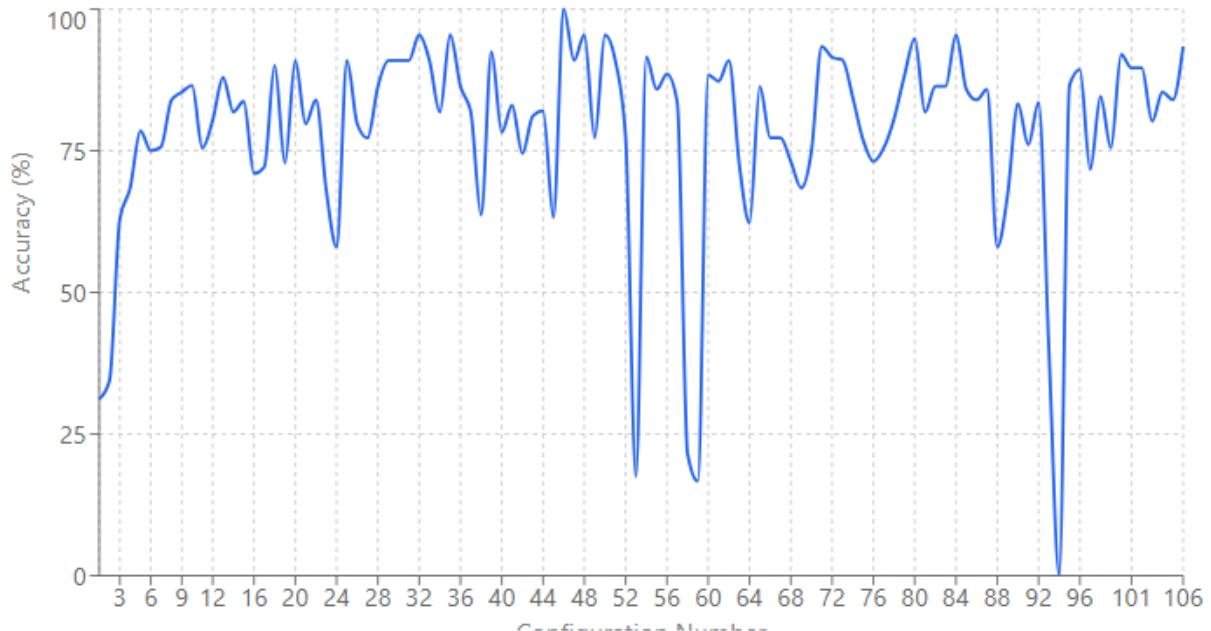


A.I. Disclaimer: Work for this assignment was completed with the aid of artificial intelligence tools and comprehensive documentation of the names of, input provided to, and output obtained from, these tools is included as part of my assignment submission.

Most of the research was assisted by AI as that was my personal means of researching and iterating tests for this project. I used my personal knowledge from my experiences and this NLP class to guide my AI prompts and strategies to test on the neural network.

My favorite part about using AI for this project was visualizing my results quite easily, the method described later in this document.



Before I was able to run the code, I had library install issues. Gensim does not work with the latest version of Numpy so I used Claude to figure out how to reinstall the correct version.

```

Pasted content
6.34 KB = 199 lines + Formating may be inconsistent from source
se.py[5]
  2 from warnings import warn
  4 import array
--> 5 from scipy._lib import visibleDeprecationWarning
  7 from _sputils import (asmatrix, check_reshape_kwarg, check_shape,
  8     get_sum_type, isdense, isscalarlike,
  9     matrix, validateaxis),
11 from _matrix import spmatrix

File <ipython-input-1-99718>
1 import numpy as np
2 from typing import (
3     Optional,
4     Union,
5     TypeChecking,
6     TypeVar,
7     )
8
9 import numpy as np
--> 10 from scipy._lib.array_api import array_namespace
11     asarray: Type[Exception]
12     asarray: Type[Warning]
13     asarray: Type[Warning]

File <ipython-input-2-99718>
14 import numpy as np
15 from scipy._lib import array_api_compat
--> 16 from scipy._lib.array_api_compat import (
17     array_namespace,
18     asarray,
19     size,
20     numpy as np_compat,
21     )
22 asarray = ["array_namespace", "asarray", "size"]
23 # to enable array API and strict array-like input validation
24 # to enable array API and strict array-like input validation

File <ipython-input-3-99718>
25 api_compatible_init_py[1]
--> 26 1 from numpy import *
27 2 # from numpy import * doesn't overwrite these builtin names
28 3 from numpy import abs, max, min, round

File <ipython-input-4-99718>
29 y[1]
30
31     raise RuntimeError("no module named 'numpy._char'")

ModuleNotFoundError: No module named 'numpy._char'

```

Reply to Claude...

I reinstalled, but then I got these results:

```

ValueError
Traceback (most recent call last) Cell In[4], line 15
  12 import json
  14 # import Word2Vec
gensim utilities --> 15 from gensim.models import KeyedVectors File
  M:\AppData\Local\ Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\localCache\local-packages\Python311\site-packages\gensim\init.py[1]:7 version = '4.3.3'
  9 import logging --> 11 from gensim import parsing, corpora, matutils, interfaces, models,
similarities, utils # noqa:F401 14 logger = logging.getLogger('gensim') 15 if not
logger.handlers: # To ensure reload() doesn't add another one File

```

ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This behaviour is the source of the following dependency conflicts.  
gensim 4.3.3 requires numpy<2.0,>=1.18.5, but you have numpy 2.1.2 which is incompatible.  
Successfully installed numpy-2.1.2

Ah, now I see the issue - there's a version conflict between NumPy and Gensim. The error shows that Gensim 4.3.3 requires NumPy version less than 2.0 (`numpy<2.0`), but you just installed NumPy 2.1.2.

Here's how to fix this:

1. First, uninstall both packages:

```

bash
Copy
pip uninstall numpy gensim

```

2. Then install the compatible versions in the correct order:

```

bash
Copy
pip install numpy==1.26.4
pip install gensim

```

This will ensure you have a compatible version of NumPy that works with Gensim. The reason for this conflict is that Gensim hasn't been updated yet to support NumPy 2.x, which introduced some breaking changes.

If you need to use both packages, you'll need to stick with NumPy 1.x for now until Gensim updates their compatibility. The error you're seeing (`numpy.dtype size changed, may indicate binary incompatibility`) is a common issue when there's a mismatch between the version of NumPy that a package was compiled against and the version that's currently installed.

Then, the neural network was running on my CPU so I had to fix that by installing the CUDA library, which I was easily guided by Claude.

The screenshot shows a conversational interface with a user question and several code snippets and instructions:

- User Question:** D I have a RTX 3070 and CUDA Version 12.6, so why did pytorch not recognize that I have a cuda device?
- Text:** This could be happening for a few reasons. Let's troubleshoot:
- Step 1:** 1. First, make sure you have the CUDA-enabled version of PyTorch installed. If you installed PyTorch using pip without specifying CUDA support, it might have installed the CPU-only version. You can check your PyTorch installation with:

```
python
import torch
print(torch.__version__)
print(torch.version.cuda)
```

- Step 2:** 2. To install the CUDA-enabled version of PyTorch, go to <https://pytorch.org/get-started/locally/> and get the correct installation command for your system. For Windows with CUDA 12.6, it should be something like:

```
bash
pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu1:
```

The next issue occurred when the labels were getting the wrong data type. The given code caused issues, but Claude identified a solution of coercing them to long types, which fixed the code on the second attempt.

Could you please help me figure out why I'm getting this error?

```
RuntimeError Traceback (most recent call last)
Cell In[24], line 4
  2 epochs = 5      # Trial and error: evaluate when validation loss stop decreasing
  consistently
  3 print_every = 100
----> 4 rnn_train(rnn_model, train_loader, epochs, print_every=print_every)

Cell In[22], line 29
  26 output = net(inputs)
  28 # Calculate the loss
---> 29 loss = rnn_criterion(output, labels)
  31 # Backpropagate
  32 loss.backward()

RuntimeError: "nll_loss_forward_reduce_cuda_kernel_2d_index" not implemented for 'Int'
```

This error occurs because of a mismatch in the data types between your model's output and the target labels when using CrossEntropyLoss. The error message "nll\_loss\_forward\_reduce\_cuda\_kernel\_2d\_index not implemented for 'Int'" suggests that the target labels are not in the correct format.

Here's how to fix it:

1. First, make sure your labels are of type Long (int64):

```
python
# Convert Labels to Long type
labels = labels.long()
```

2. Modify your training loop to ensure proper data types. Here's an example:

```
python
def rnn_train(net, train_loader, epochs, print_every=100):
    net.train()
    for e in range(epochs):
        running_loss = 0
        for i, (inputs, labels) in enumerate(train_loader):
            # Convert Labels to Long type
            labels = labels.long()
```

proj4

IPYNB

I tried multiple methods of making the labels long, but I keep getting the same issue

The error occurs because of a type mismatch between your labels and what CrossEntropyLoss expects. Let me help you fix this by making sure all the labels are properly converted to the right type. Here are a few ways to fix this:

1. Convert labels to long tensors during DataLoader creation:

```
python
# create Tensor datasets
train_data = TensorDataset(torch.from_numpy(train_x), torch.from_numpy(train_y).long())
valid_data = TensorDataset(torch.from_numpy(val_x), torch.from_numpy(val_y).long())
test_data = TensorDataset(torch.from_numpy(test_x), torch.from_numpy(test_y).long())
```

Once the code was working in its most basic form, I wanted to visualize it, so I asked Claude to see if there were any easy to use libraries, but I figured it would take more time than necessary for this assignment, and I ended up skipping these.

Great! I got it to run for the first time with 31% accuracy!

Basically, I have over 4000 speeches, labeled from canada, euro area, japan, sweden, united kingdom, and united states. The neural network identifies which area the speech is from, and does so initially with 31% accuracy.

How could I potentially visualize the data results?

