common alternatives if 'text' isn't present
        possible_cols = [col for col in examples.keys() if isinstance(examples[col][0], str)]
        if possible_cols:
```

```
            text_column = possible_cols[0] # Use the first string column found
            print(f"Auto-detected text column: '{text_column}'")
        else:
            raise ValueError("Could not automatically detect text column in dataset.")

    return tokenizer(examples[text_column], truncation=True, padding="max_length", max_length=MAX_LENGTH)
```

```
# Step 7: Apply Tokenization and Formatting
print("Tokenizing datasets...")
tokenized_datasets = prepared_datasets.map(tokenize_function, batched=True)

# Determine the original text column name to remove it
text_col_to_remove = "text" # Default
if text_col_to_remove not in tokenized_datasets['train'].column_names:
    possible_cols = [col for col in tokenized_datasets['train'].features if isinstance(tokenized_datasets['train'][0][col], str)]
    if possible_cols:
        text_col_to_remove = possible_cols[0]

columns_to_remove = [text_col_to_remove] if text_col_to_remove in tokenized_datasets['train'].column_names else []
if not columns_to_remove:
     print(f"Warning: Could not find text column '{text_col_to_remove}' to remove after tokenization.")

tokenized_datasets = tokenized_datasets.remove_columns(columns_to_remove)
tokenized_datasets = tokenized_datasets.rename_column("label", "labels") # Assumes original label column is 'label'
tokenized_datasets.set_format("torch")
```

```
Tokenizing datasets...
Map: 100%          12000/12000 [00:01<00:00, 6826.40 examples/s]
```

```
# Step 8: Data Collator
data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
```

```
# Step 9: Define Compute Metrics Function
accuracy_metric = evaluate.load("accuracy")
def compute_metrics(eval_pred):
    logits, labels = eval_pred
    predictions = np.argmax(logits, axis=-1)
    return accuracy_metric.compute(predictions=predictions, references=labels)
```

```
# Step 10: Define Parameter Check Function
def check_params(model_to_check, limit=1_000_000):
    trainable_params = sum(p.numel() for p in model_to_check.parameters() if p.requires_grad)
    print(f"--> Trainable Parameters: {trainable_params:,}")
    if trainable_params > limit:
        print(f"--> WARNING: Exceeds {limit:,} parameter limit!")
        return False, trainable_params
    else:
        print(f"--> Within {limit:,} parameter limit.")
        return True, trainable_params
```

```
!pip show transformers
```

```
Name: transformers
Version: 4.51.3
Summary: State-of-the-art Machine Learning for JAX, PyTorch and TensorFlow
Home-page: https://github.com/huggingface/transformers
Author: The Hugging Face team (past and future) with the help of all our contributors (https://github.com/huggingface/transformers/graph
Author-email: transformers@huggingface.co
License: Apache 2.0 License
Location: /usr/local/lib/python3.11/dist-packages
Requires: filelock, huggingface-hub, numpy, packaging, pyyaml, regex, requests, safetensors, tokenizers, tqdm
Required-by: peft, sentence-transformers
```

```
# Step 11: Hyperparameter Sweep Loop (Corrected & Expanded for Epochs)
##############################################################################
#                       !!! IMPORTANT NOTE !!!                               #
# Ensure the following are defined in previous cells before running this:   #
# - Necessary libraries imported (torch, os, gc, transformers, datasets, peft, etc.) #
# - Global constants: MODEL_CHECKPOINT, BATCH_SIZE, NUM_LABELS              #
# - Sweep lists: sweep_epochs, sweep_ranks, sweep_alphas,                   #
#   sweep_target_modules, sweep_learning_rates, sweep_lora_dropout         #
# - DRIVE_BASE_DIR: Path to save outputs on Google Drive                    #
# - tokenized_datasets: Processed train/validation/test datasets           #
# - tokenizer: Loaded tokenizer                                             #
# - data_collator: Initialized DataCollatorWithPadding                     #
# - compute_metrics: Defined function for evaluation metrics               #
# - check_params: Defined function to check trainable parameters           #
# - id2label_map, label2id_map: Derived or defined label mappings         #
##############################################################################

import gc
import os
import torch
import numpy as np
from transformers import (
    AutoModelForSequenceClassification,
    TrainingArguments,
    Trainer,
    get_linear_schedule_with_warmup,
    DataCollatorWithPadding # Ensure imported if not globally
)
from peft import LoraConfig, get_peft_model, TaskType, PeftModel # Ensure imported

print("\n--- Starting Expanded Hyperparameter Sweep ---")

