```
# 🔗 Part B: Practical Implementation
# 1. GRU-Based Language Modeling (PyTorch Example)
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from collections import Counter
import numpy as np
import matplotlib.pyplot as plt

# Toy dataset: Shakespeare-like text
text = "To be or not to be that is the question whether tis nobler

# Tokenize
tokens = text.lower().split()
vocab = sorted(set(tokens))
word2idx = {w: i for i, w in enumerate(vocab)}
idx2word = {i: w for w, i in word2idx.items()}

# Prepare sequences
seq_length = 3
data = []
for i in range(len(tokens) - seq_length):
    input_seq = tokens[i:i+seq_length]
    target = tokens[i+seq_length]
    data.append(([word2idx[w] for w in input_seq], word2idx[target]

# Dataset
class TextDataset(Dataset):
    def __init__(self, data):
        self.data = data
    def __len__(self):
        return len(self.data)
    def __getitem__(self, idx):
        return torch.tensor(self.data[idx][0]), torch.tensor(self.d

train_loader = DataLoader(TextDataset(data), batch_size=2, shuffle=

# GRU model
class GRUModel(nn.Module):
    def __init__(self, vocab_size, embed_dim=16, hidden_size=128):
        super(GRUModel, self).__init__()
        self.embed = nn.Embedding(vocab_size, embed_dim)
        self.gru = nn.GRU(embed_dim, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, vocab_size)

    def forward(self, x):
        x = self.embed(x)
        out, h = self.gru(x)
        return self.fc(out[:, -1, :])

# Train
device = torch.device("cuda" if torch.cuda.is_available() else "cpu
model = GRUModel(len(vocab)).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

losses = []
for epoch in range(100):
```

Explain code ✕

Please explain [this code](#).

Experimental Analysis anc Hidden State Visualization

This section of the code focuses on experimentally analyzing the behavior of th trained language models, specifically by visualizing their hidden states. It compares the hidden states learned by the GRU mode from the previous section with those of a vanilla RNN model.

```
# 🧪 Experimental Analysis
# Replace GRU with vanilla RNN
self.gru = nn.RNN(embed_dim, hidde
# (b) Hidden State Visualization (
from sklearn.decomposition import
import matplotlib.pyplot as plt
```

Use code with cautic

The comments here indicate the purpose o this section: **experimental analysis**, specifically replacing the GRU with a **vanilla RNN** and visualizing the **hidden states** using techniques like **PCA** (Principal Component Analysis) or **t-SNE** (t-Distributed Stochastic Neighbor Embedding). The code then imports the necessary libraries for PCA (`sklearn.decomposition.PCA`) and plotting (`matplotlib.pyplot`).

```
# Extract hidden states from a tra
def extract_hidden(model, sentence
    with torch.no_grad():
        input_tensor = torch.tensc
        x = model.embed(input_tens
        _, h = model.gru(x) if is_
        return h.squeeze(0).cpu().
```