Congratulations on getting the model running! There are several ways to visualize the results. Here are some common visualization approaches using matplotlib and seaborn:

1. Confusion Matrix Heatmap:

```
python
import seaborn as sns
import matplotlib.pyplot as plt

def plot_confusion_matrix(y_true, y_pred, labels):
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(10,8))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=labels,
                yticklabels=labels)
    plt.title('Confusion Matrix')
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.xticks(rotation=45)
    plt.yticks(rotation=45)
    plt.tight_layout()
    plt.show()

# Use it in your testing function:
def rnn_test(test_loader):
    true_list = []
    pred_list = []

# After testing, plot results:
labels = ['Canada', 'Euro Area', 'Japan', 'Sweden', 'UK', 'US']
plot_confusion_matrix(true_list, pred_list, labels)
```

2. Training History Plot:

```
python
def plot_training_history(train_losses, val_losses):
    plt.figure(figsize=(10,6))
    plt.plot(train_losses, label='Training Loss')
    plt.plot(val_losses, label='Validation Loss')
    plt.title('Training and Validation Loss Over Time')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)
    plt.show()

# Modify your training function to record losses:
def rnn_train(net, train_loader, epochs, print_every=100):
    train_losses = []
    val_losses = []
    # ... your training code ...
    # Record losses at each epoch
    train_losses.append(loss.item())
    val_losses.append(np.mean(val_losses))

# After training, plot:
plot_training_history(train_losses, val_losses)
```

### 3. Per-Class Accuracy Bar Chart:

python

```
def plot_class_accuracies(y_true, y_pred, labels):
    accuracies = []
    for i in range(len(labels)):
        mask = (y_true == i)
        class_acc = accuracy_score(y_true[mask], y_pred[mask])
        accuracies.append(class_acc)

    plt.figure(figsize=(10,6))
    plt.bar(labels, accuracies)
    plt.title('Accuracy by Region')
    plt.xlabel('Region')
    plt.ylabel('Accuracy')
    plt.xticks(rotation=45)
    plt.ylim(0, 1)
    for i, v in enumerate(accuracies):
        plt.text(i, v + 0.01, f'{v:.2%}', ha='center')
    plt.tight_layout()
    plt.show()
```

### 4. ROC Curves (for multi-class):

python

```
from sklearn.metrics import roc_curve, auc
from itertools import cycle

def plot_roc_curves(y_true, y_pred_proba, labels):
    # Compute ROC curve and ROC area for each class
    fpr = dict()
    tpr = dict()
    roc_auc = dict()

    # Convert to one-hot encoding
    y_test_bin = label_binarize(y_true, classes=range(len(labels)))

    plt.figure(figsize=(10,8))
    colors = cycle(['blue', 'red', 'green', 'yellow', 'purple', 'orange'])

    for i, color in zip(range(len(labels)), colors):
        fpr[i], tpr[i] = roc_curve(y_test_bin[:, i], y_pred_proba[:, i])
        roc_auc[i] = auc(fpr[i], tpr[i])

        plt.plot(fpr[i], tpr[i], color=color, lw=2,
                 label=f'ROC curve of {labels[i]} (area = {roc_auc[i]:.2f})')

    plt.plot([0, 1], [0, 1], 'k--', lw=2)
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('ROC Curves')
    plt.legend(loc="lower right")
    plt.show()

    # In your test function:
def rmn_test(test_loader):
    # ... your existing code ...
    y_pred_proba = torch.softmax(output, dim=1).cpu().numpy()
    plot_roc_curves(true_list, y_pred_proba, labels)
```

To use these visualizations, you'll need to modify your training and testing functions to collect the necessary data. You might want to create a wrapper class or function that handles all the visualizations together:

python

```
def visualize_results(true_labels, pred_labels, pred_proba=None,
                      train_history=None, labels=None):
    """Wrapper function to create all visualizations"""
    if labels is None:
        labels = ['Canada', 'Euro Area', 'Japan', 'Sweden', 'UK', 'US']

    # Create subplots
    plt.figure(figsize=(20, 10))

    # Confusion Matrix
    plt.subplot(2, 2, 1)
    plot_confusion_matrix(true_labels, pred_labels, labels)

    # Per-class accuracies
    plt.subplot(2, 2, 2)
    plot_class_accuracies(true_labels, pred_labels, labels)

    # ROC curves if probabilities are provided
    if pred_proba is not None:
        plt.subplot(2, 2, 3)
        plot_roc_curves(true_labels, pred_proba, labels)

    # Training history if provided
    if train_history is not None:
        plt.subplot(2, 2, 4)
        plot_training_history(train_history['train'], train_history['val'])

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

I wanted to try something unique to attempt to get a higher accuracy. I personally figured that TF-IDF would be helpful for this. I didn't specify TF-IDF initially because I wanted to see if Claude would either assume I meant TF-IDF or if it may have a different idea, so I treated the first few conversations here as brainstorming.

Would I be able to identify which vectors are more meaningful, and which label they most correspond with?

Yes! You can analyze the relationship between word embeddings and labels in several ways. Here are a couple of approaches:

1. Word Importance Analysis using attention weights or gradient-based methods:

```
python
def get_important_words(model, text, label, embeddings_table, top_k=10):
    # Convert text to tensor
    text_tensor = torch.LongTensor(text).unsqueeze(0).to(device)

    # Get word embeddings
    embedded = model.embedding(text_tensor)

    # Calculate gradients with respect to embeddings
    embedded.retain_grad()
    output = model(text_tensor)
    output[0, label].backward()

    # Get importance scores
    importance = torch.norm(embedded.grad[0], dim=1)

    # Get top k words
    top_indices = importance.argsort(descending=True)[:top_k]

    # Map indices back to words
    words = []
    scores = []
    for idx in top_indices:
        word_idx = text[idx]
        if word_idx != 0: # Skip padding
            word = list(embeddings_table.key_to_index.keys())[list(embeddings_table.key_to_index.values()).index(word_idx)]
            words.append(word)
            scores.append(importance[idx].item())

    return words, scores
```

## 2. Regional Word Analysis - find words most associated with each region:

```
python
def analyzeRegionalWords(model, train_loader, embeddings_table, top_k=20):
    region_words = {i: {} for i in range(6)} # Dictionary for each region

    for texts, labels in train_loader:
        for text, label in zip(texts, labels):
            words, scores = getImportantWords(model, text.tolist(), label.item(), embeddings_table)

            # Add words and scores to region dictionary
            for word, score in zip(words, scores):
                if word in region_words[label.item()]:
                    region_words[label.item()][word] += score
                else:
                    region_words[label.item()][word] = score

    # Get top words for each region
    region_names = ['Canada', 'Euro Area', 'Japan', 'Sweden', 'UK', 'US']
    for region_idx, region_name in enumerate(region_names):
        print(f"\nTop words for {region_name}:")
        sorted_words = sorted(region_words[region_idx].items(), key=lambda x: x[1], reverse=True)
        for word, score in sorted_words[:top_k]:
            print(f"{word}: {score:.4f}")
```

```

3. Visualization of Word Clusters by Region using dimensionality reduction:

Python

from sklearn.manifold import TSNE
import seaborn as sns

def visualize_word_clusters(model, train_loader, embeddings_table, n_words=100):
    # Collect most important words for each region
    region_words = {i: {} for i in range(6)}
    region_names = ['Canada', 'Euro Area', 'Japan', 'Sweden', 'UK', 'US']

    # Get important words and their embeddings
    words = []
    embeddings = []
    regions = []

    for region_idx, region_name in enumerate(region_names):
        # Get top words for this region
        region_top_words = list(region_words[region_idx].items())[:n_words]

        for word, _ in region_top_words:
            if word in embeddings_table:
                words.append(word)
                embeddings.append(embeddings_table[word])
                regions.append(region_name)

    # Convert embeddings to numpy array
    embeddings_array = np.array(embeddings)

    # Reduce dimensionality for visualization
    tsne = TSNE(n_components=2, random_state=42)
    embeddings_2d = tsne.fit_transform(embeddings_array)

    # Create DataFrame for plotting
    df = pd.DataFrame({
        'x': embeddings_2d[:, 0],
        'y': embeddings_2d[:, 1],
        'word': words,
        'region': regions
    })

```

```

# Plot
plt.figure(figsize=(12, 8))
sns.scatterplot(data=df, x='x', y='y', hue='region', alpha=0.6)

# Add word labels for some points
for idx, row in df.iterrows():
    if idx % 5 == 0: # Label every 5th word to avoid overcrowding
        plt.annotate(row['word'], (row['x'], row['y']))

plt.title('Word Embeddings Clustered by Region')
plt.show()

4. Vector Similarity Analysis:

Python

def analyze_vector_similarities(word, embeddings_table, region_words, top_k=10):
    """Find words most similar to a given word within regional contexts"""
    if word not in embeddings_table:
        return None

    word_vector = embeddings_table[word]

    similarities = {}
    for region, words in region_words.items():
        region_similarities = []
        for w in words:
            if w in embeddings_table:
                similarity = np.dot(word_vector, embeddings_table[w])
                region_similarities.append((w, similarity))

        # Get top k similar words for this region
        region_similarities.sort(key=lambda x: x[1], reverse=True)
        similarities[region] = region_similarities[:top_k]

    return similarities

```

**Example usage:**

```
python Copy
# Get important words for each region
analyzeRegionalWords(rnn_model, train_loader, embeddings_table)

# Visualize word clusters
visualizeWordClusters(rnn_model, train_loader, embeddings_table)

# Analyze specific words
word = "inflation"
similarities = analyzeVectorSimilarities(word, embeddings_table, region_words)
for region, similar_words in similarities.items():
    print(f"\nMost similar words to '{word}' in {region}:")
    for w, sim in similar_words:
        print(f'{w}: {sim:.4f}')
```

This analysis can reveal:

1. Which words are most important for classifying speeches from each region
2. How word usage differs between regions
3. Regional-specific terminology or concerns
4. Common themes across regions

You could also extend this to analyze:

- Temporal changes in word importance
- Economic terminology differences
- Policy-related vocabulary variations
- Regional-specific economic indicators

Would you like me to explain any of these approaches in more detail or help you implement a specific analysis?

Here, Claude started to use frequencies so it was picking up lots of very common words, so I needed to adjust it by explicitly using TF-IDF.

Q What is the simplest way to implement a small addition that would identify tokens that are key identifiers of each label?

Here's a simple approach to identify key tokens for each label by analyzing the frequency and uniqueness of words in each category:

```
python
def analyze_key_tokens(speeches, labels, top_n=20):
    # Create dictionaries to store word frequencies for each label
    label_word_freq = {
        0: {}, # Canada
        1: {}, # Euro area
        2: {}, # Japan
        3: {}, # Sweden
        4: {}, # UK
        5: {} # US
    }

    # Count word frequencies for each Label
    for speech, label in zip(speeches, labels):
        for word in speech:
            if word not in label_word_freq[label]:
                label_word_freq[label][word] = 1
            else:
                label_word_freq[label][word] += 1

    # Calculate distinctiveness of words for each Label
    label_names = ['Canada', 'Euro Area', 'Japan', 'Sweden', 'UK', 'US']

    for label, name in enumerate(label_names):
        print(f"\nTop {top_n} distinctive words for {name}:")

        # calculate word scores (frequency in this Label / frequency in all Labels)
        word_scores = {}
        for word in label_word_freq[label]:
            word_freq_this_label = label_word_freq[label][word]
            word_freq_total = sum(
                label_word_freq[other_label].get(word, 0)
                for other_label in label_word_freq
            )
            distinctiveness = word_freq_this_label / word_freq_total
            distinctiveness = word_freq_this_label / word_freq_total
            word_scores[word] = distinctiveness * word_freq_this_label

        # Get top words
        top_words = sorted(word_scores.items(), key=lambda x: x[1], reverse=True)[:top_n]

        # Print results
        for word, score in top_words:
            print(f"{word}: {score:.2f}")

    # Use it
    analyze_key_tokens(speeches, encoded_labels)
```

This will show you which words are most uniquely associated with each label. The score combines both:

1. How frequently the word appears in speeches from that region
2. How unique that word is to that region compared to others

Words with high scores are both common in that region's speeches and relatively uncommon in other regions' speeches, making them good identifiers for that label.

You could add this to your testing function or run it separately to analyze your dataset. Would you like me to help you integrate it with your existing code?

**D** Currently many of the top words are very frequent words, such as "the, to, of, and". Please use TF-IDF to no longer include these in the top words.

Here's a modified version using TF-IDF to remove common words and focus on more meaningful terms:

```
Python Copy
from sklearn.feature_extraction.text import TfidfVectorizer

def analyze_key_tokens_tfidf(speeches, labels, top_n=20):
    # Convert speeches to documents for TF-IDF
    documents = [' '.join(speech) for speech in speeches]

    # Create and fit TF-IDF vectorizer
    tfidf = TfidfVectorizer(stop_words='english')
    tfidf_matrix = tfidf.fit_transform(documents)

    # Get feature names (words)
    feature_names = tfidf.get_feature_names_out()

    # Create dictionaries to store word scores for each label
    label_word_scores = {
        0: {}, # Canada
        1: {}, # Euro area
        2: {}, # Japan
        3: {}, # Sweden
        4: {}, # UK
        5: {} # US
    }

    # Calculate average TF-IDF scores for each word per label
    for doc_idx, label in enumerate(labels):
        doc_scores = tfidf_matrix[doc_idx].toarray()[0]
        for word_idx, score in enumerate(doc_scores):
            word = feature_names[word_idx]
            if score > 0: # only consider words that appear in the document
                if word not in label_word_scores[label]:
                    label_word_scores[label][word] = []
                label_word_scores[label][word].append(score)
    
```

```
# Calculate average scores
for label in label_word_scores:
    for word in list(label_word_scores[label].keys()):
        scores = label_word_scores[label][word]
        label_word_scores[label][word] = sum(scores) / len(scores)

# Print top words for each Label
label_names = ['Canada', 'Euro Area', 'Japan', 'Sweden', 'UK', 'US']
for label, name in enumerate(label_names):
    print(f"\nTop {top_n} distinctive words for {name}:")

# Sort words by score
top_words = sorted(label_word_scores[label].items(),
                    key=lambda x: x[1],
                    reverse=True)[:top_n]

# Print results
for word, score in top_words:
    print(f"\t{word}: {score:.4f}")

# Use it
analyze_key_tokens_tfidf(speeches, encoded_labels)
```

This version:

1. Uses scikit-learn's TfidfVectorizer to calculate TF-IDF scores
2. Automatically removes common English stop words
3. Weights words based on their importance within documents and their uniqueness across all documents
4. Returns words that are truly distinctive for each region

You might want to customize the TfidfVectorizer further:

```
python Copy
# None customized version
tfidf = TfidfVectorizer(
    stop_words='english',
    min_df=5, # Ignore terms that appear in less than 5 documents
    max_df=0.5, # Ignore terms that appear in more than 50% of documents
    ngram_range=(1, 2), # Consider both unigrams and bigrams
    token_pattern=r'(\w|\b\w+\b)', # Match any word character
)
```

Or add custom stop words specific to your domain:

```
python Copy
def analyze_key_tokens_tfidf(speeches, labels, top_n=20):
    # Custom stop words for central bank speeches
    custom_stop_words = ['bank', 'central', 'monetary', 'policy', 'thank', 'ladies', 'gent']

    # Combine with English stop words
    stop_words = list(tfidf.get_stop_words('english')) + custom_stop_words

    tfidf = TfidfVectorizer(
        stop_words=stop_words,
        min_df=5,
        max_df=0.5,
        ngram_range=(1, 2)
    )

    # Rest of the function remains the same
    ...
```

This should give you much more meaningful results focusing on words that are truly distinctive for each region's central bank communications. Would you like me to help you customize it further for your specific needs?

Then with the TF-IDF, I needed to adjust the model's weights accordingly to the TF-IDF results:

💡 With those distinctive word frequencies, can I use them to adjust the model's results final results?

Yes! You can use the distinctive word frequencies to create a weighted prediction system that combines the model's output probabilities with the TF-IDF scores. Here's how you could implement this:

```
python Copy
```

```
class WeightedPredictor:
    def __init__(self, model, tfidf_weights, word_to_idx, alpha=0.3):
        self.model = model
        self.tfidf_weights = tfidf_weights # TF-IDF scores per word per Label
        self.word_to_idx = word_to_idx # Dictionary mapping words to indices
        self.alpha = alpha # weight for TF-IDF influence (0 to 1)

    def predict(self, text_input, labels=['Canada', 'Euro Area', 'Japan', 'Sweden', 'UK'],
               # Get model's prediction probabilities
               self.model.eval())
        with torch.no_grad():
            model_output = self.model(text_input)
            model_probs = torch.softmax(model_output, dim=1)

        # calculate TF-IDF weighted scores
        tfidf_scores = torch.zeros_like(model_probs)

        # For each word in the input text
        for idx in text_input[0]: # Assuming batch size 1
            if idx != 0: # Skip padding
                word = list(self.word_to_idx.keys())[list(self.word_to_idx.values()).index(
                    if word in self.tfidf_weights:
                        for label_idx in range(len(labels)):
                            tfidf_scores[0][label_idx] += self.tfidf_weights[label_idx].get(word))

        # Normalize TF-IDF scores
        tfidf_scores = torch.softmax(tfidf_scores, dim=1)

        # Combine predictions
        final_probs = (1 - self.alpha) * model_probs + self.alpha * tfidf_scores

    return final_probs

# Usage example:
def test_with_weights(model, test_loader, tfidf_weights, word_to_idx):
    weighted_predictor = WeightedPredictor(model, tfidf_weights, word_to_idx)
```

```
# usage example:
def test_with_weights(model, test_loader, tfidf_weights, word_to_idx):
    weighted_predictor = WeightedPredictor(model, tfidf_weights, word_to_idx)

    all_preds = []
    all_labels = []

    for inputs, labels in test_loader:
        inputs = inputs.to(device)
        weighted_probs = weighted_predictor.predict(inputs)
        preds = torch.argmax(weighted_probs, dim=1)

        all_preds.extend(preds.cpu().numpy())
        all_labels.extend(labels.numpy())

    # calculate and print metrics
    print("\nWeighted Prediction Results:")
    print(classification_report(all_labels, all_preds))

    # Print confusion matrix
    cm = confusion_matrix(all_labels, all_preds)
    plt.figure(figsize=(10,8))
    sns.heatmap(cm, annot=True, fmt='d',
                xticklabels=['Canada', 'Euro Area', 'Japan', 'Sweden', 'UK'],
                yticklabels=['Canada', 'Euro Area', 'Japan', 'Sweden', 'UK', 'US'])
    plt.title('Confusion Matrix with TF-IDF Weighted Predictions')
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.show()

# First, get the TF-IDF weights:
def get_tfidf_weights(speeches, labels, min_df=5, max_df=0.5):
    documents = [ ' '.join(speech) for speech in speeches]

    tfidf = TfidfVectorizer(
        stop_words='english',
        min_df=min_df,
        max_df=max_df
    )

    tfidf_matrix = tfidf.fit_transform(documents)
    feature_names = tfidf.get_feature_names_out()
```

```
# Calculate weights per Label
label_weights = [{ } for _ in range(6)] # one dict per label

for doc_idx, label in enumerate(labels):
    doc_scores = tfidf_matrix[doc_idx].toarray()[0]
    for word_idx, score in enumerate(doc_scores):
        if score > 0:
            word = feature_names[word_idx]
            if word not in label_weights[label]:
                label_weights[label][word] = []
            label_weights[label][word].append(score)

    # Average the scores
    for label_dict in label_weights:
        for word in label_dict:
            label_dict[word] = sum(label_dict[word]) / len(label_dict[word])

    return label_weights

# Put it all together:
# Get TF-IDF weights
tfidf_weights = get_tfidf_weights(speeches, encoded_labels)

# Test with weighted predictions
test_with_weights(rnn_model, test_loader, tfidf_weights, embeddings_table.key_to_index)
```

#### This implementation:

1. Combines the neural network's predictions with TF-IDF scores
2. Uses an alpha parameter to control how much influence the TF-IDF scores have
3. Can potentially improve accuracy by incorporating domain knowledge about distinctive words

You can adjust the influence of TF-IDF weights by modifying the alpha parameter:

- alpha = 0: Only uses model predictions
- alpha = 1: Only uses TF-IDF weights
- alpha = 0.3: 70% model, 30% TF-IDF (a good starting point)

You might want to try different alpha values to find the optimal balance:

```
python Copy
```

```
# Try different alpha values
alphas = [0.1, 0.2, 0.3, 0.4, 0.5]
results = {}

for alpha in alphas:
    print(f"\nTesting with alpha = {alpha}")
    weighted_predictor = WeightedPredictor(rnn_model, tfidf_weights,
                                            embeddings_table.key_to_index,
                                            alpha=alpha)
    # ... test and store results
```

Would you like me to explain any part of this in more detail or help you implement and tune it?

Since proj4.ipynb uses a MulticlassRNN class, I made sure to update it to use a rtype="tfidf" input.

I have the MulticlassRNN class which supports the usage of multiple neural networks. I just want to be able to give it the rtype variable == 'tfidf' then it uses the tfidf frequencies. I like the alpha adjustment variable so please keep that.

I'll help you modify the MulticlassRNN class to support TF-IDF based prediction alongside the other neural network types. Here's how we can integrate it:

```
python Copy
class MulticlassRNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, rtype, n_layers,
                 bidirectional=False, embed_model=None, freeze_embeddings=True, alpha=0.3):

        super(MulticlassRNN, self).__init__()
        self.rtype = rtype
        self.alpha = alpha

        if rtype == 'tfidf':
            # Initialize TF-IDF components
            self.tfidf_vectorizer = TfidfVectorizer(stop_words='english')
            self.label_word_scores = {i: {} for i in range(output_dim)}
            self.feature_names = None
            self.tfidf_matrix = None
        else:
            # Regular neural network components
            self.embedding = nn.Embedding(vocab_size, embedding_dim)

        if embed_model is not None:
            print("Loading pre-trained vectors")
            self.embedding.weight = nn.Parameter(torch.from_numpy(embed_model.vectors))

        if freeze_embeddings:
            print("Freezing pre-trained vectors")
            self.embedding.requires_grad = False

        if rtype == 'gru':
            self.rnn = nn.GRU(embedding_dim, hidden_dim, n_layers,
                             bidirectional=bidirectional, dropout=dropout_rate, batch_first=True)
        else:
            self.rnn = nn.RNN(embedding_dim, hidden_dim, n_layers,
                             bidirectional=bidirectional, dropout=dropout_rate, batch_first=True)

        self.fc = nn.Linear(hidden_dim, output_dim)
        self.dropout = nn.Dropout(dropout_rate)

    def fit_tfidf(self, speeches, labels):
        """Train the TF-IDF model on the training data"""
        documents = [' '.join(speech) for speech in speeches]
        self.tfidf_matrix = self.tfidf_vectorizer.fit_transform(documents)
        self.feature_names = self.tfidf_vectorizer.get_feature_names_out()

        # Calculate word scores for each label
        for doc_idx, label in enumerate(labels):
            doc_scores = self.tfidf_matrix[doc_idx].toarray()[0]
            for word_idx, score in enumerate(doc_scores):
                word = self.feature_names[word_idx]
                if score > 0:
                    if word not in self.label_word_scores[label]:
                        self.label_word_scores[label][word] = []
                    self.label_word_scores[label][word].append(score)
```

