Cheat Sheet: Generative AI Advanced Fine-Tuning for LLMs

Package/Method	Description	Code Example
CUDA-compatible GPU	Available in the system using PyTorch, a popular deep-learning framework. If a GPU is available, it assigns the device variable to "cuda" (CUDA, the parallel computing platform and application programming interface model developed by NVIDIA). If a GPU is not available, it assigns the device variable to "cpu" (which means the code will run on the CPU instead).	1. 1 2. 2 1. device = torch.device("cuda" if torch.cuda.is_available() else "cpu") 2. device Copied!
collate_fn	Shows that collate_fn function is used in conjunction with data loaders to customize the way batches are created from individual samples. A collate_batch function in PyTorch is used with data loaders to customize batch creation from individual samples. It processes a batch of data, including labels and text sequences. It applies the text_pipeline function to preprocess the text. The processed data is then converted into PyTorch tensors and returned as a tuple containing the label tensor, text tensor, and offsets tensor representing the starting positions of each text sequence in the combined tensor. The function also ensures that the returned tensors are moved to the specified device (GPU) for efficient computation.	1. 1 2. 2 3. 3 4. 4 5. 5 6. 6 7. 7 8. 8 9. 9 1. from torch.nn.utils.rnn import pad_sequence 2. def collate_batch(batch): 3. label_list, text_list = [], [] 4. for _label, _text in batch: 5. label_list.append(_label) 6. text_list.append(torch.tensor(text_pipeline(_text), dtype=torch.int64)) 7. label_list = torch.tensor(label_list, dtype=torch.int64) 8. text_list = pad_sequence(text_list, batch_first=True) 9. return label_list.to(device), text_list.to(device) Copied!
Training function	Helps in the training model, iteratively update the model's parameters to minimize the loss function. It improves the model's performance on a given task.	1. 1 2. 2 3. 3 4. 4 5. 5 6. 6 7. 7 8. 8 9. 9 10. 10 11. 11 12. 12 13. 13 14. 14 15. 15 16. 16 17. 17 18. 18 19. 19 20. 20 21. 21 22. 22 23. 23 24. 24 25. 25 26. 26 27. 27 28. 28 29. 29 30. 30 31. 31 32. 32 33. 33 34. 34 35. 35 36. 36 37. 37 38. 38 1. def train_model(model, optimizer, criterion, train_dataloader, valid_dataloader, epochs= 2. cum_loss_list = [] 3. acc_epoch = [] 4. acc_old = 0 5. model_path = os.path.join(save_dir, file_name) 6. acc_dir = os.path.join(save_dir, os.path.splitext(file_name)[0] + "_acc") 7. loss_dir = os.path.join(save_dir, os.path.splitext(file_name)[0] + "_loss") 8. time_start = time.time() 9. for epoch in tqdm(range(1, epochs + 1)): 10. model.train() 11. #print(model) 12. #for parm in model.parameters():

Package/Method	Description	Code Example

```
        #    print(parm.requires_grad)
        cum_loss = 0
        for idx, (label, text) in enumerate(train_dataloader):
            optimizer.zero_grad()
            label, text = label.to(device), text.to(device)
            predicted_label = model(text)
            loss = criterion(predicted_label, label)
            loss.backward()
            #print(loss)            torch.nn.utils.clip_grad_norm_(model.parameters(), 0.
            optimizer.step()
            cum_loss += loss.item()
        print(f"Epoch {epoch}/{epochs} - Loss: {cum_loss}")
        cum_loss_list.append(cum_loss)
        accu_val = evaluate_no_tqdm(valid_dataloader,model)
        acc_epoch.append(accu_val)
        if model_path and accu_val > acc_old:
            print(accu_val)
            acc_old = accu_val
            if save_dir is not None:
                pass
                #print("save model epoch",epoch)
                #torch.save(model.state_dict(), model_path)
                #save_list_to_file(lst=acc_epoch, filename=acc_dir)
                #save_list_to_file(lst=cum_loss_list, filename=loss_dir)
    time_end = time.time()
    print(f"Training time: {time_end - time_start}")
```

Copied!

