

2. LLaMA3-8B Fine-Tuning and Model evaluation

HF Transformers (1)

▪ Hugging Face Transformers

- NLP 모델을 위한 라이브러리
- **Model, Tokenizer, Trainer, Pipeline** 등을 포함하고 있음.
- 다양한 모델 Architecture 지원
 - **LLaMA(Large Language Model Meta AI)**, GPT 계열, OPT, Falcon, MPT, Claude
- HuggingFace repository의 pre-trained 모델들을 쉽게 활용 가능함.
- AutoModel/AutoTokenizer : 모델 구조에 따라서 적합한 모델 class를 loading함.
 - **AutoModel** 은 사용자가 주어진 모델 이름에 따라 올바른 모델 클래스(예: BertModel, RobertaMode, LLaMA3-8B 등)을 자동으로 선택하고 불러옴
 - **AutoTokenizer**는 모델의 이름에 따라 적합한 토큰라이저(예: BertTokenizer, GPT2Tokenizer 등)을 자동으로 선택하여 불러옴.

```
1 # 기본 Llama 3 모델 로드
2 from transformers import AutoModelForCausalLM
3 base_model = AutoModelForCausalLM.from_pretrained(
4     "meta-llama/Meta-Llama-3-8B",
5     quantization_config = quantization_config,
6     device_map = {"": 0}
7 )
```

- `{}:0` : 모델 전체를 GPU 0번 장치(첫번째 GPU)에 로드하라는 의미 == `cuda:0`
- `{}:"cuda"` : 가능한 GPU에 모델을 로드하라는 의미
- `device_map = "cpu"`
- CUDA(Compute Unified Device Architecture)는 NVIDIA에서 개발한 병렬 컴퓨팅 플랫폼 및 API로, GPU를 사용하여 계산을 수행할 수 있도록 설계된 기술임.

HF Transformer (1)

- Hugging Face : `pipeline()` 함수

- 사전 훈련된 모델을 간단하게 로드하고 사용할 수 있도록 도와주는 고수준 API임.
- 이 함수는 텍스트 생성, 번역, 감성 분석, 개체명 인식, 질의응답 등 다양한 NLP 작업을 간편하게 수행할 수 있음

■ 텍스트 생성(1)



1 pip install transformers torch

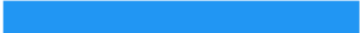
```
[6] 1 from transformers import pipeline
    2
    3 # KoGPT 모델을 사용한 텍스트 생성 파이프 라인
    4 text_generator = pipeline("text-generation", model="skt/kogpt2-base-v2")
    5
    6 # 입력 문장
    7 prompt = "오늘 날씨가 참 좋네요"
    8
    9 # 텍스트 생성 실행
   10 result = text_generator(prompt, max_length=50, do_sample=True, # 샘플링을 사용할지 여부(True : 랜덤성 증가, False: 항상 동일한 출력)
   11                        top_k=50, # 확률이 높은 50개의 단어만 고려함
   12                        top_p=0.95 ) # 확률의 누적합이 0.95 이하인 단어들만 선택함
   13 print(result[0]['generated_text'])
```



config.json: 100%  1.00k/1.00k [00:00<00:00, 39.3kB/s]

pytorch_model.bin: 100%  513M/513M [00:03<00:00, 143MB/s]

tokenizer.json: 100%  2.83M/2.83M [00:00<00:00, 20.8MB/s]

model.safetensors: 59%  304M/513M [00:04<00:01, 108MB/s]

Device set to use cpu

Truncation was not explicitly activated but `max_length` is provided a specific value, please use `truncation=True` to explicitly truncate

오늘 날씨가 참 좋네요~

가성비 좋은 #아기유모차 #유모차 #아기용품 #아기티비 #아기스타그램 #도치맘 #육아

■ 번역 (2)

```
[3] 1 !pip install transformers huggingface_hub
```

```
1 from huggingface_hub import login
2 from transformers import pipeline
3
4 # Hugging Face 로그인 (토큰 입력)
5 login(token="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")
6
7 # Facebook NLLB 모델 사용 (3.3B 모델: 높은 성능, 600M 모델: 속도 빠름)
8 model_name = "facebook/nllb-200-distilled-600M" # 또는 "facebook/nllb-200-3.3B"
9
10 # 한국어 → 영어 번역
11 translator_ko_en = pipeline("translation", model=model_name, src_lang="kor_Hang", tgt_lang="eng_Latn")
12 # 영어 → 한국어 번역
13 translator_en_ko = pipeline("translation", model=model_name, src_lang="eng_Latn", tgt_lang="kor_Hang")
14
15 # 테스트 문장
16 korean_text = "안녕하세요. 오늘 날씨는 어떨까요?"
17 english_translation = translator_ko_en(korean_text)
18 print("Korean to English:", english_translation[0]['translation_text'])
19
20 english_text = "Hello, how is the weather today?"
21 korean_translation = translator_en_ko(english_text)
22 print("English to Korean:", korean_translation[0]['translation_text'])
```

↔ Korean to English: How's the weather today?
English to Korean: 안녕하세요, 오늘 날씨가 어떠세요?

■ 감성 분석 (3)

```
[3] 1 !pip install transformers
```

```
1 from transformers import pipeline
2
3 # 한국어 감성 분석을 위한 pipeline 생성
4 sentiment_analyzer_ko = pipeline("text-classification", model="beomi/KcELECTRA-base-v2022")
5
6 # 테스트 문장 (한국어)
7 korean_text = ["이 제품 정말 마음에 들어요!", "완전 별로네요. 다시는 안 살 겁니다.", "그냥 보통이에요."]
8
9 # 감성 분석 수행
10 results_ko = sentiment_analyzer_ko(korean_text)
11
12 # 결과 출력
13 for text, result in zip(korean_text, results_ko):
14     print(f"Text: {text}")
15     print(f"Sentiment: {result['label']}, Score: {result['score']:.4f}")
16     print("-" * 40)
17
```

⚠ Some weights of ElectraForSequenceClassification were not initialized from the model checkpoint at beomi/KcELECTRA-base-v2022. You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

Device set to use cpu

Text: 이 제품 정말 마음에 들어요!

Sentiment: LABEL_1, Score: 0.5024

Text: 완전 별로네요. 다시는 안 살 겁니다.

Sentiment: LABEL_1, Score: 0.5238

Text: 그냥 보통이에요.

Sentiment: LABEL_0, Score: 0.5160

HF Transformers (2)

- HuggingFace repository의 pre-trained 모델들

- Transformer 모델

- BERT(Bidirectional Encoder Representations from Transformers) : 문장 내의 문맥을 양방향으로 이해할 수 있는 NLP 모델
 - GPT(Generative Pretrained Transformer): 언어 생성작업에 강력한 성능을 보이는 모델
 - RoBERTa(Robustly Optimized BERT Pretraining Approach) : BERT의 성능을 향상 시킨 버전

- Vision 모델

- ViT(Vision Transformer) : Transformer 구조를 사용해 이미지 분류 작업을 수행하는 모델
 - DeiT(Data-efficient Image Transformers) : ViT의 성능을 데이터 효율성을 높여 개선한 모델
 - CLIP(Contrastive Language-Image Pre-training) : 텍스트와 이미지를 함께 이해하고, 텍스트 설명에 맞는 이미지를 찾거나 반대로도 가능한 멀티모달 모델.

- 음성 모델

- Wav2Vec 2.0 : 음성 데이터를 처리해 텍스트로 변환하는 자동 음성 인식(ASR) 모델
 - Hubert : 음성 인식 및 처리 작업에 사용되는 모델로, 비지도 학습 방식을 채택

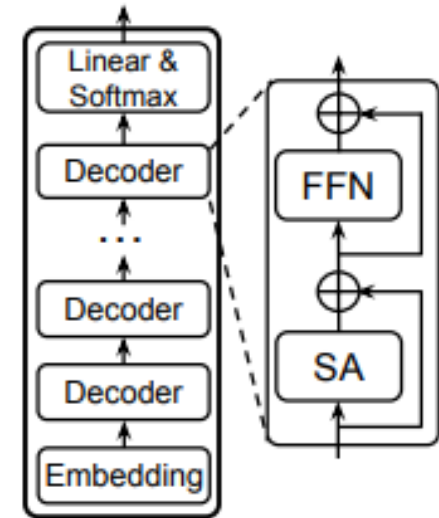
- 멀티모달 모델

- DALL-E : 텍스트 설명을 기반으로 이미지를 생성하는 모델
 - Flamingo : 텍스트와 이미지를 동시에 다루는 멀티모달 모델
 - Blip(Bootstrapping Language-Image Pretraining) : 텍스트와 이미지를 동시에 이해하고 생성하는 모델

HF Transformers(3)

- LLaMAForCausalLM

- LLaMA 모델을 이용해 텍스트 생성 작업을 수행하는데 최적화된 클래스임
- 'Causal Language Modeling'이란, 주어진 텍스트의 앞부분을 기반으로 다음 단어를 예측하는 언어 모델링 방식임.
(텍스트 생성, 자동 완성, 대화형 AI 등의 작업에 사용됨)
- Decoder-only Transformer
 - Embedding Layer : 입력을 고차원 벡터로 변환
 - Attention Module * 32
 - Attention Layer : 토큰간의 관계 파악
 - MLP Layer : Weights update
 - LM head Layer : 다음 토큰 예측함
- Fine-tuning 과정 동안 이 구조는 유지되고,
각 구조 내의 connection weights만 업데이트 됨



HF Transformers (4)

