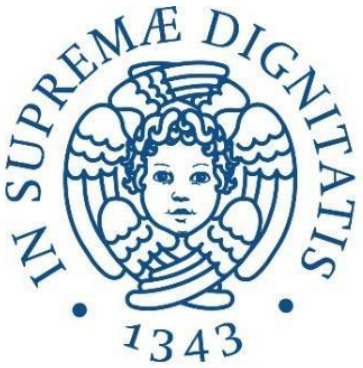


UNIVERSITÀ DI PISA

Master Degree in Artificial Intelligence and Data Engineering

Business and Project Management

Text Classifier for Bot Detection

Group Project for Business and Project Management

Academic year 2023-2024

Andrea Sottile, Gianluca Antonio Cometa

Github repository:

https://github.com/gianlucacometa/TextClassifierBotDetection

CONTENTS

Summary

Introduction

This project's main idea is to make a tool able to classify a text, trying to label it as written by a human or generated by an AI.
The main purpose of nowadays AIs is to imitate human behavior, and it is progressively harder to tell the difference with the development of new technologies.
Inspired by the concept of adversarial networks, we are trying to evaluate how hard it is for neural networks to recognize the work of their generative siblings.
Generative Adversarial Networks (GAN) are mostly oriented on trying to imitate a human product and make it believable for other humans, and they have already reached an impressive level in several art fields. About text generation and speech imitation, GAN are able to mimic humans with differences that get smaller day by day.
Their work is already under careful observation, in particular about the possibility to recognize the source: we are already seeing the issues of fake reviews, click-farms, biased surveys, and counterfeited feedback.
Therefore, we want to tackle the issue of artificially generated text detection, hoping to make a classifier able to detect the flaws of a state-of-the-art chat-bot.

Related Works

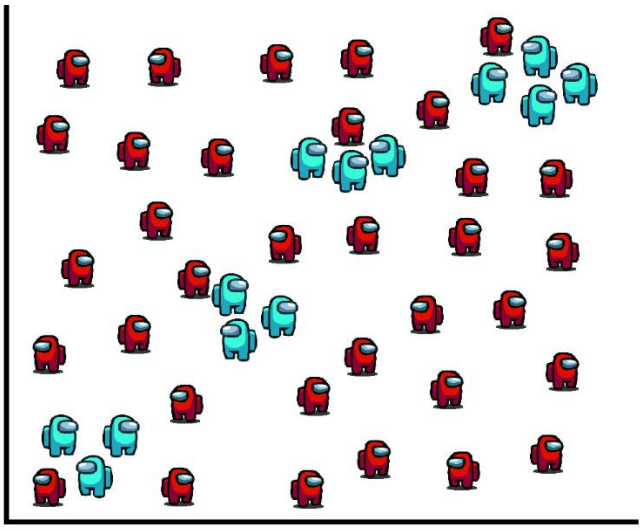
Nowadays, Chat-Bots are a common element of our daily life, in particular for the younger generations. There are plenty of generative Neural Networks able to generate text and they are usually exploited to mimic human behavior.
For example, several firms are substituting human call centers with AI Assistants which make their service more available, affordable and efficient.
State-of-art AIs are capable of providing this service with very high accuracy and almost no mistakes, at least for easy and well-documented topics. Sometimes, it's hard to even notice if the answer comes from a human or a bot.
When you want to tell the difference you usually look for specific tells like repetitions, mistakes or too broad information. Otherwise, you could try to ask tricky questions in order to mislead the chat on a circular topic and see if there are any fluctuations.
Again, modern Bots are able to fool humans in several ways: they are able to change their minds, they can add random delays between messages, and they might even be able to notice or reproduce emotions.
A modern and common threat in our society is the review bombing, i.e. an Internet phenomenon where a large number of people or a few people with multiple accounts post negative user reviews online in an attempt to harm the sales or popularity of a product, a service, or a business[1].
This problem is magnified by using AI-generated texts, which are way cheaper and faster to make than man-made ones. A great effort is required by a human team to check a massive number of reviews and eventually delete fakes.
Another problem is related to the so-called fact-checking, i.e. the process of verifying the factual accuracy of questioned reporting and statements.
There are a lot of studies about this topic and we decided to address this issue starting from one scientific paper titled "The Impact and Opportunities of Generative AI in Fact-Checking"[2].
This document explores the impact and opportunities of using generative AI in fact-checking, investigating how this technology can transform organizations involved in information verification. Through interviews with 38 participants from 29 fact-checking organizations across six continents, the study highlights both potential benefits and challenges arising from the adoption of generative AI.
Specifically, we see that the main challenges lie in three areas:

- Technological, caused by the lack of transparency and unpredictable behavior of AI.
- Organizational, because of shortage of resources and specific technical skills.
- Environmental, due to uncertain policies and evolving regulations.

This topic also affect some social and economic aspects, integrating generative AI in fact-checking can not only improve efficiency and speed in responding to misinformation but also has significant implications for society and the economy.
The ability to quickly identify and counter misinformation contributes to the health of the information ecosystem, reducing the negative impact of fake news on society. Additionally,

[1] Wikipedia: Review bomb - Wikipedia
[2] Arxiv paper on fact checking and generative AI: https://arxiv.org/pdf/2405.15985

the technology can support the creation of more reliable information, strengthening public trust in the media.
An effective solution not only supports fact-checking organizations in their work but also helps mitigate the risks associated with misinformation, with significant benefits for both economic institutions and society as a whole.
In this context, a detection solution emerges as an essential tool to ensure that information remains truthful and reliable, thus contributing to a healthy and secure information ecosystem.
Of course, one of the ways to address some of these issues could be speeding up the classification of human/bot written texts. The use of AI could be helpful for doing so: like they can deceive the human perception they can also improve it.
Therefore, the goal of our project is to test the ability of detectors to spot text-generative AIs.
According to the state-of-art scenario, which side is more developed? Can one AI detect other AIs better than humans?



Dataset Description

Since we need a large amount of text samples, we are proposing to use the IMDb Movie Reviews Dataset[3] available online.
This dataset is made of 50000 unique movie reviews, with different sizes.
There might be more than one review for each film. The dataset labels each review based on the mark given to the film by the user (positive, negative or neutral).
This dataset has been already used for a massive number of tasks, mostly oriented to sentiment analysis.
Since our goal is completely different, we are ignoring the labels, and only take the reviews as human-written texts.
The other half of the dataset is going to be made with AI-generated reviews.
We are going to generate these reviews by making an automated CoLab script sending requests to the Python interface of the commonly used ChatGPT, specifically the "gpt-3.5-turbo" version.
From our first tests we noticed that if we submit too simple prompts the AI answers with reviews that are very similar to each other. So, we used a different approach.
Our idea is to create a prompt template to ask for a fake review based on a made-up film title. Additionally, we variate the prompt asking to emulate a sentiment of the person that "wrote" the review (picked from a list).
The code below shows the implementation of this idea. As you notice, this is based on:

- Prompt template, that have 3 variables for the definition of the minimum number of words, the minimum and the maximum numbers of char in the review.
- Prompt variable, that contains the title of the film.
- Prompt sentiment, that use one element from Sentiment variable to define the mood of the writer of the review.

```
[ ]  MIN_SIZE_WORD = 100
     MIN_SIZE_CHAR = 500
     MAX_SIZE_CHAR = 2000

     PROMPT_TEMPLATE = "Generate for me a review about a fictional movie. The review should be at least "+str(MIN_SIZE_WORD*2)+" long, and with a number of characters between "+str(MIN_SIZE_CHAR)+" and "+str(MAX_SIZE_CHAR)+"."
     PROMPT_VARIABLE = "The title of the film is this: "
     PROMPT_SENTIMENT = "Pretend the writer felt this emotion while watching it: "
     SENTIMENT_VARIABLE = ["happy", "angry", "enthusiastic", "bored", "sad"]
```

```
# take a request, send to chatgpt, and return the full response
def send_prompt(t):
  # ATTENZIONE: FAR GIRARE QUESTA FUNZIONE TOGLIE I SOLDI A GIANLUCA
  data = {
      "model": MODEL_CHATGPT_GENERATOR,
      "messages": [
          {"role": "user", "content": t}
      ]
  }
  response = requests.post(URL_API_CHATGPT, headers=REQUEST_HEADER_CHATGPT, json=data)
  response_data = response.json()
  completions = response_data["choices"][0]["message"]["content"]
  return completions

# generate a fake review
def make_review(prompt):
  #prompt = produce_prompt(key, sentiment)
  response = send_prompt(prompt)

  return response

# produce a prompt with a small variance from the template
def produce_prompt(key, sentiment):
  return PROMPT_TEMPLATE+PROMPT_VARIABLE+key+". "+PROMPT_SENTIMENT+sentiment
```

So, our script takes a list of film titles (made-up also by AI) and a list of sentiments (5 in our case) and creates a review for each possible title-sentiment combination.

[3] Dataset link: https://paperswithcode.com/dataset/imdb-movie-reviews

Thanks to this approach, we have quickly generated a big number of fake reviews with consistent differences between each other.
By manually inspecting these reviews we can find a few similarities but if you take each sample it's very challenging to notice them because the general structure is very human-like.
For the first part of our project, we decided to use a small dataset to build a first crude model and we expanded it later when we were ready for the refined model with a more stabilized preprocessing.

Therefore, our final dataset is composed by:

- 12500 human generated reviews taken from the IMDb Movie Reviews Dataset;
- 7800 bot-generated reviews which:
    - 16 manually requested to ChatGPT without the involvement of the script;
    - 100 of these are created with made-up film names;
    - 588 of these are created with made-up film names and the emotional variable;
    - 15 reviews made with ChatGPT.4o version (we dropped the idea of using this generative engine because of high billing prices, and not enough differences);
    - 7081 reviews made by making a prompt based on real film names, with different prompt and set of emotional variables (shown in the figure below).

```
MIN_SIZE_WORD = 100
MIN_SIZE_CHAR = 500
MAX_SIZE_CHAR = 1500

# name template to save files
NAME_TAG = "review_23_05_GPT_3.5"

PROMPT_TEMPLATE = """Generate for me a review about a fictional movie. The review should contain a number of characters between "+str(MIN_SIZE_CHAR)+" and "+str(MAX_SIZE_CHAR)+", but you can make it as long as you want in this range.
                  The output should be ASCII text only, with no special characters, no carriage return, no text formatting nor blank lines."""
PROMPT_VARIABLE = "The title of the film is this: "
PROMPT_SENTIMENT = "Pretend the writer was "

SENTIMENT_VARIABLE = ["eager to see the sequel", "annoyed by not enough jokes", "happy", "distracted because it was boring",
                      "a young girl watching the film with her family", "an art critique", "a teenager"]
```

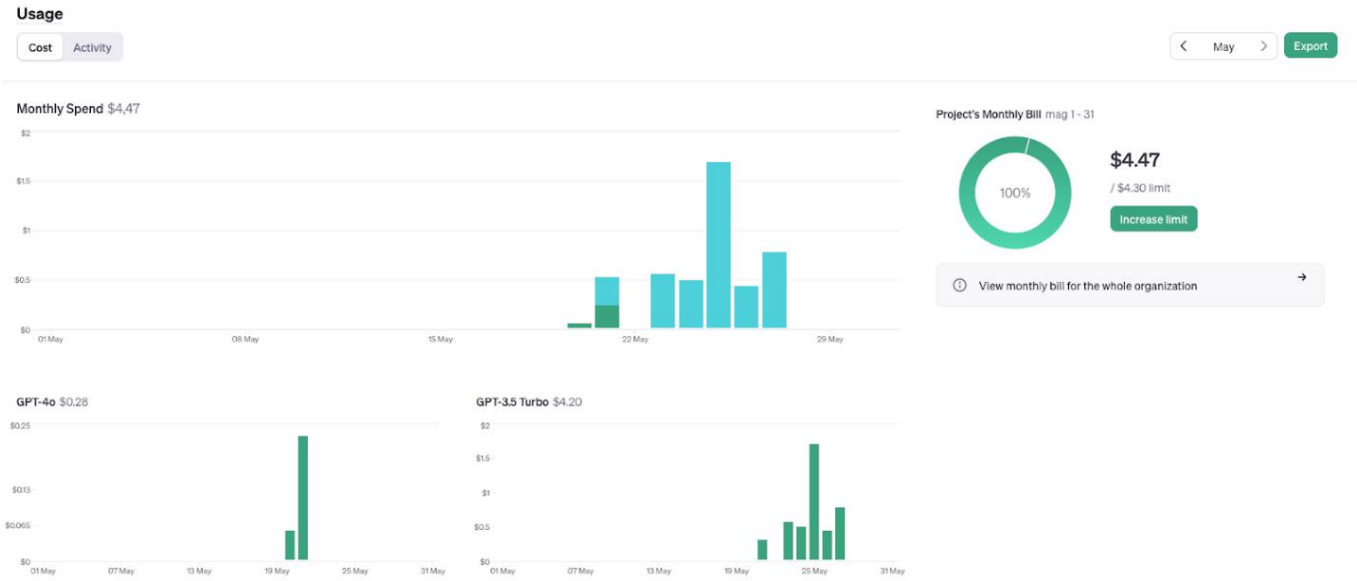
Despite changing the prompt every few hundreds of reviews, using the variable approach, and changing the film names, we are still facing the issue of creativity limits. Each single review is completely realistic and similar to man-made ones, but if you check a lot of them, you start to notice a few repetitions. This is a limit we are unable to overcome, without looking for a new dataset of fake reviews classified by humans. Anyway, we widened the number of reviews to make this issue less frequent.
This topic is also already well known for generative AIs and, in particular, we can cite some of the most important biases and limitation already documented[4]:

- Format Bias: ChatGPT's training data may contain a format bias, as it is primarily composed of text-based content from the internet. This can result in the model being less adept at generating content that reflects other forms of communication, such as spoken language or non-verbal cues.
- Limited Creativity: Although ChatGPT can generate content that appears creative, its creativity is ultimately limited by the patterns and associations it has learned from its training data. This can result in content that is derivative or lacks the novelty and originality found in human-generated creative works.

[4] we link as reference a couple of articles since it's outside of the scope of our project:
- https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9939079/
- https://www.sciencedirect.com/science/article/pii/S266734522300024X

- Overgeneralization: ChatGPT may sometimes overgeneralize when generating content, leading to responses that lack nuance or oversimplify complex topics. This can result in content that appears plausible on the surface but fails to accurately address the subtleties of a given subject.

Generating reviews by Python interface is a premium feature that requires a key generated by an account with a live subscription.
Each execution of the generative function API bills money to the account it's tied to.
Initially (24-04-24) we had 704 bot-generated reviews (100), for a total billing price of 0.38$.
We increased the number of bot-generated reviews in a later stage of the project and we reached a total of 7800, for a total billing price of 4.47$.



In order to avoid an unbalanced dataset, we are going to randomly select a smaller amount of the man-made reviews, since 50000 is likely to be an unnecessarily big number.
We are going to leave the dataset slightly unbalanced, with more human reviews, because some of them might be too short and it's going to be discarded from the training set.

