Homework 2
CSC 277 / 477
End-to-end Deep Learning
Fall 2024

Henry Yin - hyi1n2@u.rochester.edu

Deadline: 10/18/2024

Instructions

Your homework solution must be typed and prepared in LaTeX. It must be output to PDF format. To use LaTeX, we suggest using http://overleaf.com, which is free.

Your submission must cite any references used (including articles, books, code, websites, and personal communications). All solutions must be written in your own words, and you must program the algorithms yourself. **If you do work with others, you must list the people you worked with.** Submit your solutions as a PDF to Blackboard.

Your programs must be written in Python. The relevant code should be in the PDF you turn in. If a problem involves programming, then the code should be shown as part of the solution. One easy way to do this in LaTeX is to use the verbatim environment, i.e., \begin{verbatim} YOUR CODE \end{verbatim}.

Problem 1 - LoRA (22 Points)

Fine-tuning large pre-trained language models for downstream tasks is common in NLP but can be computationally expensive due to the need to update all model parameters. LoRA (LOw-Rank Adaptation) offers a more efficient alternative by only adjusting low-rank components instead of the full parameter set.

Specifically, for a pre-trained weight matrix $W_0 \in \mathbb{R}^{d\times k}$, the model update is represented with a low-rank decomposition $W_0 + \Delta W = W_0 + BA$, where $B \in \mathbb{R}^{d\times r}, A \in \mathbb{R}^{r\times k}$, and the rank $r \ll \min(d, k)$. During training, W_0 is frozen, while A and B are trainable. For $h = W_0x$, the modified forward pass yields: $h = W_0x + \Delta W x = W_0x + BAx$, as shown in Fig. **??**. In this problem, you'll fine-tune a pre-trained language model using LoRA for sentiment classification.

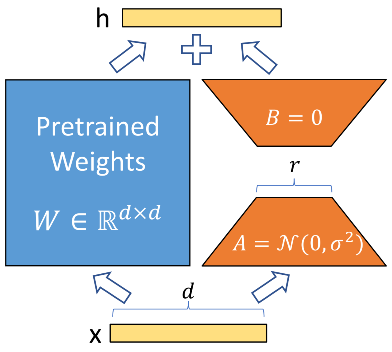
h
Pretrained Weights
$W \in \mathbb{R}^{d\times d}$
$B = 0$
r
$A = \mathcal{N}(0, \sigma^2)$
d
x

Figure 1: Illustration of LoRA. Only A and B are trainable.

Part 1: Understanding LoRA

Part 1.1: Analyzing Trainable Parameters (2 Points)

Given the description, determine the ratio of trainable parameters to the total parameters when applying LoRA to a weight matrix $W_0 \in \mathbb{R}^{d\times k}$ with the following dimensions: $d = 1024$, $k = 1024$, and a low-rank approximation of $r = 8$.
Deliverable: Provide the formula/expression for this ratio.

Answer:
Total parameters in the original weight matrix W_0 is:

$$\text{Total Parameters} = d \times k = 1024 \times 1024 = 1,048,576$$

LoRA introduces low rank decomposition where the original weight matrix W_0 is approximated by to smaller matrices. One with size $W_A \in \mathbb{R}^{d\times r}$ and one of size $W_B \in \mathbb{R}^{r\times k}$.

The number of trainable parameters in LoRA then:

$$\text{Trainable Parameters} = (d \times r) + (r \times k) = (1024 \times 8) + (8 \times 1024) = 8192 + 8192 = 16,384$$

$$\frac{\text{Trainable Parameters}}{\text{Total Parameters}} = \frac{16.384}{1,048,576} \approx 0.015625$$

The ratio of trainable parameters to total parameters is 0.0156 (i.e. 1.56%)

Part 1.2: LoRA Integration in Transformer Models (2 Points)

Read the following paragraphs in the LoRA paper:

- `Section 1 - Introduction`; specifically `Terminologies and Conventions`
- `Section 4.2 - Applying LoRA to Transformer`
- `Section 5.1 - Baselines`; specifically `LoRA.`

Question: For a Transformer architecture model, where is LoRA typically injected? (Options: query/key/value/output projection matrices)

Answer:
Query and Value projection matrices.

