درستی سنجی مدلهای HDL

نام و نام خانوادگی دستیار آموزشی: بهزاد جنتی / مهسا ابرقانی اقدم

تمرین امتیازی

پاییز ۱۴۰۴

۱ بخش اول: مقدمه و اهداف تمرین

مقدمه: در دنیای پیچیده طراحی سخت‌افزار، اطمینان از صحت عملکرد مدل‌های توصیف سخت‌افزار (HDL) یکی از چالش‌های اصلی است. Assertion-Based Verification (ABV) یک روش قدرتمند برای این منظور است که در آن، ویژگی‌ها و رفتارهای مورد انتظار مدار به صورت خاصیت‌هایی (Properties) یا همان Assertion بیان می‌شوند. نوشتن این Assertionها به صورت دستی، فرآیندی زمان‌بر و مستعد خطا است. با ظهور مدل‌های زبانی بزرگ (LLMs)، فرصت‌های جدیدی برای خودکارسازی این فرآیند به وجود آمده است.

هدف تمرین:

در این تمرین، شما با استفاده از تکنیک‌های پیشرفته پردازش زبان طبیعی و یادگیری عمیق، یک مدل زبانی بزرگ (Llama 3.2 1B) را برای تولید خودکار SystemVerilog Assertion از روی کدهای HDL آموزش خواهید داد.

شما در طی این فرآیند با مراحل زیر آشنا خواهید شد:

۱. راه‌اندازی محیط توسعه در Google Colab و آشنایی با پلتفرم Hugging Face.

۲. بارگذاری یک مدل زبانی بزرگ از پیش آموزش‌دیده و بهینه‌سازی آن با تکنیک کوانتیزاسیون (Quantization).

۳. بررسی و پیش‌پردازش دیتاست تخصصی VERT که شامل زوج‌های کد HDL و Assertion متناظر است.

۴. آموزش مدل با استفاده از روش نوین و بهینه LoRA (Low-Rank Adaptation) که یک تکنیک Parameter-Efficient Fine-Tuning (PEFT) است.

۵. ارزیابی عملکرد مدل قبل و بعد از فرآیند Fine-tuning و تحلیل نتایج به دست آمده.

۲ بخش دوم: پیش‌نیازها و آماده‌سازی محیط

قبل از شروع کدنویسی، لازم است با ابزارهای اصلی این تمرین آشنا شوید و دسترسی‌های لازم را فراهم کنید.

الف) آشنایی با Hugging Face و دریافت توکن دسترسی Hugging Face یک پلتفرم پیشرو در زمینه یادگیری ماشین است که به عنوان "GitHub برای مدل‌های هوش مصنوعی" شناخته می‌شود. در این پلتفرم می‌توانید به هزاران مدل، دیتاست و ابزار دسترسی داشته باشید. برای استفاده از مدل‌های خاص (مانند مدل Llama 3.2) نیاز به

یک حساب کاربری و توکن دسترسی دارید.

دستورالعمل:

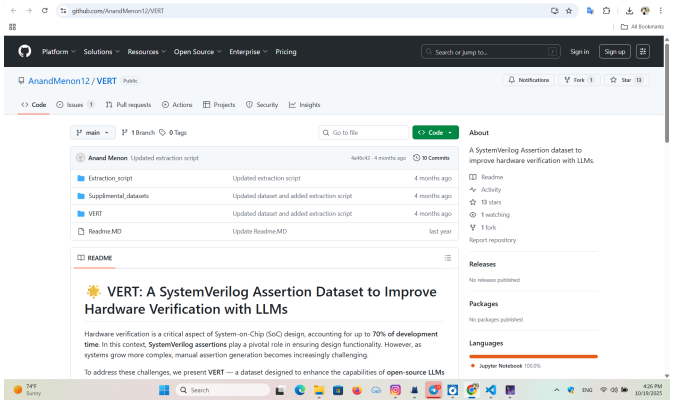
به سایت huggingface.co مراجعه کرده و یک حساب کاربری ایجاد کنید. به بخش Settings -> Access Tokens در پروفایل خود بروید یا از لینک مستقیم huggingface.co/settings/tokens استفاده کنید. یک توکن جدید (New token) با سطح دسترسی Read ایجاد کنید. این توکن را کپی کرده و در جایی امن نگه دارید. در ادامه تمرین از آن برای احراز هویت استفاده خواهیم کرد. به صفحه مدل meta-llama/Llama-3.2-1B در Hugging Face مراجعه کرده و درخواست دسترسی (Request Access) خود را ثبت کنید. معمولاً دسترسی به سرعت تایید می‌شود.