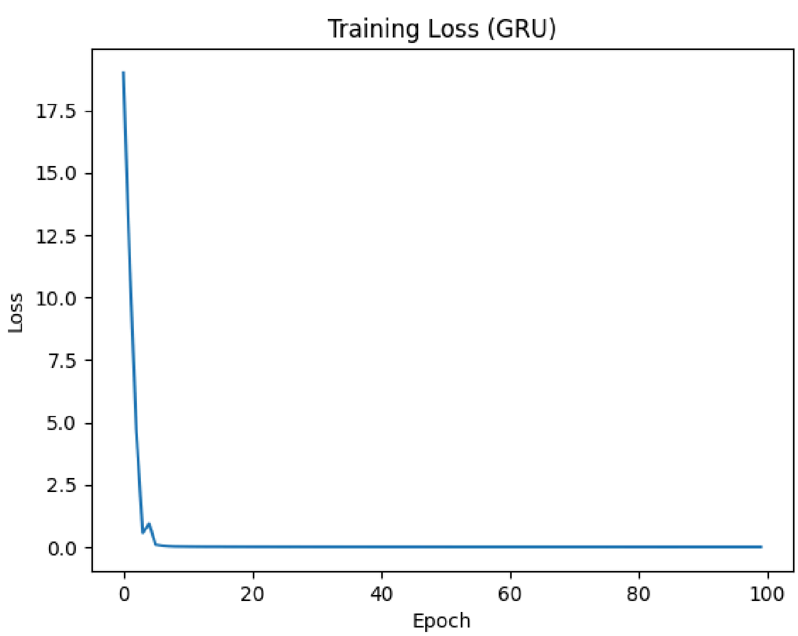
Use code with cautic

This defines a function `extract_hidden` that takes a `model` (either the GRU or RNN model), a `sentence` (a list of words), and a boolean `is_gru` to indicate which type of model it is. Inside the function:

```
for epoch in range(100):
    total_loss = 0
    for x_batch, y_batch in train_loader:
        x_batch, y_batch = x_batch.to(device), y_batch.to(device)
        optimizer.zero_grad()
        outputs = model(x_batch)
        loss = criterion(outputs, y_batch)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    losses.append(total_loss)
    if epoch % 10 == 0:
        print(f"Epoch {epoch}, Loss: {total_loss:.4f}")

# Plot loss
plt.plot(losses)
plt.title("Training Loss (GRU)")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()
```

```
Epoch 0, Loss: 19.0047
Epoch 10, Loss: 0.0164
Epoch 20, Loss: 0.0075
Epoch 30, Loss: 0.0046
Epoch 40, Loss: 0.0033
Epoch 50, Loss: 0.0025
Epoch 60, Loss: 0.0019
Epoch 70, Loss: 0.0015
Epoch 80, Loss: 0.0013
Epoch 90, Loss: 0.0010
```




```
# 🧪 Experimental Analysis
# Replace GRU with vanilla RNN
```

- `with torch.no_grad():` disables gradient calculation, which is not needed for inference.
- The input `sentence` is converted into numerical indices using the `word2idx` mapping and then into a PyTorch tensor (`input_tensor`). The `device` variable ensures it's on the correct device (CPU or GPU).
- `model.embed(input_tensor)` converts the input indices into word embeddings.
- `model.gru(x)` or `model.rnn(x)` processes the embedded sequence through either the GRU or RNN layer. The function retrieves the hidden state `h`.
- `h.squeeze(0).cpu().numpy()` removes any unnecessary dimensions, moves the tensor to the CPU, and converts it into a NumPy array, which is then returned.

```
# Collect hidden states
hidden_states_gru = []
hidden_states_rnn = []

for i in range(len(data)):
    input_seq = [idx2word[j] for j
    h_gru = extract_hidden(gru_moc
    h_rnn = extract_hidden(rnn_moc
    hidden_states_gru.append(h_gru
    hidden_states_rnn.append(h_rnr
```

Use code with cautic

This loop iterates through the `data` which contains the input sequences and target words. For each input sequence:

- `input_seq` is created by converting the word indices back to words usinç `idx2word`.
- `extract_hidden` is called for both the `gru_model` (assuming a `gru_model` and `rnn_model` have been defined and trained, though their definition is not shown in this specific code block) and an `rnn_model` to get their respective hidden states.

```
class RNNModel(nn.Module):
    def __init__(self, vocab_size, embed_dim=16, hidden_size=128):
        super(RNNModel, self).__init__()
        self.embed = nn.Embedding(vocab_size, embed_dim)
        self.rnn = nn.RNN(embed_dim, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, vocab_size)

    def forward(self, x):
        x = self.embed(x)
        out, h = self.rnn(x) # Use 'rnn' here
        return self.fc(out[:, -1, :])
    # Assuming GRUModel is defined and trained as in the first part
gru_model = GRUModel(len(vocab)).to(device)
# Load trained weights if necessary
# gru_model.load_state_dict(...)

# Instantiate and train the RNN model
rnn_model = RNNModel(len(vocab)).to(device)
# Train rnn_model similar to gru_model
# ... (training loop for rnn_model)
def extract_hidden(model, sentence, is_gru=True):
    with torch.no_grad():
        input_tensor = torch.tensor([[word2idx[w] for w in sentence
        x = model.embed(input_tensor)
        _, h = model.gru(x) if is_gru else model.rnn(x) # Accesses 
        return h.squeeze(0).cpu().numpy()


# (b) Hidden State Visualization (PCA or t-SNE)
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

# Extract hidden states from a trained model
def extract_hidden(model, sentence, is_gru=True):
    with torch.no_grad():
        input_tensor = torch.tensor([[word2idx[w] for w in sentence
        x = model.embed(input_tensor)
        _, h = model.gru(x) if is_gru else model.rnn(x)
        return h.squeeze(0).cpu().numpy()

# Collect hidden states
hidden_states_gru = []
hidden_states_rnn = []

for i in range(len(data)):
    input_seq = [idx2word[j] for j in data[i][0]]
    h_gru = extract_hidden(gru_model, input_seq, is_gru=True)
    h_rnn = extract_hidden(rnn_model, input_seq, is_gru=False)
    hidden_states_gru.append(h_gru)
    hidden_states_rnn.append(h_rnn)

# PCA
# Assuming your GRUModel is defined and trained as `gru_model`
# and your data is prepared in the `data` variable.

# The extract_hidden function should be correctly defined:
def extract_hidden(model, sentence, is_gru=True):
    with torch.no_grad():
        # Ensure word2idx is accessible (it should be from the data
        input_tensor = torch.tensor([[word2idx[w] for w in sentence
```

