UNIVERSITY OF SALERNO

DEPARTMENT OF COMPUTER SCIENCE

MASTER'S THESIS IN

COMPUTER SCIENCE

CURRICULUM

INTERNET OF THINGS

A multimodal neural network for binary classification of TV banners

Supervisor:

Prof. Genoveffa Tortora

Candidate:

Silvio Di Martino

0522501025

Academic year:

2022/2023

Abstract

This work presents a multimodal learning approach for the binary classification of TV banners as self-promotional or traditional media. *Keras-OCR* has been used to extract the text contained in the images and encode it using a *Sentence Transformer* multilingual model. The neural network architecture consists of a convolutional branch for the images, a layered fully-connected branch for the textual part, and a final layered fully-connected branch. Different experiments with different approaches have been carried out: including images only, images and text, with and without data augmentation, and their performance have been evaluated. The results show that the best performance is achieved when considering both images and text, highlighting the importance of combining multiple modalities of input for achieving high accuracy in the classification task.

Contents

Chapter 1

Introduction

The technological progress that has characterized the last few years is revolutionizing our lives. The innovation achieved in the Computer Science world has initiated a true revolution, modifying, and revitalizing the habits of society. This enormous progress has opened up new and somewhat unexpected prospects for life, responding to precise needs and desires of society.

In this context, *Artificial Intelligence* (**AI**) plays a significant role. According to McCarthy [1], AI is "the science and engineering behind the creation of intelligent machines, especially intelligent computer programs". McCarthy defined it as something related to the task of using computers to understand human intelligence, with a significant difference: "AI does not have to confine itself to methods that are biologically observable. AI researchers are free to use methods that are not observed in people or that involve much more computing than people can do". He anticipated a fundamental concept, identifying immediately the common points and differences with human intelligence, carrying on a research that actually went on for years. In fact AI research began after World War II, with several people working independently on developing intelligent machines. The English mathematician Alan M. Turing [2] has been the first: he proposed the question "Can machines think?" and discussed conditions for considering a machine to be intelligent.

Today, AI is a leading component of the field of Computer Science. With autonomous machine intelligence, AI-powered computers, mechanisms, and systems could reduce much of the burden of decision-making, repetitive actions, and immediate responses away from humans. AI technology is currently being deployed in various sectors, including transportation, healthcare, manufacturing, finance, and more.

Machine Learning (**ML**) is a subfield of AI and Computer Science that focuses on using data and algorithms to imitate the way humans learn, gradually improving its performance and accuracy. The term ML was coined by Arthur Samuel in 1959, that defined it as "the field of study that gives computers the ability to learn without being explicitly programmed."

As the volumes and variety of data continue to grow, coupled with the increasing accessibility and affordability of computational power and high-speed Internet, ML is becoming increasingly vital. Technological advancements in storage and processing power have enabled the creation of innovative products based on ML, such as recommendation engine systems and self-driving cars. With these digital transformation factors, one can swiftly and automatically develop models that are capable of quickly and accurately analyzing complex and exceedingly large datasets. Based on data collected from Google Trends over the past years, the popularity of these learning approaches, according to Sarker [3], is increasing day-by-day.

All this is possible through the use of statistical methods that allow for training algorithms to make classifications or predictions and uncover key insights in data mining projects.

1.1 Artificial Neural Networks

Among the methods and algorithms designated for *AI*, we find *Artificial Neural Networks* [4] [5] (**ANNs** or simply **NNs**): as a subset of ML, they are mathematical methods that teach machines to process data, in a way that is inspired by the biological neural networks that constitute human brains. A *NN* is based on a collection of interconnected units or nodes called *neurons*, which loosely model the neurons in a biological brain. It creates an adaptive system that computers use to learn from their mistakes and improve continuously: in fact it can help computers to make intelligent decisions with limited human assistance. This is because a NN can learn and model the relationships between input and output data.

Thus, NNs attempt to solve complicated problems and have several use cases: recognizing faces, detecting sedentary people by analyzing measurements from smartphone and smartwatch sensors, doing medical diagnosis

by medical image classification, forecasting electrical load and energy demand, predicting financial trends by processing historical data, and many more.

The human brain is the inspiration behind neural network architecture: as human neurons form a biological neural network and send electric signals to each other to help human process information, similarly a NN is made of artificial neurons that work together to solve a problem.

A basic NN has interconnected artificial neurons in three different types of layers:

- **Input layer**: data enters the NN from the input layer: input nodes process information and pass it on to the next layer;
- **Hidden layer**: each hidden layer analyzes the output from the previous layer (it may be the input layer or other hidden layers), processes it further and passes it on to the next layer;
- **Output layer**: this layer gives the final result of all the data processing by the NN. It can have single or multiple nodes, depending on the task that the NN itself must accomplish: with a multi-class classification problem, the output layer might consist of more output nodes. If we have a binary classification problem, as in the use case that will be explained in the next sections, the output layer will have one output node.

Depending on the depth of NNs, we may pass from the concept of ML to the *Deep Learning* (**DL**): the word *Deep*, in fact, refers to the depth (or number) of layers in a NN. The fundamental aspect in which ML and DL differ is in how each algorithm learns and how much data each type of algorithm uses.

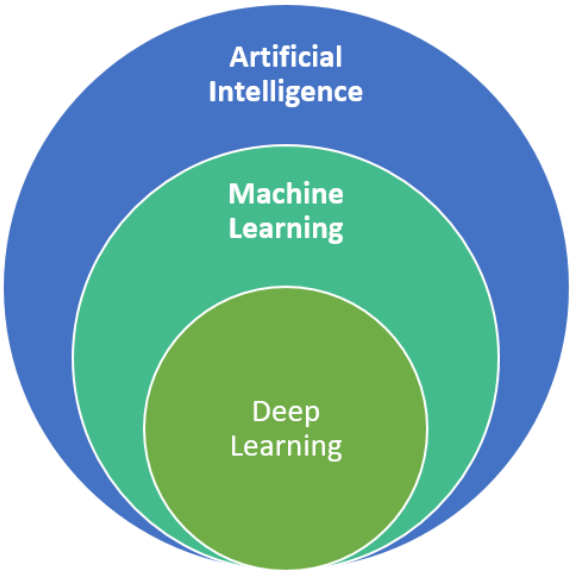

Figure 1.1: Representation of AI, ML and DL

1.2 Convolutional Neural Networks

There are different types of NNs: one of these is the *Convolutional* [6] [7] one (**CNN**), mainly utilized for image classification and computer vision tasks. CNNs leverage principles from linear algebra, like matrix multiplication, in order to identify hidden patterns within an image. However, they can also be used with speech or audio signal inputs.

CNNs have three main types of layers:

- **Convolutional layer**: the core building block of a CNN, where the majority of computation occurs. It requires the input data and a filter: the filter is a *two-dimensional* (2D) array of weights that moves across the image (this process is known as *convolution*): a dot product is calculated between the input pixels and the filter and is then fed into an output array. This filter shifts by a stride, repeating the *convolutional process* until it has swept across the entire image. The final output from the series of dot products is known as a feature map;
- **Pooling layer**: this type of layer conducts dimensionality reduction (*downsampling*), reducing the number of parameters in the input. It is useful to reduce complexity and limit risk of overfitting. Similarly to the *convolutional process* described before, the pooling operation

sweeps another filter across the entire input: however this filter does not have any weights, it simply applies an aggregation function to the values within the receptive field.

There are two main types of pooling:

- **Max pooling**: as the filter moves across the image, it selects the pixel with the maximum value;
- **Average pooling**: as the filter moves across the image, it calculates the average within the receptive field.

- **Fully-connected layer**: in this type of layer, each node in the output layer connects directly to a node in the previous layer. This layer performs the task of classification, relying on the features extracted in the previous layers.

For this thesis project, a CNN has been constructed, but as we will see in the next sections, it is just a branch of a bigger and more complex NN.

1.3 Multimodal Neural Networks

Thanks to recent advances in NNs, multimodal technologies have made possible advanced, intelligent processing of all kinds of unstructured data, including images, audio, video, text, and so on. Moreover, the world surrounding us involves multiple modalities: we see objects, hear sounds, feel texture, smell odors, and so on. Our five senses capture information from five different sources, and five different modalities. A modality refers to the way in which something happens or is experienced and a research problem is characterized as multimodal when it includes multiple such modalities.

So human brains consist of NNs that can process multiple modalities simultaneously: we can reason about what our interlocutor is saying, his emotional state and his surroundings. This allows for a more holistic view and deeper comprehension of the situation.

In order for the AI to make progress in understanding the world around us, it needs to be able to interpret and reason also about multimodal mes-

sages. Multimodal ML aims to build models that can process and relate information from multiple modalities. An example of multimodal data is data that combines text with imaging data consisting of pixel intensities and an-notation tags. As these modalities have fundamentally different statistical properties, combining them is non-trivial, which is why specialized modelling strategies and algorithms are required.

1.4 OCR and Sentence Transformers

Multimodal learning allows for a more holistic understanding of data, as well as increased accuracy and efficiency. Its applications are varied, exciting and constantly increasing: *Optical Character Recognition* (**OCR**) is one of the many applications of multimodal technologies [8] . OCR involves the use of computer vision techniques to extract text from images or documents. The ability of OCR to convert unstructured data, such as scanned documents and images, into structured data has made it an important tool in various industries, including healthcare, finance, and government.

OCR is a prime example of how the combination of multiple modalities, such as text and images, can lead to improved data processing and analysis. By using OCR, we can extract valuable information from unstructured data that would otherwise be difficult to access, providing a more comprehensive view of the data and enabling deeper insights and analysis.

Sentence Transformer [9], on the other hand, is a powerful tool that can be used for *natural language processing* (**NLP**) tasks such as text classification, question answering, and sentence similarity. It utilizes a transformer architecture that is capable of generating high-quality sentence embeddings, which can be used to compare and relate different sentences. It can be used in multimodal learning by encoding textual information into a fixed-length vector representation, which can then be combined with other modalities, such as images or audio, in a multimodal neural network. This approach allows for the integration of textual information with other modalities, which can lead to better performance in tasks such as image captioning, video classification, and speech recognition.

1.5 The problem definition

Among the various applications of multimodal learning that may involve all the technologies listed so far, the use case of this thesis project is one: it concerns the recognition of TV banners and advertisings, that are played on TV along with the content for a duration of about 10 seconds. The TV advertisings taken into account are **self-promotion** and **traditional media advertisings**:

- **Self-promotion**: as suggested by Siegert [10], it is a type of advertising of a TV broadcaster that *avdvertises itself*, i.e. promotes its brands, programs, titles or products within its own programs or titles;
- **Traditional-media advertisings**: as the name suggests, it is a type of advertising that is not related to the TV broadcaster that airs it, instead it is promoting something else (i.e. Coca Cola advertising aired by Rai1).

The goal of this project is a binary classification, i.e. classify TV banners in the two classes mentioned above.

The main idea behind this thesis project is the construction of a multimodal network, that is a NN able to take in input both the images of the banners, and the text extracted from the banner themselves. In fact, the network will have a *convolutional branch* and a *layered fully-connected branch* to take images and text respectively.

However, various experiments will be conducted in order to obtain the best performance: in addition to taking into account images and text together, in fact, we will try to test the model even without the textual information (and consequently modifying the NN, deleting the *convolutional branch*), or by applying data augmentation techniques on training data, to get more samples on which to train the model.

In the following sections all the steps will be described, starting from the dataset understanding, the text extraction with an OCR model and the following embedding operation with a Sentence Transformer, passing through the image normalization and augmentation and the construction of the neural network, including experiments and complete specifications.

1.6 Outline

The work of this thesis is organized into different chapters, here briefly summarized:

- **Chapter 1**: an introduction about the most recent advances in Computer Science and AI world, with a focus on Artificial Neural Networks, OCR and Sentence Transformers models, up to the problem definition of this thesis project;
- **Chapter 2**: state-of-the-art works and technologies about Multimodal learning;
- **Chapter 3**: the presentation of Frameworks and tools involved;
- **Chapter 4**: the main chapter. The dataset description and the operations applied on it, the Multimodal Neural Network architecture and all the experiments conducted, the problems encountered and the findings of the best solution;
- **Chapter 5**: discussions, conclusions and future directions in order to improve this work.

Chapter 2

Related works

In recent years, multimodal learning has emerged as a promising approach for modeling and processing complex data that involve different modalities such as images, text, speech, and sensor signals. Among these modalities, image and text are two of the most widely studied ones due to their rich and complementary information.

In this chapter, there will be a review about the recent advances in multimodal neural networks for processing images and text. Specifically, there will be a focus on the state-of-the-art models and techniques that have been proposed to integrate image and text modalities, that will highlight their strengths and limitations.

The first analyzed work is "*Multimodal Deep Networks for Text and Image-Based Document Classification*" [11]. It proposes a novel approach for document classification that utilizes both text and image information: it learns from both a document image and its textual content automatically extracted by *Optical Character Recognition* **OCR** to perform its classification. The authors argue that incorporating both modalities can lead to better performance compared to using only one modality.

The proposed model consists of two branches: a convolutional neural network for processing the image modality and a recurrent neural network for processing the text modality. The output of each branch is concatenated and passed through fully connected layers to produce the final classification.

The authors evaluate their approach on two publicly available image based-datasets and show that their model outperforms several baseline methods that only use a single modality. They also conduct ablation experiments to demonstrate the importance of each modality and show that the combi-

nation of both modalities leads to the best performance.

The second work analyzed is about a really interesting and original research topic nowadays: *Multimodal Meme Dataset (MultiOFF) for Identifying Offensive Content in Image and Text* [12]. It explores a new form of communication of the web: a meme is a form of media that spreads an idea or emotion across the Internet. Combining two modalities to detect offensive content is still a developing area. Memes make it even more challenging since they express humour and sarcasm in an implicit way, because of which the meme may not be offensive if we only consider the text or the image.

The authors developed a classifier for the offensive meme detection and used an early fusion technique to combine the image and text modality and compare it with a text- and in image-only baseline:

- *Logistic regression* and *Naive Bayes* have been used to classify memes based on the provided textual information for a single modality experiment. Apart from these, a deep neural network with fully connected layers, a stacked *Long Short Term Memory* (**LSTM**) network, a *Bidirectional LSTM* and also a CNN have been compared for meme classification based on text;
- a *CNN architecture* (VGG16) has been used to classify the targeted image data.

Text and image classifiers are evaluated individually, additionally the authors combined the modalities with a early fusion approach. Logistic regression performs best in predicting the offensive meme category based on the text, while results demonstrate the improvement in retaining offensive content when both text and image modality associated with the meme was considered.

The third work is about the fake news detection: the title is "*Multimodal Multi-image Fake News Detection*" [13] and its authors propose a multimodal multi-image system that combines information from different modalities in order to detect fake news posted online. In particular their system combines:

- textual information: BERT has been used to better capture the underlying semantic and contextual meaning of the text;
- visual information: VGG-16 model has been used to extract image tags from multiple images contained in the article;
- semantic representation: it refers to text-image similarity, calculated using the cosine similarity between the tile and image tags embeddings.

The experimental results on a real world dataset show that combining features from the different modalities is effective for fake news detection. In particular, the multimodal multi-image system significantly outperforms the BERT baseline by 4.19% (this show the importance of the visual features in detecting fake news) and SpotFake by 5.39% (it is a multimodal system that is based on text and image features learned with BERT and VGG-19 pre-trained on ImageNet dataset respectively).

A different architecture has been designed by the authors of "*Multimodal Spam Classification Using Deep Learning Techniques*" [14]: they investigate on the e-mail system, plagued by the unwanted influence of spam. In their work they classify a mail into spam and not-spam by analyzing the image and text content, processing it through independent classifiers using CNNs.

Since the objective is a binary classification, a small CNN has been built for image features; for text input another CNN has been involved, with the first layer that embeds words into 100 dimensional vectors using GloVE word embeddings. Two multimodal architecture have been proposed:

- 1. creating a shared representation of the features extracted by the independent CNNs and then passing it through Fully Connected layers;
- 2. using the class probabilities of the 2 classifiers and learning a rule between them. The image and text classifiers output class probabilities which then serve as input to a 3 layered Fully Connected network with Softmax at the end.

Both of their multimodal architectures give quite satisfactory results with the self-learning rule model. With an accuracy of 98.11%, it exceeds the accuracy of the independent classifiers and achieves.

The choice of the models is a very important point when trying to solve a classification task: in the work "*Exploring Hate Speech Detection in Multimodal Publications*" [15], different models that jointly analyze textual and visual information for speech detection have been proposed, comparing them with unimodal detection. Given a large scale dataset from Twitter, they use:

- a CNN for the images features;
- a single layer LSTM for the tweet text;
- the Google Vision API Text Detection module for extracting additional text from the image. Both the tweet text and the text from the image will feed the multimodal models.

Given the architectures above, to study how the multimodal context can boost the performances compared to an unimodal context, they propose a Feature Concatenation Model, a Spatial Concatenation Model and a Textual Kernels Model (all of them are CNN+RNN). Despite the model trained only with images proves that they are useful for hate speech detection, the proposed multimodal models are not able to improve the detection compared to the textual models. The causes could be the noise of data and the complexity and diversity of multimodal relations, as well as a small set of multimodal examples.

To sum up, different architectures have been involved and designed in order to address different issues and solve different tasks. Among the state-of-the-art works presented above, it is possible to note that combining two different input modalities can lead to better performances; however this is not a certainty, since it always depends on the task, but also on the dataset composition and on the type of architectures chosen to adddress the intended issue. As shown in the last reported work, in fact, it is not always true that the multimodal models outperform the textual models: for this reason conducting different experiments that consider text-only, image-only features and both of them, in order to achieve the best performances, could be be a challenging and interesting approach.

Chapter 3

Frameworks and tools

In this chapter, we will explore the various *frameworks* and *tools* used in our research, including their purpose, benefits, and limitations. By understanding, we can better analyze our data and draw meaningful conclusions.

3.1 Python

Python [16] is a popular programming language for machine learning and has become the go-to language for many data scientists and machine learning engineers. Python's popularity is due to its simplicity, flexibility, and ease of use.

Python has a vast ecosystem of libraries and frameworks that make machine learning development faster and more accessible. Some of the popular libraries for machine learning in Python are:

- **NumPy** is a library that provides support for multidimensional arrays and mathematical operations on them. NumPy is a fundamental package for scientific computing in Python and is used extensively in machine learning;
- **Pandas** is a library for data manipulation and analysis. Pandas provides data structures to efficiently manipulate large datasets and perform data analysis tasks such as data cleaning, data wrangling, and data transformation;
- **Scikit-learn** is a machine learning library in Python that provides a range of supervised and unsupervised learning algorithms, including classification, regression, clustering, and dimensionality reduction. Scikit-learn is built on top of NumPy and SciPy, which makes it easy to integrate with other Python libraries.

- **TensorFlow** is an open-source machine learning framework developed by Google. It provides a range of tools for building and training deep neural networks, including support for distributed computing, GPU acceleration, and automatic differentiation.
- **PyTorch** is another popular open-source machine learning framework that provides support for building and training deep neural networks. PyTorch is known for its dynamic computational graph, which makes it easier to debug and build complex models. It will be explained in details in the following sections.

Python's popularity in the machine learning community has led to the development of several high-level machine learning frameworks that provide abstractions and tools to simplify the machine learning workflow. These frameworks include Keras, which provides a high-level API for building and training deep learning models, and Hugging Face Transformers, which provides a range of pre-trained models for natural language processing tasks.

3.2 Google Colab

Google Colab [17] (short for "Collaboratory") is a Google service that allows running Python code in an online integrated development environment (IDE). It is particularly useful for those who need to use large amounts of computational resources, such as for deep learning or analysis of large amounts of data, without having to invest in expensive hardware or spend time configuring the development environment. In Google Colab, the code runs on an instance of a dedicated Linux virtual machine, with access to CPU, GPU, and TPU (Tensor Processing Unit). This means that users can use the processing power of Google to run their tasks without worrying about the hardware at their disposal. The main features of Google Colab include:

- Support for Python: Google Colab supports most Python libraries, including those used for deep learning such as Tensorflow, PyTorch, and Keras.
- Integrated Jupyter Notebook: Google Colab is integrated with Jupyter

Notebook, which provides an interactive interface for writing and executing Python code.

- Real-time collaboration: Google Colab offers the ability to collaborate in real-time with other users, allowing you to work on a project together with other team members.
- Google Drive storage: all files created in Google Colab are saved in Google Drive, allowing users to access their projects from any internet-connected device.
- Free of charge: Google Colab is free and does not require any installation or configuration of the development environment.
- Access to advanced processing resources: users can access Google's GPUs and TPUs to perform data processing faster and more efficiently.

However, there are also some limits to the use of Google Colab, such as:

- Limits on resource usage: users can only use a certain amount of processing and storage resources, and cannot run background processes or long-running programs.
- Session expiration: Google Colab sessions expire after a certain period of inactivity, and all data created during that session is lost.
- Dependence on Google resources: users depend on Google's resources, which means that if there are problems with the service, access to resources may be limited or interrupted.

Overall, Google Colab is an excellent tool for developers and data scientists who need to use large amounts of computational resources without having to invest in expensive hardware or spend time configuring the development environment.

3.3 keras-OCR

As said before, the main goal of this project is the recognition of TV banners through a multimodal NN, able to process both images (of the TV banners)

and **the text** (contained in the images themselves). Since we already have the images, the first step is the text extraction from the TV banners.

Optical character recognition (**OCR**) is the process of reading and recognizing characters from digital documents in any format (PDF, PNG, JPEG, TIFF, ...), without any human intervention, using computer vision and ML techniques. It concerns the conversion of images of digital or hand-written text to machine readable text. OCR engine has gained quite a popularity over the past few years, because using this type of techniques have several advantages:

- It increases productivity as it takes only some seconds (depending on the complexity of the image) to read, process, recognize and extract information from the documents;
- It is resource-saving and no manual work is required since the process is automatic;
- There is no need for manual data entry;
- Chances of error become less.

Using DL for OCR is a three-step process:

- **Preprocessing**: Images are not always in ideal conditions, ready for text detection: they can have noise, blur, skewness and other issues that needs to be handled before going on;
- **Text detection**: at this step some DL models are used to localize the text in images, usually to create bounding boxes around each word identified in the document;
- **Text recognition**: finally each bounding box is sent to the text recognition model. The final output is the text extracted.

For this thesis project, *keras-ocr* has been used: it provides out-of-the-box OCR models and an end-to-end training pipeline to build new OCR models. The package ships with an easy-to-use implementation of the CRAFT text detection model [15] and the CRNN recognition model [18].

It has been choosen since its performances in terms of latency, precision and recall is better compared to others existing OCR APIs.

3.3.1 Installation

keras-ocr supports *Python* >=3.6. The following command is used to getting started with it:

```
pip install keras-ocr
```

3.3.2 Usage

For the use case of this thesis project, a pretrained model has been used in order to detect and recognize text from the TV banners. In the next chapter, at 4.2 section, the functions used will be explained.

3.4 SentenceTransformers

SentenceTransformers is a Python framework for state-of-the-art sentence, text and image embeddings. The initial work was presented as *Sentence-BERT*, so before going further and in depth in the explanation of this Python framework, it is really better to do an introduction about *BERT* and the *Transformers*, which this framework is based on.

BERT is a new language representation model, which stands for *Bidirectional Encoder Representations from Transformers*. It is a type of Deep Neural Network architecture designed for *Natural Language Processing* (**NLP**) tasks. It was introduced by Devlin et al [19] at Google in 2018 and is a language representation model designed to pre-train Deep bidirectional representation from unlabeled text by jointly conditioning on both left and right context in all layers. One of the key innovations of BERT is that it is pre-trained on large amounts of text data using a process called *Masked Language Modeling* (**MLM**). In MLM, the model is trained to predict missing words in a sentence, given the context of the other words around it. This pre-training process allows BERT to learn general patterns and relationships in language, which can then be fine-tuned for specific NLP tasks, such as sentiment analysis or question answering. As a result, in fact, the pre-trained BERT model can be *finetuned* with just one additional output layer to create state-of-the-art models for a wide range of tasks, such as question answering and language inference, without substantial task-

specific architecture modifications.

