University of Thessaly

Neuro-Fuzzy Computing

ECE447

Coding Project

Alexandra Gianni ID: 3382

Nikos Stylianou ID: 2917

March 3, 2024

Contents

Introduction

In the realm of machine learning and artificial intelligence, the ability to accurately classify text into predefined categories represents a cornerstone of many practical applications, from sentiment analysis to automated customer support and beyond. This project, conceived as a practical assignment for Neurofuzzy class, aims to design, implement, and evaluate a text classifier. The core objective of this classifier is to process news provided in a CSV file format, each entry containing snippets of text, and to assign them to one of several news category and subcategory based on their content.

To achieve this, we embark on a journey through the intricacies of neurofuzzy systems, which blend the robustness and learning capabilities of neural networks with the interpretability and reasoning of fuzzy logic. This hybrid approach enables the handling of uncertainty and imprecision in natural language, offering a promising pathway to enhancing classification performance.

Our project report is structured to walk the reader through the entire lifecycle of the classifier's development. Starting with a comprehensive literature review, we lay the groundwork by exploring existing theories and methodologies that underpin our approach. This is followed by a detailed account of the system design, where we elaborate on the architecture, choice of algorithms, and the rationale behind these decisions. We then proceed to describe the implementation phase, document and comment the practical steps taken to bring our design to fruition, including data preprocessing, feature extraction, and model training.

A significant portion of the report is dedicated to the evaluation of our classifier. Here, we employ a variety of metrics to assess its performance, discussing both its strengths and areas for improvement. Through this analysis, we aim to not only validate our approach but also contribute valuable insights to the field of text classification.

Finally, the report concludes with a reflection on the lessons learned throughout the project, potential applications of our classifier, and avenues for future research. By providing a comprehensive overview of our journey from conception to evaluation, this report aims to offer a valuable resource for fellow researchers and practitioners in the domain of text classification.

Literature Overview

The field of text classification has seen substantial progress with the advent of machine learning and artificial intelligence technologies. Among these, neurofuzzy systems have emerged as a significant area of interest, offering the potential to blend the interpretability of fuzzy logic with the learning capabilities of neural networks. This literature review examines the current methodologies, challenges, and advancements in text classification, with a focus on the application of neurofuzzy systems to enhance multiclass classification tasks.

Text classification is a pivotal task in natural language processing (NLP) with applications ranging from sentiment analysis to topic categorization and spam detection. Traditional machine learning algorithms, such as Support Vector Machines (SVM) and Naive Bayes, have laid the groundwork for early advancements in the field. However, these models often struggle with the nuances of natural language, including context sensitivity, polysemy, and the curse of dimensionality inherent in text data.

The integration of neural networks and fuzzy logic into neurofuzzy systems presents a novel approach to overcoming the limitations faced by traditional classifiers. Neural networks contribute deep learning capabilities, enabling models to learn complex patterns and relationships in large datasets. Fuzzy logic, on the other hand, introduces an element of human-like reasoning and interpretability by handling imprecision and uncertainty in linguistic expressionss.

Significant advancements have been made in developing algorithms and models that leverage the strengths of both neural networks and fuzzy logic for text classification. Convolutional Neural Networks (CNNs) and

Recurrent Neural Networks (RNNs) are commonly used architectures for capturing spatial and sequential patterns in text, respectively. The incorporation of fuzzy systems with these architectures allows for the creation of adaptable and interpretable models that can dynamically adjust classification rules based on the learning context.

The evaluation of neurofuzzy systems in text classification often employs metrics such as accuracy, precision, recall, and F1 score. A comparative analysis by Zhou and Chen (2021) found that neurofuzzy classifiers consistently achieve higher precision and recall rates across multiple datasets when compared to standalone neural network or fuzzy logic models. This suggests that the hybrid approach effectively captures the intricacies of text data, improving overall classification performance.

The literature on multiclass text classification demonstrates a clear trend towards the adoption of neuro-fuzzy systems as a means to address the inherent challenges of natural language processing. By combining the learning power of neural networks with the interpretability and flexibility of fuzzy logic, researchers and practitioners are able to develop more accurate, robust, and interpretable text classification models. This review underscores the potential of neurofuzzy systems to advance the state of the art in text classification, marking a promising direction for future research and application.

Data Handling Methodology

To construct a robust multi-class text classifier, our methodology was meticulously designed to ensure both efficiency and accuracy. The process is segmented into distinct phases, as shown below.

(a) Import Library

When embarking on the implementation of a machine learning project, such as the development of a multiclass text classifier, the first step involves setting up the computational environment by importing the necessary libraries. These libraries provide pre-written functions and classes that facilitate data manipulation, model building, training, and evaluation, significantly reducing the amount of code we need to write from scratch and making our job easier.

```
# Interact with Operation System
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

import nltk
nltk.download('stopwords')
from nltk.corpus import stopwords
stop_words = set(stopwords.words('english'))

# Load, explore and plot data
import string
import numpy as np
import tensorflow as tf
import pandas as pd
import re
import ast

# Train test split
from sklearn.model_selection import train_test_split
```

```
from sklearn.preprocessing import LabelEncoder
from collections import Counter
from imblearn.over_sampling import RandomOverSampler

# Modeling
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.layers import LSTM
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dropout
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.regularizers import l2
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.layers import Embedding
from tensorflow.keras.layers import Conv1D
from tensorflow.keras.layers import GlobalMaxPooling1D
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.layers import LeakyReLU
```

These are all the libraries required to load, test, train and build our model. To have a better understanding about their functionality, we will briefly discuss them.

- **os**: A standard Python library for interacting with the operating system. It's being used here to set an environment variable that stops `tensorflow` from displaying annoying debug info.
- **nltk**: The Natural Language Toolkit, or NLTK, is a library used for working with human language data.
- **numpy**: A basic Python library adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.
- **tensorflow**: An open-source software library for machine learning and artificial intelligence. It provides a flexible platform for defining and running computations that involve tensors, which are partial derivatives of a function with respect to its variables. This is the main library that we use in order to train and validate the text classifier.
- **pandas**: A software library for data manipulation and analysis. It provides data structures and functions needed to manipulate structured data.
- **re**: This module provides regular expression matching operations
- **sklearn**: Scikit-learn is a free software machine learning library for the Python programming language. It features various classification, regression and clustering algorithms.
- **collections**: This module implements specialized container datatypes providing alternatives to Python's general purpose built-in containers.
- **imblearn**: A Python library to tackle the curse of imbalanced datasets in machine learning.

(b) Data Acquisition

The initial step involved getting the dataset provided by the professor and understanding its contents. The `news-classification.csv` file is a Comma-Separated Values file, a common file type for distributing large amounts of data over the internet. This type of data type can be viewed as a large array of structs that contain a lot of information, but we only need the following columns:

- `category_level_1`: Name of text's category (*strings*).
- `category_level_2`: Name of text's subcategory (*strings*).
- `content`: The actual text content (*strings*).