ب) آشنایی با دیتاست VERT

دیتاست VERT یک مجموعه داده تخصصی برای وظیفه تولید SystemVerilog Assertion از روی کدهای Verilog است. این دیتاست از طریق یک مخزن گیت‌هاب در دسترس است.

دستورالعمل:

به مخزن گیت‌هاب دیتاست در آدرس `https://github.com/AnandMenon12/VERT` مراجعه کنید. ساختار مخزن و به خصوص فایل‌های داده را بررسی کنید.



سوال ۱: دیتاست VERT چه ساختاری دارد؟ هر نمونه داده (Sample) شامل چه فیلدهایی است و هر فیلد چه اطلاعاتی را در خود جای داده است؟ مشاهدات خود را در چند خط توضیح دهید.

پاسخ خود را در در گزارش بنویسید.

۳ بخش سوم: مراحل پیاده‌سازی و کدنویسی

در این بخش، کدهای ارائه شده در نوت‌بوک را به همراه توضیحات و سوالات مرتبط دنبال کنید. شما باید تمام سلول‌های کد را اجرا کرده و به سوالات مطرح شده در سلول‌های متنی پاسخ دهید.

۱. نصب کتابخانه‌ها و راه‌اندازی محیط توضیح: در اولین قدم، کتابخانه‌های مورد نیاز پایتون را نصب می‌کنیم. این کتابخانه‌ها شامل موارد زیر هستند:

transformers: برای بارگذاری و کار با مدل‌های از پیش آموزش‌دیده.

datasets: برای مدیریت و پردازش دیتاست‌ها.

peft: برای پیاده‌سازی تکنیک‌های Fine-tuning بهینه (مانند LoRA).

bitsandbytes: برای کوانتیزاسیون مدل و کاهش مصرف حافظه.

trl: برای ساده‌سازی فرآیند آموزش مدل‌های Transformer.

پس از نصب، نسخه PyTorch و وضعیت دسترسی به GPU را بررسی می‌کنیم تا از آماده بودن محیط برای اجرای محاسبات سنگین مطمئن شویم.

سوال ۲: چرا بررسی نسخه PyTorch و فعال بودن CUDA در ابتدای یک پروژه یادگیری عمیق اهمیت دارد؟ GPU استفاده شده در این نوت‌بوک (Tesla T4) چه خصوصیاتی دارد که آن را برای این تمرین مناسب می‌کند؟

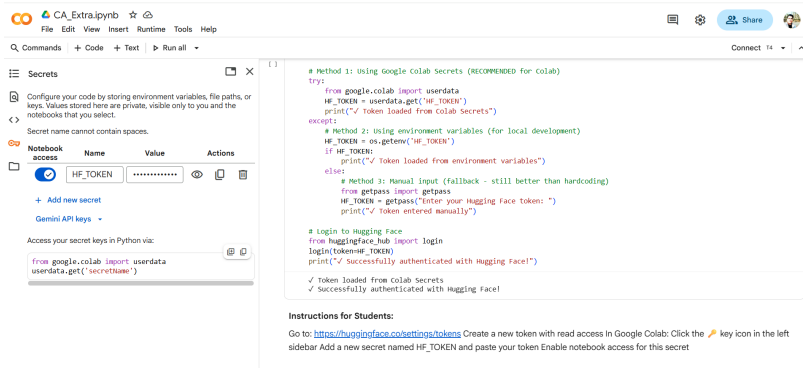
پاسخ خود را در گزارش بنویسید

۲. مدیریت امن توکن Hugging Face (امتیاز اضافه) توضیح: قراردادن مستقیم توکن‌های دسترسی در کد (Hardcoding) یک عمل بسیار ناامن است. در این بخش، روش صحیح برای استفاده از توکن Hugging Face در

محیط Google Colab با استفاده از Colab Secrets آموزش داده می‌شود. این روش اطلاعات حساس شما را به صورت امن ذخیره می‌کند.

دستورالعمل:

در محیط Google Colab، روی آیکون کلید در نوار کناری سمت چپ کلیک کنید. یک Secret جدید با نام HF_TOKEN ایجاد کنید. توکنی که از سایت Hugging Face کپی کردید را در قسمت Value وارد کنید. دسترسی نوت‌بوک (Notebook access) را برای این Secret فعال کنید.