BERT also introduced the concept of **contextual word embeddings**, which means that each word in a sentence is represented by a unique vector that takes into account its context within the sentence: the embeddings for a given word may differ depending on the context in which it appears. This is because BERT processes the input text bidirectionally, taking into account both the words that come before and after the target word. This allows BERT to better capture the nuances of language and to handle tasks where the meaning of a word depends on its context.

BERT is based on the **Transformer** architecture, which in turn was introduced by Vaswani et al. [20] in 2017. Transformer is a new simple network architecture, based solely on **attention mechanisms**, dispensing with *Recurrence Neural Networks* (**RNNs**) and *Convolutional Neural Networks* (**CNNs**). Recurrent models in fact are commonly used for processing sequences of input data, like text [21] or time-series data. However, their computation is done sequentially, one element at a time, which generates a sequence of hidden states based on the input and previous state. This sequential nature makes it difficult to parallelize within training examples, especially for longer sequences, as it becomes memory-intensive and limits batching across examples.

This is the main reason behind the introduction of *attention*: in fact the *attention mechanism* is a key component of many state-of-the-art Neural Network Architectures. At a high level, it is a way for a Neural Network to selectively focus on certain parts of the input sequence, while ignoring others. This is done by computing a set of *attention weights* that indicate how important each element of the input sequence is to the output of the Neural Network. As stated by Vaswani et al. [20], an *attention function* can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key. So the idea is that not only can all the input words be taken into account in the context vector, but relative importance should also be given to each one of them: the *attention* acts as a weak inductive

module discovering relations between input tokens.

The *Transformer* has been the first transduction model that utilized *self-attention* exclusively to compute representation of both its input and output, without relying on sequence-aligned *RNNs* or *CNNs*. *Self-attention* is a specific type of attention mechanism: the input sequence is split into a set of vectors, or embeddings, that represent each element of the sequence. Each embedding is then used to compute a set of attention weights that indicate how important that element is to the other elements of the sequence. The attention weights are then used to compute a weighted sum of the embeddings, which is used as the output of the *self-attention* layer. In particular, in the *Transformer* architecture, the *Multi-head attention* mechanism has been used: it allows the network to compute multiple sets of *attention weights* in parallel. In *Multi-head attention*, the input sequence is split into multiple subsets, or heads, and each head is used to compute a set of attention weights. The outputs of the different heads are then concatenated and linearly transformed to produce the final output of the *Multi-head attention* layer. In other words, instead of a single *self-attention* mechanism attending to all aspects of the sequence, *Multi-head attention* [22] allows the model to attend to different aspects of the sequence with different mechanisms. In fact the input sequence is first transformed into three vectors, using learned linear transformations: a query vector, a key vector, and a value vector. These three vectors are then used to compute *attention* scores between each element in the sequence, as in a regular *self-attention* mechanism. However, in *Multi-head attention*, this process is repeated several times in parallel, with different learned parameters for each head. The output of each head is then concatenated and transformed again with another learned linear transformation, resulting in a final representation of the input sequence. The advantage of using multiple heads is that each head can learn to capture different aspects of the input sequence, allowing the model to capture more complex relationships and patterns than a single self-attention mechanism would be able to. Additionally, the use of multiple heads can improve the stability and generalization performance of the model.

Now, finally, it is possible to talk and better understand *Sentence Trans-*

formers: they are a type of model that uses pre-trained *Transformer-based* architectures like BERT to generate embeddings for entire sentences or paragraphs of text. The work was initiated by Reimers and Gurevych [9], that introduced a novel approach to learning sentence embeddings using a *Siamese neural network* architecture based on BERT (**SBERT**). So *SBERT* is a *Siamese* architecture that consists of two identical BERT networks that share the same weights and are fed pairs of sentences. The goal is to learn embeddings for each sentence such that the embeddings for semantically similar sentences, found using a similarity measure like cosine-similarity or Manhattan / Euclidean distance, are closer together in the vector space than those for dissimilar sentences.

SBERT adds a pooling operation to the output of BERT to derive a fixed sized sentence embedding: Devlin et al created this type of architecture (figure 3.1) to update the weights such that the produced sentence embeddings are semantically meaningful and can be compared with cosine-similarity. For these reasons SBERT can be considered a further development of the BERT model, where the focus is on sentence-level tasks instead of token-level tasks. In addition, it has achieved state-of-the-art results on various sentence-level tasks, such as semantic textual similarity, paraphrase detection, and text classification.

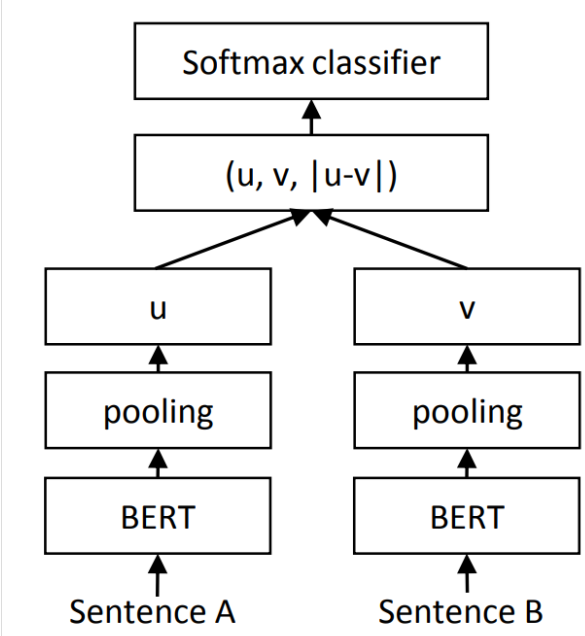

Figure 3.1: SBERT architecture with classification objective function.

Given the specific requirements of this thesis project use case, which involve the need to generate embeddings for both sentences and individual words in the Italian language, it was deemed essential to leverage a model capable of handling multiple languages. As a result, the usage of a Multilingual model, such as the *sentence-transformers/paraphrase-multilingual-mpnet-base-v2* [23], was considered a suitable solution to achieve the desired results. This model is specifically designed to produce high-quality sentence and text embeddings across multiple languages, including Italian, thereby enabling accurate and efficient processing of natural language data. It maps sentences and paragraphs to a *768 dimensional dense vector space* and can also be used for tasks like clustering or semantic search. The reason behind the emergence of these multilingual models are to be found in the fact that the vector spaces, i.e., the sentence representation, are not aligned: the sentences with the same content in different languages would be mapped to different locations in the vector space. This leads to the emergence of these new models, presented by Reimers and Gurevych [24] in 2020: the idea is that a translated sentence should be mapped to the same location in the vector space as the original sentence. They use the original monolingual model to generate sentence embeddings for the source language, and thein train a new system on translated sentences to *mimic* the original model. In other words they require a *teacher model* M for source language s and a set of parallel (translated) sentences $((s_1, t_1), \ldots, (s_n, t_n))$, where t_i is the translation of sentence s_i. Note that t_i can be in different languages. The goal is reached training a new *student model* $\hat{M}$ such that $\hat{M}(s_i) \approx M(s_i)$ and $\hat{M}(t_i) \approx M(s_i)$ using *mean squared error* (**MSE**) loss. The training procedure is illustrated in the following figure 3.2: given parallel data (e.g. an English and an Italian sentence), the student model is trained such that the vector produced for the English and Italian sentences are close to the teacher English sentence vector.

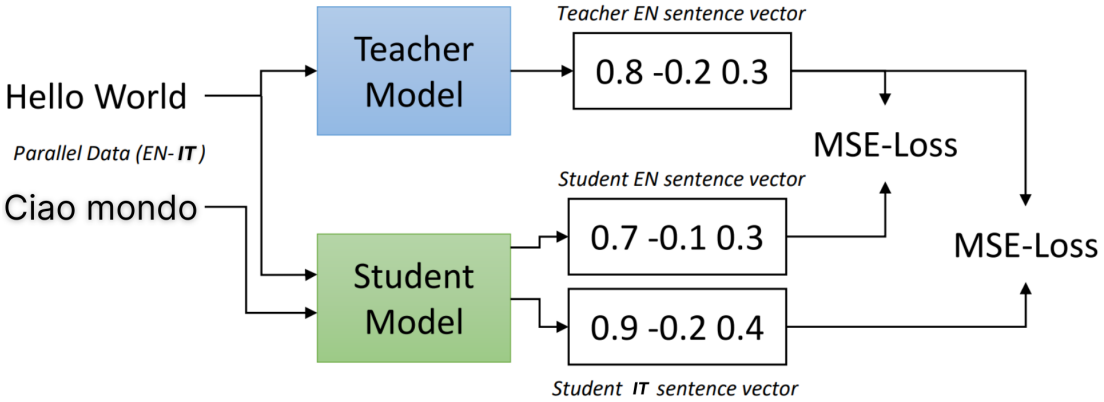

Figure 3.2: Training a multilingual model

This approach is effective for 50+ languages from various language families and has various advantages compared to other training approaces for multilingual sentence embeddings: for instance LASER, ideated by Artetxe and Schwenk [25], trains and encoder-decoder LSTM model using a translation task, and the output of the encoder is used as sentence embedding. While LASER works well and very effectively for identifying exact translations in different languages, it works less well for finding similar sentences that are not exact translations.

3.4.1 Installation

The framework *sentence-transformer* supports *Python* >=3.6. The following command is used to getting started with it:

```
pip install -U sentence-transformers
```

3.4.2 Usage

For the use case of this thesis project, it is needed to load the *sentence-transformers/paraphrase-multilingual-mpnet-base-v2* model. In the next chapter, at 4.3 section, the text embedding phase will be fully explained.

3.5 PyTorch

PyTorch [26] is an open-source machine learning framework developed by Facebook's AI Research team. It is primarily used for developing deep learning models and is based on the Torch library. PyTorch provides an

intuitive and flexible Python-based interface for building and training deep neural networks, making it a popular choice for both researchers and practitioners.

One of the key features of PyTorch is its dynamic computational graph system, which allows for easy debugging and quick iteration during model development. Unlike other popular deep learning frameworks like TensorFlow, PyTorch allows us to define and modify neural network architecture on-the-fly, rather than requiring to define it upfront and then execute it.

PyTorch also supports a variety of hardware acceleration options, including GPUs and TPUs, making it well-suited for training large-scale models. Additionally, it has a large and active community, with a wealth of resources and pre-trained models readily available.

Tensors are the fundamental data structure used in PyTorch for all operations, including building and training deep neural networks. In PyTorch, a tensor is a multi-dimensional array that can be represented using the *torch.Tensor* class. Tensors can be created from a Python list, a NumPy array, or directly from data. PyTorch tensors are similar to NumPy arrays, but they can be easily moved to a GPU for faster computation. Tensors also support automatic differentiation, which makes it easy to compute gradients during the backpropagation step of training neural networks.

PyTorch provides a wide range of functions for manipulating tensors, such as concatenation, reshaping, slicing, and indexing. These functions make it easy to transform tensors into the desired shape and format for building and training neural networks.

Below there is a list with a brief description of the packages that were useful for the development of this project:

- the ***torch*** package contains data structures for multi-dimensional tensors and defines mathematical operations over these tensors. Additionally, it provides many utilities for efficient serialization of Tensors and arbitrary types, and other useful utilities. It has a CUDA counterpart (the *torch.cuda* sub-module), that enables you to run your tensor computations on an NVIDIA GPU with compute capability >=

3.0. In the next paragraph, some modules will be presented, including *torch.utils.data*, *torch.nn* and *torch.optim*;

- the ***torchvision*** package consists of popular datasets, model architectures, and common image transformations for computer vision. It provides support for image and video datasets, pre-trained models, and image processing tools such as data transforms. It is designed to work seamlessly with PyTorch, allowing users to easily incorporate image and video data into their deep learning pipelines. Among all the modules provided by this package, *torchvision.io* and *torchvision.transforms* will be described.

During the development that lead this project to the building of the multimodal neural networks, all the modules cited above have been used: in the next sections they will be described, following the order of use.

3.5.1 More on *torch.utils.data* package

The process started with the definition and usage of some classes for data loading and preprocessing purposes up to training and testing phases. In fact this package provides an easy way to handle and operate with data. It includes several classes and functions to create custom datasets and data loaders. The most important primitive provided by PyTorch is *torch.utils.data.Dataset*, which allows us to use pre-loaded datasets as well as external data. It is an abstract class which enables the creation of *map-stye datasets*, i.e. all datasets that represent a map from keys to data samples should subclass it. All subclasses should overwrite some methods: *___init___()* is executed once when instantiating the *Dataset* object, *___getitem___()* for fetching a specific data sample for a given key, and optionally *___len___()*, which is expected to return the size of the dataset. The *Dataset* object retrieves features and labels one sample at a time: however while training a model, typically it is needed to pass samples in batches, reshuffling the data at every epoch to reduce overfitting and use Python's multiprocessing to speed up data retrieval: *torch.utils.data.DataLoader* allows to abstract this complexity. It combines a dataset and a sampler, and provides an iterable over the given dataset itself to enable easy access to the samples. In addition to the *Dataset* object,

the most important parameters that it takes in input are *batch_size* that specifies how many samples per batch to load, *shuffle* to indicate whether reshuffle data at every epoch and *num_workers* that indicates how many subprocesses have to been used for data loading (e.g. if 0, the data will be fully loaded in the main process).

Another utility class that has been useful for this project is *torch.utils.data.ConcatDataset*, that assemble multiple datasets into a single one: it takes in input a list of datasets and concatenates them along the samples dimension. It has been helpful in the experiments that involve the usage of image transformation techniques on the training set: when applying different transformations on the same set of data, it has been used the *ConcatDataset* module to concatenate the resulting transformed datasets. This allowed to easily create a larger and more diverse dataset by applying different combinations of transformations, without having to manually merge the transformed data. By using *ConcatDataset*, we were able to efficiently combine these different transformations on the same dataset, resulting in a more varied and robust training set.

3.5.2 More on *torchvision.io* package

To deal with the images of TV banners, the *torchvision.io* package has been fundamental: it provides functions for performing IO operations. They are currently specific to reading and writing video and images. This package includes several submodules, each with its own functionalities: for handling the abovementioned images, the *torchvision.io.read_image* has been used: it can read JPEG or PNG image files into a 3 dimensional RGB Tensor. The values of the output tensor are in the range 0-255. The input to this function is the path to the image file and the tensor that it returns is of size (*C, H, W*) where C is the number of color channels (1 for grayscale and 3 for RGB), H is the height of the image, and W is the width of the image. It optionally converts the image to the desired format, that can be specified through the *torchvision.io.ImageReadMode* class: it supports various modes. The one used in this project is *ImageReadMode.RGB*, which indicates that the image should be read as a 3-channel RGB image, meaning that the image has three color channels (Red, Green, Blue) and each pixel

in the image is represented by a 3-tuple of integer values between 0 and 255, specifying the intensity of each color channel.

3.5.3 More on *torchvision.transforms* package

torchvision.transforms is a PyTorch package that provides a set of common image transformations for use in deep learning. It allows to apply various image augmentation techniques to input images during the training process, which can help to improve the performance of deep learning models. In addition to training data, data augmentation can also be applied to validation and test data to improve the accuracy and robustness of the model. This is particularly useful in situations where the validation and test data have different characteristics than the training data, or for some other particular cases related to the task, to the model or to the dataset, like an imbalance of dataset classes. The package delivered by *torchvision.transforms* allows to chain together image transformations using the *torchvision.transforms.Compose* class, which takes in a list of transformations and returns a callable object that can be applied to images. Most transformations accept both *Python Imaging Library* **(PIL)** [27] images and tensor images, although some transformations are PIL-only and some are tensor-only: several functions, like *torchvision.transforms.ToPILImage* or *torchvision.transforms.ToTensor*, allow the conversion, making possible the use of all transformations. Among the numerous transformations provided by this package, here is the list of those that have been applied in this thesis project:

- **ColorJitter**: randomly change the brightness, contrast, saturation and hue of an image;
- **Resize**: resize the input image to the given size;
- **RandomHorizontalFlip**: horizontally flip the given image randomly with a given probability;
- **RandomVerticalFlip**: vertically flip the given image randomly with a given probability;
- **RandomRotation**: rotate the image by angle;

- **Normalize**: normalize a tensor image with mean and standard deviation;

The first one is part of the *Color* group of transformations: these modify the color and the graphical properties of an image, to make the model more robust to variations in lighting conditions and color schemes. The second, third and fourth transformations belong to the *Geometry* group: these transformations modify the geometry of an image by changing its size, orientation, and flipping it horizontally or vertically. The last one is part of *Miscellaneous* group, that contains a set of additional transformations that do not fit under the "Geometry", "Color" or other categories.

It is important to note that this is just a summary list, and also that the transformations have been applied in different orders and/or different times to different batches of data. The details of the transformation pipeline used in this project will be provided at section 4.4.1 .

3.5.4 More on *torch.nn* package

One of the functions and modules included in *torch*, the *torch.nn* module provides an interface for building neural networks. This module is built on top of the *autograd* module, which provides automatic differentiation for all operations on Tensors in PyTorch. It provides a set of predefined layers, such as **convolutional layers, linear layers,** and **recurrent layers**, that can be used to build neural networks. The *nn* module is designed to be flexible and modular, making it easy to create complex architectures for different types of problems. It allows you to build and train neural networks with just a few lines of code, and provides tools for monitoring and evaluating the performance of your models.

One of the key features of *nn* is the **Module** class, which is the base class for all neural network modules in PyTorch: your models should also subclass this class. The **Module** class provides methods for defining and initializing the parameters of a neural network, as well as for forward and backward propagation of data through the network.

In *torch.nn* module, there is the *Sequential* module, that allows to build neural networks by stacking layers in a linear way. It is a subclass of

Module, and is essentially a **container** for other *Module*s: when you define a *Sequential* object, in fact, you pass a sequence of *Module*s to it, that will be added to it in the order they are passed in the constructor.

As anticipated before, **convolutional layers** can be implemented trough the *torch.nn* module: they are commonly used for image and video processing. For this project, they have been implemented using *torch.nn.Conv2d module*, that applies a 2D convolution over an input signal composed of several input planes. The module takes as input a tensor of shape (N, C_{in}, H, W), where N is the number of samples in a batch, C_{in} is the number of input channels or input planes, and H and W are the spatial dimensions of the input. This module requires different arguments, including the number of output channels, kernel size, stride, padding: the number of output channels is equivalent to the number of filters or kernels used to *convolve* the input, the kernel size is a tuple specifying the height and width of the convolutional kernel, the stride is a tuple specifying the stride of the convolution along the height and width dimensions, the padding is a tuple that controls the amount of padding applied to the input.

After each convolutional layer, a *Rectified Linear Unit* (**ReLU**) activation function is applied, in order to introduce **non-linearity** to the output. Convolutional layers apply a linear operation (the *convolution*) to the input, and applying a linear activation function, the output will still be linear. In contrast, the ReLU function introduces a non-linear element by taking the output of a layer and applying a simple element-wise thresholding operation, where any value less than zero is set to zero and any value greater than or equal to zero is left unchanged. Mathematically, the ReLU function can be defined as:

$$ReLU(x) = max(0,\ x)$$

The use of ReLU helps to make the model more expressive and capable of learning complex non-linear relationships between the input and output. Additionally, the ReLU function has the advantage of being computationally efficient and easy to optimize during training, which makes it a popular choice in deep learning architectures.

After the convolutional operation, in order to conduct dimensionality re-

duction and to reduce the number of parameters and computations required for the subsequent layers, it is common to use **pooling layers**: the *torch.nn.MaxPool2d* module is one of the most commonly used. It takes as input a tensor of shape (*N, C, H, W*), where N is the number of samples in a batch, C_{in} is the number of input channels or input planes, and H and W are the spatial dimensions of the input. It performs the max pooling operation over a rectangular window of *kernel_size x kernel_size*, moving the window by a stride of stride. The output tensor has the same number of channels as the input tensor, but with smaller height and width. The values in the output tensor are the maximum values within each window.

To prevent overfitting, that occurs when the model is too complex and starts to memorize the training data instead of learning general patterns that may be applied to new data, a good solution is to use **dropout layers**. Dropout is a regularization technique that during training randomly zeroes some of the elements of the input tensor with probability p: this prevents individual neurons from being excessively dependent on other neurons and ensures that the network learns more robust and generalizable features. This technique has proven to be an effective method for regularization and preventing the co-adaption of neurons [28], and can be implemented using *torch.nn.Dropout*: it is a general-purpose layer that can be used with any type of layer in a neural network. It requires one argument, p, which is the probability of dropping out a neuron (by default it is set to *0.5*). During training, each neuron in the input tensor is set to zero with probability p and the output of the remaining neurons needs to be adjusted to compensate for the missing ones. This is done by scaling up the output of the remaining neurons by a factor of *1/(1-p)*.

Since this thesis project also concerns about text input, in the multimodal neural network that is going to be build there will be a branch made of **linear layers**: the most common approach is to first represent the text as a vector, such as through word embeddings or other text encoding techniques, and then pass this vector through one or more linear layers to perform operations on the text data. This is the reason behind our embedding operation through the *Sentence Transformers* model presented before. In PyTorch, a linear layer is implemented through the *torch.nn.Linear* mod-

ule. It is a fully connected layer where each input neuron is connected to each output neuron by a weight, and a bias term is added to the output. It takes two arguments as input: *in_features* is the size of each input sample, while *out_features* is the size of each output sample. If the input tensor is of shape (*batch_size, in_features*), the output tensor will be of shape *batch_size, out_features*). This type of layers performs a matrix multiplication between the input tensor and the weight matrix, and adds the bias term to the result (*bias* is an optional Boolean argument of this layer, indicating whether or not to include a bias term in the linear transformation). The output is then passed through an activation function, which is usually a non-linear function like ReLU.

Sigmoid [29] functions are commonly used for the task of binary classification because they map any real-valued number to a value between 0 and 1, which can be interpreted as a probability of belonging to a certain class. If the output is greater than 0.5, we can interpret that as a prediction for the positive class, and if it is less than 0.5, we can interpret that as a prediction for the negative class. The module provided by PyTorch for this is use case *torch.nn.Sigmoid*.

The *nn* module also includes loss functions, which are used to calculate the difference between the predicted and target values during training. The goal is to minimize this difference during the training process to improve the accuracy of the model. For binary classification tasks, the most commonly used loss function is *Binary Cross-Entropy* (**BCE**) [30] loss. It is suitable for this type of tasks because it measures the difference between the predicted probability of the positive class and the actual label (0 or 1) of the input.

The last used module of this package is *torch.nn.DataParallel*, that is a PyTorch module that allows to parallelize the training of a neural network across multiple GPUs. It works by replicating the same model on each GPU and splitting the input batch across these replicas. Each replica then computes the loss and gradient of the parameters for a part of the input batch. During the backward pass these gradients are accumulated and averaged across all replicas to update the shared model parameters. This module is useful for speeding up the training of neural networks with a lot

of parameters, as it allows you to leverage the computing power of multiple GPUs in parallel. It is also relatively easy to use: you simply wrap your model in the DataParallel module and specify the list of GPU devices to use.

3.5.5 More on *torch.optim* package

The *torch.optim* is a package implementing various optimization algorithms for training machine learning models. It includes popular optimization algorithms such as *Stochastic Gradient Descent* [31] (**SGD**) and *Adam* [32]: they have been chosen for this project, since the *state-of-the-art* works related to this use case used to implement these types of optimizer.

The optimization algorithms are used to update the model parameters during the training process. Most commonly used methods are already supported, and the interface is general enough, so that more sophisticated ones can also be easily integrated in the future. It also allows users to customize the optimization process by specifying various hyperparameters for the algorithms (including learning rate, momentum, weight decay and more). To use this package, you have to *construct an optimizer* object that will hold the current state and will update the parameters based on the computed gradients. All optimizers implement a *step()* method, that updates the parameters.