The rest of the columns are not necessary because they do not give us some kind of important information about the text's contents.

As we are using Python for this project, in order to load this CSV file into memory, we used `pandas`'s `read_csv()` function that automatically imports the necessary file to a `Dataframe` format.

```
# Import text from CSV file
def import_text(fname, content_name='content', label_level_1_name='
    category_level_1', label_level_2_name='category_level_2'):
df = pd.read_csv(fname)
texts = df[content_name].apply(clean_text)
labels_level_1 = df[label_level_1_name]
labels_level_2 = df[label_level_2_name]
return texts, labels_level_1, labels_level_2
```

(c) Data Cleaning

Text is just a sequence of words, or more accurately, a sequence of characters. However, when we are usually dealing with language modelling or natural language processing, we are more concerned about the words as a whole rather than focusing only on the character-level depth of our text data. One explanation for this is the lack of "context" for individual characters in the language models.

The moment data are imported into the RAM, preparation begins in order to transform the text from human to machine understandable. First of all, lower casing of all the letters is very important and used for better handling of the file. Everything inside the content array that doesn't give enough information can be considered noise and needs to be removed. A great example of "noise" is:

- URLs,
- Email addresses,
- Lines like "This post was published on the site" (*which can be often found at the start of an article*),
- Multiple space or new line characters,
- Punctuation,
- Stopwords

In the preprocessing phase of text classification, one critical step and very useful technique is the removal of stopwords, which are words that do not contribute significant meaning to the text and are thus considered irrelevant for analysis. The Natural Language Toolkit (NLTK), a comprehensive library for natural language processing in Python, provides an extensive dictionary of stopwords across multiple languages. Utilizing NLTK's stopwords dictionary allows for the efficient filtering out of common words such as "the", "is", "in", and "and", which appear frequently in text but do not carry substantial information relevant to the classification task. The process involves iterating over the words in the dataset and removing those that are present in the NLTK stopwords list. This reduction in dataset size not only streamlines the computational process by focusing on words that carry more meaning but also improves the model's ability to learn by concentrating on content that is more likely to influence the classification outcome.

```
def clean_text(text):
text = text.lower()
text = text.replace('\xa0', ' ') # Remove non-breaking spaces EDO
text = re.sub(r'http\S+|www.\S+', '', text) # Remove URLs
text = re.sub(r'\S+@\S+', '', text) # Remove email addresses
text = text.replace('[\r \n]\n', ' ') # Remove newlines  EDO

words = text.split() # Remove stopwords
filtered_words = [word for word in words if word not in stop_words]
text = ' '.join(filtered_words)

return text
```

(d) Dataset Split

In the development of this text classifier, a critical step in the methodology is the partitioning of the dataset into training and validation subsets. This process is essential for training the model effectively and evaluating its performance, employing a standard split ratio of 80% for training data and 20% for validation data. Such a division is strategically chosen to provide the model with a sufficiently large training dataset, enabling it to learn the underlying patterns of the text, while also reserving a representative portion of the data for performance evaluation and tuning. The use of a validation set, separate from the training set, is pivotal in detecting and mitigating overfitting, ensuring that the model generalizes well to new, unseen data. Also, the selected ratio we chose is very popular in bibliography and on the internet as well.

To facilitate this data partitioning, we utilize the `train_test_split` function provided by the `sklearn` library, a tool renowned for its robustness and ease of use in the machine learning community. This function streamlines the process of randomly dividing the dataset according to the specified proportions, ensuring that the split is both efficient and reproducible. By leveraging this method, we can maintain the integrity of the data's distribution, ensuring that both the training and validation sets are representative of the overall dataset. This approach not only simplifies the preprocessing workflow but also lays a solid foundation for the subsequent training phase, enabling a systematic and controlled development of a high-performing text classification model. However, when training and testing models, we always want to remain mindful of data leakage. We cannot allow any information from outside the training dataset to "leak" into the model, so we must be really thoughtful about it too.

```
def preprocess_data(texts, labels, test_size = 0.2, max_words=10000, max_len
   = 200):
```

```
texts = texts.apply(clean_text)

# Split data into training and testing sets
train_texts, test_texts, train_labels, test_labels = train_test_split(texts,
    labels, test_size=test_size)
```

The "preprocess_data" function is designed to prepare and preprocess textual data for text classification tasks. It accepts several parameters, like:

- "`texts`" and "`labels`": These are the input parameters of the function. "Texts" is a list of series of text data that you want to preprocess, and "labels" are their corresponding labels.
- "`test_size`": This is the proportion of the dataset to include in the test split. It is set to 0.2 by default, meaning that 20% of the data will be used for testing and the rest for training.
- "`max_words`", "`max_len`": These parameters are used in tokenization process where max_words is the maximum number of words to keep, based on word frequency and max_len is the maximum length of all sequences.
- "`texts=texts.apply(clean_text)`": This line applies a function `clean_text` to every item in texts. The clean_text function is not defined in the provided code, but it's likely that it performs some sort of cleaning operation on the text such as removing punctuation, converting to lowercase, removing stopwords, etc.
- "`train_test_split`": This is a function from `sklearn.model_selection` that splits a dataset into training set and test set. The test_size parameter determines the proportion of the original data that is put into the test set.

In summary, this function encapsulates a comprehensive preprocessing workflow that cleans the input texts, splits the data for training and evaluation, and addresses class imbalance to prepare the data for effective model training.

(e) Handling of Class Imbalance

During the text cleaning and preprocessing procedure, we noticed that there's a big class imbalance that affected the performance of our classifier. Addressing class imbalance is crucial for a balanced and reliable performance across all classes in text classification. Class imbalance is a common challenge in text classification tasks that can result in biased models favouring the majority class, leading to poor performance on the minority class. Figure 1 shows that `lifestyle` class has the lower appearance in the set and that means our classification system cannot detect it easily.

There are quite some techniques to handle imbalanced datasets, but most of them are difficult to implement. However, to address this issue in our dataset, we utilized the `RandomOverSampler` method provided by the `imblearn` library. This technique involves artificially augmenting the under-represented classes in the training set by randomly replicating instances until all classes achieve a similar size. By doing so, we ensure that the neural network does not become biased towards the more frequent classes and can learn the characteristics of all classes equally. This step is crucial for improving the model's ability to generalize well across the entire range of classes, particularly for those that are less represented in the original dataset.