سوال ۳: چرا استفاده از Colab Secrets یا متغیرهای محیطی (Environment Variables) برای ذخیره توکن، روش بهتری نسبت به نوشتن مستقیم آن در کد است؟ (در حد یک یا دو خط توضیح دهید)

پاسخ خود را در گزارش بنویسید

درخواست دسترسی به مدل Llama 3.2

مدل Llama 3.2 توسط شرکت Meta توسعه داده شده و برای استفاده از آن باید درخواست دسترسی ثبت کنید.

دستورالعمل: ۱. با حساب کاربری خود وارد Hugging Face شوید. ۲. به صفحه مدل meta-llama/Llama-3.2-1B در لینک زیر بروید: `https://huggingface.co/meta-llama/Llama-3.2-1B` ۳. شرایط استفاده را مطالعه و فرم درخواست دسترسی را تکمیل و ارسال کنید. معمولاً دسترسی در مدت کوتاهی (از چند دقیقه تا چند ساعت) از طریق ایمیل تایید می‌شود. ۴. خواسته شده: اسکرین‌شاتی از ایمیل تایید دسترسی خود را در گزارش نهایی قرار دهید.

۳. بارگذاری مدل و بهینه‌سازی با کوانتیزاسیون

توضیح: در این مرحله، مدل اصلی (meta-llama/Llama-3.2-1B) را از Hugging Face بارگذاری می‌کنیم. از آنجایی که مدل‌های زبانی بزرگ به حافظه GPU زیادی نیاز دارند، ما از یک تکنیک بهینه‌سازی به نام کوانتیزاسیون ۴ بیتی (4-bit Quantization) استفاده می‌کنیم. با استفاده از کتابخانه bitsandbytes، وزن‌های مدل که به صورت پیش‌فرض با دقت ۳۲ بیت (float32) هستند، با دقت ۴ بیت نمایش داده می‌شوند. این کار باعث کاهش چشمگیر مصرف حافظه شده و به ما اجازه می‌دهد مدل را روی یک GPU با حافظه محدود (مانند Tesla T4 با ۱۵

گیگابایت VRAM) بارگذاری و آموزش دهیم. همچنین Tokenizer مدل را بارگذاری می‌کنیم که وظیفه تبدیل متن به توکن‌های عددی (که برای مدل قابل فهم است) و برعکس را بر عهده دارد.

مدل‌های زبانی بزرگ حافظه گرافیکی (VRAM) زیادی مصرف می‌کنند. مدل Llama-3.2-1B حتی در اندازه کوچک خود، در حالت عادی (32-bit precision) به بیش از GB۴ حافظه نیاز دارد. برای اینکه بتوانیم این مدل را در محیط رایگان Google Colab (که معمولاً GPU با حدود VRAM GB۱۵ دارد) آموزش دهیم، از تکنیک کوانتیزاسیون (Quantization) استفاده می‌کنیم. در اینجا، ما مدل را با دقت 4-bit بارگذاری می‌کنیم. این کار حجم مدل در حافظه را به شکل چشمگیری کاهش می‌دهد.

BitsAndBytesConfig: این کلاس به ما اجازه می‌دهد پارامترهای کوانتیزاسیون را تعریف کنیم.

AutoTokenizer: وظیفه تبدیل متن به توکن (اعداد قابل فهم برای مدل) و برعکس را بر عهده دارد.

AutoModelForCausalLM: کلاس اصلی برای بارگذاری مدل‌های زبانی است که برای تولید متن (Causal Language Modeling) طراحی شده‌اند.

```python

# Section 2: Model and Tokenizer Loading
# ========================================

# 2.1: Define model name and quantization configuration
MODEL_NAME = "meta-llama/Llama-3.2-1B"

# Configure 4-bit quantization for memory efficiency
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.float16,
    bnb_4bit_use_double_quant=True,
)

print(f"Loading model: {MODEL_NAME} with 4-bit quantization...")

# 2.2: Load tokenizer
```

```
tokenizer = AutoTokenizer.from_pretrained(
    MODEL_NAME,
    trust_remote_code=True
)
# Set padding token if not present
if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token
    tokenizer.pad_token_id = tokenizer.eos_token_id

print(" Tokenizer loaded successfully!")

# 2.3: Load model with quantization
model = AutoModelForCausalLM.from_pretrained(
    MODEL_NAME,
    quantization_config=bnb_config,
    device_map="auto",
    trust_remote_code=True
)

print(" Model loaded successfully!")

# 2.4: Check GPU memory usage after loading
if torch.cuda.is_available():
    print(f"\nGPU Memory allocated{torch.cuda.memory_allocated(0) / 1e9
```