The *torch.optim.lr_scheduler* module provides several classes for adjusting the learning rate of an optimizer during training. The learning rate is a hyperparameter that controls how much the model weights are updated during each step of the optimization process: learning rate scheduling should be applied after optimizer's update. In machine learning it is often beneficial to gradually reduce the learning rate during training. This can help the model converge more effectively by allowing it to make larger updates to the weights in the early stages of training and smaller updates as it approaches a local minimum. This can improve the overall accuracy of the model and prevent it from getting stuck in a suboptimal solution. This module provides several classes for adjusting the learning rate during training. The classes choosen for this thesis project are *torch.optim.lr_scheduler.StepLR* and *lr_scheduler.ReduceLROnPlateau*. The first one updates the learning

rate in a more mechanically way, it requires in input the *optimizer* and *step_size*, an integer that indicates the period of learning rate decay: it is going the decay the learning rate by *gamma* (another parameter, set to 0.1 by default) every *step_size* epochs. So this approach can be defined as mechanic since it is not adapting to the current behaviour of the model: it does not matter if the model is learning with that specific configuration of parameters or not, after *step_size* epochs the learning rate is updated. The *ReduceLROnPlateau* scheduler is more adaptive: it reduces the learning rate when a metric has stopped improving, for example when the accuracy on the validation set is not improving and the learning is stagnating. In particular, if no improvement is seen for a '*patience*' number of epochs, the learning rate is reduced. This class takes in input the *optimizer*, the *mode* that indicates in which direction the quantity monitored should improve (i.e. it could be "min" or "max. If *mode = "min"*, the learning rate will be reduced when the monitored quantity has stopped decreasing, and viceversa with "max"), the *factor* by which the learning rate will be reduced (new_lr = lr*factor, that is set to 0.1 by default), the *patience* integer that indicates the number of epochs with no improvement after which learning rate will be reduced, and some other optional parameters for a more customizable learning rate update.

3.6 Ray Tune

Ray is an open-source distributed computing framework that allows developers to easily scale their Python applications and machine learning workloads. *Ray* provides a simple and flexible API that enables distributed computing with minimal code changes, allowing developers to focus on building their applications rather than managing the underlying infrastructure.

Ray includes several libraries for various distributed computing use cases, including RaySGD for distributed training of deep learning models, Ray Serve for building scalable and high-performance APIs, and *Ray Tune* for hyperparameter tuning of machine learning models: this is a library built on top of Ray that provides a simple and scalable solution for hyperparameter tuning of machine learning models. It includes a range of state-of-the-art

algorithms for hyperparameter optimization, including grid search, random search, and various forms of Bayesian optimization. *Ray Tune* also provides a flexible and extensible API that allows developers to easily integrate it into their existing machine learning workflows.

To optimize your hyperparameters, there is the need to define a search space. A search space defines valid values for the hyperparameters and can specify how these values are sampled (e.g. from a uniform distribution or a normal distribution). The configuration for the hyperparameter search is specified using a dictionary that maps the names of hyperparameters to their possible values.

Among the utility classes provided by *Ray Tune*, in this thesis project *ray.tune.CLIReporter* has been used: it provides a way to print a summary of the training progress in the console during a *Tune* experiment. It generates live-updating tables that display the current training progress, including metrics such as loss, accuracy, and any custom metrics you specify.

To use *CLIReporter*, there is the need to instantiate the class and pass it as an argument to your *tune.run* call, along with other parameters such as your training function, the number of trials, and the search space for hyperparameters.

The CLIReporter class provides several configuration options that allow you to customize the output of the progress table. For example, you can specify the metric to use for sorting the trials, customize the output format of the table, and control how frequently the table updates.

In particular *tune.run* is a function provided by *Ray Tune* that allows you to easily launch a hyperparameter search. It can take different parameters in input:

- **run_or_experiment**: this parameter specifies the training function or the experiment configuration, and it can be a function or a string reference to a function;
- **resources_per_trial**: this parameter specifies the resource requirements for each trial. It can be specified as a dictionary of resources such

as CPU, GPU, memory, etc. For example, *resources_per_trial*="cpu": 2, "gpu": 1 specifies that each trial should use two CPUs and one GPU;

- **config** this parameter specifies the hyperparameter search space and default values for each parameter. It can be specified as a dictionary, where each key is a hyperparameter name, and the corresponding value is a list of values or a *tune.sample()* function that generates values.
- **num_samples** this parameter specifies the number of times to sample from the hyperparameter space, i.e. the number of trials to run. It is an integer, it is set to 1 by default and if this is *-1* means that it will run trials until stopped manually;
- **progress_reporter** this parameter specifies how the results of each trial are reported. It can be an instance of a *tune.reporter.Reporter* class.
- **local_dir** this parameter specifies the directory where the results of each trial are stored. It can be a local or remote file path, and it defaults to a temporary directory if not specified.

These parameters can be used to customize the hyperparameter search process in *Ray Tune* and adapt it to various use cases and environments. It then runs the training function multiple times with different hyperparameter configurations provided and returns the best set of hyperparameters found during the search.

3.6.1 Installation

The following command is used to getting started with *Ray Tune*:

```
pip install ray
```

3.6.2 Usage

In order to execute the *tune.run* function, it is needed first of all to define the training function. In the next chapter, at 4.4.3 section, there will be the specific code snippet.

Chapter 4

The approaches

The goal is to recognize TV banners and the proposed approaches take into account four different scenarios to do this. They are tagged with different labels, depending on the presence of textual information and on the application of data augmentation techniques on the training set:

- **NOT_NOA**: the first approach tries to reach the goal **without the textual information** (*No Text* $\approx$ **NOT**). This attempt wants to reveal if and in which manner the textual part can affect the overall performance of the model. **NOA** instead stands for **no-augmentation**: in this case, in fact, the training set will not be augmented;
- **NOT_A**: also the second approach **does not take into account the textual information** (**NOT**), but **some image transformations (A) will be applied** on the training set in order to augment the amount of data, introduce variability and make the model more robust to changes in input images;
- **T_NOA**: this approach is the opposite of the previous one. Here the **textual information is taken into account (T)**, but **no augmentation techniques** will be applied on the training set (**NOA**);
- **T_A**: in this last approach, the **textual information is part** (**T**) of the experiment, and **some image transformation (A) will be applied** on the training set.

Depending on the approach, the Neural Network will be built differently and also the custom dataset class will take different parameters in input. The differences will be handled thanks to two simple *Boolean variables* used as flags to indicate whether the textual information has to be taken into account or data augmentations has to be applied. For example, if the approach is *NOT_NOA*, this will be the first code snippet executed before

all the others:

```
text = 0
augm = 0
```

In the following sections, it will be possible to see how these flags are used and which are the differences in the implementation.

4.1 About the dataset

The dataset[1] contains 220 labelled samples of different TV banners: 153 of them are banners placed on the left side or bottom of the screen (figure 4.1), 55 of them are designed with an L-shaped pattern (figure 4.2), and 12 of them are banners of the same size as the entire screen which leave a central part to the content (figure 4.3).

Figure 4.1: A classical banner

Figure 4.2: An L-shaped banner

Figure 4.3: Another type of banner

[1] *Kineton s.r.l. specifies that all the logos in the images are reference, not customers: they were used only to train and test the model for the purposes of research.*

As it was, the dataset was composed of 158 **self-promotion** banners and 62 **traditional media** ones: it needs some cleaning operations and some augmentation techniques due to the reasons that will be explained below.

4.1.1 Data cleaning

The first operation performed on the dataset has been a cleaning operation: in fact it contained some *similar* banners that differed only by some graphic changes. This operation was essential in order to ensure a good degree of variation of the images in the dataset: having many similar images could lead to bad performance of the model. So, depending on the relevance of this difference, it was decided to keep the banners in the dataset or delete them.

To be more precise, we may have two possible cases:

- in the event that in two or more *similar* banners were present textual information, and that the two or more banners taken into consideration differed for one or a few words, **only one** of the two banners would be **maintained**, **discarding the other one** (or the others); *(see the examples at Figures 4.4, 4.5)*
- in the case of a significant difference, such as the presence of elements arranged differently, whole sentences, graphic elements (i.e. logos) or even different people who were partially, totally absent, or present in a different way in the other banners taken into consideration, then it was decided to **keep them**. *(see the examples at Figures 4.6, 4.7)*

Figure 4.4

Figure 4.5

Figure 4.6

Figure 4.7

4.1.2 Some data extraction

In the original dataset there were also some *GIFs*: it was decided to extract the most significant frames from each of them, always taking into account the graphical difference between them. So it was decided to sample frames quite different from each other (according to the criterions described above), in order to maintain high variance of the dataset images. *(In the following group of images 4.8, 4.9, 4.10, 4.11, there are some examples of different frames extracted from the same GIF).*

Figure 4.8

Figure 4.9

Figure 4.10

Figure 4.11

4.1.3 A final report

After the two operations explained above, the dataset consisted of 167 TV banners, distributed in this way:

- 128 images are **self-promotion** (around 76% of the overall dataset);
- 39 images are **traditional media advertising** (around 24% of the overall dataset).

At this point the dataset was ready for all the steps required for the final binary classification.

4.2 Text extraction

For this task, a pretrained model has been used in order to detect and recognize text from the TV banners. With the following command it has been possible to automatically download pretrained weights for the detector and recognizer: *Pipeline(* in fact enables to instantiate an object of a wrapper class for a combination of text detector and recognizer.

```
1 import keras_ocr
2
3 pipeline = keras_ocr.pipeline.Pipeline()
```

At this point, two different functions were defined in order to handle the text extracted from the TV banners.

The main function is *OCR_text_recognition*. It needs different parameters:

- *pipeline* is needed to detect and recognize words;
- *path_to_images* is the path to the folder where are the TV banners are stored;
- *min_width_boxes* and *min_height_boxes* are two important parameters that have been used in order to discard the smaller boxes. As said before, in fact, *keras-ocr* draws boxes around each word detected. However in order to achieve an higher number of correctly recognized words, only the words that are *big enough* (and for this reason more

recognizable), have been extracted, i.e. only the words whose boxes dimensions were compliant to *min_widht_boxes* and *min_height_boxes* have been saved. The process behind this filtering operation will be explained in a few lines;

- finally *images_list* is the last argument, that as the name suggests is the list of *png* images.

```
import matplotlib.pyplot as plt
import os
import os.path
import re


def OCR_text_recognition(pipeline, path_to_images,
    min_width_boxes, min_heigth_boxes, images_list):
  # list that will contains the filenames
  filenames = []


  if (images_list is not None):
    for fn in images_list:
      image = keras_ocr.tools.read(fn)
      filenames.append(fn)

  # Each list of predictions in prediction_groups is a list
   of (word, box) tuples.
  prediction_groups = pipeline.recognize(images_list)

  # The dictionary that will contain all the informations
   extracted from the images
  # It will be composed in this way: the key is the filename
   , the value is the list of words
  # EXAMPLE: {0_1: ['word1', 'word2'], 0_2: ['wordx', 'wordy
   '], 1: ['myword', 'yourword'], ...}
  all_texts = dict()
  # The list of words for each image
  text_on_each_img = []


  img_key_index = 0
  img_key = ""
```

```
30  # Plot the predictions
31  fig, axs = plt.subplots(nrows=len(images_list), figsize
     =(20, 20))
32  for ax, image, predictions in zip(axs, images_list,
     prediction_groups):
33    # from extracted test, let's cut the smallest ones
34    cutted_predictions = cut_small_text(predictions,
     min_width_boxes, min_heigth_boxes)
35    text_on_each_img = []
36    for j in range (0, len(cutted_predictions)):
37      # in the list, append each word detected
38      text_on_each_img.append(cutted_predictions[j][0])
39    # img_key is obtained from the filenames list
40    # it will be the string between the path and the format
     file
41    # i.e. /content/drive/MyDrive/CALL4THESIS/images/-->0_1
     <--.png
42    img_key = re.search(path_to_images + '(.*)' + ".png",
     filenames[img_key_index]).group(1)
43    all_texts[img_key] = text_on_each_img
44    img_key_index += 1
45    # draw boxes and annotate big-enough words on each image
46    keras_ocr.tools.drawAnnotations(image=image, predictions
     =cutted_predictions, ax=ax)
47
48  return all_texts
```

The first three lines import the necessary Python libraries for the function: *matplotlib* is a plotting library that is used for visualizing the OCR text recognition output, the *os* library provides a way to interact with the operating system, and the *re* library is used for regular expressions.

The main thing occurs at line 18: it uses the *recognize()* method of the OCR pipeline object to extract text from the images in *images_list*. The *recognize()* method returns a list of *predictions*, where each prediction is a list of tuples containing the detected word and the bounding box coordinates of the word in the image.

At line 32, the function loops through each image in *images_list* and its corresponding OCR predictions. It uses a function called *cut_small_text*, that as anticipated before, is useful to remove any detected words that

have a bounding box smaller than the specified *min_width_boxes* and *min_heigth_boxes* dimensions. It then extracts the words from the remaining OCR predictions and appends them to the *text_on_each_img* list.

Next, at line 42, the function extracts the *filename key* for the current image being processed from filenames, using a regular expression to match the pattern of the path. It then adds the words for that image to the *all_texts* dictionary using the filename key.

Then, the function plots the bounding boxes and the annotated words for each image using the *drawAnnotations* function from the *keras_ocr.tools* module. That functions draws text annotations onto image, and takes three arguments as input: the image itself on which to draw, the *predictions* list object provided by *pipeline.recognize()*, and a *matplotlib axis* on which to draw.

Finally the main function returns the *all_texts* dictionary containing the extracted text information for all the processed images.

Here below, an example of a TV banner outputted by the *keras-OCR* tool, also after the preprocessing function just described:

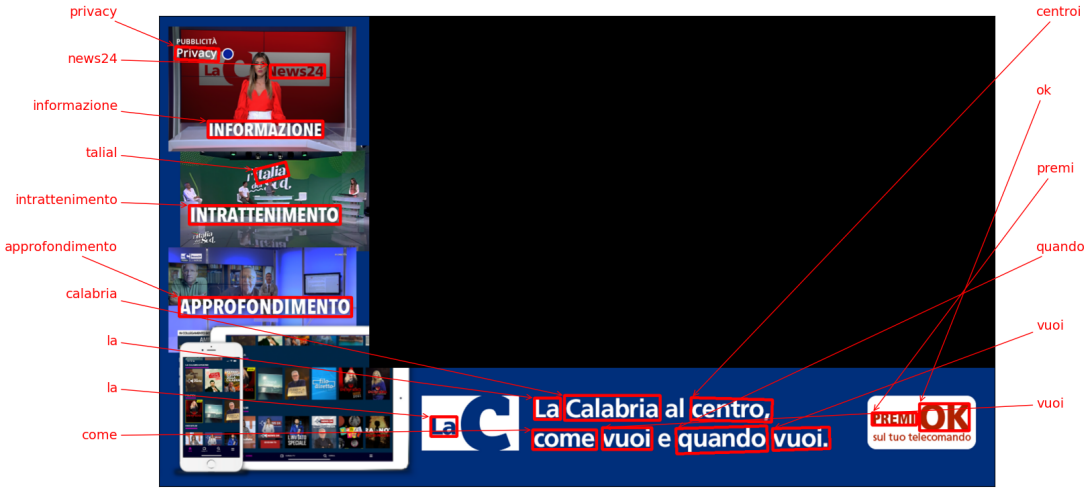

Figure 4.12: A TV banner run through OCR

It is possible to see that the *keras-OCR* model has correctly recognized a good number of words with good accuracy. The majority of the words contained in this banner are written with an easily readable font, which

facilitates the detection and recognition of the text: some words are also white on a dark background, which is another plus point too. It is also possible to notice that in the left part of the banner, there are photos of some journalists during some television broadcasts. Behind them there is a screen with some words, and the model managed to correctly recognize the words quite large, recognizing however poorly a word written slightly oblique. This depends not only on the layout of the word, but also depends on the font. In fact it is possible to notice in figure 4.13, that the model is able however to recognize with a good accuracy words that fall within this use case.

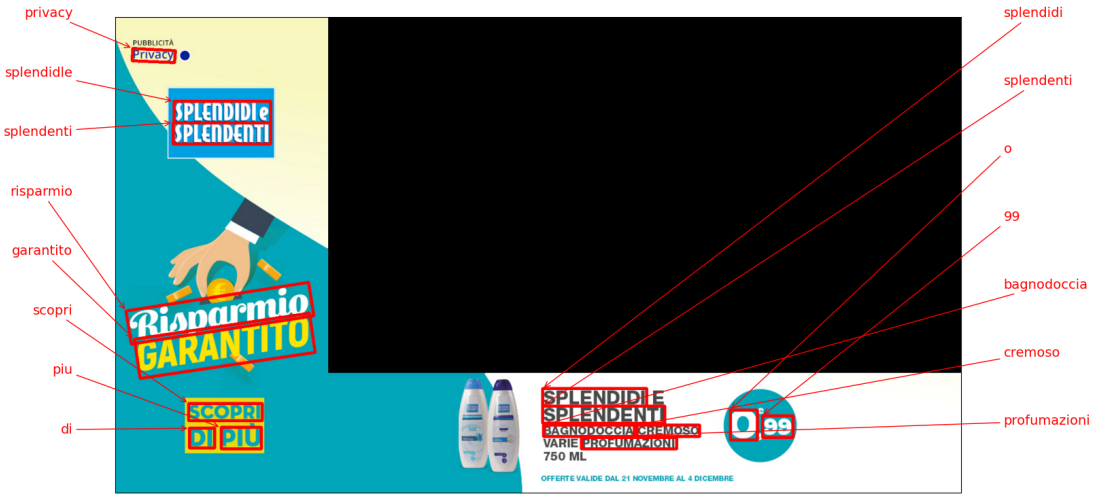

Figure 4.13: Another TV banner run through OCR

However, some words, as we can see in the bottom right of the TV banner in figure 4.12, but also some others in the lower-central part of figure 4.13, have not been detected: this is obviously not a bug, but a feature, as the behavior achieved is wanted. As anticipated before, in fact, within the main function just explained above, the function *cut_small_text* is invoked. This takes three arguments as input:

- *predictions* is a list of (word, box) tuples, where each box is a list of four points, with two coordinates for each one. Each box is drawn around the corresponding word;
- *min_width_boxes* is a threshold value, i.e. the minimum width of the box;
- *min_height_boxes* is another threshold value, i.e. the minimum height

of the box.

```
def cut_small_text(predictions, min_width_boxes,
    min_heigth_boxes):

  cutted_predictions = []

  for prediction in predictions:

    # if sides are long enough: draw box
    if(prediction[1][1][0] - prediction[1][0][0] >
   min_width_boxes and prediction[1][3][1] - prediction
   [1][0][1]> min_heigth_boxes):
      # print("long enough")
      cutted_predictions.append(prediction)

  return cutted_predictions
```

The function uses a *for* loop to iterate over each tuple in the *predictions* list (that is a list of (word, box) tuples where each box is a list of 4 points, with 2 coordinates for each one). To be clearer, it is a *list of lists of lists*, composed in this way:

- the first index level stores the four lists of coordinates for each tuple, i.e. the list *prediction[1]* will contain the lists of coordinates of the 1st box, *prediction[2]* will contain the lists of coordinates of the 2nd one, and so on...;
- the second index level stores the two coordinates for each point of the specific box, i.e. the list *prediction[1][0]* contains the coordinates of the first top-left point (labeled with A in the figure 4.14), *prediction[1][1]* contains the coordinates of the second top-right point (labeled with B in the figure 4.14), *prediction[1][2]* contains the coordinates of the third bottom-right point (labeled with C in the figure 4.14) and *prediction[1][3]* contains the coordinates of the fourth bottom-left point (labeled with D in the figure 4.14);
- the third and last index level stores the coordinates (X and Y values) of each point of the box, i.e. the values *prediction[1][0][0]* and *prediction[1][0][1]* are respectively the X and Y value of the point labeled

with A in the figure 4.14.

After this explanation it is possible to better understand what is happening within the function *cut_small_text* that was being explained a few lines above: inside the loop, the function checks whether the width (AB or DC in the following figure), and height (AD or BC) of each box, calculated through the difference between the coordinates of the respective points for the current tuple, are greater than the specified minimum thresholds. If the conditions are true, the current tuple is appended to the *cutted_predictions* list. Finally, the function returns the list, which contains only the tuples that meet the specified criteria.

Figure 4.14: The coordinates of a text box

The text extracted from the TV banners has been stored in a *csv* file for easily access and process it in the next steps.

```
1 import csv
2
3   # text_extracted is dictionary that will contain all the
     informations extracted from the images
4   # It will be composed in this way: the key is the filename
     , the value is the list of words
```

```
  # EXAMPLE: {0_1: ['word1', 'word2'], 0_2: ['wordx', 'wordy
    '], 1: ['myword', 'yourword'], ...}

with open('OCR_text.csv','w') as f:
    w = csv.writer(f)
    w.writerows(text_extracted.items())
```

This code is responsible for writing the information extracted from images into a CSV file called *"OCR_text.csv"*.

The first line of the code imports the *csv* module, which provides functionality to read from and write to CSV files.

The code then opens the *"OCR_text.csv"* file in write mode using the '*with*' statement, which ensures that the file is properly closed after writing to it is complete. The *"csv.writer"* function is then used to create a writer object that can write to the CSV file.

The '*writerows*' method is then called on the writer object, passing in the '*items*' method of the '*text_extracted*' dictionary as an argument: this structure stores all the information extracted from the images in a key-value pattern. The key is intended to be the name of the image, and the value is the corresponding text extract from the image. So the function writes all the information from the '*text_extracted*' dictionary to the CSV file in the format of key-value pairs.

The following figure 4.15 shows how the generated csv file will look like:

filename	sentence
0	['court', 'v']
19	['lno', 'esiste', 'amaro', 'perfetto', 'tedesco', 'dal', 'cuore', 'delli', 'aspromonte']
44	['forniamo', 'soluzioni', 'aziendali', 'far', 'crescere', 'tuo', 'business', 'u', 'tecnologifueficio', 'soluzioni', 'dazienda']
45	['trony', 'cosenza', 'trony', 'rende', 'caffe', 'buon', 'gusto', 'regalo', 'offre', 'trony', 'zaprile']
47	['trony', 'cosenza', 'trony', 'rende', 'regalo', 'cio', 'acquista', 'che', 'vuol', 'scegli', 'lo', 'tua', '39ge', 'partire']
75	['trony', 'cosenza', 'trony', 'rende', 'caro', 'ilmeno', 'meta', 'dal', 'maggio', 'paghila', 'lo']
81	['hcosenza', 'hrende', 's', 'a', 'scontoiva', 'dal', '26', 'maggio', 'glugno']
82	['artese', 'arte', 'see', 'vino', 'setfra', 'esetra', 'esetra', 'zlimanwi']
83	['fedelta', 'convenet', 'piu', 'scopri', '599', '269', '0999', '169', '199']

Figure 4.15: CSV file storing text extracted for each TV banner

4.3 Text embedding

For the use case of this thesis project, the multilingual model chosen from the *Sentence Transformer* framework in order to generate sentence embeddings is *"sentence-transformers/paraphrase-multilingual-mpnet-base-v2"*.

```
from sentence_transformers import SentenceTransformer
import csv

with open('OCR_text.csv', newline='') as f:
    reader = csv.reader(f)
    sentences = list(reader)

model = SentenceTransformer('sentence-transformers/
   paraphrase-multilingual-mpnet-base-v2')

embeddings = model.encode(sentences.values())

# print(embeddings)
```

After importing the required libraries, the code opens the *OCR_text.csv* file using *open()* function and *csv.reader()* to read its contents. The *newline=”* parameter is used to handle any newline characters that may be present in the CSV file. The *list()* function is then used to convert the CSV reader object reader into a list of sentences, which is stored in the variable sentences. This list contains all the sentences extracted from the OCR process, with each sentence represented as a list of words.

Then the function loads the multilingual model specified a few lines above, and finally encodes the sentences using the loaded model by passing *sentences.values()* to the *encode()* function of the model object. The resulting sentence embeddings are then stored in the *embeddings* variable.

The output type of *encode()* function could be also a *np.ndarray*, that is an n-dimensional array object from the *NumPy* library. So finally, with the following code snippet, it is possible to save the array as a binary *.npy* file with the name *embedded_sentences.npy*: the *NumPy* function *save()*, in fact, is used to write the array to file.

```
# save numpy array as npy file
```

```
from numpy import save

# save to npy file
save('embedded_sentences.npy', embeddings)
```

4.4 First run

The first run has been executed on the dataset as it is: 167 samples, 128 of which are *Internal* and 39 *External* ones.

