

Advanced NLP Assignment 3

12112627 李乐平

Q1

(1) Explain three different points between P-Tuning V1 and P-Tuning V2.

- **Applied tasks:** P-Tuning V1 can apply to Knowledge Probing and NLU tasks, while P-Tuning V2 adapts to Sequence Tagging and NLU tasks.
- **Classification head:** P-Tuning V1 uses a language modeling head to predict verbalizers, which are natural language tokens that correspond to labels. P-Tuning V2 uses a linear classifier on top of the [CLS] token or the target tokens, which is simpler and more compatible with sequence labeling tasks.
- **Reparameterization:** P-Tuning V1 always uses a reparameterization encoder such as an MLP to transform the trainable embeddings. P-Tuning V2 finds that the usefulness of reparameterization depends on the tasks and datasets, and sometimes a simple embedding layer is enough.

(2) Write code to finetune Tiny-BERT with P-Tuning v2 method using the MRPC Dataset.

```
In [1]: # !python -m pip install --user --upgrade peft datasets
# !python -m pip install --upgrade numpy
# !python -m pip install --user transformers
# !python -m pip install transformers
# !pip list

In [2]: import transformers

/usr/local/lib/python3.10/dist-packages/tqdm/auto.py:21: TqdmWarning: IProgress not found. Please update jupyter and ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html (https://ipywidgets.readthedocs.io/en/stable/user_install.html)
  from .autonotebook import tqdm as notebook_tqdm
```

```
In [3]: import transformers
from transformers import AutoTokenizer, AutoModelForSequenceClassification, TrainingArguments, Trainer
from datasets import load_dataset, load_metric
from peft import get_peft_config, get_peft_model, PrefixTuningConfig, TaskType, PeftType, PromptEncoderConfig

import torch
import transformers
import numpy as np
import evaluate
import warnings

warnings.simplefilter("ignore")

# Load the accuracy metric
global_metric = load_metric("./accuracy/accuracy.py")

# Load the MRPC dataset
dataset = load_dataset(
    path = "json",
    data_dir = "./Q1_MRPC_dataset",
    data_files = {"train": "train.jsonl", "test": "test.jsonl", "validation": "validation.jsonl"}
)

# Load the Tiny-BERT model
model_dir = "./Q1_model_checkpoint_tinybert/"
tokenizer = AutoTokenizer.from_pretrained(model_dir)
model = AutoModelForSequenceClassification.from_pretrained(model_dir, num_labels=2)

# # Apply P-Tuning V2 to the model, where "prefix_projection = True" indicates P-Tuning V2 is used
# peft_config = PrefixTuningConfig(task_type=TaskType.SEQ_CLS, num_virtual_tokens=5, prefix_projection=True)
# model = get_peft_model(model, peft_config)

# Apply P-Tuning V1 to the model
peft_config = PromptEncoderConfig(task_type=TaskType.SEQ_CLS, num_virtual_tokens=5)
model = get_peft_model(model, peft_config)

# Tokenize the dataset
def tokenize_function(example):
    return tokenizer(example["text1"], example["text2"], padding=True, truncation=True, max_length=128)

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

# Define the optimizer and the Learning rate scheduler
# # For P-tuning V2
# optimizer = torch.optim.AdamW(model.parameters(), lr=2e-4)
# scheduler = transformers.get_linear_schedule_with_warmup(optimizer, num_warmup_steps=0, num_training_steps=len(dataset["train"]) * 100)

# For P-tuning V1
optimizer = torch.optim.AdamW(model.parameters(), lr=3e-3)
scheduler = transformers.get_linear_schedule_with_warmup(optimizer, num_warmup_steps=0, num_training_steps=len(dataset["train"]) * 100)

# Define the training arguments
training_args = TrainingArguments(
    output_dir="./results",
    evaluation_strategy="epoch",
    save_strategy="epoch",
    # Learning_rate=2e-4, # For P-Tuning V2
    learning_rate=3e-3,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=64,
    num_train_epochs=100,
    # weight_decay=1e-4, # For P-Tuning V2
    weight_decay=2e-5,
    push_to_hub=False,
    logging_dir="./logs",
    logging_steps=10,
    load_best_model_at_end=False,
    metric_for_best_model="accuracy",
    greater_is_better=True,
)

# Define the metric computing function
def compute_metrics(eval_preds):
    metric = global_metric
    logits, labels = eval_preds
    predictions = np.argmax(logits, axis=-1)
    return metric.compute(predictions=predictions, references=labels)

# Define the training function
def m_train():
    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=tokenized_dataset["train"],
        eval_dataset=tokenized_dataset["validation"],
        data_collator=None,
        tokenizer=tokenizer,
        compute_metrics=compute_metrics,
    )
    trainer.train()

# print(dataset["train"])

# Train the model
m_train()
```