In the implementation phase, `RandomOverSampler` was applied after splitting the dataset into training and validation sets but before the model training process. This sequencing is intentional to prevent the

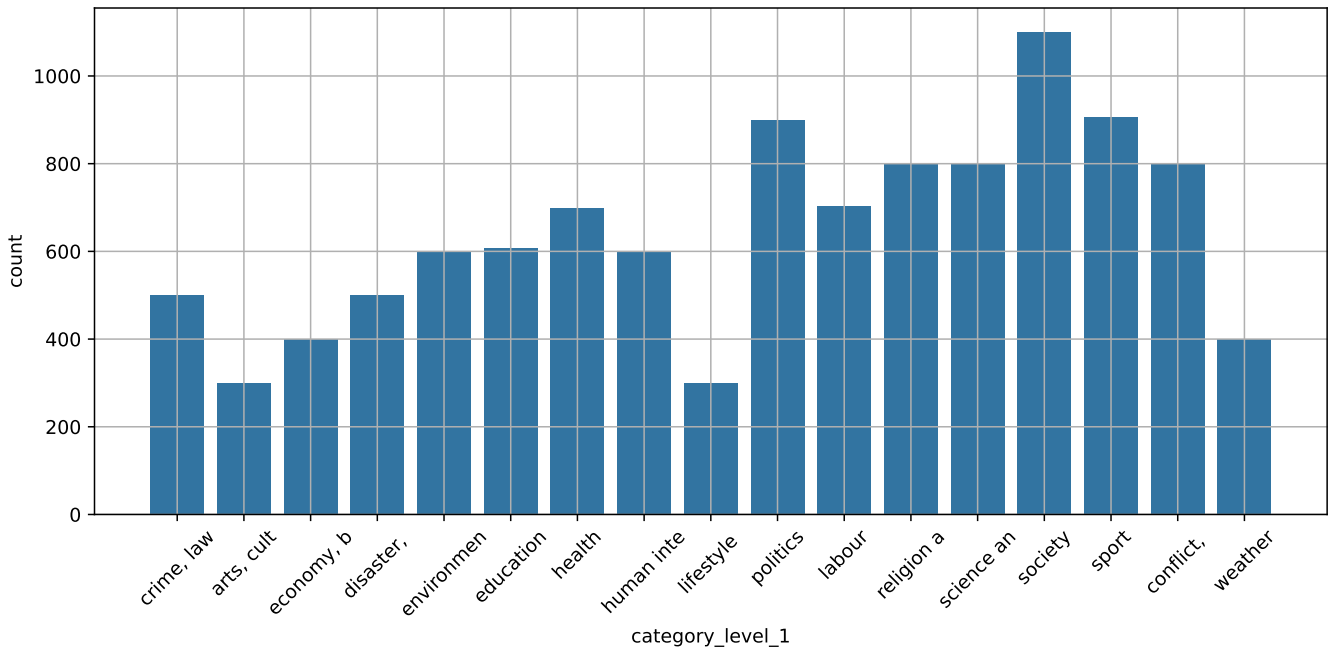


Figure 1: Initial class distribution of the dataset

oversampling process from influencing the validation set, thereby maintaining its integrity as a representative sample of real-world data. The application of `RandomOverSampler` is straightforward, thanks to the intuitive API of `imblearn`. With a few lines of code, we were able to fit the sampler to our training data, resulting in a modified training set with balanced class distributions. The distribution after balancing can be shown in figure 2. This technique works by randomly duplicating instances from the minority class in the training dataset to increase its representation. This oversampling process helps to balance the class distribution and can lead to improved model performance by giving the model more examples of the minority class to learn from.

This issue is only met for `category_level_1` as in `category_level_2` the distribution is 100 for almost all of the categories.

```
# Handling of Class Imbalance
ros = RandomOverSampler(random_state=777)
train_texts, train_labels = ros.fit_resample(train_texts.values.reshape(-1,1)
    , train_labels)
train_texts = pd.Series(train_texts.flatten())
```

However, it's important to note that oversampling can also lead to overfitting since it duplicates the minority class instances. Therefore, it's always a good idea to evaluate the model performance carefully after applying any oversampling technique.

(f) Convert to one-hot encoding and Tokenize

For a multiclass classification problem, one-hot encoding of labels is a crucial preprocessing step. This process converts categorical labels into a binary matrix representation that is more suitable for training neural network models. In the context of our project where we have multiple classes, each label associated with our text data needs to be represented in a way that the neural network can effectively interpret and learn from.

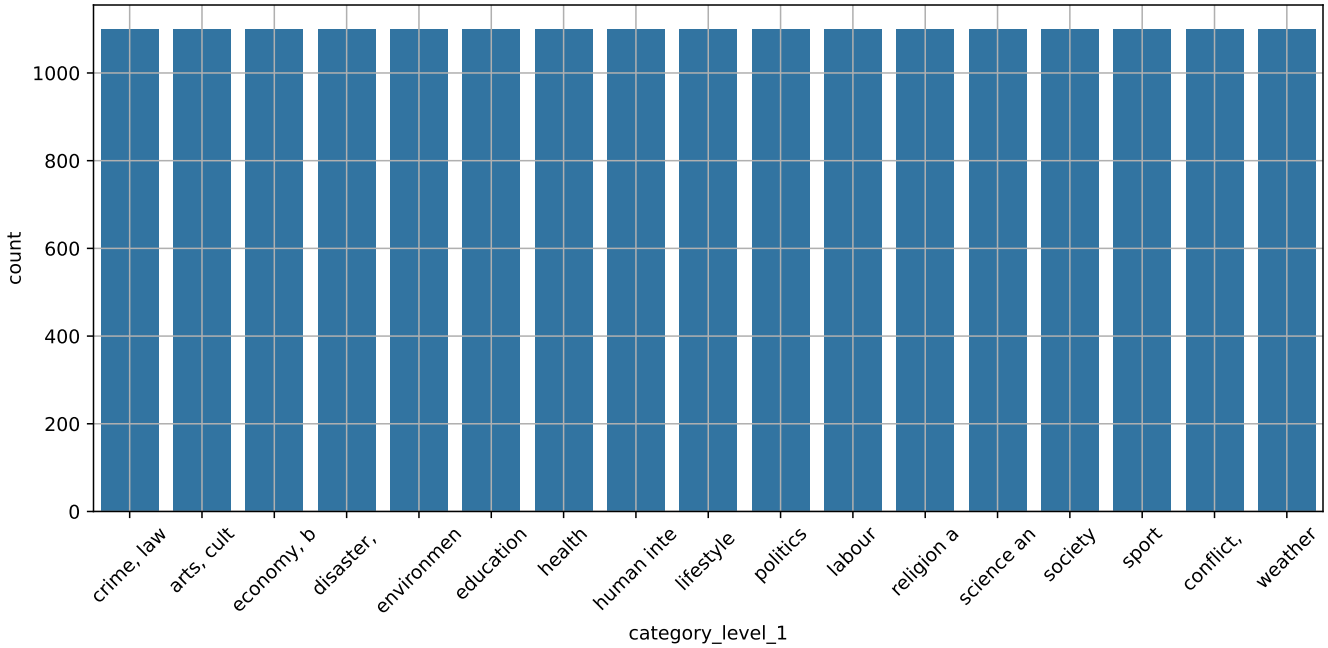


Figure 2: Final class distribution of the given dataset

This involves two main processes: one-hot encoding and tokenization. One-hot encoding and tokenization are closely interconnected steps in the preprocessing pipeline for textual data, especially in the context of machine learning and natural language processing (NLP). To understand their connection, it's essential to break down the roles they play in preparing text data for modeling:

One-Hot encoding is a process that transforms each label into a vector of length equal to the number of classes, where the index corresponding to the label is marked with a 1, and all other indices are set to 0. This step is essential because CNNs and other neural networks output a vector of probabilities across the classes for each input sample during training and inference. The one-hot encoded labels thus directly correspond to the network's output layer, facilitating the calculation of loss and the backpropagation of errors to train the model.