- Decoder-only Transformer 레이어들

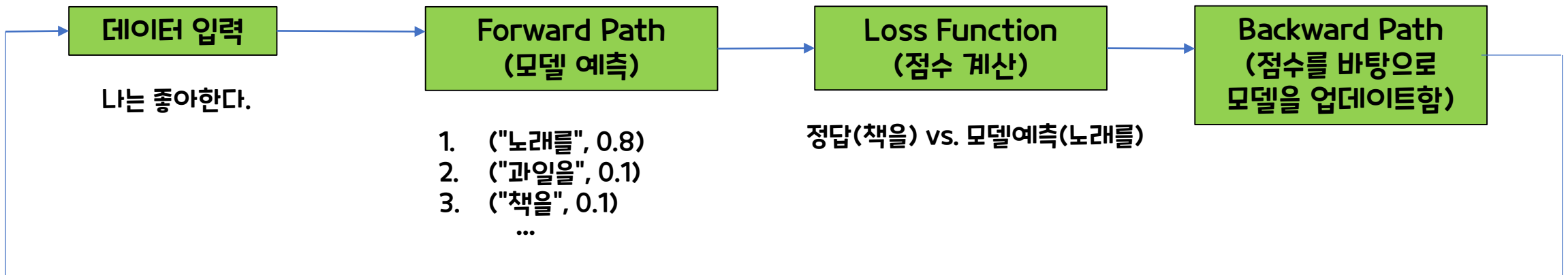
Embedding Layer	단어를 고정 길이의 벡터로 변환함
Self-Attention Layer	입력 시퀀스 내에서 각 토큰이 시퀀스의 다른 모든 토큰들과 상호작용하는 방법을 학습함 (각 토큰의 query, key, value를 계산하고 쿼리와 다른 토큰의 키들 간의 내적을 통해 중요도를 계산함. 이 중요도는 softmax를 통해 확률 분포로 변환됨)
Feed-Forward Network(FFN)	각 토큰의 어텐션 결과는 비선형 변환을 통해 레이어 정규화, 즉 각 레이어의 출력이 일정한 분포를 가지도록 정규화하여 학습 안정성을 향상 시킴.
Residual Connections	각 주요 레이어(셀프 어텐션 레이어와 피드 포워드 네트워크)에서 Residual connection을 사용하여 입력을 그대로 출력에 더함. 이는 학습이 더 빠르고 안정적으로 이루어지도록 하며, vanishing gradient 문제를 완화함.
출력 레이어	디코더의 최종 출력은 다시 텍스트로 변환됨
생성 과정	모델은 주어진 입력을 기반으로 한 번에 한 토큰씩 생성함. 생성된 토큰은 다시 입력으로 사용되어 다음 토큰을 예측함, 이 과정은 특정 길이의 텍스트가 생성될 때까지 반복됨
예측	마지막으로, 디코더의 출력을 통해 각 토큰에 대한 확률 분포를 계산하고, 가장 높은 확률을 가진 토큰을 선택하여 시퀀스의 다음 토큰으로 예측함. 이 과정이 반복되어 문장이 완성됨

Model Training (1)

- Training
 - 데이터와 모델의 구조를 제공하고, 주어진 데이터를 가장 잘 설명할 수 있는 모델로 업데이트하는 과정임
 - 직접 모델에게 처리 방법을 알려주지 않음(Traditional Coding)
 - 입력/출력 데이터를 주고 모델이 스스로 학습하도록 함.
 - Fine-Tuning : Training의 한 종류로, 이미 train된 모델을 내 Task에 맞는 데이터로 추가 training.

Model Training (2)

- Training 과정



Model Training (3)

▪ Loss Function

- 모델이 예측한 출력값과 실제 값 간의 차이를 측정하는 함수임.
- 손실 함수는 모델의 성능을 평가하는데 중요한 역할을 하며, 모델 학습의 방향을 결정짓는 지표로 사용됨.
- 손실 함수의 역할
 - 모델의 오류 측정
 - 모델 학습의 방향 제공 : 손실함수를 최소화하기 위해, Gradient Descent과 같은 최적화 알고리즘이 사용됨.
 - 모델의 성능향상 : 손실함수를 최소화하는 것이 목표임. 손실 값이 작아질수록 모델의 성능이 개선된다고 봄.
- 손실 함수의 종류
 - 회귀 문제에서의 손실 함수 : MSE(Mean Squared Error), MAE(Mean Absolute Error)
 - 분류 문제에서의 손실 함수 : Cross-Entropy Loss, Binary Cross-Entropy)
- LLM에서는 Cross-Entropy Loss를 사용
 - 크로스 엔트로피는 진짜 확률 분포 $p(x)$ 와 예측된 확률분포(모델의 출력) $g(x)$ 간의 차이를 측정하는데 사용됨.

$$H(p, q) = - \sum_{x \in \text{classes}} p(x) \log g(x)$$

$p(x)$: True probability distribution (one-hot)

$q(x)$: Your model's predicted probability distribution

Model Training (3)

▪ Loss Function

- LLM에서는 왜 Cross-Entropy Loss를 사용
 - 크로스 엔트로피는 진짜 확률 분포 $p(x)$ 와 예측된 확률분포(모델의 출력) $g(x)$ 간의 차이를 측정하는데 사용됨.

$$H(p, q) = - \sum_{x \in \text{classes}} p(x) \log g(x)$$

$p(x)$: True probability distribution (one-hot)

$q(x)$: Your model's predicted probability distribution

- 로그 값이 음수이므로 이를 음수 부호로 보정하여 양수 값으로 변환합니다.

간단한 예제:

예를 들어, 두 개의 클래스(고양이, 개)가 있는 분류 문제에서:

실제 레이블(ground truth) $p(x)$:

$$p(\text{cat}) = 1, \quad p(\text{dog}) = 0$$

모델의 예측 확률 $g(x)$:

$$g(\text{cat}) = 0.9, \quad g(\text{dog}) = 0.1$$

크로스 엔트로피 손실은 다음과 같이 계산됩니다:

$$H(p, q) = -[1 \cdot \log(0.9) + 0 \cdot \log(0.1)] = -\log(0.9) \approx 0.105$$

Model Training (4)

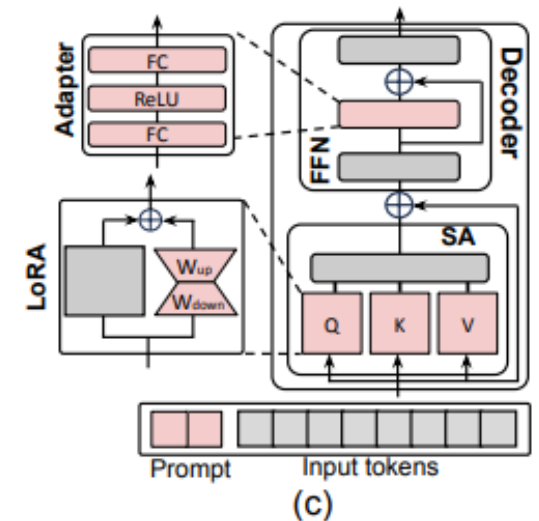
▪ Overfitting

- Training Loss : 모델이 학습 데이터셋에서 예측할 때 얼마나 틀렸는지를 나타내는 값으로 , 이 값은 모델이 학습 데이터에 맞게 가중치가 조정되는 정보를 보여줌.
- Validation Loss : 모델이 검증 데이터셋에서 예측을 수행할 때 발생하는 손실 값임. Validation Loss는 모델이 새로운 데이터에 대해 얼마나 잘 일반화할 수 있는지를 평가함.
- Overfitting signal : 훈련이 진행되면서 training loss는 계속 감소하지만, validation loss는 더 이상 감소하지 않고 오히려 증가한다면, 이는 과적합의 신호임.
- Overfitting 방지를 위한 해결방법들
 - Early stopping : validation loss가 증가하기 시작하면 학습을 멈춤
 - Hyperparameter tuning : 학습률, 레이어수 등을 조정
 - 충분한 데이터를 제공함

Model Training (5)

trl.SFTTrainer

- TRL(Transformer Reinforcement Learning)은 Transformer 라이브러리 위에 구축된 도구로 강화학습(RLHF, DPO) 및 SFT(Supervised Fine-tuning)과 같은 기능을 제공함.
- SFTTrainer는 Instruction-response 형식의 데이터셋에 최적화되어 있음.
- PEFT (parameter-efficient fine tuning) 지원
 - LoRA(Low-Rank Adaptation) : 새로운 저차원 행렬을 추가하여 모델을 파인튜닝하는 방법
 - Adapter : 각 레이어에 작은 네트워크를 추가하여 모델을 파인튜닝
 - Prefix Tuning : 입력 시퀀스 앞에 추가적인 토큰을 붙여 모델을 파인튜닝
- 토큰화 자동 처리 지원 : 토큰화의 과정 (tokenization, encoding, padding & truncation)을 쉽고 편하게 사용할 수 있도록 지원함.



Model Training (6)

trl.SFTTrainer의 주요 파라미터

- **model** : 파인튜닝할 사전 학습된 모델을 지정함.
- **args** : 학습을 지정함(learning rate, batch size, epochs 등)
- **train_dataset** : 학습에 사용할 데이터셋 지정
- **eval_dataset** : 검증에 사용할 데이터셋 지정

```
1 # Dataset Load
2 from datasets import load_dataset
3
4 dataset = load_dataset("yahma/alpaca-cleaned", split="train")
5
6 dataset = dataset.shuffle(seed=42)
7 no_input_dataset = dataset.filter(lambda example: example['input'] == '')
8 mapped_dataset = no_input_dataset.map(prompt_formatting_func, batched=True)
9 split_dataset = mapped_dataset.train_test_split(test_size=0.01, seed=42)
10
11 train_dataset = split_dataset['train']
12 test_dataset = split_dataset['test']
13
```

- **tokenizer** : 사용할 토큰라이저 지정.

```
1 # Tokenizer 설정
2 from transformers import AutoTokenizer
3
4 tokenizer = AutoTokenizer.from_pretrained("meta-llama/Meta-Llama-3-8B")
5 tokenizer.add_special_tokens({"pad_token": "<|reserved_special_token_250|>"})
```

```
2 from trl import SFTTrainer
```

```
27 trainer = SFTTrainer(
28     model = model,
29     tokenizer = tokenizer,
30     train_dataset = train_dataset,
31     eval_dataset = test_dataset,
32     peft_config=peft_config,
33     dataset_text_field = "text",
34     max_seq_length = 2048,
35     dataset_num_proc = 2,
36     packing = False,
37     args = training_arguments,
38     **data_collator_param
39 )
```

- line 5가 실행되면, 토큰라이저는 "pad_token"으로 "<|reserved_special_token_250|>"을 사용하게 됨
- 이후에 토큰라이징 과정에서 시퀀스를 특정길이로 패딩할 때 이 토큰이 사용됨.

