NLP project: Using CNNs for Text Classification

based on the article by Xiang Zhang, Junbo Zhao, Yann Lecun

Imeneo Silvia

Data Science and Scientific Computing - AA 2023-2024

1 Introduction

A Convolutional Neural Network (CNN) is a type of deep learning model primarily used for processing grid-like data, such as images. It is made up of multiple layers, including convolutional layers, pooling layers, and fully connected layers, which work together to automatically learn features from input data:

- **Convolutional layers** apply filters to detect patterns such as edges or textures. They use small filters (also called kernels, or receptive fields, or feature detectors) which are matrices of learnable weights that slide over the input data [1]. The weights are the same for the entire input and, for each position of the filter, a dot product is computed between the filter and the local region of the input data, producing a single output. The filter then moves across the input to create the feature map. The input data passes through multiple convolutional layers and the output of a convolutional layer is a set of feature maps, each of which is used to represent the spatial presence of a certain learned pattern or feature in the input data.

- **Pooling layers** are used to reduce the spatial dimensions (height and width) of the feature maps, making the network computationally more efficient while retaining the most important information. A pooling operation (like max pooling or average pooling[1]) is applied to non-overlapping regions of the feature maps, so that each region is then replaced with a single value. The output is a downsampled version of the input feature map: pooling layers are, in fact, used between convolutional layers to reduce the spatial dimensions.

[1]Max pooling takes the maximum value within the region; average pooling takes the average value within the region.

- **Fully connected (or linear) layers** are used to combine the high-level features learned by previous layers and map them to the final output, such as a classification score or regression value. In a fully connected layer, every neuron is connected to every neuron in the previous layer, meaning that each output unit is influenced by all the neurons in the previous layer. The input to each neuron is a weighted sum of the outputs from the previous layer, plus a bias term. This is then followed by a non-linear activation function (e.g., ReLU, sigmoid, or softmax for classification tasks). The output is typically a vector representing the final prediction of the network.

Unlike what happens in traditional feedforward neural networks, where each input **neuron** is connected to each output neuron in the next layer, in CNNs the use of convolutions over the input layer results in a situation where each **region** of the input is connected to a neuron in the output.
Each layer applies different filters, typically hundreds or thousands, and combines their results. During the training phase, a CNN automatically learns the values of its filters [1].

Despite being usually used in Computer Vision, CNN cal also be used in Natural Language Processing, where the input is made of textual data. The filter used for this application will not be a multi-dimensional filter, but a one-dimensional one [2]. For example, a 1x2 filter could be used to look over a bi-gram (2 sequential words in a piece of text), or a 1x3 filter for a tri-grams and so on [1].

In this work, we use one-dimensional convolutional networks to perform text classification for sentiment analysis. The task is based on the article *"Character-level Convolutional Networks for Text Classification"*, by Xiang Zhang, Junbo Zhao, Yann Lecun, and on the GitHub of protonx-tf-03-projects.

The goal of this work is to assess the different performance of large and small CNN models on datasets of different sizes.

2 Designing a CNN model

Assume we start with a discrete input function $g(x)$ and a discrete kernel function, or filter, $f(x)$, which is used to stride over the input. The convolution, $h(y)$, between $f(x)$ and $g(x)$, is:

$$h(y) = \sum_{x=1}^{k} f(x) \cdot g(y \cdot d - x + c)$$

The convolution is computed as a weighted sum of $f(x)$ and $g(x)$ for each position of the sequence. The result of this operation is a new sequence $h(y)$ that

contains the features extracted from the original data. The “stride” parameter determines how much the filter $f(x)$ moves during the convolution operation. The convolution is performed using a set of kernel functions (weights) on a set of inputs $g_i(x)$ and outputs $h_j(x)$.
In this model, we use 6 convolutional layers and 3 one-dimensional max-pooling ones, to reduce the size of the data and keep only the most important information. Eventually we also have 3 fully connected layers.
The non-linear activation function used is the ReLU [2] and the chosen optimizer is Adam [3].

