Message Copilot

Sign in

do data need to have this format: template = "Instruction:\n{instruction}\n\nResponse:\n{response}"

Yes, using a template like "Instruction:\n{instruction}\n\nResponse:\n{response}" is a common approach when fine-tuning language models, including Gemma AI. This format helps in creating clear and structured input-output pairs, which are crucial for training the model effectively.

Here’s an example of how you might structure your data:

plaintext

Instruction:

How do I bake a chocolate cake?

Response:

To bake a chocolate cake, you'll need flour, sugar, cocoa powder, baking soda, baking powder, eggs, milk, oil, and vanilla extract. Mix the dry ingredients together, then add the wet ingredients. Pour the batter into a greased baking pan and bake at 350°F for about 30 minutes.

By using this format, you can ensure that the model learns to provide accurate and relevant responses to given instructions.

Does this help clarify how to format your data for fine-tuning? If you need more details or examples, feel free to ask!

do you need to use the words Instruction and Response exactly as shown

No, you don't need to use the words "Instruction" and "Response" exactly as shown, but it's important to maintain a clear and consistent format for your data. You can choose different labels or structures as long as they clearly distinguish between the prompt and the expected output. For example:

plaintext

Prompt:

How do I bake a chocolate cake?

Answer:

To bake a chocolate cake, you'll need flour, sugar, cocoa powder, baking soda, baking powder, eggs, milk, oil, and vanilla extract. Mix the dry ingredients together, then add the wet ingredients. Pour the batter into a greased baking pan and bake at 350°F for about 30 minutes.

or

plaintext

Question:

What are the steps to bake a chocolate cake?

Reply:

To bake a chocolate cake, you'll need flour, sugar, cocoa powder, baking soda, baking powder, eggs, milk, oil, and vanilla extract. Mix the dry ingredients together, then add the wet ingredients. Pour the batter into a greased baking pan and bake at 350°F for about 30 minutes.

The key is to ensure that the format is consistent throughout your dataset so that the model can learn to identify and respond to the different parts appropriately.

Does that clarify things for you? If you have more specific requirements or examples you'd like to explore, let me know!

can you directly use json format

Yes, you can directly use JSON format for preparing your dataset for fine-tuning a language model. Here’s an example of how you might structure your JSON file:

**JSON Format Example**

json

[

{

"instruction": "How do I bake a chocolate cake?",

"response": "To bake a chocolate cake, you'll need flour, sugar, cocoa powder, baking soda, baking powder, eggs, milk, oil, and vanilla extract. Mix the dry ingredients together, then add the wet ingredients. Pour the batter into a greased baking pan and bake at 350°F for about 30 minutes."

},

{

"instruction": "What is the capital of France?",

"response": "The capital of France is Paris."

}

]

**Loading JSON Data**

You can then load this JSON data into your fine-tuning process using Python. Here’s a simple example to illustrate:

python

import json

# Load the JSON file

with open("dataset.json", "r") as file:

data = json.load(file)

# Example: Accessing the first item

print(data[0]["instruction"])

print(data[0]["response"])

**Using the Data for Fine-Tuning**

You can preprocess and format this data according to your model’s requirements. For example, you might tokenize the instructions and responses and then feed them into your model for fine-tuning.

Would you like more detailed steps on how to preprocess and use this JSON data for fine-tuning your model?

do you need further preprocessing?

Yes, further preprocessing is often necessary to prepare your dataset for fine-tuning a language model effectively. Here are some common preprocessing steps:

**1. Tokenization**

* **Convert Text to Tokens**: Use a tokenizer that fits your model (e.g., BERT, GPT) to convert text into tokens.
* Example:

python

from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from\_pretrained("model\_name")

data["tokenized\_instruction"] = data["instruction"].apply(lambda x: tokenizer.encode(x, add\_special\_tokens=True))

data["tokenized\_response"] = data["response"].apply(lambda x: tokenizer.encode(x, add\_special\_tokens=True))

