Shiraz University

Neural Network Homework (2)

Text Emotion Detection using Elman Recurrent Neural Networks

Instructed by Dr.Mansoori

Reza Tahmasebi

saeed77t@gmail.com, Stu ID: 40160957

Table of Contents

This project aims to develop a text emotion detection system using a single-hidden layer recurrent network, specifically the Elman network. The dataset provided contains 3250 text rows categorized into 5 different emotion classes. The last 150 texts of each class are reserved for the test dataset. The steps involved in this project include preprocessing the text data, tokenization, stop word removal, padding, one-hot encoding, and training an Elman network for emotion detection.

1 Data Processing

1.1 Data Preprocessing

The data preprocessing step aims to clean and prepare the text data for further analysis. It involves the following sub-steps:

1.1.1 Non-Letter Character Removal:

Non-letter characters can introduce noise and interfere with the analysis. The code removes non-letter characters from the *'Text'* column in the dataset using the regular expression pattern ***r'[^a-zA-Z\s]'***. This pattern matches any character that is not a letter or whitespace and replaces it with an empty string. It ensures that only letters and spaces remain in the text data.

1.1.2 Removing Short Words:

Short words, which typically contain only a few characters, often carry little semantic meaning and can be removed. The code removes short words (with length ≤ 2) from the 'Text' column. It splits each text into words and joins them back if their length is greater than 2. This step ensures that only meaningful words are retained.

1.1.3 Stop Word Removal:

Stop words are commonly used words (e.g., 'a', 'and', 'the') that do not contribute much to the overall meaning of the text. The code reads a list of stop words from the 'stopwords.txt' file. It then applies the ***RemoveStops*()** function to remove stop words from the 'Text' column. The ***RemoveStops*()** function removes special characters, converts the text to lowercase, and removes the stop words from the text.

1.2 Word Dictionary

To convert the tokenized sequences into integer sequences, a word dictionary is created. The word dictionary maps unique words to their corresponding indices. The steps involved are as follows:

1.2.1 Get Unique Words:

The code collects all the unique words from the dataset by iterating over the *'Padded_Text'* column.

Remove Duplicates and Sort:

The code removes duplicate words from the list of unique words and sorts them in alphabetical order.

1.2.2 Create Word Dictionary:

The code creates a dictionary of unique words where each word is a key and its corresponding index is the value. This is achieved by iterating over the unique word list and assigning an index to each word.

The number of Unique Words:

The code prints the number of unique words in the dataset, indicating the size of the word dictionary.

1.3 Split dataset to train and test

The dataset is split into training and testing sets. The last 150 texts from each class are reserved for the test dataset. The remaining data is used for training.

I used a function called ***split_train_test_data*** that is used to split the input data into training and testing sets. The function takes three parameters:

- ***X***: The input features or data that will be split into training and testing sets.
- ***labels***: The corresponding labels for the input data.
- ***test_size*** The number of instances to be included in the testing set for each class.

The function performs the following steps:

It creates a dictionary of unique classes from the ***labels*** array using the ***np. unique*** function. Each class is assigned a unique index using a dictionary comprehension (***label_map***).

It creates a new array **y** that maps each label in **labels** to its corresponding index in the ***label_map***.

It initializes an empty list of ***test_data*** to store the data for the testing set.

For each class in the **classes** array, it retrieves the instances from **X** and **y** that belong to that class. It selects the last ***test_size*** instances from each class and appends them to *the* ***test_data*** list.

It initializes an empty list ***train_data*** to store the data for the training set.

It iterates over each instance in **X** and checks if it exists in the ***test_data*** list based on the input features ***(X[i])*** and the corresponding label (**y[i]**). If the instance is not found in the ***test_data*** list, it appends it to the ***train_data*** list.

It uses the **zip** function to separate the input features and labels for the training set and testing set, storing them in **X_train**, **y_train**, **X_test**, and **y_test**.

Finally, it converts the lists ***X_train**, **y_train**, **X_test**, and **y_test*** to *NumPy* arrays using the ***np.array*** function and returns them.

1.4 One Hot Encoding

The integer sequences of both the training and testing sets are converted to one-hot encoded vectors.