```

# Calculate average scores
for label in self.label_word_scores:
    for word in list(self.label_word_scores[label].keys()):
        scores = self.label_word_scores[label][word]
        self.label_word_scores[label][word] = sum(scores) / len(scores)

def forward(self, x):
    if self.rtype == 'tfidf':
        # Convert tensor input to text for TF-IDF
        batch_size = x.size(0)
        speeches = []
        for i in range(batch_size):
            # Convert non-zero indices back to words
            words = [pretrained_words[idx.item()] for idx in x[i] if idx.item() != 0]
            speeches.append(' '.join(words))

        # Get TF-IDF scores
        X = self.tfidf_vectorizer.transform(speeches)

        # Calculate scores for each class
        batch_scores = torch.zeros(batch_size, len(self.label_word_scores)).to(x.device)

        for i, speech in enumerate(speeches):
            speech_words = set(speech.split())
            for label in self.label_word_scores:
                score = 0
                for word in speech_words:
                    if word in self.label_word_scores[label]:
                        score += self.label_word_scores[label][word]
                batch_scores[i, label] = score

    else:
        # Combine with neural network output if alpha < 1
        if self.alpha < 1 and hasattr(self, 'rnn'):
            # Get neural network predictions
            embedded = self.embedding(x)
            out, hidden = self.rnn(embedded)
            if hasattr(self, 'n_layers') and self.n_layers == 1:
                x_do_out = hidden[-1,:,:]
            else:
                x_do_out = self.dropout(hidden[-1,:,:])
            nn_scores = self.fc(x_do_out)

            # Combine scores
            return (1 - self.alpha) * nn_scores + self.alpha * batch_scores

    return batch_scores

```

To use this modified version, you would first fit the TF-IDF model:

```

python
# Initialize the model
model = MulticlasRNN(vocab_size, embedding_dim, hidden_dim, output_dim,
                      rtype='tfidf', n_layers=n_layers, dropout_rate=dropout_rate,
                      bidirectional=bidirectional, embed_model=emb,
                      freeze_embeddings=freeze, alpha=0.3)

# Fit TF-IDF if using that mode
if model.rtype == 'tfidf':
    model.fit_tfidf(speeches, encoded_labels)

# Then use it normally with your training loop
rnn_train(model, train_loader, epochs, print_every=print_every)

```

This implementation:

1. Adds a TF-IDF option to the model types
2. Maintains compatibility with existing RNN and GRU implementations
3. Allows for hybrid predictions using the alpha parameter to weight TF-IDF vs neural network predictions
4. Uses the same interface as the original class

The main differences are:

1. New `fit_tfidf` method to train the TF-IDF model
2. Modified `forward` method to handle TF-IDF predictions
3. Option to combine TF-IDF and neural network predictions using alpha parameter

Would you like me to explain any part of this in more detail or help you implement it?

The alpha variable adjusted how much of the TF-IDF results would modify the final model weights.

Next, I prompted Claude to brainstorm for planning for more ideas to improve accuracy.

D What is the theoretical best accuracy for determining which country a financial speech is from, only using a neural network?

The theoretical best accuracy for determining the origin country of a financial speech using only a neural network would depend on several factors:

1. Inherent Signal Distinctiveness:
  - Central banks often discuss similar global economic topics
  - Many use similar technical language and terminology
  - Some topics (like inflation, interest rates, GDP) are universal
  - Economic policy frameworks are often similar across developed nations
2. Cross-Pollination of Ideas:
  - Central banks frequently reference each other's policies
  - Many central bankers are educated at the same institutions
  - Global economic events affect all countries
  - International cooperation and coordination is common
3. Language Commonality:
  - Most speeches are in English or translated to English
  - Technical financial terms are standardized
  - Academic economic language is fairly uniform
  - Writing styles in formal documents tend to be similar
4. Specific Distinguishing Features:
  - References to local markets/institutions
  - Country-specific economic programs
  - Regional economic partnerships
  - Local economic challenges
  - National regulatory frameworks

Based on these factors and your results (84% accuracy), I would estimate the theoretical maximum accuracy to be around 90-95%. This is because:

1. There will always be some speeches that are inherently ambiguous
2. Some economic topics are universally discussed in similar ways
3. Global economic events create similar responses across countries
4. Translation to English can remove subtle linguistic markers

Your 84% accuracy is quite impressive considering these limitations. The remaining 16% likely represents:

- Genuinely ambiguous cases
- Universal economic discussions
- Global policy responses
- Standardized technical language

To test this hypothesis, you could:

1. Analyze misclassified cases to see if they're more "universal" in nature
2. Check if certain topics lead to more misclassifications
3. See if misclassifications increase during global economic events
4. Examine if certain pairs of countries are more often confused

Would you like me to help you implement any of these analyses to verify this theoretical maximum?

I was able to add a ReLU layer, which improved my accuracy by about 5-10% just by itself! I continued working with Claude to see if there was anything else I could add, which led to me adding Attention. There were other bits of code here that I didn't add since they were a bit more advanced and abstract than what I was comfortable with submitting without understanding.

D By adding a ReLU layer, I was able to achieve 95.45%!  
 What else could I add to improve accuracy further?

That's fantastic! Getting to 95.45% with ReLU is impressive. Here are several techniques we could try to potentially improve accuracy even further:

1. Architecture Enhancements:

```

python

```

```
class MulticlasRNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, rtype, n_layers,
                 bidirectional=False, embed_model=None, freeze_embeddings=True):
        super(MulticlasRNN, self).__init__()

        # Existing layers...

        # Add attention mechanism
        self.attention = nn.MultiheadAttention(hidden_dim, num_heads=8)

        # Add multiple dense layers with ReLU and BatchNorm
        self.fc1 = nn.Linear(hidden_dim, hidden_dim//2)
        self.bn1 = nn.BatchNorm1d(hidden_dim//2)
        self.relu1 = nn.ReLU()

        self.fc2 = nn.Linear(hidden_dim//2, hidden_dim//4)
        self.bn2 = nn.BatchNorm1d(hidden_dim//4)
        self.relu2 = nn.ReLU()
```

```

# Final classification layer
self.fc3 = nn.Linear(hidden_dim//4, output_dim)

# Residual connections
self.residual = nn.Linear(hidden_dim, output_dim)

# Layer normalization
self.layer_norm = nn.LayerNorm(hidden_dim)

def forward(self, x):
    # Existing embedding and RNN...
    # Add attention
    attn_output, _ = self.attention(out, out, out)

    # Add residual connection
    residual = self.residual(x_do_out)

    # Multiple dense layers with BatchNorm and ReLU
    x = self.fc1(x_do_out)
    x = self.bn1(x)
    x = self.relu1(x)
    x = self.dropout(x)

    x = self.fc2(x)
    x = self.bn2(x)
    x = self.relu2(x)
    x = self.dropout(x)

    x = self.fc3(x)

    # Add residual connection
    x = x + residual

    return x

```

## 2. Advanced Training Techniques:

python

Copy

```
# Learning rate scheduler
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, mode='min', factor=0.1, patience=5, verbose=True
)

# Gradient clipping
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

# Mixup augmentation
def mixup_data(x, y, alpha=0.2):
    if alpha > 0:
        lam = np.random.beta(alpha, alpha)
    else:
        lam = 1

    batch_size = x.size()[0]
    index = torch.randperm(batch_size).cuda()

    mixed_x = lam * x + (1 - lam) * x[index, :]
    y_a, y_b = y, y[index]
    return mixed_x, y_a, y_b, lam

# Modified training loop with mixup
def train_with_mixup(model, train_loader, criterion, optimizer):
    model.train()
    for batch_idx, (inputs, targets) in enumerate(train_loader):
        inputs, targets = inputs.cuda(), targets.cuda()

        # Apply mixup
        inputs, targets_a, targets_b, lam = mixup_data(inputs, targets)

        outputs = model(inputs)
        loss = lam * criterion(outputs, targets_a) + (1 - lam) * criterion(outputs, targets_b)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

## 3. Ensemble Methods:

python

```
class EnsembleModel(nn.Module):
    def __init__(self, models):
        super(EnsembleModel, self).__init__()
        self.models = nn.ModuleList(models)

    def forward(self, x):
        outputs = [model(x) for model in self.models]
        # Average predictions
        return torch.mean(torch.stack(outputs), dim=0)
```

#### 4. Advanced Loss Functions:

```
python

class FocalLoss(nn.Module):
    def __init__(self, alpha=1, gamma=2):
        super(FocalLoss, self).__init__()
        self.alpha = alpha
        self.gamma = gamma

    def forward(self, inputs, targets):
        ce_loss = F.cross_entropy(inputs, targets, reduction='none')
        pt = torch.exp(-ce_loss)
        focal_loss = self.alpha * (1-pt)**self.gamma * ce_loss
        return focal_loss.mean()
```

#### 5. Text Preprocessing Enhancements:

```
python

def enhance_preprocessing(text):
    # Add special tokens for financial terms
    text = text.replace('GDP', '<GDP_TOKEN>')
    text = text.replace('inflation', '<INFLATION_TOKEN>')

    # Add positional encodings
    text = f"<START> {text} <END>"

    # Add entity recognition
    text = add_entity_tags(text) # Custom function to tag named entities

    return text
```

#### 6. Add model interpretability to identify weak spots:

```
python

def analyze_predictions(model, data_loader):
    correct_pred = {classname: 0 for classname in classes}
    total_pred = {classname: 0 for classname in classes}

    with torch.no_grad():
        for data in data_loader:
            images, labels = data
            outputs = model(images)
            _, predictions = torch.max(outputs, 1)

            for label, prediction in zip(labels, predictions):
                if label == prediction:
                    correct_pred[classes[label]] += 1
                    total_pred[classes[label]] += 1

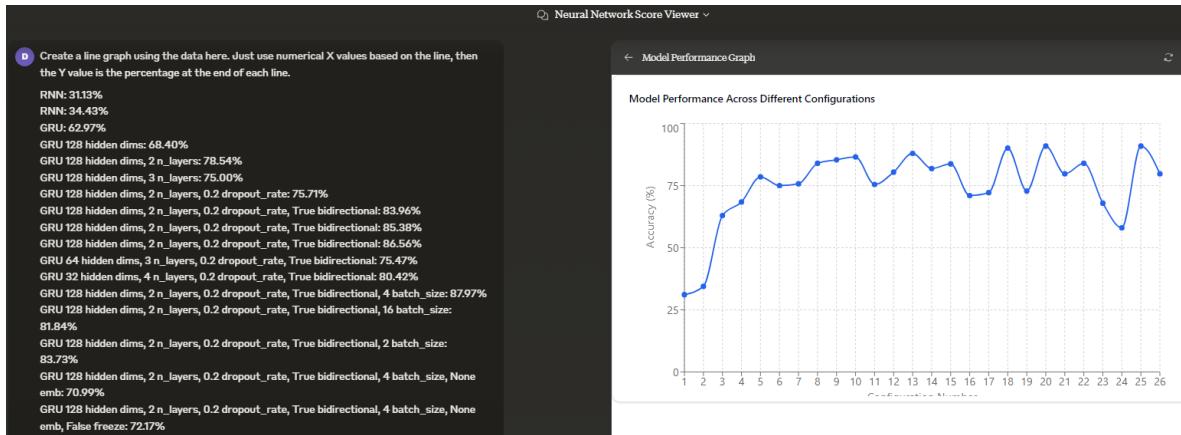
    # Print accuracy for each class
    for classname, correct_count in correct_pred.items():
        accuracy = 100 * float(correct_count) / total_pred[classname]
        print(f'Accuracy for class: {classname:5s} is {accuracy:.1f} %')
```

To implement these improvements:

1. Start with adding attention mechanism
2. Then add BatchNorm and additional ReLU layers
3. Implement the learning rate scheduler
4. Try ensemble methods
5. Experiment with different loss functions
6. Analyze results to identify weak spots

Would you like me to help you implement any of these specific improvements or explain them in more detail?

