Natural Language Processing CA#4

Student Name:
Pouya Haji Mohammadi Gohari

SID:810102113

Date of deadline
Wednesday 29th May, 2024

Dept. of Computer Engineering

University of Tehran

Contents

1 REED ME

We have created 6 different notebooks for this project(even more than 6 notebooks though.)The 6 final version of these notebooks are in the code directory.

1. Fine-tuning All parameters of ROBERTA-model: Notebook's name is CA4_Roberta_All
2. Fine-tuning with LORA on ROBERTA-model:Notebook's name is CA4-Roberta-LORA
3. P-tuning ROBERTA model: Notebook's name is roberta-p-tuning
4. Zero and One shot prompting: Notebook's name is lama-icl
5. First QLORA model on LAMA: Notebook's name is qlora-frist-part
6. Second QLORA model on LAMA:Notebook's name is qlora-clf-lama

2 Dataset

In this section, we are thrilled to discuss the dataset available on the Hugging Face dataset page. You can access this dataset through this link.
Let's dive into exploring the dataset and provide a brief explanation of each feature and target. The code implemented below will simply download the dataset and extract in you google drive.

```
drive.mount('/content/drive')
url = 'https://cims.nyu.edu/~sbowman/multinli/multinli_1.0.zip'
response = requests.get(url)

output_path = '/content/drive/My Drive/dataset.zip'
with open(output_path, 'wb') as f:
  f.write(response.content)
  dataset_path = '/content/drive/My Drive/dataset.zip'
with zipfile.ZipFile(dataset_path, 'r') as zip_ref:
  zip_ref.extractall('/content/drive/My Drive/multinli_1.0')
  train_file_path = '/content/drive/MyDrive/multinli_1.0/multinli_1.0/multinli_1.0
    ↪ _train.jsonl'
train_df = pd.read_json(train_file_path, lines=True)
```

As Hugging Face refered, this dataset is a crowd-sourced collection of 433k pairs annotated with textual entailment information.
Feature and target explanation:

1. 'annotator_labels': This feature contains the labels assigned by individual annotators. It reflects the initial classification made during the dataset creation process.

2. 'gold_label':This feature contains the final labels, determined after aggregation and resolving any disagreements among annotators.It is indeed the final label assigned for each sentence.

3. 'pairID': This is unique identifier for each pair of sentences.

4. 'promptID': This is also a unique identifier for prompt.

5. 'sentence1', 'sentence2': These two features represent two different sentences in each sample of the dataset. They are also referred to as 'premise' and 'hypothesis'.

6. 'sentence1', 'sentence2' parse: Each sentences has been parsed by the Standford PCFG parser 3.5.2. Each showing the parsed version of the sentences.

7. 'sentence1', 'sentence2'binary parse: parses in unlabeled binary-branching format.

8. 'genre': This is string feature where each sentence has own genre. For example: "government", "travel", "fiction" and etc.

Attention: Careful that we have labels as entailment(0), neural(1) and contradictions. A sample which don't have any gold label are marked with '-1' label.
Furthermore, our dataset has been split into three sets:

- Training: Is used for training models.
- Matched Validation: Contains sentence pairs from the same genres as the training set.
- Mismatched Validation: Contains sentence pairs from different genres not seen during training.

Let's proceed into get an instance of this dataset. The instance is the 10th example:

- annotator_labels: ['entailment']
- genre: fiction
- gold_labe: entailment
- pairID: 55785e
- promptID: 55785
- sentence1: I burst through a set of cabin doors, and fell to the ground-
- sentence1_binary_parse: (I (((((burst (through ((a set) (of (cabin doors))))) ,) and) (fell (to (the ground)))) -))
- sentence1_parse: (ROOT (S (NP (PRP I)) (VP (VP (VBP burst) (PP (IN through) (NP (NP (DT a) (NN set)) (PP (IN of) (NP (NN cabin) (NNS doors)))))) (, ,) (CC and) (VP (VBD fell) (PP (TO to) (NP (DT the) (NN ground))))) (: -)))
- sentence2: I burst through the doors and fell down.
- sentence2_binary_parse: (I ((((burst (through (the doors))) and) (fell down)) .))
- sentence2_parse: (ROOT (S (NP (PRP I)) (VP (VP (VBP burst) (PP (IN through) (NP (DT the) (NNS doors)))) (CC and) (VP (VBD fell) (PRT (RP down)))) (. .)))

3 Part 1: Fine Tuning Methods

3.1 Traditional Methods of Fine-Tuning and LORA

The methods we are going to discuss are tradition methods for Fine-Tuning. Fine-tuning is a transfer learning technique where the parameters of a pre-trained model are further trained on new data. Transfer learning generally consists taking a pre-trained model's layers, freezing them to maintain their learned features, adding new layers on top, and training these new layers on a new dataset.
There are several ways to apply fine-tuning where each will be explained:

- Just Update Added Layers: In order to use the pre-trained model for specific tasks, we can add additional layers on top of them. For instance, consider using BERT for sequence classification like sentiment analysis.In this example, we would add a classification layer w_c after the last output of ['CLS'] token.(Figure 1 is a visual representation of this concept). One way is just freeze all pre-trained model and just update the

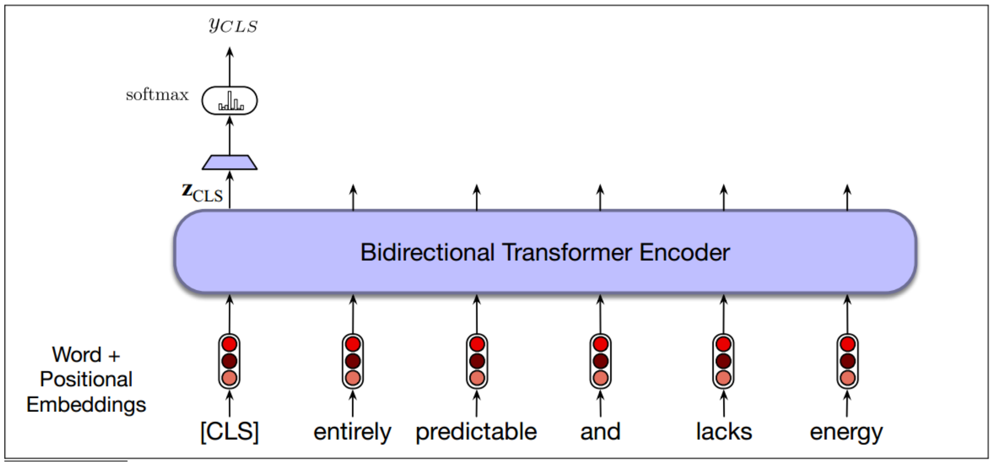


Figure 1: Sequence Classification Using BERT as Pre-Trained Model

  weight of added layers.