2.1 Details on model design: model.py script

1. The implementation is done using TensorFlow-Keras.

```
from tensorflow.keras.layers import Embedding, Conv1D, MaxPooling1D
    , Flatten, Dense, Dropout
from tensorflow.keras.initializers import RandomNormal
import tensorflow as tf
```

2. As shown below, we start by defining the class `CharCNN` to customize the basic Keras class `tf.keras.Model`, which provides common functionality for all neural network models in Keras.
In the `__init__` constructor, we pass the following parameters:

- `vocab_size`: the size of the vocabulary, so the number of unique tokens in the dataset.
- `embedding_size`: the size of the embeddings, so the dimensionality of the vector representation for each token.
- `max_length`: the maximum length of the input sequences, so the number of tokens in each sequence.
- `num_classes`: the number of output classes for our text classification.
- `feature`: a string that determines the size of the model. Later we see how we implement both a small and a large model.
- `padding`: the padding mode used for the convolutional layers. Padding is a technique used in CNNs to add extra values (usually zeroes) at the edges of the input data, before applying the convolution, in order to control the output size and the loss of information on the edges. We set it to be either

[2] ReLU essentially takes the maximum value between 0 and x. In other words, if the value is negative, it becomes 0; if it is positive, it remains unchanged.

[3] The Adam, Adaptive Moment Estimation, optimizer adjusts the learning rate for each parameter individually by computing both the first moment (the mean of how steep the slope is when it is figuring out how to improve) and second moment of the gradients (the average of how fast the slope is changing). It adapts the learning rate based on these moving averages. [3]

`same`, meaning that the output size remains the same as the input size, or `valid`, meaning that the output size decreases.

- `trainable`: a boolean that determines whether the layers of the model are trainable or not. By "trainable" we mean whether the model's parameters (such as weights and biases) in the layers can be updated during the training process to minimize the loss function, using an optimization algorithm (Adam, in our case).
- `name`: the name of the model (used by Keras).
- `dtype`: the data type of the model (e.g., float32, float64).

```python
class CharCNN(tf.keras.Model):
    def __init__(
    self,
    vocab_size,
    embedding_size,
    max_length,
    num_classes,
    feature='small',
    padding='same',
    trainable=True,
    name=None,
    dtype=None
    ):
        super(CharCNN, self).__init__(name=name, dtype=dtype)
        assert feature in ['small', 'large'], "Feature must be either 'small' or 'large'"
        assert padding in ['valid', 'same'], "Padding must be either 'valid' or 'same'"

        #assigning values passed as arguments to instance variables of the CharCNN object
        self.padding = padding
        self.num_classes = num_classes
        self.trainable = trainable
```

3. As already mentioned, we configure two model sizes: small and large. They both use the same number of layers, but they differ in:

- the number of units (neurons) in the fully connected layers: fewer neurons reduce the model's capacity to learn complex representations, but make the model more computationally efficient;
- the number of filters in the convolutional layers: the number of filters determines how many different features the model learns at each layer, but a higher number is more computationally costly;
- the standard deviation of the Gaussian distribution used to initialize the weights of the convolutional layers: a smaller standard deviation results in smaller random values for the initial weights.

All these values are set as indicated in the article of Xiang Zhang, Junbo Zhao, and Yann Lecun.

```
# Configurations based on model size
        if feature == 'small':
            self.units_fc = 1024
            self.num_filter = 256
            self.stddev = 0.05
        else:
            self.units_fc = 2048
            self.num_filter = 1024
            self.stddev = 0.02

        # Weight initialization for convolutional layers
        self.initializers = RandomNormal(mean=0., stddev=self.stddev, seed=42)

        # Model configuration
        self.vocab_size = vocab_size
        self.embedding_size = embedding_size
        self.max_length = max_length
```