The one-hot encoding represents each word in a sequence as a binary vector, with a dimension equal to the vocabulary size. Unknown words, not present in the dictionary, are handled by assigning a separate vector.

I used a function called **ThreeDim_one_hot_encoder** that performs one-hot encoding on a dataset. The function takes two parameters:

- ***dataset***: A 2D numpy array containing the dataset to be encoded. Each row represents a sample, and each column represents a word or token in the sample.
- ***dictionary***: A dictionary that maps each unique word in the dataset to an index.

The function performs the following steps:

- It determines the maximum length of the dataset.
- It determines the vocabulary size based on the maximum index value in the **dictionary**.
- It initializes an array **one_hot_word_array** with dimensions **(dataset.shape[0], max_length, vocab_size)**.
- It iterates over each sample and each word in the dataset.
- It retrieves the word at the current position.
- If the word is not an empty string, it checks if the word exists in the **dictionary**.
- If the word is in the **dictionary**, it retrieves the corresponding index from the **dictionary** and sets the corresponding element in **one_hot_word_array** to 1.
- If the word is not in the **dictionary**, it handles the unknown word by setting the element at the first index in **one_hot_word_array** to 1.
- After iterating over all samples and words in the dataset, it returns the **one_hot_word_array** containing the one-hot encoded representation of the dataset.

The output of this function is a three-dimensional *numpy* array where each element represents a word in the dataset, encoded as a one-hot vector. The dimensions of the array are **(number of samples, maximum length of the dataset, and vocabulary size)**.

2 Elman Recurrent Neural Networks

Elman RNN, named after its inventor Jeffrey Elman, is a type of recurrent neural network (RNN) architecture that is widely used for processing sequential data. It is also known as a SimpleRNN or a Vanilla RNN.

The Elman RNN consists of recurrent connections that allow information to be passed from one step to the next sequentially. It is particularly suitable for tasks where the current input depends not only on the current step but also on the previous steps of the input sequence.

The basic working principle of an Elman RNN involves three main components:

Input Layer: The input layer receives the input at each time step of the sequence. It can be a single value or a vector representing the features of the input.

Hidden Layer: The hidden layer in an Elman RNN stores the memory or the hidden state. It takes input from the previous hidden state and the current input, processes them, and produces an output. This hidden state is passed along to the next time step, allowing the network to retain information about the past inputs.

Output Layer: The output layer generates the output at each time step based on the hidden state. It can be a single value or a vector representing the predicted output.

During training, the parameters of an Elman RNN, including the weights and biases, are updated using a process called backpropagation through time (BPTT). BPTT extends the traditional backpropagation algorithm to handle sequences by unrolling the network through time and propagating the error gradients backward from the output to the input.

Elman RNNs have been widely used in various sequential data processing tasks, such as natural language processing, speech recognition, and time series analysis. However, they suffer from the vanishing gradient problem, which makes it challenging for the network to capture long-term dependencies in the input sequence.

To overcome this limitation, more advanced RNN architectures like Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) have been developed. These architectures incorporate additional gates and memory cells to allow for better capturing and preserving of long-term dependencies.

In summary, Elman RNNs are a type of recurrent neural network architecture that can process sequential data by maintaining a hidden state that captures information from previous steps. They have been widely used for various tasks but may struggle with capturing long-term dependencies.

This part utilizes *SimpleRNN* in a classification task. Here's a breakdown of the code:

The labels are extracted from a data frame and stored in the ***labels*** variable. The unique classes in the labels are then obtained and stored in the ***classes*** variable.

A Sequential model is created, which is a linear stack of layers in *Keras*.

The model architecture is defined by adding layers to the Sequential model. In this case, a *SimpleRNN* layer is added with 64 units. The ***input_shape*** parameter specifies the shape of the input sequences, which is determined by the dimensions of the training data (***X_train.shape[1]*** *and* ***X_train.shape[2]***).

After the *SimpleRNN* layer, a Dense layer is added with the number of units equal to the number of classes. The activation function used is *softmax*, which produces class probabilities.

The model is compiled by specifying the loss function, optimizer, and metrics to be used during training. In this code, the loss function is set to ***'sparse_categorical_crossentropy'***, which is suitable for multi-class classification problems. The optimizer used is Adam, a popular optimization algorithm, and the metric chosen is accuracy.