- Update All Layers: This method requires more computational power and takes more amount of time. This method will not freeze layers, instead when the loss computed, not also the added layer's parameters would be updated but the pre-trained model's parameters will also updated.

- Update Subset of Layers: This method is a most common way to fine-tune a pre-trained model. In here we can freeze all BERT layers except a subset of it's last layers.According to [2] typically, in convolution neural networks(CNNs), the earlier layers that capture low-level features are kept frozen, while the last layers, which capture more task-specific high-level features, are fine-tuned.

LORA(Low-Rank-Decomposition) is a technique introduced to efficiently fine-tune large-scale pre-trained models. LORA inspired by the idea that over-parameterized models operate within a low intrinsic dimension[5], they propose that the changes in weights during model adaption also exhibits a low "intrinsic rank". Let's explain their concept using figure 2 from the original paper:
As illustrated in figure 2, LORA introduced trainable low-rank matrices into each layer of the pre-trained model.Instead of updating all parameters, LORA injects these low-rank matrices into the layers like transformers.

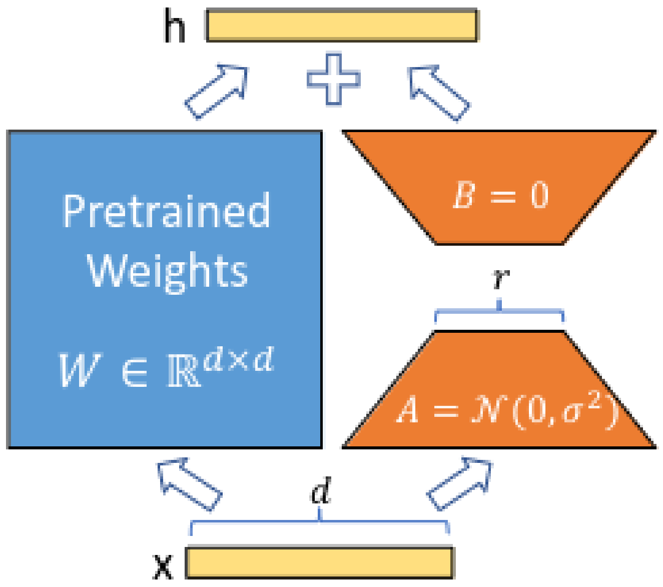

h
Pretrained Weights
$W \in \mathbb{R}^{d\times d}$
$B = 0$
r
$A = \mathcal{N}(0, \sigma^2)$
d
x

Figure 2: LORA Technique[5]

The whole idea is to decompose the weight updates into two smaller matrices that capture the most significant changes required for the new task.
There are several advantages using LORA technique for fine-tuning. The biggest advantages from adding these low-rank matrices, LORA significantly reduces the number of trainable parameters. This makes the fine-tuning process much more efficient with respect to computational resources and memory usage. In the original paper was told, compared fine-tuning GPT3 with Adam, LORA can reduce the number of trainable parameters by 10,000 times and the GPU memory requirements by 3 times.

3.2 Hard Prompt and Soft Prompt

There are two techniques can be considered as forms of fine-tuning but works slightly different. In soft prompts(also known as continuous prompts) consist learning a set of continuous embeddings that act as prompts for pre-trained language model.Despite to hard prompts(which use discrete tokens) soft prompts operate in the embedding space of the model.Soft prompts has two benefits where can be explained as follow:

1. This technique can fine-tuning pre-trained models without updating all the models parameters. Just learned embeddings are updated which is good for us with respect to computational cost.

2. With soft prompts, models can be optimized for various tasks. With minimal changes to the model's architecture, pre-trained models can be adaptable and powerful.

Hard prompts involve using actual words or phrases as prompts to guide the language model.Generally these prompts are crafted or automatically generated and will be directly used as inputs to the model as part of text sequence.Some advantages of using hard prompts can be categorized as follow:

1. Hard prompts consist natural language tokens where can be understandable and interpretable by humans.

2. Hard prompts can be created without extra training, where can be helpful to use of the existing knowledge and abilities of the pre-trained model.

4 Part 2: Training Model

4.1 Fine-Tuning All Roberta-Large Parameters

We have already used ”multi-nli” previous sections. Now we are going to use ROBERTA model in order to train all the parameters over this dataset.
Specifying hyper-parameters for future training purposes:

```
    model_name_or_path = "roberta-large"
    num_epochs = 1
    lr = 1e-5
    batch_size = 4
```

Attention: First variable(’model_name_or_path’) is not hyper-parameter but the rest of them are indeed hyper-parameters. This variable is used for specifying the model.
Now from utilizing the ’load_dataset’ method in ’datasets’ library we can load the dataset like below:

```
    dataset = load_dataset('multi_nli')
```

We should define a metric for evaluate the training steps. We have used ’evaluate’ library for this purpose:

```
    metric = evaluate.load('accuracy')
```

Define computation metrics as a function:

```
def compute_metrics(eval_pred):
    predictions, labels = eval_pred
    predictions = np.argmax(predictions, axis=1)
    return metric.compute(predictions=predictions, references=labels)
```

Since the time for each epoch of training is time consuming, we will get 10 percentage of train data for training phase:

```
def sample_10_percent(dataset):
    total_size = len(dataset)
    sample_size = max(1, total_size // 10)
    indices = random.sample(range(total_size), sample_size)
    return dataset.select(indices)
    sampled_train_dataset = sample_10_percent(dataset['train'])
    sampled_validation_dataset = sample_10_percent(dataset['validation_matched'])
```

Now it is time to pre-process the dataset in order to feed it in our model. So we should load the tokenizer of Roberta as follow:

```
    tokenizer = RobertaTokenizer.from_pretrained(model_name_or_path, padding_side=
       ↪ "right", num_labels=3)
```