# Initialize results and details list BEFORE the loops
results = {}
all_run_details = []

# --- Loop through all hyperparameter combinations ---
for epochs_val in sweep_epochs: # Iterate through the number of epochs
    for r_val in sweep_ranks:
        for alpha_val in sweep_alphas:
            # Optional: Add logic here to skip non-standard alpha/rank pairs if desired
            # e.g., if alpha_val != 2 * r_val:
            #    print(f"Skipping alpha={alpha_val} for r={r_val}")
            #    continue
            for targets in sweep_target_modules:
                for lr in sweep_learning_rates:
                    for dropout_val in sweep_lora_dropout:

                        # --- Define unique key and output directory ---
                        config_key = f"r={r_val}_alpha={alpha_val}_targets={'_'.join(targets)}_lr={lr}_dropout={dropout_val}_epochs={epochs_
                        run_output_dir = os.path.join(DRIVE_BASE_DIR, config_key) # Assumes DRIVE_BASE_DIR is defined
                        try:
                           os.makedirs(run_output_dir, exist_ok=True)
                        except OSError as ose:
                           print(f"WARNING: Could not create directory {run_output_dir}: {ose}. Check path/permissions.")
                           # Decide whether to skip or try to continue
                           # results[config_key] = {"status": "fail", "error": f"Directory Creation Error: {ose}", "accuracy": 0, "params":
                           # continue # Uncomment to skip if directory fails

                        print(f"\n--- Running Configuration: {config_key} ---")
                        print(f"Output directory: {run_output_dir}")

                        # --- Initialize variables for this run's scope ---
                        training_args_for_run = None
                        temp_model = None
                        lora_model_for_run = None
                        optimizer = None
                        lr_scheduler = None
                        trainer_for_run = None
                        trainable_params = 0 # Default

                        try:
                            # --- 1. Create LoRA Config ---
                            lora_config = LoraConfig(
                                r=r_val,
                                lora_alpha=alpha_val,
                                target_modules=targets,
```

```
    lora_dropout=dropout_val,
    bias="none",
    task_type=TaskType.SEQ_CLS
)

# --- Calculate steps per epoch (needed for scheduler/logging/eval) ---
if "train" not in tokenized_datasets:
    raise ValueError("tokenized_datasets['train'] not found.")
train_dataset_size = len(tokenized_datasets["train"])
if train_dataset_size == 0:
    raise ValueError("Training dataset is empty.")

grad_accum_steps = 1 # Assuming 1, adjust if using gradient accumulation
num_gpus = torch.cuda.device_count() if torch.cuda.is_available() else 1
effective_batch_size = BATCH_SIZE * grad_accum_steps * num_gpus
steps_per_epoch = max(1, train_dataset_size // effective_batch_size)
total_training_steps = steps_per_epoch * epochs_val
print(f"Calculated approx steps per epoch: {steps_per_epoch}")
print(f"Total training steps: {total_training_steps}")

# --- 2. Define TrainingArguments ---
print("Defining TrainingArguments...")
training_args_for_run = TrainingArguments(
    output_dir=run_output_dir,
    num_train_epochs=epochs_val, # Use epoch value from loop
    learning_rate=lr,           # Use learning rate from loop
    per_device_train_batch_size=BATCH_SIZE,
    per_device_eval_batch_size=BATCH_SIZE * 2,
    weight_decay=0.01,
    # Evaluation/Saving/Logging Strategy (Example: Evaluate/Save every epoch)
    eval_strategy="epoch",
    save_strategy="epoch",
    # OR use step-based:
    # eval_strategy="steps",
    # eval_steps=steps_per_epoch, # Evaluate every epoch equivalent
    # save_strategy="steps",
    # save_steps=steps_per_epoch, # Save every epoch equivalent
    logging_strategy="steps",   # Log more frequently
    logging_steps=max(1, steps_per_epoch // 10), # Log ~10 times per epoch
    load_best_model_at_end=True, # Important for finding best epoch checkpoint
    metric_for_best_model="accuracy", # Make sure compute_metrics returns "accuracy"
    greater_is_better=True,
    save_total_limit=1, # Save only the best checkpoint to save space
    report_to="none",  # Disable external reporting unless configured (like wandb)
    fp16=torch.cuda.is_available(), # Use mixed precision if available
)
print("Training Arguments defined successfully.")