Some weights of BertForSequenceClassification were not initialized from the model checkpoint at ./Q1_model_checkpoint_tinybert/ and are newly initialized: ['classifier.weight', 'classifier.bias']

You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

You're using a BertTokenizerFast tokenizer. Please note that with a fast tokenizer, using the `__call__` method is faster than using a method to encode the text followed by a call to the `pad` method to get a padded encoding.

Epoch	Training Loss	Validation Loss	Accuracy
1	0.667700	0.615650	0.683824
2	0.557900	0.600451	0.683824
3	0.594400	0.591131	0.696078
4	0.578600	0.585770	0.691176
5	0.657700	0.574661	0.696078
6	0.589000	0.568421	0.723039
7	0.552800	0.559171	0.710784
8	0.549000	0.553132	0.708333
9	0.596000	0.549417	0.718137
10	0.574100	0.546513	0.725490
11	0.558300	0.574469	0.710784
12	0.522300	0.540062	0.718137
13	0.551800	0.532916	0.727941
14	0.591000	0.534194	0.705882
15	0.604200	0.531001	0.723039
16	0.537300	0.534366	0.730392
17	0.559100	0.521835	0.718137
18	0.540900	0.526082	0.730392
19	0.508500	0.522787	0.732843
20	0.560200	0.526338	0.735294
21	0.493500	0.532187	0.740196
22	0.613600	0.519039	0.727941
23	0.545600	0.508480	0.723039
24	0.663200	0.507789	0.727941
25	0.461400	0.519288	0.727941
26	0.506200	0.515907	0.732843
27	0.505900	0.512596	0.732843
28	0.529400	0.508484	0.727941
29	0.540600	0.505932	0.735294
30	0.598600	0.497445	0.740196
31	0.513000	0.503402	0.737745
32	0.478000	0.511460	0.742647
33	0.497300	0.496613	0.742647
34	0.531000	0.491063	0.745098
35	0.499800	0.503439	0.747549
36	0.515600	0.508357	0.742647
37	0.514500	0.487564	0.745098
38	0.586300	0.495335	0.742647
39	0.602200	0.487529	0.750000
40	0.560900	0.492452	0.752451
41	0.527000	0.484340	0.762255
42	0.534500	0.485553	0.752451
43	0.490100	0.483834	0.750000
44	0.545000	0.489478	0.752451
45	0.502000	0.486107	0.754902
46	0.574700	0.485467	0.759804
47	0.463200	0.497947	0.764706
48	0.545000	0.494636	0.759804
49	0.533700	0.477031	0.757353
50	0.577200	0.481729	0.762255
51	0.530700	0.479748	0.754902
52	0.525400	0.467456	0.769608
53	0.493300	0.461070	0.769608
54	0.493400	0.458720	0.769608
55	0.508100	0.468278	0.764706
56	0.443900	0.471589	0.764706
57	0.488400	0.465500	0.764706
58	0.551600	0.461250	0.767157
59	0.534000	0.457032	0.772059
60	0.483100	0.466518	0.776961
61	0.471900	0.459065	0.767157
62	0.539300	0.466955	0.772059
63	0.538100	0.453998	0.767157
64	0.466800	0.460701	0.769608
65	0.478400	0.462919	0.774510
66	0.520800	0.454992	0.762255
67	0.455300	0.457343	0.774510
68	0.516600	0.454347	0.772059
69	0.547100	0.449792	0.774510
70	0.518200	0.444926	0.774510
71	0.470800	0.444348	0.769608
72	0.502000	0.455310	0.772059
73	0.505200	0.451216	0.769608
74	0.624200	0.452157	0.772059
75	0.500500	0.450320	0.779412
76	0.496200	0.444709	0.774510
77	0.489300	0.446481	0.774510
78	0.577800	0.445873	0.772059
79	0.551200	0.451090	0.769608
80	0.431000	0.449341	0.781863
81	0.561900	0.442284	0.776961
82	0.479600	0.442873	0.774510
83	0.563700	0.449420	0.781863