4.4.1 The custom Dataset definition

At this point, it is possible to prepare the dataset for the task, using the presented libraries and tools for data preprocessing and analysis.

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
```

These lines import the *Pandas* and *NumPy* libraries, which are commonly used for data manipulation and analysis in Python. This snippet also imports the *LabelEncoder* class from the *scikit-learn* library's *preprocessing* module. *LabelEncoder* is used for encoding categorical variables as numeric variables. The last line of that snippet imports the *train_test_split* function from the *scikit-learn* library's *model_selection* module. *train_test_split* is used for splitting data into training and testing sets.

```
full_dataset = pd.read_csv('/content/OCR_text.csv')
full_dataset.info()

X, y = full_dataset.iloc[:, :-1], full_dataset.iloc[:, [-1]]

X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.2, stratify = y)

X_train, X_val, y_train, y_val = train_test_split(X_train,
    y_train, test_size=0.2, stratify = y_train)

X_train, X_val, X_test = X_train.reset_index(drop = True),
```

```
11                              X_val.reset_index(drop = True),
12                              X_test.reset_index(drop = True)
13 y_train, y_val, y_test = y_train.reset_index(drop = True),
14                              y_val.reset_index(drop = True),
15                              y_test.reset_index(drop = True)
```

From the previous steps, the dataset was stored in a CSV file: the first step consists in loading it from its location using the *pd.read_csv()* function from the *pandas* library. The following line allows to display information about the dataset using the *info()* method of the *DataFrame* object.

The fourth line splits the dataset into feature and target variables, where X contains all columns except the last one, which is the target variable and belongs to y. Then there is another data splitting into training and testing sets using *train_test_split()* function from the *sklearn.model_selection* module. The *test_size* parameter is set to 0.2, indicating that 20% of the data should be used for testing. The *stratify* parameter is set to y, meaning that the split is performed in a way that preserves the proportion of samples for each class in y. The usage of this parameter is optional, however taking into consideration the distribution of the classes of this dataset, it has been fundamental to use it: the dataset, in fact, introduced a moderate imbalance, with 128 samples in the class Internal and 39 with the class External. Using *stratify* parameter, it has been possible to preserve the classes distribution both in training and in testing set. So the resulting sets have the following sizes: 141 samples in the training one and 26 samples in the other one.

The snippet further splits the training set into training and validation sets, again using *train_test_split()* function. The *test_size* parameter is set to 0.2, indicating also in this case that 20% of the training data should be used for validation. The stratify parameter in this case is set to *y_train* for a better comprehension and coherence with the current split: the proportion of samples for each class would be preserved also with y, since the distribution is the same thanks to the previous split. After this operation, the resulting sizes are: 115 samples in the training set, 26 in the validation set and 26 in the testing one.

The last operation performed resets the index of the dataframes to en-

sure they are properly aligned for later processing, using the *reset_index()* method of the *DataFrame* object with the drop parameter set to *True*. This ensures that the index of each dataframe starts from 0 and is consecutive. This reset operation was fundamental, since when randomly splitting the dataset into *X_train* and *X_test*, few of the data rows could be moved to *X_test* along with the idx, and the *Dataloader* object, after wrapping an iterable around the *X_train* object, could try to get a sample with a specific index that could be moved into *X_test*.

```
label_enc=LabelEncoder()

y_train.label=label_enc.fit_transform(y_train.label)
y_val.label=label_enc.fit_transform(y_val.label)
y_test.label = label_enc.fit_transform(y_test.label)
```

Here, there is the definition of *label_enc*, an instance of the *LabelEncoder()* class. *y_train.label*, *y_val.label*, and *y_test.label* are the label columns of the train, validation and test dataframes respectively, which contain text labels for the corresponding data samples. The *label_enc.fit_transform()* method is used to encode the labels into numerical values: this method in fact fits the *LabelEncoder()* on the label data and returns an array of encoded labels. Therefore this code snippet encodes the text labels into numerical values and replaces the original text labels with the encoded ones: *Internal* becomes *1*, *External* becomes *0*.

At this point it is possible to define our custom dataset, that should inherit *torch.utils.data.Dataset* class and implement some methods: *__init__*, *__len__*, and *__getitem__*.

```
  class CustomImageDataset(Dataset):
      def __init__(self, features, target, embeddings,
                   transform=None):
          self.features = features
          self.transform = transform
          self.target = target
          self.embeddings = embeddings

      def __len__(self):
          return len(self.features)

```

```
    def __getitem__(self, idx):
        filename = str(self.features.loc[idx, 'filename'])
        img_path = path_to_images + filename + ".png"
        image = read_image(img_path,
                           ImageReadMode.RGB).float()
        label= self.target.loc[idx, 'label']

        if self.transform:
            image = self.transform(image)


        if(text):
          if filename in self.embeddings.keys():
            text_feature = torch.from_numpy(
                           self.embeddings[filename]
                           )
        if(text):
        return image, text_feature, label

        else:
          return image, label
```

The *__init__* method takes these four arguments, which are stored as instance variables, and initializes the dataset with them:

- **features** is a dataframe that contains information about the images to be loaded;
- **target** is a dataframe that contains the corresponding labels for each image; **embeddings** is a dictionary containing the embedded vectors of the text associated with each image;
- **transform** is an optional argument which allows to pass in a list of image transformations to be applied to the images before they are fed into the model.

The *__len__* method returns the length of the dataset, which is equal to the number of images to be loaded.

The *__getitem__* method takes an index *idx* and returns a tuple containing the image, any associated text features, and the target label for the corresponding image.

First, the filename and image path are obtained from the *features* DataFrame. Then, the image is read from the file using the *read_image* function, which returns a PyTorch tensor of shape *(3, H, W)* where H and W are the height and width of the image, respectively. The tensor is then cast to float. The target label for the current image instead is obtained from the *target* DataFrame. If the *transform* argument is not None, then the image is passed through the specified image transformations: in this parameter there will be a torchvision.transforms.Compose object, which takes in a list of transformations.

The last part of the code refers to adding *text features* to the image dataset: it checks if the *text* variable is *True* and if the text *embeddings* for the current *filename* exist in the provided *embeddings* dictionary. If both conditions are met, it extracts the embeddings and converts them to a PyTorch tensor using *torch.from_numpy()*. Finally, it returns a tuple consisting of the *image* tensor, *text feature* tensor, and *label* tensor if *text* is *True*, meaning that the textual information should be taken into account for the current approach. Otherwise, it returns a tuple consisting of only the *image tensor* and *label* tensor.

The *embeddings* dictionary that will be fed to the *CustomImageDataset* objects is a data structure containing the embedded vectors of the associated to each image. These information have been stored in a *.npy* file at section 4.3: in the following code snippet it is possibile to see how to retrieve that information for the next steps.

```
import csv
# load numpy array from npy file
from numpy import load
# load array
embeddings = load('embedded_sentences.npy')

with open('OCR_text.csv', newline='') as f:
    reader = csv.reader(f)
    csv_content = list(reader)

filenames = [data[0] for data in csv_content]
filenames.pop(0)


```

```
15 filename_embeddings = dict()
16 i = 0
17 for fn in filenames:
18   filename_embeddings[fn] = embeddings[i]
19   i+=1
```

This code essentially reads in two files, *"embedded_sentences.npy"* and *"OCR_text.csv"*, and creates a dictionary that maps filenames to their corresponding embeddings.

The first step of the code is to load the embeddings from the first file using the *load()* function from the *NumPy* library. The resulting array of embeddings is assigned to the variable *embeddings*.

Next, the code reads from the '*OCR_text.csv*' file using the *csv.reader()* function from the *csv* library. The resulting data is stored in the *csv_content* variable as a list of lists, where each sub-list contains a **filename** and other information. The reading operation of this file is aimed to obtain just the list of filenames, in order to create then the abovementioned dictionary that maps each filename (the key) to the corresponding embedding vector (the value). This is how the csv file looks like:

filename	sentence
0	['court', 'v']
19	['lno', 'esiste', 'amaro', 'perfetto', 'tedesco', 'dal', 'cuore', 'delli', 'aspromonte']
44	['forniamo', 'soluzioni', 'aziendali', 'far', 'crescere', 'tuo', 'business', 'u', 'tecnologifueficio', 'soluzioni', 'dazienda']
45	['trony', 'cosenza', 'trony', 'rende', 'caffe', 'buon', 'gusto', 'regalo', 'offre', 'trony', 'zaprile']
47	['trony', 'cosenza', 'trony', 'rende', 'regalo', 'cio', 'acquista', 'che', 'vuol', 'scegli', 'lo', 'tua', '39ge', 'partire']
75	['trony', 'cosenza', 'trony', 'rende', 'caro', 'ilmeno', 'meta', 'dal', 'maggio', 'paghila', 'lo']
81	['hcosenza', 'hrende', 's', 'a', 'scontoiva', 'dal', '26', 'maggio', 'glugno']
82	['artese', 'arte', 'see', 'vino', 'setfra', 'esetra', 'esetra', 'zlimanwi']
83	['fedelta', 'convenet', 'piu', 'scopri', '599', '269', '0999', '169', '199']

Figure 4.16: CSV file storing text extracted for each TV banner

The list of filenames is extracted from the *csv_content* list using a list comprehension (line 11) that iterates over each sub-list and extracts the first item (the filename). The first sub-list is removed from the list using the *pop()* function, as it only contains the header of the CSV file.

Finally, at line 15, a dictionary named *filename_embeddings* is created to store the filename-embedding pairs. A loop is used to iterate over each filename and its corresponding embedding, and the pair is added to the

dictionary using the filename as the key and the embedding as the value. The embeddings are retrieved from the *embeddings* array using an index i that is incremented with each iteration. The resulting dictionary allows for easy access to the embeddings during the training process.

An important step to perform on the dataset before training the model is the *dataset normalization* using the *mean* and the *standard deviation*. This is an important operation since data normalization improves the performance of neural networks and helps in faster convergence during training.

In order to calculate mean and standard deviation on the training data, there is the need to wrap them in a *DataLoader* object. The following code snippet shows all the required steps:

```
if(not text):
  filename_embeddings = None

train_images = CustomImageDataset(features = X_train,
               target = y_train,
               transform=transforms.Compose(
               [transforms.ToPILImage(),
               transforms.Resize((256, 256)),
               transforms.ToTensor()]),
               embeddings = filename_embeddings)

data_loader = DataLoader(train_images, batch_size=32,
                         shuffle=True)

def get_mean_and_std(dataloader):
    channels_sum = 0
    channels_squared_sum = 0
    num_batches = 0
    for data in dataloader:

          if(text):
            imgs, text, label = data
          else:
            imgs, label = data

          channels_sum += torch.mean(imgs, dim=[0,2,3])
          channels_squared_sum += torch.mean(imgs**2,
```

```
28                                  dim=[0,2,3])
29
30              num_batches += 1
31
32          mean = channels_sum / num_batches
33
34          # std = sqrt(E[X^2] - (E[X])^2)
35          std = (channels_squared_sum / num_batches - mean ** 2)
        ** 0.5
36
37          return mean, std
38
39  mean, std = get_mean_and_std(data_loader)
```

At line 4 the code creates a *train_images* dataset using the *CustomImageDataset* class with the *X_train* and *y_train* features and target, a *transform* object with some basic image transformations, and an optional *filename_embeddings* variable: it is set to *None* at line 1 in the event that the flag variable *text* is *0*, meaning that the current approach should not take into account the textual information. By setting it to *None*, in fact, no embedded vector will be added to the dataset when creating it: this will be composed only by the *image* and *label* tensors, as seen before. Otherwise, if *text* is *1*, the variable *filename_embeddings* still maintain all the information about images, and also the *textual information* is returned when creating the dataset.

Then at line 12 this snippet creates a *DataLoader* object with the *train_images* dataset, with a batch size of 32 and shuffling the data. This object at line 36 will be passed to the the *get_mean_and_std* function: it calculates the mean and standard deviation of the dataset. It initializes *channels_sum*, *channels_squared_sum*, and *num_batches* to 0 and loops through the dataloader to retrieve batches of data. If the variable *text* is not *0*, it unpacks *imgs*, *text*, and *label* from the batch, otherwise it unpacks just *imgs* and *label*. At line 26 and 27 the *channels_sum* and *channels_squared_sum* of the images are calculated over batch, height and width, but not over the channels (*dim=[0,2,3]*): these are calculated using the *torch.mean* function, that returns the mean value of each row of the input tensor (*imgs* in this case) in the given dimension *dim*. If *dim* is a list of dimensions,

it reduces over all of them). At each iteration the batch count is incremented. The function finally calculates the mean and standard deviation of the dataset using the calculated *channels_sum*, *channels_squared_sum*, and *num_batches*, and return the values.

At this stage there is the initialization of the custom dataset in order to obtain the **normalized training data**: the dataset created in the following snippet, in fact, will be used to train the model.

```
if(not text):
  filename_embeddings = None

if(not augm):
  train = CustomImageDataset(features = X_train,
                             target= y_train,
                             transform=transforms.Compose(
                    [transforms.ToPILImage(),
                    transforms.Resize((256, 256)),
                    transforms.ToTensor(),
                    transforms.Normalize(mean,std)]),
                    embeddings = filename_embeddings)

else:
  data1 = CustomImageDataset(features = X_train,
                             target= y_train,
                             transform=transforms.Compose(
                    [transforms.ToPILImage(),
                    transforms.Resize((256, 256)),
                    transforms.ToTensor(),
                    transforms.Normalize(mean,std)]),
                    embeddings = filename_embeddings)

  data2=CustomImageDataset(features = X_train,
                             target= y_train,
                             transform=transforms.Compose(
                    [transforms.ToPILImage(),
                    transforms.Resize((256, 256)),
                    transforms.ColorJitter(brightness=0.6,
                                           contrast=1.3,
                                           saturation=1.1,
                                           hue=.2),
                    transforms.ToTensor(),
```

```
                    transforms.Normalize(mean,std)]),
                    embeddings = filename_embeddings)

data3=CustomImageDataset(features = X_train,
                             target= y_train,
                             transform=transforms.Compose(
                    [transforms.ToPILImage(),
                    transforms.Resize((256, 256)),
                    transforms.ColorJitter(brightness=1.3,
                                           contrast=.5,
                                           saturation=.5),
                    transforms.RandomHorizontalFlip(p=1),
                    transforms.ToTensor(),
                    transforms.Normalize(mean,std)]),
                    embeddings = filename_embeddings)

data4=CustomImageDataset(features = X_train,
                             target= y_train,
                             transform=transforms.Compose(
                    [transforms.ToPILImage(),
                    transforms.Resize((256, 256)),
                    transforms.ColorJitter(brightness=.5,
                                           contrast=.5,
                                           saturation=.5),
                    transforms.RandomVerticalFlip(p=1),
                    transforms.ToTensor(),
                    transforms.Normalize(mean,std)]),
                    embeddings = filename_embeddings)

data5=CustomImageDataset(features = X_train,
                             target= y_train,
                             transform=transforms.Compose(
                    [transforms.ToPILImage(),
                    transforms.Resize((256,256)),
                    transforms.RandomRotation(degrees=160),
                    transforms.ToTensor(),
                    transforms.Normalize(mean,std)]),
                    embeddings = filename_embeddings)

data6=CustomImageDataset(features = X_train,
                             target= y_train,
                             transform=transforms.Compose(
```

```
                    [transforms.ToPILImage(),
                    transforms.Resize((256,256)),
                    transforms.RandomRotation(degrees = 45),
                    transforms.RandomHorizontalFlip(p=1),
                    transforms.ToTensor(),
                    transforms.Normalize(mean,std)]),
                    embeddings = filename_embeddings)

data7=CustomImageDataset(features = X_train,
                             target= y_train,
                             transform=transforms.Compose(
                    [transforms.ToPILImage(),
                    transforms.RandomVerticalFlip(p=1),
                    transforms.RandomHorizontalFlip(p=1),
                    transforms.Resize((256,256)),
                    transforms.ColorJitter(brightness=.5,
                                           contrast=.5,
                                           saturation=.5),
                    transforms.ToTensor(),
                    transforms.Normalize(mean,std)]),
                    embeddings = filename_embeddings)


train = ConcatDataset((data1,data2,data3,
                       data4,data5,data6,data7))
```

This code block is defining some instances of the *"CustomImageDataset"* class. The purpose of this class is to load images, **optionally** the textual information, and their associated labels, apply transformations, and return them as a tensor. These instances are initialized with the training set data ("*X_train*" and "*y_train*"), the normalization and **optionally** augmentation transformations, and the mean and standard deviation of the training set data calculated before using the "*get_mean_and_std()"* function. The "*filename_embeddings*" variable is also passed to the instances. The *optionality* part depends on the choice of the approach, and it is regulated by the two conditional statements on the flag variables *text* and *augm*.

The first conditional statement checks if the "*text*" variable is *0*: in this case, as before, the "*filename_embeddings*" variable (i.e. the dictionary maintaining textual information for each image) is set to "*None*" for the

same reasons explained before.

Then, there is another conditional statement that checks if the other flag variable "*augm*" is set to "*0*". If it is, in the current approach no augmentation techniques are applied on the training set of data and only one instance of the "*CustomImageDataset*" class is defined, with only normalization transformation applied to the images. Otherwise, seven instances of the "CustomImageDataset" class are defined, each with a different set of image augmentation transformations applied. These include random rotations, random flips, color jitter, and normalization, allowing for more varied training data. In this case, since seven instances have been defined, the overall training set would contain 805 samples (the original one consisted of 115 samples). Finally the code creates a concatenated dataset from the seven different datasets just composed.

4.4.2 Building the NN

At this point, there is the need to build the Neural Network to classify TV banner. Two scenarios will be considered: one where the model processes textual information in addition to images, and one where it only processes images. For the former scenario, an additional branch will be designed in the neural network to process text data. For the latter scenario, it will be built a convolutional neural network (CNN) that processes only image data. Here is the snippet:

```
class NeuralNetwork(nn.Module):
  def __init__(self):
    super(NeuralNetwork, self).__init__()
    self.image_features_ = nn.Sequential(
    # First 2D convolutional layer, taking in 3 input
   channel (image),outputting 16 convolutional features,
   with a square kernel size of 5
    nn.Conv2d(3, 16, kernel_size=5, stride=2, padding=2),
    nn.ReLU(inplace=True),
    nn.MaxPool2d(kernel_size=3, stride=2),
    nn.Dropout(p = 0.2),
    # Second 2D convolutional layer, taking in 16 input
   channel (image), outputting 128 convolutional features,
   with a square kernel size of 5
    nn.Conv2d(16, 128, kernel_size=5, padding=2),
```

```
12      nn.ReLU(inplace=True),
13      nn.MaxPool2d(kernel_size=3, stride=2),
14      nn.Dropout(p = 0.2),
15      # Third 2D convolutional layer, taking in 128 input
    channel (image), outputting 256 convolutional features,
    with a square kernel size of 5
16      nn.Conv2d(128, 256, kernel_size=5, padding=2),
17      nn.ReLU(inplace=True),
18      nn.MaxPool2d(kernel_size=3, stride=2),
19      nn.Dropout(p = 0.2),
20      # Fourth 2D convolutional layer, taking in 256 input
    channel (image), outputting 128 convolutional features,
    with a square kernel size of 5
21      nn.Conv2d(256, 128, kernel_size=5, padding=2),
22      nn.ReLU(inplace=True),
23      nn.MaxPool2d(kernel_size=3, stride=2),
24      # Fifth 2D convolutional layer, taking in 128 input
    channel (image), outputting 64 convolutional features,
    with a square kernel size of 5
25      nn.Conv2d(128, 64, kernel_size=5, padding=2),
26      nn.ReLU(inplace=True),
27      nn.MaxPool2d(kernel_size=2, stride=2),
28      )
29      if(text):
30          self.text_features = nn.Sequential(
31              nn.Linear(768, 64),
32              nn.ReLU(inplace=True),
33              nn.Dropout(p = 0.2),
34              nn.Linear(64, 64*3),
35              nn.ReLU(inplace=True),
36              nn.Dropout(p = 0.2),
37              nn.Linear(64*3, 64*3*3),
38              nn.ReLU(inplace=True),
39              )
40          self.combined_features_ = nn.Sequential(
41              nn.Linear(64*3*3*2, 64*3*3*2*2),
42              nn.ReLU(inplace=True),
43              nn.Dropout(p = 0.2),
44              nn.Linear(64*3*3*2*2, 64*3*3*2),
45              nn.ReLU(inplace=True),
46              nn.Linear(64*3*3*2, 64*4),
47              nn.ReLU(inplace=True),
```

```
            nn.Linear(64*4, 64),
            nn.ReLU(inplace=True),
            nn.Linear(64, 2),
            nn.Sigmoid()
            )

    else:
        self.combined_features_ = nn.Sequential(
            nn.Linear(64*3*3, 64*4),
            nn.ReLU(inplace=True),
            nn.Linear(64*4, 64),
            nn.ReLU(inplace=True),
            nn.Linear(64, 2),
            nn.Sigmoid()
        )

  if(text):
    def forward(self, x,y):
        x = self.image_features_(x)
        x = x.view(-1, 64*3*3)
        y=self.text_features(y)
        z=torch.cat((x,y),1)
        z=self.combined_features_(z)
        return z
  else:
    def forward(self, x):
      x = self.image_features_(x)
      x = x.view(-1, 64*3*3)
      z= x
      z=self.combined_features_(z)
      return z
```

The *NeuralNetwork* class is defined as a subclass of *torch.nn.Module*. This is the standard way of defining a neural network in PyTorch, as it allows us to use all the functionality of the *nn.Module* class. In the *___init___* method, the neural network architecture is defined. There are two possible architectures depending on whether the *text* argument is *True* or *False*, indicating if the current approach is taking into account the textual information or not.

First the image processing branch is defined in the *image_features_* at-

tribute. It is the same for both scenarios and uses the "*nn.Sequential*" container to group the convolutional layers together. Each convolutional layer has a 2D convolution operation that takes in an input channel, output channel, kernel size, stride, and padding. The "*nn.ReLU(inplace=True)*" layer applies *ReLU* activation function element-wise to the output tensor. The "*nn.MaxPool2d*" layer performs a max pooling operation over a square window of the input tensor. Finally, the "*nn.Dropout*" layer randomly sets input elements to zero during training with a probability of p, which helps to prevent overfitting.

If *text* is True:

- the neural network will have an additional branch to process text data. This branch is defined in the *text_features* attribute and also uses the "*nn.Sequential*" container. It contains three fully connected layers, the first one takes in a vector of 768 (*BERT* embedding size) as input and output a vector of size 64. This is followed by a few more fully connected layers (also known as linear layers) with *ReLU* activation and dropout layers to prevent overfitting;
- the image and text branches are combined in the *combined_features_* attribute, also defined using the "*nn.Sequential*" container. It takes the concatenated output from the *image_features_* and *text_features* modules as input. The fully connected layers perform a series of matrix multiplications, followed by the application of a *ReLU* activation function to the output tensor. Finally, the output tensor is passed through a "*nn.Sigmoid*" layer to produce a probability distribution over the output classes.

If *text* is *False*, the *combined_features_* attribute is defined without the *text_features* branch. It just consists of three fully connected layers with *ReLU* activation and a *nn.Sigmoid* final layer.

The *forward* method is different depending on *text* variable:

- if it is *True*, the input data x and y are processed through the image and text branches, respectively. The x tensor is flattened and concatenated with the y tensor along the channel dimension using the *torch.cat* function. The resulting tensor is processed through the *com-*

*bined_features_* branch, which outputs a tensor of shape (*batch_size*, 2). The output tensor contains the probabilities for each input example to belong to each of the two classes;

- if *text* is *False*, the first two lines of the method are the same. Then the resulting tensor z is simply the flattened image tensor x and the last line applies the *combined_features_* module to the resulting tensor z, like before.

4.4.3 Hyperparameters tuning

In order to achieve the best performance, different hyperparameters will be taken into account, thanks to the function *tune.run*: as specified at section 3.6, this function takes in input different parameters. Here the preliminar snippet:

```
    import tensorflow as tf

config = {
    "lr": tune.grid_search([0.00001, 1e-4, 1e-3, 1e-2]),
    "batch_size": tune.grid_search([32, 64]),
    "optimizer": tune.choice(["Adam", "SGD w/ mom"]),
    "epochs": tune.grid_search([20, 30])
}

num_samples=1
device = tf.test.gpu_device_name()
gpus_per_trial = 1

if(not text):
  filename_embeddings = None
```

This code snippet imports the *TensorFlow* library and sets up a configuration dictionary for hyperparameter tuning using the *Ray Tune* library. The configuration dictionary contains four hyperparameters to be tuned:

- **lr (learning rate)** with four possible values in a grid search: 0.00001, $1e-4$, $1e-3$, $1e-2$;
- **batch_size** with two possible values in a grid search: *32, 64*;
- **optimizer** with two possible strings to choose from: *Adam* or *SGD*

with momentum. They are just string, then in the training function the choice will be handler and the appropriate optimizer will be defined;

- **epochs** with two possible values in a grid search: *20, 30*;

The variable *num_samples* is set to *1*, meaning that only one combination of hyperparameters will be sampled and tested.

The code then checks if the GPU is available on the device being used and sets *gpus_per_trial* to *1*.

As usual if the *text* variable is False, *filename_embeddings* is set to None: this operation is required since this dictionary, containing the textual information for each image in the overall dataset, will be passed as parameter in the *tune.run* function, in order to create the validation dataset. Doing this, if *text* is *None*, no textual information will be appended to the validation set, otherwise textual information will be included.

Now it is needed to define the *CLIReporter* object and finally define the hyperparameters tuning function:

```
reporter = CLIReporter(
      metric_columns=["train_loss", "train_acc",
                      "valid_loss", "valid_accuracy",
                      "training_iteration"])

result = tune.run(
      partial(train_dataset, train_data = train,
                valid_data_features = X_val,
                valid_data_target = y_val,
                mean = mean, std = std,
                device = device,
                embeddings = filename_embeddings,
                text = text,
                augm = augm),
      resources_per_trial={"cpu": 2, "gpu": gpus_per_trial},
      config=config,
      num_samples=num_samples,
      scheduler=scheduler,
      progress_reporter=reporter,
      local_dir = '/mydirectory/results'
)
```

The *CLIReporter* class is used to display the progress of the hyperparameter search, reporting the values of different metrics in the resulting table, such as *train loss*, *train accuracy*, *valid loss*, *valid accuracy*, and the current *training iteration*.

Finally, the *tune.run()* function is called. It has different parameters:

- **train_dataset**: it is the most important parameter, since it is the function responsible for training and validate the model;
- **train_data** = *train*: the training set defined at section 4.4.1;
- **valid_data_features** = *X_val*: the features dataframe of the validation set that will be created in the training function;
- **valid_data_target** = *y_val*: the label dataframe of the validation set;
- **mean** = mean, **std** = std: mean and standard deviation values used for the normalization of the validation set after its definitio;
- **device** = device: the device being used;
- **embeddings** = filename_embeddings: the dictionary containing textual information for each image or *None*, depending on the value of the flag variable *text*;
- **text** = text, **augm** = augm: the flag variables useful for choosing the approach to train the model and classify the TV banners.

So the first one is the *train_dataset* function. It is time to define it: different snippets will be listed for a better readability.