DPO configuration

Involves training arguments, processing logging steps, evaluation strategies, and scheduling for model updates. Configuring DPO involves setting up administrative and technical measures for complying with data protection laws and regulations.

```
# DPO configuration
training_args = DPOConfig(
    # The beta parameter for the DPO loss function
    #beta is the temperature parameter for the DPO loss, typically something in the range
    beta=0.1,
    # The output directory for the training
    output_dir="dpo",
    # The number of training epochs
    num_train_epochs=5,
    # The batch size per device during training
    per_device_train_batch_size=1,
    # The batch size per device during evaluation
    per_device_eval_batch_size=1,
    # Whether to remove unused columns from the dataset
    remove_unused_columns=False,
    # The number of steps between logging training progress
    logging_steps=10,
    # The number of gradient accumulation steps
    gradient_accumulation_steps=1,
    # The learning rate for the optimization
    learning_rate=1e-4,
    # The evaluation strategy (e.g., after each step or epoch)
    eval_strategy="epoch",
    # The number of warmup steps for the learning rate scheduler
    warmup_steps=2,
    # Whether to use 16-bit (float16) precision
    fp16=False,
    # The number of steps between saving checkpoints
    save_steps=500,
    # The maximum number of checkpoints to keep
    #save_total_limit=2,
    # The reporting backend to use (set to 'none' to disable, you can also report to wand
    report_to='none'
```

Package/Method	Description	Code Example
DPOTraining class	Designed to equip professionals with the knowledge and skills for managing and overseeing data protection strategies in compliance with relevant laws and regulations.	Copied! 1. 1 2. 2 3. 3 4. 4 5. 5 6. 6 7. 7 8. 8 9. 9 10. 10 11. 11 12. 12 13. 13 14. 14 15. 15 16. 16 17. 17 18. 18 19. 19 20. 20 21. 21 22. 22 23. 23 24. 24 25. 25 1. tokenizer.pad_token = tokenizer.eos_token 2. # Create a DPO trainer 3. # This trainer will handle the fine-tuning of the model using the DPO technique 4. trainer = DPOTrainer(5. # The model to be fine-tuned 6. model, 7. # The reference model (not used in this case because LoRA has been used) 8. ref_model=None, 9. # The DPO training configuration 10. args=training_args, 11. # The beta parameter for the DPO loss function 12. beta=0.1, 13. # The training dataset 14. train_dataset=train_dataset, 15. # The evaluation dataset 16. eval_dataset=eval_dataset, 17. # The tokenizer for the model 18. tokenizer=tokenizer, 19. # The PEFT (Parallel Efficient Finetuning) configuration 20. peft_config=peft_config, 21. # The maximum prompt length 22. max_prompt_length=512, 23. # The maximum sequence length 24. max_length=512, 25.)
Retrieve and plot the training loss versus evaluation loss	Helps to retrieve log history and save it for data frame log. It also plots train and evaluation losses for epoch.	Copied! 1. 1 2. 2 3. 3 4. 4 5. 5 6. 6 7. 7 8. 8 9. 9 1. # Retrieve log_history and save it to a dataframe 2. log = pd.DataFrame(trainer.state.log_history) 3. log_t = log[log['loss'].notna()] 4. log_e = log[log['eval_loss'].notna()] 5. # Plot train and evaluation losses 6. plt.plot(log_t["epoch"], log_t["loss"], label = "train_loss") 7. plt.plot(log_e["epoch"], log_e["eval_loss"], label = "eval_loss") 8. plt.legend() 9. plt.show()
Traverse the IMDB data set	This code snippet traverses the IMDB data set by obtaining, loading, and exploring the data set. It also performs basic operations, visualizes the data, and analyzes and interprets the data set.	Copied! 1. 1 2. 2 3. 3 4. 4 5. 5 6. 6 7. 7 8. 8 9. 9 10. 10 11. 11 12. 12 13. 13 14. 14 15. 15 16. 16 17. 17 18. 18 19. 19 20. 20