▪ Transformers.TrainingArguments

- `per_device_train_batch_size=2` : 각 GPU별로 사용할 학습배치 크기를 설정함.
- `gradient_accumulation_steps=8` : 그래디언트 업데이트를 몇 스텝마다 수행할 지를 설정함. 8 step마다 한번의 그래디언트가 업데이트
- `warmup_steps =50` : 학습 초기에 학습률을 서서히 증가시키는 단계. 50 스텝동안 워밍업이 진행됨.
- `max_steps = 100` : 총 학습 스텝수를 제한함, 100 스텝이 지나면 종료.
- `eval_steps =10` : 학습과정에서 몇 스텝마다 평가를 수행할 지 설정. 10 스텝마다 평가가 이루어짐.
- `save_steps=50` : 50 스텝마다 체크포인트가 저장됨
- `evaluation_strategy = steps` : 평가를 언제 할지에 대한 전략 설정. steps로 설정되어 있어, 정해진 스텝마다 평가가 이루어짐.
- `save_strategy=steps` : 모델 체크포인트를 저장하는 전략을 설정
- line 7은 학습 중 발생하는 로그(손실 값, 학습률등)를 tensorboard에 기록하도록 지정하는 역할음.
- tensorboard는 google이 제공하는 시각화 도구임.

```
3 from transformers import TrainingArguments
4
5 training_arguments = TrainingArguments(
6     output_dir=local_output_dir,
7     report_to = "tensorboard",
8     per_device_train_batch_size = 2,
9     per_device_eval_batch_size = 2,
10    gradient_accumulation_steps = 8,
11    warmup_steps = 50,
12    max_steps = 100,
13    eval_steps=10,
14    save_steps=50,
15    evaluation_strategy="steps",
16    save_strategy="steps",
17    learning_rate = 1e-4,
18    logging_steps = 1,
19    optim = "adamw_8bit",
20    weight_decay = 0.01,
21    lr_scheduler_type = "constant_with_warmup",
22    seed = 42,
23    gradient_checkpointing = True,
24    gradient_checkpointing_kwargs={'use_reentrant':True}
25 )
```

▪ Transformers.TrainingArguments

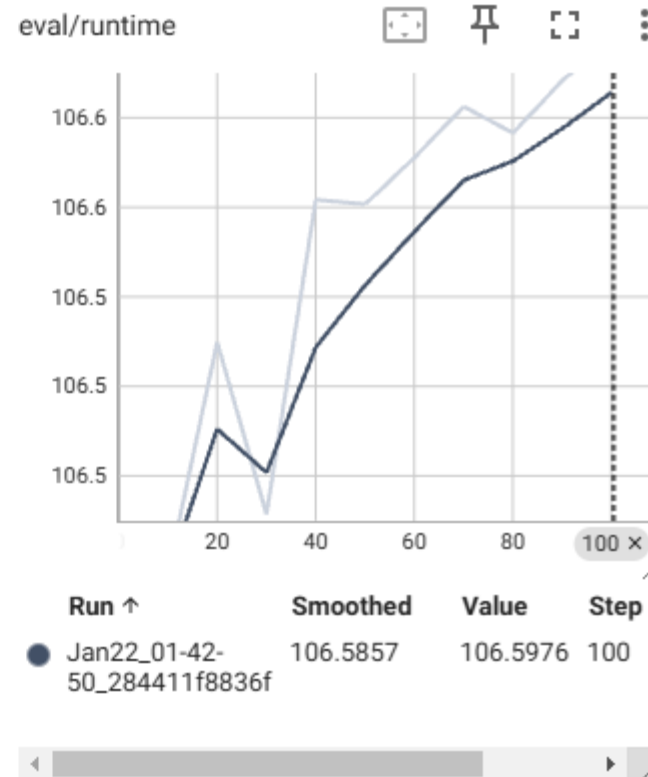
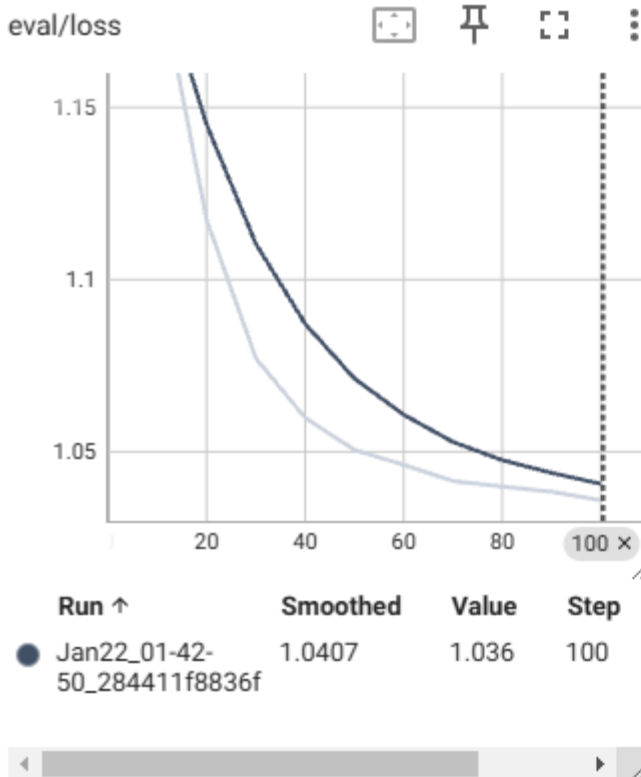
- `learning_rate` : 0.0001로 설정
- `logging_steps=1` : 몇 스텝마다 로그를 기록할지 설정
- `optim=adamw_8bit` : 사용할 옵티마이저를 설정, 여기서는 메모리 사용을 줄이기 위해 8bit 정밀도의 AdamW옵티마이저 사용함.
- `weight_decay=0.01` : 가중치 감쇠 값을 설정함. 이는 과적합을 방지하기 위해 모델 가중치에 대한 패널티를 적용하는 방식임.
- `lr_scheduler_type=constant_with_warmup` : 일정한 학습률을 사용하면서 학습 초기에 워밍업 단계에서 학습률을 점진적으로 증가.
- `gradient_checkpointing=true` : 이 기능을 사용하면 모델이 학습할 때 중요한 중간 데이터(순전파 과정에서 생성되는 활성화 값)를 모두 저장하지 않고, 역전파 과정에서만 다시 계산하도록 함, 이렇게 하면 메모리 사용량이 줄어들지만, 대신 다시 계산해야 하므로 학습 속도가 느려질 수 있음.
- `gradient_checkpointing_kwargs={use_reentrant : true}`. 재귀적인 함수 호출을 지원하도록 설정. 이 옵션은 특정 상황에서 그래디언트 체크포인트의 동작을 최적화하는데 사용됨

```
3 from transformers import TrainingArguments
4
5 training_arguments = TrainingArguments(
6     output_dir=local_output_dir,
7     report_to = "tensorboard",
8     per_device_train_batch_size = 2,
9     per_device_eval_batch_size = 2,
10    gradient_accumulation_steps = 8,
11    warmup_steps = 50,
12    max_steps = 100,
13    eval_steps=10,
14    save_steps=50,
15    evaluation_strategy="steps",
16    save_strategy="steps",
17    learning_rate = 1e-4,
18    logging_steps = 1,
19    optim = "adamw_8bit",
20    weight_decay = 0.01,
21    lr_scheduler_type = "constant_with_warmup",
22    seed = 42,
23    gradient_checkpointing = True,
24    gradient_checkpointing_kwargs={'use_reentrant':True}
25 )
```

Model Training (8)

TensorBoard에서 확인할 수 있는 정보:

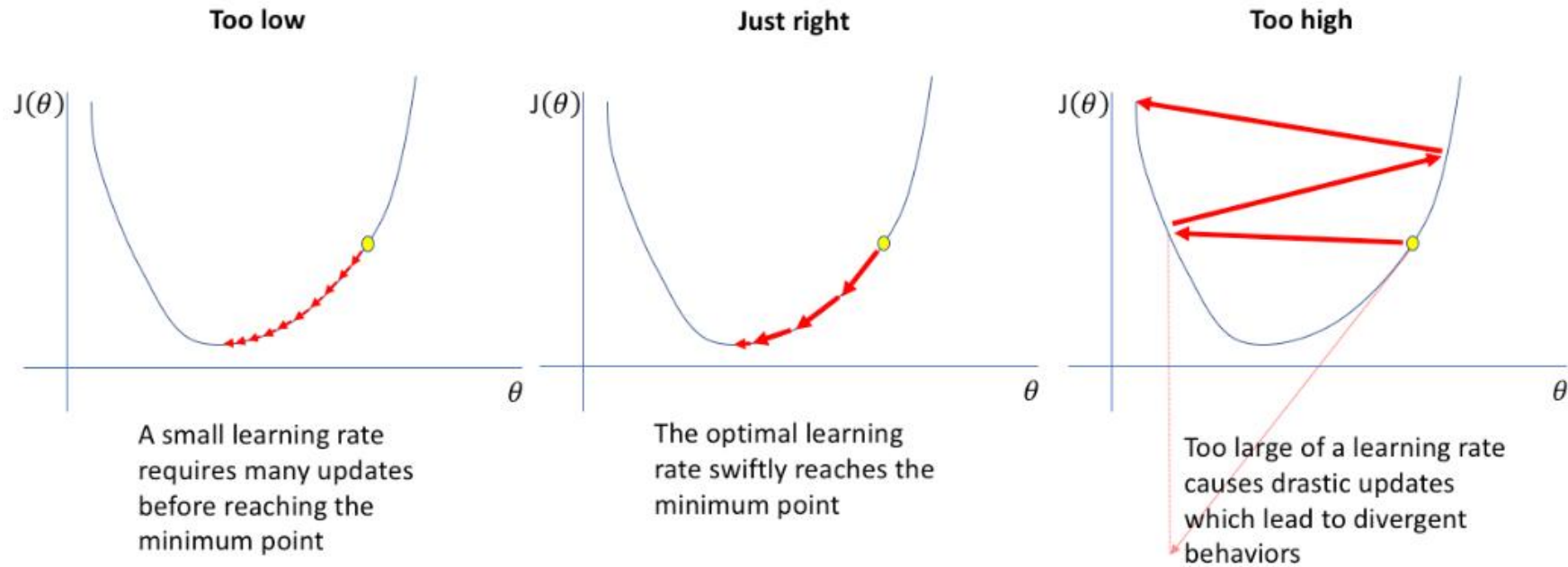
- 훈련 및 검증 손실 (Loss)
- 학습률 (Learning Rate)
- 가중치 변화 (Gradients)
- 평가 지표 (예: 정확도, 정밀도, 재현율 등)



Model Training (9)

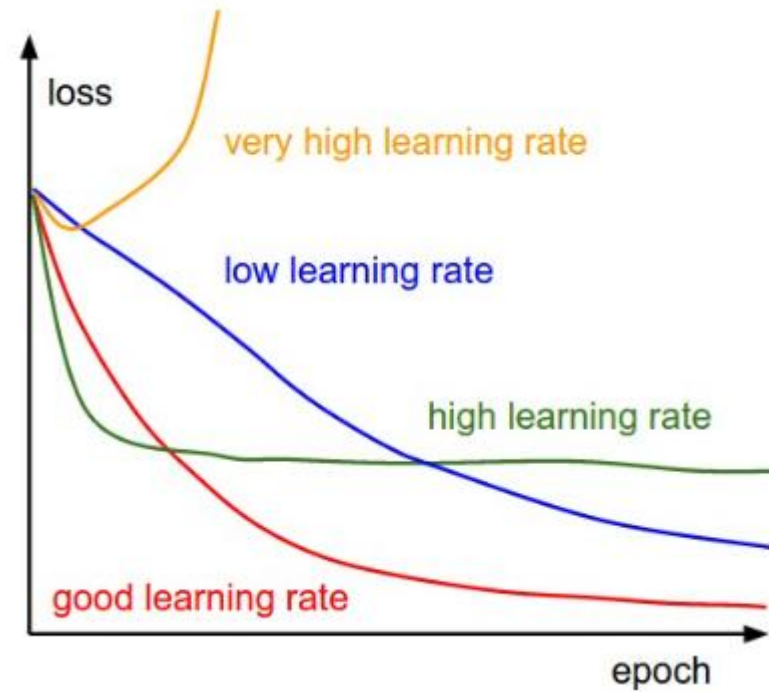
Parameter : Learning Rate

- 일반적으로 $1e-5 \sim 5e-4$ 정도에서 조정함



Model Training (10)