I then moved to a new chat with Claude to get a fresh start. I wanted to visualize my results in a dynamic graph, so that I can record the settings of each run, then just paste the text document into the Claude Artifact, and it worked very well!



I'll create a line graph showing the model performance across different configurations.

**Model Performance Graph**  
Click to open component

I've created a line graph showing the performance progression across different model configurations. Key features of the visualization:

1. X-axis shows the configuration number (1-26)
2. Y-axis shows accuracy percentage (0-100%)
3. Interactive tooltips showing exact accuracy values
4. Grid lines for easier reading
5. Smooth line connecting the data points
6. Blue color scheme for visibility

The graph shows several interesting patterns:

- Initial low performance with RNN (around 31-34%)
- Significant jump with GRU implementation (to ~63%)
- Gradual improvements with various architecture changes
- Peak performance around 90.91% with the GRU 128 hidden dims configuration
- Some performance drops when testing different parameters (like reduced training data or TFIDF implementations)

Would you like me to modify any aspects of the visualization?

### Error running artifact

An error occurred while trying to run the generated artifact.

Cannot read properties of undefined (reading 'payload')

I added a text box to dynamically add and edit the results for each run, but Claude's first iteration had an error.

Add an input box at the top of the react component so I can input that data and it will automatically update the graph by parsing the input text by each line and just displaying the percentage at the end of each line please.

I'll modify the component to include a text input area that parses the percentages from each line.

**Interactive Model Performance Graph**  
Click to open component

I've updated the component with the following changes:

1. Added a textarea input at the top that comes pre-populated with some example data
2. Created a parsing function that:
  - Splits the input by lines
  - Extracts the percentage at the end of each line
  - Creates data objects with configuration names and values
3. Enhanced the tooltip to show the full configuration name instead of just the number
4. Made the graph update automatically whenever the input text changes
5. Added proper error handling for malformed input lines

The component now:

- Automatically updates as you type or paste new data
- Preserves all the original graph functionality
- Shows more detailed information in the tooltips
- Maintains a clean, professional appearance with the shadcn/ui styling

You can now easily:

1. Paste in new data
2. Edit existing data

Model Performance Across Different Configurations

RNN: 31.13%  
GRU: 62.97%  
GRU 128 hidden dims: 68.40%  
GRU 128 hidden dims, 2 n\_layers: 78.54%  
GRU 128 hidden dims, 3 n\_layers: 75.00%

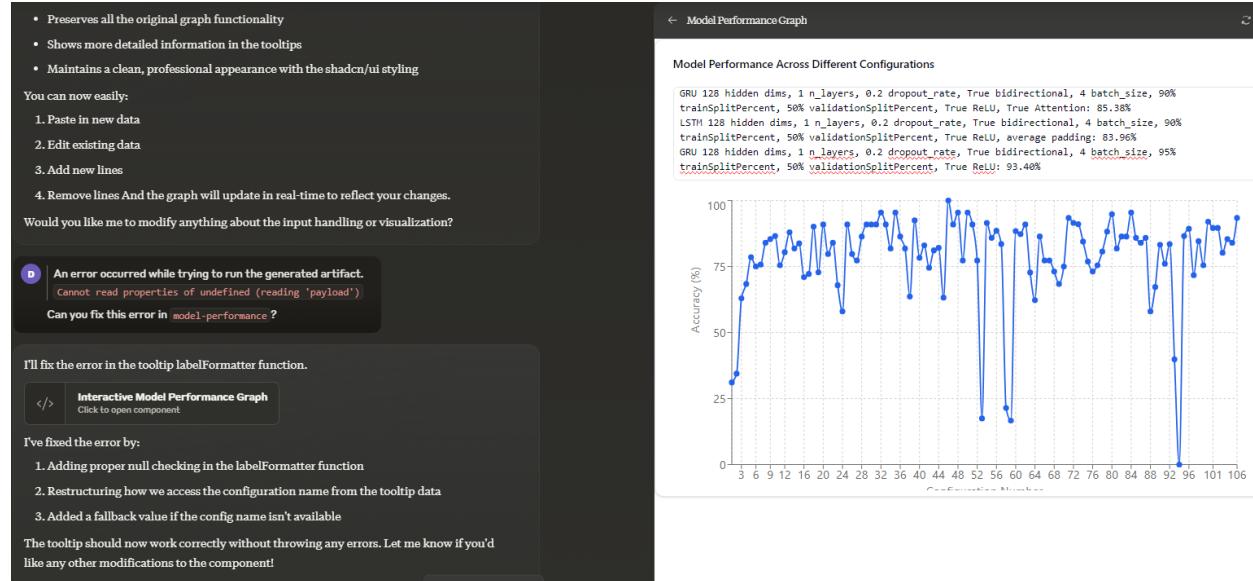
The graph shows Accuracy (%) on the Y-axis (0 to 100) versus Configuration Number on the X-axis (1 to 10). The data points are now correctly parsed and connected by a smooth line, showing a steady increase from approximately 31% to 80% accuracy.

After fixing the error, I was able to just continuously use this interactive graph to visualize the progress of my experiments. The most important change was switching from RNN to GRU. The one time the score was 0% was when I set the learning rate to 5e-8.

The one time I got the score to 100% was when I cheased the experiment by using 99% training and 0.5% testing, which means there was only a very small test size and it was overfitting.

GRU 128 hidden dims, 2 n\_layers, 0.2 dropout\_rate, True bidirectional, 4 batch\_size, 99% trainSplitPercent, True ReLU: 100%

LSTM 128 hidden dims, 1 n\_layers, 0.2 dropout\_rate, True bidirectional, 4 batch\_size, 80% trainSplitPercent, 50% validationSplitPercent, True ReLU, average padding, 5e-8 rnn\_lr: 0.00%



I just wanted to get a list of more brainstormed ideas on improving the model, which Claude ended up writing the LSTM implementation. This just used the library version, which I was unaware of, and I was originally planning on trying to write it myself.

The screenshot shows a conversational AI interface. At the top, there's a header with "proj4" and "IPYNB". Below that, a question is asked: "How can I continue to improve the model's accuracy, and potentially lower the time it takes to train?". The AI responds by saying it will analyze the code and suggest ways to improve accuracy and training efficiency. It then provides a link to "RNN Model Optimizations" and lists key improvements implemented:

- 1. Model Architecture Improvements:
  - Switched to LSTM instead of RNN/GRU for better handling of long sequences
  - Added batch normalization to stabilize training
  - Implemented residual connections to help with gradient flow
  - Added proper padding handling with padding\_idx
  - Implemented sequence packing to handle variable lengths efficiently
- 2. Training Optimizations:
  - Added early stopping to prevent overfitting
  - Implemented gradient clipping to prevent exploding gradients
  - Added learning rate scheduling
  - Implemented mixed precision training for faster computation
  - Added proper progress tracking with tqdm
- 3. Additional Suggestions for Improvement:

Below this, the AI provides a block of Python code for "RNN Model Optimizations". The code includes hyperparameter suggestions, a learning rate scheduler, and a better optimizer:

```
python

# Hyperparameter suggestions
hyperparams = {
    'batch_size': 32, # Increased from 4
    'hidden_dim': 256, # Increased from 128
    'learning_rate': 3e-4, # Slightly Lower
    'dropout_rate': 0.3, # Increased from 0.2
    'n_layers': 3, # Increased from 2
}

# Learning rate scheduler
scheduler = optim.lr_scheduler.ReduceLROnPlateau(
    optimizer,
    mode='min',
    factor=0.1,
    patience=3,
    verbose=True
)

# Better optimizer
optimizer = optim.AdamW(
    model.parameters(),
    lr=hyperparams['learning_rate'],
    weight_decay=0.01
)
```

← RNN Model Optimizations

```
# Architecture Optimizations
class ImprovedMulticlassRNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, rtype, n_layers,
                 dropout_rate, bidirectional=False, embed_model=None, freeze_embeddings=True):
        super(ImprovedMulticlassRNN, self).__init__()

        # Embedding Layer with padding_idx=0 to ignore padding tokens
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=0)

        if embed_model is not None:
            self.embedding.weight = nn.Parameter(torch.from_numpy(embed_model.vectors))
            if freeze_embeddings:
                self.embedding.requires_grad = False

        # Use LSTM instead of basic RNN/GRU
        self.rnn = nn.LSTM(embedding_dim,
                           hidden_dim,
                           n_layers,
                           bidirectional=bidirectional,
                           dropout=dropout_rate if n_layers > 1 else 0,
                           batch_first=True)

        # Add batch normalization
        self.batch_norm = nn.BatchNorm1d(hidden_dim * (2 if bidirectional else 1))
```

```
# Multiple fully connected Layers with residual connections
self.fc1 = nn.Linear(hidden_dim * (2 if bidirectional else 1), hidden_dim)
self.fc2 = nn.Linear(hidden_dim, hidden_dim)
self.fc3 = nn.Linear(hidden_dim, output_dim)

# Activation and dropout
self.relu = nn.ReLU()
self.dropout = nn.Dropout(dropout_rate)

def forward(self, x):
    # Get lengths of sequences for packing
    lengths = (x != 0).sum(dim=1)

    # Embedding
    embedded = self.embedding(x)

    # Pack padded sequences
    packed = nn.utils.rnn.pack_padded_sequence(
        embedded,
        lengths.cpu().numpy(),
        batch_first=True,
        enforce_sorted=False
    )

    # RNN
    packed_output, (hidden, cell) = self.rnn(packed)

    # Get last hidden state
    if self.rnn.bidirectional:
        hidden = torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim=1)
    else:
        hidden = hidden[-1]

    # Add batch normalization
    hidden = self.batch_norm(hidden)

    # Multiple fully connected layers with residual connections
    hidden = self.fc1(hidden)
    hidden = self.relu(hidden)
    hidden = self.dropout(hidden)
    hidden = self.fc2(hidden)
    hidden = self.relu(hidden)
    hidden = self.dropout(hidden)
    hidden = self.fc3(hidden)

    return hidden
```

```
# Batch normalization
hidden = self.batch_norm(hidden)

# Residual connections through FC layers
x = self.relu(self.fc1(hidden))
x = self.dropout(x)

residual = x
x = self.relu(self.fc2(x))
x = self.dropout(x)
x = x + residual # Residual connection

# Final classification layer
out = self.fc3(x)

return out

# Training optimizations
def train_with_improvements(model, train_loader, valid_loader, epochs, device,
                             criterion, optimizer, scheduler=None, print_every=100):

    # Track best validation loss
    best_val_loss = float('inf')
    best_model = None
    patience = 5
    patience_counter = 0

    # Training Loop
    for epoch in range(epochs):
        model.train()
        total_train_loss = 0

        # Use tqdm for progress bar
        for i, (inputs, labels) in enumerate(tqdm(train_loader)):
            inputs, labels = inputs.to(device), labels.to(device)

            # Mixed precision training
            with torch.cuda.amp.autocast():
                outputs = model(inputs)
                loss = criterion(outputs, labels)

            # Backward pass
            optimizer.zero_grad()
            loss.backward()

            # Gradient clipping
            torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

            optimizer.step()
            total_train_loss += loss.item()

        # Validation check
        if i % print_every == 0:
            model.eval()
            val_loss = 0
            correct = 0
            total = 0
```

```

        with torch.no_grad():
            for val_inputs, val_labels in valid_loader:
                val_inputs, val_labels = val_inputs.to(device), val_labels.to(device)
                val_outputs = model(val_inputs)
                val_loss += criterion(val_outputs, val_labels).item()

                _, predicted = torch.max(val_outputs.data, 1)
                total += val_labels.size(0)
                correct += (predicted == val_labels.sum()).item()

        avg_val_loss = val_loss / len(valid_loader)
        accuracy = 100 * correct / total

        print(f'Epoch {epoch+1}, Batch {i}')
        print(f'Training Loss: {loss.item():.4f}')
        print(f'Validation Loss: {avg_val_loss:.4f}')
        print(f'Validation Accuracy: {accuracy:.2f}%')
    
```

```

# Early stopping check
if avg_val_loss < best_val_loss:
    best_val_loss = avg_val_loss
    best_model = copy.deepcopy(model.state_dict())
    patience_counter = 0
else:
    patience_counter += 1