4. We then proceed with the specification of the embedding layer, which is used to transform input data into continuous-valued vectors of fixed size. Here:

- `self.vocab_size` indicates the total number of unique character tokens in our dataset. Each token is mapped to a vector.
- `self.embedding_size`: indicates the dimensionality of the embedding vectors. In the paper, the alphabet size, and so the embedding size, is set to 70, corresponding to a combination of 26 English letters, 10 digits, and 33 other characters.
- `self.max_length`: specifies the fixed length to which all input sequences will be padded or truncated.
- `self.trainable`: determines whether the weights of the embedding layer should be trainable or fixed during training.

```
        # Embedding layer
        self.embedding = Embedding(self.vocab_size, self.embedding_size, input_length=self.max_length, trainable=self.trainable)
```

5. As already mentioned, in this CNN we have 6 convolutional layers, 3 max-pooling layers and 3 fully connected layers.

In the **convolutional layers**, we use `self.num_filter` to indicate the number of kernels in the layers. This determines how many different features the model will extract from the input data. `kernel_size` indicates how many consecutive elements at the time, from the input sequence, will be looked by the kernel. The numbers 7 and 3 (indicated in the script below) are taken from the paper. As already said, the activation function used after the convolution operation is ReLU. Non-linear activation functions introduce non-linearity into the model, helping it learn more complex patterns.

Max-pooling layers are used to reduce the dimensionality of the feature map, helping to reduce computational cost and prevent overfitting. `pool_size=3` means that for every 3 consecutive elements in the feature map, the maximum value is selected.

After the convolutional and pooling layers, the output feature maps are flattened in order to convert all the learned features into a format that can be used by the fully connected layers. This is necessary because fully connected layers expect vector inputs, not multi-dimensional data [4].
Eventually, we can apply the **fully connected layers**, which take the features extracted by the convolutional layers and combine them for final classification. The first 2 layers use the ReLu activation function to introduce non-linearity, while the final one uses softmax to convert the output values into probabilities that sum to 1.

```
        # Convolutional and max-pooling layers
        self.conv1d_1 = Conv1D(self.num_filter, kernel_size=7,
    kernel_initializer=self.initializers, activation='relu',
    padding=self.padding, trainable=self.trainable)
        self.maxpooling1d_1 = MaxPooling1D(pool_size=3)

        self.conv1d_2 = Conv1D(self.num_filter, kernel_size=7,
    kernel_initializer=self.initializers, activation='relu',
    padding=self.padding, trainable=self.trainable)
        self.maxpooling1d_2 = MaxPooling1D(pool_size=3)

        self.conv1d_3 = Conv1D(self.num_filter, kernel_size=3,
    kernel_initializer=self.initializers, activation='relu',
    padding=self.padding, trainable=self.trainable)
        self.conv1d_4 = Conv1D(self.num_filter, kernel_size=3,
    kernel_initializer=self.initializers, activation='relu',
    padding=self.padding, trainable=self.trainable)
        self.conv1d_5 = Conv1D(self.num_filter, kernel_size=3,
    kernel_initializer=self.initializers, activation='relu',
    padding=self.padding, trainable=self.trainable)

        self.conv1d_6 = Conv1D(self.num_filter, kernel_size=3,
    kernel_initializer=self.initializers, activation='relu',
    padding=self.padding, trainable=self.trainable)
        self.maxpooling1d_6 = MaxPooling1D(pool_size=3)

        # Fully connected layers
        self.flatten = Flatten()
        self.fc1 = Dense(self.units_fc, activation='relu',
    trainable=self.trainable)
        self.drp1 = Dropout(0.5)
        self.fc2 = Dense(self.units_fc, activation='relu',
    trainable=self.trainable)
        self.drp2 = Dropout(0.5)
        self.fc3 = Dense(self.num_classes, activation='softmax',
    trainable=self.trainable)
```