# --- 3. Create Model and Apply PEFT ---
print("Loading base model and applying PEFT...")
# Ensure id2label_map and label2id_map are defined from earlier steps
if 'id2label_map' not in locals() or 'label2id_map' not in locals():
     print("Warning: id2label_map or label2id_map not found. Using default numerical labels.")
     # Fallback or raise error depending on strictness needed
     id2label_map = {i: f"LABEL_{i}" for i in range(NUM_LABELS)}
     label2id_map = {v: k for k, v in id2label_map.items()}

temp_model = AutoModelForSequenceClassification.from_pretrained(
    MODEL_CHECKPOINT,
    num_labels=NUM_LABELS,
    id2label=id2label_map,
    label2id=label2id_map
)
lora_model_for_run = get_peft_model(temp_model, lora_config)
print(f"Model created with LoRA applied. Trainable parameters:")
lora_model_for_run.print_trainable_parameters()


# --- 4. Check parameters ---
within_limit, trainable_params = check_params(lora_model_for_run) # Assumes function `check_params` exists
if not within_limit:
    raise ValueError(f"Parameter limit exceeded ({trainable_params} > limit)") # Raise error to be caught below

# --- 5. Define Optimizer and Scheduler ---
print("Defining Optimizer and Scheduler...")
optimizer = torch.optim.AdamW(lora_model_for_run.parameters(), lr=lr)
num_warmup_steps = int(0.1 * total_training_steps) # 10% warmup
```

```
        lr_scheduler = get_linear_schedule_with_warmup(
            optimizer=optimizer,
            num_warmup_steps=num_warmup_steps,
            num_training_steps=total_training_steps
        )
        print("Optimizer and Scheduler defined.")

        # --- 6. Initialize Trainer ---
        print("Initializing Trainer...")
         # Use validation set for evaluation during training if it exists
        eval_dataset = tokenized_datasets.get("validation")
        if eval_dataset is None:
            print("Warning: No 'validation' dataset found. Using 'test' set for evaluation during training.")
            eval_dataset = tokenized_datasets.get("test") # Fallback to test set if no validation set
        if eval_dataset is None:
             raise ValueError("Neither 'validation' nor 'test' dataset found for evaluation.")


        trainer_for_run = Trainer(
            model=lora_model_for_run,
            args=training_args_for_run,
            train_dataset=tokenized_datasets["train"],
            eval_dataset=eval_dataset, # Use validation (or test) for checkpoing evaluation
            tokenizer=tokenizer,
            data_collator=data_collator,
            compute_metrics=compute_metrics,
            optimizers=(optimizer, lr_scheduler)
        )
        print("Trainer initialized.")

        # --- 7. Train and Evaluate ---
        print(f"Starting training for {config_key}...")
        train_result = trainer_for_run.train()
        print(f"Training completed.")

        # Since load_best_model_at_end=True, the trainer has loaded the best checkpoint.
        # Evaluate this best model on the *final test set*.
        print("Evaluating best model on the final test set...")
        if "test" not in tokenized_datasets:
           raise ValueError("tokenized_datasets['test'] not found for final evaluation.")

        final_eval_results = trainer_for_run.evaluate(eval_dataset=tokenized_datasets["test"])
        accuracy = final_eval_results.get("eval_accuracy", 0) # Get accuracy on test set

        # Store results
        results[config_key] = {"status": "success", "accuracy": accuracy, "params": trainable_params}
        all_run_details.append({
            "config": config_key, "r": r_val, "alpha": alpha_val, "targets": '_'.join(targets),
            "lr": lr, "dropout": dropout_val, "epochs": epochs_val,
            "params": trainable_params, "accuracy": accuracy, "output_dir": run_output_dir
        })
        print(f"Result for {config_key}: Final Test Accuracy = {accuracy:.4f}")

        # --- 8. Save Best Adapter (Optional but Recommended) ---
        # The best adapter is already loaded, just save it from the current model state
        best_adapter_save_path = os.path.join(run_output_dir, "best_adapter") # Simplified name
        trainer_for_run.model.save_pretrained(best_adapter_save_path)
        # Also save the tokenizer and potentially config for easier reloading
        tokenizer.save_pretrained(best_adapter_save_path)
        # trainer_for_run.model.config.save_pretrained(best_adapter_save_path) # Save base model config if needed

        print(f"Best adapter, tokenizer, config saved to: {best_adapter_save_path}")

    except Exception as e:
        print(f"ERROR during run {config_key}: {e}")
        import traceback
        traceback.print_exc() # Print detailed traceback for debugging