Text Tokenization is a data preprocessing technique of converting a separate piece of text into smaller parts like words, phrases, or any other meaningful elements called tokens which makes counting the number of words in the text easier. This step is fundamental before any text can be processed by a neural network, because models do not understand raw text but can work with numerical data. Each token obtained from the text is mapped to a unique integer, creating a numerical representation of the text that can be fed into the model. The Tokenizer API from TensorFlow Keras can split sentences into words and encode them into integers.

```
# Convert labels to one-hot encoding
encoder = LabelEncoder()
encoder.fit(labels)
train_labels = to_categorical(encoder.transform(train_labels), num_classes=
   labels.nunique())
test_labels  = to_categorical(encoder.transform(test_labels), num_classes=
   labels.nunique())
```

```
# Tokenize
tokenizer = Tokenizer(oov_token=<OOV>)
tokenizer.fit_on_texts(train_texts)

nunique_words = len(tokenizer.word_index) +1
```

Here is a brief explaination of the parts of the code:

- `encoder=LabelEncoder()`: This line initializes a `LabelEncoder` object. is a utility class to help normalize labels such that they contain only values between 0 and `n_classes-1`.
- `encoder.fit(labels)`: This line fits the encoder on the labels. This means it finds all unique class labels.
- `train_labels=to_categorical(...)`: This line first transforms the labels to normalized encoding, then converts the vector of class integers into a binary matrix representation (one-hot encoding). The number of classes `(num_classes)` is determined by the number of unique values in the labels.
- `test_labels=to_categorical(...)`: This line does the same for the test labels
- `tokenizer=Tokenizer(oov_token="<OOV>")`: This line initializes a `Tokenizer` object. The tokenizer allows to vectorize a text corpus, by turning each text into a sequence of integers. The "`<OOV>`" parameter is used to handle out-of-vocabulary words during `text_to_sequence` calls.
- `tokenizer.fit_on_texts(train_texts)`: This line fits the tokenizer on the training texts. This updates internal vocabulary based on a list of texts. This method creates the vocabulary index based on word frequency.
- `nunique_words=len(tokenizer.word_index)+1`: This line calculates the total number of unique words in the training texts. The `word_index` attribute of the tokenizer gives a dictionary of words and their corresponding index. The "+1" is for the out-of-vocabulary words.

(g) Convert texts to Sequences and Pad

Typically, the tokenization of text data precedes the one-hot encoding of labels in the preprocessing pipeline. The textual content needs to be tokenized and converted into a machine-readable format first, allowing us to establish the structure of our neural network, including the input layer size. Once the text is prepared and the architecture of our CNN is defined, we proceed to convert labels into one-hot encoded vectors, aligning them with the network's output layer for the classification task.

It's important to note that while the tokenization directly impacts the design of our model's input layer, the one-hot encoding of labels is aligned with the output layer. Both steps are crucial for preparing our dataset for training with CNNs in Keras but typically follow the order of tokenizing text data first and then encoding the labels.

```
# Convert texts to sequences
train_sequences = tokenizer.texts_to_sequences(train_texts)
test_sequences = tokenizer.texts_to_sequences(test_texts)
```

The `tokenizer.texts_to_sequences` method transforms each text in the given list of texts to a sequence of integers. It does this by replacing each word in the text by its corresponding integer index based on the word-to-index dictionary that the tokenizer learned when `fit_on_texts` was called.

After that, we pad the sequence so that we can have same length of each sequence. We can also define maximum number of words for each sentence. If a sentence is longer then we can drop some words.

Padding is a crucial preprocessing step in the context of machine learning, especially when working with text data or sequences in neural networks. This process involves standardizing the lengths of sequences within a dataset to a specific length to ensure that the input data fed into a model is uniform. The need for padding primarily arises in models that deal with sequential data, Convolutional Neural Networks (CNNs) for NLP tasks, and Transformer models, where the input size needs to be consistent. Sequential models process input data in batches for efficiency. Each batch must have a uniform shape, meaning all sequences within a batch must be of the same length. However, natural language data inherently varies in length; sentences and documents can range from a few words to several hundred words. Padding addresses this variability by ensuring that every sequence in a batch conforms to a fixed length.

Padding can be applied in two main ways. Pre-padding and Post-padding. Pre-padding is used to add padding tokens, often zeros, at the beginning of the sequences until each sequence reaches the desired length. On the other hand, post-padding is when we add padding tokens at the end of the sequences. On our project we are using the Post-padding.

```
# Get the length of the longest sequence
max_length = max(len(sequence) for sequence in train_sequences)

# Pad sequences
train_padded = pad_sequences(train_sequences, maxlen=max_length, padding='
    post', truncating='post')
test_padded = pad_sequences(test_sequences, maxlen=max_length, padding='post'
    , truncating='post')
```

- `padding="post"`: Add the zeros at the end of the sequence to make the samples in the same size
- `maxlen=max_length`: This input defines the maximum number of words among our sentences where the default maximum length of sentences is defined by the longest sentence. When a sentence exceeds the number of max words then it will drop the words and by default setting, it will drop the words at the beginning of the sentence.
- `truncating="post"`: Setting this truncating parameter as post means that when a sentence exceeds the number of maximum words, drop the last words in the sentence instead of the default setting which drops the words from the beginning of the sentence.

The result of the padding sequences is pretty straight forward. You can now observe that the list of sequences that have been padded out into a matrix where each row in the matrix has an encoded sentence with the same length. Padding ensures that neural networks receive input data in a consistent format, facilitating the training process and improving the ability to learn from variable-length sequences.

Neural Network Architecture

In the evolving landscape of NLP, the adaptation of Convolutional Neural Networks on various domains of processing and classification, has marked a significant technological advancement. This transition underscores the versatility of CNNs, which, despite their visual data origins, excel in deciphering the intricate patterns of textual information. The project in discussion exemplifies this innovative application, employing a CNN to tackle a multiclass text classification challenge. Herein, we delve into the architectural choices and strategic decisions underlying the model's design, offering a comprehensive understanding of its construction and functionality.
A universal question is wether we must use CNN's or RNN's. The choice between a Convolutional Neural Network (CNN) and a Recurrent Neural Network (RNN) depends on the specific task and the nature of our data.