**2. Padding/Truncation**

* **Ensure Uniform Length**: Adjust tokenized sequences to ensure they are of a uniform length by padding or truncating them.
* Example:

python

max\_length = 128

data["padded\_instruction"] = data["tokenized\_instruction"].apply(lambda x: x + [0]\*(max\_length-len(x)) if len(x) < max\_length else x[:max\_length])

data["padded\_response"] = data["tokenized\_response"].apply(lambda x: x + [0]\*(max\_length-len(x)) if len(x) < max\_length else x[:max\_length])

**3. Formatting for Model**

* **Create Model-Compatible Format**: Convert your preprocessed data into a format compatible with your model training framework (e.g., PyTorch, TensorFlow).
* Example:

python

import torch

input\_ids = torch.tensor(data["padded\_instruction"].tolist())

attention\_mask = torch.tensor([[1]\*len(seq) for seq in data["tokenized\_instruction"].tolist()])

labels = torch.tensor(data["padded\_response"].tolist())

**4. Creating Dataset Object**

* **Custom Dataset Class**: Create a custom dataset class to use with your model's training framework.
* Example:

python

from torch.utils.data import Dataset, DataLoader

class CustomDataset(Dataset):

def \_\_init\_\_(self, input\_ids, attention\_mask, labels):

self.input\_ids = input\_ids

self.attention\_mask = attention\_mask

self.labels = labels

def \_\_len\_\_(self):

return len(self.input\_ids)

def \_\_getitem\_\_(self, idx):

return {

'input\_ids': self.input\_ids[idx],

'attention\_mask': self.attention\_mask[idx],

'labels': self.labels[idx]

}

dataset = CustomDataset(input\_ids, attention\_mask, labels)

dataloader = DataLoader(dataset, batch\_size=8, shuffle=True)

These preprocessing steps ensure that your data is in the right format and ready for fine-tuning, improving the performance and accuracy of your model.

Would you like more details on any specific preprocessing step or guidance on a different part of the fine-tuning process?

python

Sure! Here's a more detailed Python example for preprocessing, tokenizing, and preparing your dataset for fine-tuning a language model:

**Full Python Code for Preprocessing and Data Preparation**

python

import json

from transformers import AutoTokenizer

import torch

from torch.utils.data import Dataset, DataLoader

# Load your JSON data

with open("databricks-dolly-15k.jsonl", "r") as file:

data = [json.loads(line) for line in file]

# Filter examples and create a dataframe

filtered\_data = [item for item in data if not item["context"]]

print(f"Filtered data size: {len(filtered\_data)}")

# Initialize tokenizer

tokenizer = AutoTokenizer.from\_pretrained("gpt-2")

# Define max length for padding/truncation

max\_length = 128

# Tokenize the dataset

def tokenize\_function(example):

instruction = example["instruction"]

response = example["response"]

prompt = f"Instruction:\n{instruction}\n\nResponse:\n{response}"

return tokenizer(prompt, padding="max\_length", truncation=True, max\_length=max\_length)

# Apply tokenization

tokenized\_data = [tokenize\_function(item) for item in filtered\_data]

# Prepare tensors

input\_ids = torch.tensor([item['input\_ids'] for item in tokenized\_data])

attention\_mask = torch.tensor([item['attention\_mask'] for item in tokenized\_data])

# Create custom dataset class

class CustomDataset(Dataset):

def \_\_init\_\_(self, input\_ids, attention\_mask):

self.input\_ids = input\_ids

self.attention\_mask = attention\_mask

def \_\_len\_\_(self):

return len(self.input\_ids)

def \_\_getitem\_\_(self, idx):

return {

'input\_ids': self.input\_ids[idx],

'attention\_mask': self.attention\_mask[idx]

}

# Create dataset and dataloader

dataset = CustomDataset(input\_ids, attention\_mask)

dataloader = DataLoader(dataset, batch\_size=8, shuffle=True)

# Example: Print a batch of data

for batch in dataloader:

print(batch)

break

**Explanation:**

1. **Loading Data**: The JSONL file is loaded and filtered to remove entries with context.
2. **Tokenization**: The dataset is tokenized using the AutoTokenizer from the Hugging Face Transformers library.
3. **Padding/Truncation**: The tokenized data is padded or truncated to a uniform length.
4. **Tensor Preparation**: The tokenized data is converted to PyTorch tensors.
5. **Dataset Class**: A custom dataset class is created to handle the data.
6. **DataLoader**: A DataLoader is used to batch and shuffle the data for training.