        # Attempt to get parameter count even on failure, if model was created
        error_params = 'N/A'
        if 'lora_model_for_run' in locals() and lora_model_for_run is not None:
            try:
                _, error_params = check_params(lora_model_for_run, limit=float('inf')) # Use check_params if defined
            except: pass

        results[config_key] = {"status": "fail", "error": str(e), "accuracy": 0, "params": error_params}
```

```
                        # Optionally add failed run details to all_run_details if desired for analysis

                    finally:
                        # --- Cleanup resources for this iteration ---
                        print(f"Cleaning up resources for run {config_key}...")
                        # Delete objects in reverse order of creation (roughly)
                        del trainer_for_run
                        del optimizer
                        del lr_scheduler
                        del lora_model_for_run
                        del temp_model
                        del training_args_for_run
                        # Force garbage collection and empty CUDA cache
                        gc.collect()
                        if torch.cuda.is_available():
                            torch.cuda.empty_cache()
                        print("-" * 60)


print("\n--- Completed Hyperparameter Sweep ---")

# --- The rest of the script follows ---
# Step 12: Find Best Configuration
# Step 13: Load Best Overall Model and Save Final Adapter
# Step 14: Display Results Summary
# ...
```

```
--- Starting Expanded Hyperparameter Sweep ---

--- Running Configuration: r=4_alpha=8_targets=query_value_lr=5e-05_dropout=0.1_epochs=1 ---
Output directory: /content/drive/MyDrive/Colab_Checkpoints/AGNEWS_LORA_Project/r=4_alpha=8_targets=query_value_lr=5e-05_dropout=0.1_e
Calculated approx steps per epoch: 6750
Total training steps: 6750
Defining TrainingArguments...
TrainingArguments defined successfully.
Loading base model and applying PEFT...
Some weights of RobertaForSequenceClassification were not initialized from the model checkpoint at roberta-base and are newly initial
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
<ipython-input-30-32dffcd20c1f>:174: FutureWarning: `tokenizer` is deprecated and will be removed in version 5.0.0 for `Trainer.__ini
  trainer_for_run = Trainer(
No label_names provided for model class `PeftModelForSequenceClassification`. Since `PeftModel` hides base models input arguments, if
Model created with LoRA applied. Trainable parameters:
trainable params: 741,124 || all params: 125,389,832 || trainable%: 0.5911
--> Trainable Parameters: 741,124
--> Within 1,000,000 parameter limit.
Defining Optimizer and Scheduler...
Optimizer and Scheduler defined.
Initializing Trainer...
Trainer initialized.
Starting training for r=4_alpha=8_targets=query_value_lr=5e-05_dropout=0.1_epochs=1...
```

[6750/6750 09:23, Epoch 1/1]

Epoch	Training Loss	Validation Loss	Accuracy
1	0.255700	0.244946	0.918250

```
Training completed.
Evaluating best model on the final test set...
```

[238/238 00:14]

```
Result for r=4_alpha=8_targets=query_value_lr=5e-05_dropout=0.1_epochs=1: Final Test Accuracy = 0.9189
Best adapter, tokenizer, config saved to: /content/drive/MyDrive/Colab_Checkpoints/AGNEWS_LORA_Project/r=4_alpha=8_targets=query_valu
Cleaning up resources for run r=4_alpha=8_targets=query_value_lr=5e-05_dropout=0.1_epochs=1...
------------------------------------------------------------

--- Running Configuration: r=4_alpha=8_targets=query_value_lr=0.0001_dropout=0.1_epochs=1 ---
Output directory: /content/drive/MyDrive/Colab_Checkpoints/AGNEWS_LORA_Project/r=4_alpha=8_targets=query_value_lr=0.0001_dropout=0.1_
Calculated approx steps per epoch: 6750
Total training steps: 6750
Defining TrainingArguments...
TrainingArguments defined successfully.
Loading base model and applying PEFT...
Some weights of RobertaForSequenceClassification were not initialized from the model checkpoint at roberta-base and are newly initial
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
<ipython-input-30-32dffcd20c1f>:174: FutureWarning: `tokenizer` is deprecated and will be removed in version 5.0.0 for `Trainer.__ini
  trainer_for_run = Trainer(
No label_names provided for model class `PeftModelForSequenceClassification`. Since `PeftModel` hides base models input arguments, if
Model created with LoRA applied. Trainable parameters:
trainable params: 741,124 || all params: 125,389,832 || trainable%: 0.5911
--> Trainable Parameters: 741,124
--> Within 1,000,000 parameter limit.
Defining Optimizer and Scheduler...
Optimizer and Scheduler defined.
Initializing Trainer...
Trainer initialized.
```