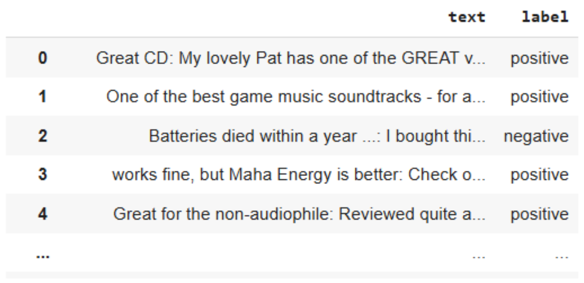
6. After the definition of the `__init__` constructor, the `call` method is used to define how data flow through the various layers of the network. As already said, the input (tokenized text) is mapped to vectors during the embedding step, then filters (kernels) are applied to learn local features in the input sequence, and sequentially max-pooling is applied to reduce dimensionality of feature maps. Eventually, the fully connected layers process the flattened features to learn more complex relationships and dropout is applied to prevents overfitting by randomly dropping neurons during training. The output is the class probability, obtained using softmax.

```
    def call(self, inputs):

        # Embedding
        x = self.embedding(inputs)

        # Convolutional layers and max-pooling
        x = self.maxpooling1d_1(self.conv1d_1(x))
        x = self.maxpooling1d_2(self.conv1d_2(x))
        x = self.conv1d_3(x)
        x = self.conv1d_4(x)
        x = self.conv1d_5(x)
        x = self.maxpooling1d_6(self.conv1d_6(x))

        # Flattening
        x = self.flatten(x)

        # Fully connected layers
        x = self.drp1(self.fc1(x))
        x = self.drp2(self.fc2(x))
        outputs = self.fc3(x)

        return outputs
```

3 Dataset

The dataset that we used is made of a total of 4 millions reviews of Amazon users [4], initially divided in a training set (3.6 million reviews) and a testing one (400K reviews).
We started by editing the dataset in order to have it made of two different columns, `text` and `label`, as shown in figure 1.

[4]The original dataset can be downloaded from here.

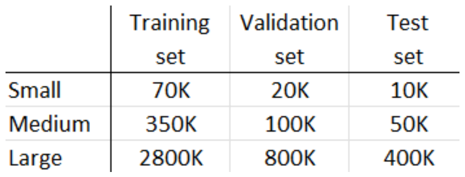


	text	label
0	Great CD: My lovely Pat has one of the GREAT v...	positive
1	One of the best game music soundtracks - for a...	positive
2	Batteries died within a year ...: I bought thi...	negative
3	works fine, but Maha Energy is better: Check o...	positive
4	Great for the non-audiophile: Reviewed quite a...	positive
...	...	...

Figure 1: Dataset

From the initial training set, 800K reviews had been taken to make the validation set.
Eventually, our dataset was made of: 2,800,000 reviews for the training set; 800,000 reviews for the validation set; 400,000 reviews for the test set.

Given that our idea was to see how the size of the dataset would influence the performance of the char-based CNN, we generated two other smaller datasets from the initial one, so eventually we ended up with a small, a medium and a large dataset:



	Training set	Validation set	Test set
Small	70K	20K	10K
Medium	350K	100K	50K
Large	2800K	800K	400K

Figure 2: Composition of training, validation and test sets in the small, medium and large datasets (in number of reviews).

3.1 Data preprocessing

Before being used to train the model, data is preprocessed by removing punctuation and any HTML tags, URLs and emojis that could potentially be present in the text.
A tokenizer is then built using the `Tokenizer` class from Keras. The tokenizer is built and trained on the input data and it assigns an integer index to each unique token based on its frequency: the most frequent token gets index 1, the second-most frequent gets index 2, and so on. We usually do not consider all the unique tokens, but just the most frequent ones, here indicated with `vocab_size`. The `oov_token` parameter indicates how to handle potential out-of-vocabulary tokens, meaning words that the tokenizer did not encounter during the training step.
Once the tokenizer is built, it is then applied so that each token in the text is replaced by its corresponding index (token ID). `pad_sequences` is used to