سوال ۴: در کد بالا، کلاس BitsAndBytesConfig برای کوانتیزاسیون مدل استفاده شده است. به طور خلاصه توضیح دهید هر یک از پارامترهای load_in_4bit، bnb_4bit_quant_type و bnb_4bit_compute_dtype چه نقشی دارند؟ استفاده از کوانتیزاسیون چه مزایا و معایب بالقوه‌ای دارد؟

پاسخ خود را در گزارش بنویسید

۴: بارگذاری و تحلیل دیتاست VERT

توضیح:

در این مرحله، دیتاست VERT را از مخزن گیت‌هاب آن دریافت کرده و بارگذاری می‌کنیم. پس از بارگذاری، ساختار داده‌ها را بررسی کرده و چند نمونه را برای درک بهتر وظیفه (تولید Assertion از روی کد HDL) نمایش می‌دهیم. همچنین، با رسم نمودار، آماری از ویژگی‌های دیتاست مانند طول کدها، طول Assertionها و توزیعAssertionهای همزمان (Synchronous) و غیرهمزمان(Asynchronous) به دست می‌آوریم.

```
# Section 3: Dataset Loading and Exploration
# ==========================================

# 3.1: Clone the VERT dataset repository
!git clone https://github.com/AnandMenon12/VERT.git

# 3.2: Load the dataset from the JSON file
try:
    with open('/content/VERT/VERT/VERT.json', 'r') as f:
        # Each line is a JSON object
        dataset_list = [json.loads(line) for line in f]
    print(f" Successfully loaded {len(dataset_list)} samples from VERT
except Exception as e:
    print(f"Error loading dataset: {e}")
    dataset_list = []

# 3.3: Split into train and test sets
from sklearn.model_selection import train_test_split
train_data, test_data = train_test_split(dataset_list, test_size=0.2, r

print(f"   Training samples: {len(train_data)}")
print(f"   Test samples: {len(test_data)}")

# 3.4: Display a few samples to understand the structure
```

```
print("\n" + "="*80)
print("SAMPLE DATA ANALYSIS")
print("="*80)
for i, sample in enumerate(train_data[:2], 1):
    print(f"\n—— Sample {i} ——")
    print(f"Code: {str(sample.get('Code', 'N/A'))[:200]}...")
    print(f"Assertion: {str(sample.get('Assertion', 'N/A'))[:200]}...")
    print(f"Synchronous: {sample.get('Synchronous', 'N/A')}")
    print(f"Clock: {sample.get('Clock', 'N/A')}")

# 3.5: Analyze and visualize dataset statistics
assertion_lengths = [len(str(s.get('Assertion', '')).split()) for s in
code_lengths = [len(str(s.get('Code', '')).split()) for s in train_data
sync_count = sum(1 for s in train_data if s.get('Synchronous'))

plt.figure(figsize=(15, 5))
plt.subplot(1, 2, 1)
sns.histplot(assertion_lengths, bins=30, kde=True, color='blue')
plt.title('Distribution of Assertion Lengths (in words)')
plt.subplot(1, 2, 2)
sns.histplot(code_lengths, bins=30, kde=True, color='green')
plt.title('Distribution of Code Lengths (in words)')
plt.show()

print(f"\nPercentage of Synchronous Assertions in training data: {sync_
```

سوال ۵: الف) بر اساس نمودارهای توزیع طول کد و Assertion، چه نتیجه‌ای در مورد ماهیت دیتاست VERT می‌توان گرفت؟ آیا طول Assertionها با طول کدها رابطه مستقیمی دارد؟ ب) چرا تقسیم داده‌ها به دو مجموعه آموزش (Train) و آزمون (Test) یک مرحله ضروری در پروژه‌های یادگیری ماشین است؟

پاسخ خود را در گزارش بنویسید

۵: فرمت‌بندی داده‌ها برای آموزش

توضیح:

برای اینکه مدل Llama 3.2 یاد بگیرد که چگونه از یک کد HDL، یک Assertion تولید کند، باید داده‌ها را در یک فرمت خاص به نام InstructionFollowing آماده کنیم. در این فرمت، ما یک "دستورالعمل" (شامل توضیح وظیفه و کد ورودی) و یک "پاسخ" (Assertion مطلوب) ایجاد می‌کنیم. این کار به مدل کمک می‌کند تا بفهمد ورودی و خروجی مورد انتظار چه ساختاری دارند. تابع format_instruction این وظیفه را انجام می‌دهد.

```
# Section 4: Data Formatting for Instruction Tuning
# ==================================================

def format_instruction(sample):
    """Formats a sample into the LLaMA 3.2 instruction format."""
    code = sample.get('Code', '')
    assertion = sample.get('Assertion', '')
    context = f"This is a synchronous assertion that executes at {samp

    # LLaMA 3.2 instruction format
    return f"""<|begin_of_text|><|start_header_id|>system<|end_header_

You are an expert in hardware verification. Generate SystemVerilog asse

### HDL Code:
{code}

### Assertion:
{assertion}<|eot_id|>"""

# Format the datasets
train_texts = [format_instruction(sample) for sample in train_data]
test_texts = [format_instruction(sample) for sample in test_data]
```

```

train_dataset = Dataset.from_dict({"text": train_texts})
test_dataset = Dataset.from_dict({"text": test_texts})

print(" Datasets formatted for instruction tuning.")
print("\n—— Example of a Formatted Training Sample ——")
print(train_texts[0])
```

سوال ۶: در تابع format_instruction، از توکن‌های خاصی مانند <|begin_of_text|>، <|start_header_id|> و <|eot_id|> استفاده شده است. نقش این توکن‌های خاص در آموزش مدل‌های زبانی چیست و چرا استفاده از فرمت صحیح برای یک مدل خاص (در اینجا Llama 3.2) اهمیت دارد؟

پاسخ خود را در گزارش بنویسید

۴ بخش چهارم: آموزش مدل با استفاده از LoRA

۶. پیکربندی LoRA برای FineTuning بهینه

توضیح: Full Fine-Tuning یا آموزش کامل یک مدل زبانی بزرگ، به ده‌ها یا صدها گیگابایت حافظه GPU نیاز دارد. برای حل این مشکل، از روش‌های Parameter-Efficient Fine-Tuning (PEFT) استفاده می‌کنیم. یکی از محبوب‌ترین این روش‌ها، LoRA (LowRank Adaptation) است.

در تکنیک LoRA، به جای به‌روزرسانی تمام پارامترهای مدل اصلی (که میلیاردها پارامتر هستند)، وزن‌های اصلی مدل را فریز (Freeze) کرده و فقط ماتریس‌های کوچکی با رتبه پایین (LowRank) را که به لایه‌های مشخصی از مدل (معمولاً لایه‌های Attention) اضافه می‌شوند، آموزش می‌دهیم. این کار باعث می‌شود تعداد پارامترهای قابل آموزش به شدت کاهش یابد (معمولاً کمتر از 1 کل پارامترها) و فرآیند Finetuning با منابع سخت‌افزاری محدود نیز ممکن شود. در کد زیر، ما LoRA را برای لایه‌های کلیدی مدل (q_proj، k_proj، v_proj، و غیره) پیکربندی می‌کنیم.

```
# Prepare model for k-bit training
model = prepare_model_for_kbit_training(model)

# Configure LoRA for parameter-efficient fine-tuning
lora_config = LoraConfig(
```

```
    r=16,                          # Rank of the update matrices
    lora_alpha=32,                   # Scaling factor for LoRA weights
    target_modules=[
        "q_proj", "k_proj", "v_proj", "o_proj",
        "gate_proj", "up_proj", "down_proj"
    ],
    lora_dropout=0.05,               # Dropout probability for LoRA layers
    bias="none",
    task_type="CAUSAL_LM"
)

# Apply LoRA to the model
model = get_peft_model(model, lora_config)

print("\u2713 LoRA configured successfully!")

# Print the percentage of trainable parameters
model.print_trainable_parameters()

# Calculate parameter efficiency
trainable_params = sum(p.numel() for p in model.parameters() if p.requi
total_params = sum(p.numel() for p in model.parameters())
trainable_percentage = 100 * trainable_params / total_params
print(f"\n\ud83d\udca1 Percentage of trainable parameters: {trainable_p
```