Methods and experiments

Our application design is going to be similar to the various examples of neural networks presented in the course laboratories.
Since those focused on computer vision and image elaboration, we are going to include some text-based processing methods learnt in other courses.
Our experiments will be conducted in two distinct phases:

- the first phase, we will be working with a reduced dataset in order to analyze the behavior of our networks on a smaller amount of data.
  More specifically, this dataset will consist of 2204 number of elements, distributed as 1500 for the "Human reviews" class and 704 for the "Bot reviews" class;
- Following the completion of the first phase, we will transition to the second phase. In this phase, we will be utilizing a larger dataset for our analyses. This will enable us to test the robustness of our model, its scalability, and its performance under more complex and realistic conditions.

Data Preparation

Human and Bot reviews are going to be saved in separate folders in Google Drive.
The most common and most effective approach to deal with text-based inputs is tokenization, which solves most of our preprocessing needs: padding to a standard length, removing stop words, and loading the data into memory in a tensor.
We use this standard approach for the pretrained models.
Since we noticed their excellent performance, we decided to try something different in our classifier made from scratch. Instead of using the Tokenizer+Classification architecture we want to design a single model able to perform the same task without the preprocessing phase.
So, our ideal model is going to accept input texts as raw as possible.
The only "data preparation" step that we kept is the conversion of the letters of the Reviews to numbers (Integer values).
More specifically, we convert each char in their equivalent numerical representation in ASCII code.

```
# CONVERSION CHAR TO INT

def convert_char2int_array(matrix,raw_data,row_index):
  col_index = 0
  for c in raw_data[0]:
    # truncation happens for input elements over the desired size
    if(col_index >= desired_size_char):
      break
    matrix[row_index][col_index] = ord(c)
    col_index += 1
```

Pre-trained Models

Our project is going to have two neural networks:

- the tokenizer described above;
- the classifier.

Both of these types of NN are widely available online, and (as project specifics requires) we are going to pick two pre-trained models to be used and compared.

BERT

Google's BERT[5] (or "Bidirectional Encoder Representations from Transformers") is a family of language models with several different implementations.
Primed in 2018, it's massively employed in natural language processing and sentiment analysis.
One of the key innovations of BERT is its bidirectional nature. Traditional language models are either trained to predict the next word in a sentence (right-context) or the previous word (left-context). BERT, however, is trained to predict a word based on the context in both directions, which gives it a deeper understanding of the meaning of a word in a sentence.
BERT's architecture is based on three main components:

- **Preprocessing**: BERT requires a preprocessing step to transform raw text into a format that the model can understand. This involves normalizing the text, splitting it into tokens, and mapping these tokens to their index numbers in BERT's vocabulary.
- **Tokenizer**: The tokenizer is a critical component of BERT's preprocessing. It breaks down the text into tokens, which can be words or subwords. BERT uses a WordPiece tokenizer that can efficiently handle a large vocabulary of words and subwords.
- **Encoder**: After tokenization, BERT uses an encoder to convert the tokens into embeddings. These embeddings are high-dimensional vectors that represent the tokens in a way that captures their meaning and context within the sentence.

BERT's ability to understand the context of words in a sentence is what makes it so powerful for natural language processing tasks. The combination of preprocessing, tokenization, and encoding allows BERT to capture the nuances of language and perform tasks like sentiment analysis, question answering, and language translation with high accuracy.
BERT has been pre-trained on a large corpus of text and can be fine-tuned with just one additional output layer to create state-of-the-art models for a wide range of tasks, such as

[5] A link for more details: https://en.wikipedia.org/wiki/BERT_(language_model)

question answering and language inference, without substantial task-specific architecture modifications.

It's important to note that while BERT's bidirectionality allows it to understand the context of a word in relation to all the other words in a sentence, it can lead to increased training times compared to other models like GPT-2 (described in the next paragraph).

Usage of the Pre-Trained model in our Project

```
Model: "model"
__________________________________________________________________________________________________
 Layer (type)                Output Shape              Param #     Connected to
==================================================================================================
 text (InputLayer)           [(None,)]                 0           []

 preprocessing (KerasLayer)  {'input_mask': (Non       0           ['text[0][0]']
                             e, 128),
                              'input_word_ids':
                             (None, 128),
                              'input_type_ids':
                             (None, 128)}

 BERT_encoder (KerasLayer)   {'encoder_outputs':       28763649    ['preprocessing[0][0]',
                              [(None, 128, 512),                    'preprocessing[0][1]',
                              (None, 128, 512),                     'preprocessing[0][2]']
                              (None, 128, 512),
                              (None, 128, 512)],
                              'default': (None,
                             512),
                              'pooled_output': (
                             None, 512),
                              'sequence_output':
                              (None, 128, 512)}

 dropout (Dropout)           (None, 512)               0           ['BERT_encoder[0][5]']

 classifier (Dense)          (None, 1)                 513         ['dropout[0][0]']

==================================================================================================
Total params: 28,764,162
Trainable params: 28,764,161
Non-trainable params: 1
__________________________________________________________________________________________________
```

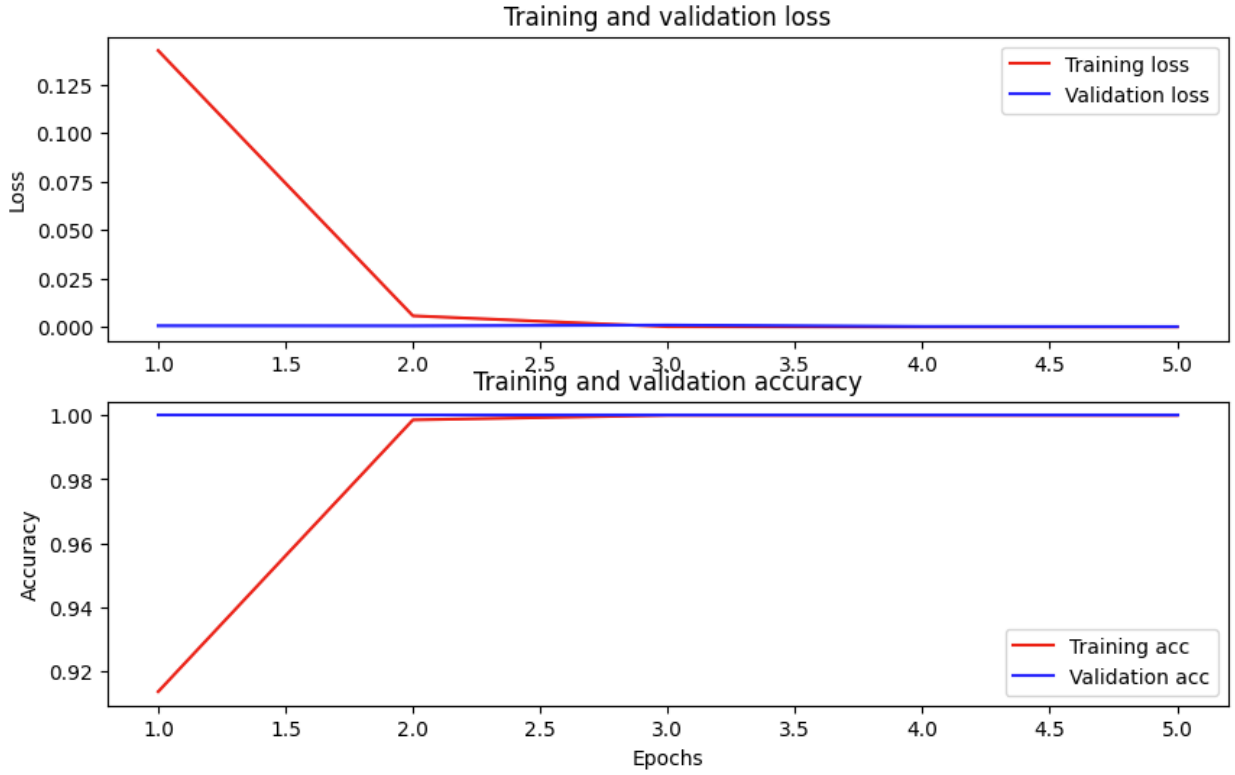
The first step in our model building process involves preprocessing the input data using the tfhub_handle_preprocess function. This function maps our input data into a format that the BERT model can understand.
We then feed this preprocessed data into the BERT model using the hub.KerasLayer(tfhub_handle_encoder, trainable=True, name='BERT_encoder') function, making the BERT model trainable to adapt to our specific task.
The output from the BERT encoder was then passed through a dropout layer with a rate of 0.1 to prevent overfitting.

Subsequently, we added a dense layer with a single neuron that serves as our binary classifier. The activation function for this layer was intentionally left undefined because we used a sigmoid function later on the raw output of the model to convert the logits into probabilities.
This approach allowed us to leverage the pretrained BERT model effectively while also customizing it to suit our specific needs for binary classification.
From the figure below we can see that the results achieved suggest that our binary classifier model is performing exceptionally well.

```
Training model with https://tfhub.dev/tensorflow/small_bert/bert_en_uncased_L-4_H-512_A-8/1
Epoch 1/5
141/141 [==============================] - 166s 1s/step - loss: 0.1426 - binary_accuracy: 0.9135 - val_loss: 6.6327e-04 - val_binary_accuracy: 1.0000
Epoch 2/5
141/141 [==============================] - 160s 1s/step - loss: 0.0058 - binary_accuracy: 0.9986 - val_loss: 6.2130e-04 - val_binary_accuracy: 1.0000
Epoch 3/5
141/141 [==============================] - 158s 1s/step - loss: 1.9361e-04 - binary_accuracy: 1.0000 - val_loss: 0.0010 - val_binary_accuracy: 1.0000
Epoch 4/5
141/141 [==============================] - 159s 1s/step - loss: 8.3947e-05 - binary_accuracy: 1.0000 - val_loss: 3.2508e-04 - val_binary_accuracy: 1.000
0
Epoch 5/5
141/141 [==============================] - 157s 1s/step - loss: 6.9024e-05 - binary_accuracy: 1.0000 - val_loss: 2.3566e-04 - val_binary_accuracy: 1.000
0
```

```python
#EVALUATION
loss, accuracy = classifier_model.evaluate(test_ds)

print(f'Loss: {loss}')
print(f'Accuracy: {accuracy}')
```

```
45/45 [==============================] - 12s 259ms/step - loss: 0.0292 - binary_accuracy: 0.9932
Loss: 0.029212845489382744
Accuracy: 0.9932126402854919
```



For the implementation and running of this architecture is necessary to use a Conda Environment due to a well-known bug on Colab that makes unavailable some models from Tensorflow Hub.

GPT-2

OpenAi's GPT-2[6] (or "Generative Pretrained Transformer 2") is a large transformer-based language model, which is available for free, developed by OpenAI.
Unlike BERT, which is a discriminative model designed for a wide range of natural language processing tasks, GPT-2 is more a generative model.
It's designed to generate human-like text by predicting the likelihood of a word given the previous words used in the text.
GPT-2 is trained on a diverse range of internet text and can generate coherent paragraphs and even whole articles.
It's also capable of translation, question-answering, and summarization tasks, all without task-specific training data, illustrating the power of transfer learning.
Its architecture is specifically a transformer model, which uses attention instead of older recurrence and convolution-based architectures.
The attention mechanisms allow the model to selectively focus on segments of input text it predicts to be the most relevant.
This model allows for greatly increased parallelization and outperforms previous benchmarks for RNN/CNN/LSTM-based models.

Usage of the Pre-Trained model in our Project

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, None)]	0	[]
input_2 (InputLayer)	[(None, None)]	0	[]
tfgpt2_model (TFGPT2Model)	TFBaseModelOutputWithPastAndCrossAttentions(last_hidden_state=(None, None, 768), past_key_values=None, hidden_states=None, attentions=None, cross_attentions=None)	124440576	['input_1[0][0]', 'input_2[0][0]']
tf.math.reduce_mean (TFOpLambda)	(None, 768)	0	['tfgpt2_model[0][0]']
dropout_37 (Dropout)	(None, 768)	0	['tf.math.reduce_mean[0][0]']
dense (Dense)	(None, 1)	769	['dropout_37[0][0]']

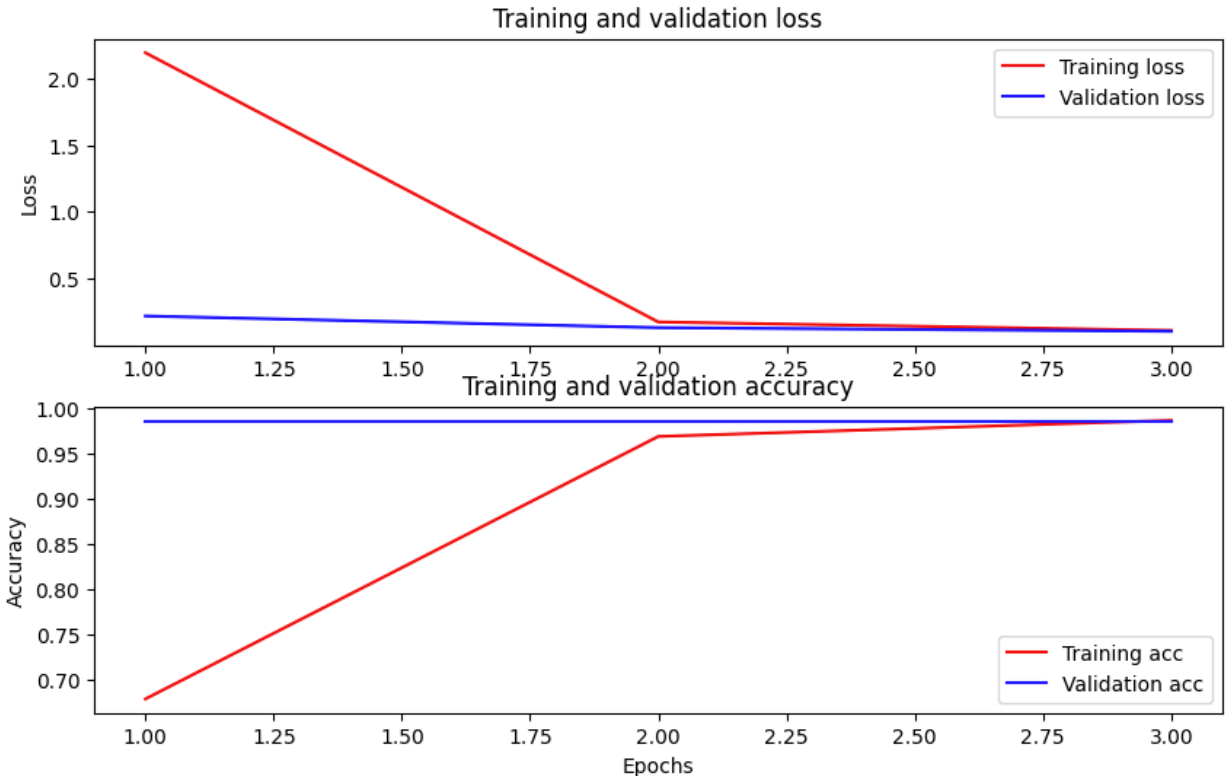
Total params: 124441345 (474.71 MB)
Trainable params: 769 (3.00 KB)
Non-trainable params: 124440576 (474.70 MB)