Package/Method	Description	Code Example
		1. class IMDBDataset(Dataset): 2. def __init__(self, root_dir, train=True): 3. """ 4. root_dir: The base directory of the IMDB dataset. 5. train: A boolean flag indicating whether to use training or test data. 6. """ 7. self.root_dir = os.path.join(root_dir, "train" if train else "test") 8. self.neg_files = [os.path.join(self.root_dir, "neg", f) for f in os.listdir(os.pa 9. self.pos_files = [os.path.join(self.root_dir, "pos", f) for f in os.listdir(os.pa 10. self.files = self.neg_files + self.pos_files 11. self.labels = [0] * len(self.neg_files) + [1] * len(self.pos_files) 12. self.pos_inx=len(self.pos_files) 13. def __len__(self): 14. return len(self.files) 15. def __getitem__(self, idx): 16. file_path = self.files[idx] 17. label = self.labels[idx] 18. with open(file_path, 'r', encoding='utf-8') as file: 19. content = file.read() 20. return label, content Copied!
Iterators to train and test data sets	This code snippet indicates a path to the IMDB data set directory by combining temporary and subdirectory names. This code sets up the training and testing data iterators, retrieves the starting index of the training data, and prints the items from the training data set at indices.	1. 1 2. 2 3. 3 4. 4 5. 5 6. 6 7. 7 8. 8 1. root_dir = tempdir.name + '/' + 'imdb_dataset' 2. train_iter = IMDBDataset(root_dir=root_dir, train=True) # For 3. training data 4. test_iter = IMDBDataset(root_dir=root_dir, train=False) # For test 5. data 6. start=train_iter.pos_inx 7. for i in range(-10,10): 8. print(train_iter[start+i]) Copied!
yield_tokens function	Generates tokens from the collection of text data samples. The code snippet processes each text in 'data_iter' through the tokenizer and yields tokens to generate efficient, on-the-fly token generation suitable for tasks such as training machine learning models.	1. 1 2. 2 3. 3 4. 4 5. 5 1. tokenizer = get_tokenizer("basic_english") 2. def yield_tokens(data_iter): 3. """Yield tokens for each data sample.""" 4. for _, text in data_iter: 5. yield tokenizer(text) Copied!
Load pretrained model and its evaluation on test data	This code snippet helps download a pretrained model from URL, loads it into a specific architecture, and evaluates it on a test data set for assessing its performance.	1. 1 2. 2 3. 3 4. 4 1. urlopened = urlopen('https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/q 2. model_ = Net(vocab_size=vocab_size, num_class=2).to(device) 3. model_.load_state_dict(torch.load(io.BytesIO(urlopened.read()), map_location=device)) 4. evaluate(test_dataloader, model_) Copied!
Loading the Hugging Face model	This code snippet initiates a tokenizer using a pretrained 'bert-base-cased' model. It also downloads a pretrained model for the masked language model (MLM) task, and how to load the model configurations from a pretrained model.	1. 1 2. 2 3. 3 4. 4 5. 5 6. 6 7. 7 8. 8 9. 9 1. # Instantiate a tokenizer using the BERT base cased model 2. tokenizer = AutoTokenizer.from_pretrained("bert-base-cased") 3. # Download pretrained model from huggingface.co and cache. 4. model = BertForMaskedLM.from_pretrained('bert-base-cased') 5. # You can also start training from scratch by loading the model 6. configuration 7. # config = 8. AutoConfig.from_pretrained("google-bert/bert-base-cased") 9. # model = BertForMaskedLM.from_config(config) Copied!
Training a BERT model for MLM task	This code snippet trains the model with the specified parameters and data set. However, ensure that the 'SFTTrainer' is the appropriate trainer class for the task and the	1. 1 2. 2 3. 3 4. 4 5. 5 6. 6 7. 7