This setup should help you prepare your data for fine-tuning a language model. Let me know if you need further assistance!

translate visual basic code to python

Sure, I'd be happy to help you with that! Please provide the Visual Basic code you'd like to translate to Python, and I'll get started on it for you.

Function GetalInWoorden(Getal As Long) As String Dim Eenheden As Variant Dim Tientallen As Variant Dim Honderdtallen As String Dim Duizendtallen As String Dim Miljoentallen As String ' Array met eenheden Eenheden = Array("", "een", "twee", "drie", "vier", "vijf", "zes", "zeven", "acht", "negen") ' Array met tientallen Tientallen = Array("", "", "twintig", "dertig", "veertig", "vijftig", "zestig", "zeventig", "tachtig", "negentig") ' Controleer miljoentallen If Getal >= 1000000 Then Miljoentallen = GetalNaarHonderden(Getal \ 1000000) & "miljoen " Getal = Getal Mod 1000000 End If ' Controleer duizendtallen If Getal >= 1000 Then Duizendtallen = GetalNaarHonderden(Getal \ 1000) & "duizend " Getal = Getal Mod 1000 End If ' Combineer alle strings GetalInWoorden = Miljoentallen & Duizendtallen & GetalNaarHonderden(Getal) End Function Function GetalNaarHonderden(num As Long) As String Dim Eenheden As Variant Dim Tientallen As Variant Dim res As String ' Array met eenheden Eenheden = Array("", "een", "twee", "drie", "vier", "vijf", "zes", "zeven", "acht", "negen") ' Array met tientallen Tientallen = Array("", "", "twintig", "dertig", "veertig", "vijftig", "zestig", "zeventig", "tachtig", "negentig") ' Controleer honderdtallen If num >= 100 Then res = Eenheden(num \ 100) & "honderd" num = num Mod 100 End If ' Speciale gevallen voor de tientallen If num > 10 And num < 20 Then Select Case num Case 11: res = res & "elf" Case 12: res = res & "twaalf" Case 13: res = res & "dertien" Case 14: res = res & "veertien" Case 15: res = res & "vijftien" Case 16: res = res & "zestien" Case 17: res = res & "zeventien" Case 18: res = res & "achttien" Case 19: res = res & "negentien" End Select ElseIf num >= 20 Then If num Mod 10 <> 0 Then res = res & Eenheden(num Mod 10) & "en" End If res = res & Tientallen(num \ 10) Else res = res & Eenheden(num) End If GetalNaarHonderden = res End Function

Here is the translated Visual Basic code into Python:

**Original Visual Basic Code**

vb

Function GetalInWoorden(Getal As Long) As String

Dim Eenheden As Variant

Dim Tientallen As Variant

Dim Honderdtallen As String

Dim Duizendtallen As String

Dim Miljoentallen As String

' Array met eenheden

Eenheden = Array("", "een", "twee", "drie", "vier", "vijf", "zes", "zeven", "acht", "negen")

' Array met tientallen

Tientallen = Array("", "", "twintig", "dertig", "veertig", "vijftig", "zestig", "zeventig", "tachtig", "negentig")

' Controleer miljoentallen

If Getal >= 1000000 Then

Miljoentallen = GetalNaarHonderden(Getal \ 1000000) & "miljoen "

Getal = Getal Mod 1000000

End If

' Controleer duizendtallen

If Getal >= 1000 Then

Duizendtallen = GetalNaarHonderden(Getal \ 1000) & "duizend "

Getal = Getal Mod 1000

End If

' Combineer alle strings

GetalInWoorden = Miljoentallen & Duizendtallen & GetalNaarHonderden(Getal)

End Function

Function GetalNaarHonderden(num As Long) As String

Dim Eenheden As Variant

Dim Tientallen As Variant

Dim res As String

' Array met eenheden

Eenheden = Array("", "een", "twee", "drie", "vier", "vijf", "zes", "zeven", "acht", "negen")

' Array met tientallen

Tientallen = Array("", "", "twintig", "dertig", "veertig", "vijftig", "zestig", "zeventig", "tachtig", "negentig")