```
def train_dataset(config, train_data = None,
    valid_data_features = None, valid_data_target = None,
    mean = None, std = None, device = None,
    embeddings = None, text = None, augm = None,
    checkpoint_dir=None):

    net = NeuralNetwork()
    if torch.cuda.is_available():
        device = "cuda:0"
        if torch.cuda.device_count() > 1:
            net = nn.DataParallel(net)
```

```
    net.to(device)
```

The function takes in several inputs, including *config, train_data, valid_data_features, valid_data_target, mean, std, device, embeddings, text, augm, and checkpoint_dir.* As specified a few lines above and as it will be possible to see in the next snippets, these inputs are used to set up the data loaders (both for training and validation loops), the neural network model, and the optimizer, as well as to define various hyperparameters such as the learning rate, batch size, and number of epochs.

NeuralNetwork() is the custom PyTorch model that is being used to define the neural network architecture. Then the function checks if a GPU is available and if so, sets the device to "*cuda:0*". If multiple GPUs are available, it uses *nn.DataParallel()* to parallelize the model across multiple devices.

```
    criterion = nn.BCELoss()

    if optimizer == "Adam":
      optimizer = optim.Adam(net.parameters(),
                             lr = config["lr"])
    else:
      optimizer = optim.SGD(net.parameters(),
                             lr=config["lr"],
                             momentum=0.9)


    train_data, val_data = load_data(train_data,
                                     valid_data_features,
                                     valid_data_target,
                                     mean,
                                     std,
                                     embeddings)

    num_workers = 2
    trainloader = torch.utils.data.DataLoader(
        train_data,
        batch_size=int(config["batch_size"]),
        shuffle=True,
        num_workers=num_workers)

```

```
26      valloader = torch.utils.data.DataLoader(
27          val_data,
28          batch_size=int(config["batch_size"]),
29          shuffle=True,
30          num_workers=num_workers)
```

The loss function at line 1 is defined using *nn.BCELoss()*: this creates a criterion that measures the *Binary Cross Entropy* between the target and the input probabilities.

The optimizer at line 3 is defined based on the value of the string variable *optimizer*, that belongs to the *config* dictionary defined above: if that variable is set to "*Adam*", the Adam optimizer is used. Otherwise, the *Stochastic Gradient Descent* (**SGD**) optimizer with *momentum* is used (*momentum* is set to *0.9*, a commonly used value for the momentum parameter with the SGD optimizer). For both the optimizers the learning rate is not fixed: it will be different for each trial run with *Ray Tune*, and will be chosen from the corresponding entry in the *config* dictionary.

The *load_data function* is called at line 12 to set up the training and validation data loaders created at lines 20 and 26. This function is very simple and just defines the validation dataset: for this reason it will not be defined here in order to not break the thread. It will be defined at snippet 4.4.3, now the *train_dataset* function will continue:

```
1       # if num_epochs:
2       num_epochs = int(config["epochs"])
3
4       # loop over the dataset multiple times
5       for epoch in range(1, num_epochs+1):
6           running_loss = 0.0
7           total_training_steps = 0
8           correct_training = 0
9           epoch_steps = 0
10          net.train()
11          for i, data in enumerate(trainloader, 0):
12              # data is a list of [inputs, labels]
13              if(text):
14                imgs, text_feature, label = data
15                text_feature = text_feature.to(device)
16              else:
```

```
            imgs, label = data

        imgs = imgs.to(device)
        label = label.to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward pass:  make predictions using our
        # model and calculate loss based on those
        # predictions and our actual labels
        # forward + backward + optimize
        if(text):
            outputs = net(imgs, text_feature)
        else:
            outputs = net(imgs)

        _, predicted = torch.max(outputs.data, 1)


        loss = criterion(outputs, label)
        # calculate the new gradients using the
        # loss.backward() function
        loss.backward()
        # we update the weights with the
        # optimizer.step() function
        optimizer.step()

        running_loss += loss.item()
        epoch_steps += 1

        total_training_steps += label.size(0)
        correct_training+=
                    (predicted==label).sum().item()

        # print every 2000 mini-batches
        if i % 2000 == 1999:
            print("[%d, %5d] loss: %.3f" %
                    (epoch + 1, i + 1,
                    running_loss / epoch_steps))

            running_loss = 0.0
```

The training loop runs over the dataset for a certain number of epochs (*num_epochs*): this variable also is not fixed and will change at each trial run with *Ray Tune*, being chosen from the corresponding entry in the *config* dictionary. For each epoch, the *running_loss, total_training_steps, correct_training*, and *epoch_steps* are initialized to zero: these values will be useful to calculate the performances of the model and report them. At line 10, the network is put in training mode using the *net.train()* function.

The training data is then loaded in batches using a PyTorch DataLoader object called *trainloader*, previosly defined. For each batch:

- if the *text* flag is set to True, then the data consists of images, the corresponding text features and the labels;
- otherwise, it only contains images and labels.

At line 19 and 20, *imgs* and *label* tensors are allocated on the device.

At line 23 the optimizer's gradients are set to zero using the *optimizer.zero_grad()* function.

The forward pass is done using the neural network model *net* and the input data at line 30 or 32:

- if the *text* flag is set to *True*, then both the image and text features are passed to the model;
- otherwise, only the images are used.

The predicted labels are computed from the model outputs using the *torch.max()* function: this function in fact returns a *namedtuple (value, indices)*, where *values* mantain the probability values for both classes of each row of the *outputs.data* tensor. With *_, predicted*, in fact, it is possible to get the index of the maximum value, i.e. it is possible to get the class (*0* or *1*). Then at line 37 the loss is calculated based on *outputs* and the actual *labels* tensors.

The gradients are computed using the *loss.backward()* function at line 40, and the weights are updated using the *optimizer.step()* function at line 43.

Finally, from lines 45 to 49, The *running_loss* and *epoch_steps* variables

are updated, and the accuracy of the training predictions is computed.

Now there is the validation loop:

```
        val_loss = 0.0
        val_steps = 0
        total_val_steps = 0
        correct_val = 0
        net.eval()
        for i, data  in enumerate(valloader, 0):
            with torch.no_grad():
              if(text):
                imgs, text_feature, label = data
                text_feature=text_feature.to(device)
              else:
                imgs, label = data

              imgs=imgs.to(device)
              label = label.to(device)

              if(text):
                outputs = net(imgs,text_feature)
              else:
                outputs = net(imgs)

              _, predicted = torch.max(outputs.data, 1)

              loss = criterion(outputs, label)

              total_val_steps += label.size(0)
              correct_val+=(predicted == label).sum().item()
              val_loss += loss.cpu().numpy()
              val_steps += 1
              val_acc = correct_val/total_val_steps

        tune.report(valid_loss = (val_loss / val_steps),
                    valid_accuracy = val_acc,
                    train_loss = (running_loss/epoch_steps),
                    train_acc=correct_training/
                                    total_training_steps)


        with tune.checkpoint_dir(epoch) as checkpoint_dir:
```

```
40                path = os.path.join(checkpoint_dir,"checkpoint")
41                torch.save((net.state_dict(),
42                            optimizer.state_dict()), path)
```

The *val_loss*, *val_steps, total_val_steps*, and *correct_val* variables are initialized to zero: also these variable will be useful to evaluate the model performances. The net model is set to evaluation mode using *net.eval()* at line 6.

Then, a loop over the validation set is started with enumerate*(valloader, 0)*. As before, for each batch:

- if the *text* variable is *True*, then each batch of validation data includes both images and text features;
- otherwise, it only includes images

The data is then moved to the GPU, if available, using *imgs.to(device)* and *label.to(device)*.

The forward pass is then performed using the net model with *outputs = net(imgs, text_feature)* or *outputs = net(imgs)*, depending on whether *text* is *True*.

As in the training loop, the *predicted* class is obtained from the model's output tensor using *_, predicted = torch.max(outputs.data, 1)*.

The loss is calculated using the criterion function, at line 24.

The number of correctly classified samples is counted using *(predicted == label).sum().item()* and added to the *correct_val* variable. The total number of samples is added to the *total* variable, while the validation loss is accumulated in *val_loss* and the number of validation steps in *val_steps*. The *val_acc* variable is then computed as the ratio of correctly classified samples (in the validation stage) to the total number of samples.

At line 32 the *tune.report()* function is called with the validation loss, and validation accuracy, as well as the training loss and training accuracy values: these values will be reported in the table that will be outputted at every epoch.

Finally, the current state of the network and optimizer are saved in a checkpoint file using *torch.save()*, with the file name including the current *epoch* number.

Overall, this function is used in order to train the network and evaluate its performances on the validation set: the values that matter most, i.e. the loss and accuracy values boh on training and validation set, are calculated for each epoch and finally reported.

The *load_data() function*

The following function, instead, is *load_data*: it has been invoked at the beginning of the training function, but it has been properly defined here in order to not break the thread before, during the explanation of the more complex training function.

```
def load_data(training_data, valid_data_features,
                valid_data_target, mean, std, embeddings):

    val_data = CustomImageDataset(
                features = valid_data_features,
                target = valid_data_target,
                transform=transforms.Compose([
                            transforms.ToPILImage(),
                            transforms.Resize((256, 256)),
                            transforms.ToTensor(),
                            transforms.Normalize(mean, std)
                            ]),
                            embeddings = embeddings)

    return training_data, val_data
```

4.4.4 First results

The baseline approach to train our model and classify the TV banners is **NOT_NOA**: no textual information have been involved, and no augmentation techniques have been applied on the training set of data. The performance have been measured through the loss and accuracy values both on training and on validation set and here are listed the summary tables. Each one has four columns: *valid loss, valid acc, train loss* and *train acc*,

respectively the loss and accuracy values on the validation set and the loss and accuracy values on the training one.

valid_los	valid_acc	train_los	train_acc
0.68485	0.76471	0.69023	0.76768
0.68906	0.76471	0.68696	0.76768
0.67791	0.76471	0.68079	0.76768
0.67174	0.76471	0.67557	0.76768
0.6644	0.76471	0.6748	0.76768
0.65697	0.76471	0.67745	0.76768
0.64563	0.76471	0.6474	0.76768
0.62715	0.76471	0.64671	0.76768
0.58454	0.76471	0.63265	0.76768
0.49979	0.76471	0.54424	0.76768
0.641	0.76471	0.57747	0.76768
0.6721	0.76471	0.494	0.76768
0.43913	0.76471	0.49301	0.76768
0.43864	0.76471	0.64392	0.76768
0.44256	0.76471	0.56847	0.76768
0.43841	0.76471	0.49295	0.76768
0.67522	0.76471	0.49295	0.76768
0.43838	0.76471	0.56846	0.76768
0.43838	0.76471	0.64399	0.76768
0.43836	0.76471	0.56847	0.76768

Figure 4.17: batch32,epochs20,lr0.01,SGD

valid_los	valid_acc	train_los	train_acc
0.6899	0.7647	0.6916	0.7677
0.6919	0.7647	0.6901	0.7677
0.6919	0.7647	0.6900	0.7677
0.6897	0.7647	0.6899	0.7677
0.6918	0.7647	0.6900	0.7677
0.6918	0.7647	0.6906	0.7677
0.6896	0.7647	0.6906	0.7677
0.6894	0.7647	0.6904	0.7677
0.6894	0.7647	0.6897	0.7677
0.6917	0.7647	0.6893	0.7677
0.6892	0.7647	0.6912	0.7677
0.6891	0.7647	0.6893	0.7677
0.6890	0.7647	0.6893	0.7677
0.6916	0.7647	0.6890	0.7677
0.6889	0.7647	0.6900	0.7677
0.6915	0.7647	0.6900	0.7677
0.6915	0.7647	0.6909	0.7677
0.6886	0.7647	0.6899	0.7677
0.6885	0.7647	0.6898	0.7677
0.6887	0.7647	0.6886	0.7677

Figure 4.18: batch32,epochs20,lr0.0001,SGD

valid_los	valid_acc	train_los	train_acc
0.6935	0.2353	0.6941	0.2323
0.6936	0.2353	0.6941	0.2323
0.6940	0.2353	0.6941	0.2323
0.6935	0.2353	0.6941	0.2323
0.6940	0.2353	0.6941	0.2323
0.6935	0.2353	0.6941	0.2323
0.6939	0.2353	0.6938	0.2323
0.6940	0.2353	0.6941	0.2323
0.6941	0.2353	0.6941	0.2323
0.6940	0.2353	0.6939	0.2323
0.6940	0.2353	0.6938	0.2323
0.6935	0.2353	0.6939	0.2323
0.6939	0.2353	0.6939	0.2323
0.6935	0.2353	0.6937	0.2323
0.6934	0.2353	0.6939	0.2323
0.6930	0.2353	0.6938	0.2323
0.6939	0.2353	0.6938	0.2323
0.6939	0.2353	0.6938	0.2323
0.6931	0.2353	0.6939	0.2323
0.6930	0.2353	0.6940	0.2323

Figure 4.19: batch32,epochs20,lr0.00001,SGD

valid_los	valid_acc	train_los	train_acc
0.68219	0.76471	0.68184	0.76768
0.68211	0.76471	0.68255	0.76768
0.68199	0.76471	0.68195	0.76768
0.68184	0.76471	0.68149	0.76768
0.68167	0.76471	0.68266	0.76768
0.68149	0.76471	0.68178	0.76768
0.68129	0.76471	0.68096	0.76768
0.68108	0.76471	0.68125	0.76768
0.68086	0.76471	0.68034	0.76768
0.68063	0.76471	0.6812	0.76768
0.68039	0.76471	0.67933	0.76768
0.68015	0.76471	0.6803	0.76768
0.67991	0.76471	0.68029	0.76768
0.67966	0.76471	0.67845	0.76768
0.67941	0.76471	0.67887	0.76768
0.67916	0.76471	0.67811	0.76768
0.6789	0.76471	0.67816	0.76768
0.67864	0.76471	0.67829	0.76768
0.67838	0.76471	0.67775	0.76768
0.67812	0.76471	0.67573	0.76768

Figure 4.20: batch=64,epochs20,lr0.001,SGD

valid_los	valid_acc	train_los	train_acc
0.6895	0.7647	0.6894	0.7677
0.6895	0.7647	0.6896	0.7677
0.6895	0.7647	0.6897	0.7677
0.6895	0.7647	0.6894	0.7677
0.6895	0.7647	0.6897	0.7677
0.6895	0.7647	0.6898	0.7677
0.6894	0.7647	0.6896	0.7677
0.6894	0.7647	0.6893	0.7677
0.6894	0.7647	0.6897	0.7677
0.6894	0.7647	0.6895	0.7677
0.6894	0.7647	0.6893	0.7677
0.6893	0.7647	0.6893	0.7677
0.6893	0.7647	0.6893	0.7677
0.6893	0.7647	0.6895	0.7677
0.6893	0.7647	0.6892	0.7677
0.6893	0.7647	0.6895	0.7677
0.6892	0.7647	0.6890	0.7677
0.6892	0.7647	0.6891	0.7677
0.6892	0.7647	0.6891	0.7677
0.6892	0.7647	0.6887	0.7677

Figure 4.21: batch64,epochs20,lr0.0001,SGD

valid_los	valid_acc	train_los	train_acc
0.6936	0.2353	0.6941	0.2323
0.6936	0.2353	0.6941	0.2323
0.6936	0.2353	0.6942	0.2323
0.6936	0.2353	0.6941	0.2525
0.6936	0.2353	0.6941	0.2222
0.6936	0.2353	0.6941	0.2323
0.6936	0.2353	0.6942	0.2222
0.6936	0.2353	0.6942	0.2323
0.6936	0.2353	0.6941	0.2323
0.6935	0.2353	0.6941	0.2424
0.6935	0.2353	0.6941	0.2525
0.6935	0.2353	0.6940	0.2323
0.6935	0.2353	0.6943	0.2323
0.6935	0.2353	0.6941	0.2424
0.6935	0.2353	0.6942	0.2222
0.6935	0.2353	0.6942	0.2323
0.6935	0.2353	0.6941	0.2222
0.6935	0.2353	0.6940	0.2323
0.6935	0.2353	0.6941	0.2424
0.6935	0.2353	0.6941	0.2424

Figure 4.22: batch64,epochs20,lr0.00001,SGD

valid_lo	valid_ac	train_lo	train_ac
0.6976	0.2353	0.6947	0.2323
0.6953	0.2353	0.6941	0.2828
0.6899	0.7647	0.6902	0.7576
0.6832	0.7647	0.688	0.7677
0.6749	0.7647	0.6742	0.7677
0.6798	0.7647	0.6701	0.7677
0.6374	0.7647	0.6513	0.7677
0.5715	0.7647	0.6545	0.7677
0.5126	0.7647	0.5855	0.7677
0.6291	0.7647	0.4989	0.7677
0.4533	0.7647	0.5645	0.7677
0.6528	0.7647	0.6376	0.7677
0.6323	0.7647	0.4937	0.7677
0.4383	0.7647	0.5669	0.7677
0.6537	0.7647	0.4912	0.7677
0.4442	0.7647	0.4911	0.7677
0.4411	0.7647	0.4908	0.7677
0.4383	0.7647	0.5561	0.7677
0.4513	0.7647	0.4933	0.7677
0.4433	0.7647	0.5525	0.7677
0.4773	0.7647	0.5623	0.7677
0.6531	0.7647	0.5618	0.7677
0.6444	0.7647	0.4955	0.7677
0.4477	0.7647	0.6419	0.7677
0.4692	0.7647	0.5446	0.7677
0.4805	0.7647	0.6242	0.7677
0.629	0.7647	0.4948	0.7677
0.4482	0.7647	0.4933	0.7677
0.4929	0.7647	0.5437	0.7677
0.4415	0.7647	0.4852	0.7677

Figure 4.23: batch32,epochs30,lr0.01,SGD

valid_lo	valid_ac	train_lo	train_ac
0.6996	0.2353	0.6974	0.2323
0.6956	0.2353	0.6961	0.2323
0.6996	0.2353	0.6973	0.2323
0.6994	0.2353	0.6985	0.2323
0.6994	0.2353	0.6985	0.2323
0.6955	0.2353	0.6983	0.2323
0.6992	0.2353	0.6984	0.2323
0.6992	0.2353	0.6982	0.2323
0.6991	0.2353	0.6970	0.2323
0.6990	0.2353	0.6980	0.2323
0.6953	0.2353	0.6957	0.2323
0.6989	0.2353	0.6956	0.2323
0.6988	0.2353	0.6979	0.2323
0.6953	0.2353	0.6967	0.2323
0.6952	0.2353	0.6978	0.2323
0.6986	0.2353	0.6977	0.2323
0.6952	0.2353	0.6965	0.2323
0.6985	0.2353	0.6964	0.2323
0.6983	0.2353	0.6974	0.2323
0.6951	0.2353	0.6973	0.2323
0.6982	0.2353	0.6973	0.2323
0.6980	0.2353	0.6972	0.2323
0.6980	0.2353	0.6970	0.2323
0.6980	0.2353	0.6970	0.2323
0.6979	0.2353	0.6969	0.2323
0.6949	0.2353	0.6968	0.2323
0.6977	0.2353	0.6966	0.2323
0.6920	0.2353	0.6965	0.2323
0.6948	0.2353	0.6965	0.2323
0.6947	0.2353	0.6955	0.2323

Figure 4.24: batch32,epochs30,lr0.0001,SGD

valid_lo	valid_ac	train_los	train_ac
0.6967	0.2353	0.6975	0.2323
0.6959	0.2353	0.6964	0.2323
0.6948	0.2353	0.6955	0.2323
0.6934	0.2647	0.6938	0.2727
0.6918	0.7647	0.692	0.7273
0.6901	0.7647	0.6897	0.7677
0.6882	0.7647	0.6876	0.7677
0.6862	0.7647	0.6853	0.7677
0.6841	0.7647	0.6834	0.7677
0.6819	0.7647	0.6801	0.7677
0.6795	0.7647	0.6766	0.7677
0.6768	0.7647	0.6733	0.7677
0.6737	0.7647	0.6675	0.7677
0.6698	0.7647	0.6626	0.7677
0.6647	0.7647	0.6542	0.7677
0.6575	0.7647	0.6507	0.7677
0.647	0.7647	0.6375	0.7677
0.6307	0.7647	0.6135	0.7677
0.6057	0.7647	0.5831	0.7677
0.5752	0.7647	0.5818	0.7677
0.5542	0.7647	0.5445	0.7677
0.5457	0.7647	0.5512	0.7677
0.5441	0.7647	0.5647	0.7677
0.5445	0.7647	0.5337	0.7677
0.5451	0.7647	0.5342	0.7677
0.5456	0.7647	0.5354	0.7677
0.5459	0.7647	0.5485	0.7677
0.5461	0.7647	0.568	0.7677
0.5462	0.7647	0.5289	0.7677
0.5462	0.7647	0.53	0.7677

Figure 4.25:
batch64,epochs30,lr0.01,SGD

valid_lo	valid_ac	train_lo	train_ac
0.6952	0.2353	0.6959	0.2323
0.6952	0.2353	0.6959	0.2323
0.6951	0.2353	0.6957	0.2323
0.6949	0.2353	0.6956	0.2323
0.6948	0.2353	0.6954	0.2323
0.6946	0.2353	0.6951	0.2323
0.6944	0.2353	0.695	0.2323
0.6943	0.2353	0.6948	0.2323
0.6941	0.2353	0.6945	0.2323
0.6939	0.2353	0.6942	0.2323
0.6937	0.2353	0.694	0.2323
0.6935	0.2353	0.6938	0.2323
0.6932	0.2353	0.6935	0.2323
0.693	0.7647	0.6933	0.3636
0.6928	0.7647	0.693	0.6566
0.6926	0.7647	0.6928	0.7576
0.6924	0.7647	0.6926	0.7677
0.6922	0.7647	0.6923	0.7677
0.692	0.7647	0.692	0.7677
0.6918	0.7647	0.6918	0.7677
0.6915	0.7647	0.6916	0.7677
0.6913	0.7647	0.6913	0.7677
0.6911	0.7647	0.6911	0.7677
0.6909	0.7647	0.6909	0.7677
0.6907	0.7647	0.6906	0.7677
0.6905	0.7647	0.6904	0.7677
0.6903	0.7647	0.6899	0.7677
0.6901	0.7647	0.6901	0.7677
0.6898	0.7647	0.69	0.7677
0.6896	0.7647	0.6895	0.7677

Figure 4.26:
batch64,epochs30,lr0.001,SGD

valid_lo	valid_ac	train_los	train_ac
0.6906	0.7647	0.6907	0.7677
0.6906	0.7647	0.6907	0.7677
0.6906	0.7647	0.6910	0.7677
0.6906	0.7647	0.6908	0.7677
0.6906	0.7647	0.6907	0.7677
0.6906	0.7647	0.6906	0.7677
0.6906	0.7647	0.6907	0.7677
0.6906	0.7647	0.6911	0.7677
0.6906	0.7647	0.6907	0.7677
0.6906	0.7647	0.6907	0.7677
0.6906	0.7647	0.6909	0.7677
0.6906	0.7647	0.6908	0.7677
0.6906	0.7647	0.6907	0.7677
0.6906	0.7647	0.6907	0.7677
0.6906	0.7647	0.6906	0.7677
0.6906	0.7647	0.6906	0.7677
0.6906	0.7647	0.6909	0.7677
0.6906	0.7647	0.6909	0.7677
0.6906	0.7647	0.6907	0.7677
0.6906	0.7647	0.6906	0.7677
0.6906	0.7647	0.6906	0.7677
0.6906	0.7647	0.6907	0.7677
0.6906	0.7647	0.6906	0.7677
0.6906	0.7647	0.6908	0.7677
0.6906	0.7647	0.6907	0.7677
0.6906	0.7647	0.6910	0.7677
0.6906	0.7647	0.6908	0.7677
0.6906	0.7647	0.6908	0.7677
0.6905	0.7647	0.6907	0.7677
0.6905	0.7647	0.6907	0.7677

Figure 4.27:
batch64,epochs30,lr0.00001,SGD

valid_los	valid_ac	train_los	train_acc
0.4383	0.7647	0.7204	0.5455
0.4383	0.7647	0.4929	0.7677
0.907	0.7647	0.5685	0.7677
0.4383	0.7647	0.5685	0.7677
0.6726	0.7647	0.5685	0.7677
0.6726	0.7647	0.5685	0.7677
0.6726	0.7647	0.5685	0.7677
0.4383	0.7647	0.5685	0.7677
0.4383	0.7647	0.5685	0.7677
0.4383	0.7647	0.644	0.7677
0.4383	0.7647	0.4929	0.7677
0.4383	0.7647	0.644	0.7677
0.4383	0.7647	0.5685	0.7677
0.4383	0.7647	0.4929	0.7677
0.6726	0.7647	0.5685	0.7677
0.4383	0.7647	0.5685	0.7677
0.4383	0.7647	0.5685	0.7677
0.4383	0.7647	0.4929	0.7677
0.6726	0.7647	0.4929	0.7677
0.4383	0.7647	0.4929	0.7677

Figure 4.28:
batch32,epochs20,lr0.001,Adam

valid_lo	valid_ac	train_lo	train_ac
0.4383	0.7647	0.5481	0.6162
0.4383	0.7647	0.4929	0.7677
0.4383	0.7647	0.644	0.7677
0.4383	0.7647	0.4929	0.7677
0.4383	0.7647	0.4929	0.7677
0.4383	0.7647	0.5685	0.7677
0.907	0.7647	0.644	0.7677
0.4383	0.7647	0.5685	0.7677
0.6726	0.7647	0.5685	0.7677
0.6726	0.7647	0.4929	0.7677
0.6726	0.7647	0.4929	0.7677
0.4383	0.7647	0.4929	0.7677
0.6726	0.7647	0.5685	0.7677
0.4383	0.7647	0.4929	0.7677
0.4383	0.7647	0.4929	0.7677
0.6726	0.7647	0.4929	0.7677
0.4383	0.7647	0.4929	0.7677
0.4383	0.7647	0.5685	0.7677
0.4383	0.7647	0.5685	0.7677
0.6726	0.7647	0.4929	0.7677
0.6726	0.7647	0.4929	0.7677
0.4383	0.7647	0.5685	0.7677
0.4383	0.7647	0.5685	0.7677
0.4383	0.7647	0.644	0.7677
0.4383	0.7647	0.5685	0.7677
0.6726	0.7647	0.4929	0.7677
0.4383	0.7647	0.4929	0.7677
0.6726	0.7647	0.5685	0.7677
0.4383	0.7647	0.5685	0.7677
0.6726	0.7647	0.5685	0.7677

Figure 4.29:
batch32,epochs30,lr0.001,Adam

valid_lo	valid_ac	train_los	train_ac
0.6924	0.2353	0.6951	0.2323
0.6926	0.2647	0.6932	0.4646
0.6932	0.4412	0.6902	0.7677
0.6921	0.7353	0.6868	0.7677
0.6890	0.7647	0.6851	0.7677
0.6904	0.7647	0.6790	0.7677
0.6821	0.7647	0.6771	0.7677
0.6771	0.7647	0.6646	0.7677
0.6837	0.7647	0.6652	0.7677
0.6807	0.7647	0.6553	0.7677
0.6790	0.7647	0.6645	0.7677
0.6701	0.7647	0.6568	0.7677
0.6673	0.7647	0.6229	0.7677
0.6077	0.7647	0.5738	0.7677
0.5823	0.7647	0.5513	0.7677
0.7357	0.7647	0.6332	0.7677
0.5235	0.7647	0.5803	0.7677
0.6192	0.7647	0.5764	0.7677
0.5101	0.7647	0.5058	0.7677
0.5016	0.7647	0.6368	0.7677
0.5051	0.7647	0.6316	0.7677
0.4998	0.7647	0.5676	0.7677
0.6662	0.7647	0.5688	0.7677
0.6568	0.7647	0.5688	0.7677
0.4889	0.7647	0.5682	0.7677
0.8135	0.7647	0.5707	0.7677
0.6477	0.7647	0.5682	0.7677
0.6507	0.7647	0.5666	0.7677
0.6379	0.7647	0.4991	0.7677
0.6307	0.7647	0.6369	0.7677

Figure 4.30:
batch32,epochs30,lr0.00001,Adam

valid_los	valid_ac	train_los	train_acc
0.5486	0.7647	0.6261	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5447	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5577	0.7677
0.5486	0.7647	0.5642	0.7677
0.5486	0.7647	0.5318	0.7677
0.5486	0.7647	0.5577	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5577	0.7677
0.5486	0.7647	0.5642	0.7677
0.5486	0.7647	0.5771	0.7677
0.5486	0.7647	0.5253	0.7677
0.5486	0.7647	0.5577	0.7677
0.5486	0.7647	0.5318	0.7677

Figure 4.31: batch64,epochs20,lr0.01,Adam

valid_lo	valid_ac	train_lo	train_ac
0.6788	0.7647	0.6809	0.7677
0.6681	0.7647	0.6676	0.7677
0.6427	0.7647	0.6406	0.7677
0.5976	0.7647	0.6063	0.7677
0.5589	0.7647	0.5602	0.7677
0.5497	0.7647	0.5589	0.7677
0.549	0.7647	0.5447	0.7677
0.5488	0.7647	0.5383	0.7677
0.5487	0.7647	0.5642	0.7677
0.5486	0.7647	0.5124	0.7677
0.5486	0.7647	0.5447	0.7677
0.5486	0.7647	0.5447	0.7677
0.5486	0.7647	0.5642	0.7677
0.5486	0.7647	0.5512	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5253	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5512	0.7677
0.5486	0.7647	0.5706	0.7677
0.5486	0.7647	0.5642	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5318	0.7677
0.5486	0.7647	0.5512	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5318	0.7677
0.5486	0.7647	0.5447	0.7677
0.5486	0.7647	0.5706	0.7677

Figure 4.32: batch64,epochs30,lr0.0001,Adam

4.4.5 Discussions

Based on these values, it seems that the model is not performing well, as both the validation loss and accuracy remain constant throughout the epochs, indicating that the model is not learning any new information. Additionally, the training loss and accuracy are also constant, indicating that the model is not fitting to the training data well. Particularly bad behaviours can be observed at figures 4.19, 4.22 and 4.24: those trials are characterized by a learning too low, that affects the training of the model, whose greater value achieved on the training set is *0.2323*. This is a very bad performance.

However, also in the other trials of the current approach, the values are always constant: this happens both with 20 or 30 epochs, with both optimizers and with both batch sizes.

Since the performance are very bad, no others approaches (**NOT_A, T_NOA, T_A**) will be taken into account at the moment. To improve the performance of the model, a different hyperparameters configuration will be set.

4.5 Second run

For this run different settings will be applied in order to train the model: the most important difference is that in this case the learning rate is not configurable with different parameters as before, but it is fixed at the beginning (it starts from *0.001*) and will be **updated dinamically** (up to a threshold equal to *0.00001*) by the schedulers provided by *torch.optim*. This is *lr_scheduler.ReduceLROnPlateau*, that as specified in the previous sections can reduce the learning rate by a factor specified in input. In this way the performance may really improve, since the updates will be dynamic: a quantity will be monitored, and if there will not be improvements after some epochs, it probably means that the learning rate is not appropriate. And for this reason, it will be updated, decaying it by a factor, up to a given threshold.

4.5.1 Modification to hyperparameters tuning