From utilizing this tokenizer we will pre-process the dataset with the following function:

```
def preprocess_function(examples):
    outputs = tokenizer(examples['premise'], examples['hypothesis'], truncation=
       ↪ True, max_length=None)
    return outputs
    train_tokenized_dataset = sampled_train_dataset.map(preprocess_function,
       ↪ batched=True, remove_columns=["promptID", "pairID", "premise", "
       ↪ premise_binary_parse", "premise_parse", "hypothesis", "
       ↪ hypothesis_binary_parse", "hypothesis_parse", "genre"])
    train_tokenized_dataset = train_tokenized_dataset.rename_column("label", "
       ↪ labels")
    match_val_tokenized_dataset = sampled_validation_dataset.map(
       ↪ preprocess_function, batched=True, remove_columns=["promptID", "premise"
       ↪ , "pairID", "premise_binary_parse", "premise_parse", "hypothesis", "
       ↪ hypothesis_binary_parse", "hypothesis_parse", "genre"])
    match_val_tokenized_dataset = match_val_tokenized_dataset.rename_column("label
       ↪ ", "labels")
```

Now we need a data collector with padding that pad each sequences with longest length of sequences.

```
data_collator = DataCollatorWithPadding(tokenizer=tokenizer, padding="longest")
```

Adding the ROBERTA model:

```
model = RobertaForSequenceClassification.from_pretrained(model_name_or_path,
    ↪ num_labels=3).to('cuda')
```

Define the arguments for the training:

```
training_args = TrainingArguments(
    output_dir="pouya/roberta-large-peft-lora",
    learning_rate=lr,
    per_device_train_batch_size=batch_size,
    gradient_accumulation_steps=batch_size,
    num_train_epochs=num_epochs,
    warmup_steps=10,
    weight_decay=0.01,
    evaluation_strategy="epoch",
    save_strategy="epoch",
    load_best_model_at_end=True,
    logging_dir='./logs',
    report_to="none",
    logging_steps= 100,
)
```

We don't want any report so we set the 'report_to' none. Some of these arguments descriptions are:

1. gradient_accumulation_steps: This parameter accumulates gradients over multiple steps before performing a backward/update pass.It allows us to effectively increase the batch size without requiring more memory.

2. warmup_steps: This parameter defines the number of steps for the learning rate warmup. During warmup phase, the learning rate increases linearly from 0 to specified learning rate.

Final step is implementing trainer with above arguments:

```
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_tokenized_dataset,
    eval_dataset=match_val_tokenized_dataset,
    tokenizer=tokenizer,
    data_collator=data_collator,
    compute_metrics=compute_metrics,
)
trainer.train()
```

The hyper parameters has been illustrated in tabel 1: The model's architecture:

Batch Size	Learning Rate	Number of Epoch	Percentage of Train Data	Model Name
4	1e-5	1	10%	Roberta Sequence Classification

Table 1: Hyper Parameters

```
RobertaForSequenceClassification(
  (roberta): RobertaModel(
    (embeddings): RobertaEmbeddings(
      (word_embeddings): Embedding(50265, 1024, padding_idx=1)
      (position_embeddings): Embedding(514, 1024, padding_idx=1)
      (token_type_embeddings): Embedding(1, 1024)
      (LayerNorm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): RobertaEncoder(
      (layer): ModuleList(
        (0-23): 24 x RobertaLayer(
          (attention): RobertaAttention(
            (self): RobertaSelfAttention(
              (query): Linear(in_features=1024, out_features=1024, bias=True)
              (key): Linear(in_features=1024, out_features=1024, bias=True)
              (value): Linear(in_features=1024, out_features=1024, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): RobertaSelfOutput(
              (dense): Linear(in_features=1024, out_features=1024, bias=True)
              (LayerNorm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): RobertaIntermediate(
            (dense): Linear(in_features=1024, out_features=4096, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): RobertaOutput(
            (dense): Linear(in_features=4096, out_features=1024, bias=True)
            (LayerNorm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
  )
  (classifier): RobertaClassificationHead(
    (dense): Linear(in_features=1024, out_features=1024, bias=True)
    (dropout): Dropout(p=0.1, inplace=False)
    (out_proj): Linear(in_features=1024, out_features=3, bias=True)
  )
)
```

The number of training parameters along with the time needed for training is shown in the table 2:

Time of Training	36:52
Training parameters	355,362,819

Table 2: Training Parameters and Training Time

Accuracy for fine-tuning all the parameters of ROBERTA is about 88%.

4.2 LORA

The code is same but we need to apply LORA on the ROBERTA model with the 'peft' library as code below shows:

```
    target_modules = [
    "query",
    "keys",
    "values",
    "dense"
]
peft_config = LoraConfig(
    task_type="SEQ_CLS", inference_mode=False, r=8, lora_alpha=0.5, lora_dropout
       ↪ =0.1, bias="none", target_modules=target_modules
)
```

We have specified modules where LORA should be applied. We consider to apply LORA on the query, key and value for transformer's layers and dense layers as well. We have set the lora_alpha to 0.5,as scaling factor for the low-rank matrices.
Next step would be combine the model:

```
    model = RobertaForSequenceClassification.from_pretrained(model_name_or_path,
       ↪ return_dict=True, num_labels=3)
    model = get_peft_model(model, peft_config)
    model.print_trainable_parameters()
```

Now we are all set to train our LORA model. The hyper-paramters are shown in table 3:

Batch Size	Learning Rate	Number of Epoch	Percentage of Train Data	Model Name
8	3e-5	3	10%	Roberta Sequence Classification