Epoch	Training Loss	Validation Loss	Accuracy
84	0.541400	0.442921	0.779412
85	0.466100	0.448166	0.779412
86	0.477000	0.442990	0.776961
87	0.488000	0.443780	0.774510
88	0.482200	0.445645	0.776961
89	0.479400	0.448386	0.784314
90	0.461700	0.442891	0.784314
91	0.500300	0.443472	0.781863
92	0.401600	0.443937	0.781863
93	0.481900	0.443317	0.781863
94	0.488900	0.445842	0.784314
95	0.517400	0.442357	0.779412
96	0.435900	0.442563	0.776961
97	0.454400	0.441283	0.779412
98	0.532500	0.441215	0.779412
99	0.530300	0.441079	0.779412
100	0.492500	0.441259	0.776961

```
In [4]: # Define the evaluation function
def m_evaluate():
    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=None,
        eval_dataset=tokenized_dataset["test"],
        data_collator=None,
        tokenizer=tokenizer,
        compute_metrics=compute_metrics,
    )
    result = trainer.evaluate()
    return result

m_evaluate()
```

[27/27 00:00]

```
Out[4]: {'eval_loss': 0.49014005064964294,
'eval_accuracy': 0.7576811594202899,
'eval_runtime': 0.7411,
'eval_samples_per_second': 2327.537,
'eval_steps_per_second': 36.431}
```

Q2

Efficient finetuning with adapter method.

1 Explain the adapter-based finetuning. You can also refer to the Adapter Hub blog.

Answer:

Adapter-based fine-tuning is a methodology used in natural language processing (NLP) where pre-trained language models, such as BERT or GPT, are augmented with task-specific adapters rather than retraining the entire model from scratch. Adapters are small, task-specific neural network modules that are inserted into the pre-trained model's architecture, connecting to its hidden layers. During fine-tuning, only the parameters within the adapters and a minimal set of surrounding parameters are updated, leaving the majority of the pre-trained model untouched. This approach allows for efficient adaptation to new tasks with significantly fewer training resources compared to full model retraining, enabling faster deployment and reducing the risk of overfitting to the specific task at hand. Adapter-based fine-tuning strikes a balance between leveraging the generalization capabilities of pre-trained models and adapting to task-specific requirements.

2 Suppose the dimension of the input to the adapter is d_m . How to change the dimension of the hidden state before the non-linear function to $\frac{8d_m}{3}$?

Answer: According to the source code, we can find out that the dimensional change is done in the following codes:

```
In [ ]: # 31 class Adapter(nn.Module):
# 32     """
# 33     Implementation of a sequential bottleneck adapter block.
# 34     """
# 35
# 36     def __init__(
# 37         self,
# 38         adapter_name,
# 39         input_size,
# 40         down_sample,
# 41         config: BnConfig,
# 42     ):
#
# ...
# 73         if config["phm_layer"]:
# 74             # Linear down projection of the input
# 75             seq_list.append(PHMLayer(adapter_name, self.input_size, self.down_sample, "down", config))
# 76         else:
# 77             seq_list.append(nn.Linear(self.input_size, self.down_sample))
# 78
# 79         # select non-linearity
# 80         self.non_linearity = ActivationFunctionClass(config["non_linearity"].Lower())
# 81
# 82         seq_list.append(self.non_linearity)
```