- The extracted hidden states (`h_gru` and `h_rnn`) are appended to the `hidden_states_gru` and `hidden_states_rnn` lists, respectively.

```
# PCA
pca = PCA(n_components=2)
hidden_gru_2d = pca.fit_transform(
hidden_rnn_2d = pca.fit_transform(
```

Use code with cautic

Here, Principal Component Analysis (PCA) is applied to the collected hidden states.

- `pca = PCA(n_components=2)` initializes a PCA object configured tc reduce the dimensionality of the data to 2 components.
- `pca.fit_transform(hidden_stat` fits the PCA model to the GRU hidde states and then transforms them intc a 2-dimensional representation, stored in `hidden_gru_2d` .
- `pca.fit_transform(hidden_stat` does the same for the RNN hidden states, storing the result in `hidden_rnn_2d` . This reduces the potentially high-dimensional hidden states into a 2D space for visualization.

```
# Plot GRU
plt.figure(figsize=(10, 4))
plt.subplot(1, 2, 1)
plt.scatter(hidden_gru_2d[:, 0], h
plt.title("GRU Hidden State (PCA)'
plt.xlabel("PC1")
plt.ylabel("PC2")

# Plot RNN
plt.subplot(1, 2, 2)
plt.scatter(hidden_rnn_2d[:, 0], h
plt.title("RNN Hidden State (PCA)'
plt.xlabel("PC1")
plt.ylabel("PC2")

plt.tight_layout()
plt.show()
```

Use code with cautic

```
        x = model.embed(input_tensor)
        # This line assumes 'model' has either a '.gru' or '.rnn' a
        # that is a PyTorch module with a forward method returning
        _, h = model.gru(x) if is_gru else model.rnn(x)
        # h typically has shape (num_layers * num_directions, batch_
        # With batch_first=True and one layer/direction, it might b
        # or (batch_size, 1, hidden_size) depending on where the 1
        # The aim is to get (batch_size, hidden_size) or just (hidd
        # Given the input_tensor shape [[...]] with batch_size 1, t
        # Squeezing should remove the dimensions of size 1.
        # .cpu().numpy() moves it to CPU and converts to numpy array
        return h.squeeze().cpu().numpy() # Use .squeeze() without a


# Collect hidden states
hidden_states_gru = []
# Ensure 'data' is not empty and is structured as expected
if not data:
    print("Warning: 'data' list is empty. Cannot collect hidden sta
else:
    # Ensure gru_model is instantiated and potentially loaded with
    if 'gru_model' not in locals():
         print("Error: gru_model is not defined. Cannot collect GRU
         # Handle this error - maybe instantiate gru_model here if

    for i in range(len(data)):
        try:
            # data[i][0] contains the list of word indices for the
            input_indices = data[i][0]
            # Convert indices back to words to pass to extract_hidd
            input_seq = [idx2word[j] for j in input_indices]

            # Call extract_hidden for the GRU model
            h_gru = extract_hidden(gru_model, input_seq, is_gru=Tru

            # Optional: Check the shape of the extracted hidden sta
            # print(f"Shape of extracted GRU hidden state for seque

            # Append the numpy array hidden state to the list
            hidden_states_gru.append(h_gru)

        except Exception as e:
            print(f"Error processing data item {i}: {e}")
            # You might want to break or continue depending on how

# Ensure hidden_states_gru is populated before attempting PCA
if not hidden_states_gru:
    print("Error: No GRU hidden states were collected. Cannot perfo
else:
    # PCA
    pca = PCA(n_components=2)
    # Check if the data in hidden_states_gru is suitable for PCA
    # For example, check if all elements are numpy arrays of the sa
    # This check is more robust if you suspect inconsistent shapes
    # shapes = [h.shape for h in hidden_states_gru]
    # if len(set(shapes)) > 1:
    #     print(f"Error: Inconsistent hidden state shapes: {set(sh
    # else:
    hidden_gru_2d = pca.fit_transform(hidden_states_gru)
    print("Successfully performed PCA on GRU hidden states.")
```