Table 3: Hyper Parameters for LORA

The model architecture is:

```
PeftModelForSequenceClassification(
  (base_model): LoraModel(
    (model): RobertaForSequenceClassification(
      (roberta): RobertaModel(
        (embeddings): RobertaEmbeddings(
          (word_embeddings): Embedding(50265, 1024, padding_idx=1)
          (position_embeddings): Embedding(514, 1024, padding_idx=1)
          (token_type_embeddings): Embedding(1, 1024)
          (LayerNorm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
          (dropout): Dropout(p=0.1, inplace=False)
        )
        (encoder): RobertaEncoder(
          (layer): ModuleList(
            (0-23): 24 x RobertaLayer(
              (attention): RobertaAttention(
                (self): RobertaSelfAttention(
                  (query): lora.Linear(
                    (base_layer): Linear(in_features=1024, out_features=1024, bias
                      ↪ =True)
                    (lora_dropout): ModuleDict(
                      (default): Dropout(p=0.1, inplace=False)
                    )
                    (lora_A): ModuleDict(
                      (default): Linear(in_features=1024, out_features=8, bias=
                        ↪ False)
                    )
                    (lora_B): ModuleDict(
                      (default): Linear(in_features=8, out_features=1024, bias=
                        ↪ False)
                    )
                    (lora_embedding_A): ParameterDict()
                    (lora_embedding_B): ParameterDict()
                  )
                  (key): Linear(in_features=1024, out_features=1024, bias=True)
                  (value): Linear(in_features=1024, out_features=1024, bias=True)
                  (dropout): Dropout(p=0.1, inplace=False)
                )
              (output): RobertaSelfOutput(
                (dense): lora.Linear(
                  (base_layer): Linear(in_features=1024, out_features=1024, bias=True)
                  (lora_dropout): ModuleDict(
                    (default): Dropout(p=0.1, inplace=False)
                  )
                  (lora_A): ModuleDict(
                    (default): Linear(in_features=1024, out_features=8, bias=False)
                  )
                  (lora_B): ModuleDict(
                    (default): Linear(in_features=8, out_features=1024, bias=False)
                  )
                  (lora_embedding_A): ParameterDict()
                  (lora_embedding_B): ParameterDict()
                )
                (LayerNorm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
                (dropout): Dropout(p=0.1, inplace=False)
              )
            )
            (intermediate): RobertaIntermediate(
              (dense): lora.Linear(
                (base_layer): Linear(in_features=1024, out_features=4096, bias=True)
                (lora_dropout): ModuleDict(
                  (default): Dropout(p=0.1, inplace=False)
                )
                (lora_A): ModuleDict(
                  (default): Linear(in_features=1024, out_features=8, bias=False)
                )
                (lora_B): ModuleDict(
                  (default): Linear(in_features=8, out_features=4096, bias=False)
                )
                (lora_embedding_A): ParameterDict()
                (lora_embedding_B): ParameterDict()
              )
              (intermediate_act_fn): GELUActivation()
            )
            (output): RobertaOutput(
              (dense): lora.Linear(
                (base_layer): Linear(in_features=4096, out_features=1024, bias=True)
```

```
                  (lora_dropout): ModuleDict(
                    (default): Dropout(p=0.1, inplace=False)
                  )
                  (lora_A): ModuleDict(
                    (default): Linear(in_features=4096, out_features=8, bias=False
                       ↪ )
                  )
                  (lora_B): ModuleDict(
                    (default): Linear(in_features=8, out_features=1024, bias=False
                       ↪ )
                  )
                  (lora_embedding_A): ParameterDict()
                  (lora_embedding_B): ParameterDict()
                )
                (LayerNorm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True
                   ↪ )
                (dropout): Dropout(p=0.1, inplace=False)
              )
            )
          )
        )
      )
      (classifier): ModulesToSaveWrapper(
        (original_module): RobertaClassificationHead(
          (dense): lora.Linear(
            (base_layer): Linear(in_features=1024, out_features=1024, bias=True)
            (lora_dropout): ModuleDict(
              (default): Dropout(p=0.1, inplace=False)
            )
            (lora_A): ModuleDict(
              (default): Linear(in_features=1024, out_features=8, bias=False)
            )
            (lora_B): ModuleDict(
              (default): Linear(in_features=8, out_features=1024, bias=False)
            )
            (lora_embedding_A): ParameterDict()
            (lora_embedding_B): ParameterDict()
          )
          (dropout): Dropout(p=0.1, inplace=False)
          (out_proj): Linear(in_features=1024, out_features=3, bias=True)
        )
        (modules_to_save): ModuleDict(
          (default): RobertaClassificationHead(
            (dense): lora.Linear(
              (base_layer): Linear(in_features=1024, out_features=1024, bias=True)
              (lora_dropout): ModuleDict(
                (default): Dropout(p=0.1, inplace=False)
              )
              (lora_A): ModuleDict(
                (default): Linear(in_features=1024, out_features=8, bias=False)
              )
              (lora_B): ModuleDict(
                (default): Linear(in_features=8, out_features=1024, bias=False)
              )
              (lora_embedding_A): ParameterDict()
              (lora_embedding_B): ParameterDict()
            )
            (dropout): Dropout(p=0.1, inplace=False)
            (out_proj): Linear(in_features=1024, out_features=3, bias=True)
          )
        )
      )
    )
  )
)
```

The number of training parameters along with the time needed for training is shown in the table 4:

Time of Training	47:53
Training parameters	3,821,571

Table 4: Training Parameters and Training Time

Accuracy for model with LORA is about 84%.

4.3 Comparison of LORA and Traditional

We will discuss the comparison between this model and traditional techniques in the PART 3(Why LORA?).

4.4 P-tuning

Utilizing 'peft' library, we will implement p-tuning technique where consider as soft prompting.
P-tuning adds trainable prompt tokens to the input which are optimized by a prompt encoder to find the best prompts. This means we don't have to manually create prompts!.Moreover, p-tuning introduces anchor tokens to enhance performance[4].
Utilizing peft for our code:

```
peft_config = PromptEncoderConfig(
    peft_type="P_TUNING",
    task_type="SEQ_CLS",
    num_virtual_tokens=30,
    token_dim=1024,
    num_transformer_submodules=1,
    num_attention_heads=12,
    num_layers=12,
    encoder_reparameterization_type="MLP",
    encoder_hidden_size=512,
)
```

Some explanation for this configuration:

- peft_type: Specify the task which is P-TUNING.
- task_type: Sequence classification is our task.
- num_virtual_tokens: The total number of virtual tokens of the prompt encoder is set to 30.
- token_dim: The hidden embedding dimension of the base transformer model which is 1024.
- num_attention_heads and num_layers: These parameters are set to the layers attention layers and number of layers of base model.
- encoder_reparameterization_type:Number of transformer submodules, typically 1.
- encoder_hidden_size: Hidden size of the encoder, matching base model.