Note that the projection is done in line 75/77, so what we need to do is set down_sample = 8 * dm // 3.

```
In [ ]: # adapter = Adapter("MyAdapter", my_input_size, 8 * my_input_size // 3) # Sample usage
```

Q3

Select an LLM like ChatGPT, 讯飞星火, etc. Design prompts to write story for 5-year-old child. The topic, style, length and clues can be provided as constraints.

1. Write your prompt and generated story.

Prompt:

Topic: Adventure in a Magical Forest

Style: Playful and Imaginative

Length: 300-500 words

Clues: A talking squirrel, a flying rainbow butterfly, a friendly dragon, a hidden treasure

Generated Story:

Once upon a time, in a land far, far away, there was a magical forest known as Enchanted Eldwood. This whimsical woodland was home to creatures of extraordinary charm and wonder. Our adventure begins with a curious explorer named Lily, who, armed with a heart full of curiosity and a backpack full of snacks, stepped into the lush greenery of Enchanted Eldwood.

As Lily ventured deeper into the forest, the trees whispered secrets, and the leaves giggled with delight. Suddenly, a chittering sound caught her attention. To her surprise, a talking squirrel named Nutters appeared, perched on a tree branch.

"Hello, dear adventurer! Welcome to Enchanted Eldwood," Nutters greeted with a mischievous twinkle in his eye. "I've been waiting for someone like you to share a grand adventure!"

With Nutters as her guide, Lily continued her journey, the foliage parting to reveal a breathtaking sight—a flying rainbow butterfly gracefully dancing in the air. Its wings shimmered with every color of the spectrum, leaving a trail of magic in its wake.

"Meet Iris, the Rainbow Flutterfly!" Nutters exclaimed. "Legend has it that following Iris leads to the heart of the forest, where a friendly dragon guards a hidden treasure."

Eagerly, Lily followed Iris through the enchanted grove until they reached a cozy clearing. There, nestled among the wildflowers, was a dragon named Ember, its scales reflecting the warm hues of a crackling bonfire. To Lily's surprise, Ember greeted her with a gentle smile.

"Welcome, kind adventurer! Nutters told me you would come seeking the treasure," Ember rumbled in a voice as soothing as the rustling leaves.

Ember shared the tale of the hidden treasure—a magical chest containing dreams, laughter, and joy. To unlock it, Lily needed to solve a riddle that echoed through the ancient trees.

As Lily pondered the riddle, Nutters gathered acorns, and Iris fluttered around in anticipation. With a gleam in her eye, Lily solved the riddle, and the magical chest opened, releasing a burst of shimmering light that filled the entire forest.

The once-hidden treasure transformed Enchanted Eldwood into an even more enchanting haven, where laughter echoed through the trees and dreams floated on the breeze.

With gratitude in her heart, Lily bid farewell to her newfound friends—Nutters, Iris, and Ember. As she stepped out of the magical forest, she carried with her the joy and wonder of Enchanted Eldwood, forever grateful for the playful adventure that had unfolded among the talking squirrel, flying rainbow butterfly, friendly dragon, and the hidden treasure that sparked a world of magic.

2. Discuss approaches to improve the diversity and quality of generated story.

Answer:

First, prompts should encourage specificity, prompting the model with detailed settings, character backgrounds, or unique scenarios to steer the narrative towards more intricate and imaginative paths. Additionally, incorporating emotional cues and thematic elements in prompts can help infuse stories with depth and resonance. To address diversity, prompts should encompass a broad spectrum of genres, cultures, and perspectives, fostering inclusivity and avoiding unintentional biases. Experimenting with open-ended prompts that allow for creative interpretation can also stimulate the model to produce narratives that transcend conventional story arcs, resulting in a richer and more varied storytelling output.

.