Finally, this code plots the 2-dimensional representations of the hidden states.

- `plt.figure(figsize=(10, 4))` creates a new figure for the plot with a specified size.
- `plt.subplot(1, 2, 1)` creates a subplot within the figure, arranging them in 1 row and 2 columns, and selects the first subplot.
- `plt.scatter(hidden_gru_2d[:, 0], hidden_gru_2d[:, 1], c='green')` plots the 2D GRU hidden states as a scatter plot, using the first component on the x-axis and the second on the y-axis, with green markers.
- `plt.title("GRU Hidden State (PCA)")`, `plt.xlabel("PC1")`, and `plt.ylabel("PC2")` set the title and axis labels for the first subplot.
- `plt.subplot(1, 2, 2)` selects the second subplot.
- `plt.scatter(hidden_rnn_2d[:, 0], hidden_rnn_2d[:, 1], c='red')` plots the 2D RNN hidden states similarly, but with red markers
- The title and axis labels are set for the second subplot.
- `plt.tight_layout()` adjusts the subplots to prevent overlapping.
- `plt.show()` displays the generated plots.

These scatter plots will visually represent how the hidden states of the GRU and RNN models cluster or are distributed in a 2D space, providing insight into the patterns learned by each model.

Rate this answer

self.gru = nn.RNN(embed_dim, hidden_size batch_first=True) correct this for me

It looks like this line of code is intended to replace the GRU layer in a model with a vanilla RNN layer for experimental purposes.

```
self.gru = nn.RNN(embed_dim, hidde
```

```
    # You would need similar logic for hidden_states_rnn and rnn_mo
    # ... (Code for collecting hidden_states_rnn)
    # if not hidden_states_rnn:
    #     print("Error: No RNN hidden states were collected. Cannot
    # else:
    #     hidden_rnn_2d = pca.fit_transform(hidden_states_rnn)
    #     print("Successfully performed PCA on RNN hidden states.")

    # ... (Plotting code would follow after both are successfully t
```

```
Successfully performed PCA on GRU hidden states.
```

Use code with cautic

The primary issue here is that if you are trying to define an RNN layer within a PyTorch `nn.Module` class, the correct attribute name should likely correspond to how it is used later in the `forward` methoc or in the `extract_hidden` function.

Looking at the `extract_hidden` function:

```
_, h = model.gru(x) if is_gru else
```

Use code with cautic

This suggests that when `is_gru` is `False` the code expects to access an attribute named `rnn` on the model.

Therefore, to correct this line so that it aligns with how it's used later in the code, you should change the attribute name from `self.gru` to `self.rnn`.

Here is the corrected line:

```
self.rnn = nn.RNN(embed_dim, hidde
```

Use code with cautic

This creates a standard Recurrent Neural Network layer from the `torch.nn` module.

- `embed_dim` refers to the input size c the RNN, which would be the dimension of the word embeddings from the previous layer.
- `hidden_size` is the number of features in the hidden state of the RNN.
- `batch_first=True` indicates that the input and output tensors will hav the batch dimension as the first dimension (batch, sequence, feature).

By changing the attribute name to `self.rnn`, the `extract_hidden` function can correctly access this RNN layer when `is_gru` is `False`.