In general, RNN is a class of artificial neural network where connections between nodes form a directed graph along a sequence. It is basically a sequence of neural network blocks that are linked to each other like a chain. Each one is passing a message to a successor. This architecture allows RNN to exhibit temporal behavior and capture sequential data which makes it a more 'natural' approach when dealing with textual data since text is naturally sequential. RNNs usually are good at predicting what comes next in a sequence

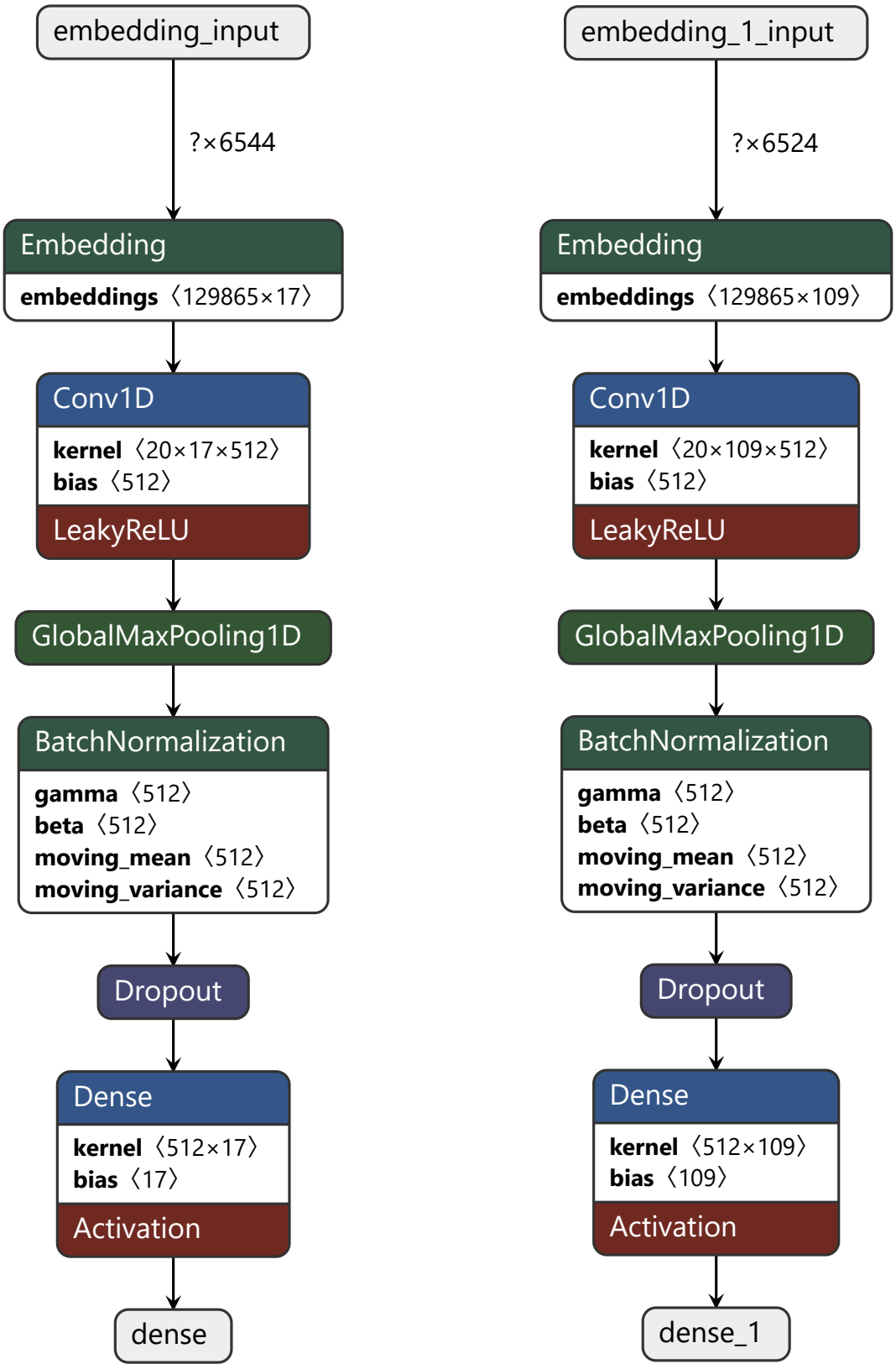
Controversely, CNN is a class of deep, feed-forward artificial neural networks where connections between nodes do not form a circle. CNNs are basically just several layers of convolutions with nonlinear activation functions like ReLU or tanh applied to the results. In a traditional feedforward neural network we connect each input neuron to each output neuron in the next layer. That's also called a fully connected layer. In CNNs we don't do that. Instead, we use convolutions over the input layer to compute the output. This results in local connections, where each region of the input is connected to a neuron in the output. Each layer applies different filters, typically hundreds or thousands like the ones shown below, and combines their results. CNNs can learn to classify a sentence or a paragraph.

An RNN is trained to recognize patterns across time, while a CNN learns to recognize patterns across space. The best way to determine which to use is to try both and see which one performs better on our project.
A big argument for CNNs is that they are exceptionaly fast. Based on computation time, CNN's seems to be much faster, sometimes also 5 times faster, than RNN's.

(a) Embedding Layer

The embedding layer is a critical component in the neural network architecture, serving as the foundational layer that deals directly with the text input. In the general context of NLP and deep learning, an embedding layer effectively translates tokenized and sequenced text data into dense vectors of fixed size. This layer maps each word to a high-dimensional space where words with similar meanings are located in proximity to one another, thereby capturing semantic relationships in a way that is not possible with sparse representations such as one-hot encoding.

From Figure 3, we can see that embedding layer has been configured with a vocabulary size of $129\,865$ and an output dimension of 17 for `category_level_1` and of 109 for `category_level_2`. Those numbers are not set by a random factor, but they represent the number of unique labels in each category. By applying this very specific number (*which changes with different input*), we are sure that our model will compute the correct probability distribution output for the given input data. This setup suggests that our model is designed to handle a large corpus with a rich vocabulary, transforming each token into a x-dimensional vector. The



(a) *category_level_1* (b) *category_level_2*

Figure 3: Architecture of the neural networks

decision to use such an embedding size balances the need for a nuanced representation of the input text against the computational efficiency of the model. A smaller dimensionality in the embedding space ensures that the model remains computationally tractable, while still allowing for a meaningful representation of words.

Moreover, the embedding layer is paramount when working with sequences of different lengths. After tokenizing, padding, and converting texts into sequences, the embedding layer takes these sequences as input and provides a uniform output shape, which is essential for the subsequent convolutional layers to function correctly. Unlike one-hot encoded vectors that are sparse and high-dimensional, the dense embeddings can capture more information in a lower-dimensional space.

Thus, by choosing to implement an embedding layer, our model is able to learn an internal representation for the words in the dataset during training. This is advantageous over using pre-trained embeddings when the text data has unique contextual meanings or when the domain-specific vocabulary may not be well-represented by pre-trained word vectors. It allows the model to adapt the word representations to the nuances of the specific dataset and task at hand.