[6] Here is a link for more details: https://en.wikipedia.org/wiki/GPT-2

In our project, we are not interested in the generative capabilities of GPT, and we use only its attention mask as feature extractor. Then, we add a Dropout and a Dense (size=1) layer to classify the features in our binary output, using the sigmoid as activation since we have two classes.
Just like BERT, this network requires a pre-processing based on tokenization, made with its specific GPT2Tokenizer. The pre-trained network is blocked as untrainable and the fit on the training set is used only to fine-tune the classification layer. This is enough since we reach a very high validation accuracy with less execution time.

```
Epoch 1/3
282/282 [==============================] - 176s 598ms/step - loss: 2.1934 - accuracy: 0.6787 - val_loss: 0.2199 - val_accuracy: 0.9858
Epoch 2/3
282/282 [==============================] - 172s 593ms/step - loss: 0.1751 - accuracy: 0.9688 - val_loss: 0.1326 - val_accuracy: 0.9858
Epoch 3/3
282/282 [==============================] - 169s 599ms/step - loss: 0.1117 - accuracy: 0.9865 - val_loss: 0.1055 - val_accuracy: 0.9858
```

```
14/14 [==============================] - 9s 619ms/step - loss: 0.1004 - accuracy: 0.9932
14/14 [==============================] - 8s 603ms/step
```



CNN From Scratch

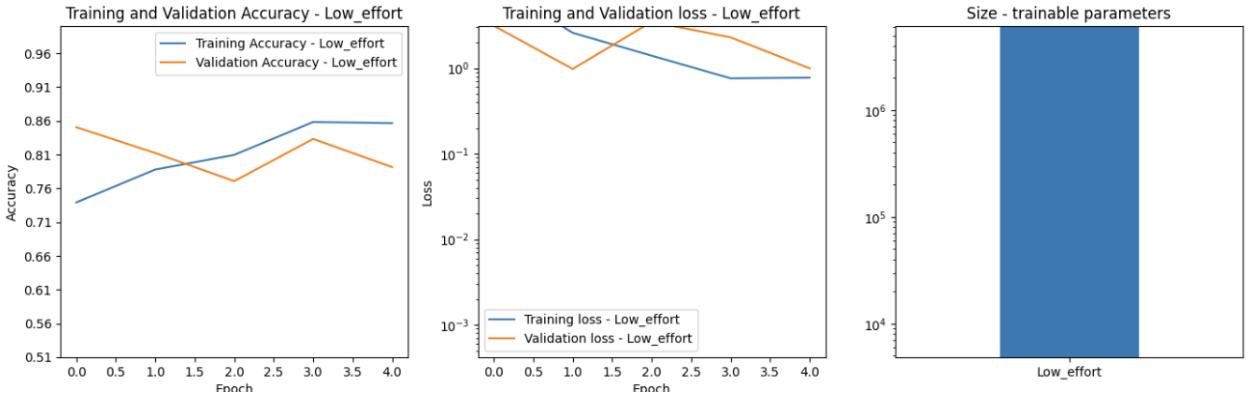
Since the architecture for text-based neural networks is quite commonly divided in two parts (tokenizer, classifier), we want to try merging the two elements in one network, hopefully optimizing it for this task.
The tokenizer networks we have seen are generating a human-readable output, maintaining concepts such as words, repetitions, and stop-words.
By merging the two networks, we might be able to achieve something different.
Before committing to one architecture, we have run several experiments in order to understand the best structure for the final neural network version.
In these experiments we have used a reduced portion of the dataset to have a faster training phase and quick comparisons.
We're not interested in the absolute performance yet and we are just after finding the best model.
We are going to optimize later the model performance on our classification problem.

1° Experiment: Low_effort Network

```
Model: "sequential_58"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_53 (Flatten)        (618, 2000)               0

 dense_93 (Dense)            (618, 2000)               4002000

 dense_94 (Dense)            (618, 1000)               2001000

 dense_95 (Dense)            (618, 100)                100100

 dense_96 (Dense)            (618, 1)                  101

=================================================================
Total params: 6103201 (23.28 MB)
Trainable params: 6103201 (23.28 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
<keras.src.engine.sequential.Sequential at 0x7dd1094ccdc0>
```

In order to have a baseline for other models performance we implement a simple network made of a Flatten layer and 4 Dense layers. This network has almost no design effort and we expect it will be the slowest to train because of its unnecessary big number of parameters.

```
 Training low effort model

Epoch 1/20
58/58 [==============================] - 4s 13ms/step - loss: 9.8772 - binary_accuracy: 0.7394 - val_loss: 3.1322 - val_binary_accuracy: 0.8507
Epoch 2/20
58/58 [==============================] - 0s 7ms/step - loss: 2.5891 - binary_accuracy: 0.7880 - val_loss: 0.9792 - val_binary_accuracy: 0.8125
Epoch 3/20
58/58 [==============================] - 0s 7ms/step - loss: 1.3994 - binary_accuracy: 0.8097 - val_loss: 3.4793 - val_binary_accuracy: 0.7708
Epoch 4/20
58/58 [==============================] - 0s 7ms/step - loss: 0.7608 - binary_accuracy: 0.8584 - val_loss: 2.2916 - val_binary_accuracy: 0.8333
Epoch 5/20
58/58 [==============================] - 0s 7ms/step - loss: 0.7748 - binary_accuracy: 0.8566 - val_loss: 0.9958 - val_binary_accuracy: 0.7917
```



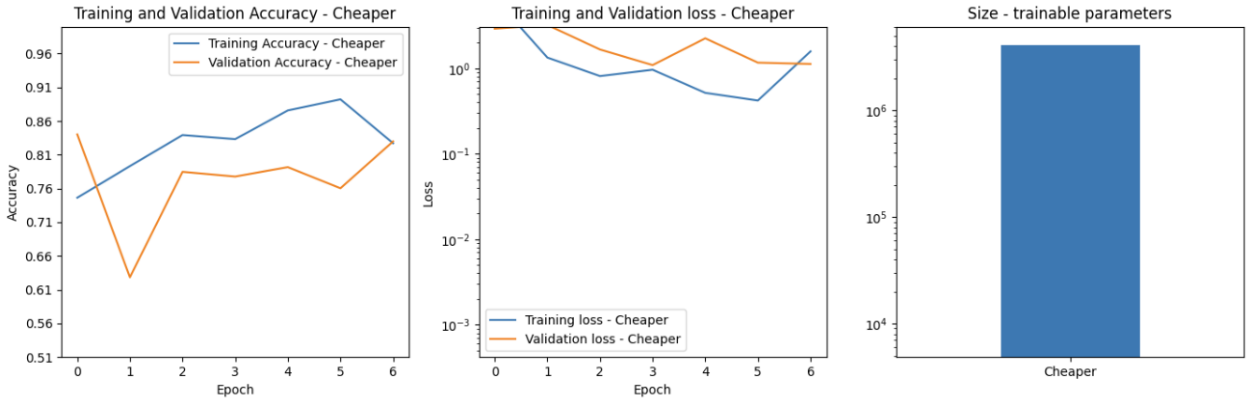
As we can see from the images above, we have a pretty decent performance. It's quite surprising to find such a good accuracy having removed the preprocessing phase. However, we aim to achieve a better performance on smaller networks by improving the design.

2° Experiment: Cheaper Network

```
Model: "sequential_59"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_97 (Dense)            (618, 1500)               3001500

 dense_98 (Dense)            (618, 700)                1050700

 dense_99 (Dense)            (618, 120)                84120

 dense_100 (Dense)           (618, 1)                  121

=================================================================
Total params: 4136441 (15.78 MB)
Trainable params: 4136441 (15.78 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
<keras.src.engine.sequential.Sequential at 0x7dd11af908e0>
```

The second architecture was implemented by removing the Flatten layer and reducing the number of parameters for the remaining layers by 30% (with a total amount of 4 million parameters).
The scope of this experiment is to see how the performance declines with these changes.

```
 Training cheaper model

Epoch 1/20
58/58 [==============================] - 2s 8ms/step - loss: 6.5828 - binary_accuracy: 0.7464 - val_loss: 2.9143 - val_binary_accuracy: 0.8403
Epoch 2/20
58/58 [==============================] - 0s 5ms/step - loss: 1.3387 - binary_accuracy: 0.7932 - val_loss: 3.2195 - val_binary_accuracy: 0.6285
Epoch 3/20
58/58 [==============================] - 0s 5ms/step - loss: 0.8130 - binary_accuracy: 0.8393 - val_loss: 1.6641 - val_binary_accuracy: 0.7847
Epoch 4/20
58/58 [==============================] - 0s 6ms/step - loss: 0.9648 - binary_accuracy: 0.8332 - val_loss: 1.0924 - val_binary_accuracy: 0.7778
Epoch 5/20
58/58 [==============================] - 0s 5ms/step - loss: 0.5170 - binary_accuracy: 0.8758 - val_loss: 2.2479 - val_binary_accuracy: 0.7917
Epoch 6/20
58/58 [==============================] - 0s 6ms/step - loss: 0.4208 - binary_accuracy: 0.8923 - val_loss: 1.1670 - val_binary_accuracy: 0.7604
Epoch 7/20
58/58 [==============================] - 0s 5ms/step - loss: 1.5821 - binary_accuracy: 0.8271 - val_loss: 1.1265 - val_binary_accuracy: 0.8299
```



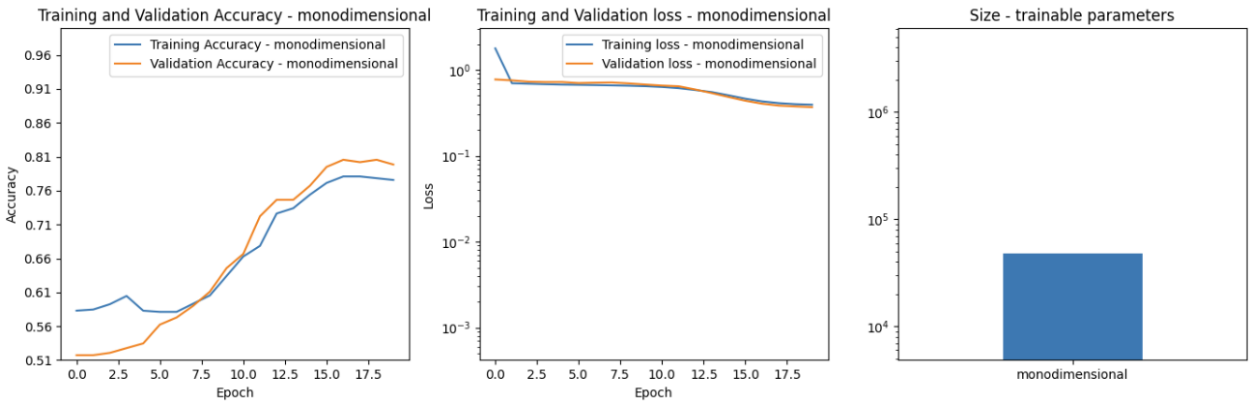
We have seen the loss worsening but the accuracy hasn't changed very much.
Additionally, the Flatten layer has no effect on our network; we could have already guessed it since each input sample is a mono-dimensional array.

3° Experiment: Monodimensional Network

```
Model: "sequential_61"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 reshape_5 (Reshape)         (618, 2000, 1)            0

 conv1d_131 (Conv1D)         (618, 1994, 128)          1024

 max_pooling1d_131 (MaxPool  (618, 664, 128)           0
 ing1D)

 conv1d_132 (Conv1D)         (618, 660, 64)            41024

 max_pooling1d_132 (MaxPool  (618, 330, 64)            0
 ing1D)

 conv1d_133 (Conv1D)         (618, 328, 32)            6176

 max_pooling1d_133 (MaxPool  (618, 164, 32)            0
 ing1D)

 dense_102 (Dense)           (618, 164, 1)             33

 flatten_55 (Flatten)        (618, 164)                0

 dense_103 (Dense)           (618, 1)                  165

=================================================================
Total params: 48422 (189.15 KB)
Trainable params: 48422 (189.15 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
<keras.src.engine.sequential.Sequential at 0x7dd1600bc0a0>
```

For this experiment we try a completely different architecture by introducing Convolutional and Pooling layers.
We know from the laboratory lessons that Conv + Pool pairs are great for pattern recognition and feature extraction and they also have a significantly smaller number of parameters.
So, we want to see the performance now that the network has less connections than the previous Dense models.

```
Training monodimensional model

Epoch 1/20
58/58 [==============================] - 5s 18ms/step - loss: 1.7830 - binary_accuracy: 0.5830 - val_loss: 0.7749 - val_binary_accuracy: 0.5174
Epoch 2/20
58/58 [==============================] - 1s 9ms/step - loss: 0.6996 - binary_accuracy: 0.5847 - val_loss: 0.7558 - val_binary_accuracy: 0.5174
Epoch 3/20
58/58 [==============================] - 1s 9ms/step - loss: 0.6901 - binary_accuracy: 0.5925 - val_loss: 0.7291 - val_binary_accuracy: 0.5208
Epoch 4/20
58/58 [==============================] - 1s 12ms/step - loss: 0.6813 - binary_accuracy: 0.6047 - val_loss: 0.7219 - val_binary_accuracy: 0.5278
Epoch 5/20
58/58 [==============================] - 1s 12ms/step - loss: 0.6742 - binary_accuracy: 0.5830 - val_loss: 0.7223 - val_binary_accuracy: 0.5347
Epoch 6/20
58/58 [==============================] - 1s 12ms/step - loss: 0.6701 - binary_accuracy: 0.5812 - val_loss: 0.7025 - val_binary_accuracy: 0.5625
Epoch 7/20
58/58 [==============================] - 1s 12ms/step - loss: 0.6656 - binary_accuracy: 0.5812 - val_loss: 0.7097 - val_binary_accuracy: 0.5729
Epoch 8/20
58/58 [==============================] - 1s 13ms/step - loss: 0.6602 - binary_accuracy: 0.5934 - val_loss: 0.7133 - val_binary_accuracy: 0.5903
Epoch 9/20
58/58 [==============================] - 1s 10ms/step - loss: 0.6545 - binary_accuracy: 0.6056 - val_loss: 0.6964 - val_binary_accuracy: 0.6111
Epoch 10/20
58/58 [==============================] - 1s 9ms/step - loss: 0.6460 - binary_accuracy: 0.6342 - val_loss: 0.6752 - val_binary_accuracy: 0.6458
Epoch 11/20
58/58 [==============================] - 1s 9ms/step - loss: 0.6331 - binary_accuracy: 0.6629 - val_loss: 0.6567 - val_binary_accuracy: 0.6667
Epoch 12/20
58/58 [==============================] - 1s 9ms/step - loss: 0.6132 - binary_accuracy: 0.6785 - val_loss: 0.6447 - val_binary_accuracy: 0.7222
Epoch 13/20
58/58 [==============================] - 1s 9ms/step - loss: 0.5831 - binary_accuracy: 0.7263 - val_loss: 0.5912 - val_binary_accuracy: 0.7465
Epoch 14/20
58/58 [==============================] - 1s 9ms/step - loss: 0.5513 - binary_accuracy: 0.7341 - val_loss: 0.5365 - val_binary_accuracy: 0.7465
Epoch 15/20
58/58 [==============================] - 1s 9ms/step - loss: 0.5060 - binary_accuracy: 0.7541 - val_loss: 0.4833 - val_binary_accuracy: 0.7674
Epoch 16/20
58/58 [==============================] - 1s 9ms/step - loss: 0.4625 - binary_accuracy: 0.7715 - val_loss: 0.4373 - val_binary_accuracy: 0.7951
Epoch 17/20
58/58 [==============================] - 1s 9ms/step - loss: 0.4299 - binary_accuracy: 0.7811 - val_loss: 0.4047 - val_binary_accuracy: 0.8056
Epoch 18/20
58/58 [==============================] - 1s 9ms/step - loss: 0.4102 - binary_accuracy: 0.7811 - val_loss: 0.3847 - val_binary_accuracy: 0.8021
Epoch 19/20
58/58 [==============================] - 1s 9ms/step - loss: 0.3992 - binary_accuracy: 0.7785 - val_loss: 0.3762 - val_binary_accuracy: 0.8056
Epoch 20/20
58/58 [==============================] - 1s 9ms/step - loss: 0.3936 - binary_accuracy: 0.7758 - val_loss: 0.3689 - val_binary_accuracy: 0.7986
```



As expected, the accuracy has remarkably declined but on this network we found a better loss.