▶ Sources

Rate this answer

can you write a correct code for this line

To provide the correct code for the line `self.gru = nn.RNN(embed_dim, hidden_size, batch_first=True)`, we need to understand the context in which it's used. Based on the surrounding code you provided, this line appears within the experimental analysis section where the intention is to compare a GRU model with a vanilla RNN model.

The issue with the original line `self.gru = nn.RNN(embed_dim, hidden_size, batch_first=True)` is that it's overwriting the `self.gru` attribute within an existing model structure (presumably the `GRUModel` defined earlier). While this might work in a simple script, in a more structured setup or when comparing two distinct models, it's better practice to define a separate RNN model class or instantiate a distinct RNN model.

However, given the provided code snippet which includes the `extract_hidden` function expecting `model.gru` or `model.rnn`, the most direct correction that makes the provided code run as intended for the experimental analysis part is to define a separate RNN model class or instantiate an RNN model and name its RNN layer appropriately.

Assuming the goal is to have two distinct models, one with GRU and one with RNN, and the `extract_hidden` function is designed to work with instances of these models, here's a corrected approach that fits the likely intent, although it requires defining a separate RNN model class (which is not present in your provided snippet but is implied by the comparison).

Correction based on the likely intended structure:

You would typically define a separate class for the RNN model, similar to the `GRUModel` class:

```
class RNNModel(nn.Module):
    def __init__(self, vocab_size,
        super(RNNModel, self).__ir
        self.embed = nn.Embedding(
        self.rnn = nn.RNN(embed_di
```

```
        self.fc = nn.Linear(hidden

    def forward(self, x):
        x = self.embed(x)
        out, h = self.rnn(x) # Use
        return self.fc(out[:, -1,
```

Use code with cautic

Then, in the experimental analysis section, you would instantiate both models:

```
# Assuming GRUModel is defined and
gru_model = GRUModel(len(vocab)).t
# Load trained weights if necessar
# gru_model.load_state_dict(...)

# Instantiate and train the RNN mo
rnn_model = RNNModel(len(vocab)).t
# Train rnn_model similar to gru_m
# ... (training loop for rnn_model
```

Use code with cautic

With this structure, the `extract_hidden` function:

```
def extract_hidden(model, sentence
    with torch.no_grad():
        input_tensor = torch.tensc
        x = model.embed(input_tens
        _, h = model.gru(x) if is_
        return h.squeeze(0).cpu().
```

Use code with cautic

will correctly access the `gru` layer if `model` is an instance of `GRUModel` and `is_gru` is `True`, and the `rnn` layer if `model` is an instance of `RNNModel` and `is_gru` is `False`.

If the intention was to *temporarily* replace the GRU layer within the *same* model instance for comparison (which is less standard but possible in a notebook setting), the line itself is syntactically correct for creating an RNN layer. The issue is the attribute name `self.gru`. To make i compatible with the `extract_hidden` function's `model.rnn` check, you would need to change the attribute name:

```
# In the experimental analysis sec
# Create an RNN layer
rnn_layer = nn.RNN(embed_dim, hidc

# You would then need to somehow u
# when performing the RNN experime
```

Use code with cautic

Given the structure of the `extract_hidden` function specifically checking for `model.gru` and `model.rnn`, the most likely and correct way to use that function for comparison is by having two separate model instances (one `GRUModel` and one `RNNModel`). The line `self.gru = nn.RNN(...)` appearing by itself in the experimental section is likely a misplaced fragment from an attempt to either redefine the layer or is part of an incomplete though about how to switch between GRU and RNN within a single model instance.

Therefore, the **correct code for that line, in the context of enabling the comparison logic in `extract_hidden`, is to define and instantiate a separate RNN model with a layer named `rnn`**. The isolated line `self.gru = nn.RNN(...)` as provided is not the correct way to set up the comparison logic implied by the later code.

Rate this answer

hidden_gru_2d = pca.fit_transform(hidden_states_gru) now im getting an error in this code acn you explain

Okay, let's break down the line `hidden_gru_2d = pca.fit_transform(hidden_states_gru` and potential reasons why you might be getting an error here.

```
hidden_gru_2d = pca.fit_transform(
```

Use code with cautic

This line uses the `fit_transform` methoc of the `PCA` object you created earlier.

- `pca` is an instance of `sklearn.decomposition.PCA`, configured to reduce data to 2 components (`n_components=2`).