The model is trained using the ***fit*** method, which takes the training data ***(X_train*** *and* ***y_train)*** as input. It is trained for 10 epochs with a batch size of 32. The validation *data* ***(X_test*** *and* ***y_test)*** is also provided to monitor the model's performance during training.

After training, the model is evaluated on both the training and test data using the ***evaluate*** method. The resulting loss and accuracy values for both datasets are stored in variables.

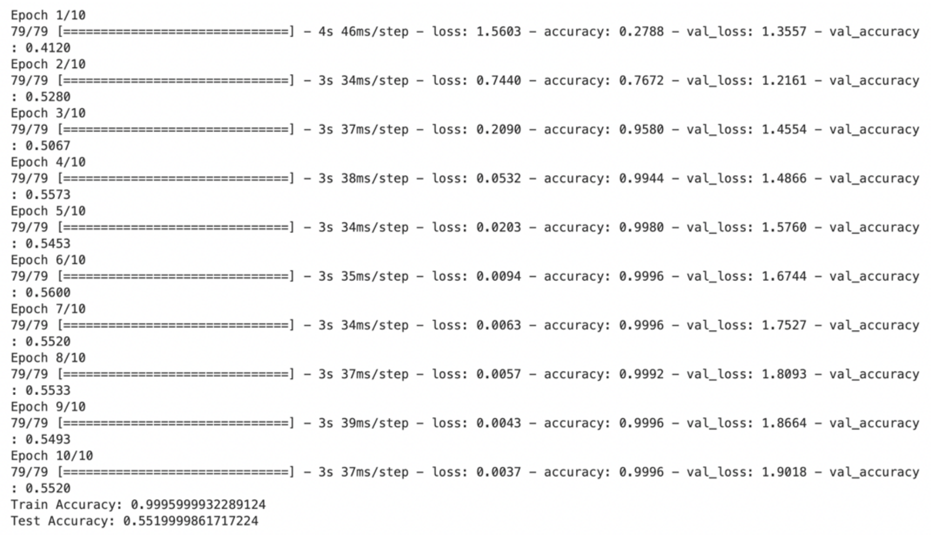
Finally, the train and test accuracies are printed to evaluate the performance of the trained model.

Overall, this code implements a *SimpleRNN*-based model for classification, trains it on the given data, and evaluates its performance using accuracy as the metric.

3 Results

The has run many times, here are the results of a few times running :

3.1 Run1



```
Epoch 1/10
79/79 [==============================] - 4s 46ms/step - loss: 1.5603 - accuracy: 0.2788 - val_loss: 1.3557 - val_accuracy
: 0.4120
Epoch 2/10
79/79 [==============================] - 3s 34ms/step - loss: 0.7440 - accuracy: 0.7672 - val_loss: 1.2161 - val_accuracy
: 0.5280
Epoch 3/10
79/79 [==============================] - 3s 37ms/step - loss: 0.2090 - accuracy: 0.9580 - val_loss: 1.4554 - val_accuracy
: 0.5067
Epoch 4/10
79/79 [==============================] - 3s 38ms/step - loss: 0.0532 - accuracy: 0.9944 - val_loss: 1.4866 - val_accuracy
: 0.5573
Epoch 5/10
79/79 [==============================] - 3s 34ms/step - loss: 0.0203 - accuracy: 0.9980 - val_loss: 1.5760 - val_accuracy
: 0.5453
Epoch 6/10
79/79 [==============================] - 3s 35ms/step - loss: 0.0094 - accuracy: 0.9996 - val_loss: 1.6744 - val_accuracy
: 0.5600
Epoch 7/10
79/79 [==============================] - 3s 34ms/step - loss: 0.0063 - accuracy: 0.9996 - val_loss: 1.7527 - val_accuracy
: 0.5520
Epoch 8/10
79/79 [==============================] - 3s 37ms/step - loss: 0.0057 - accuracy: 0.9992 - val_loss: 1.8093 - val_accuracy
: 0.5533
Epoch 9/10
79/79 [==============================] - 3s 39ms/step - loss: 0.0043 - accuracy: 0.9996 - val_loss: 1.8664 - val_accuracy
: 0.5493
Epoch 10/10
79/79 [==============================] - 3s 37ms/step - loss: 0.0037 - accuracy: 0.9996 - val_loss: 1.9018 - val_accuracy
: 0.5520
Train Accuracy: 0.9995999932289124
Test Accuracy: 0.5519999861717224
```