ensure that all the sequences have the same length, specified by `maxlen`. Where `padding` is needed, it will be applied at the end of the sequence.

```
from tensorflow.keras.preprocessing.text import tokenizer_from_json

class Dataset:
...
...
    def build_tokenizer(self, texts, vocab_size):
        # Creating tokenizer and training it on provided texts
        tokenizer = Tokenizer(vocab_size, oov_token="<OOV>")
        tokenizer.fit_on_texts(texts)
        return tokenizer

    def tokenize(self, tokenizer, texts, max_len):
        # Transforming texts in sequences and padding them for uniform lenght
        tensor = tokenizer.texts_to_sequences(texts)
        tensor = pad_sequences(tensor, maxlen=max_len, padding="post")
        return tensor
```

4 Implementation of the model

4.1 Training

Considering that we have three datasets of different size and that we want to implement both a small and a large version of the model, we trained a total of six different models. During the training step, the data is prepared and the model is built as described before.

The customization of arguments is left to the users via the command-line. The arguments that can be customized are:

```
    parser.add_argument("--batch-size", default=128, type=int,
        help="Batch size for training")
    parser.add_argument("--mode", default="small", type=str,
        help="Model mode: 'small', 'large', or 'all'")
    parser.add_argument("--vocab-folder", default=f'{home_dir}/saved_vocab/CharCNN/', type=str,
        help="Folder where to save the vocabulary")
    parser.add_argument("--train-file", default='data.csv', type=str, help="Path to the training file")
    parser.add_argument("--epochs", default=3, type=int,
        help="Number of epochs for training")
    parser.add_argument("--embedding-size", default=100, type=int,
        help="Number of characteristics of a token represented in the embedding vector")
    parser.add_argument("--test-size", default=0.3, type=float,
        help="Percentage of data for testing")
    parser.add_argument("--num-classes", default=2, type=int,
        help="Number of output classes")
    parser.add_argument("--learning-rate", default=0.001, type=float, help="Learning rate for the optimizer")
    parser.add_argument("--smallCharCNN-folder", default="
    smallCharCNN", type=str,
        help="Folder where to save the small model")
    parser.add_argument("--largeCharCNN-folder", default="
    largeCharCNN", type=str,
        help="Folder where to save the large model")
    parser.add_argument("--padding", default="same", type=str,
        help="Padding mode: 'same' or 'valid'")
```

Below an example of the command-line arguments set for the small version of the model applied to the small dataset.

```

# SMALL DATASET - SMALL MODEL
!python train.py
    --train-file train_70k.csv
    --mode small
    --vocab-folder /content/NLP/saved_vocab/smallModel_smallDataset
    /
    --smallCharCNN-folder smallModel_smallDataset
    --epochs 3
    --test-size 0.3
    --num-classes 2
    --learning-rate 0.001
    --batch-size 128
```

For each of the 6 models, we initially set:

- `--epochs 3`: the number of times the entire dataset is passed through the model during training;
- `--learning-rate 0.001`: it is the learning rate of the optimizer and it controls how much the model's weights are updated during each step of training;
- `--batch-size 128`: during each training step, the dataset is divided into 128 batches, and the model updates its weights after processing each batch. This indicates how many examples the model sees before updating its weights.