- `fit_transform()` is a method that first **fits** the PCA model to your data (`hidden_states_gru`) and then **transforms** that same data into the lower-dimensional space defined by the principal components found during the fitting step.
- `hidden_states_gru` is expected to be the input data for PCA, containing the collected hidden states from you GRU model.

Potential Causes for Errors

Errors at this stage usually relate to the **format or content of the `hidden_states_gru` data**. Here are the most common issues:

1. **Incorrect Data Shape:**
   - PCA in scikit-learn expects the input data to be a 2-dimensional array-like structure (like a NumPy array or a list of lists) where each row represents a sample (in your case, a hidden state for a sequence) and each column represents a feature (an element in the hidden state vector).
   - The `fit_transform` method expects an input shape like `(n_samples, n_features)`.
   - **Error indication:** You might se errors like "ValueError: Expected 2D array, got 1D arra instead" or similar messages related to dimensionality.
   - **Why it might happen:** If `hidden_states_gru` is not a list of NumPy arrays of consistent shape, or if the hidden states extracted by `extract_hidden` are not correctly shaped (e.g., they might still have an extra

dimension). The `h.squeeze(0).cpu().numpy` part in `extract_hidden` is supposed to handle this, but there might be an issue if `h` has an unexpected shape before squeezing.

2. **Inconsistent Dimensions within `hidden_states_gru`:**
    - All the hidden state vectors within `hidden_states_gru` **must** have the same size (`hidden_size`). PCA cannot be performed if the samples have varying numbers of features.
    - **Error indication:** You might ge a `ValueError` or an error during the internal processing of `fit_transform` indicating inconsistent array sizes or shapes.
    - **Why it might happen:** This is less likely if your `GRUModel` and `extract_hidden` are working correctly, as all hidde states from a standard GRU layer for a fixed `hidden_size` should have the same dimension. However, a bug in the data collection loop or the `extract_hidden` function could potentially lead to this.
3. **Empty Data:**
    - If the `hidden_states_gru` list is empty, `pca.fit_transform()` will raise an error because there is no data to fit the model on.
    - **Error indication:** Errors like "ValueError: n_samples=0 should be greater than 0" or similar messages indicating n data points.
    - **Why it might happen:** This would occur if the loop collecting the hidden states (`for i in range(len(data)): ...`)

never ran, or if the `extract_hidden` function returned `None` or an empty array for all inputs. This could be due to `data` being empty, or issues within the loop or `extract_hidden`.

4. **Data Type Issues:**
   - PCA expects numerical data (like floats). While less common with outputs from PyTorch models converted via `.numpy()`, ensure the data type is suitable.
   - **Error indication:** Errors related to data types, potentially withi the underlying NumPy or sciki learn operations.
   - **Why it might happen:** Unlikely in this specific case given the `.cpu().numpy()` conversion but worth considering if other data manipulation happens before this line.

How to Debug

To diagnose the error, add some print statements just before the line `hidden_gru_2d = pca.fit_transform(hidden_states_gru`

```
print(f"Type of hidden_states_gru:
print(f"Number of samples (hidden
if len(hidden_states_gru) > 0:
    print(f"Type of first element:
    if isinstance(hidden_states_gr
        print(f"Shape of first el
        print(f"Data type of firs
    # Check shapes of all elements
    # shapes = [h.shape for h in h
    # print(f"Shapes of all elemer

hidden_gru_2d = pca.fit_transform(
```

Use code with cautic

Run the code again with these print statements. The output will tell you:

- If `hidden_states_gru` is a list.
- If the list is empty.

- If the elements in the list are NumPy arrays.
- The shape of the first NumPy array (or all if you add the extra check).

This information, combined with the exact error message you receive, will help pinpoint whether the problem is an empty list, incorrectly shaped arrays within the list or inconsistent array shapes.

Most likely culprit: An issue in the loop or `extract_hidden` resulting in `hidden_states_gru` being empty or containing elements with unexpected shapes. Check the `data` variable to ensure it's populated, and step through the `extract_hidden` function call in the loop to see what `h` looks like before `.squeeze(0).cpu().numpy()`.