```
    config = {
    "patience_onplat": tune.grid_search([4,5,6,7]),
    "batch_size": tune.choice([32, 64]),
    "epochs": tune.choice([30])
}
```

This code snippet sets up a configuration dictionary for hyperparameter tuning using the *Ray Tune* library. The configuration dictionary contains three hyperparameters to be tuned:

- **patience_onplat** with four possible values: this is the number of epochs with no improvements after which the learning rate will be up-

dated thanks to the *ReduceLROnPlateau* scheduler. These values have been chosen considering the learning rate threshold and the number of epochs, in order to let the learning make all the updates if needed;

- **batch_size** with two possible values in a grid search: *32, 64*;
- **epochs** with only a possible value that is *30*;

4.5.2 Modification in the training function

For the sake of brevity, the training function is not reported entirely: it essentially remains the same and only the changes are shown in the following snippet. Changes are only related to the *optimizer* and *scheduler* configuration.

```
1       optimizer = optim.Adam(net.parameters(), lr = 0.001)
2       scheduler = ReduceLROnPlateau(optimizer = optimizer,
3               mode = 'max', factor = 0.1,
4               patience = config["patience_onplat"],
5               min_lr=1e-06, verbose = True)
6         [...]
7       [...]
8   # AFTER EACH EPOCH:
9   scheduler.step(val_acc)
```

First, it initializes an Adam optimizer with a learning rate of 0.001.

Then the code sets up a *ReduceLROnPlateau* scheduler. This scheduler reduces the learning rate when the monitored quantity, in this case, validation accuracy ("*val_acc*", see at line 9), has stopped improving. The parameters for this scheduler include the "*mode", "factor", "patience*", and "*min_lr*". The "*mode*" is set to "*max*" because the validation accuracy has to increase. The "*factor*" parameter specifies the factor by which the learning rate will be reduced, and is set to *0.1*. The "*patience*" parameter specifies the number of epochs with no improvement after which the learning rate will be reduced, and is set to the value passed in from the configuration, *"config["patience_onplat"]*". Finally, the "*min_lr*" parameter specifies the minimum learning rate, i.e. the threshold, which is set to 1e-06.

The instruction shown at line 9 is actually executed **at the end of each epoch:** it updates the learning rate and checks if the validation accuracy

has improved. If there is no improvement in validation accuracy for *patience* number of epochs, the learning rate will be reduced by the *factor*.

4.5.3 Second results

Also in this case the baseline approach considered to train our model and classify the TV banners is **NOT_NOA**: no textual information have been involved, and no augmentation techniques have been applied on the training set of data. As before here are listed the summary tables. Each one has four columns: *valid loss, valid acc, train loss* and *train acc*, respectively the loss and accuracy values on the validation set and the loss and accuracy values on the training one.

valid_lo	valid_ac	train_lo	train_ac
0.4383	0.7647	0.6219	0.6061
0.6726	0.7647	0.5685	0.7677
0.4383	0.7647	0.4929	0.7677
0.907	0.7647	0.5685	0.7677
0.4383	0.7647	0.5685	0.7677
0.6726	0.7647	0.4929	0.7677
0.6726	0.7647	0.5685	0.7677
0.6726	0.7647	0.4929	0.7677
0.6726	0.7647	0.4929	0.7677
0.6726	0.7647	0.5685	0.7677
0.4383	0.7647	0.4929	0.7677
0.4383	0.7647	0.644	0.7677
0.4383	0.7647	0.4929	0.7677
0.4383	0.7647	0.4929	0.7677
0.4383	0.7647	0.4929	0.7677
0.6726	0.7647	0.4929	0.7677
0.4383	0.7647	0.5685	0.7677
0.4383	0.7647	0.4929	0.7677
0.6726	0.7647	0.644	0.7677
0.4383	0.7647	0.4929	0.7677
0.907	0.7647	0.4929	0.7677
0.6726	0.7647	0.5685	0.7677
0.6726	0.7647	0.4929	0.7677
0.4383	0.7647	0.4929	0.7677
0.6726	0.7647	0.4929	0.7677
0.4383	0.7647	0.4929	0.7677
0.4383	0.7647	0.4929	0.7677
0.4383	0.7647	0.5685	0.7677
0.4383	0.7647	0.5685	0.7677
0.907	0.7647	0.4929	0.7677

Figure 4.33: Adam, batch32, patience4

valid_lo	valid_ac	train_lo	train_ac
0.907	0.7647	0.5362	0.7677
0.6726	0.7647	0.4929	0.7677
0.6726	0.7647	0.644	0.7677
0.6726	0.7647	0.644	0.7677
0.4383	0.7647	0.4929	0.7677
0.4383	0.7647	0.5685	0.7677
0.6726	0.7647	0.4929	0.7677
0.4383	0.7647	0.5685	0.7677
0.6726	0.7647	0.4929	0.7677
0.4383	0.7647	0.644	0.7677
0.4383	0.7647	0.5685	0.7677
0.4383	0.7647	0.4929	0.7677
0.6726	0.7647	0.4929	0.7677
0.907	0.7647	0.4929	0.7677
0.4383	0.7647	0.644	0.7677
0.6726	0.7647	0.4929	0.7677
0.6726	0.7647	0.5685	0.7677
0.4383	0.7647	0.5685	0.7677
0.6726	0.7647	0.4929	0.7677
0.4383	0.7647	0.644	0.7677
0.4383	0.7647	0.644	0.7677
0.4383	0.7647	0.5685	0.7677
0.4383	0.7647	0.5685	0.7677
0.6726	0.7647	0.5685	0.7677
0.4383	0.7647	0.5685	0.7677
0.6726	0.7647	0.5685	0.7677
0.6726	0.7647	0.4929	0.7677
0.4383	0.7647	0.5685	0.7677
0.4383	0.7647	0.4929	0.7677
0.4383	0.7647	0.7195	0.7677

Figure 4.34: Adam, batch32, patience5

valid_lo	valid_ac	train_lo	train_ac
0.6726	0.7647	0.6305	0.6263
0.6726	0.7647	0.5685	0.7677
0.6726	0.7647	0.4929	0.7677
0.4383	0.7647	0.4929	0.7677
0.4383	0.7647	0.4929	0.7677
0.4383	0.7647	0.4929	0.7677
0.6726	0.7647	0.4929	0.7677
0.4383	0.7647	0.4929	0.7677
0.6726	0.7647	0.4929	0.7677
0.4383	0.7647	0.4929	0.7677
0.6726	0.7647	0.4929	0.7677
0.907	0.7647	0.5685	0.7677
0.4383	0.7647	0.644	0.7677
0.6726	0.7647	0.5685	0.7677
0.4383	0.7647	0.4929	0.7677
0.6726	0.7647	0.4929	0.7677
0.4383	0.7647	0.5685	0.7677
0.4383	0.7647	0.644	0.7677
0.907	0.7647	0.4929	0.7677
0.6726	0.7647	0.5685	0.7677
0.907	0.7647	0.4929	0.7677
0.6726	0.7647	0.4929	0.7677
0.6726	0.7647	0.5685	0.7677
0.4383	0.7647	0.4929	0.7677
0.4383	0.7647	0.5685	0.7677
0.4383	0.7647	0.4929	0.7677
0.4383	0.7647	0.5685	0.7677
0.907	0.7647	0.4929	0.7677
0.4383	0.7647	0.4929	0.7677
0.4383	0.7647	0.644	0.7677

Figure 4.35: Adam, batch32, patience6

valid_lo	valid_ac	train_lo	train_ac
0.4383	0.7647	0.6061	0.7374
0.4383	0.7647	0.4929	0.7677
0.6726	0.7647	0.4929	0.7677
0.6726	0.7647	0.5685	0.7677
0.6726	0.7647	0.4929	0.7677
0.4383	0.7647	0.5685	0.7677
0.907	0.7647	0.4929	0.7677
0.907	0.7647	0.5685	0.7677
0.6726	0.7647	0.4929	0.7677
0.4383	0.7647	0.5685	0.7677
0.6726	0.7647	0.644	0.7677
0.4383	0.7647	0.5685	0.7677
0.4383	0.7647	0.5685	0.7677
0.4383	0.7647	0.5685	0.7677
0.4383	0.7647	0.5685	0.7677
0.6726	0.7647	0.4929	0.7677
0.4383	0.7647	0.5685	0.7677
0.4383	0.7647	0.7195	0.7677
0.907	0.7647	0.4929	0.7677
0.4383	0.7647	0.4929	0.7677
0.4383	0.7647	0.5685	0.7677
0.4383	0.7647	0.5685	0.7677
0.907	0.7647	0.4929	0.7677
0.4383	0.7647	0.4929	0.7677
0.6726	0.7647	0.4929	0.7677
0.4383	0.7647	0.4929	0.7677
0.4383	0.7647	0.644	0.7677
0.6726	0.7647	0.5685	0.7677
0.6726	0.7647	0.5685	0.7677
0.4383	0.7647	0.5685	0.7677

Figure 4.36: Adam, batch32, patience7

valid_lo	valid_ac	train_lo	train_ac
0.5491	0.7647	0.6772	0.3838
0.5486	0.7647	0.5512	0.7677
0.5486	0.7647	0.5447	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5512	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5447	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5706	0.7677
0.5486	0.7647	0.5447	0.7677
0.5486	0.7647	0.5512	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5318	0.7677
0.5486	0.7647	0.5706	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5447	0.7677
0.5486	0.7647	0.5253	0.7677
0.5486	0.7647	0.5706	0.7677
0.5486	0.7647	0.5318	0.7677
0.5486	0.7647	0.5577	0.7677
0.5486	0.7647	0.5642	0.7677
0.5486	0.7647	0.5577	0.7677
0.5486	0.7647	0.5706	0.7677
0.5486	0.7647	0.5512	0.7677
0.5486	0.7647	0.5642	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5512	0.7677
0.5486	0.7647	0.5512	0.7677

Figure 4.37: Adam, batch64, patience4

valid_lo	valid_ac	train_lo	train_ac
0.907	0.7647	0.5362	0.7677
0.6726	0.7647	0.4929	0.7677
0.6726	0.7647	0.644	0.7677
0.6726	0.7647	0.644	0.7677
0.4383	0.7647	0.4929	0.7677
0.4383	0.7647	0.5685	0.7677
0.6726	0.7647	0.4929	0.7677
0.4383	0.7647	0.5685	0.7677
0.6726	0.7647	0.4929	0.7677
0.4383	0.7647	0.644	0.7677
0.4383	0.7647	0.5685	0.7677
0.4383	0.7647	0.4929	0.7677
0.6726	0.7647	0.4929	0.7677
0.907	0.7647	0.4929	0.7677
0.4383	0.7647	0.644	0.7677
0.6726	0.7647	0.4929	0.7677
0.6726	0.7647	0.5685	0.7677
0.4383	0.7647	0.5685	0.7677
0.6726	0.7647	0.4929	0.7677
0.4383	0.7647	0.644	0.7677
0.4383	0.7647	0.644	0.7677
0.4383	0.7647	0.5685	0.7677
0.4383	0.7647	0.5685	0.7677
0.6726	0.7647	0.5685	0.7677
0.4383	0.7647	0.5685	0.7677
0.6726	0.7647	0.5685	0.7677
0.6726	0.7647	0.4929	0.7677
0.4383	0.7647	0.5685	0.7677
0.4383	0.7647	0.4929	0.7677
0.4383	0.7647	0.7195	0.7677

Figure 4.38: Adam, batch64, patience5

valid_lo	valid_ac	train_lo	train_ac
0.5492	0.7647	0.6606	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5706	0.7677
0.5486	0.7647	0.5318	0.7677
0.5486	0.7647	0.5577	0.7677
0.5486	0.7647	0.5577	0.7677
0.5486	0.7647	0.5447	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5318	0.7677
0.5486	0.7647	0.5447	0.7677
0.5486	0.7647	0.5253	0.7677
0.5486	0.7647	0.5447	0.7677
0.5486	0.7647	0.5447	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5124	0.7677
0.5486	0.7647	0.5577	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5318	0.7677
0.5486	0.7647	0.5318	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5188	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5318	0.7677
0.5486	0.7647	0.5383	0.7677
0.5486	0.7647	0.5512	0.7677
0.5486	0.7647	0.5512	0.7677
0.5486	0.7647	0.5577	0.7677
0.5486	0.7647	0.5447	0.7677

Figure 4.39: Adam, batch64, patience6

valid_lo	valid_ac	train_lo	train_ac
0.4383	0.7647	0.6061	0.7374
0.4383	0.7647	0.4929	0.7677
0.6726	0.7647	0.4929	0.7677
0.6726	0.7647	0.5685	0.7677
0.6726	0.7647	0.4929	0.7677
0.4383	0.7647	0.5685	0.7677
0.907	0.7647	0.4929	0.7677
0.907	0.7647	0.5685	0.7677
0.6726	0.7647	0.4929	0.7677
0.4383	0.7647	0.5685	0.7677
0.6726	0.7647	0.644	0.7677
0.4383	0.7647	0.5685	0.7677
0.4383	0.7647	0.5685	0.7677
0.4383	0.7647	0.5685	0.7677
0.4383	0.7647	0.5685	0.7677
0.6726	0.7647	0.4929	0.7677
0.4383	0.7647	0.5685	0.7677
0.4383	0.7647	0.7195	0.7677
0.907	0.7647	0.4929	0.7677
0.4383	0.7647	0.4929	0.7677
0.4383	0.7647	0.5685	0.7677
0.4383	0.7647	0.5685	0.7677
0.907	0.7647	0.4929	0.7677
0.4383	0.7647	0.4929	0.7677
0.6726	0.7647	0.4929	0.7677
0.4383	0.7647	0.4929	0.7677
0.4383	0.7647	0.644	0.7677
0.6726	0.7647	0.5685	0.7677
0.6726	0.7647	0.5685	0.7677
0.4383	0.7647	0.5685	0.7677

Figure 4.40: Adam, batch64, patience7

4.5.4 Discussions

Based on these new values, it seems that the model is not performing well, once again. Within all these trials the accuracy on the training set is not blocked to very low values as before, however its value is constant during the whole training step. The same is for the accuracy on validation set, indicating that the model is not learning any new information.

The loss values seem to be experiencing some fluctuations, changing from one epoch to another and also changing among the different trials.

So also in this case the performances are bad: again no other approaches

(NOT_A, T_NOA, T_A) will be taken into account at the moment, since the issue appears to be neither the model, nor the hyperparameters. It is worth noting that the dataset introduced a moderate imbalance, with 128 samples in the class Internal and 39 with the class External: before going on with further experiments, some augmentation techniques will be applied to the minority class in order to handle this inconsistency and try to improve the overall model performance.

4.6 Third run

The dataset presented a moderate imbalance with 128 samples in the class Internal and 39 with the class External: this can be the reason behind the bad performances achieved until now. To address this issue, some augmentation techniques will be applied to the External class, in order to balance the classes distribution. To do this, the minority class will be augmented with 78 samples resulting from the application of image transformations techniques applied on the 39 original External samples that were present in the original dataset. These include random rotations, random flips, color jitter, and normalization, allowing for more varied training data. These modify the color and the graphical properties of an image, or the geometry of an image by changing its size, orientation, and flipping it horizontally or vertically, in order to make the model more robust to variations in lighting conditions, color schemes and geometrical design.

In order to add these 78 samples, the modifications start from the *csv* file that has been used before to load the dataset. Here an example of how it will appear now:

136_A	['piacevolr	External
136_B	['piacevolr	External
138_A	['privacy', '	External
138_B	['privacy', '	External
140_A	['splendidl	External
140_B	['splendidl	External
148_A	['privacy', '	External
148_B	['privacy', '	External
151_A	['privacy', '	External
151_B	['privacy', '	External
158_A	['privacy', '	External
158_B	['privacy', '	External
160_A	['privacy', '	External
160_B	['privacy', '	External
166_A	['privacy', '	External
166_B	['privacy', '	External
168_A	['pura', '66	External
168_B	['pura', '66	External
184_A	['privacy', '	External
184_B	['privacy', '	External
19_A	['lno', 'esis	External
19_B	['lno', 'esis	External
193_A	['privacy', '	External
193_B	['privacy', '	External
195_A	['privacy', '	External

Figure 4.41: An extract of the new CSV file for augmenting minority class

It is possible to see that in a very simple way 78 additional External samples have been added to the original dataset: these have been simply duplicated (they were 39 samples before). To be more precise, only the sentence part has been really duplicated, while the first column still maintains information about the filename of each image but with a modification: the difference here is that for each External sample originally present in the dataset, **two** further samples have been added (this allows to triplicate the External class, which will be made of 39 original samples and 78 transformed ones). So while the two sentences added are the same, their filenames change: in fact they are followed by a suffix $\_A$ or $\_B$. This letter will be useful in the custom dataset definition in order to apply a set A of transformations to the image whose filename ends with $\_A$, and a different set of transformations to the other image whose filename ends with $\_B$.

(*E.G. The dataset has an External sample with filename 0 and sentence "this is a sentence". The two samples added to augment the class will have the same sentence but different filenames: 0_A and 0_B. So essentially the image is still the same, but a set A of transformations will be applied to 0_A, for example modifying its colors, and a set 0_B of different transformation will be applied to 0_B, for example modifying it by some geometrical*

criteria. At the end, in the dataset three different versions of the same image will be present: the original one (0), the image transformed by its colors (0_A,) and the last one transformed geometrically (0_B).)

To sum up, the *newly* composed dataset contains 245 samples: 128 Internal and 117 External.

In order to better explain the changes, the workflow will be retraced, however highlighting only the differences occured after the modifications explained above.

4.6.1 A new custom Dataset definition