The model architecture for this model alongside the base model is just same as ROBERTA expecpt having following layer:

```
  (prompt_encoder): ModuleDict(
    (default): PromptEncoder(
      (embedding): Embedding(30, 1024)
      (mlp_head): Sequential(
        (0): Linear(in_features=1024, out_features=256, bias=True)
        (1): ReLU()
        (2): Linear(in_features=256, out_features=256, bias=True)
        (3): ReLU()
        (4): Linear(in_features=256, out_features=1024, bias=True)
      )
    )
  )
  (word_embeddings): Embedding(50265, 1024, padding_idx=1)
```

The hyper parameters for our configuration of p-tuning was declared earlier and other hyper-parameters are shown in table 5:

Batch Size	Learning Rate	Number of Epoch	Percentage of Train Data	Model Name
4	3e-5	3	10%	Roberta Sequence Classification

Table 5: Hyper-parameters

Table 6 present the number of trainable parameters and time needed:

Time of Training	49:52
Training parameters	1,674,755

Table 6: Training Parameters and Training Time

Accuracy of the model based on the p-tuning is 47%.
Comparison: By updating all parameters model can more effectively learn the specific patterns of the new dataset(multi-nli) leading to better perfromance.
However, p-tuning generally works well for tasks more related to the pre-trained models' original tasks[6]. For more complex or specific tasks like multi-nli involving natural language inference, p-tuning may not provide sufficient capacity for the model to be adapted.Thus resulting in lower performace. Furthermore, p-tuning is for to be parameter-effective by adjusting the soft prompts while keeping the rest of the model fixed or minimally updated.

5 Part 3: Why LORA?

Traditional methods like full fine-tuning like what we did on ROBERTA at first place, involving update all the parameters of a pre-trained model. This technique is very time consuming while achieving higher accuracy since the model can adopt more on the NLI task with full fine-tuning compare to LORA. For example in our case, full fine-tuning took one hour per epoch and reached 88% accuracy.
However, LORA only updates specific layers of the model while keeping the rest of parameters fixed.(In our case, we just apply LORA on query, key, values and dense layers). This technique is more faster and requires less computational resources although it may result in lower accuracy.In our scenario LORA achieves 84% accuracy in three epochs. Total time amount used to training was just 50 minutes.
If we want to use a single model like ROBERTA for multiple tasks(e.g, sentiment analysis and question answering) without refine-tuning the main parameters for each task would be differ:

1. Traditional Methods: For each task, the model must be fined-tuned separately and different versions of the model are saved for each task.However this method requires more time and computational resources.
2. LORA: In LORA we can keep a base model like in our case ROBERTA, and only update specific layers for each task. We will store these task-specific parameters separately while sharing the base model across all tasks. This allows quick switching between tasks and require less computational power.

According to LORA[5], one of the main advantages is when using a base mode for multiple tasks where is indeed feasible.Although using traditional methods must save and fine-tun each model correspond to each task.Lora can store the specific layers for each task and reuses the base model(ROBERTA). This would be efficient in aspect of time and resources.

6 Question 2

6.1 Part 1: ICL

6.2 Zero Shot Prompting

Zero-shot prompting is a technique in natural language processing where a pre-trained language model is given a specific prompt describing a task it hasn't been explicitly trained on.
Moreover, the model will use its knowledge and understanding to perform the task based on the provided prompt[1].
For instruct model of lama 8b, a specific chat template needs to be followed. So we consider a dialogue structure for each prompt given to the model.

```
    dialouge = [
        {'role': ' system', 'content': 'This is a system prompt.'},
        {'role': ' user', 'content': prompt},
    ]
```

As you can see the structure is based on defining roles and their content. First the role is system and then user in the case us! will give the prompt. After that, the model(as assistant) will respond to user's prompt.
Now we will create a prompt for each given example based on the premise and hypothesis:

```
    prompt = (
        f"Identify the relationship between the following statements:\n\n"
        f"Premise: {premise}\n"
        f"Hypothesis: {hypothesis}\n\n"
        "Just tell me the best answer among the following three options indicating
           ↪  the relationship between Premise and Hypothesis:\n"
        "1. entailment\n"
        "2. neutral\n"
        "3. contradiction\n"
    )
```

The given prompt is so simple. First we will declare the model you must choose and determine the relationship between two statements(premise and hypothesis in our scenario) then clarify the model that you must choose the best answer between three options(neutral, entailment and contradiction).
Using these prompt, we will implement a zero-shot prompting function as follow:

```
    def zero_shot_prompting(premise, hypothesis, tokenizer, model, max_length=10,
      ↪ temperature=0.2, top_p=0.9, top_k=50):
    prompt = (
        f"Identify the relationship between the following statements:\n\n"
        f"Premise: {premise}\n"
        f"Hypothesis: {hypothesis}\n\n"
        "Just tell me the best answer among the following three options indicating
           ↪  the relationship between Premise and Hypothesis:\n"
        "1. entailment\n"
        "2. neutral\n"
        "3. contradiction\n"
    )

    dialouge = [
        {'role': ' system', 'content': 'This is a system prompt.'},
        {'role': ' user', 'content': prompt},
    ]
    inputs = tokenizer.apply_chat_template(dialouge, add_generation_prompt=True,
       ↪ return_tensors='pt').to(device)
    with torch.no_grad():
        outputs = model.generate(
            input_ids=inputs,
            max_length=inputs.shape[1] + max_length,
            top_p=top_p,
            top_k=top_k,
            temperature=temperature,
            pad_token_id=tokenizer.eos_token_id,
        )
    generated_tokens = outputs[0][len(inputs[0]):]
    generated_text = tokenizer.decode(generated_tokens, skip_special_tokens=Trprue
       ↪ )
    return generated_text
```

Utilizing tokenizer's apply chat template we will create appropriate chat template for the LAMA-8B-Instruct. then we will generate the ouptuts.
Let's illustrate this by given an example:
Premise: The new rights are nice enough
Hypothesis: Everyone really likes the newest benefits
Model's response: The correct answer is:
1. entailment
The hyper-parameters are reported in table 7: Explanation:

Temperature	max length	top-p	top-k
0.2	10	0.9	50

Table 7: Hyper-Parameters for One-Shot

1. Temperature: Our task is simply. We just need three words(neural, contradiction, entailment). As we know when our goal is to generate diverse outputs, we usually set the temperature high. Otherwise we want quality.Thus we consider to set the temperature low since we want quality not diversity.