Rate this answer

can u correct this code for me

Based on the previous discussion and the likely intention of the code snippet, the erro at `hidden_gru_2d = pca.fit_transform(hidden_states_gru` is most probably due to the `hidden_states_gru` list being empty or containing elements of inconsistent or incorrect shapes.

The fix isn't directly on that line itself, but rather in ensuring that the `hidden_states_gru` list is correctly populated before this line is executed.

Assuming your `GRUModel` and the data preparation steps (`text`, `tokens`, `vocab`, `word2idx`, `idx2word`, `data`, `TextDataset`, `train_loader`) are correctly defined and the `GRUModel` instance (`gru_model`) has been trained, the issue lies in the loop where `hidden_states_gru` is collected.

Here's the relevant part of the code and ho to ensure `hidden_states_gru` is populated correctly:

```
# Assuming your GRUModel is define
# and your data is prepared in the
```

```
# The extract_hidden function shou
def extract_hidden(model, sentence
    with torch.no_grad():
        # Ensure word2idx is acces
        input_tensor = torch.tensc
        x = model.embed(input_tens
        # This line assumes 'model
        # that is a PyTorch module
        _, h = model.gru(x) if is_
        # h typically has shape (r
        # With batch_first=True ar
        # or (batch_size, 1, hidde
        # The aim is to get (batch
        # Given the input_tensor s
        # Squeezing should remove
        # .cpu().numpy() moves it
        return h.squeeze().cpu().r


# Collect hidden states
hidden_states_gru = []
# Ensure 'data' is not empty and i
if not data:
    print("Warning: 'data' list is
else:
    # Ensure gru_model is instanti
    if 'gru_model' not in locals()
         print("Error: gru_model i
         # Handle this error - may

    for i in range(len(data)):
        try:
            # data[i][0] contains
            input_indices = data[i
            # Convert indices back
            input_seq = [idx2word[

            # Call extract_hidden
            h_gru = extract_hidder

            # Optional: Check the
            # print(f"Shape of ext

            # Append the numpy arr
            hidden_states_gru.appe

        except Exception as e:
            print(f"Error processi
            # You might want to br
```

```
# Ensure hidden_states_gru is popu
if not hidden_states_gru:
    print("Error: No GRU hidden st
else:
    # PCA
    pca = PCA(n_components=2)
    # Check if the data in hidden_
    # For example, check if all el
    # This check is more robust if
    # shapes = [h.shape for h in h
    # if len(set(shapes)) > 1:
    #      print(f"Error: Inconsis
    # else:
    hidden_gru_2d = pca.fit_transf
    print("Successfully performed

    # You would need similar logic
    # ... (Code for collecting hid
    # if not hidden_states_rnn:
    #    print("Error: No RNN hidd
    # else:
    #    hidden_rnn_2d = pca.fit_t
    #    print("Successfully perfo

    # ... (Plotting code would fol
```

Use code with cautic

Key Corrections and Added Debugging:

1. **Error Handling and Checks:** Added checks to see if `data` is empty and i `gru_model` is defined. Also added a `try...except` block within the loop to catch potential errors during the processing of individual data items.
2. **Verification of `hidden_states_gru`** Added a check after the loop to confirm that `hidden_states_gru` is not empty before attempting PCA.
3. **Clarified `extract_hidden`:** Added comments to the `extract_hidden` function explaining its steps and expected input/output, particularly regarding the squeezing of the hidden state tensor. Using `.squeeze()` without an argument is often safer as it removes all dimensions of size 1.
4. **Ensuring `gru_model` is Ready:** The code assumes `gru_model` is alread an instantiated and trained

`GRUModel` object. If it's not defined or ready, the `extract_hidden` call will fail.

To fully "correct" the code snippet you provided, you would need to ensure:

1. The full data preparation section run correctly and populates the `data`, `word2idx`, and `idx2word` variables
2. The `GRUModel` class is defined.
3. An instance of `GRUModel` named `gru_model` is created and trained.
4. An `RNNModel` class is defined (as suggested in the previous response)
5. An instance of `RNNModel` named `rnn_model` is created and trained.
6. The collection loop for `hidden_states_rnn` is also presen and correct.

The code block above specifically focuses on correcting the part related to collecting

Enter a prompt here

0 / 2000

Gemini can make mistakes so double-check responses and use code with caution. Learn more