- Parameter : Learning Rate



Model Training (11)

▪ Parameter : Learning rate 관련

- lr_scheduler_type : 학습이 진행됨에 따라 learning_rate를 어떤식으로 변경할 것인가?
 - constant_with_warmup : warm_step동안 학습률이 0에서 시작해서 선형적으로 증가한 후, 워밍업 단계 후에는, 학습률을 일정하게 유지함.
 - linear_decay : 워밍업 후에는 학습이 끝날때까지 학습률이 선형적으로 감소하고 학습종료시점에는 학습률이 0
 - Cosine_decay : 워밍업 후에는 코사인 함수의 형태에 따라 감소함.
- warmup_steps : 학습 초기 단계에서 학습률을 서서히 증가시키는 스텝 수를 의미함
 - warmup_steps = 50 : 첫 50 스텝동안 학습률이 점진적으로 증가하여 이후 설정된 스케줄에 따라 학습률이 적용됨

Model Training (12)

▪ Parameter : Batch Size & Steps

- **Batch size** : 하나의 배치(batch)에 포함되는 샘플의 개수며, **한 번에 샘플 수 만큼 처리한다는 의미임.**
 - `per_device_train_batch_size` : 하나의 배치에 처리할 샘플이 개수
(GPU 메모리 한계치까지 가능한 크게 설정함. (2/4/8/16/32, `batch_size = 2` : 한 번에 2개의 샘플을 처리함))
 - `gradient_accumulation_steps` : 한 번에 몇 개의 배치를 함께 사용
 - **`batch_size=2 & gradient_accumulation_steps =8`** : 한 번의 배치는 2개의 샘플을 동시에 처리한 후 가중치를 계산만하고, 8번의 스텝이 지나면 그 동안 누적된 gradient를 사용하여 한 번 가중치를 업데이트 함.
- **Steps** : Step은 모델이 하나의 배치 데이터를 처리한 후, 그에 대한 가중치를 업데이트하는 과정을 의미함.
Epoch는 전체 데이터셋을 한 번 완전히 학습시키는 단위

$$\text{Total Steps} = \frac{\text{Dataset Size}}{\text{Batch Size}} \times \text{Epochs}$$

- `max_steps = 100` : 모델은 100개의 스텝 동안만 학습을 진행함. 즉 100번의 배치 업데이트 후 학습 종료함.

Model Training (13)

▪ Parameter : max_seq_length

- **context window** : 모델 아키텍처 자체에서 정의된 최대 문맥 크기를 의미함.
- 이 크기는 모델 architecture별로 다름.
- **max_seq_length**는 데이터 처리 및 모델 학습에서 입력 시퀀스의 최대 길이임, **사용자가 설정하는 하이퍼파라미터임**
- 만약, 입력 시퀀스가 max_seq_length를 초과하면, 초과된 부분을 자르고, 처음 max_seq_length 토큰만 고려됨
- 시퀀스가 max_seq_length보다 짧으면, 일반적으로 특수한 패딩 토큰으로 패딩되어 모든 입력 시퀀스의 길이가 동일하게 맞춰짐.

```
5 tokenizer.add_special_tokens({"pad_token": "<|reserved_special_token_250|>"})
```
- context window 즉 최대 문맥 크기는 모델이 한 번에 처리할 수 있는 이론적인 한계를 말하고
- max_seq_length는 실제 모델 입력에 적용되는 설정 가능한 길이임 (**GPU 메모리를 고려하여**)
- $\text{max_seq_length} \leq \text{context window}$ 로 설정해야 함.

Model Training (14)

- **Monitoring : Tensorboard**
 - Training이 진행됨에 따라 실시간으로 모니터링 가능

```
1 # tensorboard 설정
2 %load_ext tensorboard
3 %tensorboard --logdir '{local_output_dir}/runs'
```

```
5 training_arguments = TrainingArguments(
6     output_dir=local_output_dir,
7     report_to = "tensorboard",
8     per_device_train_batch_size = 2,
9     per_device_eval_batch_size = 2,
10    gradient_accumulation_steps = 8,
```

Tokenizer (1)

- Tokenizer :
 - NLP에서 텍스트 데이터를 모델이 이해할 수 있는 형식으로 변환하는 과정에서 사용됨.
 - 토큰나이저는 텍스트를 '토큰' 이라는 작은 단위로 분할하는 역할

토큰화(Tokenization)

텍스트를 개별 단어 또는 부분 단어로 분할함



인코딩(Encoding)

텍스트를 토큰으로 분할 한 후, 각 토큰을 고유한 숫자 ID로 변환함.
이 숫자 ID는 모델의 입력으로 사용되며, 모델이 텍스트를 처리할 수 있게 함



디코딩(Decoding)

모델이 생성한 숫자 ID 시퀀스를 다시 인간이 읽을 수 있는 텍스트로 변환하는 과정임.

Tokenizer (2)

- <https://huggingface.co/spaces/Xenova/the-tokenizer-playground> (실습해볼것)

The Tokenizer Playground

Experiment with different tokenizers (running locally in your browser).

Llama 3

LLM 중에서 가장 유명한 LLM은 뭐야?

TOKENS CHARACTERS

14

24

LLM 중에서 가장 유명한 LLM은 뭐야?

[4178, 44, 72043, 57575, 107120, 101003, 80732, 24486, 445, 11237, 34804, 113792, 90759, 5380]

Tokenizer (3)

▪ Special Token

- 모델에서 특정 역할을 수행하는 특별한 토큰들로, 일반적인 텍스트 토큰과 구별됨.
- 이러한 토큰은 모델이 문장의 시작과 끝, 패딩, 또는 특별한 상황을 처리하는데 사용됨.
- https://huggingface.co/meta-LLaMA/Meta-LLaMA-3-B/blob/main/tokenizer_config.json 
- https://huggingface.co/meta-LLaMA/Meta-LLaMA-3-8B/blob/main/special_tokens_map.json

```
{  
  "bos_token": "<|begin_of_text|>",  
  "eos_token": "<|end_of_text|>?"  
}
```

```
{  
  "added_tokens_decoder": {  
    "128000": {  
      "content": "<|begin_of_text|>",  
      "lstrip": false,  
      "normalized": false,  
      "rstrip": false,  
      "single_word": false,  
      "special": true  
    },  
    "128001": {  
      "content": "<|end_of_text|>",  
      "lstrip": false,  
      "normalized": false,  
      "rstrip": false,  
      "single_word": false,  
      "special": true  
    },  
    "128002": {  
      "content": "<|reserved_special_token_0|>",  
      "lstrip": false,  
      "normalized": false,  
      "rstrip": false,  
      "single_word": false,  
      "special": true  
    }  
  },  
}
```

Tokenizer (4)

- eos_token : end_of_sequence
 - 입력 시퀀스의 끝났음을 알려주는 토큰
 - LLaMA3 모델의 경우 : <|end_of_text|>

Llama 3



I love AI but it is complex to implement.<|end_of_text|>

[40, 3021, 15592, 719, 433, 374, 6485, 311, 4305, 13, 128001]

Tokenizer (5)

- `bos_token` : `beginning_of_sequence`
 - 입력 시퀀스의 시작을 알려주는 토큰
 - LLaMA3 모델의 경우 : `<|begin_of_text|>`
 - 다른 모델(Gemma)중에는 `bos_token`이 반드시 들어가야 fine-tuning이 잘 되는 경우도 있음

Llama 3

```
<|begin_of_text|>I love AI but it is complex to implement.  
<|end_of_text|>
```

```
[128000, 40, 3021, 15592, 719, 433, 374, 6485, 311, 4305, 13, 128001]
```

Tokenizer (6)

- pad_token :
 - 모델 Fine-Tuning시 모든 입력이 같은 길이를 가지도록 조정함.
 - 병렬 처리를 통한 효율적인 training을 하기 위함.
 - 같은 길이로 맞추는 방법
 - truncation : 잘라내기
 - padding : 채워넣기, 실제로 학습이 발생하지 않은 의미없는 토큰 채움.
 - pad_token의 Loss는 무시됨.
 - 확률 분포도 계산하지 않고, 배우지도 않음.

Tokenizer (7)

- Training시 pad_token을 따로 설정해야 함.
 - 아래와 같이 reserved_special_token을 활용하거나
 - 추가 토큰 <|pad|>를 설정하거나 해야 함.

```
1 # Tokenizer 설정
2 from transformers import AutoTokenizer
3
4 tokenizer = AutoTokenizer.from_pretrained("meta-llama/Meta-Llama-3-8B")
5 tokenizer.add_special_tokens({"pad_token": "<|reserved_special_token_250|>"})
6 model.config.pad_token_id = tokenizer.pad_token_id
```


Tokenizer (8)

- Training data에 eos_token 추가하기
 - 모델이 언제 답을 끝낼지 학습시켜야 함.

```
12 def prompt_formatting_func(examples):
13     instructions = examples["instruction"]
14     outputs      = examples["output"]
15     texts = []
16     for instruction, output in zip(instructions, outputs):
17         alpaca_formatted_str = convert_to_alpaca_format(instruction, output) + EOS_TOKEN
18         texts.append(alpaca_formatted_str)
19     return { "text" : texts, }
```

Quantization & LoRA (1)

- 모델 경량화 : Quantization & LoRA
- Quantization : 효율성 Up, 정밀도 Down
 - Connection weight 등 각 숫자를 어느 정도의 정밀도로 저장할 것인가?
 - Quantization
 - 정밀도를 낮추고(32bit→8bit)
 - 효율성을 up (적은 메모리, 빠른 추론)
 - Quantization 종류
 - Dynamic Quantization : 모델 추론 실행시 동적으로 양자화
 - Static Quantization : 모델 저장/로드시 양자화
 - Quantization-Aware Training(QAT) : 훈련 중에 양자화의 효과를 시뮬레이션함.