' Controleer honderdtallen

If num >= 100 Then

res = Eenheden(num \ 100) & "honderd"

num = num Mod 100

End If

' Speciale gevallen voor de tientallen

If num > 10 And num < 20 Then

Select Case num

Case 11: res = res & "elf"

Case 12: res = res & "twaalf"

Case 13: res = res & "dertien"

Case 14: res = res & "veertien"

Case 15: res = res & "vijftien"

Case 16: res = res & "zestien"

Case 17: res = res & "zeventien"

Case 18: res = res & "achttien"

Case 19: res = res & "negentien"

End Select

ElseIf num >= 20 Then

If num Mod 10 <> 0 Then

res = res & Eenheden(num Mod 10) & "en"

End If

res = res & Tientallen(num \ 10)

Else

res = res & Eenheden(num)

End If

GetalNaarHonderden = res

End Function

**Translated Python Code**

python

def getal\_in\_woorden(getal):

eenheden = ["", "een", "twee", "drie", "vier", "vijf", "zes", "zeven", "acht", "negen"]

tientallen = ["", "", "twintig", "dertig", "veertig", "vijftig", "zestig", "zeventig", "tachtig", "negentig"]

miljoentallen = ""

duizendtallen = ""

if getal >= 1000000:

miljoentallen = getal\_naar\_honderden(getal // 1000000) + "miljoen "

getal = getal % 1000000

if getal >= 1000:

duizendtallen = getal\_naar\_honderden(getal // 1000) + "duizend "

getal = getal % 1000

return miljoentallen + duizendtallen + getal\_naar\_honderden(getal)

def getal\_naar\_honderden(num):

eenheden = ["", "een", "twee", "drie", "vier", "vijf", "zes", "zeven", "acht", "negen"]

tientallen = ["", "", "twintig", "dertig", "veertig", "vijftig", "zestig", "zeventig", "tachtig", "negentig"]

res = ""

if num >= 100:

res = eenheden[num // 100] + "honderd"

num = num % 100

if 10 < num < 20:

special\_cases = {

11: "elf", 12: "twaalf", 13: "dertien", 14: "veertien", 15: "vijftien",

16: "zestien", 17: "zeventien", 18: "achttien", 19: "negentien"

}

res += special\_cases[num]

elif num >= 20:

if num % 10 != 0:

res += eenheden[num % 10] + "en"

res += tientallen[num // 10]

else:

res += eenheden[num]

return res

# Example usage

getal = 1234567

print(getal\_in\_woorden(getal))

This Python code should provide the same functionality as the original Visual Basic code, converting a number into words in Dutch. Let me know if you have any questions or need further assistance!

python append line to file

Appending a line to a file in Python is straightforward. Here's how you can do it:

**Step-by-Step Guide**

1. **Open the File**: Use the open function with the append mode ('a').
2. **Write the Line**: Use the write method to add the line to the file.
3. **Close the File**: Always close the file to ensure the data is written properly.

**Example Code**

python

# Specify the file name

file\_name = "example.txt"

# The line you want to append

new\_line = "This is the new line that will be added.\n"

# Open the file in append mode

with open(file\_name, 'a') as file:

file.write(new\_line)

print("Line appended successfully!")

**Explanation:**

1. **Opening the File**: The open(file\_name, 'a') opens the file in append mode. If the file does not exist, it will be created.
2. **Writing to the File**: The file.write(new\_line) appends the specified line to the file.
3. **Closing the File**: Using the with statement ensures the file is properly closed after writing.

This will append the specified line to the end of the file. Let me know if you have any further questions or need more examples!

python empty file

Emptying a file in Python is simple. You can do this by opening the file in write mode ('w'), which will truncate the file to zero length, effectively emptying it. Here’s how you can do it:

**Example Code**

python

# Specify the file name

file\_name = "example.txt"

# Open the file in write mode to empty it

with open(file\_name, 'w') as file:

pass

print("File emptied successfully!")

**Explanation:**

1. **Opening the File**: The open(file\_name, 'w') opens the file in write mode. This mode clears the file's contents immediately.
2. **Using** pass: The pass statement is used as a placeholder since you don’t need to perform any actions after opening the file.