2. max_len: Please careful. We use this parameter as number of tokens that model should generate.(See the code above as reference). We expect the model tell us just the label and relation ship.Thus we consider to set the max len to 10.

3. top_k: Top-k is for setting a sampling limits the model's choices to the top-k most likely next words. Since the length of sequence have variation and we declare just tell us the label prompt we expect model to pick 50 most likely candidates.

4. top_p: Top-p sampling is a technique that choose smallest set of words where cumulative probability exceeds the threshold 'p'. We consider again this cumulative probability high since top-k is already 50 best therefore the best 50 candidates would be chosen. Since the last words of input are most likely to be generated again therefore we consider a high cumulative probabiliy since our last words of each prompt is labels.

The accuracy is reasonable. The accuracy for just 981 samples of validation matched dataset is about 48%.

6.3 One Shot Prompting

One shot prompting is a method where a LM[1] is given a single example or prompt to guide its understanding and generation of text based on that specific context of format. This will guide the LM to perform a task or generate context with respect to given example[1]. Through investigating the dataset we consider to choose the two following statements as an example of one shot prompting technique.

- Premise: Clark also expressed the hope that he and Redgrave could continue with their marriage.
- Hypothesis: Clark hoped that he could continue their marriage.
- Label: Entailment

These two statements are well-defined for guiding model in our task since the relationship between these two statements are clearly entail each other and model can understand it better.
The example prompt are created with the following structure:

```
example_prompt = (
    f"The relationship between two following statements:\n\n"
    f"Premise: {filtered_dataset[0]['premise']}\n"
    f"Hypothesis: {filtered_dataset[0]['hypothesis']}\n"
    f"is 1. entailment.\n"
    "Now given by this example\n"
)
```

Utilizing the example prompt, we will implement the following code to apply one shot technique.

```
def one_shot_prompting(premise, hypothesis, example_prompt, tokenizer, model,
   ↪ max_length=10, temperature=0.2, top_p=0.9, top_k=50):
    prompt = (
        f"Identify the relationship between the following statements:\n\n"
        f"Premise: {premise}\n"
        f"Hypothesis: {hypothesis}\n\n"
        "Just tell me the best answer among the following three options indicating
           ↪  the relationship between Premise and Hypothesis:\n"
        "1. entailment\n"
        "2. neutral\n"
        "3. contradiction\n"
    )
    one_shot_prompt = f"{example_prompt}\n\n\n{prompt}"
    dialouge = [
```

[1]Language Model

```python
        {'role': ' system', 'content': 'This is a system prompt.'},
        {'role': ' user', 'content': prompt},
    ]
    inputs = tokenizer.apply_chat_template(dialouge, add_generation_prompt=True,
      ↪ return_tensors='pt').to(device)
    with torch.no_grad():
        outputs = model.generate(
            input_ids=inputs,
            max_length=inputs.shape[1] + max_length,
            top_p=top_p,
            top_k=top_k,
            temperature=temperature,
            pad_token_id=tokenizer.eos_token_id,
        )
    generated_tokens = outputs[0][len(inputs[0]):]
    generated_text = tokenizer.decode(generated_tokens, skip_special_tokens=True)
    return generated_text
```

The hyper-parameter are as the previous section discussed. The accuracy for either zero or one shot prompting is based on the following code:

```python
numerical_predictions = []
for prediction in results['prediction']:
    pred_label = -1
    if "entailment" in prediction.lower():
        pred_label = 0
    elif "neutral" in prediction.lower():
        pred_label = 1
    elif "contradiction" in prediction.lower():
        pred_label = 2
    numerical_predictions.append(pred_label)
```

We will just simply examine the each prediction in oder to find the target label. If no label was found, predicted label would be -1. The accuracy after would be calculcated by accuracy score by 'sklearn' library.
The accuracy for one-shot prompting for 981 samples from validation's matched dataset is about 52%.
The increase in the accuracy between zero and one shot is due to we can guide the model toward a specific task by given a well-formed example.This accuracy indicates that even with a less example we can help the model to be more accurate.

6.4 Training Model With QLORA

This is quantized version of LORA. QLORA focuses on reducing memory requirements necessary for fine-tuning large models like lama model with 8B parameters and making easier to fine-tuning with accessible hardware like one GPU.According to paper of QLORA[3] model will be quantized into 4bits it would reduce the size of the model with reducing the precision of the parameters while maintaining performance.QLORA method is benefical with resepect to several reasons as follow:

1. QLORA reduces the memory requirement for fine-tuning large models from hundreds of gigabytes to less than 50GB, making it feasible to fine-tune a 65 billion parameter model on a single 48GB GPU.[3]

2. However QLORA reduce the memory message and parameter percision, it would maintain performance also.

For this purpose we will utilize BitsAndBytesConfig to load the lama-8b-instruct in 4bit.

```
bnb_config = BitsAndBytesConfig(
    load_in_4bit= True,
    bnb_4bit_quant_type= "nf4",
    bnb_4bit_compute_dtype= torch.float16,
    bnb_4bit_use_double_quant= False,
)
```

’nf4’: Specifies that the 4-bit quantization should use the noise-free type, focusing on accuracy and minimal disruption from quantization.Next step we shoul load the model in quantized version:

```
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    quantization_config= bnb_config,
    device_map="auto",
)
```

After this we will prepare the model for 4-bit training.Utilizing the ’peft’ library from hugging face we can configure LORA to apply to our model as follow:

```
peft_config = LoraConfig(
    lora_alpha= 16,
    lora_dropout= 0.05,
    r= 16,
    bias="none",
    task_type="CAUSAL_LM",
    target_modules= ["q_proj", "k_proj", "v_proj", "o_proj",
                     "gate_proj", "up_proj", "down_proj"]
)
```

Once again using ’peft’ we can apply our LORA configuration and add the LORA layers.

```
model = get_peft_model(model, peft_config)
model.print_trainable_parameters()
```