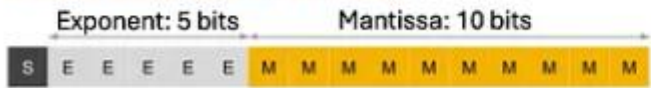
- QAT는 훈련 중 부동소수점(FP32)을 유지하면서도 , 정수 (INT8) 연산의 영향을 흉내 내는 연산을 함. 모델의 가중치와 활성화값을 INT8 범위(예: -128 ~ 127)로 매핑하는 과정을 거치면서, 훈련 중 정확도를 보정할 수 있도록 함

Quantization & LoRA (2)

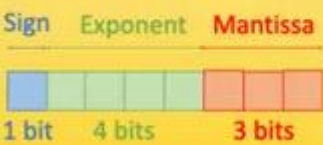
float32: IEEE single-precision



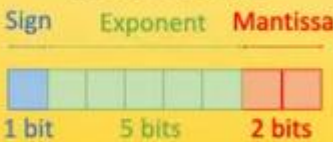
float16: IEEE half-precision



Float 8 E4M3



Float 8 E5M2



Quantization & LoRA (3)

▪ BitsAndBytes

- HF Transformer 모델을 활용할 때 사용할 수 있는 Quantization Library.
- 이 라이브러리는 모델의 메모리 사용량을 줄이고, 훈련과 추론을 가속화하기 위한 방법을 제공함.
- 저정밀도 계산(예: 8-bit, 4-bit 연산)을 통해 모델의 효율성을 극대화하는데 중점을 둠.

```
1 # 모델 경량화: Quantization 설정
2 from transformers import BitsAndBytesConfig
3 import torch
4
5 quantization_config=BitsAndBytesConfig(
6     load_in_4bit=True,
7     bnb_4bit_compute_dtype=torch.bfloat16,
8     bnb_4bit_use_double_quant=True,
9     bnb_4bit_quant_type='nf4'
10 )
```

```
1 # 기본 Llama 3 모델 로드
2 from transformers import AutoModelForCausalLM
3 model = AutoModelForCausalLM.from_pretrained(
4     "meta-llama/Meta-Llama-3-8B",
5     quantization_config = quantization_config,
6     device_map = {"": 0}
7 )
```

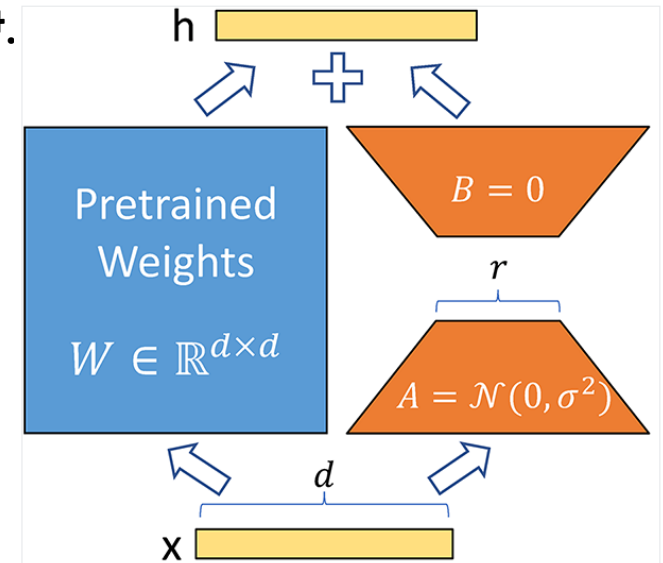
- **nf4(normalized float 4-bit)** 양자화에서는 FP32를 미리 정의된 16개(2^4)의 값 세트로 매핑함.

- **bfloat16**은 양자화된 4bit 값으로 계산을 수행할 때, 연산의 중간 결과나 가중치 업데이트를 위한 데이터 타입으로 사용됨.
- bfloat16의 지수 비트는 FP32(8비트)와 동일하지만, 가수 비트가 7 비트만 사용됨.(FP32 대비 절반의 메모리 사용량을 가지므로, 더 큰 모델을 로드할 수 있음.)

Quantization & LoRA (4)

▪ LoRA : Low-Rank Adaptation

- LoRA는 모델의 가중치 행렬을 저차원(low-rank) 공간에서 조정하는 방식으로, 기존의 파라미터를 크게 변경하지 않으면서, 저차원 공간에서 작은 보정을 추가하는 방식으로 작동함.
- LoRA에서는 기존 모델의 파라미터(가중치 행렬)를 직접 업데이트하는 대신, 저차원 공간에서 학습된 adapter 행렬을 추가함. 이는 모델의 원래 가중치에 더해져 모델이 새로운 작업에 적응할 수 있게 함.
- 실제 weights를 training 하는 대신, 옆에 붙인 Adapter(BA)들을 training 함.
- $w = w_0 + BA$, 여기서 w_0 는 사전 학습된 가중치이고, BA는 더 낮은 랭크로 분해된 행렬, w_0 는 고정시켜 놓고 BA만 학습함
- 메모리 효율성 : 적은 수의 파라미터만 저장
- 계산 효율성 : 작은 행렬 곱으로 대체
- 각종 태스크별로 Adapter를 따로 만들어서 활용하는 것도 가능함.



Quantization & LoRA (5)

- peft : Parameter-Efficient Fine-Tuning) Libaray
 - LoRA를 비롯한 다양한 효율적인 Fine-tuning 방법(Adapter, Prefix-Tuning, P-Tuning) 을 제공함
 - peft는 대규모 모델의 전체 파라미터를 조정하지 않고, 특정 작업에 대해 모델을 효과적으로 조정할 수 있는 다양한 기법을 제공
 - Base Model + LoRA Config → LoRA Model

```
1 # 모델 경량화: Lora 설정
2 from peft import LoraConfig
3 peft_config = LoraConfig(
4     lora_alpha=16,
5     lora_dropout=0,
6     r=16,
7     bias="none",
8     task_type="CAUSAL_LM",
9     target_modules=["q_proj", "v_proj", "k_proj", "o_proj", "gate_proj", "up_proj", "down_proj"]
10 )
```

Self-Attention관련 FFN 관련

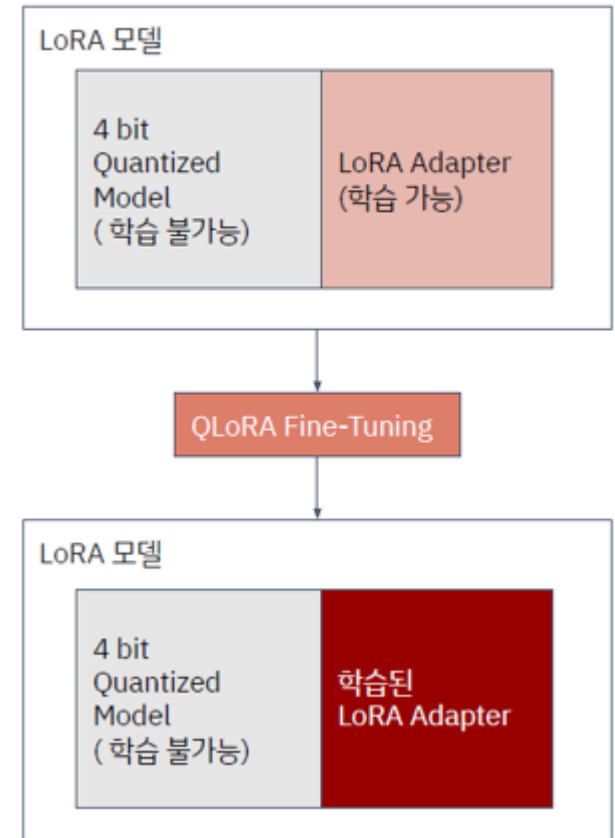
```
1 from peft import get_peft_model
2 lora_model = get_peft_model(model, peft_config)
```

- **lora_alpha=16**은, 저차원으로 학습된 가중치를 원래 차원으로 확장할 때 스케일링 함. (이는 학습된 LoRA 가중치를 원래 가중치에 추가할 때, 학습된 변화를 조절하는 역할)
- **r=16**은 기존 모델의 고차원 가중치를 r차원(16차원)의 저차원 행렬로 축소함. (학습속도 높이고, 메모리 사용량 감소)

Quantization & LoRA (6)

▪ QLoRA : Quantization + LoRA

- Quantization과 LoRA의 장점을 한번에
 - 4bit Quantization 모델 혼자만으로는 가중치를 업데이트 할때 정밀도가 크기 손실되어 학습하지 못할 수도 있음
 - 그러나 여기에 LoRA Adapter를 붙이면 학습이 가능해짐.
(왜냐하면 LoRA를 사용하면 4Bit로 양자화된 가중치는 고정된 상태로 두고, 추가된 저차원 LoRA adapter에서 학습이 이루어지게 됨)
 - 적은 메모리, 빠른 연산이 가능하며, 괜찮은 성능
 - 다양한 파라미터값에 대한 실험을 빠르게 시도해볼 수 있음
- QLoRA로 Fine-Tuning 해보고 성능이 충분히 나오지 않을 경우 Full Fine-Tuning시도



Data Loading & Processing (1)

▪ HF datasets 라이브러리

- 머신러닝을 위한 데이터셋 관리 라이브러리로
- 다양한 공개 데이터셋에 쉽게 접근 및 사용 가능
- 데이터셋 로딩 : `load_dataset()` 함수
- 다양한 포맷지원(CSV, JSON, Parquet 등)
- 직접 만든 데이터셋도 로드해서 사용가능

```
1 # Dataset Load
2 from datasets import load_dataset
3
4 dataset = load_dataset("yahma/alpaca-cleaned", split="train")
5
6 dataset = dataset.shuffle(seed=42)
7 no_input_dataset = dataset.filter(lambda example: example['input'] == '')
8 mapped_dataset = no_input_dataset.map(prompt_formatting_func, batched=True)
9 split_dataset = mapped_dataset.train_test_split(test_size=0.01, seed=42)
10
11 train_dataset = split_dataset['train']
12 test_dataset = split_dataset['test']
13
14 print(f"Train dataset size: {len(train_dataset)}")
15 print(f"Test dataset size: {len(test_dataset)}")
```

- **Parquet 파일**은 데이터를 컬럼 단위로 저장함. 즉, 테이블의 각 열이 연속된 데이터 블록에 저장되어 특정 컬럼에 대한 접근이 매우 빠름. 이런 컬럼 형식의 저장 포맷은 주로 빅데이터 처리 시스템에서 사용되며, Apache Hadoop, Apache Spark와 같은 빅데이터 프레임워크와의 호환성이 뛰어나.

Data Loading & Processing (2)

▪ Alpaca-Cleaned 데이터셋

- <https://huggingface.co/datasets/yahma/alpaca-cleaned>
- <https://github.com/gururise/AlpacaDataCleaned> (Alpaca에 대해 설명이 잘되어 있음)
- Stanford univ에서 2023년에 발표된 것으로 Alpaca Dataset의 개선 버전임.
- chatGPT와 유사한 모델을 만드는데 사용된 52,000개의 instruction-response 쌍을 포함하고 있음.
- 각 데이터 항목은 다음과 같은 세가지 필드로 구성되어 있음
 1. instruction : 모델이 수행해야 할 작업을 설명하는 지시문임. 각 지시문은 고유함
 2. input : 항상 빈 문자열로 ("")로 설정되어 있으며, 데이터셋에는 실제 input 데이터가 없음
 3. output : Instruction을 기반으로 한 적절한 응답임.

```
{  
  "instruction": "Describe the structure of a neuron.",  
  "input": "",  
  "output": "A neuron consists of a cell body, dendrites, and an axon..."  
}
```

Data Loading & Processing (3)

▪ Instruction File-Tuning

- NLP에서 사전 학습된 언어 모델(GPT, Llama)을 특정 작업을 수행하도록 파인튜닝하는 과정임.
- 이 과정은 명령(instruction)-응답(response) 쌍으로 구성된 데이터셋을 사용하여 모델을 추가 학습함.