Part 2: Fine-Tuning for Sentiment Classification

Part 2.1: Fine-Tuning Without LoRA (6 Points)

Hugging Face provides a user-friendly framework for natural language processing tasks. If you haven't used it before, this is a great opportunity to get familiar with it.

1. Follow the Hugging Face fine-tuning tutorial and install the necessary packages to set up the components required for training: `transformers` (required), `datasets`(required), and `evaluate` (optional, depending on your implementation).
2. Fine-tune the `roberta-base` model on the Tweet Eval Sentiment dataset. Make sure to set the `num_labels` parameter correctly. You can load the dataset using: `datasets.load_dataset("tweet_eval", name="sentiment")`.
3. For training settings, fine-tune the model for **1 epoch** using Hugging Face's PyTorch Trainer. Default parameters like learning rate can be used. For batch size, adjust based on your computational resources. Estimated computational cost with a batch size of 16: GPU memory of 6.6 G and runtime within 10 Min. CPU runtime: 1 H.
4. Record the following metrics: **(a)** Number of *total* and *trainable* parameters; **(b)** Training time; **(c)** GPU memory usage during training (optional but encouraged); **(d)** Performance on the test set (Accuracy, F1 score, and loss).

If implemented correctly, the accuracy score on the test set should be above 0.6.

Deliverable:

1. Recorded metrics as described in Step 4 in LaTeX table(s).
2. Your code implementation.

Answer:
Metrics Table Without Using LoRA:

Metric	**Value**
Total Parameters	125535750
Trainable Parameters	125535750
Training Time (seconds)	870.40
GPU Allocated Memory (GB)	1.420
GPU Reserved Memory (GB)	9.086
Max GPU Allocated Memory (GB)	8.819
Test Accuracy	0.70612
Test F1 Score	0.70453
Test Loss	0.65381

Table 1: Model Performance Metrics Without LoRA Fine-Tuning

Code Implementation:

```
import time
import torch
import wandb
from datasets import load_dataset
from transformers import RobertaTokenizer,
    RobertaForSequenceClassification, Trainer, TrainingArguments
from sklearn.metrics import accuracy_score, f1_score

wandb.init(project="sentiment-classification", config={
    "model": "roberta-base",
    "dataset": "Tweet Eval Sentiment",
    "epochs": 1,
    "batch_size": 16,
})

dataset = load_dataset("tweet_eval", name="sentiment")
```

```
tokenizer = RobertaTokenizer.from_pretrained("roberta-base")

def tokenize_function(examples):
    return tokenizer(examples["text"], padding="max_length", truncation=True)

tokenized_datasets = dataset.map(tokenize_function, batched=True)

model = RobertaForSequenceClassification.from_pretrained("roberta-base", num_labels=3)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

# Record total and trainable parameters
total_params = sum(p.numel() for p in model.parameters())
trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
wandb.config.update({
    "total_params": total_params,
    "trainable_params": trainable_params
})

training_args = TrainingArguments(
    output_dir="./results",
    num_train_epochs=1,
    per_device_train_batch_size=16,
    evaluation_strategy="epoch",
    save_strategy="epoch",
    fp16=True,
    save_total_limit=1,
    load_best_model_at_end=True,
    report_to="wandb",
)

# Define compute_metrics function calculate accuracy and F1 score
def compute_metrics(p):
    preds = p.predictions.argmax(-1)
    accuracy = accuracy_score(p.label_ids, preds)
    f1 = f1_score(p.label_ids, preds, average="weighted")
    return {"accuracy": accuracy, "f1": f1}

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_datasets["train"],
    eval_dataset=tokenized_datasets["validation"],
    compute_metrics=compute_metrics,
)

start_time = time.time()

trainer.train()

training_time = time.time() - start_time
wandb.log({"training_time_seconds": training_time})

if torch.cuda.is_available():
    allocated_memory = torch.cuda.memory_allocated() / (1024 ** 3)
    reserved_memory = torch.cuda.memory_reserved() / (1024 ** 3)
    max_allocated = torch.cuda.max_memory_allocated() / (1024 ** 3)
    print(f"GPU allocated memory: {allocated_memory:.3f} GB")
    print(f"GPU reserved memory: {reserved_memory:.3f} GB")
    print(f"Max GPU allocated memory: {max_allocated:.3f} GB")


metrics = trainer.evaluate(tokenized_datasets["test"])
wandb.log({
    "test_accuracy": metrics["eval_accuracy"],
    "test_f1_score": metrics["eval_f1"],
    "test_loss": metrics["eval_loss"],
})

wandb.finish()
```

Part 2.2: Fine-Tuning With LoRA using PEFT (4 Points)

The PEFT (Parameter-Efficient Fine-Tuning) repository provides efficient methods for adapting models, including LoRA, and integrates with Hugging Face. In this section, you'll fine-tune RoBERTa with LoRA using PEFT.