We need to refine this architecture design to improve the accuracy again. Then, convolutions are not enough and we have to add different layer types in order to do that.

We tried several combinations looking for a better tradeoff between number of parameters and performance.

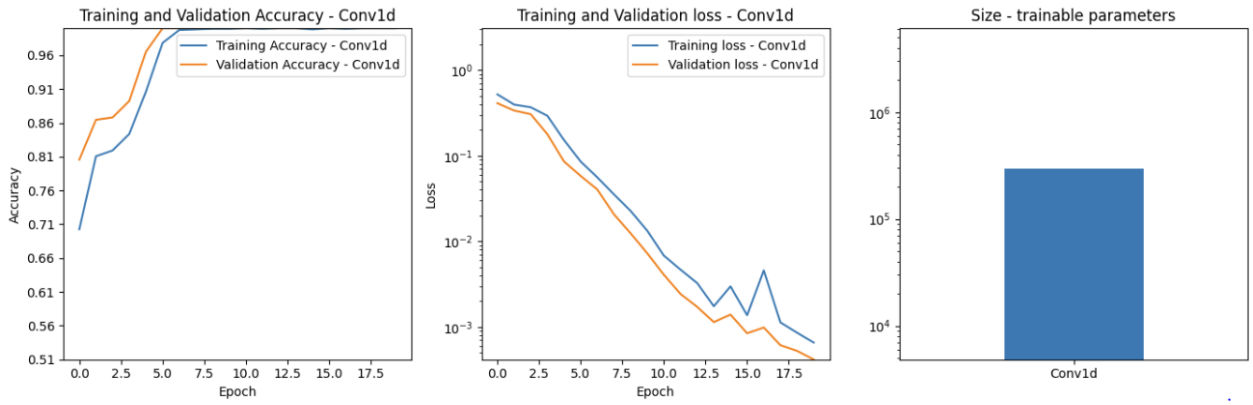
4° Experiment: Conv1d+Maxpooling1d Network

```
Model: "sequential_60"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_43 (Embedding)    (None, 2000, 2000)        40000

 conv1d_128 (Conv1D)         (None, 1998, 32)          192032

 max_pooling1d_128 (MaxPool  (None, 999, 32)           0
 ing1D)

 conv1d_129 (Conv1D)         (None, 997, 64)           6208

 max_pooling1d_129 (MaxPool  (None, 498, 64)           0
 ing1D)

 conv1d_130 (Conv1D)         (None, 496, 128)          24704

 max_pooling1d_130 (MaxPool  (None, 248, 128)          0
 ing1D)

 flatten_54 (Flatten)        (None, 31744)             0

 dropout_43 (Dropout)        (None, 31744)             0

 dense_101 (Dense)           (None, 1)                 31745

=================================================================
Total params: 294689 (1.12 MB)
Trainable params: 294689 (1.12 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
<keras.src.engine.sequential.Sequential at 0x7dd1600beb30>
```

Most notable improvement we have found out is adding an Embedding layer on the top of our network.
The Embedding layer is known for its good behavior when we care about feature learning, more specifically this kind of layer maps each word (or character) in our vocabulary to a dense vector of real numbers (the embeddings), which are learned by the model.
Then, these learned word embeddings capture semantic meanings and relationships between the words or characters.

The use of this type of layers is frequently used when we talk about the Bag of Words methods which is common in Word Embedding and Tokenization.

```
 Training conv1d model

Epoch 1/20
58/58 [==============================] - 10s 101ms/step - loss: 0.5211 - binary_accuracy: 0.7026 - val_loss: 0.4107 - val_binary_accuracy: 0.8056
Epoch 2/20
58/58 [==============================] - 2s 33ms/step - loss: 0.3963 - binary_accuracy: 0.8106 - val_loss: 0.3377 - val_binary_accuracy: 0.8646
Epoch 3/20
58/58 [==============================] - 2s 40ms/step - loss: 0.3687 - binary_accuracy: 0.8193 - val_loss: 0.3059 - val_binary_accuracy: 0.8681
Epoch 4/20
58/58 [==============================] - 2s 38ms/step - loss: 0.2931 - binary_accuracy: 0.8436 - val_loss: 0.1786 - val_binary_accuracy: 0.8924
Epoch 5/20
58/58 [==============================] - 2s 36ms/step - loss: 0.1526 - binary_accuracy: 0.9062 - val_loss: 0.0857 - val_binary_accuracy: 0.9653
Epoch 6/20
58/58 [==============================] - 2s 37ms/step - loss: 0.0856 - binary_accuracy: 0.9783 - val_loss: 0.0581 - val_binary_accuracy: 1.0000
Epoch 7/20
58/58 [==============================] - 2s 40ms/step - loss: 0.0559 - binary_accuracy: 0.9974 - val_loss: 0.0404 - val_binary_accuracy: 1.0000
Epoch 8/20
58/58 [==============================] - 2s 38ms/step - loss: 0.0354 - binary_accuracy: 0.9983 - val_loss: 0.0207 - val_binary_accuracy: 1.0000
Epoch 9/20
58/58 [==============================] - 2s 33ms/step - loss: 0.0227 - binary_accuracy: 0.9991 - val_loss: 0.0124 - val_binary_accuracy: 1.0000
Epoch 10/20
58/58 [==============================] - 2s 36ms/step - loss: 0.0133 - binary_accuracy: 0.9991 - val_loss: 0.0073 - val_binary_accuracy: 1.0000
Epoch 11/20
58/58 [==============================] - 2s 36ms/step - loss: 0.0068 - binary_accuracy: 1.0000 - val_loss: 0.0041 - val_binary_accuracy: 1.0000
Epoch 12/20
58/58 [==============================] - 2s 40ms/step - loss: 0.0047 - binary_accuracy: 0.9991 - val_loss: 0.0024 - val_binary_accuracy: 1.0000
Epoch 13/20
58/58 [==============================] - 2s 37ms/step - loss: 0.0033 - binary_accuracy: 1.0000 - val_loss: 0.0017 - val_binary_accuracy: 1.0000
Epoch 14/20
58/58 [==============================] - 2s 36ms/step - loss: 0.0017 - binary_accuracy: 1.0000 - val_loss: 0.0011 - val_binary_accuracy: 1.0000
Epoch 15/20
58/58 [==============================] - 2s 36ms/step - loss: 0.0030 - binary_accuracy: 0.9983 - val_loss: 0.0014 - val_binary_accuracy: 1.0000
Epoch 16/20
58/58 [==============================] - 2s 34ms/step - loss: 0.0014 - binary_accuracy: 1.0000 - val_loss: 8.4405e-04 - val_binary_accuracy: 1.0000
Epoch 17/20
58/58 [==============================] - 2s 36ms/step - loss: 0.0046 - binary_accuracy: 0.9991 - val_loss: 9.8638e-04 - val_binary_accuracy: 1.0000
Epoch 18/20
58/58 [==============================] - 2s 40ms/step - loss: 0.0011 - binary_accuracy: 1.0000 - val_loss: 6.1253e-04 - val_binary_accuracy: 1.0000
Epoch 19/20
58/58 [==============================] - 2s 40ms/step - loss: 8.5761e-04 - binary_accuracy: 1.0000 - val_loss: 5.2624e-04 - val_binary_accuracy: 1.0000
Epoch 20/20
58/58 [==============================] - 2s 37ms/step - loss: 6.5860e-04 - binary_accuracy: 1.0000 - val_loss: 4.1736e-04 - val_binary_accuracy: 1.0000
```



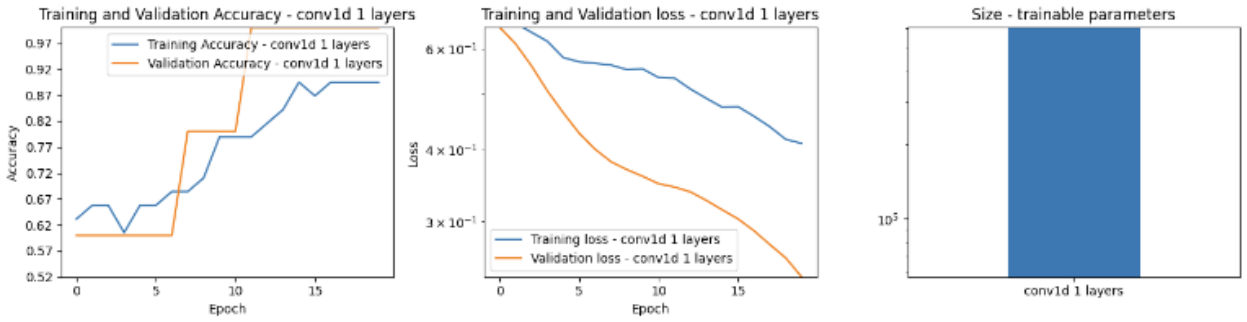
So far, this model has the best performance despite being 1/10 the size of the Dense baseline.

We still have some work to do because we have used random parameters and hyperparameters. Before optimizing them, we are going to check the optimal number of Convolutional layers.

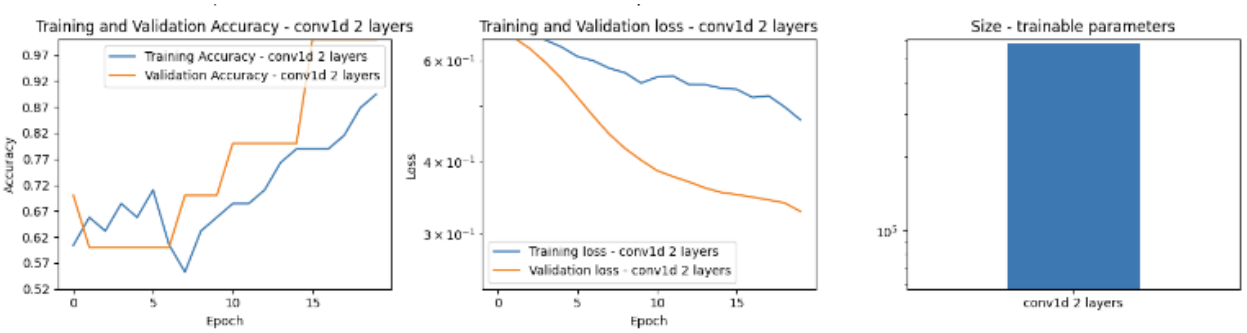
Experiment on sequential number of layers in Conv1d Network

We tested manually how the network behaves by incrementally changing the number of Conv+Pool pairs. We started from 1 pair, and we noticed that it's not useful to put a lot of them.
These are the results of each attempt:

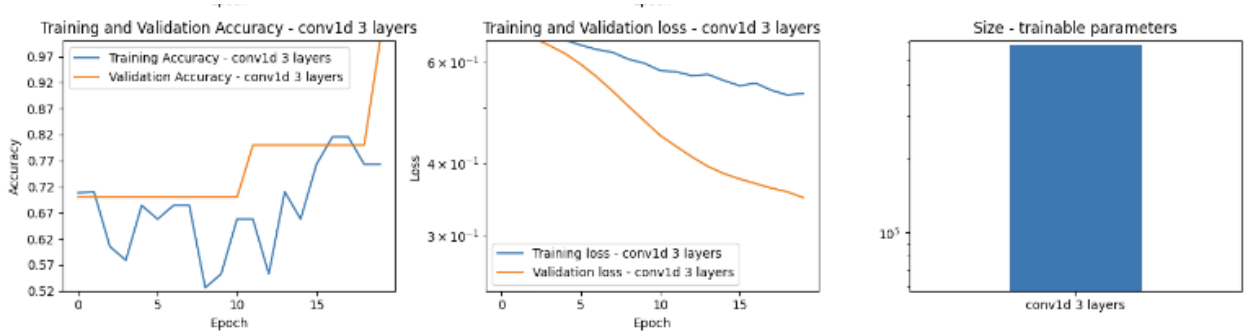
1 Layer:



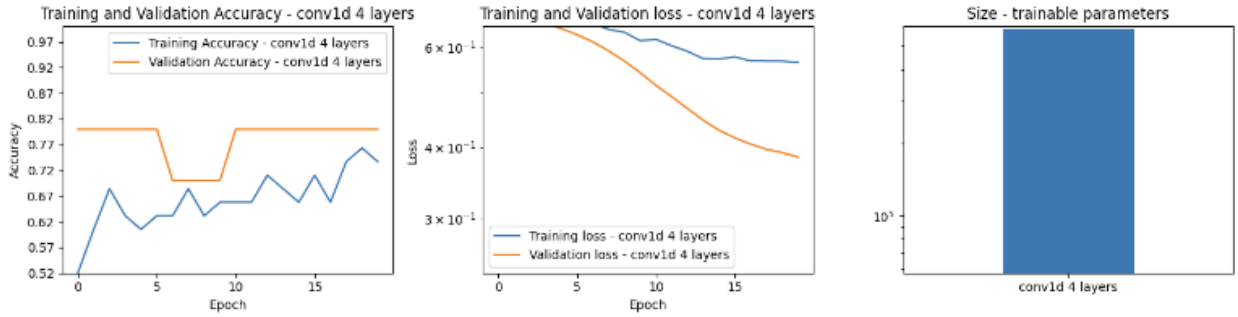
2 Layers:



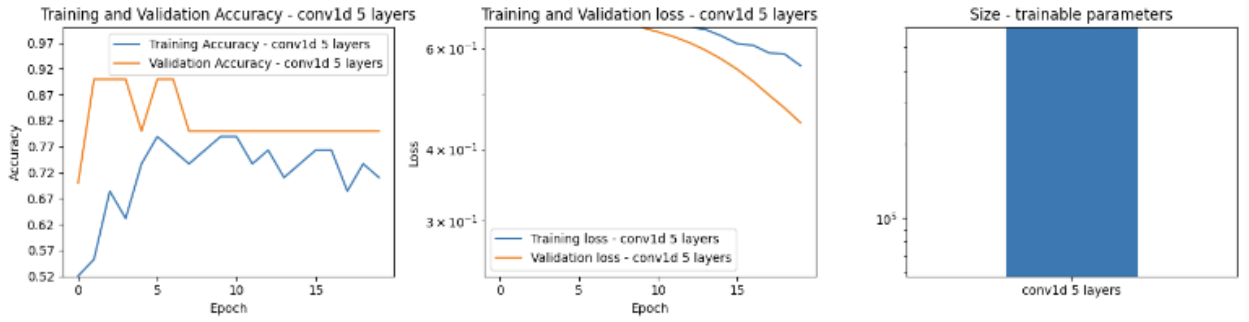
3 Layers:



4 Layers:



5 Layers:



Conclusions on the Convolutional+Maxpooling layers

The size of the network does not increase since there are no trainable parameters in these layers.