Package/Method	Description	Code Example
	model is properly defined for training.	8. 8 9. 9 10. 10 11. 11 12. 12 13. 13 14. 14 15. 15 16. 16 17. 17 1. training_args = TrainingArguments(2. output_dir="./trained_model", # Specify the output directory for the trained model 3. overwrite_output_dir=True, 4. do_eval=False, 5. learning_rate=5e-5, 6. num_train_epochs=1, # Specify the number of training epochs 7. per_device_train_batch_size=2, # Set the batch size for training 8. save_total_limit=2, # Limit the total number of saved checkpoints 9. logging_steps = 20 10.) 11. dataset = load_dataset("imdb", split="train") 12. trainer = SFTTrainer(13. model, 14. args=training_args, 15. train_dataset=dataset, 16. dataset_text_field="text", 17.) Copied!
Load the model and tokenizer	Useful for tasks where you need to quickly classify the sentiment of a piece of text with a pretrained, efficient transformer model.	1. 1 2. 2 1. tokenizer = DistilBertTokenizer.from_pretrained("distilbert-base-uncased-finetuned-sst-2- 2. model = DistilBertForSequenceClassification.from_pretrained("distilbert-base-uncased-fine Copied!
torch.no_grad()	The torch.no_grad() context manager disables gradient calculation. This reduces memory consumption and speeds up computation, as gradients are unnecessary for inference (i.e., when you are not training the model). The **inputs syntax is used to unpack a dictionary of keyword arguments in Python.	1. 1 2. 2 3. 3 1. # Perform inference 2. with torch.no_grad(): 3. outputs = model(**inputs) Copied!
Logits	The raw, unnormalized predictions of the model. Let's extract the logits from the model's outputs to perform further processing, such as determining the predicted class or calculating probabilities.	1. 1 2. 2 1. logits = outputs.logits 2. logits.shape Copied!
GPT-2 tokenizer	Helps to initialize the GPT-2 tokenizer using a pretrained model to handle encoding and decoding.	1. 1 2. 2 1. # Load the tokenizer and model 2. tokenizer = GPT2Tokenizer.from_pretrained("gpt2") Copied!
Load GPT-2 model	This code snippet initializes and loads the pretrained GPT-2 model. This code makes the GPT-2 model ready for generating text or other language tasks.	1. 1 2. 2 1. # Load the tokenizer and model 2. model = GPT2LMHeadModel.from_pretrained("gpt2") Copied!
Generate text	This code snippet generates text sequences based on the input and doesn't compute the gradient to generate output.	1. 1 2. 2 3. 3 4. 4 5. 5 6. 6 7. 7 8. 8 9. 9 10. 10 11. 11 12. 12 13. 13 1. # Generate text 2. output_ids = model.generate(3. inputs.input_ids, 4. attention_mask=inputs.attention_mask, 5. pad_token_id=tokenizer.eos_token_id, 6. max_length=50, 7. num_return_sequences=1 8.) 9. output_ids 10. or

Package/Method	Description	Code Example
		11. with torch.no_grad(): 12. outputs = model(**inputs) 13. outputs Copied! 1. 1 2. 2 3. 3
Decode the generated text	This code snippet decodes the text from the token IDs generated by a model. The code also decodes it into a readable string to print it.	1. # Decode the generated text 2. generated_text = tokenizer.decode(output_ids[0], skip_special_tokens=True) 3. print(generated_text) Copied! 1. 1 2. 2 3. 3 4. 4 5. 5 6. 6 7. 7 8. 8 9. 9 10. 10 11. 11 12. 12
Hugging Face pipeline() function	The pipeline() function from the Hugging Face Transformers library is a high-level API designed to simplify the usage of pretrained models for various natural language processing (NLP) tasks. It abstracts the complexities of model loading, tokenization, inference, and post-processing, allowing users to perform complex NLP tasks with just a few lines of code.	1. transformers.pipeline(2. task: str, 3. model: Optional = None, 4. config: Optional = None, 5. tokenizer: Optional = None, 6. feature_extractor: Optional = None, 7. framework: Optional = None, 8. revision: str = 'main', 9. use_fast: bool = True, 10. model_kwargs: Dict[str, Any] = None, 11. **kwargs 12.) Copied! 1. 1 2. 2 3. 3 4. 4 5. 5 6. 6 7. 7 8. 8
expected_outputs	Tokenize instructions and the instructions_with_responses. Then, count the number of tokens in instructions, and discard the equivalent amount of tokens from the beginning of the tokenized instructions_with_responses vector. Finally, discard the final token in instructions_with_responses, corresponding to the eos token. Decode the resulting vector using the tokenizer, resulting in the expected_output.	1. expected_outputs = [] 2. instructions_with_responses = formatting_prompts_func(test_dataset) 3. instructions = formatting_prompts_func_no_response(test_dataset) 4. for i in tqdm(range(len(instructions_with_responses))): 5. tokenized_instruction_with_response = tokenizer(instructions_with_responses[i], retur 6. tokenized_instruction = tokenizer(instructions[i], return_tensors="pt") 7. expected_output = tokenizer.decode(tokenized_instruction_with_response['input_ids'][0 8. expected_outputs.append(expected_output) Copied! 1. 1 2. 2 3. 3 4. 4 5. 5 6. 6 7. 7 8. 8
ListDataset	Inherits from Dataset and creates a torch Dataset from a list. This class is then used to generate a Dataset object from instructions.	1. class ListDataset(Dataset): 2. def __init__(self, original_list): 3. self.original_list = original_list 4. def __len__(self): 5. return len(self.original_list) 6. def __getitem__(self, i): 7. return self.original_list[i] 8. instructions_torch = ListDataset(instructions) Copied! 1. 1 2. 2 3. 3 4. 4 5. 5 6. 6 7. 7 8. 8 9. 9
gen_pipeline	This code snippet takes the token IDs from the model output, decodes it from the table text, and prints the responses.	1. gen_pipeline = pipeline("text-generation", 2. model=model, 3. tokenizer=tokenizer, 4. device=device, 5. batch_size=2,

