به نام خدا

دانشگاه تهران

دانشکدگان فنی

دانشکده مهندسی برق و کامپیوتر

درس پردازش زبان طبیعی

پاسخ تمرین شماره سه

نام و نام خانوادگی: امیرحسین شاه قلی

شماره دانشجویی: ۸۱۰۱۹۹۴۴۱

خرداد ماه ۱۴۰۳

Table of Contents

Answer of the question one

Dataset:

The Multi-Genre Natural Language Inference (MultiNLI) dataset is designed for use in natural language understanding tasks. The dataset expands upon the Stanford NLI (SNLI) dataset by introducing a diverse range of genres of spoken and written text, and it supports a variety of NLP tasks including text classification, entailment, and contradiction. MultiNLI includes 433k sentence pairs annotated with textual entailment information. These pairs are drawn from multiple sources and represent different genres, providing a challenging and broad testing ground for models aimed at understanding natural language inference. The dataset splits into training, validation, and test sets, allowing for thorough model training and evaluation across different text genres.

Each instance in this dataset consists of the following elements:

promptID: Unique identifier for prompt

pairID: Unique identifier for pair

{premise,hypothesis}: combination of premise and hypothesis

{premise,hypothesis} parse: Each sentence as parsed by the Stanford PCFG Parser 3.5.2

{premise,hypothesis} binary parse: parses in unlabeled binary-branching format

genre: a string feature.

label: a classification label, with possible values including entailment (0), neutral (1), contradiction (2). Dataset instances which don't have any gold label are marked with -1 label. Make sure you filter them before starting the training using datasets.Dataset.filter.

Answer of the first part:

Q1)

Fine-Tuning All Model Parameters: In this strategy, all the parameters of a pretrained model are retrained with data specific to a new domain or task. This method allows the model to fully adapt itself to the specific features of the new data but can require significant computational resources and time and it raises the risk of overfitting.

Fine-Tuning Specific Layers of the Model: In this strategy, only parts of the pre-trained model, typically the upper layers (closer to the output), are retrained. The lower layers, responsible for extracting more fundamental features, remain unchanged. Reducing the number of trainable parameters decreases the necessary time and resources for training and often helps prevent overfitting.

LORA Approach:

The key idea behind LORA is to perform adaptations to a model's weights through a low-rank decomposition approach rather than directly modifying the original weights extensively. Specifically, for each layer A in the pre-trained model, the adaptation is achieved by introducing two low-rank matrices L and R. The original weight matrix A is decomposed into the sum of the product of these two matrices and the original weight, formulated as: $A'=A+LR$. Where A is the original set of weights of the layer. L and R are smaller matrices that represent the low-rank updates.

When applying LORA:

The model's original parameters are largely retained. Only the low-rank matrices L and R for each adapted layer are optimized during the fine-tuning process.The size of these matrices serves as a hyperparameter that can be tuned based on the desired balance between performance and efficiency.

This method is particularly aligned with scenarios where one wishes to fine-tune a large model on a specific task without the risk of overfitting and while keeping the computational demands manageable. It also upholds the benefits of deep transfer learning by leveraging pre-trained networks, ensuring that fine-tuning is both effective in enhancing model performance on specific tasks and efficient in terms of computational resources.

Q2)

Hard Prompt: In this method, hard or fixed text prompts are given to the model, where the input text is structured to include specific instructions for the model. These instructions help the model produce more relevant and accurate outputs.

Soft Prompt: Instead of using text prompts defined by the user, soft prompts typically involve vectors defined in the model's feature space. These vectors are provided as part of the model's input and can be dynamically adjusted to help the model better interact with the input data and the specific problem at hand.