1. Copy your code from Part 2.1 (fine-tuning without LoRA).
2. Read the PEFT quick tour. Prepare the model for fine-tuning with LoRA with the following settings: Set the rank to 8; Adjust the `inference_mode` and `task_type` parameters to appropriate values; Keep all other parameters as default (only adjust the three mentioned).
3. Apply the same training recipe as in Part 2.1 and fine-tune RoBERTa with LoRA.

Deliverable:

1. Recorded Metrics as described in Part 2.1 Step 4 in LaTeX table(s)
2. Your code snippet of the implementation of LoRA into the model.

Answer:

Metrics Table After Using LoRA:

Metric	**Value**
Total Parameters	125535750
Trainable Parameters	887811
Training Time (seconds)	767.65
GPU Allocated Memory (GB)	0.491
GPU Reserved Memory (GB)	7.240
Max GPU Allocated Memory (GB)	7.063
Test Accuracy	0.69945
Test F1 Score	0.69857
Test Loss	0.66546

Table 2: Model Performance Metrics for LoRA Fine-Tuning

Code Implementation:
Every things remains the same in the previous question with the following amendements of training RoberTa-base model with LoRA implementation.

```
# Import PEFT-related modules
from peft import LoraConfig, get_peft_model, TaskType
```

```
# Set up PEFT with LoRA
peft_config = LoraConfig(
    task_type=TaskType.SEQ_CLS,
    inference_mode=False,        # training, not inferring
    r=8,
    lora_alpha=32,               # Default scaling factor for LoRA
    lora_dropout=0.1
)

# Convert the RoberTa-base model to a LoRA model using PEFT
model = get_peft_model(model, peft_config)
```

Part 2.3: Comparison and Analysis (3 Points)

Now that you've fine-tuned the RoBERTa model with and without LoRA, compare their performance using the following criteria:

1. **Efficiency**: Compare total parameters, trainable parameters, GPU memory usage (optional), and training time.
2. **Performance**: Compare test set results in terms of accuracy, F1 score, and loss.
3. Consider other aspects: drawing inspiration from the LoRA paper `Section 4.2 - APPLYING LORA TO TRANSFORMER - Practical Benefits and Limitations.`

Deliverable: Provide concise answers to these three aspects, each with one or two sentences, to summarize your findings and insights.

Answer:

1. **Efficiency**: LoRA reduces the number of trainable parameters but increases memory usage and training time.
2. **Performance**: The model without LoRA achieved better performance in terms of accuracy, F1 score, and loss compared to the LoRA fine-tuned model.
3. **Other Aspects**: LoRA is beneficial in scenarios requiring memory-efficient fine-tuning, though it did not perform as well for this particular sentiment classification task.

Part 3: Influence of Model Size (5 Points)

In this part, you will replicate the experiment from Part 2, but using a much smaller model, TinyBERT. Fine-tune the model both with and without LoRA. Simply replace the model name in your previous code, keeping the same training settings and logging metrics. The expected accuracy should exceed 0.50.

Deliverable:

1. Provide the same metrics (with and without LoRA) as in Part 2 in LaTeX table(s).
2. Compare the performance of your models with a naive predictor that always guesses the majority class. Which one is better?
3. Reflect on your Part 2 analysis. Determine if the same observations apply to this smaller model and discuss factors that could explain differences (if any).