What is the capital of KOREA?



The capital of KOREA is Seoul.

- 이 방법은 모델이 명확한 명령어를 받았을 때 원하는 방식으로 응답을 생성하도록 조정하는 것을 목표로 함.

Data Loading & Processing (4)

▪ Instruction Formatting

- 정확하게 Fine-tuning된 모델이 다루게 될 입/출력의 형태로 데이터를 변환

```
1 # Prompt/Response Format 관련 설정
2 EOS_TOKEN = tokenizer.eos_token
3
4 def convert_to_alpaca_format(instruction, response):
5     alpaca_format_str = f"""Below is an instruction that describes a task. Write a response that appropriately completes the request.
6     \n\n### Instruction: \n{instruction}\n\n### Response: \n{response}\n
7     """
8
9     return alpaca_format_str
```

Below is an instruction that describes a task.
Write a response that appropriately completes the request.

Instruction
LLM이 무엇인가요

###Response
LLM은 AI 모델 중 하나입니다.

```

12 def prompt_formatting_func(examples):
13     instructions = examples["instruction"]
14     outputs      = examples["output"]
15     texts = []
16     for instruction, output in zip(instructions, outputs):
17         alpaca_formatted_str = convert_to_alpaca_format(instruction, output) + EOS_TOKEN
18         texts.append(alpaca_formatted_str)
19     return { "text" : texts, }

```

- dataset을 받아서 alpaca_format_str으로 만든 다음 texts 리스트를 만들
- 이렇게 하는 이유는 instruction, output형태로 명확하게 만들기 위해서임.(Instruction Tuning)

```

1 # Dataset Load
2 from datasets import load_dataset
3
4 dataset = load_dataset("yahma/alpaca-cleaned", split="train")
5
6 dataset = dataset.shuffle(seed=42)
7 no_input_dataset = dataset.filter(lambda example: example['input'] == '')
8 mapped_dataset = no_input_dataset.map(prompt_formatting_func, batched=True)
9 split_dataset = mapped_dataset.train_test_split(test_size=0.01, seed=42)

```

- line 7은 input값이 빈 문자열(' ')인 데이터만 남기고, input에 값이 들어 있는 데이터는 제거함.
- line 8은 input이 비어 있는 데이터셋(no_input_dataset)을 Alpaca형식의 문자열(alpaca_format_string)로 변환한 texts 리스트를 받아서 mapped_dataset에 저장됨

Data Loading & Processing (5)

▪ Data Collator

- 이것은 배치 생성, 패딩, 마스크 생성 등의 작업을 통해 데이터를 전처리하고, 모델에 입력할 수 있는 형태로 준비하는 역할을 함.
- **배치 처리를 위한 데이터 준비 클래스임.**
- 다양한 길이의 입력을 단일 배치로 변환, 패딩 적용등을 수행

▪ DataCollatorForCompletionOnlyLM :

- 질문 /답 형태의 데이터에서 '답' 부분에 대해서만 모델이 배우도록 하는 Collator임

```
1 # Data Collator 설정
2 from trl import DataCollatorForCompletionOnlyLM
3 data_collator_param = {}
4 response_template = "### Response:\n"
5 collator = DataCollatorForCompletionOnlyLM(response_template=response_template, tokenizer=tokenizer, mlm=False)
6 data_collator_param["data_collator"] = collator
```

- line 5는 collator 인스턴스를 만들어서 collator에 저장한 후
- line 6에서 딕셔너리로 key는 data_collator, value는 collator 인스턴스를 저장함

- **mlm(Masked Language Modeling)**은 입력 텍스트의 일부 토큰을 마스킹한 후, 모델이 이 마스킹된 부분을 예측하도록 훈련함(BERT같은 모델 훈련할 때 사용).
- **mlm=False로 설정하면 mlm 방식이 아닌 일반적인 언어 모델링(시퀀스 예측) 방식으로 데이터를 처리함.**

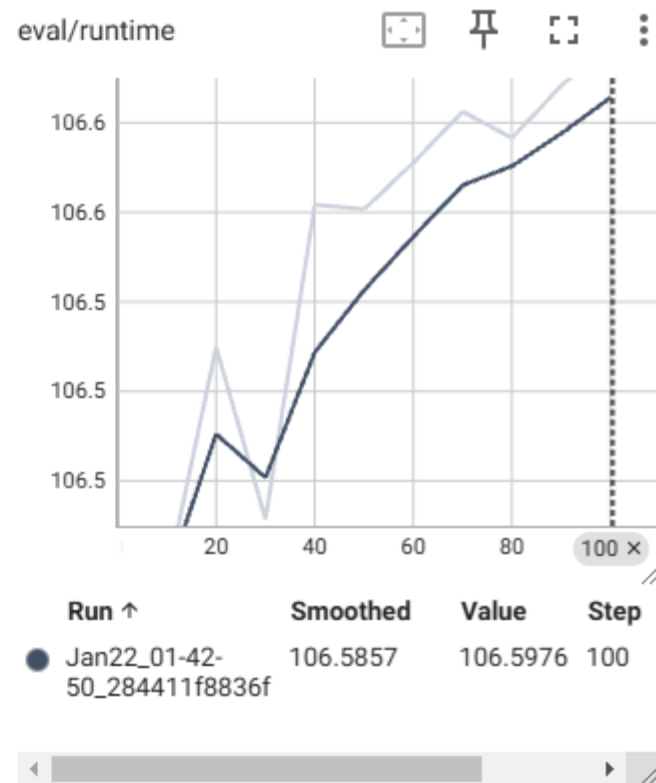
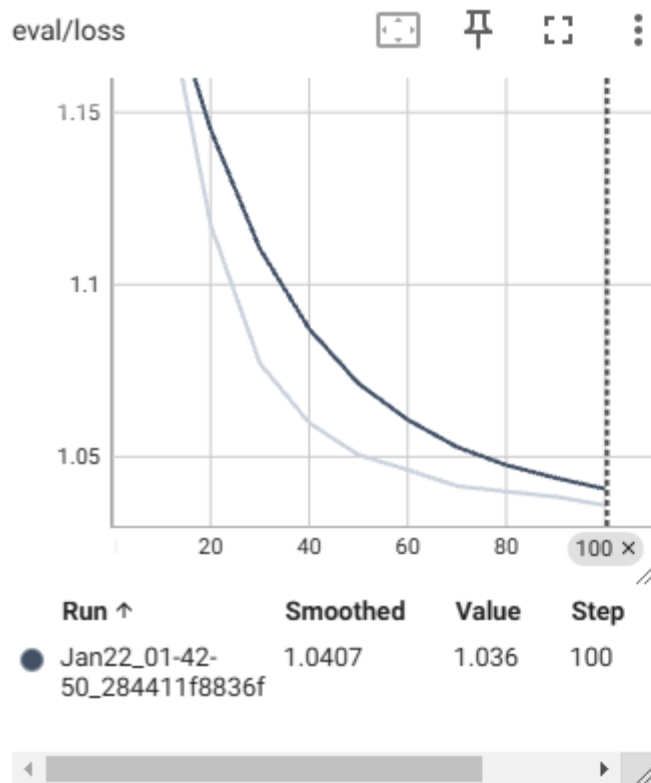
Fine-Tuning 실행 (1)

- Fine-Tuning 실행

- 1. LLaMA3-Fine-Tuning
<https://drive.google.com/drive/folders/1hThY15IJbI3E6t0E4L5d5dQVDDKB8FrK> (mydrive, public)
 - 파일 > 드라이브에 사본 저장
 - Connect to T4 (A100)
 - 실행
- 학습 실행시간 : 약 3시간 30분 (약 45분)
- 결과물은 local disk의 /content/fine_tune_outpu에 저장됨
- 실행 후 Google Drive의 fine_tune_output으로 복사

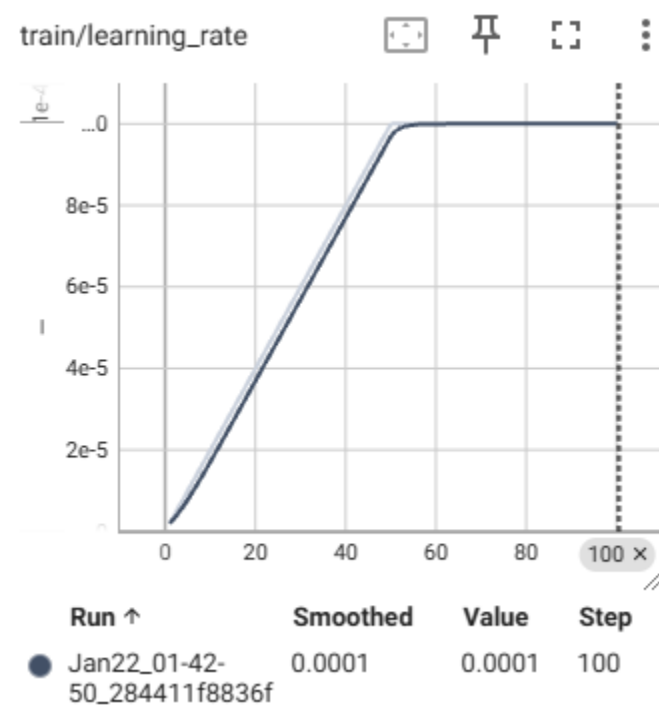
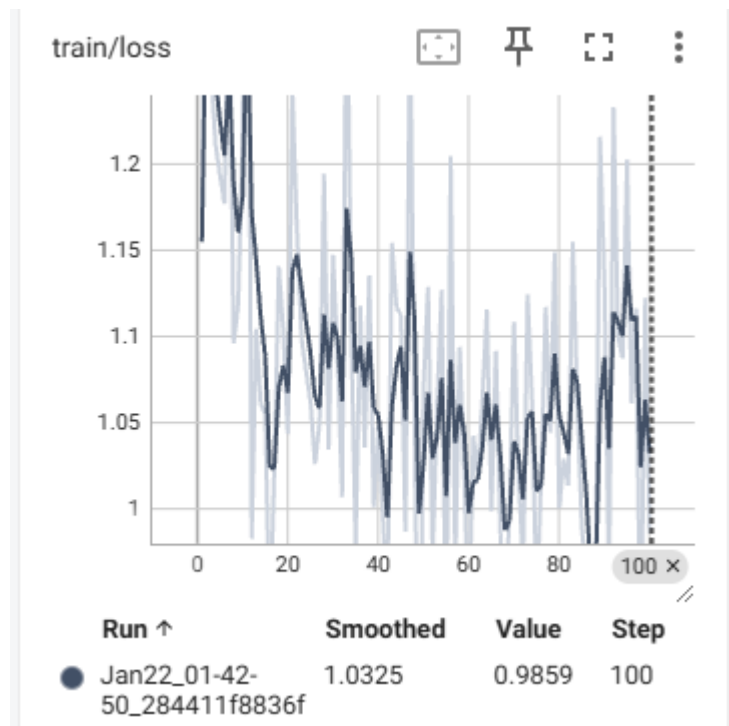
Fine-Tuning 실행 (2)