4.1.1 Results of the training step

During training, the model's performance is evaluated on a validation set after each epoch, to monitor overfitting. As shown in Fig.3, for all the 6 models, and especially for those considering the smallest dataset, the training accuracy improves rapidly over the epochs, indicating that the model is successfully learning from the training data. The training loss decreases, showing that the model is getting better at minimizing the error on the training set and adjusting its weights to fit the data. On the other side, validation accuracy, despite being quite high, slightly decreases over the epoch, and validation loss increases, indicating that the model might be overfitting the training data when we use 3

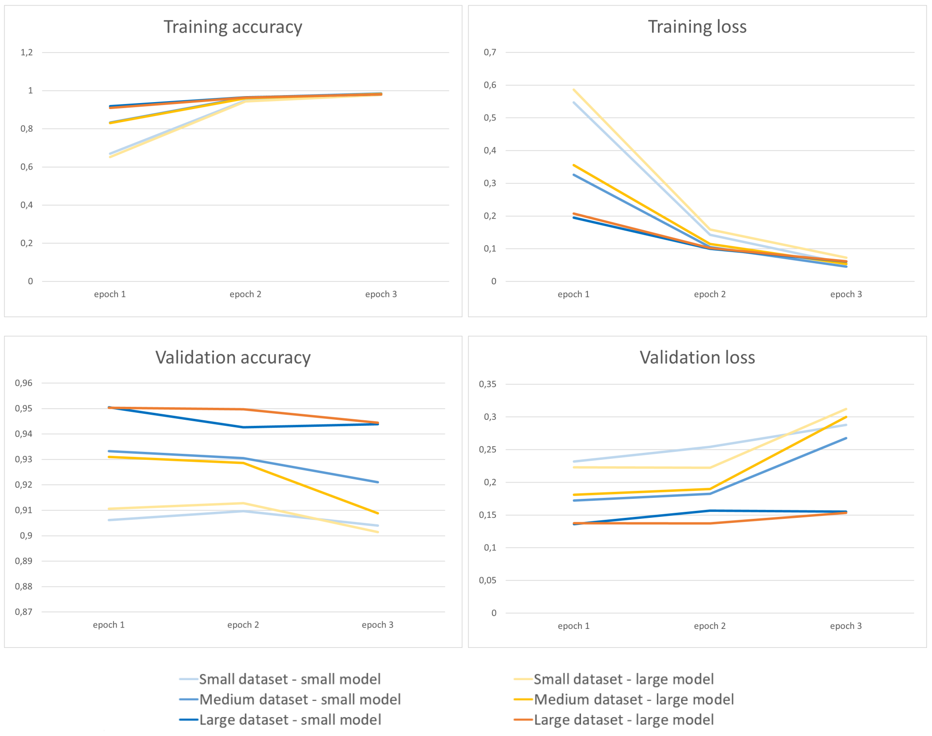


Figure 3: Training results when using 3 epochs

epochs.

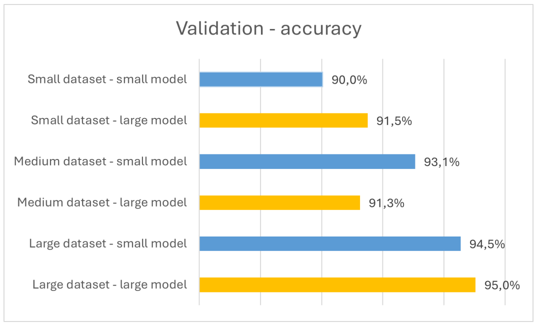
By observing the graphs in Fig.3, it also seems that the performance of the model does not really change based on the size of the model (so small or large models), but is more influenced by the different size of the dataset, with larger ones leading to higher accuracy than smaller datasets.

4.1.2 Prediction step

Having seen that the use of 3 epochs tends to lead to overfitting, we run again the training stage of the six models using 2 epochs and noticed an improvement.
We hence decided to assess the prediction ability of all the models on new, unseen, validation datasets, made of 20K (small), 100K (medium) and 800K (large) words.
Once again, the user is left with the possibility to indicate which parameters to consider. Below the ones used for the small dataset and the small model:

```

# SMALL DATASET - SMALL MODEL
!python predict.py
    --model small
    --test-file prediction_input_validation_20k.csv
    --vocab-folder /content/NLP/saved_vocab/smallModel_smallDataset
    /
    --smallCharCNN-folder smallModel_smallDataset
    --result-file result_validation_20k_small.csv
```