Table1 summarizing the results

Epoch	**Training Loss**	**Training Accuracy**	**Validation Loss**	**Validation Accuracy**
1	1.5603	0.2788	1.3557	0.4120
2	0.7440	0.7672	1.2161	0.5280
3	0.2090	0.9580	1.4554	0.5067
4	0.0532	0.9944	1.4866	0.5573
5	0.0203	0.9980	1.5760	0.5453
6	0.0094	0.9996	1.6744	0.5600
7	0.0063	0.9996	1.7527	0.5520
8	0.0057	0.9992	1.8093	0.5533
9	0.0043	0.9996	1.8664	0.5493
10	0.0037	0.9996	1.9018	0.5520

Train Accuracy: 0.9995999932289124

Test Accuracy: 0.5519999861717224

This table provides a concise overview of the training and validation metrics, including loss and accuracy, for each epoch. The final row represents the evaluation results on the train and test datasets.

Table2 Train Classification Report

Class	Precision	Recall	F1-Score	Support
0	1.00	1.00	1.00	500
1	1.00	1.00	1.00	500
2	1.00	1.00	1.00	500
3	1.00	1.00	1.00	500
4	1.00	1.00	1.00	500
-	1.00	1.00	1.00	2500

Table3 Test Classification Report

Class	Precision	Recall	F1-Score	Support
0	0.57	0.54	0.55	150
1	0.47	0.56	0.51	150
2	0.35	0.27	0.31	150
3	0.51	0.50	0.51	150
4	0.45	0.49	0.47	150
-	0.47	0.47	0.47	750

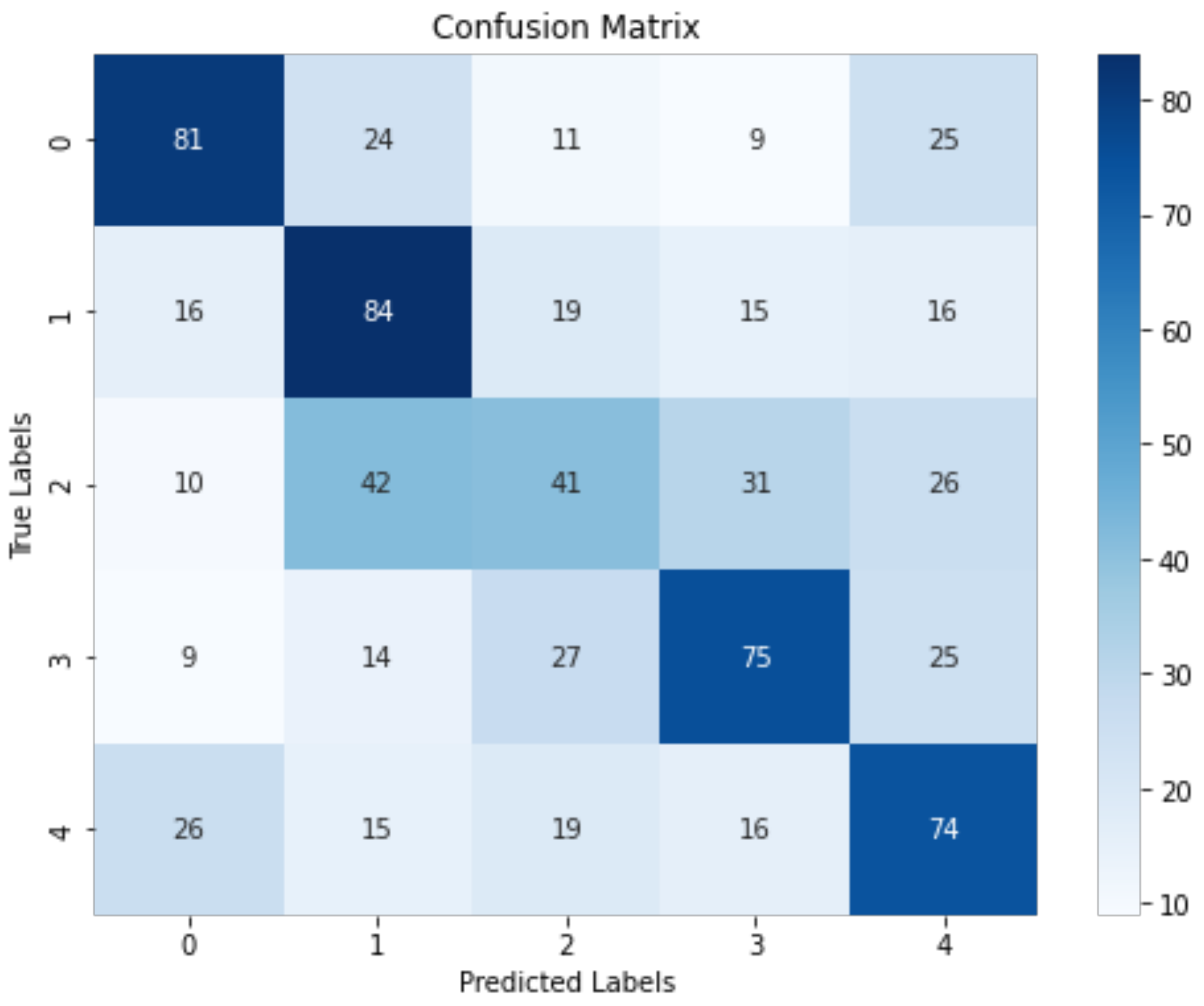
The classification reports provide metrics such as precision, recall, F1-score, and support for each class in the training and test datasets. The table presents these metrics for each class, as well as the macro average and weighted average metrics.