- Fine-Tuning 실행
 - Tensorboard 실행 결과



Fine-Tuning 실행 (3)

- Fine-Tuning 실행
 - Tensorboard 실행 결과



Fine-Tuning 실행 (4)

- Google drive : Fine_tune_output

내 드라이브 > fine_tune_output > checkpoint-50 ▾					✓ ≡	☰	ⓘ
유형 ▾ 사람 ▾ 수정 날짜 ▾							
이름 ▾		소유자	마지막으로 수정한 날짜 ▾	파일 크기			
≡ training_args.bin		나	오후 4:48 나	5KB			
📄 trainer_state.json		나	오후 4:48 나	9KB			
📄 tokenizer.json		나	오후 4:48 나	8.7MB			
📄 tokenizer_config.json		나	오후 4:48 나	49KB			
📄 special_tokens_map.json		나	오후 4:48 나	464바이트			
≡ scheduler.pt		나	오후 4:48 나	1KB			
≡ rng_state.pth		나	오후 4:48 나	14KB			
📄 README.md		나	오후 4:48 나	5KB			
≡ optimizer.pt		나	오후 4:48 나	80.7MB			
📄 adapter_model.safetensors		나	오후 4:48 나	160.1MB			
📄 adapter_config.json		나	오후 4:48 나	726바이트			

LLaMA3-8B Fine-Tuning 코드 실습(1)

```
[ ] 1 # 필요 Library install
    2 !pip install transformers==4.39.2 peft==0.10.0 trl==0.8.6 bitsandbytes==0.43.0 accelerate==0.29.0
```

⇌ 숨겨진 출력 표시

```
[ ] 1 # HF token 설정
    2 from huggingface_hub import login
    3 login(token="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")
```

```
[ ] 1 # 모델 경량화: Quantization 설정
    2 from transformers import BitsAndBytesConfig
    3 import torch
    4
    5 quantization_config=BitsAndBytesConfig(
    6     load_in_4bit=True,
    7     bnb_4bit_compute_dtype=torch.bfloat16,
    8     bnb_4bit_use_double_quant=True,
    9     bnb_4bit_quant_type='nf4'
   10 )
```

LLaMA3-8B Fine-Tuning 코드 실습 (2)

```
1 # 기본 Llama 3 모델 로드
2 from transformers import AutoModelForCausalLM
3 model = AutoModelForCausalLM.from_pretrained(
4     "meta-llama/Meta-Llama-3-8B",
5     quantization_config = quantization_config,
6     device_map = {"": 0}
7 )
```

```
1 # Tokenizer 설정
2 from transformers import AutoTokenizer
3
4 tokenizer = AutoTokenizer.from_pretrained("meta-llama/Meta-Llama-3-8B")
5 tokenizer.add_special_tokens({"pad_token": "<|reserved_special_token_250|>"})
6 model.config.pad_token_id = tokenizer.pad_token_id
7
```

LLaMA3-8B Fine-Tuning 코드 실습 (3)

```
[ ] 1 # Prompt/Response Format 관련 설정
    2 EOS_TOKEN = tokenizer.eos_token
    3
    4 def convert_to_alpaca_format(instruction, response):
    5     alpaca_format_str = f"""Below is an instruction that describes a task. Write a response that appropriately completes the request.
    6     \n\n### Instruction:\n{instruction}\n\n### Response:\n{response}
    7     """
    8
    9     return alpaca_format_str
   10
   11
   12 def prompt_formatting_func(examples):
   13     instructions = examples["instruction"]
   14     outputs      = examples["output"]
   15     texts = []
   16     for instruction, output in zip(instructions, outputs):
   17         alpaca_formatted_str = convert_to_alpaca_format(instruction, output) + EOS_TOKEN
   18         texts.append(alpaca_formatted_str)
   19     return { "text" : texts, }
```

LLaMA3-8B Fine-Tuning 코드 실습 (4)

```
[ ] 1 # Dataset Load
    2 from datasets import load_dataset
    3
    4 dataset = load_dataset("yahma/alpaca-cleaned", split="train")
    5
    6 dataset = dataset.shuffle(seed=42)
    7 no_input_dataset = dataset.filter(lambda example: example['input'] == '')
    8 mapped_dataset = no_input_dataset.map(prompt_formatting_func, batched=True)
    9 split_dataset = mapped_dataset.train_test_split(test_size=0.01, seed=42)
   10
   11 train_dataset = split_dataset['train']
   12 test_dataset = split_dataset['test']
   13
   14 print(f"Train dataset size: {len(train_dataset)}")
   15 print(f"Test dataset size: {len(test_dataset)}")
```

```
[ ] 1 # Data Collator 설정
    2 from trl import DataCollatorForCompletionOnlyLM
    3 data_collator_param = {}
    4 response_template = "### Response:\n"
    5 collator = DataCollatorForCompletionOnlyLM(response_template=response_template, tokenizer=tokenizer, mlm=False)
    6 data_collator_param["data_collator"] = collator
```

```
[ ] 1 # local output dir 설정
    2 local_output_dir = "/content/fine_tune_output"
```

```
[ ] 1 !mkdir {local_output_dir}
```

LLaMA3-8B Fine-Tuning 코드 실습 (5)

```
[ ] 1 # Training setup
    2 from trl import SFTTrainer
    3 from transformers import TrainingArguments
    4
    5 training_arguments = TrainingArguments(
    6     output_dir=local_output_dir,
    7     report_to = "tensorboard",
    8     per_device_train_batch_size = 2,
    9     per_device_eval_batch_size = 2,
   10     gradient_accumulation_steps = 8,
   11     warmup_steps = 50,
   12     max_steps = 100,
   13     eval_steps=10,
   14     save_steps=50,
   15     evaluation_strategy="steps",
   16     save_strategy="steps",
   17     learning_rate = 1e-4,
   18     logging_steps = 1,
   19     optim = "adamw_8bit",
   20     weight_decay = 0.01,
   21     lr_scheduler_type = "constant_with_warmup",
   22     seed = 42,
   23     gradient_checkpointing = True,
   24     gradient_checkpointing_kwargs={'use_reentrant':True}
   25 )
```

LLaMA3-8B Fine-Tuning 코드 실습 (6)

```
--
27 trainer = SFTTrainer(
28     model = model,
29     tokenizer = tokenizer,
30     train_dataset = train_dataset,
31     eval_dataset = test_dataset,
32     peft_config=peft_config,
33     dataset_text_field = "text",
34     max_seq_length = 2048,
35     dataset_num_proc = 2,
36     packing = False,
37     args = training_arguments,
38     **data_collator_param
39 )
```

```
[ ] 1 train_stats = trainer.train()
```

```
[ ] 1 # Google drive로 복사
2 from google.colab import drive
3 drive.mount('/content/drive')
```

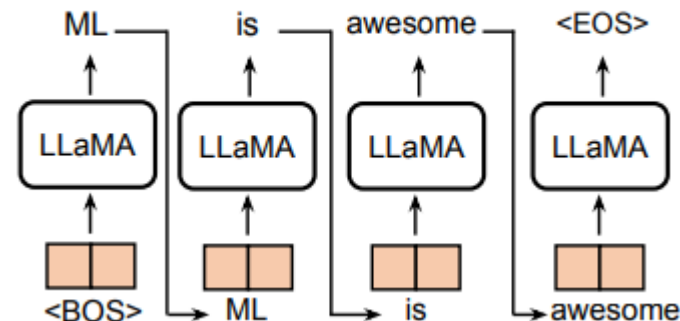
```
[ ] 1 !cp -r {local_output_dir} /content/drive/MyDrive
```

```
[ ] 1 # tensorboard 설정
2 %load_ext tensorboard
3 %tensorboard --logdir '{local_output_dir}/runs'
```

Inference (1)

▪ Inference

- 만들어진 모델을 이용해서 실제로 다음에 올 단어를 추측하는 과정임.
 - 모델의 출력값 : 다음에 올 토큰의 확률들
 - 확률을 바탕으로 다음에 올 토큰을 결정함.
 - 원하는 길이에 도달할 때까지 반복함.



- `model.generate()` : 입력 prompt에 대해 다음에 오게 될 토큰들을 순차적으로 생성함
- `tokenizer.decode()` : 생성된 토큰 id들을 자연어로 변환함.
- 실제 사용시에는 더 효율적인 library 활용이 필요 : vLLM(Virtual LLM)등

(Llama3-8B-Fine-Tuning(Model evaluation))

```
1 def test_model(instruction_str, model):
2     inputs = tokenizer(
3         [
4             convert_to_alpaca_format(instruction_str, ""),
5         ], return_tensors = "pt").to("cuda")
6     outputs = model.generate(**inputs, max_new_tokens = 128, use_cache = True, temperature = 0.05, top_p = 0.95)
7     return tokenizer.batch_decode(outputs)[0]
```


Inference (2)

▪ Inference : Parameters

- Temperature (예: 0.05 ~ 0.5)
 - 모델 출력의 주작위성 결정. 낮으면 더 안전하고 예측 가능한 답, 높으면 더 다양하고 창의적인 답
 - 실제 서비스 구현시에는 낮은값을 추천
- top_k: 상위 k개의 단어만 고려
- top_p : 모델이 각 후보 토큰에 할당한 확률을 높은 순서대로 더해나가며 누적확률이 top_p 이상이 되면 추가하지 않고, 그 안에서 무작위로 다음 토큰을 선택함
- max_new_tokens : 새로 추가되는 토큰의 개수
- stop_words : 이 단어를 보면 생성을 멈춰라

(Llama3-8B-Fine-Tuning(Model evaluation))

```
1 def test_model(instruction_str, model):
2     inputs = tokenizer(
3         [
4             convert_to_alpaca_format(instruction_str, ""),
5         ], return_tensors = "pt").to("cuda")
6     outputs = model.generate(**inputs, max_new_tokens = 128, use_cache = True, temperature = 0.05, top_p = 0.95)
7     return tokenizer.batch_decode(outputs)[0])
```

Practice (1)

- Fine-Tuning : 언제 시작하고 어떻게 접근할 것인가
 - Fine-Tuning을 해야 하나?
 1. LLM으로 풀어야 하는 문제인가? ➡ Yes
 2. 큰 유료 모델을 쓰지 말고 오픈소스 작은 모델을 사용할 이유가 있는가? ➡ Yes
 3. Prompt Engineering을 충분히 해봤는데 잘 안되는가? ➡ Yes
 4. 내가 LLM으로 부터 답변을 얻고자 하는 문제의 형태가 명확한가? ➡ Yes
 5. 데이터를 충분히 구할 수 있는가? ➡ Yes
 - ➡ 위의 상황이면 Fine-Tuning을 해볼 만함.
- Fine-Tuning 전략 : 가볍게 시작해서 여러 번 반복
 - 가장 Cost가 적게 들고 빠른 환경을 구성한다
 - 적당한 GPU(A100 1개 등) + 작은 모델(Llama3 8B) + QLoRA
 - 괜찮은 성능이 나오는 조합(Parameter/Data)를 찾을 때까지 여러 시도를 빠르게 반복함

Practice (2)

- Fine-Tuning 성능 평가 : 명확한 기준 설정하기
 - Evaluation : '잘 된다 ' 가 무엇인지 정확한 정의가 필요함
 - Benchmark를 만들 수 있다면 최고
 - 정확한 정답이 있는 Benchmark를 만들 수 있다면 좋음
 - 객관식 문제, 단답형 문제등
 - Blind Testing
 - 두 답중 뭐가 더 좋은가?
 - LLM as a judge도
 - 보통 모델들이 Generation 능력보다 Evaluation 능력이 더 좋다.
 - 특정 Task에 대해서 가장 성능 좋은 거대 모델과 비스/ 또는 그 이상을 추구하는 것이 현실적임.