The input file used, “prediction_input_validation_20k.csv”, has been obtained by taking a csv which includes both reviews and labels (“validation_20.csv”), and then making a new file which contains reviews only. After the prediction, the initial labels are joined to the output file, “result_validation_20k_small.csv”, in order to assess accuracy, precision and recall.
The results obtained by this validation step on unseen data can be observed in Fig. 4



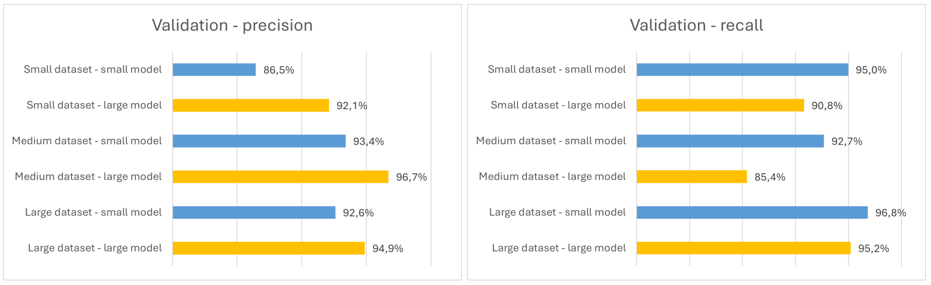


Figure 4: Validation step on unseen data

Observing the graph of the accuracy, we can see that the results are very high for every type of model. Larger models tend to have a slightly better performance, but, more than using a larger model, what seems to have a higher influence on the results is again the use of larger datasets.
Being this a classification task, we also analyzed precision and recall. As known, **precision** assesses the model’s ability to correctly predict positive instances and it is important when **we want to minimize false positives**. It quantifies the proportion of the true positive predictions out of all the positive ones (being both true and false positives). For our models, results are again high for all the six cases, with larger models having a higher precision than smaller ones.

Recall is instead the proportion of the true positive predictions out of all the true positive ones (so both true positives and false negatives), and it is crucial when **we do not want to miss true positives**. In our case, we see that, as predictable given the previous results, for all the three datasets, smaller models appear to have a higher recall than larger ones.

4.1.3 Testing step

As very final step, we test again our six models on three new, unseen, smaller datasets, made of 10K (small), 50K (medium) and 400K (large) words.
The results are almost overlapping with those just seen for the previous step, confirming the performance of the models.