In the train classification report, all classes achieved perfect precision, recall, and F1-scores of 1.00. The support column represents the number of samples for each class. The overall accuracy for the train dataset is 1.00.

In the test classification report, the metrics vary across the classes. The precision, recall, and F1-scores range from 0.35 to 0.57, indicating different levels of performance for each class. The support column represents the number of samples for each class. The overall accuracy for the test dataset is 0.47.

The macro average and weighted average metrics provide an overall summary of the model's performance across all classes. The macro average takes the average of the metrics for each class, while the weighted average considers the support (sample count) for each class.

These classification reports give insights into the model's performance on both the training and test datasets, highlighting any variations in accuracy and performance across different classes.



Confusion Matrix of Run1

3.2 Run 2

Table4 summarizing the results

Epoch	**Training Loss**	**Training Accuracy**	**Validation Loss**	**Validation Accuracy**
1	1.5648	0.2648	1.3834	0.4267
2	0.7392	0.7708	1.3413	0.4893
3	0.1418	0.9744	1.4130	0.5093
4	0.0271	0.9984	1.5490	0.5400
5	0.0108	0.9992	1.5951	0.5320
6	0.0072	0.9996	1.6905	0.5360
7	0.0052	0.9992	1.7665	0.5333
8	0.0037	0.9996	1.7778	0.5213
9	0.0044	0.9992	1.7914	0.5280
10	0.0030	0.9992	1.8542	0.5253

The table provides information on the training and validation performance of the model for each epoch.

The training loss and accuracy represent the model's performance on the training data, while the validation loss and accuracy indicate its performance on the validation data.

In this case, the training loss decreases gradually from 1.5648 to 0.0030, indicating that the model is improving its ability to fit the training data. The training accuracy increases from 0.2648 to 0.9992, indicating that the model is getting better at correctly classifying the training samples.

The validation loss fluctuates throughout the epochs, with a slight increase towards the end. This suggests that the model may be starting to overfit the training data, as it is not generalizing well to the validation data. The validation accuracy ranges from 0.4267 to 0.5400, with a similar pattern of fluctuation.

The final training accuracy is 0.9996, indicating that the model achieves a high accuracy on the training data. The test accuracy is 0.5253, suggesting that the model's performance on unseen test data is relatively lower compared to the training data.

Overall, the model shows good training performance, with high accuracy and decreasing loss. However, it seems to struggle with generalization to unseen data, as indicated by the lower test accuracy and the fluctuating validation metrics. Further analysis and possibly adjustments to the model or training process may be necessary to improve its performance on unseen data.