ANSWER OF THE SECOND PART:

part1:

Hyper Parameters:

```
training_args = TrainingArguments(
   evaluation_strategy='epoch',
   learning_rate=2e-5,
   per_device_train_batch_size=12,
   per_device_eval_batch_size=12,
   num_train_epochs=3,
   weight_decay=0.01,
   gradient_accumulation_steps=2,
   fp16=True,
   dataloader_num_workers=8,
   report_to="none",
)
```

Training Time: 1:16:22

Results:

{'eval_loss': 0.5437459349632263, 'eval_accuracy': 0.8819154355578197, 'eval_runtime': 68.7286, 'eval_samples_per_second': 142.808, 'eval_steps_per_second': 11.902, 'epoch': 2.999083409715857}

Epoch	Training Loss	Validation Loss	Accuracy
0	0.405800	0.354279	0.874885
2	0.137000	0.543746	0.881915

part2 (LoRA):

Hyper Parameters:

```
training_args = TrainingArguments(
   output_dir='./results',
   evaluation_strategy='epoch',
   learning_rate=2e-5,
   per_device_train_batch_size=64,
   per_device_eval_batch_size=64,
   num_train_epochs=3,
   weight_decay=0.01,
   gradient_accumulation_steps=2,
   fp16=True,
   dataloader_num_workers=8,
   report_to="none",
)
```

Lora config:

```
lora_config = LoraConfig(
   task_type=TaskType.SEQ_CLS,
   inference_mode=False,
   r=16,
   lora_alpha=32,
   lora_dropout=0.1
)
```

Training Time: 32:02

Results:

{'eval_loss': 0.5309885740280151, 'eval_accuracy': 0.795109526235354, 'eval_runtime': 58.6601, 'eval_samples_per_second': 167.32, 'eval_steps_per_second': 2.625, 'epoch': 3.0}

Epoch	Training Loss	Validation Loss	Accuracy
1	No log	1.076687	0.444116
2	1.030900	0.592851	0.766378
3	1.030900	0.530989	0.795110

part3 (P-Tunning):

Hyper Parameters:

```
training_args = TrainingArguments(
   output_dir='./results',
   evaluation_strategy='epoch',
   learning_rate=2e-5,
   per_device_train_batch_size=8,
   per_device_eval_batch_size=8,
   num_train_epochs=3,
   weight_decay=0.01,
   gradient_accumulation_steps=2,
   fp16=True,
   dataloader_num_workers=8,
   report_to="none",
)
```

Training Time: 1:24:51

Results:

Epoch	Training Loss	Validation Loss	Accuracy
0	0.419600	0.354077	0.871116
2	0.258700	0.413778	0.878248

Comparing three parts:

Performance Efficiency:

- Traditional Fine-Tuning shows the highest accuracy, which suggests that full-scale parameter adjustments can capture nuances in the dataset effectively.
- LORA provides a balanced approach, reducing the risk of overfitting and computational load but at some cost to accuracy.
- P-Tuning likely offers flexibility and efficiency but potentially at a slight compromise on performance compared to full parameter fine-tuning.

Resource Utilization:

- Traditional Fine-Tuning is the most resource-intensive, suitable for scenarios where hardware capabilities are not a constraint.
- LORA and P-Tuning are more resource-efficient, making them practical for situations with limited computational resources or where quick iterations are necessary.

Use Case Suitability:

- Traditional Fine-Tuning is ideal when the utmost performance is necessary, and one has the resources to support extensive training.
- LORA could be the go-to method for continuous learning scenarios or when deploying multiple models simultaneously, necessitating less frequent full model retraining.
- P-Tuning could serve niches where specific input shapes or novel prompt integration is crucial, perhaps in highly specialized or creative text generation tasks.

ANSWER OF THE THIRD PART:

To compare the implementations of the LoRA method and traditional fine-tuning approaches using **roberta-large**, let's delve into their differences and understand how each method performs when adapting the model for specialized tasks without significantly altering the core parameters.

Traditional Fine-Tuning of Roberta-Large:

In traditional fine-tuning, all model parameters are updated. This is exhaustive but ensures that the model closely adapts to the specifics of the new task.