Actually, it decreases by a very small amount, since the shape of the latter layers is reduced. There is no meaningful change in the performance, so we decide to adopt the model with 2 pairs. It's simpler than the 5-pairs one, and it's where we see the biggest drop in network size. The differences are still very narrow, since all the candidates have their accuracy between 0.86 and 0.88, and about a 5% size variance.

Final Architecture: two Conv1d+Maxpooling1d Network

We have now found out the best architecture in terms of layers and parameters for our classification problem.
The code below shows the implementation of this network. Since we have used heuristic numbers as layers parameters, for the final model we decide to leave these as function arguments.

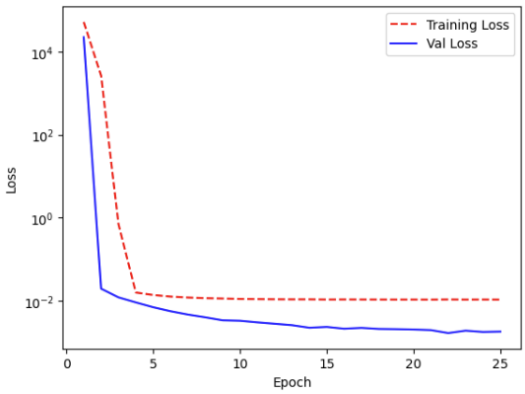
```
# TIME TO BUILD AN OPTIMIZED MODEL
def build_definitive_model(embed_dim,l1_size,l2_size,kernel_1,kernel_2,pool_1,pool_2,learning,drop):
  model = Sequential()

  model.add(Embedding(input_dim=NN_batch_size, output_dim=embed_dim,  input_length=desired_size_char))

  # KERNEL SIZE: calculated with Keras Tuner
  model.add(Conv1D(filters=l1_size, kernel_size=kernel_1, activation='relu', padding="valid"))
  # POOLING SIZE: calculated with Keras Tuner
  model.add(MaxPooling1D(pool_1,padding="valid"))
  # KERNEL SIZE: calculated with Keras Tuner
  model.add(Conv1D(filters=l2_size, kernel_size=kernel_2, activation='relu', padding="valid"))
  # POOLING SIZE: calculated with Keras Tuner
  model.add(MaxPooling1D(pool_2,padding="valid"))

  # DROPOUT RATE: calculated with Keras Tuner
  model.add(Flatten())

  model.add(Dropout(drop))

  model.add(Dense(1, activation='sigmoid'))
  model.build(trainX.shape)
  model.compile(loss=loss, optimizer=Adam(learning_rate=learning), metrics=metrics)

  return model
```

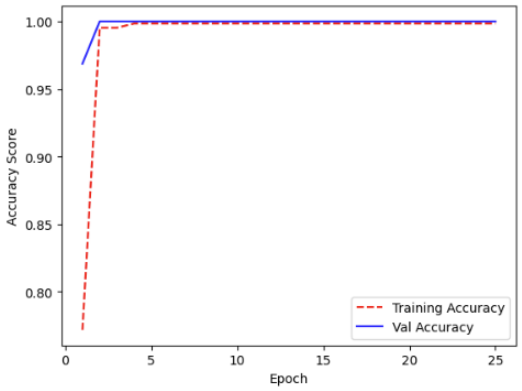
This function is then used to instantiate the model for Keras Tuner which is going to optimize it for us.
More specifically, we pick Hyperband as tuning algorithm and we turn every parameter into a variable for it.
We started the tuning with our heuristic values and had the tuner search into a wide range around them; then we reduce the range and the look-up step, running the tuner more than once to be sure to have the best combination.
We were curious about the hyper-bandit problem behind the algorithm, so we added some extra code to compare the result of the tuner with a different approach: using all the numbers as constant but one, which is the variable to be tuned.
We compare the result of a loop on this "Single-Armed" Bandit tuner with what we obtained from the Multi-Armed tuning (the standard HyperBand) and we see that the results are more or less the same.
Finally, the model summary of this optimized network is shown below:

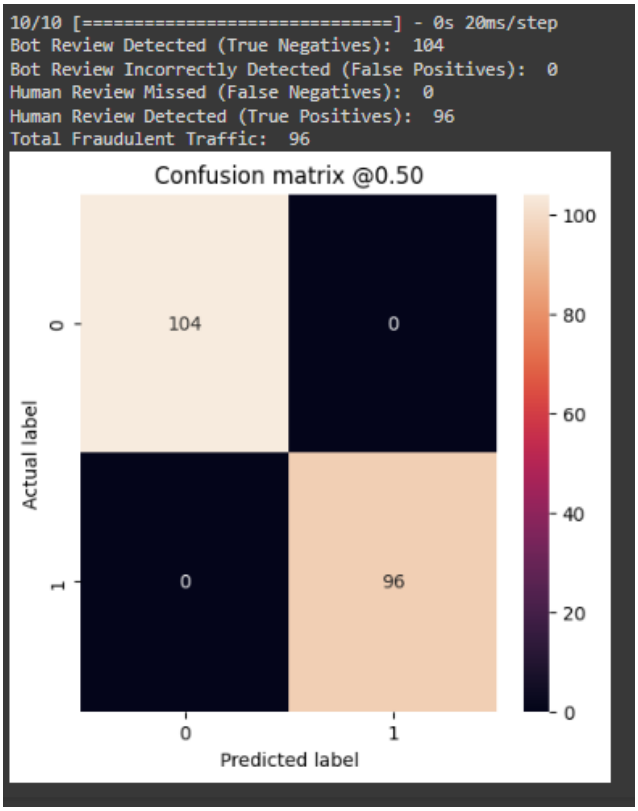
```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_3 (Embedding)     (None, 2000, 350)         7000

 conv1d_6 (Conv1D)           (None, 1996, 300)         525300

 max_pooling1d_6 (MaxPoolin  (None, 499, 300)          0
 g1D)

 conv1d_7 (Conv1D)           (None, 496, 150)          180150

 max_pooling1d_7 (MaxPoolin  (None, 165, 150)          0
 g1D)

 flatten_3 (Flatten)         (None, 24750)             0

 dropout_3 (Dropout)         (None, 24750)             0

 dense_3 (Dense)             (None, 1)                 24751

=================================================================
Total params: 737201 (2.81 MB)
Trainable params: 737201 (2.81 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

We did one final training before moving on the test set for the evaluation. The results are surprisingly good considering we have completely removed the preprocessing.
Additionally, the size of the network is smaller than the Dense model despite having a much better validation accuracy.

```
Epoch 1/60
32/32 [==============================] - 13s 178ms/step - loss: 52376.7891 - binary_accuracy: 0.7719 - val_loss: 22465.3906 - val_binary_accuracy: 0.9688
Epoch 2/60
32/32 [==============================] - 2s 75ms/step - loss: 2665.9299 - binary_accuracy: 0.9953 - val_loss: 0.0194 - val_binary_accuracy: 1.0000
Epoch 3/60
32/32 [==============================] - 2s 63ms/step - loss: 0.7040 - binary_accuracy: 0.9953 - val_loss: 0.0120 - val_binary_accuracy: 1.0000
Epoch 4/60
32/32 [==============================] - 2s 63ms/step - loss: 0.0157 - binary_accuracy: 0.9984 - val_loss: 0.0091 - val_binary_accuracy: 1.0000
Epoch 5/60
32/32 [==============================] - 2s 79ms/step - loss: 0.0137 - binary_accuracy: 0.9984 - val_loss: 0.0069 - val_binary_accuracy: 1.0000
Epoch 6/60
32/32 [==============================] - 2s 67ms/step - loss: 0.0126 - binary_accuracy: 0.9984 - val_loss: 0.0056 - val_binary_accuracy: 1.0000
Epoch 7/60
32/32 [==============================] - 2s 58ms/step - loss: 0.0119 - binary_accuracy: 0.9984 - val_loss: 0.0046 - val_binary_accuracy: 1.0000
Epoch 8/60
32/32 [==============================] - 2s 57ms/step - loss: 0.0115 - binary_accuracy: 0.9984 - val_loss: 0.0040 - val_binary_accuracy: 1.0000
Epoch 9/60
32/32 [==============================] - 2s 58ms/step - loss: 0.0113 - binary_accuracy: 0.9984 - val_loss: 0.0034 - val_binary_accuracy: 1.0000
Epoch 10/60
32/32 [==============================] - 2s 62ms/step - loss: 0.0110 - binary_accuracy: 0.9984 - val_loss: 0.0033 - val_binary_accuracy: 1.0000
Epoch 11/60
32/32 [==============================] - 2s 62ms/step - loss: 0.0109 - binary_accuracy: 0.9984 - val_loss: 0.0030 - val_binary_accuracy: 1.0000
Epoch 12/60
32/32 [==============================] - 2s 58ms/step - loss: 0.0108 - binary_accuracy: 0.9984 - val_loss: 0.0028 - val_binary_accuracy: 1.0000
Epoch 13/60
32/32 [==============================] - 2s 65ms/step - loss: 0.0108 - binary_accuracy: 0.9984 - val_loss: 0.0026 - val_binary_accuracy: 1.0000
Epoch 14/60
32/32 [==============================] - 2s 58ms/step - loss: 0.0108 - binary_accuracy: 0.9984 - val_loss: 0.0022 - val_binary_accuracy: 1.0000
Epoch 15/60
32/32 [==============================] - 2s 59ms/step - loss: 0.0107 - binary_accuracy: 0.9984 - val_loss: 0.0023 - val_binary_accuracy: 1.0000
Epoch 16/60
32/32 [==============================] - 2s 61ms/step - loss: 0.0107 - binary_accuracy: 0.9984 - val_loss: 0.0021 - val_binary_accuracy: 1.0000
Epoch 17/60
32/32 [==============================] - 2s 56ms/step - loss: 0.0107 - binary_accuracy: 0.9984 - val_loss: 0.0022 - val_binary_accuracy: 1.0000
Epoch 18/60
32/32 [==============================] - 2s 56ms/step - loss: 0.0107 - binary_accuracy: 0.9984 - val_loss: 0.0021 - val_binary_accuracy: 1.0000
Epoch 19/60
32/32 [==============================] - 2s 61ms/step - loss: 0.0106 - binary_accuracy: 0.9984 - val_loss: 0.0021 - val_binary_accuracy: 1.0000
Epoch 20/60
32/32 [==============================] - 2s 65ms/step - loss: 0.0106 - binary_accuracy: 0.9984 - val_loss: 0.0020 - val_binary_accuracy: 1.0000
Epoch 21/60
32/32 [==============================] - 2s 57ms/step - loss: 0.0106 - binary_accuracy: 0.9984 - val_loss: 0.0020 - val_binary_accuracy: 1.0000
Epoch 22/60
32/32 [==============================] - 2s 55ms/step - loss: 0.0107 - binary_accuracy: 0.9984 - val_loss: 0.0017 - val_binary_accuracy: 1.0000
Epoch 23/60
32/32 [==============================] - 2s 56ms/step - loss: 0.0106 - binary_accuracy: 0.9984 - val_loss: 0.0019 - val_binary_accuracy: 1.0000
Epoch 24/60
32/32 [==============================] - 2s 56ms/step - loss: 0.0107 - binary_accuracy: 0.9984 - val_loss: 0.0018 - val_binary_accuracy: 1.0000
Epoch 25/60
32/32 [==============================] - 2s 58ms/step - loss: 0.0106 - binary_accuracy: 0.9984 - val_loss: 0.0018 - val_binary_accuracy: 1.0000
```







The bias

The unreasonably perfect behavior is very likely due to some heavy bias we introduced: removing the preprocessing phase caused the fake reviews to be all similar to each other in formatting and text structures, while human ones contain unique traits like html tags that should have been cleaned, repeated punctuation signs, and multiple unnecessary spaces. Therefore, the work made so far is still correct in logic, but it's flawed for the performance evaluation. Before moving to the final tests, and using the biggest dataset chunk, we had to go back to the sample loading and fix the preprocessing issue.

We still avoid to use a tokenizer, but we run these steps in order to standardize the data:

- removing unnecessary spaces and blank lines
- filtering out html tags (in detail, words encased in <.....>)
- replacing multiple punctuation marks with a single one
- transforming everything to lowercase

Now, every sample is a continuous string of text with no format, and a slightly reduced readability for humans. This should be enough to remove the bias, making the task less trivial.

```python
# PREPROCESSING FUNCTION

# this function aim to create uniformity between files, to make them have the same shape

def preprocess(text):
    cleaned_text = ""
    del cleaned_text
    # Remove spaces, newlines, and extra spaces
    cleaned_text = ' '.join(re.sub(r'\s+', ' ', text).split())

    # Remove HTML tags (including <br>)
    cleaned_text = re.sub(r'<.*?>', '', cleaned_text)

    # Replace multiple punctuation marks with a single one
    cleaned_text = re.sub(r'[!?.]+', '.', cleaned_text)

    # Convert all characters to lowercase
    cleaned_text = cleaned_text.lower()

    return cleaned_text
```

Results with the full dataset

In order to complete our project, we run our model on two versions of the dataset: the smaller one used until now, and the big one as described in the introduction.
From the full dataset we only use 7500 human reviews of the 12500 available for two reasons:

- keep the labels distribution balanced.
- Some of the human reviews are discarded because they are too long or too short, but we were not sure of how many, so we needed a number that is divisible for the batch size.

For the sake of avoiding repetition, in the following paragraphs we will only include the results obtained from the various networks without focusing again on the structure of the network.
We can already say the results are more or less the same as the smaller dataset.

BERT

From the figure below we can see that now the working dataset is the bigger one.