Practice (3)

- Fine-Tuning 성능 향상을 위한 핵심 체크포인트
 - 점검 사항 : 잘 안된다면 일단 확인해 볼것들
 - Data
 - Fine-tuning 한정, 질이 양보다 중요하다.
 - 데이터는 잘 구조화 되어 있는가?
 - 데이터가 부족할 경우 : Synthetized Data를 활용해보만 함.
 - Train/Eval Loss 확인
 - Train Loss와 Eval Loss가 다 잘 내려 가는가?
 - 적당한 시점에서 끊었는가?
 - Tokenizer 확인
 - PAD token, EOS token등을 제대로 설정했는가?
 - Parameter들 조정
 - 가장 효과 좋은 Parameter들 : Learning Rate / Lora R, Alpha/Batch Size

Practice (4)

- Fine-Tuning 다음 단계
 - 잘 되는 조합을 찾고 난 후
 - Sweep the learning rate : 다른 변수는 유지한채로 Learning Rate만 바꿔가면서 다시 시도
 - 같은 조합으로 더 큰 Base 모델 + Full Fine-Tuning을 고려해보자
 - 새로운 기술과 모델이 계속해서 쏟아져 나오는 분야임
 - 오픈 소스/오픈 모델 커뮤니티를 주시

LLaMA3-8B Fine-Tuning Model evaluation 코드 실습 (1)

```
[1] 1 # 필요 Library install  
2 !pip install transformers==4.39.2 peft==0.10.0 trl==0.8.6 bitsandbytes==0.43.0 accelerate==0.29.0
```

⇨ 숨겨진 출력 표시

```
[2] 1 # HF token 설정  
2 from huggingface_hub import login  
3 login(token="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")
```

```
[3] 1 # Google Drive Import  
2 from google.colab import drive  
3 drive.mount('/content/drive')
```

⇨ Mounted at /content/drive

```
[4] 1 fine_tuned_model_path = "/content/drive/MyDrive/fine_tune_output/checkpoint-50"
```

```
[5] 1 # 모델 경량화: Quantization 설정  
2 from transformers import BitsAndBytesConfig  
3 import torch  
4  
5 quantization_config=BitsAndBytesConfig(  
6     load_in_4bit=True,  
7     bnb_4bit_compute_dtype=torch.bfloat16,  
8     bnb_4bit_use_double_quant=True,  
9     bnb_4bit_quant_type='nf4'  
10 )
```

LLaMA3-8B Fine-Tuning Model evaluation 코드 실습 (2)

```
1 # 기본 Llama 3 모델 로드
2 from transformers import AutoModelForCausalLM
3 base_model = AutoModelForCausalLM.from_pretrained(
4     "meta-llama/Meta-Llama-3-8B",
5     quantization_config = quantization_config,
6     device_map = {"": 0}
7 )
```



숨겨진 출력 표시

```
[ ] 1 # Fine Tune 된 모델 로드
2 from transformers import AutoModelForCausalLM
3 fine_tuned_model_cp_50 = AutoModelForCausalLM.from_pretrained(
4     fine_tuned_model_path,
5     quantization_config=quantization_config,
6     device_map = {"": 0}
7 )
```

```
[ ] 1 # Tokenizer 로드
2 from transformers import AutoTokenizer
3
4 tokenizer = AutoTokenizer.from_pretrained(fine_tuned_model_path)
```

LLaMA3-8B Fine-Tuning Model evaluation 코드 실습 (3)

```
[9] 1 # Prompt/Response Format 관련 설정
    2 EOS_TOKEN = tokenizer.eos_token
    3
    4 def convert_to_alpaca_format(instruction, response):
    5     alpaca_format_str = f"""Below is an instruction that describes a task. Write a response that appropriately completes the request.
    6     \n\n### Instruction: \n{instruction}\n\n### Response: \n{response}
    7     """
    8
    9     return alpaca_format_str
```

```
[10] 1 def test_model(instruction_str, model):
    2     inputs = tokenizer(
    3         [
    4             convert_to_alpaca_format(instruction_str, "")
    5         ], return_tensors = "pt").to("cuda")
    6     outputs = model.generate(**inputs, max_new_tokens = 128, use_cache = True, temperature = 0.05, top_p = 0.95)
    7     return(tokenizer.batch_decode(outputs)[0])
```

```
[11] 1 questions = [
    2     "Parameter-Efficient Fine Tuning에 대해 알려줘",
    3     "LLM에서 가장 유명한 LLM에는 뭐가 있어?",
    4     "What is a famous tall tower in Seoul?",
    5     "LLM에서 파인튜닝이 뭐야?",
    6     "운영체제가 뭐하는 거야?.",
    7     "메모리가 뭐야?"
    8 ]
```


LLaMA3-8B Fine-Tuning Model evaluation 코드 실습 (4)

```
1 answers_dict = {  
2   "fine_tuned_model_answers": []  
3 }  
4 for idx, question in enumerate(questions):  
5     print(f"Processing EXAMPLE {idx}=====")  
6     fine_tuned_model_output = test_model(question, fine_tuned_model_cp_50)  
7     answers_dict['fine_tuned_model_answers'].append(fine_tuned_model_output)
```

 숨겨진 출력 표시

```
[13] 1 def simple_format(text, width=120):  
2     return '\n'.join(line[i:i+width] for line in text.split('\n') for i in range(0, len(line), width))  
3  
4  
5 for idx, question in enumerate(questions):  
6     print(f"EXAMPLE {idx}=====")  
7     print(f"Question: {question}")  
8  
9     print("<<Fine Tuning Model 답변>>")  
10    fine_tuned_model_answer = answers_dict["fine_tuned_model_answers"][idx].split("### Response:")[1]  
11    print(simple_format(fine_tuned_model_answer))  
12    print()
```

LLaMA3-8B Fine-Tuning Model evaluation 코드 실습 (5)



EXAMPLE 0=====

Question: Parameter-Efficient Fine Tuning에 대해 알려줘

<<Fine Tuning Model 답변>>

Parameter-Efficient Fine Tuning은 기존의 모델을 재학습하는 기존의 방법과는 다르게, 모델의 파라미터를 효율적으로 사용하는 방법이다. 기존의 방법은 모델의 모든 파라미터를 재학습하는 반면, Parameter-Efficient Fine Tuning은 모델의 일부 파라미터만 재학습하고, 나머지 파라미터는 기존의 파라미터를 그대로 사용한다. 이 방법은 모델의 파라미터를 효율적으로 사용하면서도, 모델의 성능을 향상시킨다.

EXAMPLE 1=====

Question: LLM에서 가장 유명한 LLM에는 뭐가 있어?

<<Fine Tuning Model 답변>>

LLM에서 가장 유명한 LLM에는 여러 가지가 있습니다. 하지만, 가장 유명한 LLM은 GPT-3, DALL-E, ChatGPT, Midjourney, Stable Diffusion, Claude, Jasper, and OpenAI Codex가 있습니다. 이 LLM들은 모두 인공지능(AI) 기술을 기반으로 하고 있으며, 각기 다른 기능을 가지고 있습니다. GPT-3는 자연어 처리(NLP) 분야에서 가장 유명한 LLM이며, 대규모 언어 모델을 기반으로 한 인공지능(AI) 기술입니다. DALL-E는 이미지 생성

EXAMPLE 2=====

Question: What is a famous tall tower in Seoul?

<<Fine Tuning Model 답변>>

One of the most famous tall towers in Seoul is the N Seoul Tower, also known as Namsan Tower. It is located on Namsan Mountain, which is the highest point in the city, and stands at a height of 236 meters. The tower is a popular tourist attraction, offering panoramic views of the city and its surroundings. Visitors can take an elevator to the top of the tower, where there are several observation decks, restaurants, and a revolving restaurant. The tower is also home to the Seoul Sky, an observation deck that offers 360-degree views of the city. <|end_of_text|>

LLaMA3-8B Fine-Tuning Model evaluation 코드 실습 (6)

EXAMPLE 3=====

Question: LLM에서 파인튜닝이 뭐야?

<<Fine Tuning Model 답변>>

파인튜닝은 기존에 학습된 모델을 새로운 데이터에 맞게 재학습하는 것을 말합니다. 파인튜닝은 기존에 학습된 모델의 구조를 그대로 유지하면서 새로운 데이터에 맞게 모델을 재학습하는 '학습된 모델의 성능을 유지하면서 새로운 데이터에 맞는 모델을 만들기 위해 사용됩니다. 파인튜닝은 기존에 학습된 모델의 구조를 그대로 유지하면서 새로운 데이터에 맞게 모델을 재학습하는

EXAMPLE 4=====

Question: 운영체제가 뭐하는 거야?.

<<Fine Tuning Model 답변>>

운영체제는 컴퓨터의 하드웨어와 소프트웨어를 연결하고 관리하는 소프트웨어입니다. 운영체제는 컴퓨터의 자원(예: CPU, 메모리, 디스크 공간)을 관리하고, 사용자와 컴퓨터 간의 상호작용 프로그램을 실행하고 관리하는 역할을 합니다. 운영체제는 컴퓨터의 하드웨어를 추상화하고, 사용자가 컴퓨터의 하드웨어를 직접 제어하지 않고도 컴퓨터를 사용할 수 있도록 해줍니다. 운영체제는 .

EXAMPLE 5=====

Question: 메모리가 뭐야?

<<Fine Tuning Model 답변>>

메모리는 컴퓨터에서 데이터를 저장하고 읽기 위해 사용하는 장치입니다. 메모리는 컴퓨터의 CPU와 연결되어, 데이터를 저장하고 읽는 역할을 합니다. 메모리는 컴퓨터의 속도와 성능에 큰 컴퓨터의 성능을 높이기 위해 메모리의 용량을 늘리는 것이 일반적입니다. 메모리는 여러 종류가 있는데, 대표적인 메모리는 램(RAM), ROM, SSD 등이 있습니다. 램은 데이터를 임시로 저장하고

수고했어요!!!