- **Advantages:**

  - Highly adaptive to the new task nuances.

  - Potentially better performance on the specific domain.

- **Disadvantages:**

  - Higher risk of overfitting, especially with limited training data.

  - Significant computational resources are required for training.

  - Not ideal for multi-task applications where rapid switching between tasks is necessary; re-training for every new task can be resource-intensive.

Best when maximum performance on a highly specific task is critical, and computational resources are plentiful. Not ideally suited for scenarios where the model needs to frequently switch tasks.

Fine-Tuning with LoRA:

LoRA (Low Rank Adaptation) involves modifying a small number of additional parameters that interact with the pre-existing parameters. This allows for effective adaptation with minimal changes to the core model.

- **Advantages:**

  - Efficient adaptation with less risk of overfitting.

  - Lower computational costs as fewer parameters are updated.

  - Easier adaptation when using the model for multiple tasks; you can flexibly update the model for new tasks without significant re-training.

- **Disadvantages:**

  - Might not capture as nuanced behaviors of the domain as full model re-training.

  - Potential performance trade-offs if task-specific adaptations require extensive changes to model behavior.

A more practical choice for scenarios requiring the model to handle multiple tasks or when computational efficiency is a priority. The ability to adapt quickly and with fewer resource commitments makes it particularly advantageous in multi-task settings or when frequently updating model capabilities based on new data.

Answer of the question two:

Note: I can not run llama-3 on colab and all sessions crash because of heavy load on RAM and I just write the codes of this part in Q2.ipynb.

Answer of the first part:

Zero-Shot Prompting:

In zero-shot learning, the model makes predictions on data that it has never seen during training without any fine-tuning or additional training intervention related to that specific task. This is typically enabled by using models that are pre-trained on large, diverse datasets and have developed a generalized understanding of language or tasks. The model uses this acquired knowledge to infer answers to unseen tasks.

Examples:

- **Language Models**: A pre-trained model like GPT-3 can generate text based on prompts despite never having specifically trained on the exact query or domain. It uses its extensive pre-training on a broad corpus to deduce an appropriate response.

- **Classification:** A text classifier trained on diverse topics could categorize texts into emotion categories even if it wasn't explicitly trained on those specific texts.

Zero-shot learning is useful when it's impractical to obtain labeled data for every potential task. It's applied in scenarios where versatility without extensive retraining is beneficial, such as generalized question answering systems or initial exploratory analysis.

One-Shoterior Prompting:

In one-shot learning, the model tries to understand and generalize from a single example or very few examples. This is particularly more challenging than zero-shot learning because the model is expected to grasp the nuances of the task from minimal interaction.

Examples:

- **Image Recognition**: After seeing just one image of a novel object, such as a specific model of car, a model could be expected to identify future instances of the same model in differing contexts or angles.

- **NLP Tasks:** It can be applied to generate text or perform a complex task modeled from a single example provided in the prompt, like translating a sentence between two rare languages by providing a one-off example.

One-shot learning is crucial in domains where the data is scarce or costly to annotate. It's significant in medical fields, rare object identification, or linguistic tasks involving low-resource languages.

Zero-Shot vs One-Shot:

Zero-shot learning does not use any specific examples at inference time, relying solely on pre-trained knowledge. In contrast, one-shot learning uses precisely one or a few examples to guide the prediction and also Zero-shot is highly flexible and can be applied broadly, but might lack the nuances that might be captured through one-shot learning when specific examples are available.

Zero-shot might perform less accurately on tasks that are very particular and have not been encompassed in the pre-training regime. One-shot learning, although potentially more tuned to the immediate task via the example, also critically depends on the representativeness of the given example. Both methods represent the cutting edge of making machine learning models more flexible and capable of dealing with data or tasks in real-world settings, minimizing the dependency on large labeled datasets.