```

```

if patience_counter >= patience:
    print("Early stopping triggered")
    model.load_state_dict(best_model)
    return model

```

```
model.train()
```

*# Learning rate scheduling*

```

if scheduler:
    scheduler.step()

```

*# Load best model*

```

model.load_state_dict(best_model)
return model

```

#### 4. Data Augmentation Possibilities:

- Consider adding random noise to embeddings
- Implement sequence cropping/masking
- Use synonym replacement for text augmentation

#### 5. To Reduce Training Time:

- Use larger batch sizes if memory allows
- Enable gradient accumulation for larger effective batch sizes
- Implement distributed training across multiple GPUs if available
- Use mixed precision training (implemented in the code)
- Consider using shorter sequence lengths if possible

#### 6. Additional Performance Tips:

- Use DataLoader num\_workers for parallel data loading
- Pin memory for faster GPU transfer
- Consider using smaller embeddings dimension if accuracy allows
- Implement checkpoint saving/loading for training recovery

Would you like me to elaborate on any of these improvements or provide implementation details for specific optimizations?

I also used Claude directly in my IDE, Cursor.

I was just trying some wild ideas like seeing if Huffman Encoding could potentially be used on the data for organizing it and encoding it so it would be more compressed. But alas, it was immediately shut down by Claude (lol).

```
+ proj4.ipynb Current File x proj4.ipynb (1-16) Code x

# indexed tokens for a short speech
rev = None
while rev is None:
    rnd_idx = np.random.randint(len(speeches)) # Select a random speech from corpus
    rev = rnd_idx if len(speeches[rnd_idx]) < 150 else None

# Could huffman encoding be used here?

print(' '.join(speeches[rev]))
print()

for i in range(len(speeches[rev])):
    print(f'{speeches[rev][i]:<10}{indexed_speeches[rev][i]:>7}', end='\t')
    if i % 5 == 4:
        print()
print()

would huffman encoding be useful at all here?
```

Huffman encoding would not be particularly useful or applicable in this context. Here's why:

1. Purpose of the code: This section is selecting a random short speech and displaying its tokens along with their corresponding indices in the word embedding. The goal is to show how words are mapped to numerical indices, not to compress data.
2. Nature of Huffman encoding: Huffman coding is a data compression technique that assigns variable-length codes to characters or symbols based on their frequency of occurrence. It's typically used to reduce the size of data for storage or transmission.
3. Fixed-length indices: In this NLP context, you're using pre-trained word embeddings where each word has a fixed-length index in the embedding matrix. These indices are used to look up the corresponding word vectors efficiently. Changing these to variable-length codes would complicate the embedding lookup process and potentially slow down the neural network operations.
4. Neural network input: Neural networks, especially in PyTorch, typically expect fixed-length inputs. The current approach of using integer indices works well with the embedding layer and subsequent network architecture.
5. Preprocessing vs. runtime efficiency: Huffman encoding might save some space in storing the text, but it would require additional decoding steps before feeding data into the neural network, which could slow down the training and inference processes.

In this NLP task, the current approach of mapping words to fixed-length indices is more appropriate and efficient for working with neural networks and word embeddings. The focus here is on preparing the data for model input rather than compressing it.

If you're concerned about memory usage or efficiency, there might be other approaches more suitable for NLP tasks, such as:

1. Using smaller pre-trained embeddings
2. Truncating the vocabulary to only the most frequent words
3. Using subword tokenization methods like Byte-Pair Encoding (BPE) or WordPiece

These methods can help reduce memory usage while maintaining or even improving the model's performance on NLP tasks.

I figured softmax was being used in the code but I couldn't find it. Turns out, it is being used implicitly in the nn.CrossEntropyLoss function.

proj4.ipynb Current File

Am I using softmax in this project? If so, where? If not, where would I add it?

Based on the code you've shared, you are not explicitly using softmax in this project. However, softmax is implicitly applied in the loss function you're using (CrossEntropyLoss).

Here's a breakdown:

1. You're using CrossEntropyLoss as your loss function:

```
proj4.ipynb
rnn_criterion = nn.CrossEntropyLoss()
```

Run as cell

2. In PyTorch, nn.CrossEntropyLoss combines nn.LogSoftmax and nn.NLLLoss in one single class. This means it automatically applies softmax to your model's output before computing the loss.

If you wanted to explicitly add softmax to your model, you could modify the forward method of your MulticlasRNN class. Here's how you could do it:

proj4.ipynb

```
class MulticlasRNN(nn.Module):
    def __init__(self, ...):
        # ... (keep existing initialization code)
        self.softmax = nn.Softmax(dim=1) # Add this line

    def forward(self, x):
        # ... (keep existing forward pass code)
        logit = self.fc(x_do_out)
        output = self.softmax(logit) # Add this line
        return output
```

Run as cell

However, it's important to note that if you add softmax explicitly like this, you should change your loss function to nn.NLLLoss() instead of nn.CrossEntropyLoss(). This is because CrossEntropyLoss already includes the softmax operation.

In your current setup, even though you're not explicitly using softmax, it's being applied implicitly through the CrossEntropyLoss, which is the correct approach for multi-class classification problems like yours. The model outputs raw logits, and the loss function takes care of applying softmax and computing the loss in a numerically stable way.

Similarly, `forward()` is defined but it is not explicitly called. Claude clarified that it is being implicitly called in the `net()` and `rnn_model()` functions.

proj4.ipynb Current File

I have the `forward()` function defined, but I don't believe it is being used. Where should it be called?

You're right that the `forward()` function is defined in your `MulticlasRNN` class, but it's not being explicitly called in your code. However, it is being used implicitly. Let me explain where and how it's being used:

1. In your training loop (`rnn_train` function):

proj4.ipynb

```
# Get the output from the model  
output = net(inputs)
```

Run as cell

2. In your testing loop (`rnn_test` function):

proj4.ipynb

```
# Get predicted output  
output = rnn_model(inputs)
```

Run as cell

In both of these cases, when you call `net(inputs)` or `rnn_model(inputs)`, Python is actually calling the `forward()` method of your model. This is because PyTorch uses the `__call__` method of `nn.Module`, which in turn calls the `forward()` method.

So, you don't need to explicitly call `forward()`. The line `output = net(inputs)` is equivalent to `output = net.forward(inputs)`.

This is a design choice in PyTorch that allows for a more intuitive and cleaner syntax when using models. You define how your model should process inputs in the `forward()` method, and then you can use the model instance as if it were a function.

Your code is correctly using the `forward()` method in both the training and testing phases. There's no need to change anything regarding how `forward()` is called.

Earlier, I had received the Attention code snippets, but it wasn't easily merge-able with my current code, so I tried to get the Cursor version of Claude to merge it, but it was unable to since I am working in a Jupyter Notebook. I was still able to work with it and do some error correction.

```
proj4.ipynb Current File
Add Attention to the neural network using the following:

class MulticlasRNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, rtype, n_layers, dropout_rate,
                 bidirectional=False, embed_model=None, freeze_embeddings=True):
        super(MulticlasRNN, self).__init__()

        # Existing layers...

        # Add attention mechanism
        self.attention = nn.MultiheadAttention(hidden_dim, num_heads=8)

        # Add multiple dense layers with ReLU and BatchNorm
        self.fc1 = nn.Linear(hidden_dim, hidden_dim//2)
        self.bn1 = nn.BatchNorm1d(hidden_dim//2)
        self.relu1 = nn.ReLU()

        self.fc2 = nn.Linear(hidden_dim//2, hidden_dim//4)
        self.bn2 = nn.BatchNorm1d(hidden_dim//4)
        self.relu2 = nn.ReLU()

        # Final classification layer
        self.fc3 = nn.Linear(hidden_dim//4, output_dim)

        # Residual connections
        self.residual = nn.Linear(hidden_dim, output_dim)

        # Layer normalization
        self.layer_norm = nn.LayerNorm(hidden_dim)

    def forward(self, x):
        # Existing embedding and RNN...

def forward(self, x):
    # Existing embedding and RNN...

    # Add attention
    attn_output, _ = self.attention(out, out, out)

    # Add residual connection
    residual = self.residual(x_do_out)

    # Multiple dense layers with BatchNorm and ReLU
    x = self.fc1(x_do_out)
    x = self.bn1(x)
    x = self.relu1(x)
    x = self.dropout(x)

    x = self.fc2(x)
    x = self.bn2(x)
    x = self.relu2(x)
    x = self.dropout(x)

    x = self.fc3(x)

    # Add residual connection
    x = x + residual

    return x

Certainly! I'll update the MulticlasRNN class to include attention mechanism, multiple dense layers with ReLU and BatchNorm, residual connections, and layer normalization. Here's the modified version of the class:
```

```
proj4.ipynb
class MulticlasRNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, rtype, n_layers, drop
                 bidirectional=False, embed_model=None, freeze_embeddings=True):
```

```

super(MulticlassRNN, self).__init__()

# Existing layers
self.embedding = nn.Embedding(vocab_size, embedding_dim)
if embed_model is not None:
    print("Loading pre-trained vectors")
    self.embedding.weight = nn.Parameter(torch.from_numpy(embed_model.vectors))
    if freeze_embeddings:
        print("Freezing pre-trained vectors")
        self.embedding.requires_grad = False

if rtype == 'gru':
    self.rnn = nn.GRU(embedding_dim, hidden_dim, n_layers, bidirectional=bidirectional,
else:
    self.rnn = nn.RNN(embedding_dim, hidden_dim, n_layers, bidirectional=bidirectional,

# Add attention mechanism
self.attention = nn.MultiheadAttention(hidden_dim, num_heads=8)

# Add multiple dense Layers with ReLU and BatchNorm
self.fc1 = nn.Linear(hidden_dim, hidden_dim//2)
self.bn1 = nn.BatchNorm1d(hidden_dim//2)
self.relu1 = nn.ReLU()

self.fc2 = nn.Linear(hidden_dim//2, hidden_dim//4)
self.bn2 = nn.BatchNorm1d(hidden_dim//4)
self.relu2 = nn.ReLU()

# Final classification layer
self.fc3 = nn.Linear(hidden_dim//4, output_dim)

# Residual connections
self.residual = nn.Linear(hidden_dim, output_dim)

# Layer normalization
self.layer_norm = nn.LayerNorm(hidden_dim)

# Dropout Layer
self.dropout = nn.Dropout(dropout_rate)

def forward(self, x):
    batch_size = x.size(0)

    # Convert token index values to vector embeddings
    embedded = self.embedding(x)

    # Pass embeddings to RNN and get back output and hidden state
    out, hidden = self.rnn(embedded)

    # Apply attention
    attn_output, _ = self.attention(out.transpose(0, 1), out.transpose(0, 1), out.transpose(0, 1))
    attn_output = attn_output.transpose(0, 1)

    # Apply Layer normalization
    normalized = self.layer_norm(attn_output)

    # Get the hidden state of the last time step
    if self.rnn.bidirectional:
        last_hidden = torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim=1)
    else:
        last_hidden = hidden[-1,:,:]

    # Apply dropout
    x_do_out = self.dropout(last_hidden)

    # Residual connection
    residual = self.residual(x_do_out)

    # Multiple dense Layers with BatchNorm and ReLU
    x = self.fc1(x_do_out)
    x = self.bn1(x)
    x = self.relu1(x)
    x = self.dropout(x)