سوال ۷: الف) تکنیک LoRA چیست و چگونه به کاهش چشمگیر مصرف حافظه و منابع محاسباتی در فرآیند Finetuning کمک می‌کند؟ ب) خروجی دستور model.print_trainable_parameters() چه چیزی را نشان می‌دهد؟ چرا تعداد پارامترهای قابل آموزش (trainable params) بسیار کمتر از کل پارامترهای مدل است؟

پاسخ خود را در گزارش بنویسید

۷. پیکربندی و اجرای فرآیند آموزش

توضیح: در این مرحله، هایپرپارامترهای فرآیند آموزش را با استفاده از کلاس TrainingArguments مشخص می‌کنیم. این پارامترها شامل تعداد epoch، اندازه batch، نرخ یادگیری (learning rate) و □ هستند. یکی از پارامترهای مهم در اینجا gradient_accumulation_steps است. این تکنیک به ما اجازه می‌دهد تا یک batch size بزرگتر را شبیه‌سازی کنیم. به این صورت که گرادیان‌ها در چندین مرحله (در اینجا ۴ مرحله) محاسبه و تجمیع می‌شوند و تنها پس از آن، وزن‌های مدل یک بار به‌روزرسانی می‌شوند. این کار برای آموزش روی GPUهای با حافظه محدود بسیار مفید است. در نهایت، با استفاده از کلاس SFTTrainer (Supervised Fine-tuning Trainer) از کتابخانه trl، فرآیند آموزش را آغاز می‌کنیم. این کلاس به طور خاص برای Instruction Fine-Tuning مدل‌های Transformer بهینه‌سازی شده است.

```
# Training Arguments Configuration
training_args = TrainingArguments(
    output_dir="./llama-vert-lora-checkpoints",
    num_train_epochs=1,                    # We will train for one full p
    per_device_train_batch_size=2,         # Batch size per GPU
    gradient_accumulation_steps=4,         # Accumulate gradients over 4
    gradient_checkpointing=True,           # Use gradient checkpointing to
    learning_rate=2e-4,                    # Learning rate
    fp16=True,                             # Use 16-bit precision for tra
    max_grad_norm=0.3,                     # Gradient clipping
    warmup_ratio=0.03,                     # Warmup steps ratio
    lr_scheduler_type="cosine",            # Cosine learning rate schedul
    logging_steps=25,                      # Log training progress every
    save_strategy="epoch",                 # Save model at the end of eac
    report_to="none"
)

# Initialize the SFTTrainer
trainer = SFTTrainer(
    model=model,
    train_dataset=train_dataset,
```

```
    peft_config=lora_config,
    dataset_text_field="text",
    max_seq_length=512,                    # Truncate long sequences
    tokenizer=tokenizer,
    args=training_args,
)

print("\u2713 Trainer initialized successfully!")

# Start the training process
print("\n" + "="*80)
print("\ud83d\ude80 STARTING FINE-TUNING PROCESS...")
print("="*80)
trainer.train()
print("\u2713 Training completed successfully!")

# Save the final fine-tuned model
final_model_path = "./llama-vert-lora-final"
trainer.save_model(final_model_path)
print(f"\u2713 Final model saved to: {final_model_path}")
```

سوال ۸: الف) در لاگ‌های خروجی فرآیند آموزش، مقدار Loss چه چیزی را نشان می‌دهد و چرا انتظار داریم این مقدار در طول زمان کاهش یابد؟

ب) با توجه به مقادیر per_device_train_batch_size و gradient_accumulation_steps، اندازه batch مؤثر (Effective Batch Size) در این آموزش چقدر است؟ این تکنیک چه مزیتی دارد؟

پاسخ خود را در گزارش بنویسید:

۵ بخش پنجم: ارزیابی و تحلیل نتایج

حالا زمان آن رسیده که ببینیم فرآیند Fine-tuning چقدر موفقیت‌آمیز بوده است. در این بخش، عملکرد مدل را پس از آموزش ارزیابی کرده و آن را با عملکرد اولیه مدل مقایسه می‌کنیم.