```
# LOADING THE DATASET

#TRAIN PART OF THE DATASET
raw_train_ds = tf.keras.utils.text_dataset_from_directory(
    dataset_path_train,
    batch_size=batch_size,
    validation_split=0.2,
    subset='training',
    seed=seed)

class_names = raw_train_ds.class_names
train_ds = raw_train_ds.cache().prefetch(buffer_size=AUTOTUNE)

#VALIDATION PART OF THE DATASET
val_ds = tf.keras.utils.text_dataset_from_directory(
    dataset_path_train,
    batch_size=batch_size,
    validation_split=0.2,
    subset='validation',
    seed=seed)

val_ds = val_ds.cache().prefetch(buffer_size=AUTOTUNE)

#TEST PART OF THE DATASET
test_ds = tf.keras.utils.text_dataset_from_directory(
    dataset_path_test,
    batch_size=batch_size)

test_ds = test_ds.cache().prefetch(buffer_size=AUTOTUNE)
```

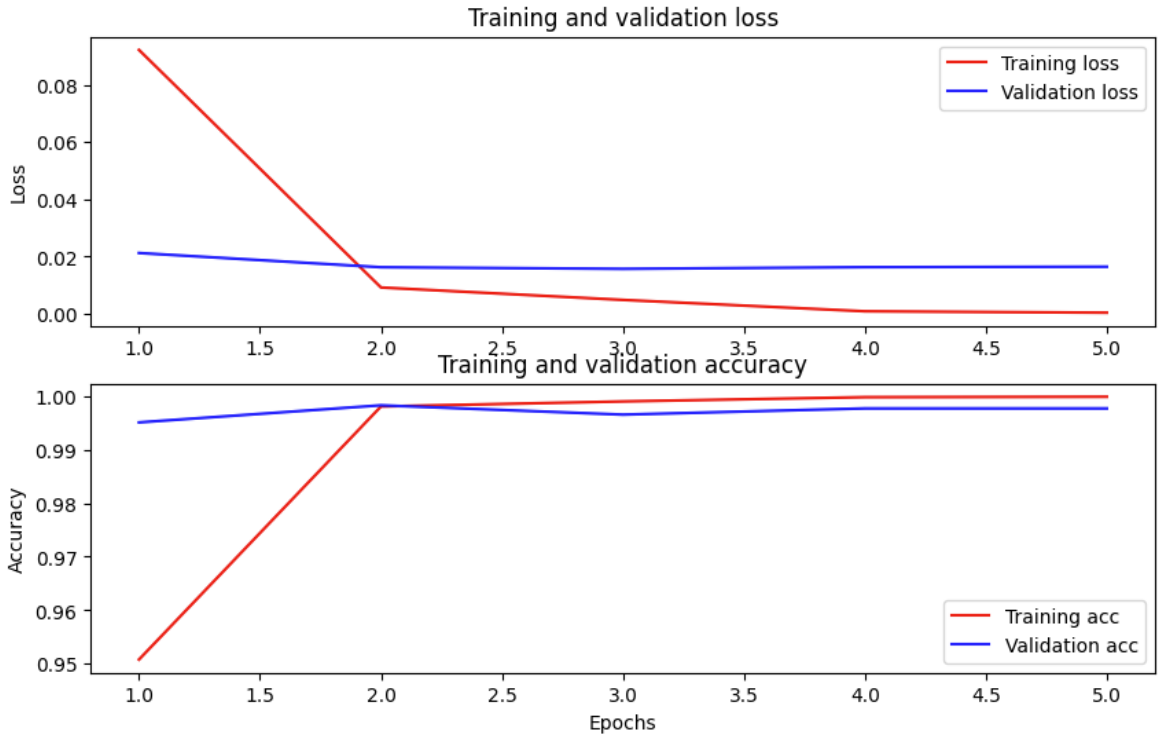
```
Found 17255 files belonging to 2 classes.
Using 13804 files for training.
Found 17255 files belonging to 2 classes.
Using 3451 files for validation.
Found 3056 files belonging to 2 classes.
```

Also in this case the results achieved show that our binary classifier model is performing really well.

```
Training model with https://tfhub.dev/tensorflow/small_bert/bert_en_uncased_L-4_H-512_A-8/1
Epoch 1/5
1381/1381 [==============================] - 1753s 1s/step - loss: 0.0922 - binary_accuracy: 0.9508 - val_loss: 0.0212 - val_binary_accuracy: 0.9951
Epoch 2/5
1381/1381 [==============================] - 1494s 1s/step - loss: 0.0092 - binary_accuracy: 0.9980 - val_loss: 0.0163 - val_binary_accuracy: 0.9983
Epoch 3/5
1381/1381 [==============================] - 1281s 927ms/step - loss: 0.0048 - binary_accuracy: 0.9990 - val_loss: 0.0157 - val_binary_accuracy: 0.9965
Epoch 4/5
1381/1381 [==============================] - 1270s 919ms/step - loss: 8.7312e-04 - binary_accuracy: 0.9998 - val_loss: 0.0163 - val_binary_accuracy: 0.9
977
Epoch 5/5
1381/1381 [==============================] - 1263s 915ms/step - loss: 3.8366e-04 - binary_accuracy: 0.9999 - val_loss: 0.0165 - val_binary_accuracy: 0.9
977
```

```python
#EVALUATION
loss, accuracy = classifier_model.evaluate(test_ds)

print(f'Loss: {loss}')
print(f'Accuracy: {accuracy}')
```

```
306/306 [==============================] - 59s 192ms/step - loss: 0.0163 - binary_accuracy: 0.9971
Loss: 0.016312792897224426
Accuracy: 0.9970549941062927
```



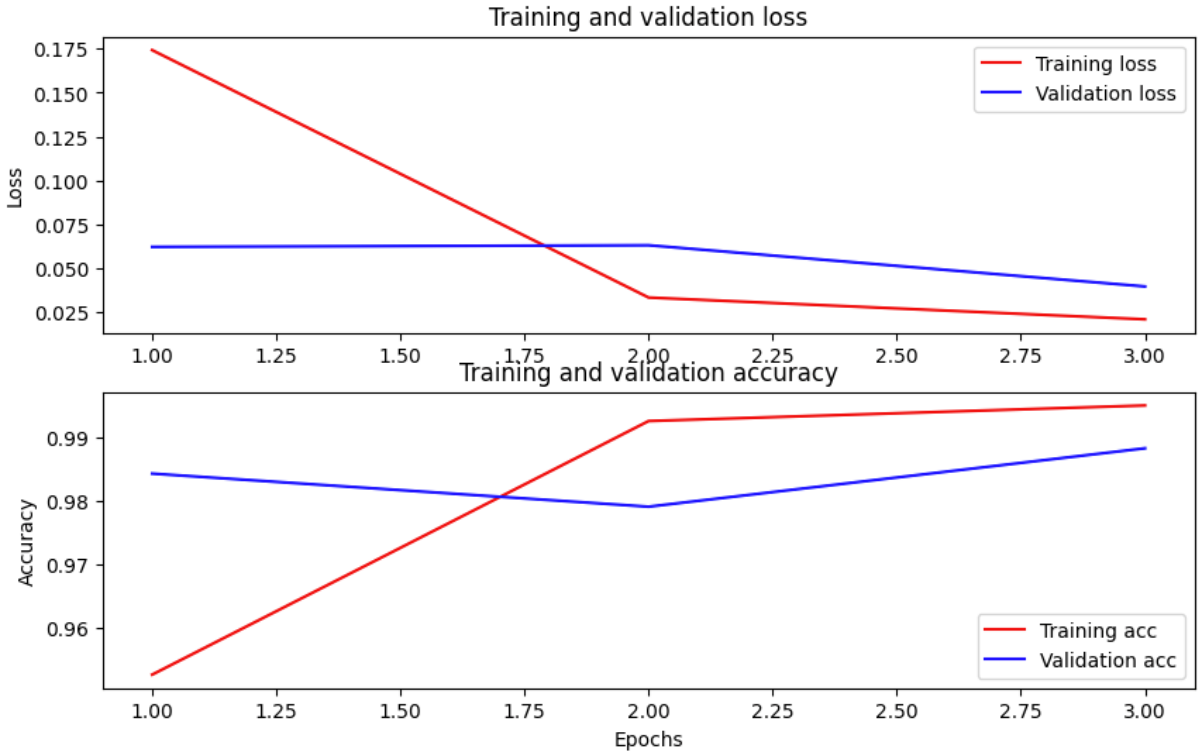
GPT-2

The comparison also on this pretrained model is unchanged since the validation accuracy is still near 99%.
The execution time has now increased from about 180 seconds to 1400 seconds which is not justified with no performance improvement.

```
Model: "model"
_________________________________________________________________________________________________
 Layer (type)                Output Shape              Param #   Connected to
=================================================================================================
 input_1 (InputLayer)        [(None, None)]            0         []

 input_2 (InputLayer)        [(None, None)]            0         []

 tfgpt2_model (TFGPT2Model)  TFBaseModelOutputWithPastA  1244405   ['input_1[0][0]',
                             ndCrossAttentions(last_hid  76         'input_2[0][0]']
                             den_state=(None, None, 768
                             ),
                              past_key_values=None, hid
                             den_states=None, attention
                             s=None, cross_attentions=N
                             one)

 tf.math.reduce_mean (TFOpL  (None, 768)               0         ['tfgpt2_model[0][0]']
 ambda)

 dropout_37 (Dropout)        (None, 768)               0         ['tf.math.reduce_mean[0][0]']

 dense (Dense)               (None, 1)                 769       ['dropout_37[0][0]']

=================================================================================================
Total params: 124441345 (474.71 MB)
Trainable params: 769 (3.00 KB)
Non-trainable params: 124440576 (474.70 MB)
_________________________________________________________________________________________________
```

```
Epoch 1/3
2600/2600 [==============================] - 1423s 546ms/step - loss: 0.1742 - accuracy: 0.9525 - val_loss: 0.0621 - val_accuracy: 0.9843
Epoch 2/3
2600/2600 [==============================] - 1403s 540ms/step - loss: 0.0332 - accuracy: 0.9926 - val_loss: 0.0630 - val_accuracy: 0.9791
Epoch 3/3
2600/2600 [==============================] - 1384s 532ms/step - loss: 0.0208 - accuracy: 0.9951 - val_loss: 0.0395 - val_accuracy: 0.9883
```

```
127/127 [==============================] - 50s 395ms/step - loss: 0.0412 - accuracy: 0.9892
127/127 [==============================] - 47s 374ms/step
```



When training this model, we encountered a challenge related to execution time. The free version of Google Colab was insufficient to handle the additional resource usage required. As a result, we opted to perform the training on Kaggle instead.

CNN From Scratch

It is important to remember that the reviews used as input for this network are now preprocessed with the function described before in the "the bias" paragraph.
This is the reason why the performance are now lower, but we can notice no other meaningful differences due to the dataset size's change.
Again, to avoid repetitions we restrict the comparisons to only two architecture: the dense model ("Low Effort network") and the definitive one ("Final Architecture").
We notice that after the tuning of the hyperparameters the Final architecture is now bigger than the original Low Effort model, which is acceptable for the higher accuracy but it makes the comparison harder. So, we implement a bigger Dense model with the same size of our Final one and we compare the performance of a total of 3 models.

Low_effort Network

The figures below shows the structure and the performance of original Low Effort model on the big dataset.
This is the smallest of the 3 models that we considered and the one we expect the worst performance from.

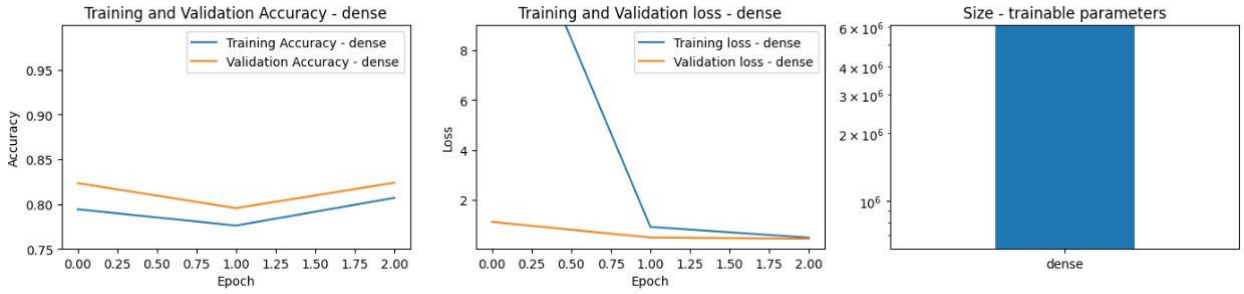
```
dense_model.summary()
```

Model: "sequential_26"

Layer (type)	Output Shape	Param #
flatten_26 (Flatten)	(12000, 2000)	0
dense_86 (Dense)	(12000, 2000)	4,002,000
dense_87 (Dense)	(12000, 1000)	2,001,000
dense_88 (Dense)	(12000, 100)	100,100
dense_89 (Dense)	(12000, 1)	101

Total params: 6,103,201 (23.28 MB)
Trainable params: 6,103,201 (23.28 MB)
Non-trainable params: 0 (0.00 B)

```
Epoch 1/20
192/192 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - binary_accuracy: 0.8377 - loss: 41.3981 - val_binary_accuracy: 0.8233 - val_loss: 1.1138
Epoch 2/20
192/192 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - binary_accuracy: 0.7668 - loss: 1.0875 - val_binary_accuracy: 0.7954 - val_loss: 0.4851
Epoch 3/20
192/192 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - binary_accuracy: 0.7954 - loss: 0.5107 - val_binary_accuracy: 0.8238 - val_loss: 0.4367
```



Instead on the figures below we have the bigger Dense model:

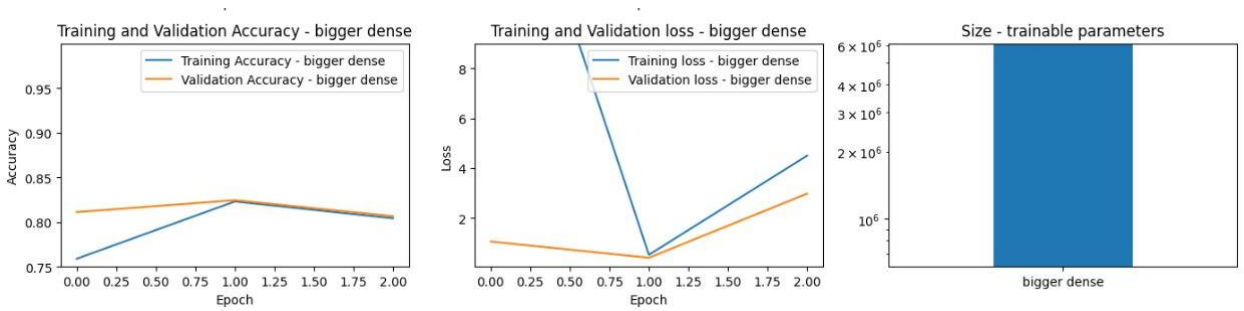
```
big_model.summary()
```

Model: "sequential_28"

Layer (type)	Output Shape	Param #
flatten_28 (Flatten)	(12000, 2000)	0
dense_91 (Dense)	(12000, 2400)	4,802,400
dense_92 (Dense)	(12000, 1200)	2,881,200
dense_93 (Dense)	(12000, 100)	120,100
dense_94 (Dense)	(12000, 1)	101

Total params: 7,803,801 (29.77 MB)
Trainable params: 7,803,801 (29.77 MB)
Non-trainable params: 0 (0.00 B)

```
Epoch 1/20
192/192 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - binary_accuracy: 0.7699 - loss: 50.6915 - val_binary_accuracy: 0.8112 - val_loss: 1.0548
Epoch 2/20
192/192 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - binary_accuracy: 0.8154 - loss: 0.7841 - val_binary_accuracy: 0.8246 - val_loss: 0.4022
Epoch 3/20
192/192 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - binary_accuracy: 0.8250 - loss: 1.2652 - val_binary_accuracy: 0.8067 - val_loss: 2.9731
```



As we can see, the performances of the two models are pretty much the same despite the increase of the size of the network on the second case.

Final Architecture: two Conv1d+Maxpooling1d Network

We have run the Keras tuner (Hyperband) again obtaining the following set of hyperparameters.