```
1  import os
2  label_enc=LabelEncoder()
3
4  full_dataset = pd.read_csv('/content/new_OCR_text.csv')
5  full_dataset.info()
6
7  X, y = full_dataset.iloc[:, :-1], full_dataset.iloc[:, [-1]]
8
9  X_train, X_test, y_train, y_test = train_test_split(X, y,
10                                test_size=0.2, stratify = y)
11
12 X_train, X_val, y_train, y_val = train_test_split(X_train,
13                 y_train,test_size=0.2, stratify = y_train)
14
15 X_train, X_val, X_test = X_train.reset_index(drop = True),
16                           X_val.reset_index(drop = True),
17                           X_test.reset_index(drop = True)
18 y_train, y_val, y_test = y_train.reset_index(drop = True),
19                           y_val.reset_index(drop = True),
20                           y_test.reset_index(drop = True)
```

The first step consists in loading the *new* dataset from its location using the *pd.read_csv()* function from the *pandas* library. The following line allows to display information about the dataset using the *info()* method of the *DataFrame* object.

At line 7, the code splits the dataset into feature and target variables, where X contains all columns except the last one, which is the target variable and

belongs to *y*. Then there is another data splitting into training and testing sets using *train_test_split()* function from the *sklearn.model_selection* module. The *test_size* parameter is set to 0.2, indicating that 20% of the data should be used for testing. The *stratify* parameter is set to *y*, meaning that the split is performed in a way that preserves the proportion of samples for each class in *y*. The usage of this parameter is optional, however it has been used in order to preserve the classes distribution both in training and in testing set. So the resulting sets have the following sizes: 196 samples in the training one and 49 samples in the other one.

The snippet further splits the training set into training and validation sets, again using *train_test_split()* function. The *test_size* parameter is set to 0.2, indicating also in this case that 20% of the training data should be used for validation. The stratify parameter in this case is set to *y_train* for a better comprehension and coherence with the current split: the proportion of samples for each class would be preserved also with *y*, since the distribution is the same thanks to the previous split. After this operation, the resulting sizes are: 156 samples in the training set, 40 in the validation set and 49 in the testing one. In particular this is the classes distribution among the different sets:

- **Training set**: 81 Internal, 75 External;
- **Validation set**: 21 Internal, 19 External;
- **Test set**: 26 Internal, 23 External.

The last operations at line 15 and 18 performed resets the index of the dataframes to ensure they are properly aligned for later processing.

At this point it is possible to re-define our custom dataset:

```
1 y_train.label=label_enc.fit_transform(y_train.label)
2 y_val.label=label_enc.fit_transform(y_val.label)
3 y_test.label = label_enc.fit_transform(y_test.label)
4
5 path_to_images = "/my_path_to_images"
6
7 if(text):
8   class CustomImageDataset(Dataset):
9       def __init__(self, features, target, embeddings,
```

```
    transform=None):
10          self.features = features
11          self.transform = transform
12          self.target = target
13          self.embeddings = embeddings
14
15      def __len__(self):
16          return len(self.features)
17
18      def __getitem__(self, idx):
19          filename = str(self.features.loc[idx, 'filename'])
20          img_path = path_to_images + filename + ".png"
21          if os.path.isfile(img_path):
22            image = read_image(img_path,
23                               ImageReadMode.RGB).float()
24
25            if self.transform:
26              image = self.transform(image)
27          else:
28            if(filename.endswith('A')):
29              filename = filename[:len(filename)-2]
30              img_path = path_to_images + filename + ".png"
31              image = read_image(img_path,
32                                 ImageReadMode.RGB).float()
33              transform = transform=transforms.Compose([
34                        transforms.ToPILImage(),
35                        transforms.Resize((256, 256)),
36                        transforms.ColorJitter(
37                            brightness=.5,
38                            contrast=.5,
39                            saturation=.5),
40                        transforms.RandomVerticalFlip(p=1),
41                        transforms.ToTensor()])
42              image = transform(image)
43
44            else:
45              filename = filename[:len(filename)-2]
46              img_path = path_to_images + filename + ".png"
47              image = read_image(img_path,
48                        ImageReadMode.RGB).float()
49              transform = transforms.Compose([
50                        transforms.ToPILImage(),
```

```
                        transforms.Resize((256, 256)),
                        transforms.ColorJitter(
                            brightness=0.6,
                            contrast=1.3,
                            saturation=1.1,
                            hue=.2),
                        transforms.ToTensor()])
            image = transform(image)

        label= self.target.loc[idx, 'label']

        if(text):
          if filename in self.embeddings.keys():
            text_feature =
              torch.from_numpy(self.embeddings[filename])
            return image, text_feature, label

        else:
           return image, label
```

Essentially the class is the same as before: the *__init__* method still takes four arguments, which are stored as instance variables, and initializes the dataset with them:

- **features** is a dataframe that contains information about the images to be loaded;
- **target** is a dataframe that contains the corresponding labels for each image; **embeddings** is a dictionary containing the embedded vectors of the text associated with each image;
- **transform** is an optional argument which allows to pass in a list of image transformations to be applied to the images before they are fed into the model.

The *__len__* method returns the length of the dataset, which is equal to the number of images to be loaded.

The *__getitem__* method takes an index *idx* and returns a tuple containing the image, any associated text features, and the target label for the corresponding image.

First, the filename and image path are obtained from the *features* DataFrame. Then at line 21 the code checks the image file exists at the specified path: in this case, it means that the image belongs to the Internal class or it is an original External sample, since the path has not been modified in none of the previously described ways. So the image is read using the *read_image* function and transformed if the *transform* argument is provided.

But if the file does not exist(line 27) the scenario is different and it is already possible to understand that the current image is an External sample modified, since the code is trying to read an image that does not exist (*i.e. the snippet is looking for 0_A or 0_B file, but it should look for 0 instead).* Consequently, at line, 28, the code checks the *filename* to see if it ends with the letter '*A*' or not:

- if the *filename* ends with '*A*', the corresponding original image is read, after recovering its real path (just removing *"_A"*), and data augmentation techniques like resizing, cropping, and color jittering are applied to the image (lines 33-42);
- otherwise, at line 44, the *filename* ends with *B*. Also in this case the corresponding original image is read by recovering its real path (just removing *"_B"*) and only resizing and color jittering are applied to the image.

The target label for the current image instead is obtained from the *target* DataFrame at line 60.

The last part of the code refers to adding *text features* to the image dataset: it checks if the *text* variable is *True* and if the text *embeddings* for the current *filename* exist in the provided *embeddings* dictionary. If both conditions are met, it extracts the embeddings and converts them to a PyTorch tensor using *torch.from_numpy()*. Finally, it returns a tuple consisting of the *image* tensor, *text feature* tensor, and *label* tensor if *text* is *True*, meaning that the textual information should be taken into account for the current approach. Otherwise, it returns a tuple consisting of only the *image tensor* and *label* tensor.

The *embeddings* dictionary that will be fed to the *CustomImageDataset* objects is a data structure containing the embedded vectors of the associated

to each image. These information have been stored in a *.npy* file at section 4.3: the code snippet is not shown again since it remains the same.

Mean and standard deviation calculations still are the same as before (please refer to 4.4.1).

At this stage there is the initialization of the custom dataset in order to obtain the **normalized training data**: this snippet also remains the same (please refer to 4.4.1).

Also the Neural Network is not affected by any modification (please refer to 4.4.2).

4.6.2 Modifications to hyperparameters tuning

For this run there is another configuration dictionary for hyperparameter tuning using the *Ray Tune* library: the learning rate is not configurable with different parameters through *Ray Tune*, but it is fixed at the beginning (it starts from *0.001*) and will be updated automatically by the scheduler provided by *torch.optim*. This is *lr_scheduler.ReduceLROnPlateau*, that as specified in the previous sections can reduce the learning rate by a factor specified in input, when the monitored quantity has no improvements.