Table5 summarizes the classification reports for both the training and test sets

	Precision	**Recall**	**F1-Score**	**Support**
Train Set				
Class 0	1.00	1.00	1.00	500
Class 1	1.00	1.00	1.00	500
Class 2	1.00	1.00	1.00	500
Class 3	1.00	1.00	1.00	500
Class 4	1.00	1.00	1.00	500
Macro Avg	1.00	1.00	1.00	2500
Weighted Avg	1.00	1.00	1.00	2500
Test Set				
Class 0	0.54	0.61	0.57	150
Class 1	0.53	0.53	0.53	150
Class 2	0.47	0.31	0.38	150
Class 3	0.57	0.72	0.64	150
Class 4	0.48	0.46	0.47	150
Macro Avg	0.52	0.53	0.52	750
Weighted Avg	0.52	0.53	0.52	750

3.3 Best run:

The best run gave 74% of accuracy but it happened one time and didn’t repeat.

```
Epoch 2/20
65/65 [==============================] - 0s 3ms/step - loss: 1.5654 - accuracy: 0.3303 - val_loss: 1.5492 - val_accuracy:
0.4808
Epoch 3/20
65/65 [==============================] - 0s 3ms/step - loss: 1.4170 - accuracy: 0.4837 - val_loss: 1.3619 - val_accuracy:
0.6154
Epoch 4/20
65/65 [==============================] - 0s 3ms/step - loss: 1.1065 - accuracy: 0.6264 - val_loss: 1.0847 - val_accuracy:
0.7077
Epoch 5/20
65/65 [==============================] - 0s 3ms/step - loss: 0.7874 - accuracy: 0.7423 - val_loss: 0.8697 - val_accuracy:
0.7442
Epoch 6/20
65/65 [==============================] - 0s 3ms/step - loss: 0.5616 - accuracy: 0.8231 - val_loss: 0.7451 - val_accuracy:
0.7769
Epoch 7/20
65/65 [==============================] - 0s 3ms/step - loss: 0.4108 - accuracy: 0.8913 - val_loss: 0.6775 - val_accuracy:
0.7769
Epoch 8/20
65/65 [==============================] - 0s 3ms/step - loss: 0.3011 - accuracy: 0.9192 - val_loss: 0.6383 - val_accuracy:
0.7904
Epoch 9/20
65/65 [==============================] - 0s 3ms/step - loss: 0.2453 - accuracy: 0.9317 - val_loss: 0.6153 - val_accuracy:
0.7846
Epoch 10/20
65/65 [==============================] - 0s 3ms/step - loss: 0.1961 - accuracy: 0.9476 - val_loss: 0.6206 - val_accuracy:
0.7942
Epoch 11/20
65/65 [==============================] - 0s 3ms/step - loss: 0.1648 - accuracy: 0.9558 - val_loss: 0.6270 - val_accuracy:
0.7846
Epoch 12/20
65/65 [==============================] - 0s 3ms/step - loss: 0.1473 - accuracy: 0.9572 - val_loss: 0.6282 - val_accuracy:
0.7904
65/65 [==============================] - 0s 1ms/step - loss: 0.0105 - accuracy: 1.0000
21/21 [==============================] - 0s 1ms/step - loss: 0.7483 - accuracy: 0.7415
Train Loss: 0.0104924151673913
Train Accuracy: 1.0
Test Loss: 0.7483240962028503
Test Accuracy: 0.7415384650230408
```

4 Question part

How this problem can be solved using Hopfield recurrent network? Explain your idea.

A Hopfield network is a type of recurrent neural network (RNN) that can store and recall patterns. It consists of a set of interconnected nodes or neurons, where each neuron can be in one of two states: "on" or "off". The connections between neurons have weights that determine the influence of one neuron on another. The network updates its state iteratively until it reaches a stable state. There is a way may we can do it on Hopfield Net that work well :

Encoding Emotion States: Assign a binary representation to each emotion class, such as "happy" as [1, 0, 0, 0, 0], "sad" as [0, 1, 0, 0, 0], and so on. This encoding scheme represents the desired stable states in the Hopfield network.

Creating Memory Patterns: Use the training dataset to create memory patterns for each emotion class. Each memory pattern represents a particular emotion. For each input text, convert it into a binary vector representation based on the presence or absence of specific words or features associated with each emotion class.