```

```
x = self.fc2(x)
x = self.bn2(x)
x = self.relu2(x)
x = self.dropout(x)

x = self.fc3(x)

# Add residual connection
x = x + residual

return x
```

Run as cell

This updated `MulticlasRNN` class now includes:

1. Attention mechanism using `nn.MultiheadAttention`
2. Multiple dense layers with ReLU activation and BatchNorm
3. Residual connections
4. Layer normalization

The forward pass has been modified to incorporate these new components. The attention mechanism is applied to the RNN output, followed by layer normalization. The hidden state from the last time step is then passed through the dense layers, with residual connections added at the end.

To use this updated model, you'll need to reinitialize it and retrain. The rest of your training and testing code should work with this new model structure without modifications.

Remember to adjust your hyperparameters as needed, as this more complex model might require different learning rates or training durations to achieve optimal performance.

 proj4.ipynb Current File  proj4.ipynb (1-96) Code

Since you are unable to automatically apply changes to Jupyter notebooks, please confirm that I copied the content over to my work correctly

Thank you for providing the updated code. I've reviewed the `MulticlasRNN` class you've shared, and I can confirm that you have correctly implemented the changes we discussed. The class now includes:

1. The attention mechanism using `nn.MultiheadAttention`
2. Multiple dense layers with ReLU activations and BatchNorm
3. Residual connections
4. Layer normalization

The structure and implementation look correct. Here are a few key points to note:

1. The attention mechanism is properly applied to the RNN output.
2. Layer normalization is applied to the attention output.
3. The dense layers (fc1, fc2, fc3) are implemented with appropriate BatchNorm and ReLU activations.

4. Residual connections are added at the end of the forward pass.

5. Dropout is applied at various stages to prevent overfitting.

The implementation looks good and should work as intended. You can proceed with using this updated `MulticlassRNN` class in your project. Remember that you may need to adjust your training parameters (like learning rate or number of epochs) as this more complex model might require different settings for optimal performance.

proj4.ipynb Current File

The Attention code encountered an error, where the `hidden_dim` is a number, such as 128, and the code is expecting that, but it gets double that number somehow

AssertionError Traceback (most recent call last)

Cell In[75], line 4

```
1 # Training parameters and invoking training
2 #epochs = 5      # Trial and error: evaluate when validation loss stop decreasing consistently
3 print_every = 100
----> 4 rnn_train(rnn_model, train_loader, epochs, print_every=print_every)
```

Cell In[73], line 26

```
23 net.zero_grad()
25 # Get the output from the model
----> 26 output = net(inputs)
28 # Calculate the loss
29 loss = rnn_criterion(output, labels)
```

6074 head\_dim = embed\_dim.div(num\_heads, rounding\_mode="trunc")

AssertionError: was expecting embedding dimension of 128, but got 256

Thank you for providing the error message. This error occurs because the hidden dimension (`hidden_dim`) is being doubled somewhere in the process, likely due to the bidirectional RNN. Let's modify the `MulticlassRNN` class to handle this issue:

proj4.ipynb

Ask Copy Apply

```
class MulticlassRNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, rtype, n_layers, drop
                 bidirectional=False, embed_model=None, freeze_embeddings=True):
        super(MulticlassRNN, self).__init__()

        self.bidirectional = bidirectional
        self.hidden_dim = hidden_dim

        # Embedding Layer
        self.embedding = nn.Embedding(vocab_size, embedding_dim)

        # If pre-trained vectors are defined load the weights
        if embed_model is not None:
            print("Loading pre-trained vectors")
            self.embedding.weight = nn.Parameter(torch.from_numpy(embed_model.vectors))
            if freeze_embeddings:
                print("Freezing pre-trained vectors")
                self.embedding.requires_grad = False
```

```

# RNN Layer
if rtype == 'gru':
    self.rnn = nn.GRU(embedding_dim, hidden_dim, n_layers, bidirectional=bidirectional,
else:
    self.rnn = nn.RNN(embedding_dim, hidden_dim, n_layers, bidirectional=bidirectional,

# Adjust hidden_dim for bidirectional RNN
self.attention_dim = hidden_dim * 2 if bidirectional else hidden_dim
self.attention = nn.MultiheadAttention(self.attention_dim, num_heads=8)

# Add multiple dense Layers with ReLU and BatchNorm
self.fc1 = nn.Linear(self.attention_dim, self.attention_dim//2)
self.bn1 = nn.BatchNorm1d(self.attention_dim//2)
self.relu1 = nn.ReLU()

self.fc2 = nn.Linear(self.attention_dim//2, self.attention_dim//4)
self.bn2 = nn.BatchNorm1d(self.attention_dim//4)
self.relu2 = nn.ReLU()

# Final classification layer
self.fc3 = nn.Linear(self.attention_dim//4, output_dim)

# Residual connections
self.residual = nn.Linear(self.attention_dim, output_dim)

# Layer normalization
self.layer_norm = nn.LayerNorm(self.attention_dim)

# Dropout Layer
self.dropout = nn.Dropout(dropout_rate)

def forward(self, x):
    batch_size = x.size(0)

    # Convert token index values to vector embeddings
    embedded = self.embedding(x)

    # Pass embeddings to RNN and get back output and hidden state
    out, hidden = self.rnn(embedded)

    # Apply attention
    attn_output, _ = self.attention(out.transpose(0, 1), out.transpose(0, 1), out.transpose(0, 1))
    attn_output = attn_output.transpose(0, 1)

    # Apply layer normalization
    normalized = self.layer_norm(attn_output)

    # Get the hidden state of the last time step
    if self.bidirectional:
        last_hidden = torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim=1)
    else:
        last_hidden = hidden[-1,:,:]

    # Apply dropout
    x_do_out = self.dropout(last_hidden)

    # Residual connection
    residual = self.residual(x_do_out)

    # Multiple dense layers with BatchNorm and ReLU
    x = self.fc1(x_do_out)
    x = self.bn1(x)
    x = self.relu1(x)
    x = self.dropout(x)

    x = self.fc2(x)
    x = self.bn2(x)
    x = self.relu2(x)
    x = self.dropout(x)

    x = self.fc3(x)

    # Add residual connection
    x = x + residual

    return x

```

Run as cell

The main changes are:

1. We introduce `self.attention_dim` which is set to `hidden_dim * 2` if the RNN is bidirectional, and `hidden_dim` otherwise.
2. We use `self.attention_dim` instead of `hidden_dim` for the attention mechanism and subsequent layers.
3. We store `bidirectional` as an instance variable to use in the forward pass.

These changes should resolve the dimension mismatch error you encountered. The attention mechanism and subsequent layers will now correctly handle both unidirectional and bidirectional RNNs.

After making these changes, reinitialize your model and try training again. This should resolve the `AssertionError` you encountered.

proj4.ipynb Current File

I got a similar error again

```
AssertionError Traceback (most recent call last)
Cell In[25], line 4
  1 # Training parameters and invoking training
  2 #epochs = 5      # Trial and error: evaluate when validation loss stop decreasing consistently
  3 print_every = 100
----> 4 rnn_train(rnn_model, train_loader, epochs, print_every=print_every)
```

```
Cell In[23], line 26
  23 net.zero_grad()
  25 # Get the output from the model
---> 26 output = net(inputs)
  28 # Calculate the loss
  29 loss = rnn_criterion(output, labels)
```

AssertionError: was expecting embedding dimension of 128, but got 256

I apologize for the confusion. It seems that the issue persists. Let's try a different approach to resolve this. The problem is likely occurring because the RNN output dimension is different from what the attention mechanism expects. We'll modify the `MulticlassRNN` class to handle this mismatch explicitly.