Answer:

Metric	Value
Total Parameters	14351187
Trainable Parameters	14351187
Training Time (seconds)	49.36
GPU Allocated Memory (GB)	0.178
GPU Reserved Memory (GB)	0.449
Max GPU Allocated Memory (GB)	0.345
Test Accuracy	0.6734
Test F1 Score	0.67359
Test Loss	0.71558

Table 3: TinyBERT without LoRA Fine-Tuning

Metric	Value
Total Parameters	14351187
Trainable Parameters	40875
Training Time (seconds)	47.64
GPU Allocated Memory (GB)	0.070
GPU Reserved Memory (GB)	0.246
Max GPU Allocated Memory (GB)	0.208
Test Accuracy	0.52979
Test F1 Score	0.42741
Test Loss	0.94995

Table 4: TinyBERT with LoRA Fine-Tuning

The TinyBERT fine-tuned model significantly outperforms the naive predictor that always guesses the majority class. While the naive predictor achieves an accuracy of 48.33% and an F1 score of 31.50

- **Efficiency**: Similar to the larger model, **LoRA** for TinyBERT reduces the number of trainable parameters, but the overall memory usage and training time increase. Since TinyBERT is much smaller, the effect of LoRA on memory efficiency and training time is less pronounced compared to a larger model like RoBERTa.
- **Performance**: As with the RoBERTa model, the **TinyBERT model without LoRA** achieves better performance in terms of accuracy and F1 score compared to the LoRA fine-tuned version. This indicates that fine-tuning all parameters of

a smaller model like TinyBERT may still be more effective than using parameter-efficient techniques like LoRA.

- **Other Aspects**: While **LoRA** is generally useful for reducing the computational cost of fine-tuning large models, its benefits are less evident with smaller models like TinyBERT, where the overhead introduced by LoRA can outweigh the gains from reducing the number of trainable parameters. For this sentiment classification task, fine-tuning the entire TinyBERT model directly provides better results.

In summary, the same general observations from Part 2 apply to TinyBERT, though the impact of LoRA on efficiency and performance is less pronounced due to the model's smaller size. The overall performance suggests that fine-tuning all parameters in smaller models might be more effective than applying LoRA.

Problem 2 - Using Pretrained-Model Embedding (20 Points)

Pretrained models help transfer knowledge to new tasks by generating meaningful data representations, which can be used for downstream tasks like classification. In this problem, you'll use pretrained models to generate embeddings for the Visual Question Answering (VQA) task. The task is simplified into a classification problem, where the model must choose the correct answer based on an image and a question. We'll use the DAQUAR dataset, available here, but will replace the original files with new versions (`new_data_train.csv`, `new_data_val.csv`, `new_data_test.csv`) that reduce the answer space to 30 classes.

To solve the task, you'll need two encoders: one for images and one for text. You will explore two setups for extracting embeddings. It's recommended to save these extracted embeddings to avoid repeated computation. If implemented correctly, the test set accuracy is **at least 0.35**. **Save the models' test set predictions** for use in Part 3.

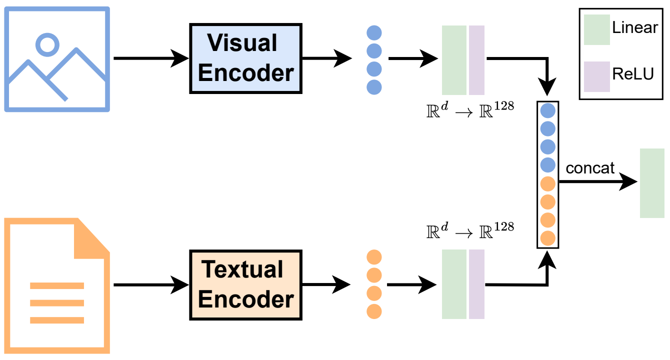
Visual Encoder
Textual Encoder
$\mathbb{R}^d \to \mathbb{R}^{128}$
$\mathbb{R}^d \to \mathbb{R}^{128}$
concat
Linear
ReLU

Figure 2: The model architecture.

Part 1: ResNet + SBERT (7 points)

Utilize a ResNet-50 model pretrained on ImageNet, to extract image embeddings just **before** the classification head. Use the sentence transformer all-MiniLM-L6-v2 to extract sentence embeddings. Refer to this tutorial for implementation.

Implement the model as shown in Fig. **??**. The model involves a linear layer with ReLU activation for dimension reduction, followed by the concatenation of the processed embeddings. Finally, this concatenated representation is passed through a linear classifier. Train the model and evaluate its performance on the test set.