Package/Method	Description	Code Example
		6. max_length=50, 7. truncation=True, 8. padding=False, 9. return_full_text=False)
torch.no_grad()	This code generates text from the given input using a pipeline while optimizing resource usage by limiting input size and reducing gradient calculations.	Copied! 1. 1 2. 2 3. 3 4. 4 5. 5 6. 6 7. 7 8. 8 9. 9 1. with torch.no_grad(): 2. # Due to resource limitation, only apply the function on 3 records using "instruction 3. pipeline_iterator= gen_pipeline(instructions_torch[:3], 4. max_length=50, # this is set to 50 due to resource co 5. num_beams=5, 6. early_stopping=True,) 7. generated_outputs_base = [] 8. for text in pipeline_iterator: 9. generated_outputs_base.append(text[0]["generated_text"])
load_dataset	The data set is loaded using the load_dataset function from the data sets library, specifically loading the "train" split.	Copied! 1. 1 2. 2 3. 3 4. 4 5. 5 6. 6 7. 7 8. 8 9. 9 1. dataset_name = "imdb" 2. ds = load_dataset(dataset_name, split = "train") 3. N = 5 4. for sample in range(N): 5. print('text',ds[sample]['text']) 6. print('label',ds[sample]['label']) 7. ds = ds.rename_columns({"text": "review"}) 8. ds 9. ds = ds.filter(lambda x: len(x["review"]) > 200, batched=False)
build_dataset	Incorporates the necessary steps to build a data set object for use as an input to PPOTrainer.	Copied! 1. 1 2. 2 3. 3 4. 4 5. 5 6. 6 7. 7 8. 8 9. 9 10. 10 11. 11 12. 12 13. 13 14. 14 15. 15 16. 16 17. 17 18. 18 19. 19 20. 20 21. 21 22. 22 23. 23 24. 24 25. 25 26. 26 27. 27 28. 28 29. 29 1. del(ds) 2. dataset_name="imdb" 3. ds = load_dataset(dataset_name, split="train") 4. ds = ds.rename_columns({"text": "review"}) 5. def build_dataset(config, dataset_name="imdb", input_min_text_length=2, input_max_text_le 6. """ 7. Build dataset for training. This builds the dataset from `load_dataset`, one should 8. customize this function to train the model on its own dataset. 9. Args: 10. dataset_name (`str`): 11. The name of the dataset to be loaded. 12. Returns: 13. dataloader (`torch.utils.data.DataLoader`): 14. The dataloader for the dataset. 15. """ 16. tokenizer = AutoTokenizer.from_pretrained(config.model_name) 17. tokenizer.pad_token = tokenizer.eos_token