```
    config = {
    "patience_onplat": tune.grid_search([4,5,6,7]),
    "batch_size": tune.grid_search([16]),
    "epochs": tune.choice([30])
}
```

The configuration dictionary contains three hyperparameters:

- **patience_onplat** with four possible values: this is the number of epochs with no improvements after which the learning rate will be updated thanks to the *ReduceLROnPlateau* scheduler. These values have been chosen considering the threshold and the number of epochs, in order to let the learning rate make all the updates if needed;
- **batch_size** with only a possible value that is *16*;
- **epochs** with only a possible value that is *30*;

The training function has not been further modified after the last run, please refer to 4.5.2.

4.6.3 Third results

The baseline approach considered to train our model and classify the TV banners is always **NOT_NOA**: no textual information have been involved, and no augmentation techniques have been applied on the training set of data. As before here are listed the summary tables. Each one has four columns: *valid loss, valid acc, train loss* and *train acc*, respectively the loss and accuracy values on the validation set and the loss and accuracy values on the training one. Each table has at least one row highlighted in green which identifies the *best* combination of values: the choice of the *best* epoch is made by analyzing accuracy values and loss values on training and validation set. The first criteria is to pick an epoch whose accuracy on training is high enough. Then of course also the accuracy on validation should have an high enough value; finally the performances are analyzed also with the loss values, that should be as low as possible.

NOT_NOA results

valid_lo	valid_ac	train_lo	train_ac
0.6191	0.5263	0.6849	0.5235
0.5791	0.8684	0.6115	0.6235
0.5425	0.8421	0.5438	0.8294
0.6158	0.7368	0.6662	0.5765
0.6931	0.4737	0.6081	0.5176
0.6931	0.4737	0.6931	0.4765
0.6931	0.4737	0.6931	0.4765
0.6931	0.4737	0.6931	0.4765
0.6931	0.4737	0.6931	0.4765
0.6931	0.4737	0.6931	0.4765
0.6931	0.4737	0.6931	0.4765
0.6931	0.4737	0.6931	0.4765
0.6931	0.4737	0.6931	0.4765
0.6931	0.4737	0.6931	0.4765
0.6931	0.4737	0.6931	0.4765
0.6931	0.4737	0.6931	0.4765
0.6931	0.4737	0.6931	0.4765
0.6931	0.4737	0.6931	0.4765
0.6931	0.4737	0.6931	0.4765
0.6931	0.4737	0.6931	0.4765
0.6931	0.4737	0.6931	0.4765
0.6931	0.4737	0.6931	0.4765
0.6931	0.4737	0.6931	0.4765
0.6931	0.4737	0.6931	0.4765
0.6931	0.4737	0.6931	0.4765
0.6931	0.4737	0.6931	0.4765
0.6931	0.4737	0.6931	0.4765
0.6931	0.4737	0.6931	0.4765
0.6931	0.4737	0.6931	0.4765
0.6931	0.4737	0.6931	0.4765
0.6931	0.4737	0.6931	0.4765

Figure 4.42: NOT_NOA patience 4

valid_lo	valid_ac	train_lo	train_ac
0.693	0.4737	0.7006	0.4647
0.6273	0.8684	0.6942	0.7176
0.527	0.8421	0.6001	0.8
0.699	0.6316	0.5426	0.7765
0.4753	0.8421	0.6503	0.6588
0.6001	0.7368	0.5521	0.7706
0.5531	0.7895	0.5759	0.7353
0.7505	0.5789	0.5786	0.7412
0.6017	0.7105	0.6994	0.6059
0.5563	0.7368	0.6113	0.7059
0.5216	0.7368	0.5836	0.7294
0.5781	0.7368	0.5832	0.7294
0.5147	0.7895	0.5758	0.7353
0.592	0.7632	0.5326	0.7765
0.5226	0.7632	0.5303	0.7824
0.5574	0.7632	0.5304	0.7824
0.5604	0.7632	0.5277	0.7824
0.5587	0.7632	0.5266	0.7824
0.5226	0.7632	0.5269	0.7824
0.4878	0.7632	0.5235	0.7824
0.4878	0.7632	0.5303	0.7824
0.501	0.7632	0.5337	0.7824
0.5226	0.7632	0.5269	0.7824
0.5573	0.7632	0.5269	0.7824
0.592	0.7632	0.5235	0.7824
0.5226	0.7632	0.5303	0.7824
0.5226	0.7632	0.5337	0.7824
0.5573	0.7632	0.5337	0.7824
0.5226	0.7632	0.5303	0.7824
0.5226	0.7632	0.5269	0.7824

Figure 4.43: NOT_NOA patience 5

valid_lo	valid_ac	train_lo	train_ac
0.637	0.8421	0.6528	0.5824
0.5986	0.6316	0.6284	0.7059
0.5468	0.8158	0.6167	0.7412
0.7994	0.4737	0.5387	0.7588
0.8688	0.4737	0.8292	0.4765
0.8688	0.4737	0.8292	0.4765
0.8688	0.4737	0.8326	0.4765
0.7994	0.4737	0.8326	0.4765
0.7994	0.4737	0.836	0.4765
0.9035	0.4737	0.8292	0.4765
0.7647	0.4737	0.8292	0.4765
0.8341	0.4737	0.8326	0.4765
0.8341	0.4737	0.8394	0.4765
0.7994	0.4737	0.8258	0.4765
0.7647	0.4737	0.8428	0.4765
0.7994	0.4737	0.8326	0.4765
0.8688	0.4737	0.836	0.4765
0.9035	0.4737	0.836	0.4765
0.7647	0.4737	0.8394	0.4765
0.8341	0.4737	0.8292	0.4765
0.9383	0.4737	0.8428	0.4765
0.7994	0.4737	0.8326	0.4765
0.9035	0.4737	0.8394	0.4765
0.7647	0.4737	0.8292	0.4765
0.8341	0.4737	0.836	0.4765
0.8341	0.4737	0.8428	0.4765
0.7994	0.4737	0.8394	0.4765
0.8688	0.4737	0.836	0.4765
0.8341	0.4737	0.8394	0.4765
0.8341	0.4737	0.8462	0.4765

Figure 4.44: NOT_NOA patience 6

valid_lo	valid_ac	train_lo	train_ac
0.6109	0.5263	0.6821	0.5235
0.5584	0.8684	0.6051	0.6059
0.4607	0.8684	0.5834	0.8235
0.4174	0.8684	0.5148	0.7941
0.4174	0.8684	0.5022	0.8118
0.4174	0.8684	0.4873	0.8235
0.4174	0.8684	0.4849	0.8294
0.4869	0.8684	0.4849	0.8294
0.4174	0.8684	0.4814	0.8294
0.4522	0.8684	0.4814	0.8294
0.4174	0.8684	0.4849	0.8294
0.4522	0.8684	0.478	0.8294
0.4174	0.8684	0.478	0.8294
0.4522	0.8684	0.4883	0.8294
0.4869	0.8684	0.4814	0.8294
0.4522	0.8684	0.4883	0.8294
0.4869	0.8684	0.478	0.8294
0.4174	0.8684	0.4883	0.8294
0.4522	0.8684	0.4814	0.8294
0.4174	0.8684	0.4814	0.8294
0.4174	0.8684	0.4814	0.8294
0.4174	0.8684	0.4814	0.8294
0.4869	0.8684	0.4849	0.8294
0.4522	0.8684	0.4883	0.8294
0.4174	0.8684	0.478	0.8294
0.4869	0.8684	0.4849	0.8294
0.4522	0.8684	0.4917	0.8294
0.4522	0.8684	0.4814	0.8294
0.4174	0.8684	0.4849	0.8294
0.4869	0.8684	0.4849	0.8294

Figure 4.45: NOT_NOA patience 7

The approaches with *patience* 4 and 6 have a quite similar pattern (figure 4.42 and 4.44): the validation accuracy increases up to values higher than *0.80*, before dropping to *0.473* in the following epochs. Also the training accuracy starts low (*0.524* and *0.5824*) and gradually increases to a high of *0.829* and *0.759* respectively, before dropping to *0.476* in the following epochs.

About the other values:

- with *patience* 4, the validation loss is highest at the beginning (*0.619*), drops to a low of *0.542* on the third epoch, then increases to *0.693* from the seventh epoch onwards. The training loss starts high (*0.684*)

and gradually decreases to a low of *0.543* on the third epoch before increasing to *0.693* too from the seventh epoch onwards.

- with *patience 6*, the validation and accuracy loss start low but then increases to a high of *0.903* and *0.846* respectively

The best trial among these shown above seems to be the one with *patience* 5: it appears that over time, the model seems to have improved, as the values for *valid_accuracy* and *train_acc* increased, and the values for *valid_loss* and *train_loss* decreased. In particular the second row highlighted in green, in the center of the table, seems to be the best until now: with respect with the third epoch, in fact here the accuracy on training and validation sets are slightly lower, but the loss values have improved.

So with respect with the other trials, this is not bad at all and allows to experiment also with other approaches (**NOT_A, T_NOA, T_A**).

NOT_A results

valid_los	valid_ac	train_los	train_ac
0.4383	0.9	0.5923	0.7308
0.4174	0.9	0.5224	0.791
0.3966	0.9	0.5212	0.7923
0.4174	0.9	0.5203	0.7923
0.3966	0.9	0.5301	0.7833
0.4383	0.9	0.5233	0.7897
0.3966	0.9	0.5216	0.7923
0.3966	0.9	0.5225	0.7897
0.4383	0.9	0.522	0.791
0.4174	0.9	0.528	0.7859
0.4383	0.9	0.5233	0.791
0.3966	0.9	0.5237	0.7897
0.3966	0.9	0.5216	0.7923
0.3966	0.9	0.5195	0.7936
0.4799	0.9	0.5216	0.7923
0.3966	0.9	0.522	0.791
0.4174	0.9	0.5258	0.7872
0.3966	0.9	0.5203	0.7923
0.4174	0.9	0.5178	0.7949
0.4383	0.9	0.525	0.7885
0.3966	0.9	0.5224	0.791
0.3966	0.9	0.5203	0.7936
0.4174	0.9	0.5237	0.7897
0.3966	0.9	0.5216	0.7923
0.4174	0.9	0.5241	0.7897
0.3966	0.9	0.519	0.7936
0.4383	0.9	0.5254	0.7872
0.3966	0.9	0.5281	0.7846
0.4174	0.9	0.5241	0.7885
0.4174	0.9	0.5207	0.7923

Figure 4.46: NOT_A patience 4

valid_los	valid_ac	train_los	train_ac
0.4595	0.875	0.6539	0.6013
0.3966	0.9	0.5412	0.7731
0.7716	0.525	0.7853	0.5269
0.7716	0.525	0.7937	0.5192
0.7924	0.525	0.7933	0.5192
0.7716	0.525	0.795	0.5192
0.7716	0.525	0.7946	0.5192
0.8341	0.525	0.7954	0.5192
0.7924	0.525	0.7941	0.5192
0.7716	0.525	0.7954	0.5192
0.7924	0.525	0.7933	0.5192
0.7299	0.525	0.7946	0.5192
0.8341	0.525	0.795	0.5192
0.7716	0.525	0.7941	0.5192
0.8549	0.525	0.7954	0.5192
0.8341	0.525	0.7941	0.5192
0.7924	0.525	0.7937	0.5192
0.7716	0.525	0.7946	0.5192
0.7716	0.525	0.795	0.5192
0.7924	0.525	0.7933	0.5192
0.8341	0.525	0.7929	0.5192
0.7716	0.525	0.7937	0.5192
0.7299	0.525	0.7933	0.5192
0.7508	0.525	0.7933	0.5192
0.8549	0.525	0.7946	0.5192
0.7716	0.525	0.7937	0.5192
0.8133	0.525	0.7937	0.5192
0.8133	0.525	0.7937	0.5192
0.8133	0.525	0.7929	0.5192
0.7924	0.525	0.7946	0.5192

Figure 4.47: NOT_A patience 5

valid_los	valid_ac	train_los	train_ac
0.4175	0.9	0.6242	0.7256
0.4174	0.9	0.531	0.7821
0.3966	0.9	0.5297	0.7846
0.8549	0.475	0.6354	0.6769
0.8549	0.475	0.8328	0.4808
0.8341	0.475	0.832	0.4808
0.8549	0.475	0.8332	0.4808
0.8341	0.475	0.8324	0.4808
0.8133	0.475	0.8337	0.4808
0.7924	0.475	0.832	0.4808
0.8133	0.475	0.8332	0.4808
0.8341	0.475	0.8337	0.4808
0.8133	0.475	0.8332	0.4808
0.8758	0.475	0.8324	0.4808
0.8341	0.475	0.832	0.4808
0.8133	0.475	0.8332	0.4808
0.8341	0.475	0.8332	0.4808
0.8341	0.475	0.8332	0.4808
0.8341	0.475	0.8324	0.4808
0.8341	0.475	0.832	0.4808
0.8758	0.475	0.8315	0.4808
0.8133	0.475	0.8328	0.4808
0.8341	0.475	0.8337	0.4808
0.8133	0.475	0.8337	0.4808
0.8133	0.475	0.8315	0.4808
0.7716	0.475	0.8328	0.4808
0.8341	0.475	0.8324	0.4808
0.8133	0.475	0.8328	0.4808
0.7924	0.475	0.8324	0.4808
0.8341	0.475	0.832	0.4808

Figure 4.48: NOT_A patience 6

valid_los	valid_ac	train_los	train_ac
0.4591	0.85	0.6289	0.6474
0.7716	0.525	0.7181	0.5962
0.7716	0.525	0.7933	0.5192
0.7924	0.525	0.7946	0.5192
0.7924	0.525	0.7937	0.5192
0.7716	0.525	0.7941	0.5192
0.8549	0.525	0.7946	0.5192
0.7924	0.525	0.7946	0.5192
0.8133	0.525	0.7937	0.5192
0.7716	0.525	0.7946	0.5192
0.7924	0.525	0.795	0.5192
0.8133	0.525	0.7929	0.5192
0.7716	0.525	0.7946	0.5192
0.7924	0.525	0.7941	0.5192
0.7716	0.525	0.795	0.5192
0.8341	0.525	0.7937	0.5192
0.8133	0.525	0.7937	0.5192
0.7716	0.525	0.7937	0.5192
0.7924	0.525	0.7929	0.5192
0.7716	0.525	0.7937	0.5192
0.8133	0.525	0.7946	0.5192
0.7508	0.525	0.795	0.5192
0.7716	0.525	0.7946	0.5192
0.7924	0.525	0.7933	0.5192
0.7508	0.525	0.7946	0.5192
0.8133	0.525	0.7924	0.5192
0.7716	0.525	0.7946	0.5192
0.7924	0.525	0.7941	0.5192
0.7716	0.525	0.7929	0.5192
0.7716	0.525	0.795	0.5192

Figure 4.49: NOT_A patience 7

A good result has been achieved with *patience* 4 here: it seems that the model is performing well. Both the training and validation accuracy values are consistently high (*0.9*), even if the validation one is constant from the beginning. However the loss values for both the training and validation sets indicate an overall good performance of the model: in fact are relatively low, indicating that the model is fitting the data well. The best epochs can be considered the ones highlighted in green in the center of the table, reached after a prolonged training. Regarding the fact that the accuracy is higher on the validation with respect to the training set, this may be caused by the Dropout layers: when training, a percentage of the features are set to zero. When validating and testing, all features are used (and are

scaled appropriately). So the model at test time could be more robust and can lead to higher testing accuracies. Another problem could be related to the dataset: smaller datasets have smaller variance. In general an higher number of samples could lead to overall better performances (especially, an higher number of *original* samples, since the External class has more augmented samples than real ones).

The other trials in this case are not good as this: with *patience* 5, for example, the loss values are high both on validation and training. And the accuracy on training is stuck already at the fourth epoch. About the same performances with *patience* 6; while with *patience* 7 the training on the accuracy is constantly low throughout the overall run: clearly the worse performance.

T_NOA results

valid_lo	valid_ac	train_lo	train_ac
0.5475	0.9211	0.6487	0.5824
0.4125	0.9211	0.5581	0.8353
0.3758	0.9211	0.4759	0.8353
0.3764	0.9211	0.4758	0.8353
0.4105	0.9211	0.4813	0.8294
0.3758	0.9211	0.486	0.8353
0.3758	0.9211	0.4758	0.8353
0.3758	0.9211	0.4779	0.8353
0.3758	0.9211	0.4792	0.8353
0.3758	0.9211	0.4826	0.8353
0.3758	0.9211	0.4792	0.8353
0.3758	0.9211	0.4792	0.8353
0.3758	0.9211	0.4792	0.8353
0.4105	0.9211	0.4758	0.8353
0.3758	0.9211	0.4826	0.8353
0.4105	0.9211	0.4792	0.8353
0.4452	0.9211	0.4724	0.8353
0.3758	0.9211	0.4826	0.8353
0.3758	0.9211	0.4758	0.8353
0.3758	0.9211	0.4826	0.8353
0.4105	0.9211	0.4758	0.8353
0.4105	0.9211	0.4758	0.8353
0.4452	0.9211	0.4758	0.8353
0.3758	0.9211	0.4724	0.8353
0.3758	0.9211	0.4792	0.8353
0.4105	0.9211	0.4724	0.8353
0.3758	0.9211	0.4724	0.8353
0.4105	0.9211	0.4724	0.8353
0.3758	0.9211	0.4724	0.8353
0.4105	0.9211	0.4792	0.8353

Figure 4.50: T_NOA patience 4

valid_lo	valid_ac	train_lo	train_ac
0.652	0.6053	0.6937	0.5235
0.7959	0.4737	0.5592	0.7647
0.3575	0.9474	0.5216	0.7824
0.4105	0.9211	0.4724	0.8353
0.3758	0.9211	0.4792	0.8353
0.3758	0.9211	0.4724	0.8353
0.3758	0.9211	0.4724	0.8353
0.3758	0.9211	0.486	0.8353
0.3758	0.9211	0.4758	0.8353
0.4105	0.9211	0.4792	0.8353
0.3758	0.9211	0.4724	0.8353
0.3758	0.9211	0.4724	0.8353
0.3758	0.9211	0.4826	0.8353
0.4105	0.9211	0.4792	0.8353
0.4452	0.9211	0.4758	0.8353
0.3758	0.9211	0.4758	0.8353
0.3758	0.9211	0.4826	0.8353
0.3758	0.9211	0.4758	0.8353
0.3758	0.9211	0.4792	0.8353
0.3758	0.9211	0.4826	0.8353
0.3758	0.9211	0.4758	0.8353
0.3758	0.9211	0.4792	0.8353
0.3758	0.9211	0.4758	0.8353
0.3758	0.9211	0.486	0.8353
0.3758	0.9211	0.4724	0.8353
0.4105	0.9211	0.4758	0.8353
0.4105	0.9211	0.4792	0.8353
0.3758	0.9211	0.4758	0.8353
0.3758	0.9211	0.4724	0.8353
0.4452	0.9211	0.4724	0.8353

Figure 4.51: T_NOApatience 5

valid_lo	valid_ac	train_lo	train_ac
0.5923	0.5263	0.6789	0.5235
0.5177	0.9211	0.6081	0.7059
0.4105	0.9211	0.4999	0.8353
0.3864	0.8947	0.4758	0.8353
0.4312	0.8947	0.486	0.8353
0.3965	0.8947	0.4758	0.8353
0.3897	0.9474	0.5218	0.7941
0.3897	0.9474	0.4826	0.8353
0.3549	0.9474	0.4859	0.8235
0.7994	0.4737	0.5031	0.7824
0.8688	0.4737	0.8289	0.4765
0.7994	0.4737	0.8462	0.4765
0.8688	0.4737	0.8394	0.4765
0.7994	0.4737	0.8326	0.4765
0.7994	0.4737	0.8428	0.4765
0.7647	0.4737	0.8394	0.4765
0.9035	0.4737	0.8462	0.4765
0.8688	0.4737	0.8394	0.4765
0.8688	0.4737	0.8326	0.4765
0.7994	0.4737	0.8292	0.4765
0.8341	0.4737	0.8428	0.4765
0.8341	0.4737	0.8394	0.4765
0.7994	0.4737	0.8394	0.4765
0.8688	0.4737	0.8292	0.4765
0.8341	0.4737	0.836	0.4765
0.7647	0.4737	0.8326	0.4765
0.7994	0.4737	0.8326	0.4765
0.9035	0.4737	0.836	0.4765
0.9035	0.4737	0.836	0.4765
0.8341	0.4737	0.8394	0.4765

Figure 4.52: T_NOA patience 6

valid_lo	valid_ac	train_lo	train_ac
0.6847	0.5263	0.6934	0.4765
0.5738	0.6842	0.6669	0.5647
0.5567	0.9211	0.6071	0.7471
0.3758	0.9211	0.5199	0.8294
0.4105	0.9211	0.4849	0.8294
0.3549	0.9474	0.4758	0.8353
0.3758	0.9211	0.4814	0.8294
0.4105	0.9211	0.4826	0.8353
0.4105	0.9211	0.486	0.8353
0.4105	0.9211	0.4826	0.8353
0.3758	0.9211	0.486	0.8353
0.3758	0.9211	0.4792	0.8353
0.3758	0.9211	0.4758	0.8353
0.3758	0.9211	0.4758	0.8353
0.3758	0.9211	0.4724	0.8353
0.3758	0.9211	0.4758	0.8353
0.4105	0.9211	0.4758	0.8353
0.4105	0.9211	0.4792	0.8353
0.3758	0.9211	0.4792	0.8353
0.3758	0.9211	0.4724	0.8353
0.3758	0.9211	0.4792	0.8353
0.3758	0.9211	0.4724	0.8353
0.3758	0.9211	0.4792	0.8353
0.4105	0.9211	0.486	0.8353
0.3758	0.9211	0.4758	0.8353
0.4452	0.9211	0.4758	0.8353
0.3758	0.9211	0.4758	0.8353
0.4105	0.9211	0.4792	0.8353
0.3758	0.9211	0.4792	0.8353
0.3758	0.9211	0.4724	0.8353

Figure 4.53: T_NOA patience 7

Here the models with *patience* 6 and 7 are doing much better on the validation set than on the training set: the overall performances are reasonably well but the reasons behind this issue could be the same as before (Dropout layers, small dataset).

T_A results

valid_lo	valid_ac	train_lo	train_ac
0.5008	0.825	0.5285	0.7564
0.8549	0.475	0.76	0.5372
0.8549	0.475	0.832	0.4808
0.8549	0.475	0.8332	0.4808
0.8341	0.475	0.8324	0.4808
0.8549	0.475	0.8328	0.4808
0.8758	0.475	0.8337	0.4808
0.8549	0.475	0.8328	0.4808
0.8549	0.475	0.8332	0.4808
0.8341	0.475	0.8328	0.4808
0.8133	0.475	0.8328	0.4808
0.8549	0.475	0.832	0.4808
0.8133	0.475	0.8328	0.4808
0.8341	0.475	0.8337	0.4808
0.8341	0.475	0.832	0.4808
0.8341	0.475	0.8324	0.4808
0.8341	0.475	0.832	0.4808
0.8133	0.475	0.8332	0.4808
0.8549	0.475	0.8324	0.4808
0.8549	0.475	0.8337	0.4808
0.8341	0.475	0.8328	0.4808
0.8549	0.475	0.8328	0.4808
0.8341	0.475	0.8332	0.4808
0.8758	0.475	0.8341	0.4808
0.8341	0.475	0.8332	0.4808
0.8966	0.475	0.8332	0.4808
0.8133	0.475	0.8324	0.4808
0.9174	0.475	0.8328	0.4808
0.8341	0.475	0.8328	0.4808
0.8341	0.475	0.8315	0.4808

Figure 4.54: T_A patience 4

valid_lo	valid_ac	train_lo	train_ac
0.5216	0.825	0.5461	0.7769
0.4799	0.825	0.4668	0.8474
0.7924	0.525	0.6326	0.6808
0.5633	0.775	0.755	0.5577
0.5008	0.825	0.5091	0.8038
0.5008	0.825	0.4678	0.8449
0.4591	0.825	0.4678	0.8423
0.4799	0.825	0.4648	0.8474
0.4591	0.825	0.4703	0.841
0.5008	0.825	0.4678	0.8436
0.5008	0.825	0.4673	0.8449
0.4591	0.825	0.4664	0.8449
0.5008	0.825	0.4686	0.8423
0.4799	0.825	0.4717	0.8372
0.5008	0.825	0.4669	0.8449
0.5008	0.825	0.4725	0.8385
0.4799	0.825	0.4708	0.841
0.5008	0.825	0.4703	0.841
0.5008	0.825	0.4661	0.8449
0.4591	0.825	0.4694	0.8436
0.4799	0.825	0.4652	0.8449
0.4591	0.825	0.4683	0.8423
0.5008	0.825	0.4688	0.841
0.4799	0.825	0.4686	0.8423
0.5216	0.825	0.4678	0.8423
0.4799	0.825	0.4678	0.8436
0.4799	0.825	0.4652	0.8449
0.5008	0.825	0.4667	0.8423
0.4799	0.825	0.4736	0.8397
0.5216	0.825	0.4677	0.8449

Figure 4.55: T_A patience 5

valid_lo	valid_ac	train_lo	train_ac
0.4799	0.825	0.526	0.791
0.8133	0.475	0.4881	0.8256
0.7924	0.475	0.832	0.4808
0.8133	0.475	0.8328	0.4808
0.8549	0.475	0.8337	0.4808
0.8549	0.475	0.8328	0.4808
0.8549	0.475	0.8332	0.4808
0.8341	0.475	0.8324	0.4808
0.8549	0.475	0.8324	0.4808
0.7924	0.475	0.8328	0.4808
0.8341	0.475	0.8328	0.4808
0.8341	0.475	0.8341	0.4808
0.7924	0.475	0.8337	0.4808
0.8341	0.475	0.8337	0.4808
0.8549	0.475	0.8307	0.4808
0.8133	0.475	0.8328	0.4808
0.8341	0.475	0.8311	0.4808
0.8549	0.475	0.8328	0.4808
0.8549	0.475	0.832	0.4808
0.8549	0.475	0.8328	0.4808
0.8341	0.475	0.8328	0.4808
0.7924	0.475	0.832	0.4808
0.8758	0.475	0.832	0.4808
0.8133	0.475	0.832	0.4808
0.8133	0.475	0.832	0.4808
0.8341	0.475	0.8315	0.4808
0.8341	0.475	0.8324	0.4808
0.8549	0.475	0.8311	0.4808
0.8341	0.475	0.8332	0.4808
0.8341	0.475	0.8315	0.4808

Figure 4.56: T_A patience 6

valid_lo	valid_ac	train_lo	train_ac
0.6116	0.775	0.5505	0.7462
0.6931	0.375	0.746	0.5218
0.6931	0.375	0.6931	0.4628
0.6931	0.4	0.6931	0.4795
0.6931	0.425	0.6931	0.4795
0.6931	0.4	0.6931	0.4769
0.6931	0.4	0.6931	0.4795
0.6931	0.425	0.6931	0.4795
0.6931	0.425	0.6931	0.4782
0.6931	0.425	0.6931	0.4808
0.6931	0.35	0.6931	0.4821
0.6931	0.475	0.6931	0.4782
0.6931	0.35	0.6931	0.4795
0.6931	0.35	0.6931	0.4795
0.6931	0.4	0.6931	0.4808
0.6931	0.4	0.6931	0.4769
0.6931	0.4	0.6931	0.4795
0.6931	0.35	0.6931	0.4833
0.6931	0.425	0.6931	0.4795
0.6931	0.425	0.6931	0.4821
0.6931	0.425	0.6931	0.4808
0.6931	0.4	0.6931	0.4808
0.6931	0.35	0.6931	0.4795
0.6931	0.375	0.6931	0.4795
0.6931	0.375	0.6931	0.4808
0.6931	0.425	0.6931	0.4782
0.6931	0.4	0.6931	0.4808
0.6931	0.425	0.6931	0.4782
0.6931	0.4	0.6931	0.4795
0.6931	0.45	0.6931	0.4795

Figure 4.57: T_A patience 7

From the table related to the trial with *patience 5*, it appears that the model has achieved a relatively high level of validation accuracy, consistently achieving a value of 0.825 across many epochs. The validation loss fluctuates somewhat, ranging from a low of 0.459 to a high of 0.792. This suggests that the model is able to make accurate predictions resulting in a overall good performance, achieved after a prolonged training, with respect to the other trials of this approach, whose best performances are reached in the first epochs: they are not reliable at all.

Chapter 5

Final discussions

At the end of these runs, mainly referring to the last, i.e. the third, it is possible to note that except for the strange cases where the highest performances are reached in the first epochs, the other results are quite stable because taken with a loss that is diminished constantly throughout the epoch. Apart from rare cases in which the accuracy on training is stuck, in most cases the performances are reasonable: below, is shown a table with the accuracy obtained by feeding the test set to each of the models of the third approach, with corresponding epoch.

	patience = 4		patience = 5		patience = 6		patience = 7	
	EP	TEST ACC	EP	TEST ACC	EP	TEST ACC	EP	TEST ACC
NOT_NOA	3	0.675675676	15	0.648648649	3	0.72972973	3	0.702702703
NOT_A	19	0.837837838	2	0.837837838	3	0.837837838		
T_NOA	2	0.891891892	4	0.891891892	8	0.891891892	6	0.891891892
T_A	1	0.810810811	8	0.810810811	1	0.810810811	1	0.810810811

Figure 5.1: Test accuracy on 3rd run models

It is possible to see that the best accuracy values have been obtained with the third approach, i.e. *0.89* with **T_NOA**, that takes into account the textual information to classify the TV banners but no augmentation techniques have been applied to the training set. It can be considered a good result taking into account the use case and the small amount of real data.

The fact that the best approach is **T_NOA** can be seen as an important indicator: the textual information is important for the use case considered in this thesis project. In fact also the fourth approach, i.e. **T_A**, has good accuracy values, even if they are slightly lower than the second one

(**NOT_A**), which however had a very bad performance with *patience* 7, and for this reason no test accuracy has been reported.

About the first approach, i.e. **NOT_NOA**, it has the worst overall performances: it seems that based on the dataset and on its variance, the model, as it is made, needs at least the textual information or the application of augmentation techniques to perform sufficiently. Without both of them, it has poor performances.

About the fact that the accuracy is higher on the validation with respect to the training set, this may be caused by heavy Dropout layers utilized when building the neural network: when training, due to disabling neurons (i.e. a percentage of the features are set to zero), some of the information about each sample is lost, and the subsequent layers attempt to construct predictions basing on incomplete representations. However, during validation and testing all of the units are available, so the network has its full computational power and might perform better than in training, resulting in a more robust performance.

Another problem could be related to the dataset: smaller datasets have smaller variance. The original dataset was in fact composed by 167 samples, with 128 Internal and only 39 External. It was unbalanced, and the application of transforming operations, while balancing the class distribution, may have decreased the variance even more. In general an higher number of samples could lead to overall better performances, especially, an higher number of *original* samples, since after the transforming operations the External class has more augmented samples than real ones.

5.1 Future developments

A new design of the model without or with less dropout layers could be an approach to take into account for the future developments of this project.

Another direction could be the application of different data augmentation techniques, applied in a variety of ways to the data, in order to obtain even more transformed images, with the hope of achieving a much higher final result.

Given the amount of data available and their distribution, another attempt could be the application of a cross-validation technique: splitting the dataset into several folds, training the model on some of these folds, while testing it on the remaining ones and repeating the process multiple times could lead to a more accurate estimate of how well the model will generalize to new, unseen data. This is because it allows to assess the model's performance on multiple different subsets of the data, rather than just one.

Bibliography

[1] John McCarthy. “What is Artificial Intelligence”. In: (Nov. 2007).

[2] Alan M. Turing. “Computing Machinery and Intelligence”. In: (Jan. 1950).

[3] Iqbal H.Sarker. “Machine Learning: Algorithms, Real-World Applications and Research Directions”. In: *SN Computer Science* (Jan. 2021).

[4] Ani1 K. Jain, Jianchang Mao, and K.M. Mohiuddin. “Artificial neural networks: a tutorial”. In: (Mar. 1996).

[5] Oludare Isaac Abiodun, Aman Jantan, Abiodun Esther Omolara, Kemi Victoria Dada, and Abubakar Malah Umar et al. “Comprehensive Review of Artificial Neural Network Applications to Pattern Recognition”. In: (Sept. 2019).

[6] Oludare Isaac Abiodun, Aman Jantan, Abiodun Esther Omolara, Kemi Victoria Dada, and Abubakar Malah Umar et al. “An Introduction to Convolutional Neural Networks”. In: (Dec. 2015).

[7] Zewen Li, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. “A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects”. In: (Dec. 2022).

[8] Jin Wang, Jinhui Tang, and Jiebo Luo. “Multimodal Attention with Image Text Spatial Relationship for OCR-Based Image Captioning”. In: (Oct. 2020).

[9] Nils Reimers and Iryna Gurevych. “Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks”. In: (Aug. 2019).

[10] Gabriele Siegert. “Self promotion: Pole position in Media Brand Management”. In: (Jan. 2008).

[11] Nicolas Audebert, Catherine Herold, Kuider Slimani, and Cedric Vidal. “Multimodal deep networks for text and image-based document classification”. In: (July 2019).

[12] Shardul Suryawanshi, Bharati Raja Chakravarthi, Mihael Arcan, and Paul Buitelaar. “Multimodal Meme Dataset (MultiOFF) for Identifying Offensive Content in Image and Text”. In: (May 2020).

[13] Shardul Suryawanshi, Bharati Raja Chakravarthi, Mihael Arcan, and Paul Buitelaar. “Multimodal Meme Dataset (MultiOFF) for Identifying Offensive Content in Image and Text”. In: (May 2020).

[14] Anastasia Giachanou, Guobiao Zhang, and Paolo Rosso. “Multimodal Spam Classification Using Deep Learning Techniques”. In: (Oct. 2020).

[15] Raul Gomez, Jaume Gibert, Lluis Gomez, and Dimosthenis Karatzas. “Exploring Hate Speech Detection in Multimodal Publications”. In: (Jan. 2020).

[16] https://www.python.org. “Python docs”. In: ().

[17] https://colab.research.google.com. “Google Colab”. In: ().

[18] Keras implementation of Convolutional Recurrent Neural Network. “https://github.com/janzd/CRNN”. In: (Apr. 2021).

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: (May 2019).

[20] Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, and Lukasz Kaiser. “Attention Is All You Need”. In: *Conference on Neural Information processing System* (Dec. 2017).

[21] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Holger Schwenk Fethi Bougares, and Yoshua Bengio. “Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation”. In: *Conference on Empirical Methods in Natural Language Processing (EMNLP)* (Oct. 2014).

[22] Jean-Baptiste Cordonnier, Andreas Loukas, and Martin Jaggi. “Multi-Head Attention: Collaborate Instead of Concatenate”. In: (May 2021).

[23] sentence-transformers/paraphrase-multilingual-mpnet-base v2. “https://huggingface.co/sentence-transformers/paraphrase-multilingual-mpnet-base-v2”. In: ().

[24] Nils Reimers and Iryna Gurevych. “Making Monolingual Sentence Embeddings Multilingual using Knowledge Distillation”. In: (Oct. 2020).

[25] Mikel Artetxe and Holger Schwenk. “Massively Multilingual Sentence Embeddings for Zero-Shot Cross-Lingual Transfer and Beyond”. In: (Sept. 2011).

[26] PyTorch. “https://pytorch.org”. In: ().

[27] Pillow: the friendly Python Imaging Library fork. “https://pillow.readthedocs.io/en/stable/”. In: ().

[28] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. “Improving neural networks by preventing co-adaptation of feature detectors”. In: (July 2012).

[29] Nikolay Kyurkchiev and Svetoslav Markov. “Sigmoid functions: some approximation and modelling aspects”. In: (July 2015).

[30] A.Usha Ruby, Prasannavenkatesan Theerthagiri, I.Jeena Jacob, and Y.Vamsidhar. “Binary cross entropy with deep learning technique for Image classification”. In: (July 2020).

[31] Rob G.J. Wijnhoven and Peter H.N de With. “Fast Training of Object Detection using Stochastic Gradient Descent”. In: (May 2010).

[32] Seda Postalcioglu. “Performance Analysis of Different Optimizers for Deep Learning-Based Image Recognition”. In: (June 2019).