```
# Step 12: Find Best Configuration After Sweep
print("\n--- Sweep Finished ---")

successful_runs = {k: v for k, v in results.items() if v.get("status") == "success"} # Use .get() for safety

if successful_runs:
    # Find best config based on accuracy
    best_config_key = max(successful_runs, key=lambda k: successful_runs[k]["accuracy"])
    best_accuracy = successful_runs[best_config_key]["accuracy"]
    best_params = successful_runs[best_config_key]["params"]
    # Find the corresponding details from all_run_details
    best_run_details = next((run for run in all_run_details if run["config"] == best_config_key), None) # Added default None

    print(f"Best configuration found: {best_config_key}")
    print(f"Best Accuracy: {best_accuracy:.4f}")
    print(f"Trainable Parameters: {best_params:,}")

    # Step 13: Load the Overall Best Model and Save Final Adapter
    if best_run_details and DRIVE_MOUNTED:
        best_run_output_dir = best_run_details["output_dir"]
        # Path to the adapter saved at the end of the best run
        best_adapter_path = os.path.join(best_run_output_dir, "best_adapter_for_run") # Corrected path variable name

        if os.path.exists(best_adapter_path):
```

```
            print(f"\nLoading best adapter from: {best_adapter_path}")
            try:
                # Load the base model first
                base_model = AutoModelForSequenceClassification.from_pretrained(
                    MODEL_CHECKPOINT, num_labels=NUM_LABELS
                )
                # Load the LoRA adapter onto the base model
                best_lora_model = PeftModel.from_pretrained(base_model, best_adapter_path) # Correct variable name
                best_lora_model.to('cuda' if torch.cuda.is_available() else 'cpu') # Move model to device
                print("Successfully loaded best LoRA model.")

                # Define path for the final overall best adapter
                final_best_adapter_dir = os.path.join(DRIVE_BASE_DIR, "final_best_adapter") # Correct variable name
                os.makedirs(final_best_adapter_dir, exist_ok=True)

                # Save the final best adapter weights
                best_lora_model.save_pretrained(final_best_adapter_dir)
                print(f"Final best adapter saved to: {final_best_adapter_dir}")

            except Exception as e:
                print(f"Error loading or saving the final best model: {e}")
        else:
             print(f"Could not find best adapter files at expected location: {best_adapter_path}") # Correct variable name
             print("Skipping final model loading and saving.")

    elif not DRIVE_MOUNTED:
        print("\nSkipping final model loading and saving as Google Drive was not mounted.")
    # Added case where best_run_details might be None even if successful_runs has items (shouldn't happen with current logic but safe)
    elif not best_run_details:
         print("\nCould not find details for the best run. Skipping final model loading.")

else:
    print("No successful runs completed in the hyperparameter sweep.")


# Step 14: Display Results Summary (Example)
print("\n--- Run Summary ---")
if all_run_details:
    # Ensure 'accuracy' and 'params' columns exist before sorting/displaying
    summary_df = pd.DataFrame(all_run_details)
    display_cols = ['config', 'params', 'accuracy']
    # Filter columns to only those present in the DataFrame
    display_cols = [col for col in display_cols if col in summary_df.columns]
    if 'accuracy' in display_cols:
      print(summary_df[display_cols].sort_values(by='accuracy', ascending=False))
    elif display_cols: # If accuracy column is missing, print available cols
        print(summary_df[display_cols])
    else:
        print("No relevant columns found in run details.")

else:
    print("No run details available.")


print("\nCode execution finished.")
```

```
     TrainingArguments defined successfully
```

```
# KAGGLE SUBMISSION INFERENCE (Adapted for Google Drive & Corrected)
import pandas as pd
import pickle # Import the pickle library
from datasets import Dataset # Ensure Dataset is imported
import os
import torch
import numpy as np
from transformers import Trainer, TrainingArguments, AutoTokenizer, DataCollatorWithPadding
# Ensure 'PeftModel', 'AutoModelForSequenceClassification' were imported earlier
# Ensure 'best_lora_model' variable holds your trained PEFT model loaded onto the base model.
# Ensure 'tokenizer' and 'data_collator' are defined from training or reloaded
# Ensure BATCH_SIZE and MAX_LENGTH constants are defined from training

print("--- KAGGLE SUBMISSION INFERENCE START ---")

# --- Mount Google Drive if not already mounted ---
from google.colab import drive
try:
    if not os.path.exists("/content/drive/MyDrive"):
      print("Mounting Google Drive...")
```

```
      drive.mount('/content/drive')
      print("Google Drive mounted.")
    else:
      print("Google Drive already mounted.")
except Exception as e:
    print(f"Error mounting Google Drive: {e}")
    raise SystemExit("Google Drive mounting failed, cannot access test data.")