Package/Method	Description	Code Example
		18. # load imdb with datasets 19. ds = load_dataset(dataset_name, split="train") 20. ds = ds.rename_columns({"text": "review"}) 21. ds = ds.filter(lambda x: len(x["review"]) > 200, batched=False) 22. input_size = LengthSampler(input_min_text_length, input_max_text_length) 23. def tokenize(sample): 24. sample["input_ids"] = tokenizer.encode(sample["review"])[: input_size()] 25. sample["query"] = tokenizer.decode(sample["input_ids"]) 26. return sample 27. ds = ds.map(tokenize, batched=False) 28. ds.set_format(type="torch") 29. return ds Copied!
Initialize PPOTrainer	Proximal policy optimization (PPO) is a reinforcement learning algorithm that is particularly well-suited for training generative models, including those used for chatbots. It helps address specific challenges in training these models, such as maintaining coherent and contextually appropriate dialogues. It improves policy gradient methods for chatbots by using a clipped objective function, which ensures gradual and stable policy updates. Specified configuration and components. config: Configuration settings for PPO training, such as learning rate and model name. model: The primary model to be fine-tuned using PPO. tokenizer: Tokenizer corresponding to the model, used for processing input text. dataset: Data set to be used for training, providing the input data for the model. data_collator: Data collator to handle batching and formatting of the input data.	1. 1 2. 2 3. 3 4. 4 5. 5 6. 6 1. ppo_trainer = PPOTrainer(config, model, ref_model, tokenizer, dataset=dataset, data_colla 2. print("ppo_trainer object ",ppo_trainer) 3. device = ppo_trainer.accelerator.device 4. if ppo_trainer.accelerator.num_processes == 1: 5. device = 0 if torch.cuda.is_available() else "cpu" 6. print(device) Copied!
output_length_sampler	Initialized with LengthSampler(output_min_length, output_max_length). This object is used to sample output lengths for the generated sequences, ensuring they fall within the specified minimum and maximum length range. By varying the lengths, we can produce more diverse and natural outputs from the language model, preventing the generation of overly short or excessively long sequences and enhancing the overall quality of the responses.	1. 1 2. 2 3. 3 1. output_min_length = 4 2. output_max_length = 16 3. output_length_sampler = LengthSampler(output_min_length, output_max_length) Copied!
max_new_tokens	Set the max_new_tokens parameter in the generation_kwargs dictionary to the value of gen_len, which was sampled from output_length_sampler. This ensures that the maximum number of new tokens generated by the language model is within the desired length range, promoting more controlled and appropriately lengthened responses.	1. 1 2. 2 1. generation_kwargs["max_new_tokens"] = gen_len 2. generation_kwargs Copied!
Text generation function	Tokenizes input text, generates a response, and decodes it.	1. 1 2. 2 3. 3 4. 4 5. 5 6. 6 7. 7 8. 8 1. gen_kwargs = {"min_length": -1, "top_k": 0.0, "top_p": 1.0, "do_sample": True, "pad_token 2. def generate_some_text(input_text,my_model): 3. # Tokenize the input text 4. input_ids = tokenizer(input_text, return_tensors='pt').input_ids.to(device) 5. generated_ids = my_model.generate(input_ids,**gen_kwargs) 6. # Decode the generated text 7. generated_text_ = tokenizer.decode(generated_ids[0], skip_special_tokens=True) 8. return generated_text_

Package/Method	Description	Code Example
Generate text with PPO model	Generate text using the PPO-trained model.	Copied! 1. 1 2. 2 3. 3 1. input_text = "Once upon a time in a land far" 2. generated_text=generate_some_text(input_text,model_1) 3. generated_text
Sentiment analysis	Analyze the sentiment of the generated text using sentiment_pipe.	Copied! 1. 1 2. 2 1. pipe_outputs = sentiment_pipe(generated_text, **sent_kwargs) 2. pipe_outputs
Generate text with reference model	Generate text using the reference model.	Copied! 1. 1 2. 2 1. generated_text = generate_some_text(input_text,ref_model) 2. generated_text
compare_models_on_dataset	This code snippet initializes the generation parameters, prepares the data set, and converts queries into tensors for the model input. It also generates a model, helps decode the text, and uses sentiment analysis to compute sentiment scores for concentrated texts before and after applying the model. This code also compiles results, stores original queries, and converts the dictionary to a pandas data frame for easy analysis.	Copied! 1. 1 2. 2 3. 3 4. 4 5. 5 6. 6 7. 7 8. 8 9. 9 10. 10 11. 11 12. 12 13. 13 14. 14 15. 15 16. 16 17. 17 18. 18 19. 19 20. 20 21. 21 22. 22 23. 23 24. 24 25. 25 26. 26 27. 27 28. 28 29. 29 30. 30 31. 31 32. 32 33. 33 34. 34 35. 35 36. 36 1. def compare_models_on_dataset(model, ref_model, dataset, tokenizer, sentiment_pipe, sent_ 2. gen_kwargs = { 3. "min_length": -1, 4. "top_k": 0.0, 5. "top_p": 1.0, 6. "do_sample": True, 7. "pad_token_id": tokenizer.eos_token_id 8. } 9. bs = 16 10. game_data = dict() 11. dataset.set_format("pandas") 12. df_batch = dataset[:].sample(bs) 13. game_data["query"] = df_batch["query"].tolist() 14. query_tensors = df_batch["input_ids"].tolist() 15. response_tensors_ref, response_tensors = [], [] 16. for i in range(bs): 17. gen_len = output_length_sampler() 18. output = ref_model.generate(19. torch.tensor(query_tensors[i]).unsqueeze(dim=0).to(device), 20. max_new_tokens=gen_len, 21. **gen_kwargs 22.).squeeze()[-gen_len:] 23. response_tensors_ref.append(output) 24. output = model.generate(torch.tensor(query_tensors[i]).unsqueeze(dim=0).to(devi 25. max_new_tokens=gen_len, 26. **gen_kwargs 27.).squeeze()[-gen_len:] 28. response_tensors.append(output) 29. game_data["response (before)"] = [tokenizer.decode(response_tensors_ref[i]) for i in 30. game_data["response (after)"] = [tokenizer.decode(response_tensors[i]) for i in range 31. texts_before = [q + r for q, r in zip(game_data["query"], game_data["response (before 32. game_data["rewards (before)"] = [output[1]["score"] for output in sentiment_pipe(text 33. texts_after = [q + r for q, r in zip(game_data["query"], game_data["response (after)"

Package/Method	Description	Code Example

```
    game_data["rewards (after)"] = [output[1]["score"] for output in sentiment_pipe(texts
    df_results = pd.DataFrame(game_data)
    return df_results
```

Copied!

This code snippet defines a function 'compare_models_on_dataset' for comparing the performance of two models by initializing generation parameters and setting the batch size, preparing the data set in the pandas format, and sampling the batch queries.

Tokenizing data

```
# Instantiate a tokenizer using the BERT base cased model
tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
# Define a function to tokenize examples
def tokenize_function(examples):
    # Tokenize the text using the tokenizer
    # Apply padding to ensure all sequences have the same length
    # Apply truncation to limit the maximum sequence length
    return tokenizer(examples["text"], padding="max_length", truncation=True)
# Apply the tokenize function to the dataset in batches
tokenized_datasets = dataset.map(tokenize_function, batched=True)
```

Copied!

Training loop

The train_model function trains a model using a set of training data provided through a dataloader. It begins by setting up a progress bar to help monitor the training progress visually. The model is switched to training mode, which is necessary for certain model behaviors like dropout to work correctly during training. The function processes the data in batches for each epoch, which involves several steps for each batch: transferring the data to the correct device (like a GPU), running the data through the model to get outputs and calculate loss, updating the model's parameters using the calculated gradients, adjusting the learning rate, and clearing the old gradients.

```
def train_model(model,tr_dataloader):
    # Create a progress bar to track the training progress
    progress_bar = tqdm(range(num_training_steps))
    # Set the model in training mode
    model.train()
    tr_losses=[]
    # Training loop
    for epoch in range(num_epochs):
        total_loss = 0
        # Iterate over the training data batches
        for batch in tr_dataloader:
            # Move the batch to the appropriate device
            batch = {k: v.to(device) for k, v in batch.items()}
            # Forward pass through the model
            outputs = model(**batch)
            # Compute the loss
            loss = outputs.loss
            # Backward pass (compute gradients)
            loss.backward()
            total_loss += loss.item()
            # Update the model parameters
            optimizer.step()
            # Update the learning rate scheduler
            lr_scheduler.step()
            # Clear the gradients
            optimizer.zero_grad()
            # Update the progress bar
            progress_bar.update(1)
        tr_losses.append(total_loss/len(tr_dataloader))
    #plot loss
    plt.plot(tr_losses)
```