```
Best set of hyperparameters so far:
Embedding size: 1900
Conv1D layer 1 size: 600
Conv1d Kernel 1 size: 6
MaxPooling layer 1 size: 7
Conv1D layer 2 size: 450
Conv1d Kernel 2 size: 3
MaxPooling layer 2 size: 5
dropout: 3
learning rate: 0.001
```

```
final_model.summary()
```

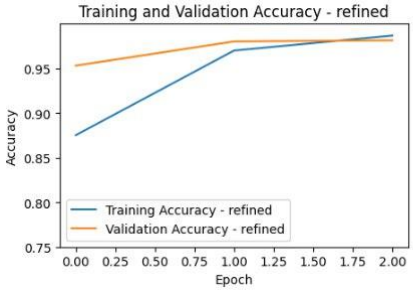
Model: "sequential_27"

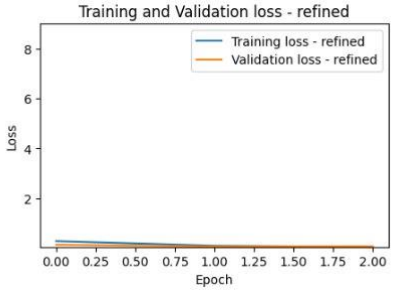
Layer (type)	Output Shape	Param #
embedding_6 (Embedding)	(12000, 2000, 1900)	95,000
conv1d_12 (Conv1D)	(12000, 1995, 600)	6,840,600
max_pooling1d_12 (MaxPooling1D)	(12000, 285, 600)	0
conv1d_13 (Conv1D)	(12000, 283, 450)	810,450
max_pooling1d_13 (MaxPooling1D)	(12000, 56, 450)	0
flatten_27 (Flatten)	(12000, 25200)	0
dropout_6 (Dropout)	(12000, 25200)	0
dense_90 (Dense)	(12000, 1)	25,201

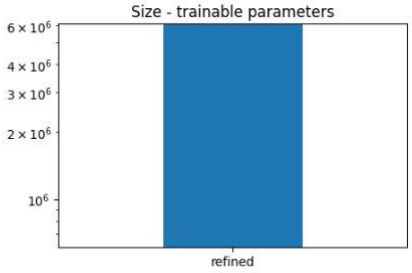
Total params: 7,771,251 (29.64 MB)
Trainable params: 7,771,251 (29.64 MB)
Non-trainable params: 0 (0.00 B)

The testing phase gave us the results shown in the images below.
Now that we have a big amount of test samples, we are sure that our solution to the problem is working properly.

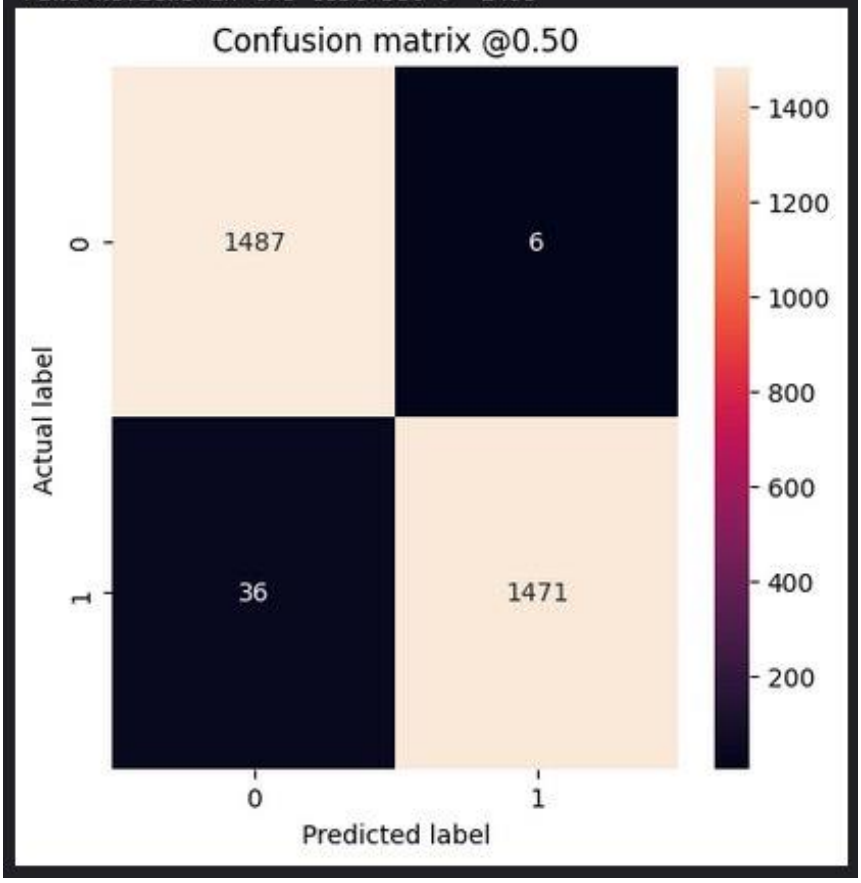
```
Epoch 1/20
192/192 ━━━━━━━━━━━━━━━━━━━━ 83s 417ms/step - binary_accuracy: 0.8340 - loss: 0.3796 - val_binary_accuracy: 0.9529 - val_loss: 0.1287
Epoch 2/20
192/192 ━━━━━━━━━━━━━━━━━━━━ 80s 415ms/step - binary_accuracy: 0.9617 - loss: 0.1126 - val_binary_accuracy: 0.9800 - val_loss: 0.0606
Epoch 3/20
192/192 ━━━━━━━━━━━━━━━━━━━━ 80s 415ms/step - binary_accuracy: 0.9827 - loss: 0.0478 - val_binary_accuracy: 0.9812 - val_loss: 0.0642
```