This will empty the contents of the specified file. Let me know if you need any further assistance!

kaggle check if training is still running

To check if your Kaggle training is still running, you can follow these steps:

1. **Check the Notebook Interface**: Open your Kaggle notebook and look at the progress indicator. If the cell is still showing as "Running" or "In Progress," your training is likely still ongoing.
2. **Monitor Resource Usage**: If you have access to the system's resource usage (like CPU or GPU usage), you can check if there's still activity. High resource usage typically indicates that the training process is still active.
3. **Check for Checkpoints**: If your training script saves checkpoints, you can check if new checkpoints have been created recently.
4. **Refresh the Page**: Sometimes, simply refreshing the Kaggle notebook page can update the status and show if the training has completed or is still running.

If you're still unsure, you can also try restarting the kernel or the entire Kaggle session to see if the training process restarts or continues from where it left off.

Does this help clarify things for you? If you have any more questions or need further assistance, feel free to ask!

few-shot prompting, retrieval-augmented generation

Sure! Let's break down both concepts:

**Few-Shot Prompting**

**Few-shot prompting** is a technique where a model is given a few examples of a task within the prompt to guide its response. This helps the model understand the task better and generate more accurate outputs1. For instance, if you want the model to classify sentiments of movie reviews, you might provide a few examples of positive and negative reviews in the prompt.

**Retrieval-Augmented Generation (RAG)**

**Retrieval-Augmented Generation (RAG)** is a method that enhances the capabilities of generative AI models by incorporating external information. When a query is made, the model retrieves relevant documents or data from an external source and uses this information to generate a more accurate and contextually relevant response3. This is particularly useful for providing up-to-date or domain-specific information.

Would you like to see an example of how these techniques can be applied together?

yes, for Gemma 2

Great! Let's see how we can apply few-shot prompting and retrieval-augmented generation (RAG) using Gemma 2.

**Few-Shot Prompting with Gemma 2**

To use few-shot prompting with Gemma 2, you provide the model with a few examples within the prompt to guide its response. Here’s an example:

python

# Example prompt for sentiment analysis

prompt = """

Instruction: Classify the sentiment of the movie review as positive or negative.

Examples:

1. Review: "I loved the movie! The plot was engaging and the characters were well-developed."

Sentiment: Positive

2. Review: "The movie was boring and predictable. I wouldn't recommend it."

Sentiment: Negative

3. Review: "The movie had some good moments, but overall it was just okay."

Sentiment: Neutral

Classify the sentiment of the following review:

Review: "The movie was fantastic! I was on the edge of my seat the whole time."

"""

# Use Gemma 2 to classify sentiment

response = gemma2\_model.generate(prompt)

print(response)

**Retrieval-Augmented Generation (RAG) with Gemma 2**

To implement RAG, you combine an information retrieval component with Gemma 2 to provide contextually relevant responses. Here’s a simplified example:

1. **Retrieve Relevant Information**: Use an information retrieval system to fetch relevant documents or data.
2. **Generate Response**: Use Gemma 2 to generate a response based on the retrieved information.

python

# Example of RAG for answering a question

def retrieve\_information(query):

# Placeholder function to simulate information retrieval

# In practice, you would use a search engine or database

documents = [

{"title": "Document 1", "content": "Information about cats"},

{"title": "Document 2", "content": "Information about dogs"},

{"title": "Document 3", "content": "Information about birds"}

]

# Filter documents based on relevance to the query

relevant\_documents = [doc for doc in documents if query.lower() in doc["content"].lower()]

return relevant\_documents

query = "What are the benefits of having a pet?"

relevant\_docs = retrieve\_information(query)

# Generate response using Gemma 2

prompt = f"""

Instruction: Summarize the benefits of having a pet based on the following documents:

{relevant\_docs[0]["content"]}

{relevant\_docs[1]["content"]}

{relevant\_docs[2]["content"]}

Summary:

"""

response = gemma2\_model.generate(prompt)

print(response)

**Explanation:**

* **Few-Shot Prompting**: The prompt includes a few examples to guide the model in classifying the sentiment of a movie review.
* **Retrieval-Augmented Generation**: The retrieve\_information function simulates retrieving relevant documents, and the prompt is constructed to include these documents for generating a summary.

Would you like more details on any specific part of this process?