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).	`device = torch.device("cuda" if torch.cuda.is_available() else "cpu")` `device`

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.	`from torch.nn.utils.rnn import pad_sequence`
		`def collate_batch(batch):`
		`  label_list, text_list = [], []`
		`  for _label, _text in batch:`
		`  label_list.append(_label)`
		`  text_list.append(torch.tensor(text_pipeline(_text), dtype=torch.int64)`
		`  label_list = torch.tensor(label_list, dtype=torch.int64)`
		`  text_list = pad_sequence(text_list, batch_first=True)`
		`  return label_list.to(device), text_list.to(device)`

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.	`def train_model(model, optimizer, criterion, train_dataloader, valid_dataloade`
		`  cum_loss_list = []`
		`  acc_epoch = []`
		`  acc_old = 0`
		`  model_path = os.path.join(save_dir, file_name)`
		`  acc_dir = os.path.join(save_dir, os.path.splitext(file_name)[0] + "_acc")`
		`  loss_dir = os.path.join(save_dir, os.path.splitext(file_name)[0] + "_loss"`
		`  time_start = time.time()`
		`  for epoch in tqdm(range(1, epochs + 1)):`
		`  model.train()`
		`  #print(model)`
		`  #for parm in model.parameters():`
		`  #  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.param`
		`  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}")`

		`# DPO configuration`
		`training_args = DPOConfig(`
		`  # The beta parameter for the DPO loss function`
		`  #beta is the temperature parameter for the DPO loss, typically something i`
		`  beta=0.1,`
		`  # The output directory for the training`
		`  output_dir="dpo",`
		`  # The number of training epochs`

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.	`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 rep`
		`report_to='none'`

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.	`tokenizer.pad_token = tokenizer.eos_token`
		`# Create a DPO trainer`
		`# This trainer will handle the fine-tuning of the model using the DPO techniqu`
		`trainer = DPOTrainer(`
		`# The model to be fine-tuned`
		`model,`
		`# The reference model (not used in this case because LoRA has been use`
		`ref_model=None,`
		`# The DPO training configuration`
		`args=training_args,`
		`# The beta parameter for the DPO loss function`
		`beta=0.1,`
		`# The training dataset`
		`train_dataset=train_dataset,`
		`# The evaluation dataset`
		`eval_dataset=eval_dataset,`
		`# The tokenizer for the model`
		`tokenizer=tokenizer,`
		`# The PEFT (Parallel Efficient Finetuning) configuration`
		`peft_config=peft_config,`
		`# The maximum prompt length`
		`max_prompt_length=512,`
		`# The maximum sequence length`
		`max_length=512,`
		`)`

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.	`# Retrieve log_history and save it to a dataframe`
		`log = pd.DataFrame(trainer.state.log_history)`
		`log_t = log[log['loss'].notna()]`
		`log_e = log[log['eval_loss'].notna()]`
		`# Plot train and evaluation losses`
		`plt.plot(log_t["epoch"], log_t["loss"], label = "train_loss")`
		`plt.plot(log_e["epoch"], log_e["eval_loss"], label = "eval_loss")`
		`plt.legend()`
		`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.	`class IMDBDataset(Dataset):`
		`def __init__(self, root_dir, train=True):`
		`"""`
		`root_dir: The base directory of the IMDB dataset.`
		`train: A boolean flag indicating whether to use training or test data.`
		`"""`
		`self.root_dir = os.path.join(root_dir, "train" if train else "test")`
		`self.neg_files = [os.path.join(self.root_dir, "neg", f) for f in os.li`
		`self.pos_files = [os.path.join(self.root_dir, "pos", f) for f in os.li`
		`self.files = self.neg_files + self.pos_files`
		`self.labels = [0] * len(self.neg_files) + [1] * len(self.pos_files)`
		`self.pos_inx=len(self.pos_files)`
		`def __len__(self):`
		`return len(self.files)`
		`def __getitem__(self, idx):`
		`file_path = self.files[idx]`
		`label = self.labels[idx]`
		`with open(file_path, 'r', encoding='utf-8') as file:`
		`content = file.read()`
		`return label, content`

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	`root_dir = tempdir.name + '/' + 'imdb_dataset'`
		`train_iter = IMDBDataset(root_dir=root_dir, train=True) # For`
		`training data`
		`test_iter = IMDBDataset(root_dir=root_dir, train=False) # For test`
		`data`
		`start=train_iter.pos_inx`
		`for i in range(-10,10):`
		`print(train_iter[start+i])`

	data set at indices.	

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.	`tokenizer = get_tokenizer("basic_english")`
		`def yield_tokens(data_iter):`
		`  """Yield tokens for each data sample."""`
		`  for _, text in data_iter:`
		`  yield tokenizer(text)`

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.	`urlopened = urlopen('https://cf-courses-data.s3.us.cloud-object-storage.appdom`
		`model_ = Net(vocab_size=vocab_size, num_class=2).to(device)`
		`model_.load_state_dict(torch.load(io.BytesIO(urlopened.read()), map_location=d`
		`evaluate(test_dataloader, model_)`

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.	`# Instantiate a tokenizer using the BERT base cased model`
		`tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")`
		`# Download pretrained model from huggingface.co and cache.`
		`model = BertForMaskedLM.from_pretrained('bert-base-cased')`
		`# You can also start training from scratch by loading the model configuration`
		`# config =`
		`AutoConfig.from_pretrained("google-bert/bert-base-cased")`
		`# model = BertForMaskedLM.from_config(config)`

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 model is properly defined for training.	`training_args = TrainingArguments(`
		`  output_dir="./trained_model",  # Specify the output directory for the trai`
		`  overwrite_output_dir=True,`
		`  do_eval=False,`
		`  learning_rate=5e-5,`
		`  num_train_epochs=1,  # Specify the number of training epochs`
		`  per_device_train_batch_size=2,  # Set the batch size for training`
		`  save_total_limit=2,  # Limit the total number of saved checkpoints`
		`  logging_steps = 20`
		`)`
		`dataset = load_dataset("imdb", split="train")`
		`trainer = SFTTrainer(`
		`  model,`
		`  args=training_args,`
		`  train_dataset=dataset,`
		`  dataset_text_field="text",`
		`)`

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.	`tokenizer = DistilBertTokenizer.from_pretrained("distilbert-base-uncased-finet` `model = DistilBertForSequenceClassification.from_pretrained("distilbert-base-u`

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.	`# Perform inference`
		`with torch.no_grad():`
		`  outputs = model(**inputs)`

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.	`logits = outputs.logits` `logits.shape`
GPT-2 tokenizer	Helps to initialize the GPT-2 tokenizer using a pretrained model to handle encoding and decoding.	`# Load the tokenizer and model` `tokenizer = GPT2Tokenizer.from_pretrained("gpt2")`
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.	`# Load the tokenizer and model` `model = GPT2LMHeadModel.from_pretrained("gpt2")`

Generate text	This code snippet generates text sequences based on the input and doesn't compute the gradient to	`# Generate text`
		`output_ids = model.generate(`
		`  inputs.input_ids,`
		`  attention_mask=inputs.attention_mask,`
		`  pad_token_id=tokenizer.eos_token_id,`
		`  max_length=50,`
		`  num_return_sequences=1`
		`)`

	generate output.	output_ids or with torch.no_grad(): outputs = model(**inputs) outputs

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.	# Decode the generated text
		generated_text = tokenizer.decode(output_ids[0], skip_special_tokens=True)
		print(generated_text)

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.	transformers.pipeline(task: str, model: Optional = None, config: Optional = None, tokenizer: Optional = None, feature_extractor: Optional = None, framework: Optional = None, revision: str = 'main', use_fast: bool = True, model_kwargs: Dict[str, Any] = None, **kwargs)
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.	expected_outputs = [] instructions_with_responses = formatting_prompts_func(test_dataset) instructions = formatting_prompts_func_no_response(test_dataset) for i in tqdm(range(len(instructions_with_responses))): tokenized_instruction_with_response = tokenizer(instructions_with_response tokenized_instruction = tokenizer(instructions[i], return_tensors="pt") expected_output = tokenizer.decode(tokenized_instruction_with_response['in expected_outputs.append(expected_output)

ListDataset	Inherits from Dataset and creates a torch Dataset from a list. This class is then used to generate a Dataset object from instructions.	class ListDataset(Dataset):
		def __init__(self, original_list):
		self.original_list = original_list
		def __len__(self):
		return len(self.original_list)
		def __getitem__(self, i):
		return self.original_list[i]
		instructions_torch = ListDataset(instructions)

gen_pipeline	This code snippet takes the token IDs from the model output, decodes it from the table text, and prints the responses.	gen_pipeline = pipeline("text-generation", model=model, tokenizer=tokenizer, device=device, batch_size=2, max_length=50, truncation=True, padding=False, 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.	with torch.no_grad():
		# Due to resource limitation, only apply the function on 3 records using "
		pipeline_iterator= gen_pipeline(instructions_torch[:3],
		max_length=50, # this is set to 50 due to
		num_beams=5,
		early_stopping=True,)
		generated_outputs_base = []
		for text in pipeline_iterator:
		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.	dataset_name = "imdb" ds = load_dataset(dataset_name, split = "train") N = 5 for sample in range(N): print('text',ds[sample]['text']) print('label',ds[sample]['label']) ds = ds.rename_columns({"text": "review"}) ds ds = ds.filter(lambda x: len(x["review"]) > 200, batched=False)
		del(ds) dataset_name="imdb" ds = load_dataset(dataset_name, split="train") ds = ds.rename_columns({"text": "review"}) def build_dataset(config, dataset_name="imdb", input_min_text_length=2, input_ """ Build dataset for training. This builds the dataset from `load_dataset`, o customize this function to train the model on its own dataset. Args: dataset name (`str`):

build_dataset	Incorporates the necessary steps to build a data set object for use as an input to PPOTrainer.	The name of the dataset to be loaded.
		Returns:
		dataloader (`torch.utils.data.DataLoader`):
		The dataloader for the dataset.
		"""
		tokenizer = AutoTokenizer.from_pretrained(config.model_name)
		tokenizer.pad_token = tokenizer.eos_token
		# load imdb with datasets
		ds = load_dataset(dataset_name, split="train")
		ds = ds.rename_columns({"text": "review"})
		ds = ds.filter(lambda x: len(x["review"]) > 200, batched=False)
		input_size = LengthSampler(input_min_text_length, input_max_text_length)
		def tokenize(sample):
		sample["input_ids"] = tokenizer.encode(sample["review"])[: input_size(
		sample["query"] = tokenizer.decode(sample["input_ids"])
		return sample
		ds = ds.map(tokenize, batched=False)
		ds.set_format(type="torch")
		return ds

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.	ppo_trainer = PPOTrainer(config, model, ref_model, tokenizer, dataset=dataset, print("ppo_trainer object ",ppo_trainer) device = ppo_trainer.accelerator.device if ppo_trainer.accelerator.num_processes == 1: device = 0 if torch.cuda.is_available() else "cpu" print(device)
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.	output_min_length = 4 output_max_length = 16 output_length_sampler = LengthSampler(output_min_length, output_max_length)
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.	generation_kwargs["max_new_tokens"] = gen_len generation_kwargs

Text generation function	Tokenizes input text, generates a response, and decodes it.	gen_kwargs = {"min_length": -1, "top_k": 0.0, "top_p": 1.0, "do_sample": True,
		def generate_some_text(input_text,my_model):
		# Tokenize the input text
		input_ids = tokenizer(input_text, return_tensors='pt').input_ids.to(device
		generated_ids = my_model.generate(input_ids,**gen_kwargs)
		# Decode the generated text
		generated_text_ = tokenizer.decode(generated_ids[0], skip_special_tokens=T
		return generated_text_

Generate text with PPO model	Generate text using the PPO-trained model.	`input_text = "Once upon a time in a land far"`
		`generated_text=generate_some_text(input_text,model_1)`
		`generated_text`

Sentiment analysis	Analyze the sentiment of the generated text using sentiment_pipe.	`pipe_outputs = sentiment_pipe(generated_text, **sent_kwargs)` `pipe_outputs`
Generate text with reference model	Generate text using the reference model.	`generated_text = generate_some_text(input_text,ref_model)` `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.	`def compare_models_on_dataset(model, ref_model, dataset, tokenizer, sentiment_`
		`  gen_kwargs = {`
		`  "min_length": -1,`
		`  "top_k": 0.0,`
		`  "top_p": 1.0,`
		`  "do_sample": True,`
		`  "pad_token_id": tokenizer.eos_token_id`
		`  }`
		`  bs = 16`
		`  game_data = dict()`
		`  dataset.set_format("pandas")`
		`  df_batch = dataset[:].sample(bs)`
		`  game_data["query"] = df_batch["query"].tolist()`
		`  query_tensors = df_batch["input_ids"].tolist()`
		`  response_tensors_ref, response_tensors = [], []`
		`  for i in range(bs):`
		`  gen_len = output_length_sampler()`
		`  output = ref_model.generate(`
		`torch.tensor(query_tensors[i]).unsqueeze(dim=0).to(device),`
		`  max_new_tokens=gen_len,`
		`  **gen_kwargs`
		`  ).squeeze()[-gen_len:]`
		`  response_tensors_ref.append(output)`
		`  output = model.generate(  torch.tensor(query_tensors[i]).unsqueeze(dim`
		`  max_new_tokens=gen_len,`
		`  **gen_kwargs`
		`  ).squeeze()[-gen_len:]`
		`  response_tensors.append(output)`
		`  game_data["response (before)"] = [tokenizer.decode(response_tensors_ref[i]`
		`  game_data["response (after)"] = [tokenizer.decode(response_tensors[i]) for`
		`  texts_before = [q + r for q, r in zip(game_data["query"], game_data["respo`
		`  game_data["rewards (before)"] = [output[1]["score"] for output in sentimen`
		`  texts_after = [q + r for q, r in zip(game_data["query"], game_data["respon`
		`  game_data["rewards (after)"] = [output[1]["score"] for output in sentiment_`
		`  df_results = pd.DataFrame(game_data)`
		`  return df_results`

Tokenizing data	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.	`# 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)`

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

		plt.show()

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
		  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())

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:
		  default_params.update(params)
		  parameters = {
		  GenParams.MAX_NEW_TOKENS: default_params["max_new_tokens"], # this co
		  GenParams.MIN_NEW_TOKENS: default_params["min_new_tokens"], # this con
		  GenParams.TEMPERATURE: default_params["temperature"], # this randomnes
		  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

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.	# 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,
		)