```
60/60 ━━━━━━━━━━━━━━━━━━━━ 5s 80ms/step
Bot Review Detected (True Negatives):  1487
Bot Review Incorrectly Detected (False Positives):  6
Human Review Missed (False Negatives):  36
Human Review Detected (True Positives):  1471
Human Reviews in the test set :  1507
Fake Reviews in the test set :  1493
```

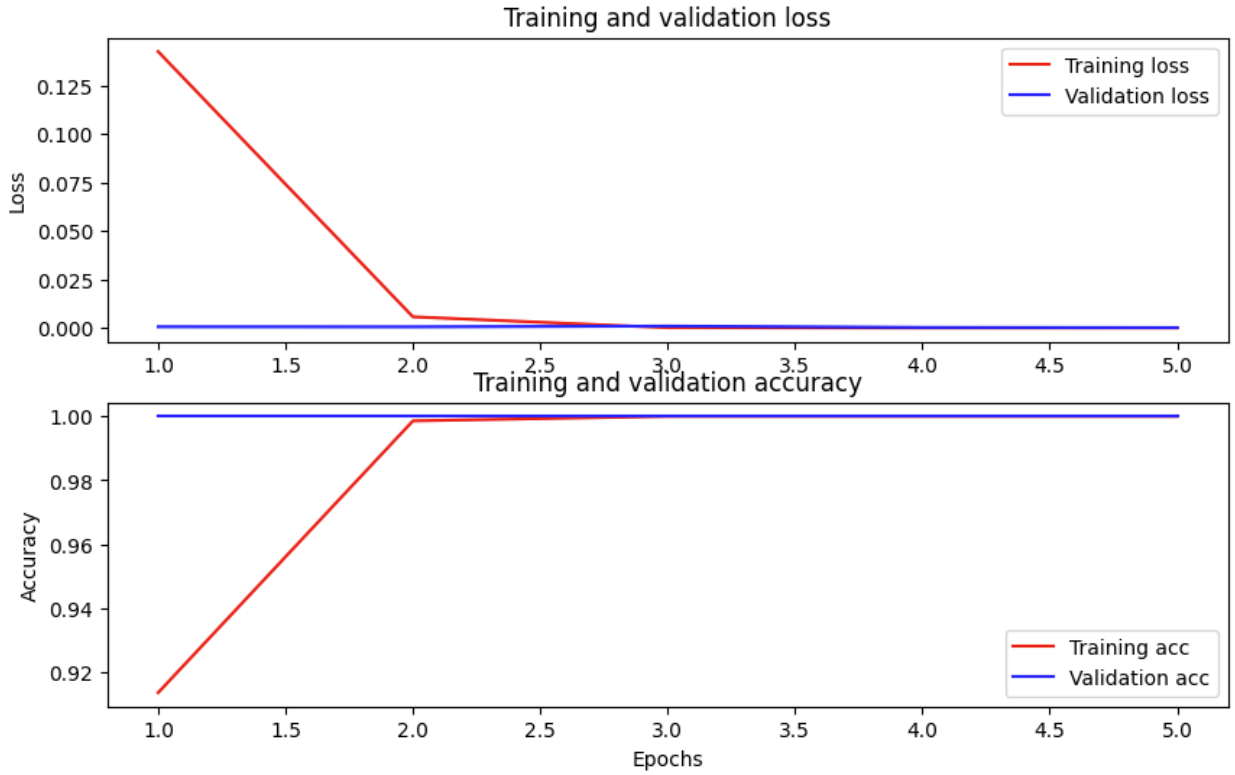


Graphical Summary for Model Comparison

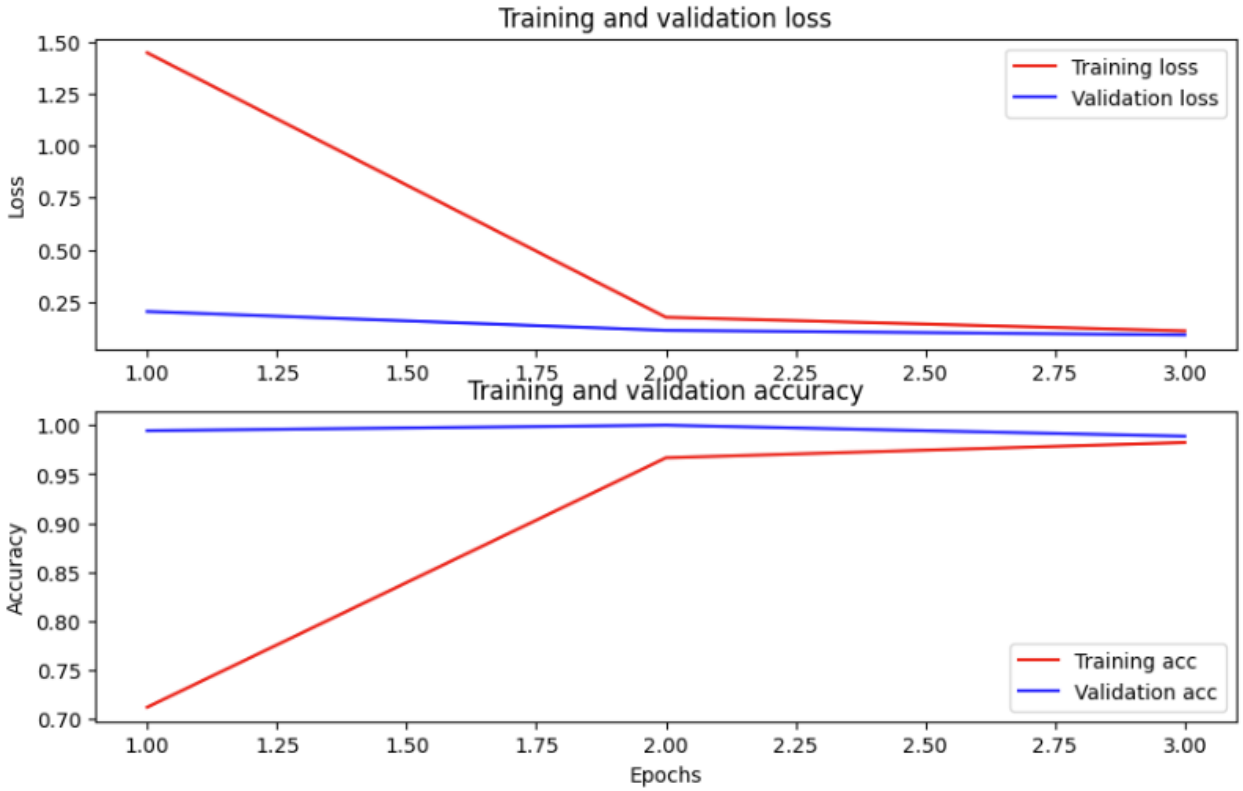
This subchapter serves as the final summary, focusing exclusively on the graphical representation of the neural network models discussed in the document.
The purpose of presenting these graphs is to facilitate a clear and direct comparison of the models' performances.
By isolating the visual data, we enable a straightforward and comparative evaluation, allowing for an immediate grasp of the relative strengths and efficiencies of each model.

Reduced Dataset

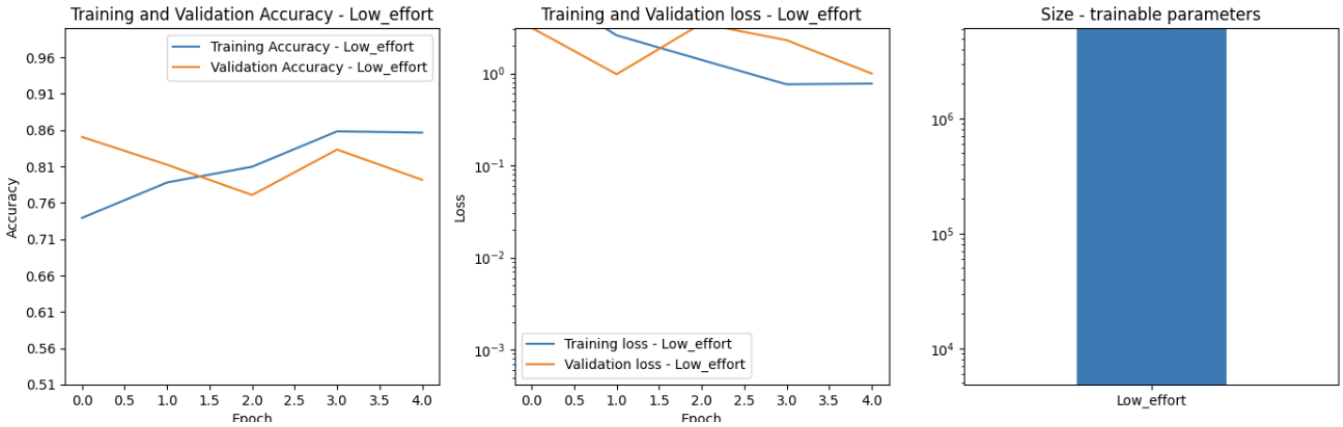
BERT



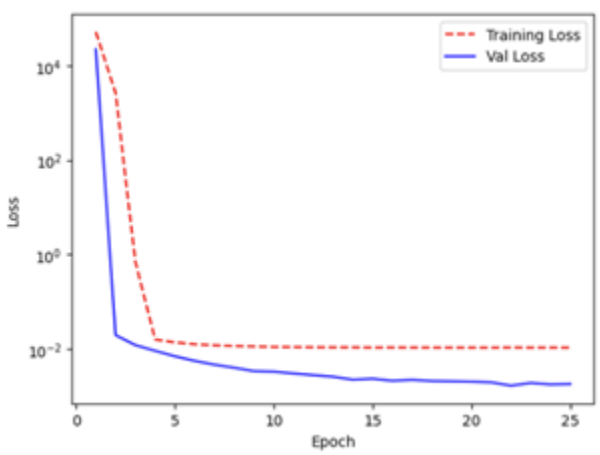
GPT2

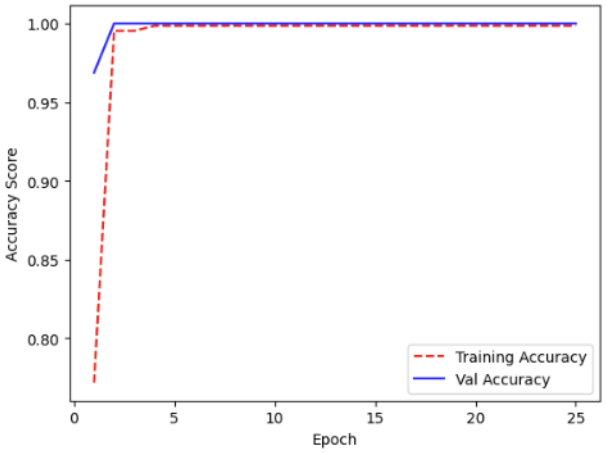


LOW EFFORT



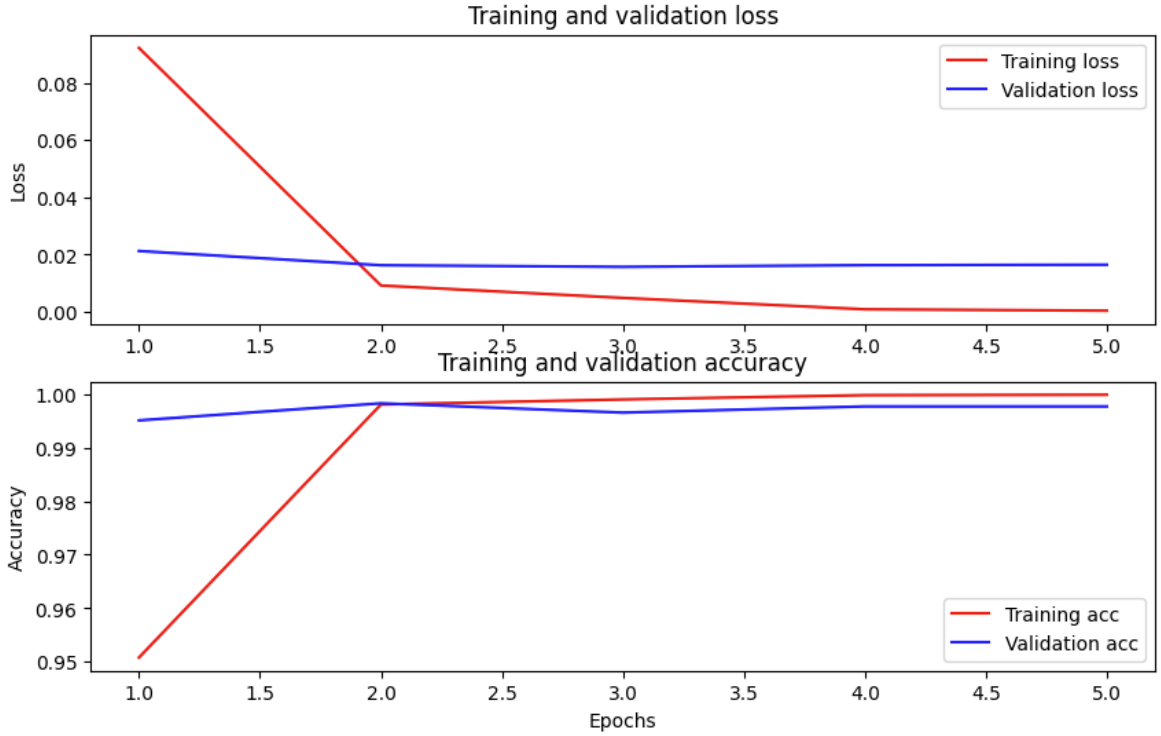
FINAL ARCHITECTURE: TWO CONV1D+MAXPOOLING1D NETWORK



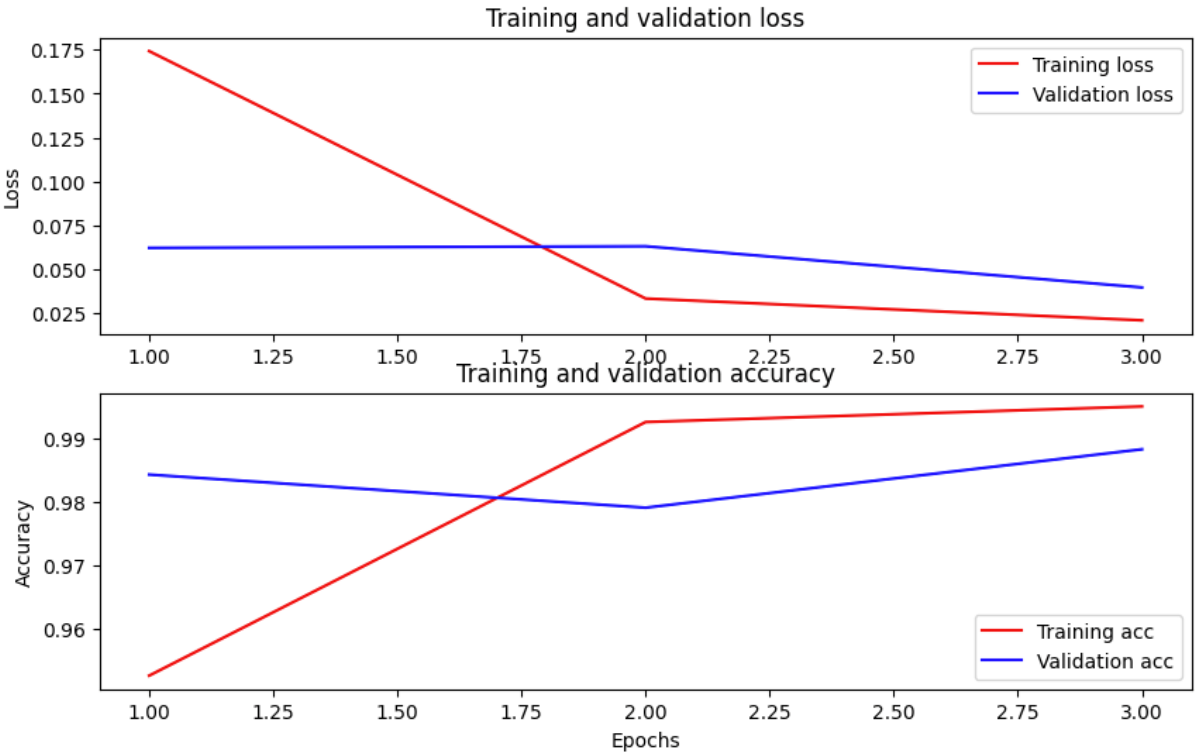


Full Dataset

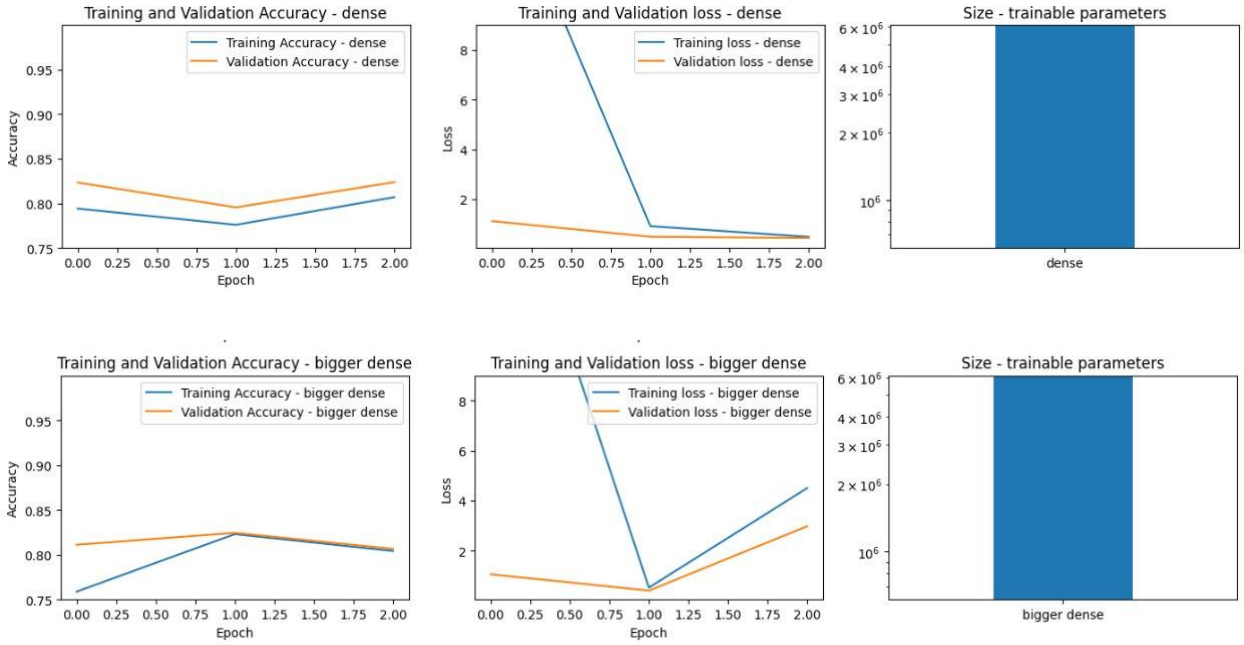
BERT



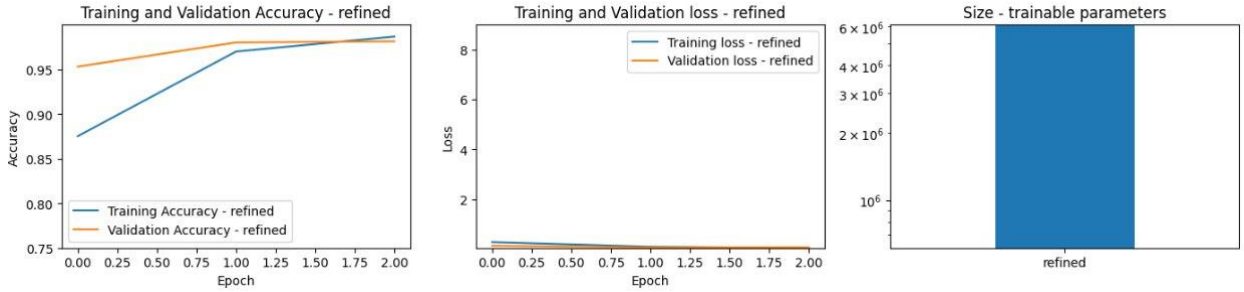
GPT2



LOW EFFORT



FINAL ARCHITECTURE: TWO CONV1D+MAXPOOLING1D NETWORK



Conclusions

Testing the different datasets brings us to the same conclusions: the classifier is able to recognize the fake reviews almost perfectly, as the other pre-trained models.
The design and the tuning of our original classifier is worth the effort, since "simple" Dense networks did not show any particular performance increase by enlarging their size. We managed to reach an accuracy similar to the pre-trained models thanks to the Keras hypertuner, but it's important to remember the difference in size:

- GPT2 contains about 144 millions parameters
- BERT contains about 28 millions parameters
- our model contains 7.7 millions parameters

It's obvious that our model is smaller because it is designed for this specific task, while the other ones are focused on sentiment analysis and they are able to recognize features we're not using.
We can explain this almost perfect results because of the bias we discussed when introducing the preprocessing step: the classifiers are exploiting the limited AI creativity and they are able to notice the similarities in their lexical patterns. Despite variating the generative prompts and mixing several kinds of reviews, there still is a strong limit on how well they can produce new content.
Could a human perform the same task?
Yes, but there is a significant difference: reading all these reviews takes a lot of time, and you have to read many of them before being able to notice the similarities in the fake ones. Therefore, introducing a bot classification to filter out fakes is going to be a critical step for any kind of business that would be bothered by the noise of generative AIs.
Nowadays there are several systems that are strongly influenced by reviews (Amazon, Tripadvisor, IMDB, etc..), and quickly generating a big amount of them is a valuable (but malicious) asset.

Possible improvements

Testing different dataset

This task should be evaluated on different datasets, in particular by gathering more data about real/fake content without involving a generative bot, and using content marked as fake by humans. This kind of data could prove to be challenging to fetch, but it would minimize the bias of AI creativity as much as possible.

Multimedia classification

It's already circulating the idea of detectors able to recognize images or audio that have been generated artificially. It's a similar task, so it could prove to lead to similar conclusions.

Recurrent neural networks

We decided to take as input a fixed amount of characters, by padding or cropping the reviews we had. A different and more elaborate approach (similar to GPT's one) could be making a RNN instead of a CNN, able to take as input a stream of words with indefinite length.

Adversarial networks

Since the classification can be automated, our network could be used in a GAN to produce better reviews that are harder to detect as fake. It would also be interesting to make a human try to read those reviews.

Fake-checking services SaaS

When a model performs well enough, it could be useful to others, and so it can be sold. It could be a possible business idea to implement a web-app that offers the APIs to check if some text looks fake or not, as long as it's on the same subject as the training samples.

References

- IMDB dataset:
https://paperswithcode.com/dataset/imdb-movie-reviews

- short article on chatGPT limitations:
https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9939079/

- paper on chatGPT limitations:
https://www.sciencedirect.com/science/article/pii/S266734522300024X

- paper on fact-checking and generative AI:
https://arxiv.org/pdf/2405.15985

- BERT introduction and tutorial:
https://www.kaggle.com/code/harshjain123/bert-for-everyone-tutorial-implementation

- OpenAI pretrained models:
https://huggingface.co/docs/transformers/en/model_doc/openai-gpt

- Wikipedia review bombing page:
Review bomb - Wikipedia

- Wikipedia GPT-2 model page:
https://en.wikipedia.org/wiki/GPT-2

- Wikipedia BERT model page:
https://en.wikipedia.org/wiki/BERT_(language_model)