The number of parameters approximately are: $41,943,040$. Then we will consider 5 percentage of training data in order model got trained. For training lama-8b-instruct version we should follow a chat template.Thus we consider create a feature called 'prompt'. Our prompts are simply follow lama's template where model can interact with.According to this template, a instruction would be made as system prompt and user input would be a question based on how these statements (premise and hypothesis) are related. At the end assistant prompt will be used to guide the model how it must be respond where it is indeed the label of each sample.The following function will create this prompt for each sample of training set:

```
def create_prompt(example, instruction, mapper):
    premise = example['premise']
    hypothesis = example['hypothesis']
    label = example['label']
    user_input = f"\nPremise: {premise}\nHypothesis: {hypothesis}\n"
    prompt = f"""<|start_header_id|>system<|end_header_id|> {instruction}<|eot_id
       ↪ |><|start_header_id|>user<|end_header_id|> Are these two statements
       ↪ entailment, contradiction, or neutral?: {user_input}<|eot_id|><|
       ↪ start_header_id|>assistant<|end_header_id|> {mapper[label]}<|eot_id|>"""
    return prompt
```

Since this model is decoder only, therefore we consider it to generate words.Thus we conduct a dictionary mapper to convert the labels to their words.

```
mapper = {0: 'entailment', 1: 'neutral', 2: 'contradiction'}
```

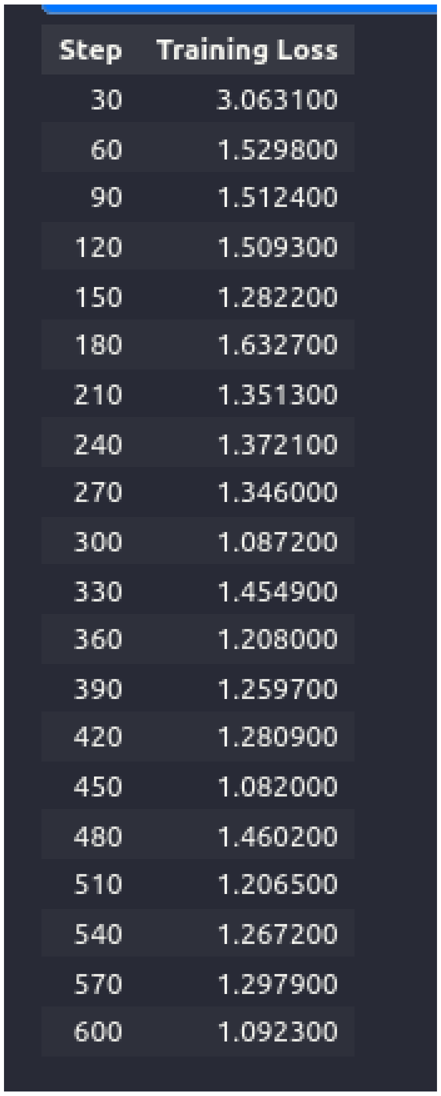
The instruction prompt is:

INSTRUCTION = Decide whether the following statements are entailment, contradiction, or neutral.

Now just based on the work shop guide, we will use Trainer in order to train the model.

```
from transformers import Trainer, TrainingArguments,
   ↪ DataCollatorForLanguageModeling
trainer = Trainer(
    model=model,
    train_dataset=train_tokenized_dataset,
    eval_dataset=match_val_tokenized_dataset,
    args=TrainingArguments(
        # num_train_epochs= 1,
        per_device_train_batch_size=4,
        gradient_accumulation_steps=4,
        logging_steps= 30,
        warmup_steps=5,
        max_steps=600,
        learning_rate=2e-5,
        weight_decay= 0.01,
        fp16= False,
        bf16= False,
        max_grad_norm= 0.3,
        group_by_length= True,
        optim="paged_adamw_8bit",
        output_dir= "/content/drive/MyDrive/weigths",
        report_to="none",
    ),
    data_collator=DataCollatorForLanguageModeling(tokenizer, mlm=False),
    compute_metrics=compute_metrics,
)
```

The loss function is illustrated in figure 3:



Step	Training Loss
30	3.063100
60	1.529800
90	1.512400
120	1.509300
150	1.282200
180	1.632700
210	1.351300
240	1.372100
270	1.346000
300	1.087200
330	1.454900
360	1.208000
390	1.259700
420	1.280900
450	1.082000
480	1.460200
510	1.206500
540	1.267200
570	1.297900
600	1.092300

Figure 3: Loss over each 30 steps for QLORA

For evaluating model, we could not merge the model since the ram would be overflowed in even Kaggle and Colab environment. And even with $merge\_and\_unload$ we can not reach a good accuracy since the parameters of LAMA is quantized while the LORA model is not. So we should use the same model to evaluate with not merged with LORA weights. For evaluating model we should create another prompt for the validation matched since we don not have any prompt for them and we will use a technique like zero-shot to obtain accuracy. The accuracy of the model for 5% of data is about 37%. The hyper parameters are illustrated in table 8:(Note that we have already defined what LORA configuration)

lora_alpha	16
r	16
lora_dropout	0.05
Number Of Trainable Parameters	41,943,040
Percentage data(val and train)	5%
Training Batch size	4
Gradient accumalation steps	4
Max Steps	600
Learning Rate	2e-5
Total Time	2:09:30
Target Modules of LORA	[”q_proj”, ”k_proj”, ”v_proj”, ”o_proj”, ”gate_proj”, ”up_proj”, ”down_proj”]