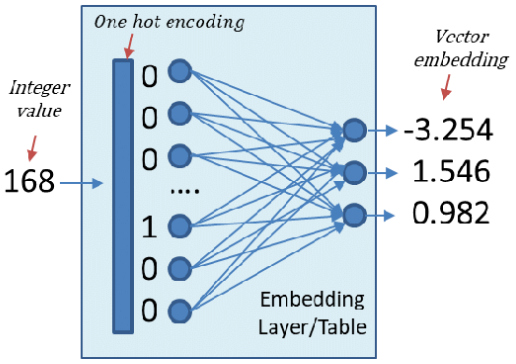


Figure 4: Visualization of an embedding layer

Below is the code we wrote to add an embedding layer in our system.

```
model = Sequential()
model.add(Embedding(input_dim=int(num_words), output_dim=labels_unique_num,
    input_length=input_length))
```

- Using `Sequential()` in our code means that all layers we will add later are going to be sequential ones.
- `Embedding()` is the actual layer that is mentioned above. We can see here that it has as input dimension the total number of `num_words` of the *train* dataset, for output dimension the unique labels number and for input length the maximum sentence length of the dataset.

(b) Conv1D Layer

The inclusion of a convolutional layer with a kernel size of 20 and 512 filters in the neural network architecture is a calculated choice tailored for the demands of text classification tasks. The kernel size, or filter size, determines the width of the convolution window that scans across the input data. In this case, a kernel size of 20 allows the model to examine twenty adjacent words at a time, enabling the capture of context within these word sequences. This can be particularly effective for recognizing patterns or features in text that span multiple words, such as phrases or specific syntactic constructions, which are often pivotal for understanding the overall meaning and sentiment of the text.

The use of 512 filters within this layer significantly increases the model's capacity to extract features. Each filter can be thought of as a feature detector, looking for different types of patterns in the text. With 512 filters, the network is well-equipped to identify a wide array of textual features, making it robust in the face of the complexity and variability inherent in natural language. This high number of filters is indicative of the model's deep architecture, designed to handle the intricate task of classifying texts into multiple categories, where a nuanced understanding of the language is crucial.

Furthermore, the choice of the `Leaky ReLU` activation function following the convolutional layer adds an element of non-linearity to the model, allowing for more complex relationships to be learned. `LeakyReLU` is

particularly chosen over the standard `ReLU` to mitigate the issue of neurons "dying" during training; it allows a small, non-zero gradient when the unit is not active, thus maintaining a gradient flow even for neurons that output negative values. This can lead to more efficient learning and better performance, especially in deeper networks that are prone to saturation and dead neurons.

```
model.add(Conv1D(512, 20, activation=LeakyReLU(alpha=0.01)))
```

- `LeakyReLU` is used as activation layer here with $\alpha = 0.01$. Initially, we made use of the classic `ReLU` but we wanted to be sure that dead neurons will not be a problem for our case.

(c) Global Max Pooling 1D

Pooling is a technique used in the field of neural networks, particularly in CNNs to reduce the spatial dimensions of the input volume for the next layer in the network. It is a form of downsampling that reduces the number of parameters and computation and helps to achieve spatial invariance to input transformations. However, the pooling layer contains no parameters. Instead, pooling operators are deterministic.

Pooling layers typically follow Convolutional Layers and come in different types, with Max Pooling being one of the most common. Max pooling is a deterministic pooling operator that calculates the maximum value in each patch of the feature map. Max Pooling helps in making the detection of features somewhat invariant to scale and orientation changes. It also reduces the computational cost by reducing the number of parameters.

Nevertheless, the amount of data that we have in this project is huge. So we must find a faster and less complex method to reduce our computations. That is why we are using Global Max Pooling 1D. Global Max Pooling 1D is a specific type of pooling operation designed for 1-dimensional input, such as in the case of time series data or sequences. Unlike the traditional Max Pooling operation that looks at patches of the input volume, Global Max Pooling operates over the entire length of the input data in each dimension and takes the maximum value over the entire dimension. In other words, The role of Global Max Pooling 1D in this context is to capture the most important feature (the highest value) for each feature map across the entire sequence, which can be especially beneficial for identifying key signals in the text for classification purposes.
By reducing the output of the convolutional layers from a 3D tensor (batch size, sequence length, features) to a 2D tensor (batch size, features) and by capturing the key features each time, Global Max Pooling reduces overfitting. It reduces overfitting by simplifying the model architecture and focusing on the most important features.

```
#Add Global_Max_Pooling Layer
model.add(GlobalMaxPooling1D())
```

This snippet of code does exactly what we explained. It is adding a `GlobalMaxPooling1D` layer to the model. This will reduce the output of the previous layer to its maximum value over the time dimension.

(d) Batch Normalization

Batch Normalization is an excellent layer that significantly lowers our neural network's overall complexity [1]. This layer standardizes the inputs to a layer for each mini-batch, stabilizing the learning process. By normalizing the features to have a mean of zero and a standard deviation of one, it ensures that no particular

feature dominates the gradient updates during training, which can lead to faster convergence. This is particularly beneficial after pooling layers (*just as we did here*), as the aggregation can sometimes lead to a shift in the distribution of the pooled features.

Moreover, the `BatchNormalization` layer is strategically placed. It is positioned soon following pooling to ensure that activations sent into succeeding layers support good gradient flow and promptly address any potential distribution shifts resulting from the pooling procedure. In this setup, the BatchNormalization layer serves as a regulatory checkpoint in the network, guaranteeing that the feature representations retain favorable attributes that promote effective learning.

```
model.add(BatchNormalization())
```

The snippet above just adds a `BatchNormalization` layer from `tensorflow`'s library.

(e) Dropout Layer

The Dropout layer is an essential technique in the training of deep neural networks to prevent overfitting. The term overfitting is used in the context of predictive models that are too specific to the training data set and thus learn the scatter of the data along with it. This often happens when the model has too complex a structure for the underlying data or because certain neurons from different layers influence each other. The problem then is that the trained model generalizes very poorly, i.e., provides inadequate predictions for new, unseen data. The performance on the training data set, on the other hand, was very good, which is why one could assume a high model quality.
With deep neural networks, it can happen that the complex model learns the statistical noise of the training data set and thus delivers good results in training. In the test data set, however, and especially afterwards in the application, this noise is no longer present, and therefore the generalization of the model is very poor.

However, we do not want to abandon the deep and complex architecture of the network, as this is the only way to learn complex relationships and thus solve difficult problems.

This is why we are using a Dropout Layer. The Dropout layer combats this by randomly setting a fraction of the input units to zero at each step during training time, which helps to prevent complex co-adaptations on the training data. It forces the network to learn more robust features that are useful in conjunction with various different random subsets of the other neurons. Thus, they have no influence on the prediction and also in the backpropagation.