Deliverable: **(a)** Dimensions of the embeddings; **(b)** Experimental result; **(c)** Code implementation.

Answer:

Part 2: CLIP (7 points)

Use the CLIP model (ViT-B/32)'s visual and textual encoder to extract the required embeddings. Refer to its official implementation details here. Similarly, implement and train the model, then report: **(a)** Dimensions of the embeddings; **(b)** Experimental result; **(c)** Code implementation.

Answer:

Part 3: Comparison and Analysis (6 points)

Analyze the pattern of the questions in the DAQUAR dataset. Review Section 3 and Table 1 of this paper. Determine how many types of questions DAQUAR (the subset used in this question) is composed of based on the paper's definition. Then divide DAQUAR by question types and analyze and compare the results from both approaches. Discuss potential reasons for any observed differences, considering factors such as the pertaining schedule and their suitability for feature extraction.

Deliverable:

- A table containing question types and the number of samples for each type in the dataset (training, validation, and test set).
- Accuracy scores of both models on the entire test set and for each question type.
- A comparison of both models for each question type and your analysis.

Answer:

Problem 3: Prompt Engineering Techniques (10 Points)

In this problem, you will experiment with different prompt styles to see how they affect the outputs of a pre-trained Microsoft Phi-1.5 model.

Background

Prompt engineering is an important skill when working with language models. Depending on how you ask a model to perform a task, the quality of the result can change. In this problem, you'll work with Hugging Face's transformers library and apply different prompts to a fact checking task.

Microsoft Phi-1.5 Model

The Microsoft Phi-1.5 model is designed to be efficient and powerful for a variety of tasks, including text generation and prompt-based learning. Phi-1.5 is known for its smaller architecture, which enables quicker responses while still maintaining the ability to perform well across many tasks. You can find more information about the Phi-1.5 model on this page.

In this problem, you will experiment with three prompt styles:

1. **Short and Direct**: Minimal instructions provided to the model.
2. **Few-Shot Learning**: The model is provided with labeled examples before classifying the target text.

Part 3.1: Testing Prompt Variations (5 Points)

Use the following sentences and test two of your own sentences for sentiment classification:

- “The Great Pyramid of Giza is located in Egypt.”
- “$4 + 4 = 16$.”
- “Mount Everest is the tallest mountain on Earth.”
- “Bats are blind.”
- “Sharks are mammals.”

Now, add two of your own sentences for testing.

Prompts:

- **Short and Direct**: “Classify the sentiment as positive or negative: [text].”
- **Few-Shot Learning**:

```
Statement: "The moon is made of cheese."
Answer: False
Statement: "The Eiffel Tower is located in Paris."
Answer: True
[text]
Answer:
```

Deliverables:

- Run the provided Python code in the separate file `problem_3.py` and test the two prompt strategies on each of the five given texts plus two sentences of your own.
- Provide outputs.
- Summarize how the structure of the prompt affected the model's responses. Compare the outputs for the different prompt styles and explain the differences.

Provided Code

You will use the Python code provided in the file `problem_3.py` to complete the task. Make sure to modify the sentences and experiment with the different prompt styles as described.

Part 3.2: Advanced Prompt Engineering (5 Points)

In this part, you will experiment with a more advanced prompt engineering technique: **Expert Prompting**. This technique asks the model to assume the role of an expert or a knowledgeable entity while performing the task. You will compare this approach to the simpler prompt styles used in Part 3.1.

Prompts for Expert Prompting:

- **Expert Prompting**: "You are a world-renowned fact-checker. Please carefully verify the following statement and explain whether it is true or false in detail: [text]."

Use the same sentences you used in Part 3.1

Deliverables:

- Run the Expert Prompting example on each sentence and compare the results to the output from Part 3.1 (Short and Direct and Few-Shot Learning).
- Provide the modified python code and outputs.
- Discuss whether the Expert Prompting technique improved the quality of the model's sentiment analysis. Did giving the model an "expert personality" help generate more coherent or accurate responses?

Useful Links:

- Microsoft Phi-1.5 Model: https://huggingface.co/microsoft/phi-1_5
- Hugging Face Pipelines: https://huggingface.co/docs/transformers/main_classes/pipelines