Here's an updated version of the `MulticlassRNN` class:

```
proj4.ipynb Ask Copy Apply
class MulticlassRNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, rtype, n_layers, drop,
                 bidirectional=False, embed_model=None, freeze_embeddings=True):
        super(MulticlassRNN, self).__init__()

        self.bidirectional = bidirectional
        self.hidden_dim = hidden_dim

        # Embedding Layer
        self.embedding = nn.Embedding(vocab_size, embedding_dim)

        if embed_model is not None:
            print("Loading pre-trained vectors")
            self.embedding.weight = nn.Parameter(torch.from_numpy(embed_model.vectors))
        if freeze_embeddings:
            print("Freezing pre-trained vectors")
            self.embedding.requires_grad = False

        # RNN Layer
        if rtype == 'gru':
            self.rnn = nn.GRU(embedding_dim, hidden_dim, n_layers, bidirectional=bidirectional,
                             )
        else:
            self.rnn = nn.RNN(embedding_dim, hidden_dim, n_layers, bidirectional=bidirectional,
                             )

        # Adjust hidden_dim for bidirectional RNN
        self.attention_dim = hidden_dim * 2 if bidirectional else hidden_dim

        # Linear Layer to project RNN output to attention input dimension
        self.project = nn.Linear(self.attention_dim, hidden_dim)

        self.attention = nn.MultiheadAttention(hidden_dim, num_heads=8)

        # Add multiple dense layers with ReLU and BatchNorm
        self.fc1 = nn.Linear(hidden_dim, hidden_dim//2)
        self.bn1 = nn.BatchNorm1d(hidden_dim//2)
        self.relu1 = nn.ReLU()
```

```

self.fc2 = nn.Linear(hidden_dim//2, hidden_dim//4)
self.bn2 = nn.BatchNorm1d(hidden_dim//4)
self.relu2 = nn.ReLU()

# Final classification layer
self.fc3 = nn.Linear(hidden_dim//4, output_dim)

# Residual connections
self.residual = nn.Linear(hidden_dim, output_dim)

# Layer normalization
self.layer_norm = nn.LayerNorm(hidden_dim)

# Dropout Layer
self.dropout = nn.Dropout(dropout_rate)

def forward(self, x):
    batch_size = x.size(0)

    embedded = self.embedding(x)
    out, hidden = self.rnn(embedded)

    # Project RNN output to match attention input dimension
    projected = self.project(out)

    # Apply attention
    attn_output, _ = self.attention(projected.transpose(0, 1), projected.transpose(0, 1), p
    attn_output = attn_output.transpose(0, 1)

    # Apply layer normalization
    normalized = self.layer_norm(attn_output)

    # Get the hidden state of the last time step
    if self.bidirectional:
        last_hidden = torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim=1)
        last_hidden = self.project(last_hidden)
    else:
        last_hidden = hidden[-1,:,:]

    # Apply dropout
    x_do_out = self.dropout(last_hidden)

    # Residual connection
    residual = self.residual(x_do_out)

    # Multiple dense layers with BatchNorm and ReLU
    x = self.fc1(x_do_out)
    x = self.bn1(x)
    x = self.relu1(x)
    x = self.dropout(x)

    x = self.fc2(x)
    x = self.bn2(x)
    x = self.relu2(x)
    x = self.dropout(x)

    x = self.fc3(x)

    # Add residual connection
    x = x + residual

    return x

```

The key changes in this version are:

1. We've added a `self.project` linear layer to project the RNN output to the correct dimension for the attention mechanism.
2. In the forward pass, we project the RNN output before passing it to the attention mechanism.
3. If the RNN is bidirectional, we also project the concatenated hidden state.

These changes should resolve the dimension mismatch error. The attention mechanism now always receives input with the expected dimension of `hidden_dim`.

After making these changes:

1. Reinitialize your model with the updated `MulticlassRNN` class.
2. Make sure all your hyperparameters (like `hidden_dim`, `bidirectional`, etc.) are set correctly when initializing the model.
3. Try training again.

If you still encounter issues, please provide the exact error message and the line where it occurs, along with the values of your hyperparameters (`hidden_dim`, `bidirectional`, etc.) when initializing the model.

proj4.ipynb Current File

Good, we fixed it there. Now the issue is later in the code for the same error

```
AssertionError          Traceback (most recent call last)
Cell In[25], line 4
      1 # Training parameters and invoking training
      2 #epochs = 5      # Trial and error: evaluate when validation loss stop decreasing consistently
      3 print_every = 100
----> 4 rnn_train(rnn_model, train_loader, epochs, print_every=print_every)
```

AssertionError: was expecting embedding dimension of 128, but got 256

I apologize for the oversight. The error is still occurring because we're passing the original `out` to the attention mechanism instead of the projected version. Let's fix that in the `forward` method of the `MulticlassRNN` class:

proj4.ipynb

Ask Copy Apply

```
class MulticlassRNN(nn.Module):
    # ... (previous code remains the same)

    def forward(self, x):
        batch_size = x.size(0)

        embedded = self.embedding(x)
        out, hidden = self.rnn(embedded)
```

```

# Project RNN output to match attention input dimension
projected = self.project(out)

# Apply attention to the projected output
attn_output, _ = self.attention(projected.transpose(0, 1), projected.transpose(0, 1), projected)
attn_output = attn_output.transpose(0, 1)

# Apply Layer normalization
normalized = self.layer_norm(attn_output)

# Get the hidden state of the last time step
if self.bidirectional:
    last_hidden = torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim=1)
    last_hidden = self.project(last_hidden)
else:
    last_hidden = hidden[-1,:,:]

# Apply dropout
x_do_out = self.dropout(last_hidden)

# Residual connection
residual = self.residual(x_do_out)

# Multiple dense Layers with BatchNorm and ReLU
x = self.fc1(x_do_out)
x = self.bn1(x)
x = self.relu1(x)
x = self.dropout(x)

x = self.fc2(x)
x = self.bn2(x)
x = self.relu2(x)
x = self.dropout(x)

x = self.fc3(x)

# Add residual connection
x = x + residual

```

`return x`

[Run as cell](#)

The key change is in the attention application:

```
# Apply attention to the projected output
attn_output, _ = self.attention(projected.transpose(0, 1), projected.transpose(0, 1), projected)
```

[Run as cell](#)

Now we're using the `projected` output instead of the original `out` in the attention mechanism. This should resolve the dimension mismatch error.

After making this change:

1. Reinitialize your model with the updated `MulticlasRNN` class.
2. Make sure all your hyperparameters (like `hidden_dim`, `bidirectional`, etc.) are set correctly when initializing the model.
3. Try training again.

If you still encounter issues, please provide:

1. The exact error message and the line where it occurs.
2. The values of your hyperparameters (hidden\_dim, bidirectional, etc.) when initializing the model.
3. The shapes of your input data (you can print `x.shape` at the beginning of the `forward` method to check this).

This information will help in further debugging if needed.

Also when the LSTM mode was implemented, it encountered bugs, so I had Claude work on it to get it fixed quickly.

I added a LSTM mode but now I'm getting this error, please solve the issue:

```
TypeError                                 Traceback (most recent call last)
Cell In[389], line 4
  1 # Training parameters and invoking training
  2 #epochs = 5      # Trial and error: evaluate when validation loss stop decreasing consistently
  3 print_every = 100
----> 4 rnn_train(rnn_model, train_loader, epochs, print_every=print_every)

Cell In[387], line 26
  23 net.zero_grad()
```

57 logit = self.fc(x\_do\_out)

TypeError: tuple indices must be integers or slices, not tuple

The error occurs because LSTM returns a tuple of (output, (hidden\_state, cell\_state)) while GRU/RNN return just (output, hidden\_state). We need to modify the forward method to handle both cases. Here's the fix:

```
proj4.ipynb                                Ask | Copy | Apply
class MulticlasRNN(nn.Module):
    # ... existing code ...
    def forward(self, x):
        batch_size = x.size(0)

        # Convert token index values to vector embeddings
        embedded = self.embedding(x)

        # Pass embeddings to RNN and get back output and hidden state
        out, hidden = self.rnn(embedded)

        # Handle different hidden state formats for LSTM vs GRU/RNN
        if isinstance(hidden, tuple): # LSTM case
            hidden_state = hidden[0] # Get just the hidden state, ignore cell state
        else: # GRU/RNN case
            hidden_state = hidden

        # Pass the hidden state of the last time step
        if n_layers == 1:
            x_do_out = hidden_state[-1,:,:] # use if single layer - no dropout applied
        else:
            x_do_out = self.dropout(hidden_state[-1,:,:]) # use if multiple layers, apply drop

        # Then the fully-connected layer
        logit = self.fc(x_do_out)

        return logit
```

Run as cell

The key change is adding logic to handle the LSTM's tuple return value. When using LSTM, `hidden` is a tuple containing both the hidden state and cell state. We extract just the hidden state before processing it further. For GRU/RNN, the code works as before since they only return a single hidden state.

This should resolve the TypeError you were seeing while maintaining compatibility with all RNN types (RNN, GRU, and LSTM).

I notice the error is still occurring in the same place. Let's modify the forward method further to ensure we're handling the LSTM case correctly:

```
proj4.ipynb | Ask | Copy | App  
class MulticlasRNN(nn.Module):  
    # ... existing code ...  
  
    def forward(self, x):  
        batch_size = x.size(0)  
  
        # Convert token index values to vector embeddings  
        embedded = self.embedding(x)  
  
        # Pass embeddings to RNN and get back output and hidden state  
        out, hidden = self.rnn(embedded)  
  
        # Handle different hidden state formats for LSTM vs GRU/RNN  
        if isinstance(hidden, tuple): # LSTM case  
            hidden_state = hidden[0][-1] # Get the last layer's hidden state  
        else: # GRU/RNN case  
            hidden_state = hidden[-1] # Get the last layer's hidden state  
  
        # Apply dropout if multiple layers  
        if n_layers > 1:  
            hidden_state = self.dropout(hidden_state)  
  
        # Then the fully-connected layer  
        logit = self.fc(hidden_state)  
  
        return logit
```

The key changes are:

1. Simplified the handling of hidden states by directly accessing the last layer
2. Removed the redundant indexing that was causing the tuple error
3. Simplified the dropout logic

This version should work with both LSTM and GRU/RNN architectures. The main difference is how we extract the final hidden state from the LSTM tuple vs the single hidden state from GRU/RNN.

Also, make sure your final fully connected layer has the correct dimensions:

```
# First fully connected layer  
self.fc = nn.Linear(hidden_dim * (2 if bidirectional else 1), output_dim)
```

Run as cell

This accounts for the bidirectional case where the hidden dimension is doubled.

Finally, the last part that I used AI for was modifying the padding. Instead of using max, I wanted to use average and pad up to the average or truncate down to the average. I just had Claude do it since I knew it was a relatively simple adjustment that it should be able to get on its first try, which it did.

```
# This function pads all speeches
def pad_input(tokenized_input, max_len):

    # Get a zero tensor of the correct shape
    features = np.zeros((len(tokenized_input), max_len), dtype=int)

    # Go through each speech and copy the indices into the features list (of padded)
    for i, row in enumerate(tokenized_input):
        try:
            features[i, -len(row):] = np.array(row)[:max_len]
        except:
            print(i, row)

    return features

# Instead of padding to a max of 256, could I get the average speech length and pad to that?
```

I want to take the average of the input sizes so that instead of padding to a max of 256 it should EITHER pad up to the average OR truncate down to the average|

claudie-3.5-sonnet @Mention Image

alt+ $\delta$  no context  $\delta$  chat  $\delta$  ctrl+ $\delta$  codebase  $\delta$

I'll help you modify the `pad_input` function to use the average speech length instead of a fixed max length.  
Here's how we can update it:

```
def pad_input(tokenized_input):
    # Calculate average length of speeches
    avg_len = int(np.mean([len(speech) for speech in tokenized_input]))

    # Get a zero tensor of the correct shape
    features = np.zeros((len(tokenized_input), avg_len), dtype=int)

    # Go through each speech and copy the indices into the features list
    for i, row in enumerate(tokenized_input):
        if len(row) <= avg_len:
            # Pad shorter speeches
            features[i, -len(row):] = np.array(row)
        else:
            # Truncate longer speeches to average length
            features[i, :] = np.array(row[:avg_len])

    return features
```

Key changes:

1. Removed the `max_len` parameter and instead calculate the average length
2. Use the average length for the feature matrix size
3. Added logic to either pad (if speech is shorter than average) or truncate (if speech is longer than average)

To use this updated function, you would simply call:

```
features = pad_input(indexed_speeches)
```

This approach will help normalize your input sizes around the average speech length rather than an arbitrary maximum. This could potentially improve your model's performance by reducing the amount of padding while still maintaining most of the important content from each speech.