In our model, adding `Dropout(0.2)` means that during training. each input to the Dropout Layer has a 20% chance of being set to zero. Generally, the Dropout rate is set as the percent of the test_data. In our project, we have set `test_data=0.2`. Equivalently, the `dropout_rate=0.2`.

```
# Add Dropout Layer
model.add(Dropout(0.2))
```

Dropout is one of the most effective and commonly used regularization techniques to achieve good generalization, which is particularly important in a multiclass classification task where the model must discern between multiple different classes based on the input CSV file. Therefore, the use of a Dropout layer aligns with the overall objective of our project to create a robust and accurate multiclass text classifier by leveraging the strengths of neural networks in handling complex patterns in the data while ensuring that the model's performance is not hindered by overfitting.

(f) Dense Layer

The final stage of our multi-class text classifier incorporates a Dense layer using `softmax` as activation function. As the layer that generates the final predictions, it is the last and most important layer in the network.

The Dense layer, or fully connected layer, is the layer where all neurons from the previous layer are connected to each neuron in the Dense layer. In your model, the Dense layer has a configuration of 512 units connecting to `labels_unique_num` units, corresponding to the number of unique labels in your classification task. The choice of having that many units suggests that the network is designed to capture a broad and complex representation of the features learned from previous layers. These 512 units serve as a high-dimensional feature space in which the learned representations from the convolutional and pooling layers are combined and transformed.

Each of these units carries forward the learned patterns to the output layer, which consists of a specified amount of neurons, one for each class. This allows the model to make a decision for each class based on the features extracted throughout the network. The `softmax` activation function then takes these decisions and converts them into probability values, which are easier to interpret. `Softmax` is the appropriate choice for multiclass classification problems because it ensures that the output probabilities are normalized, meaning they sum up to one. This normalization allows for a direct probabilistic interpretation of the model's outputs, enabling the selection of the class with the highest probability as the predicted class label.

The architecture's culmination with a Dense layer and `softmax` activation is a testament to the model's overall design, reflecting a clear pathway from raw text input to a refined set of probabilities indicating class membership. It encapsulates the end-to-end process of feature extraction, transformation, and classification that is fundamental to the success of deep learning models in text classification tasks. Below lies a code snippet that shows how the dense layer mentioned above is implemented in our code.

```
model.add(Dense(labels_unique_num, activation='softmax'))
```

Training Algorithm

An adaptive learning rate optimization technique created especially for deep neural network training is called the Adam optimizer. It is an extension of the stochastic gradient descent (SGD) algorithm and is designed to update the weights of a neural network during training. The term "Adam" is derived from "adaptive moment estimation," highlighting its ability to adaptively adjust the learning rate for each network weight individually. Unlike SGD, which maintains a single learning rate throughout training, Adam optimizer dynamically computes individual learning rates based on the past gradients and their second moments. The creators of Adam optimizer incorporated the beneficial features of other optimization algorithms such as AdaGrad and RMSProp. Similar to RMSProp, Adam optimizer considers the second moment of the gradients, but unlike RMSProp, it calculates the uncentered variance of the gradients (without subtracting the mean).

By incorporating both the first moment (mean) and second moment (uncentered variance) of the gradients, Adam optimizer achieves an adaptive learning rate that can efficiently navigate the optimization landscape during training. This adaptivity helps in faster convergence and improved performance of the neural network.

While working on our project, we were unsure about which optimizer to utilize as a training algorithm. Nevertheless, for a number of reasons, we concluded that the ADAM optimizer was the best option.

- Efficiency: Adam is computationally efficient, has little memory requirements and is invariant to diagonal rescale of the gradients.
- Adaptive Learning Rates: Adam adjusts the learning rate throughout training, which can lead to better performance and convergence than methods with a fixed learning rate.
- Suitability for Problems with Noisy or Sparse Gradients: The adaptive nature of Adam makes it well-suited for dealing with noisy problems like text classification, where the data and gradients can be sparse and noisy.
- Robustness: Adam is generally robust to the choice of hyperparameters, though the learning rate might sometimes need to be changed from the default.

Because text data is inherently rough and high-dimensional, Adam offers a dependable and efficient way to navigate the optimization environment while developing a multiclass text classifier. Adam is a good option for our model's optimizer because of his flexibility in adjusting the learning rate to various parameters and his resilience to initial hyperparameter settings. These qualities will accelerate convergence and increase the likelihood of obtaining lower loss values on our classification jobs.
To summarize, the results of the Adam optimizer are generally better than every other optimization algorithm, have faster computation time, and require fewer parameters for tuning. That's why we thought it was the ideal optimizer for our project based on multiclass text classification.

```
# Optimizer as train algorithm
optimizer = Adam(learning_rate=lr)
```

- `optimizer=Adam(learning_rate=lr)`: The Adam class is instantiated with a specific learning rate (lr). The learning rate (lr) is a hyperparameter that controls how much to change the model in response to the estimated error each time the model weights are updated.

Train

To train the model we have made a custom function. This custom training function encapsulates the training loop, providing a convenient and reusable way to train our neural network on any given dataset.

- `model`: This is the neural network model that we have already defined and compiled with the necessary architecture, loss function, optimizer, and metrics.
- `train_texts`: These are the training data inputs. In the context of our code it is a preprocessed and tokenized text data.
- `train_labels`: These are the corresponding labels for the train_texts. In our case, the labels are one-hot encoded vectors.
- `test_texts`: These are the testing data features. Similar to train_texts, but this data is not used for training the model. Instead, it's used to evaluate the model's performance at the end of each epoch to see how well it generalizes to unseen data.
- `test_labels`: These are the testing data labels. They are the targets for the test_texts and are used to calculate the validation loss and accuracy of the model.

- `batch_size`: This parameter defines the number of samples that will be propagated through the network at one time. It is a crucial hyperparameter that can affect the convergence and performance of the neural network. Here the batch_size = 64.
- `epochs`: The number of times the learning algorithm will work through the entire training dataset. Here max_epoch = 10.

```
# Train the model
def train(model, train_texts, train_labels, test_texts, test_labels,
    batch_size, epochs=40):

history = model.fit(train_texts, train_labels, validation_data=(test_texts,
    test_labels), epochs=epochs, batch_size=batch_size, use_multiprocessing=
    True)

return model, history
```

When this train function is called with the appropriate parameters, it will execute the training process, iterating over the training data in mini-batches of size batch_size for a total of epochs times through the dataset. During each epoch, the model's weights will be updated in an attempt to minimize the loss function, and after every epoch, the model's performance will be evaluated on the test set to monitor its accuracy and generalization capability.