۸. ارزیابی مدل Fine-tune شده توضیح: در این بخش، ابتدا مدل آموزش‌دیده را به همراه آداپتورهای LoRA بارگذاری می‌کنیم. سپس یک pipeline برای تولید متن ایجاد کرده و از آن برای تولید Assertion روی همان ۱۰ نمونه از مجموعه آزمون که در بخش ارزیابی اولیه استفاده کردیم، بهره می‌گیریم. این کار به ما اجازه می‌دهد تا مقایسه‌ای مستقیم بین خروجی‌های مدل قبل و بعد از آموزش داشته باشیم.

```
# Import necessary libraries for evaluation
from peft import PeftModel
from transformers import AutoModelForCausalLM, AutoTokenizer, pipeline
import torch

# Path to the base model and the saved LoRA adapter
base_model_name = "meta-llama/Llama-3.2-1B"
adapter_path = "./llama-vert-lora-final"

print("\n\ud83d\udd0d Loading base model and merging LoRA adapter for

# Load the base model in 4-bit
base_model = AutoModelForCausalLM.from_pretrained(
    base_model_name,
    quantization_config=bnb_config,
    device_map="auto",
    trust_remote_code=True
)

# Load the LoRA model and merge it with the base model
model_finetuned = PeftModel.from_pretrained(base_model, adapter_path)
model_finetuned = model_finetuned.merge_and_unload() # Merge weights fo
```

```

tokenizer = AutoTokenizer.from_pretrained(base_model_name, trust_remote
if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token

print("\u2713 Fine-tuned model loaded and merged successfully!")

# Create a new pipeline for the fine-tuned model
pipe_finetuned = pipeline(
    "text-generation",
    model=model_finetuned,
    tokenizer=tokenizer,
    max_new_tokens=256,
    do_sample=True,
    temperature=0.7,
    top_p=0.95
)

print("\n\ud83d\udd0d Generating assertions with the FINE-TUNED model:\

# Run evaluation on the same test samples
# (The 'generate_assertion' function is defined in the pre-fine-tuning
finetuned_results = []
for i, sample in enumerate(test_data[:10], 1):
    print(f"\n{'='*80}")
    print(f"Test Sample {i}")
    print(f"{'='*80}")
    generated = generate_assertion(sample, pipe_finetuned)
    print(f"\ud83d\udd39 Ground Truth:\n   {sample['Assertion'][:200]}
```

```
print(f"\n\ud83d\udd2e Model Generated (AFTER Fine-tuning):\n    {ge
finetuned_results.append({'ground_truth': sample['Assertion'], 'ge
```

سوال ۹: نتایج تولید شده توسط مدل Fine-tune شده را با نتایج مدل پایه (قبل از آموزش) مقایسه کنید. حداقل دو نمونه را انتخاب کرده و به صورت کیفی توضیح دهید که عملکرد مدل در چه جنبه‌هایی (مانند رعایت سینتکس SystemVerilog، ارتباط منطقی با کد HDL، تولید ساختارهای صحیح property و endproperty) بهبود یافته است.

بخش ششم: تحلیل نهایی و جمع‌بندی در این بخش پایانی، شما باید بر اساس تمام مراحل انجام شده، یک تحلیل کلی از پروژه ارائه دهید.

سوال ۱۱ (تحلیل کلی): کل فرآیند انجام شده در این تمرین را در چند پاراگراف خلاصه کنید. هدف چه بود، چه مراحلی طی شد و نتیجه اصلی چه بود؟

پاسخ خود را در گزارش بنویسید:

سوال ۱۲ (نقاط ضعف و محدودیت‌ها): با بررسی خروجی‌های مدل Fine-tune شده، فکر می‌کنید بزرگترین نقاط ضعف آن چیست؟ آیا مدل توانایی تولید Assertionهای پیچیده (مانند Assertionهای مبتنی بر دنباله یا Sequence-based) را دارد؟ در چه مواردی بیشتر دچار خطا می‌شود؟

پاسخ خود را در گزارش بنویسید:

سوال ۱۳ (پیشنهادات برای بهبود): بر اساس مشاهدات خود و مفاهیمی که در کلاس آموخته‌اید، حداقل سه راهکار مشخص برای بهبود عملکرد این مدل پیشنهاد دهید. این راهکارها می‌تواند شامل استفاده از مدل‌های متفاوت، تغییر در داده‌ها یا روش پیش‌پردازش، تنظیم هایپرپارامترها، یا استفاده از معیارهای ارزیابی پیشرفته‌تر باشد. برای هر پیشنهاد، دلیل خود را نیز به طور خلاصه بیان کنید.

پاسخ خود را در گزارش بنویسید: