Cheat Sheet : Generative AI Engineering and Fine-Tuning Transformers

<table>
<tr><th>Package/ Method</th><th>Description</th><th>Code Example</th></tr>
<tr><td>Positional encoding</td><td>Pivotal in transformers and sequence-to-sequence models, conveying critical information regarding the positions or sequencing of elements within a given sequence.</td><td><pre><code>class PositionalEncoding(nn.Module):
    """
    https://pytorch.org/tutorials/beginner/transformer_tutorial.html
    """

    def __init__(self, d_model, vocab_size=5000, dropout=0.1):
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)

        pe = torch.zeros(vocab_size, d_model)
        position = torch.arange(0, vocab_size, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(
            torch.arange(0, d_model, 2).float()
            * (-math.log(10000.0) / d_model)
)
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)
        self.register_buffer("pe", pe)

    def forward(self, x):
        x = x + self.pe[:, : x.size(1), :]
        return self.dropout(x)</code></pre></td></tr>
</table>

Importing IMBD dataset	The **IMDB dataset** contains movie reviews from the Internet Movie Database (IMDB) and is commonly used for binary sentiment classification tasks. It's a popular dataset for training and testing models in natural language processing (NLP), particularly in sentiment analysis.	`urlopened = urlopen('https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/35t-FeC-2uN1ozOwPs7wFg.gz')` `tar = tarfile.open(fileobj=io.BytesIO(urlopened.read()))` `tempdir = tempfile.TemporaryDirectory()` `tar.extractall(tempdir.name)` `tar.close()`
IMDBDataset class to create iterators for the train and test datasets	Creates iterators for training and testing data sets that involve various steps, such as data loading, preprocessing, and creating iterators.	`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])`
GloVe embeddings	An unsupervised learning algorithm to obtain vector representations for words. GloVe model is trained on the aggregated global word-to-word co-occurrence statistics from a corpus, and the resulting representations show linear substructures of the word vector base.	`class GloVe_override(Vectors):` `    url = {` `        "6B": "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/tQdezXocAJMBMPfUJx_iUg/glove-6B.zip",` `    }` `    def __init__(self, name="6B", dim=100, **kwargs) -> None:` `        url = self.url[name]` `        name = "glove.{}.{}d.txt".format(name, str(dim))` `        #name = "glove.{}/glove.{}.{}d.txt".format(name, name, str(dim))` `        super(GloVe_override, self).__init__(name, url=url, **kwargs)`

		class GloVe_override2(Vectors): url = { "6B": "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/tQdezXocAJMBMPfUJx_iUg/glove-6B.zip", } def __init__(self, name="6B", dim=100, **kwargs) -> None: url = self.url[name] #name = "glove.{}.{}d.txt".format(name, str(dim)) name = "glove.{}/glove.{}.{}d.txt".format(name, name, str(dim)) super(GloVe_override2, self).__init__(name, url=url, **kwargs) try: glove_embedding = GloVe_override(name="6B", dim=100) except: try: glove_embedding = GloVe_override2(name="6B", dim=100) except: glove_embedding = GloVe(name="6B", dim=100)
Building vocabulary object from pretrained GloVe word embedding model	Involves various steps for creating a structured representation of words and their corresponding vector embeddings.	from torchtext.vocab import GloVe,vocab # Build vocab from glove_vectors vocab = vocab(glove_embedding .stoi, 0,specials=('<unk>', '<pad>')) vocab.set_default_index(vocab["<unk>"])

<table>
<tr>
<td>Convert the training and testing iterators to map-style datasets</td>
<td>The training data set will contain 95% of the samples in the original training set, while the validation data set will contain the remaining 5%. These data sets can be used for training and evaluating a machine-learning model for text classification on the IMDB data set. The final performance of the model will be evaluated on the hold-out test set.</td>
<td><pre><code>train_dataset = to_map_style_dataset(train_iter)
test_dataset = to_map_style_dataset(test_iter)</code></pre></td>
</tr>
<tr>
<td>CUDA-compatible GPU</td>
<td>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).</td>
<td><pre><code>device = torch.device("cuda" if
torch.cuda.is_available() else "cpu")
device</code></pre></td>
</tr>
<tr>
<td>collate_fn</td>
<td>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</td>
<td><pre><code>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))</code></pre></td>
</tr>
</table>

	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.	`    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)`
Convert the data set objects to data loaders	Used in PyTorch-based projects. It includes creating data set objects, specifying data loading parameters, and converting these datasets into data loaders.	`ATCH_SIZE = 32` `train_dataloader = DataLoader(` `    split_train_, batch_size=BATCH_SIZE, shuffle=True,` `collate_fn=collate_batch` `)` `valid_dataloader = DataLoader(` `    split_valid_, batch_size=BATCH_SIZE, shuffle=True,` `collate_fn=collate_batch` `)` `test_dataloader = DataLoader(` `    test_dataset, batch_size=BATCH_SIZE, shuffle=True,` `collate_fn=collate_batch` `)`
Predict function	The predict function takes in a text, a text pipeline, and a model as inputs. It uses a pretrained model passed as a parameter to predict the label of the text for text classification on the IMDB data set.	`def predict(text, text_pipeline, model):` `    with torch.no_grad():` `        text =` `torch.unsqueeze(torch.tensor(text_pipeline(text)),0).to(` `device)` `        model.to(device)` `        output = model(text)`

<table>
<tr>
<td></td>
<td></td>
<td>

```
        return imdb_label[output.argmax(1).item()]
```

</td>
</tr>
<tr>
<td>Training function</td>
<td>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.</td>
<td>

```
def train_model(model, optimizer, criterion,
train_dataloader, valid_dataloader,  epochs=1000,
save_dir="", file_name=None):
    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.parameters(), 0.1)
```

</td>
</tr>
</table>

<table>
<tr>
<td></td>
<td></td>
<td>
<pre><code>   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}")</code></pre>
</td>
</tr>
<tr>
<td>Fine-tune a model in the AG News data set</td>
<td>Fine-tuning a model on the pretrained AG News data set is to categorize news articles into one of four categories: Sports, Business, Sci/Tech, or World. Start training a model from scratch on the AG News data set. If you want to train the model for 2 epochs on a smaller data set to demonstrate</td>
<td>
<pre><code>train_iter_ag_news = AG_NEWS(split="train")

num_class_ag_news = len(set([label for (label, text) in
train_iter_ag_news]))
num_class_ag_news

Split the dataset into training and testing iterators.
train_iter_ag_news, test_iter_ag_news = AG_NEWS()</code></pre>
</td>
</tr>
</table>

<table>
<tr>
<td></td>
<td>what the training process would look like, uncomment the part that says ### Uncomment to Train ### before running the cell. Training for 2 epochs on the reduced data set can take approximately 3 minutes.</td>
<td>

```
# Convert the training and testing iterators to map-style datasets.
train_dataset_ag_news = to_map_style_dataset(train_iter_ag_news)
test_dataset_ag_news = to_map_style_dataset(test_iter_ag_news)

# Determine the number of samples to be used for training and validation (5% for validation).
num_train_ag_news = int(len(train_dataset_ag_news) * 0.95)

# Randomly split the training dataset into training and validation datasets using `random_split`.
# The training dataset will contain 95% of the samples, and the validation dataset will contain the remaining 5%.
split_train_ag_news_, split_valid_ag_news_ = random_split(train_dataset_ag_news, [num_train_ag_news, len(train_dataset_ag_news) - num_train_ag_news])

# Make the training set smaller to allow it to run fast as an example.
# IF YOU WANT TO TRAIN ON THE AG_NEWS DATASET, COMMENT OUT THE 2 LINEs BELOW.
# HOWEVER, NOTE THAT TRAINING WILL TAKE A LONG TIME
num_train_ag_news = int(len(train_dataset_ag_news) * 0.05)
split_train_ag_news_, _ = random_split(split_train_ag_news_, [num_train_ag_news, len(split_train_ag_news_) - num_train_ag_news])
```

</td>
</tr>
</table>

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

def label_pipeline(x):
   return int(x) - 1

from torch.nn.utils.rnn import pad_sequence

def collate_batch_ag_news(batch):
    label_list, text_list = [], []
    for _label, _text in batch:
        label_list.append(label_pipeline(_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)

BATCH_SIZE = 32

train_dataloader_ag_news = DataLoader(
    split_train_ag_news_, batch_size=BATCH_SIZE,
shuffle=True, collate_fn=collate_batch_ag_news
)
valid_dataloader_ag_news = DataLoader(
```

		`` split_valid_ag_news_, batch_size=BATCH_SIZE, shuffle=True, collate_fn=collate_batch_ag_news) test_dataloader_ag_news = DataLoader(test_dataset_ag_news, batch_size=BATCH_SIZE, shuffle=True, collate_fn=collate_batch_ag_news) model_ag_news = Net(num_class=4,vocab_size=vocab_size).to(device) model_ag_news.to(device) ''' ### Uncomment to Train ### LR=1 criterion = torch.nn.CrossEntropyLoss() optimizer = torch.optim.SGD(model_ag_news.parameters(), lr=LR) scheduler = torch.optim.lr_scheduler.StepLR(optimizer, 1.0, gamma=0.1) save_dir = "" file_name = "model_AG News small1.pth" train_model(model=model_ag_news, optimizer=optimizer, criterion=criterion, train_dataloader=train_dataloader_ag_news, valid_dataloader=valid_dataloader_ag_news, epochs=2, save_dir=save_dir, file_name=file_name) ''' ```
Cost and validation data accuracy for each epoch	Plots the cost and validation data accuracy for each epoch of the pretrained model up to and including the epoch that yielded	``` acc_urlopened = urlopen('https://cf-courses- data.s3.us.cloud-object- storage.appdomain.cloud/bQk8mJu3Uct3I4JEsEtRnw/model- AG%20News%20small1-acc') ```

	the highest accuracy. As you can see, the pretrained model achieved a high accuracy of over 90% on the AG News validation set.	`loss_urlopened = urlopen('https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/KNQkqJWWwY_XfbFBRFhZNA/model-AG%20News%20small1-loss')` `acc_epoch = pickle.load(acc_urlopened)` `cum_loss_list = pickle.load(loss_urlopened)` `plot(cum_loss_list,acc_epoch)`
Fine-tune the final layer	Fine-tuning the final output layer of a neural network is similar to fine-tuning the whole model. You can begin by loading the pretrained model you would like to fine-tune. In this case, the same model is pretrained on the AG News data set.	`urlopened = urlopen('https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/9c3Dh2O_jsYBShBuchUNlg/model-AG%20News%20small1.pth')` `model_fine2 = Net(vocab_size=vocab_size, num_class=4).to(device)` `model_fine2.load_state_dict(torch.load(io.BytesIO(urlopened.read()), map_location=device))`
Fine-tune full IMDB training set for 100 epoch	The code snippet helps achieve a well-optimized model that accurately classifies movie reviews into positive or negative sentiments.	`acc_urlopened = urlopen('https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/UdR3ApQnxSeV2mrA0CbiLg/model-fine2-acc')` `loss_urlopened = urlopen('https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/rWGDIF-uL2dEngWcIo9teQ/model-fine2-loss')` `acc_epoch = pickle.load(acc_urlopened)` `cum_loss_list = pickle.load(loss_urlopened)` `plot(cum_loss_list,acc_epoch)`
Adaptor model	FeatureAdapter is a neural network module that introduces a low-dimensional bottleneck in a transformer architecture to allow fine-tuning with fewer parameters. It compresses the original high-dimensional embeddings into a lower dimension, applies a	`class FeatureAdapter(nn.Module):` `    """` `    Attributes:` `        size (int): The bottleneck dimension to which the embeddings are temporarily reduced.` `        model_dim (int): The original dimension of the embeddings or features in the transformer model.`

nonlinear transformation, and then expands it back to the original dimension. This process is followed by a residual connection that adds the transformed output back to the original input to preserve information and promote gradient flow.

```
    """
    def __init__(self, bottleneck_size=50, model_dim=100):
        super().__init__()
        self.bottleneck_transform = nn.Sequential(
            nn.Linear(model_dim, bottleneck_size),  # Down-project to a smaller dimension
            nn.ReLU(),                               # Apply non-linearity
            nn.Linear(bottleneck_size, model_dim)  # Up-project back to the original dimension
        )

    def forward(self, x):
        """
        Forward pass of the FeatureAdapter. Applies the bottleneck transformation to the input
        tensor and adds a skip connection.

        Args:
            x (Tensor): Input tensor with shape (batch_size, seq_length, model_dim).

        Returns:
            Tensor: Output tensor after applying the adapter transformation and skip connection,
                    maintaining the original input shape.
        """
        transformed_features = self.bottleneck_transform(x)  # Transform features through the bottleneck
        output_with_residual = transformed_features + x  # Add the residual connection
```

<table>
<tr>
<td></td>
<td></td>
<td>

```
        return output_with_residual
```

</td>
</tr>
<tr>
<td>Traverse the IMDB data set</td>
<td>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.</td>
<td>

```
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.listdir(os.path.join(self.root_dir, "neg")) if f.endswith('.txt')]
        self.pos_files = [os.path.join(self.root_dir, "pos", f) for f in os.listdir(os.path.join(self.root_dir, "pos")) if f.endswith('.txt')]
        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()
```

</td>
</tr>
</table>

<table>
<tr><td></td><td></td><td><pre><code>   return label, content</code></pre></td></tr>
<tr><td>Iterators to train and test datasets</td><td>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 dataset at indices.</td><td><pre><code>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])</code></pre></td></tr>
<tr><td>yield_tokens function</td><td>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.</td><td><pre><code>tokenizer = get_tokenizer("basic_english")
def yield_tokens(data_iter):
    """Yield tokens for each data sample."""
    for _, text in data_iter:
        yield tokenizer(text)</code></pre></td></tr>
<tr><td>Load pretrained model and its evaluation on test data</td><td>This code snippet helps download a pretrained model from URL, loads it into a specific architecture, and evaluates it on a test dataset for assessing its performance.</td><td><pre><code>urlopened = urlopen('https://cf-courses-
data.s3.us.cloud-object-
storage.appdomain.cloud/q66IH6a7lglkZ4haM6hB1w/model-
IMDB%20dataset%20small2.pth')
model_ = Net(vocab_size=vocab_size,
num_class=2).to(device)
model_.load_state_dict(torch.load(io.BytesIO(urlopened.r
ead()), map_location=device))
evaluate(test_dataloader, model_)</code></pre></td></tr>
<tr><td>Loading the Hugging Face model</td><td>This code snippet initiates a tokenizer using a pretrained ‘bert-base-cased’ model. It also downloads a pretrained model for the masked language model</td><td><pre><code># Instantiate a tokenizer using the BERT base cased
model
tokenizer = AutoTokenizer.from_pretrained("bert-base-
cased")</code></pre></td></tr>
</table>

	(MLM) task, and how to load the model configurations from a pretrained model.	`# 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 dataset. However, ensure that the 'SFTTrainer' is the appropriate trainer class for the task and that the model is properly defined for training.	`training_args = TrainingArguments(` `    output_dir="./trained_model",  # Specify the output directory for the trained model` `    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-finetuned-sst-2-english")` `model = DistilBertForSequenceClassification.from_pretrained("distilbert-base-uncased-finetuned-sst-2-english")`
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 (for example, 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)`
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 output.	`# Generate text` `output_ids = model.generate(` `    inputs.input_ids,` `    attention_mask=inputs.attention_mask,` `    pad_token_id=tokenizer.eos_token_id,` `    max_length=50,`

<table>
<tr>
<td></td>
<td></td>
<td><pre><code>   num_return_sequences=1
)

output_ids

or
with torch.no_grad():
    outputs = model(**inputs)
outputs</code></pre></td>
</tr>
<tr>
<td>Decode the generated text</td>
<td>This code snippet decodes the text from the token IDs generated by a model. It also decodes it into a readable string to print it.</td>
<td><pre><code># Decode the generated text
generated_text = tokenizer.decode(output_ids[0],
skip_special_tokens=True)

print(generated_text)</code></pre></td>
</tr>
<tr>
<td>Hugging Face pipeline() function</td>
<td>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.</td>
<td><pre><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
)</code></pre></td>
</tr>
<tr>
<td>formatting_prompts_func_no_response function</td>
<td>The prompt function generates formatted text prompts from a dataset by using the instructions from the dataset. It creates strings</td>
<td><pre><code>def formatting_prompts_func(mydataset):
    output_texts = []
    for i in range(len(mydataset['instruction'])):
        text = (</code></pre></td>
</tr>
</table>

	that include only the instruction and a placeholder for the response.	`f"### Instruction:\n{mydataset['instruction'][i]}"` `f"\n\n### Response:\n{mydataset['output'][i]}</s>"` `)` `output_texts.append(text)` `return output_texts` `def formatting_prompts_func_no_response(mydataset):` `output_texts = []` `for i in range(len(mydataset['instruction'])):` `text = (` `f"### Instruction:\n{mydataset['instruction'][i]}"` `f"\n\n### Response:\n"` `)` `output_texts.append(text)` `return output_texts`
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_responses[i], return_tensors="pt", max_length=1024, truncation=True, padding=False)` `tokenized_instruction = tokenizer(instructions[i], return_tensors="pt")` `expected_output = tokenizer.decode(tokenized_instruction_with_response['in`

<table>
<tr><td></td><td></td><td><pre>put_ids'][0][len(tokenized_instruction['input_ids'][0])-
1:], skip_special_tokens=True)
    expected_outputs.append(expected_output)</pre></td></tr>
<tr><td>ListDataset</td><td>Inherits from Dataset and creates a torch Dataset from a list. This class is then used to generate a Dataset object from instructions.</td><td><pre>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)</pre></td></tr>
<tr><td>gen_pipeline</td><td>This code snippet takes the token IDs from the model output, decodes it from the table text, and prints the responses.</td><td><pre>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)</pre></td></tr>
<tr><td>torch.no_grad()</td><td>This code generates text from the given input using a pipeline while optimizing resource usage by limiting input size and reducing gradient calculations.</td><td><pre>with torch.no_grad():
    # Due to resource limitation, only apply the
function on 3 records using "instructions_torch[:10]"
    pipeline_iterator=
gen_pipeline(instructions_torch[:3],
                                    max_length=50, #
this is set to 50 due to resource constraint, using a
GPU, you can increase it to the length of your choice
                                    num_beams=5,</pre></td></tr>
</table>

<table>
<tr><td></td><td></td><td><pre><code>early_stopping=True,)

generated_outputs_base = []
for text in pipeline_iterator:

generated_outputs_base.append(text[0]["generated_text"])</code></pre></td></tr>
<tr><td>SFTTrainer</td><td>This code snippet sets and initializes a training configuration for a model using ‘SFTTrainer’ by specifying parameters and initializes the ‘SFTTrainer’ with the model, datasets, and additional settings.</td><td><pre><code>training_args = SFTConfig(
    output_dir="/tmp",
    num_train_epochs=10,
    save_strategy="epoch",
    fp16=True,
    per_device_train_batch_size=2, # Reduce batch size
    per_device_eval_batch_size=2, # Reduce batch size
    max_seq_length=1024,
    do_eval=True
)

trainer = SFTTrainer(
    model,
    train_dataset=train_dataset,
    eval_dataset=test_dataset,
    formatting_func=formatting_prompts_func,
    args=training_args,
    packing=False,
    data_collator=collator,
)</code></pre></td></tr>
<tr><td>torch.no_grad()</td><td>This code snippet helps generate text sequences from the pipeline function. It ensures that the gradient computations are disabled and optimizes the performance and memory usage.</td><td><pre><code>with torch.no_grad():
    # Due to resource limitation, only apply the
function on 3 records using "instructions_torch[:10]"
    pipeline_iterator=
gen_pipeline(instructions_torch[:3],</code></pre></td></tr>
</table>

		max_length=50, # this is set to 50 due to resource constraint, using a GPU, you can increase it to the length of your choice num_beams=5, early_stopping=True,) generated_outputs_lora = [] for text in pipeline_iterator: generated_outputs_lora.append(text[0]["generated_text"])
load_summarize_chain	This code snippet uses LangChain library for loading and using a summarization chain with a specific language model and chain type. This chain type will be applied to web data to print a resulting summary.	from langchain.chains.summarize import load_summarize_chain chain = load_summarize_chain(llm=mixtral_llm, chain_type="stuff", verbose=False) response = chain.invoke(web_data) print(response['output_text'])n
TextClassifier	Represents a simple text classifier that uses an embedding layer, a hidden linear layer with a ReLU avtivation, and an output linear layer. The constructor takes the following arguments: num_class: The number of classes to classify. freeze: Whether to freeze the embedding layer.	from torch import nn class TextClassifier(nn.Module): def __init__(self, num_classes,freeze=False): super(TextClassifier, self).__init__() self.embedding = nn.Embedding.from_pretrained(glove_embedding.vectors.to(device),freeze=freeze) # An example of adding additional layers: A linear layer and a ReLU activation self.fc1 = nn.Linear(in_features=100, out_features=128) self.relu = nn.ReLU() # The output layer that gives the final probabilities for the classes

<table>
<tr><td></td><td></td><td>

```
        self.fc2 = nn.Linear(in_features=128,
out_features=num_classes)

    def forward(self, x):
        # Pass the input through the embedding layer
        x = self.embedding(x)
        # Here you can use a simple mean pooling

        x = torch.mean(x, dim=1)
        # Pass the pooled embeddings through the
additional layers
        x = self.fc1(x)
        x = self.relu(x)
        return self.fc2(x)
```

</td></tr>
<tr><td>Train the model</td><td>This code snippet outlines the function to train a machine learning model using PyTorch. This function trains the model over a specified number of epochs, tracks them, and evaluates the performance on the data set.</td><td>

```
def train_model(model, optimizer, criterion,
train_dataloader, valid_dataloader, epochs=100,
model_name="my_modeldrop"):
    cum_loss_list = []
    acc_epoch = []
    best_acc = 0
    file_name = model_name

    for epoch in tqdm(range(1, epochs + 1)):
        model.train()
        cum_loss = 0
        for _, (label, text) in
enumerate(train_dataloader):
            optimizer.zero_grad()
            predicted_label = model(text)
            loss = criterion(predicted_label, label)
            loss.backward()

torch.nn.utils.clip_grad_norm_(model.parameters(), 0.1)
```

</td></tr>
</table>

		<pre> optimizer.step() cum_loss += loss.item() #print("Loss:", cum_loss) cum_loss_list.append(cum_loss) acc_val = evaluate(valid_dataloader, model, device) acc_epoch.append(acc_val) if acc_val > best_acc: best_acc = acc_val print(f"New best accuracy: {acc_val:.4f}") #torch.save(model.state_dict(), f"{model_name}.pth") #save_list_to_file(cum_loss_list, f"{model_name}_loss.pkl") #save_list_to_file(acc_epoch, f"{model_name}_acc.pkl")</pre>
def plot_matrix_and_subspace(F)	The code snippet is useful for understanding the vectors in the 3D space.	<pre>def plot_matrix_and_subspace(F): assert F.shape[0] == 3, "Matrix F must have rows equal to 3 for 3D visualization." ax = plt.figure().add_subplot(projection='3d') # Plot each column vector of F as a point and line from the origin for i in range(F.shape[1]): ax.quiver(0, 0, 0, F[0, i], F[1, i], F[2, i], color='blue', arrow_length_ratio=0.1, label=f'Column {i+1}') if F.shape[1] == 2:</pre>

		# Calculate the normal to the plane spanned by the columns of F if they are exactly two normal_vector = np.cross(F[:, 0], F[:, 1]) # Plot the plane xx, yy = np.meshgrid(np.linspace(-3, 3, 10), np.linspace(-3, 3, 10)) zz = (-normal_vector[0] * xx - normal_vector[1] * yy) / normal_vector[2] if normal_vector[2] != 0 else 0 ax.plot_surface(xx, yy, zz, alpha=0.5, color='green', label='Spanned Plane') # Set plot limits and labels ax.set_xlim([-3, 3]) ax.set_ylim([-3, 3]) ax.set_zlim([-3, 3]) ax.set_xlabel('x_{1}') ax.set_ylabel('x_{2}') ax.set_zlabel('x_{3}') #ax.legend() plt.show()
nn.Parameter	The provided code is useful for defining the parameters of the 'LoRALayer' module during the training. The 'LoRALayer' has been used as an intermediate layer in a simple neural network.	class LoRALayer(torch.nn.Module): def __init__(self, in_dim, out_dim, rank, alpha): super().__init__() std_dev = 1 / torch.sqrt(torch.tensor(rank).float()) self.A = torch.nn.Parameter(torch.randn(in_dim, rank) * std_dev) self.B = torch.nn.Parameter(torch.zeros(rank, out_dim)) self.alpha = alpha

		`` def forward(self, x): x = self.alpha * (x @ self.A @ self.B) return x ```
LinearWithLoRA class	This code snippet defines the custom neural network layer called 'LoRALayer' using PyTorch. It uses 'nn.Parameter' to create learnable parameters for optimizing the training process.	``` class LinearWithLoRA(torch.nn.Module): def __init__(self, linear, rank, alpha): super().__init__() self.linear = linear.to(device) self.lora = LoRALayer(linear.in_features, linear.out_features, rank, alpha).to(device) def forward(self, x): return self.linear(x) + self.lora(x) ```
Applying LoRA	To fine-tune with LoRA, first, load a pretrained TextClassifier model with LoRA (while freezing its layers), load its pretrained state	``` from urllib.request import urlopen import io model_lora=TextClassifier(num_classes=4,freeze=False) ```

	from a file, and then disable gradient updates for all its parameters to prevent further training. Here, you will load a model that was pretrained on the AG NEWS data set, which is a data set that has 4 classes. Note that when you initialize this model, you set num_classes to 4. Moreover, the pretrained AG_News model was trained with the embedding layer unfrozen. Hence, you will initialize the model with freeze=False. Although you are initializing the model with layers unfrozen and the wrong number of classes for your task, you will make modifications to the model later that correct this.	model_lora.to(device) urlopened = urlopen('https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/uGC04Pom651hQs1XrZ0NsQ/my-model-freeze-false.pth') stream = io.BytesIO(urlopened.read()) state_dict = torch.load(stream, map_location=device) model_lora.load_state_dict(state_dict) # Here, you freeze all layers: for parm in model_lora.parameters(): parm.requires_grad=False model_lora
Select rank and alpha	The given code spinet evaluates the performance of a text classification model varying configurations of ‘LoRALayer’. It assesses the combination of rank and alpha hyperparameters, trains the model, and records the accuracy of each configuration.	ranks = [1, 2, 5, 10] alphas = [0.1, 0.5, 1.0, 2.0, 5.0] results=[] accuracy_old=0 # Loop over each combination of 'r' and 'alpha' for r in ranks: for alpha in alphas: print(f"Testing with rank = {r} and alpha = {alpha}") model_name=f"model_lora_rank{r}_alpha{alpha}_AGtoIBDM_final_adam_" model_lora=TextClassifier(num_classes=4,freeze=False)

<table>
<tr><td></td><td></td><td>

```
        model_lora.to(device)

        urlopened = urlopen('https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/uGC04Pom651hQs1XrZ0NsQ/my-model-freeze-false.pth')

        stream = io.BytesIO(urlopened.read())
        state_dict = torch.load(stream, map_location=device)
        model_lora.load_state_dict(state_dict)

        for parm in model_lora.parameters():
            parm.requires_grad=False

        model_lora.fc2=nn.Linear(in_features=128, out_features=2, bias=True)

model_lora.fc1=LinearWithLoRA(model_lora.fc1,rank=r, alpha=alpha )
        optimizer = torch.optim.Adam(model_lora.parameters(), lr=LR)

        scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.1)

        model_lora.to(device)

        train_model(model_lora, optimizer, criterion, train_dataloader, valid_dataloader, epochs=300, model_name=model_name)
```

</td></tr>
</table>

		`accuracy=evaluate(valid_dataloader , model_lora, device)` `result = {` `'rank': r,` `'alpha': alpha,` `'accuracy':accuracy` `}` `# Append the dictionary to the results list` `results.append(result)` `if accuracy>accuracy_old:` `print(f"Testing with rank = {r} and alpha = {alpha}")` `print(f"accuracy: {accuracy} accuracy_old: {accuracy_old}" )` `accuracy_old=accuracy` `torch.save(model.state_dict(), f"{model_name}.pth")` `save_list_to_file(cum_loss_list, f"{model_name}_loss.pkl")` `save_list_to_file(acc_epoch, f"{model_name}_acc.pkl")`
model_lora model	Sets up the training components for the model, defining a learning rate of 1, using cross-entropy loss as the criterion, optimizing with stochastic gradient descent (SGD), and scheduling the learning rate to decay by a factor of 0.1 at each epoch.	`LR=1` `criterion = torch.nn.CrossEntropyLoss()` `optimizer = torch.optim.SGD(model_lora.parameters(), lr=LR)` `scheduler = torch.optim.lr_scheduler.StepLR(optimizer, 1.0, gamma=0.1)`

<table>
<tr><td>load_dataset</td><td>The data set is loaded using the load_dataset function from the data sets library, specifically loading the "train" split.</td><td><pre>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)</pre></td></tr>
<tr><td>build_dataset</td><td>Incorporates the necessary steps to build a data set object for use as an input to PPOTrainer.</td><td><pre>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_max_text_length=8,tokenizer=tokenizer):
    """
    Build dataset for training. This builds the dataset
from `load_dataset`, one should
    customize this function to train the model on its
own dataset.

    Args:
        dataset_name (`str`):
            The name of the dataset to be loaded.

    Returns:
        dataloader (`torch.utils.data.DataLoader`):</pre></td></tr>
</table>

		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
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, "pad_token_id": tokenizer.eos_token_id} 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=True) return generated_text_ ```
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	``` def train_model(model,tr_dataloader): # Create a progress bar to track the training progress progress_bar = tqdm(range(num_training_steps)) ```

<table>
<tr>
<td></td>
<td>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.</td>
<td><pre><code>   # 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()</code></pre></td>
</tr>
</table>

<table>
<tr>
<td>

evaluate_model function

</td>
<td>

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.

</td>
<td>

```
def evaluate_model(model, evl_dataloader):
    # Create an instance of the Accuracy metric for multiclass classification with 5 classes
    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())
```

</td>
</tr>
</table>

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.	<pre>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 controls the maximum number of tokens in the generated output GenParams.MIN_NEW_TOKENS: default_params["min_new_tokens"], # this controls the minimum number of tokens in the generated output GenParams.TEMPERATURE: default_params["temperature"], # this randomness or creativity of the model's responses 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"</pre>

		`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`
class_names	This code snippet maps numerical labels to their corresponding textual descriptions to classify tasks. This code helps in machine learning to interpret the output model, where the model's predictions are numerical and should be presented in a more human-readable format.	`class_names = {0: "negative", 1: "positive"}` `class_names`
DistilBERT tokenizer	This code snippet uses 'AutoTokenizer' for preprocessing text data for DistilBERT, a lighter version of BERT. It tokenizes input text into a format suitable for model processing by converting words into token IDs, handling special tokens, padding, and truncating sequences as needed.	`tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")`
Tokenize input IDs	This code snippet tokenizes text data and inspects the resulting token IDs, attention masks, and token type IDs for further processing the natural language processing (NLP) tasks.	`my_tokens=tokenizer(imdb['train'][0]['text'])` `# Print the tokenized input IDs` `print("Input IDs:", my_tokens['input_ids'])`

		`# Print the attention mask` `print("Attention Mask:", my_tokens['attention_mask'])` `# If token_type_ids is present, print it` `if 'token_type_ids' in my_tokens:` `print("Token Type IDs:",` `my_tokens['token_type_ids'])`
Preprocessing function tokenizer	This code snippet explains how to use a tokenizer for preprocessing text data from the IMDB data set. The tokenizer is applied to review the training data set and convert text into tokenized input IDs, an attention mask, and token type IDs.	`def preprocess_function(examples):` `return tokenizer(examples["text"], padding=True,` `truncation=True, max_length=512)` `small_tokenized_train =` `small_train_dataset.map(preprocess_function,` `batched=True)` `small_tokenized_test =` `small_test_dataset.map(preprocess_function,` `batched=True)` `medium_tokenized_train =` `medium_train_dataset.map(preprocess_function,` `batched=True)` `medium_tokenized_test =` `medium_test_dataset.map(preprocess_function,` `batched=True)`
compute_metrics funcion	Evaluates model performance using accuracy.	`def compute_metrics(eval_pred):` `load_accuracy = load_metric("accuracy",` `trust_remote_code=True)` `logits, labels = eval_pred` `predictions = np.argmax(logits, axis=-1)` `accuracy =` `load_accuracy.compute(predictions=predictions,` `references=labels)["accuracy"]`

		`return {"accuracy": accuracy}`
Configure BitsAndBytes	Defines the quantization parameters.	`config_bnb = BitsAndBytesConfig(` `    load_in_4bit=True, # quantize the model to 4-bits when you load it` `    bnb_4bit_quant_type="nf4", # use a special 4-bit data type for weights initialized from a normal distribution` `    bnb_4bit_use_double_quant=True, # nested quantization scheme to quantize the already quantized weights` `    bnb_4bit_compute_dtype=torch.bfloat16, # use bfloat16 for faster computation` `    llm_int8_skip_modules=["classifier", "pre_classifier"] #  Don't convert the "classifier" and "pre_classifier" layers to 8-bit` `)`
id2label	Maps IDs to text labels for the two classes in this problem.	`id2label = {0: "NEGATIVE", 1: "POSITIVE"}`
label2id	Swaps the keys and the values to map the text labels to the IDs.	`label2id = dict((v,k) for k,v in id2label.items())`
model_qlora	This code snippet initializes a tokenizer using text data from the IMDB data set, creates a model called model_qlora for sequence classification using DistilBERT, and configures with id2label and label2id mappings. This code provides two output labels, including quantization configuration using config_bnb settings.	`model_qlora = AutoModelForSequenceClassification.from_pretrained("distilbert-base-uncased",` `id2label=id2label,` `label2id=label2id,` `num_labels=2,` `quantization_config=config_bnb`

		``)```
training_args	This code snippet initializes training arguments to train a model. It specifies the output directory for results, sets the number of training epochs to 10 and the learning rate to 2e-5, and defines the batch size for training and evaluation. This code also specifies the assessment strategies for each epoch.	```training_args = TrainingArguments(output_dir="./results_qlora", num_train_epochs=10, per_device_train_batch_size=16, per_device_eval_batch_size=64, learning_rate=2e-5, evaluation_strategy="epoch", weight_decay=0.01)```
text_to_emb	Designed to convert a list of text strings into their corresponding embeddings using a pre-defined tokenizer.	```def text_to_emb(list_of_text,max_input=512): data_token_index = tokenizer.batch_encode_plus(list_of_text, add_special_tokens=True,padding=True,truncation=True,max_length=max_input) question_embeddings=aggregate_embeddings(data_token_index['input_ids'], data_token_index['attention_mask']) return question_embeddings```
model_name_or_path	This code snippet defines the model name to 'gpt2' and initializes the token and model using the GPT-2 model. In this code, add special tokens for padding by keeping the maximum sequence length to 1024.	```# Define the model name or path model_name_or_path = "gpt2" # Initialize tokenizer and model tokenizer = GPT2Tokenizer.from_pretrained(model_name_or_path, use_fast=True) model = GPT2ForSequenceClassification.from_pretrained(model_name_or_path, num_labels=1) # Add special tokens if necessary```

		`` tokenizer.pad_token = tokenizer.eos_token model.config.pad_token_id = model.config.eos_token_id # Define the maximum length max_length = 1024 ```