Results

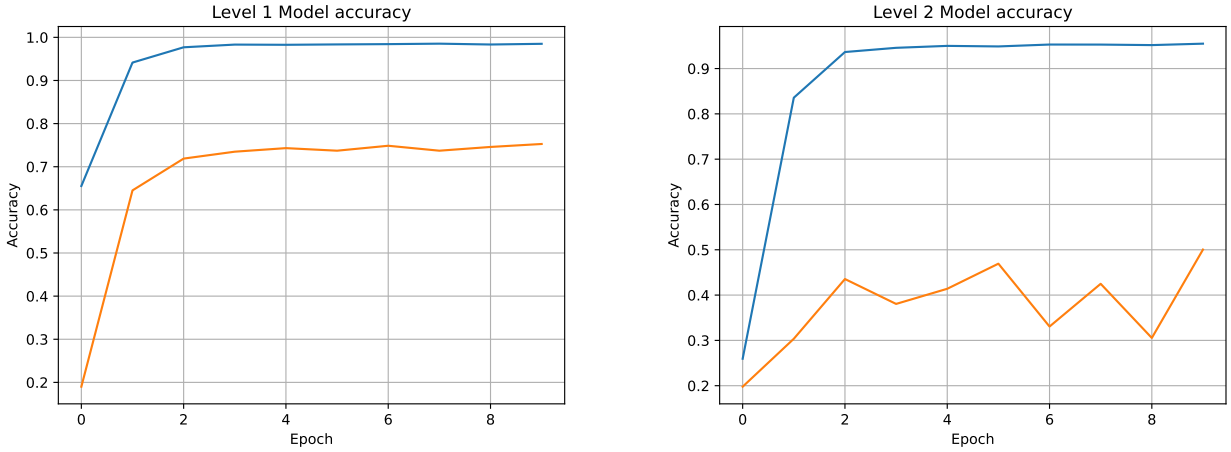
The model created earlier is trained and then tested on separate data with the code in the following snippet:

```
# Training
history = model.fit(train_texts, train_labels, validation_data=(test_texts,
    test_labels), epochs=epochs, batch_size=batch_size, use_multiprocessing=
    True)

# Evaluation
loss, accuracy = model.evaluate(test_texts, test_labels)
```

The evaluation of our multiclass text classifier has provided insightful observations into its performance, with the network achieving an overall accuracy of around 78% using `category_level_1` labels as shown in Figure 5a. This metric signifies that $\approx 3/4$ of the text samples were classified correctly across the various classes.
While this demonstrates a solid foundation in the classifier's ability to discern and categorize text data accurately, it also indicates room for improvement. An accuracy of 78% suggests that, although the classifier is generally reliable, there are instances where it may struggle to correctly identify the class labels. This level of performance sets a benchmark for future iterations of the model, where enhancements can be targeted towards reducing the classification errors and increasing the accuracy.

Upon evaluating our multiclass text classifier with `category_level_2`, we observed a marked decrease in performance, with the model achieving an overall accuracy of just 50%. This stark contrast from the previously discussed accuracy highlights the challenges inherent in text classification tasks, especially when dealing with diverse or more complex label sets. An accuracy of 50% indicates that the classifier struggles significantly to correctly identify the class labels for a majority of the text samples. This performance

level suggests that the classifier may be unable to capture the nuances and variations required to distinguish between the classes effectively in this particular label set. Such a result prompts a critical review of the classifier's design, feature extraction methods, and training process, signaling a need for substantial adjustments to improve its ability to generalize across different sets of labels.



(a) *category_level_1* (b) *category_level_2*

Figure 5: Accuracy with respect to iterations number

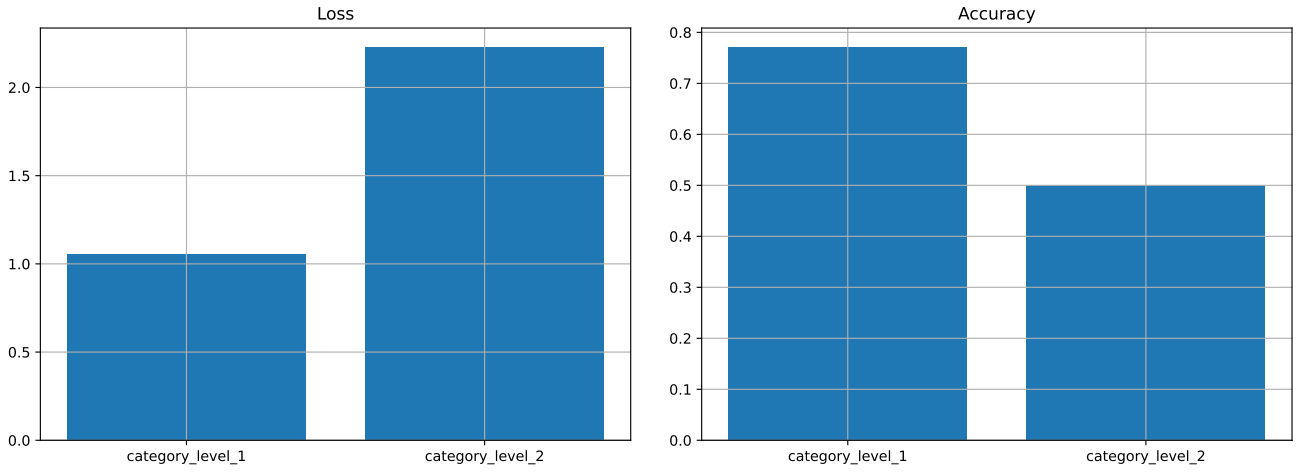
As far as the final parameters are concerned, we can not write each and every parameter here. The reason why can be clear once we print some insights about one of our models. After execution, we used `print(model_level_1.summary())` and it returned the following:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, 6544, 17)          2207705

conv1d_2 (Conv1D)            (None, 6533, 512)         104960

global_max_pooling1d_2 (Gl   (None, 512)               0
obalMaxPooling1D)

batch_normalization_2 (Bat   (None, 512)               2048
chNormalization)

dropout_2 (Dropout)          (None, 512)               0

dense_2 (Dense)              (None, 17)                8721

=================================================================
Total params: 2323434 (8.86 MB)
Trainable params: 2322410 (8.86 MB)
Non-trainable params: 1024 (4.00 KB)
_________________________________________________________________
```

As we can see here, there is no way of printing all of the parameters that models use. Indicatively, some are listed in the table below:

Layer	Weight/Bias	Value
Embedding	$embedding[0][0]$	0.01644432
Conv	$weight[0][0][0]$	-0.04686261
Dense	$weight[0][0]$	-0.15257788
Dense	$bias[0]$	-0.01133604

Table 1: Weights and biases of some layers



(a) (b)

Figure 6

References

[1] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," 2015.

[2] A. Hamza, "Effectively pre-processing the text data part 1: Text cleaning,"

[3] A. H. S. S. V. D. S. S. Sayyaparaju, "Sentiment analysis of imdb movie reviews,"

[4] N. K. Nissa, "Text messages classification using lstm, bi-lstm, and gru."

[5] R. Pramoditha, "How to choose the right activation function for neural networks."

[6] S. Saxena, "Understanding embedding layer in keras."

[7] C. Klinik, "Padding for nlp."

[8] S. Y. Min Lin, Qiang Chen, "Network in network,"