Table 8: Hyper Parameters

6.5 Second Method of QLORA

Based on the source code of Lama For Sequence Classification, the last hidden state would be consider to classify the output.Thus based on the source code we will implement our classifier:(Note that we utilize the source code of Lama For Sequence Classification and not the model!)

```python
import torch.nn as nn
import bitsandbytes as bnb
from torch.nn import CrossEntropyLoss
class CLF(LlamaPreTrainedModel):
    def __init__(self, config, num_labels):
        super().__init__(config)
        self.num_labels = num_labels
        self.model = LlamaModel(config)
        self.score = nn.Linear(config.hidden_size, self.num_labels, bias=False)
        self.post_init()

    def forward(self, input_ids=None,
                attention_mask=None,
                labels=None,
                position_ids=None,
                past_key_values=None,
                inputs_embeds=None,
                use_cache=None,
                output_attentions=None,
                output_hidden_states=None,
                return_dict=None):

        transformers_outputs = self.model(
                input_ids=input_ids,
                attention_mask=attention_mask,
                position_ids=position_ids,
                past_key_values=past_key_values,
                inputs_embeds=inputs_embeds,
                use_cache=use_cache,
                output_attentions=output_attentions,
                output_hidden_states=output_hidden_states,
                return_dict=return_dict)

        hidden_states = transformers_outputs.last_hidden_state
        logits = self.score(hidden_states)

        batch_size = input_ids.shape[0]

        sequence_lengths = torch.eq(input_ids, self.config.pad_token_id).int().
           ↪ argmax(-1) - 1
        sequence_lengths = sequence_lengths % input_ids.shape[-1]
        sequence_lengths = sequence_lengths.to(logits.device)
        pooled_logits = logits[torch.arange(batch_size, device=logits.device),
           ↪ sequence_lengths]

        loss = None
        if labels is not None:
            labels = labels.to(logits.device)
            print(labels.shape, batch_size, pooled_logits.view(-1, self.num_labels
               ↪ ).shape)
            loss_fct = CrossEntropyLoss()
            loss = loss_fct(pooled_logits.view(-1, self.num_labels), labels.view
               ↪ (-1))

        if not return_dict:
            output = (pooled_logits,) + transformer_outputs[1:]
            return ((loss,) + output) if loss is not None else output

        return SequenceClassifierOutputWithPast(
            loss=loss,
            logits=pooled_logits,
            past_key_values=transformers_outputs.past_key_values,
            hidden_states=transformers_outputs.hidden_states,
            attentions=transformers_outputs.attentions,
        )
```

This class simply inherent the base model where is lama-instruct version and add a linear layer at top of it. Forward method will get the last hidden state from transforms output of base model.Then pass it through the linear layer to classify the sequence.Then based on the last PAD token of the each sequence the logits would be obtained. Then from CrossEntropyLoss, loss would be calculated, and back propagated by the trainer. The Sequence Classifier output with past would return loss, logits, past, key, value and hidden states and attentions. Loading model with same quantiziation configure(bnb) as follow:

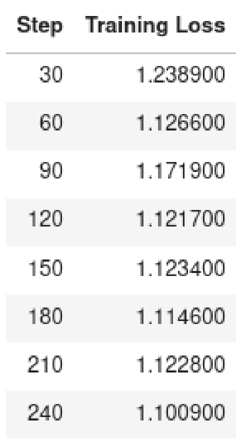
```python
config = AutoConfig.from_pretrained(model_name, num_labels=len(id2label))
config.id2label = id2label
config.label2id = label2id
config.pad_token_id = tokenizer.pad_token_id
model = CLF.from_pretrained(
    model_name,
    config=config,
    num_labels=len(id2label),
    quantization_config=bnb_config,
    torch_dtype=torch.bfloat16,
    attn_implementation="eager",
)
```

The configuration of LORA is same as previous section.(We set r to 16 since the base LORA paper said so). The training set also would be formatted by prompt template like before. But we create a custom collator for this since if labels goes with the DataCollatorForLanguageModeling then labels would be padded to! Our data collator is as follow:

```python
class CustomDataCollator:
    def __init__(self, tokenizer):
        self.tokenizer = tokenizer

    def __call__(self, features):
        batch = self.tokenizer.pad(
            features,
            padding=True,
            return_tensors="pt"
        )
        return batch
```

The loss per 30 steps for this model has been shown in figure 4:



Step	Training Loss
30	1.238900
60	1.126600
90	1.171900
120	1.121700
150	1.123400
180	1.114600
210	1.122800
240	1.100900

Figure 4: Loss per each 30 steps

In order to evaluate the model on 5 percentage of validation dataset, we can use the following data loader to obtain all predictions alongside labels:

```python
model.eval()

all_predictions = []
all_labels = []

with torch.no_grad():
    for batch in val_loader:
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)

        outputs = model(input_ids=input_ids, attention_mask=attention_mask)
        logits = outputs.logits
        predictions = torch.argmax(logits, dim=-1)

        all_predictions.extend(predictions.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())
```

The accuracy score is: Also the hyper parameters for this model is same as before to equally judge them with respect to just one factor which is accuracy.

lora_alpha	same
r	same
lora_dropout	same
Number Of Trainable Parameters	42,020,912
Percentage data(val and train)	5%
Training Batch size	same
Gradient accumulation steps	same
Max Steps	120
Learning Rate	same
Total Time	
Target Modules of LORA	same

Table 9: Hyper Parameters

Different metrics to differentiate two models are illustrated in table 10.

Model	First QLORA	Second QLORA
Accuracy for 5 percentage of validation matche	37%	0.35%
Training Time	2:09:30	0:49:36
Number Of Trainable Parameters	41,943,040	42,020,912

Table 10: 3 results of different models

The first QLORA model takes 600 steps for 2 hours and 10 minutes to reach the accuracy for 37% while second model with QLORA with 240 steps for 50 minutes reach the 35% accuracy.The second model since having classifier layer has even more parameters than first model.I think with 10 percentage of data we for the second model we can reach even more higher accuracy.Although because of lack of time we did not try this.

References

[1] Tom B. Brown et al. "Language Models are Few-Shot Learners". In: *Advances in Neural Information Processing Systems*. NeurIPS. 2020.

[2] Wikipedia contributors. *Fine-tuning (deep learning)*. Accessed: 2023-05-22. 2023. URL: `https://en.wikipedia.org/wiki/Fine-tuning_(deep_learning)`.

[3] Tim Dettmers et al. *QLORA: Efficient Finetuning of Quantized LLMs*. 2023. arXiv: `2305.14314 [cs.LG]`. URL: `https://arxiv.org/abs/2305.14314`.

[4] Hugging Face. *Parameter-Efficient Fine-Tuning (PEFT) - P-tuning*. Accessed: 2024-05-27. 2024. URL: `\url{https://huggingface.co/docs/peft/main/en/package_reference/p_tuning#peft.PromptEncoderConfig}`.

[5] Edward J. Hu et al. *LoRA: Low-Rank Adaptation of Large Language Models*. 2021. arXiv: `2106.09685 [cs.CL]`. URL: `https://arxiv.org/abs/2106.09685`.

[6] Brian Lester, Rami Al-Rfou, and Noah Constant. "The Power of Scale for Parameter-Efficient Prompt Tuning". In: *arXiv preprint arXiv:2104.08691* (2021). URL: `https://arxiv.org/abs/2104.08691`.