Answer of the second part:

QLoRA (Quantization-aware LoRA) integrates both LoRA and quantization. LoRA decomposes weight updates into the product of two lower-dimensional matrices, A and B. Instead of updating the entire weight matrix W and computes W + A * B, with A and B being trainable, which reduces the number of parameters to be fine-tuned. In QLoRA, the parameters of the original model W are quantized to lower precision, such as 8-bit or 4-bit instead of 32-bit floating point. By keeping the quantized weights fixed and only fine-tuning the smaller low-rank matrices A and B, QLoRA maintains high performance while significantly reducing computational demands. The training process starts with a pre-trained large language model, whose weights are then quantized. Low-rank matrices A and B are introduced into the transformer layers, and during fine-tuning, only these matrices are updated while the quantized weights remain fixed. This approach has several benefits. Quantizing the weights reduces model size and computational resource requirements, making fine-tuning more efficient. It also allows for the fine-tuning of very large language models on consumer-grade hardware, such as a single GPU. QLoRA carefully balances the trade-offs between compression and the ability to effectively fine-tune the model, showing that fine-tuning the auxiliary low-rank matrices can retain or even improve model performance. In practical terms, QLoRA enables the customization of large language models for specific tasks without

needing extensive computational resources. It broadens accessibility to powerful language models for researchers and organizations with limited budgets, while still maintaining high performance despite quantization. This makes QLoRA a highly efficient and scalable approach for fine-tuning large language models.

ANSWER OF THE THIRD PART:

Explanation of Hyperparameters and Reason for Selection:

1. Number of Training Epochs (num_train_epochs = 3): This value is a common choice to ensure that the model has enough time to learn from the data without overfitting. Three epochs provide a balance between training time and performance.

2. Batch Size (per_device_train_batch_size = 8): A smaller batch size is chosen considering the memory limitations of the GPU. It allows efficient use of available resources and prevents out-of-memory errors.

3. Warmup Steps (warmup_steps = 500): Warmup steps are used to gradually increase the learning rate from a very small value to the initial learning rate. This helps in stabilizing the training process and reducing large oscillations in the beginning.

4. Weight Decay (weight_decay = 0.01): This value is used to prevent overfitting by penalizing large weights. Weight decay is a form of regularization that helps the model generalize better to unseen data.

5. Evaluation Strategy (evaluation_strategy = "steps"): The evaluation strategy "steps" means that evaluation is performed periodically during training after a specified number of steps (default is based on logging steps). This helps in monitoring the model's performance and can be used to perform early stopping if necessary.

Explanation of Steps:

1. Define Model Classes:

- QLoRaLayer: Defines a QLoRA layer that consists of two linear layers. The query layer compresses the hidden state into a lower-dimensional space, and the key layer projects it back to the original dimension. The output is the sum of the original hidden state and the modified states.

– LlamaWithLinearAndQLoRa: This class defines a composite model that includes the original LLaMA model, QLoRA layers added to each transformer block, and an additional linear layer for classification. Instead of using LlamaForSequenceClassification, it manually adds a linear layer for more control.

2. Add Linear Layer:

- The linear layer is added at the end of the model. It takes the hidden state corresponding to the [CLS] token (usually the first token) and projects it to the number of labels (classification classes). This layer is responsible for producing the logits used for classification.

3. Freeze Original Model Parameters:

- The parameters of the original LLaMA model are frozen to ensure they do not get updated during training. This means only the parameters of the QLoRA layers and the final linear layer are trained. This reduces computational load and focuses fine-tuning on fewer parameters.

4. Define Training Arguments:

- Training arguments are defined to control various aspects of the training process like number of epochs, batch size, steps for learning rate warmup, weight decay for regularization, and the strategy for periodic evaluation.

5. Training and Evaluation:

- The model is trained using the defined training arguments. After training, the model is evaluated on the validation set to measure its performance. The results of the evaluation are printed to provide insights into the model's accuracy and other metrics.