# --- Load the Unlabelled Test Data from PKL file ---
test_pkl_path = "/content/drive/MyDrive/kaggle/input/deep-learning-spring-2025-project-2/test_unlabelled.pkl"
print(f"Attempting to load test data from: {test_pkl_path}")

try:
    if not os.path.exists(test_pkl_path):
        raise FileNotFoundError(f"File not found at the specified Google Drive path: {test_pkl_path}")

    # Load the data (which we know results in a Dataset object)
    test_dataset_hf = pd.read_pickle(test_pkl_path) # Load directly into the final variable name
    print(f"Loaded test data. Object type: {type(test_dataset_hf)}")

    # Verify it's a Dataset and has the 'text' feature
    if not isinstance(test_dataset_hf, Dataset):
         raise TypeError(f"Loaded data is not a Dataset object, but {type(test_dataset_hf)}")
    if 'text' not in test_dataset_hf.features:
        raise ValueError(f"Loaded dataset does not contain the required 'text' feature. Features found: {test_dataset_hf.features}")

    print("\nTest Dataset Info:")
    print(test_dataset_hf)
    print("\nTest Dataset Features:")
    print(test_dataset_hf.features)
    print("\nFirst 5 examples:")
    print(test_dataset_hf[0:5]) # Correct way to view head

except FileNotFoundError as fnf_e:
    print(f"Error: {fnf_e}")
    print("Please double-check the file path and ensure Google Drive is mounted correctly.")
    raise SystemExit("Failed to load test data due to FileNotFoundError.")
except Exception as e:
    print(f"An unexpected error occurred during data loading: {e}")
    raise SystemExit("Failed to load or validate test data.")

# --- Set Text Column Name ---
text_column_name = "text" # Confirmed from features

# --- Preprocess/Tokenize the Test Data ---
# Define tokenization function using the correct text column name
def tokenize_for_inference(examples):
    return tokenizer(examples[text_column_name], truncation=True, padding="max_length", max_length=MAX_LENGTH)

print(f"\nTokenizing test data using column: '{text_column_name}'...")
# Apply map directly to the loaded dataset
tokenized_test_kaggle = test_dataset_hf.map(tokenize_for_inference, batched=True, remove_columns=[text_column_name]) # Remove original text

# Set format for PyTorch
tokenized_test_kaggle.set_format("torch")
print(f"Test data tokenized. Columns kept: {tokenized_test_kaggle.column_names}") # Show remaining columns

# --- Generate Predictions ---
print("\nGenerating predictions on Kaggle test set...")
# Ensure the model is on the correct device (GPU if available)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# Make sure best_lora_model is loaded and moved to device
if 'best_lora_model' not in locals():
     raise NameError("Variable 'best_lora_model' not defined. Load the model before inference.")
best_lora_model.to(device)
best_lora_model.eval() # Set model to evaluation mode

# Create a temporary Trainer for prediction
temp_training_args = TrainingArguments(
    output_dir="/content/temp_preds", # Use a writable path in Colab
    per_device_eval_batch_size=BATCH_SIZE * 2,
    fp16=torch.cuda.is_available(),
    report_to="none",
    logging_dir=None, # Disable logging for prediction
)

# Ensure data_collator is defined
```

```python
if 'data_collator' not in locals():
    print("Warning: 'data_collator' not defined. Defining default DataCollatorWithPadding.")
    if 'tokenizer' not in locals():
         raise NameError("Cannot define data_collator because 'tokenizer' is not defined.")
    data_collator = DataCollatorWithPadding(tokenizer=tokenizer)


pred_trainer = Trainer(
    model=best_lora_model,
    args=temp_training_args,
    tokenizer=tokenizer,
    data_collator=data_collator,
)

print("Running prediction...")
test_predictions = pred_trainer.predict(tokenized_test_kaggle)
predicted_logits = test_predictions.predictions

# --- Generate Predicted Label IDs (0-3) --- Corrected mapping
predicted_label_ids = np.argmax(predicted_logits, axis=-1) # Direct 0-3 labels

# --- Generate Sequential IDs ---
num_predictions = len(predicted_label_ids)
print(f"Generated {num_predictions} predictions.")
```