Package/Method	Description	Code Example

```
    plt.title("Training loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.show()
```

Copied!

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
```

evaluate_model function	Works similarly to the train_model function but is used for evaluating the model's performance instead of training it. It uses a dataloader to process data in batches, setting the model to evaluation mode to ensure accuracy in measurements and disabling gradient calculations since it's not training. The function calculates predictions for each batch, updates an accuracy metric, and finally, prints the overall accuracy after processing all batches.	

```
def evaluate_model(model, evl_dataloader):
    # Create an instance of the Accuracy metric for multiclass classification with 5 clas
    metric = Accuracy(task="multiclass", num_classes=5).to(device)
    # Set the model in evaluation mode
    model.eval()
    # Disable gradient calculation during evaluation
    with torch.no_grad():
        # Iterate over the evaluation data batches
        for batch in evl_dataloader:
            # Move the batch to the appropriate device
            batch = {k: v.to(device) for k, v in batch.items()}
            # Forward pass through the model
            outputs = model(**batch)
            # Get the predicted class labels
            logits = outputs.logits
            predictions = torch.argmax(logits, dim=-1)
            # Accumulate the predictions and labels for the metric
            metric(predictions, batch["labels"])
    # Compute the accuracy
    accuracy = metric.compute()
    # Print the accuracy
    print("Accuracy:", accuracy.item())
```

Copied!

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
```

llm_model	This code snippet defines function 'llm_model' for generating text using the language model from the mistral.ai platform, specifically the 'mitral-8x7b-instruct-v01' model. The function helps in customizing generating parameters and interacts with IBM Watson's machine learning services.	

```
def llm_model(prompt_txt, params=None):
    model_id = 'mistralai/mixtral-8x7b-instruct-v01'
    default_params = {
        "max_new_tokens": 256,
        "min_new_tokens": 0,
        "temperature": 0.5,
        "top_p": 0.2,
        "top_k": 1
    }
    if params:
```

Package/Method	Description	Code Example

```
        default_params.update(params)
    parameters = {
        GenParams.MAX_NEW_TOKENS: default_params["max_new_tokens"],  # this controls the
        GenParams.MIN_NEW_TOKENS: default_params["min_new_tokens"], # this controls the m
        GenParams.TEMPERATURE: default_params["temperature"], # this randomness or creati
        GenParams.TOP_P: default_params["top_p"],
        GenParams.TOP_K: default_params["top_k"]
    }
    credentials = {
        "url": "https://us-south.ml.cloud.ibm.com"
    }
    project_id = "skills-network"
    model = Model(
        model_id=model_id,
        params=parameters,
        credentials=credentials,
        project_id=project_id
    )
    mixtral_llm = WatsonxLLM(model=model)
    response  = mixtral_llm.invoke(prompt_txt)
    return response
```

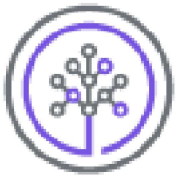
Copied!

RewardTrainer	The RewardTrainer orchestrates the training process, handling tasks such as batching, optimization, evaluation, and saving model checkpoints. It is particularly useful for training models that need to learn from feedback signals, improving their ability to generate high-quality responses.	

```
1
2
3
4
5
6
7
8
9
```

```
# Initialize RewardTrainer
trainer = RewardTrainer(
    model=model,
    args=training_args,
    tokenizer=tokenizer,
    train_dataset=dataset_dict['train'],
    eval_dataset=dataset_dict['test'],
    peft_config=peft_config,
)
```

Copied!



Skills Network