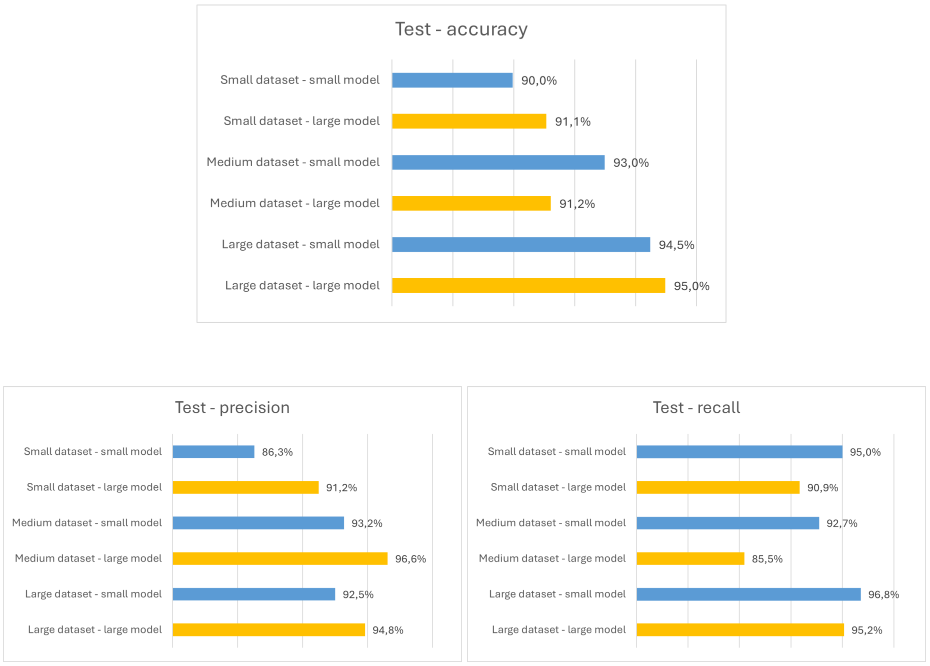


Figure 5: Test step on unseen data

5 Conclusions

The use of Convolutional Neural Networks for this classification task on textual data led to positive results, with very good performances for all the six models. Larger models tend to have higher accuracy than smaller ones, nevertheless, what seems to have the biggest influence on performance is the size of the dataset: had we to choose between increasing the dataset's size and the model's

size, the former would seem the best option to go with, based on our results and if we had to care about quality only. But, as for every Machine Learning task, quality is not the only parameter that we usually consider. Indeed, for this study, computational resources and time had undoubtedly a significant impact. We had to use the resources of Google Colab PRO, since our regular PC (with 8GB RAM, Intel i5-8250U CPU and around 40GB of free memory) was not even able to complete a full training step, and the basic Google Colab was requiring several days to train all the six models.
In Fig.6, we can easily see how the training time would drastically increase from the combination of "small dataset - small model", to the one of "large dataset - large model", going from less than two minutes to almost three hours:

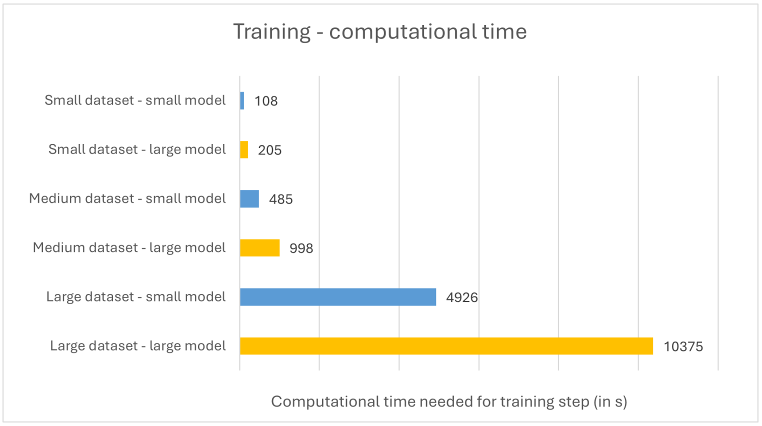


Figure 6: Computational time (in s) needed for the training step using 2 epoch

In conclusion, as always, the trade off between resources and quality depends on the peculiarities of the problem faced. In our case, the application of convolutional neural networks was able to provide very satisfying results already when using a small dataset and a small model and the higher performance reached by the most complex model is not worth the significantly larger amount of time needed to train it.
Further tests could be run to see how accuracy and resources change when editing not only epochs but other parameters too, as learning rate or batch size.

References

[1] Denny Britz. *Understanding Convolutional Neural Networks for NLP*. 2015. URL: `https://dennybritz.com/posts/wildml/understanding-convolutional-neural-networks-for-nlp/`.

[2] Xiang Zhang, Junbo Zhao, and Yann LeCun. "Character-level Convolutional Networks for Text Classification". In: *arXiv:1509.01626* (2016).

[3] Neha Vishwakarma. *What is Adam Optimizer?* URL: `https://www.analyticsvidhya.com/blog/2023/09/what-is-adam-optimizer/#:~:text=The%20Adam%20optimizer%2C%20short%20for,Developed%20by%20Diederik%20P.`.

[4] Machine Learning in Plain English. *Convolutional Neural Network — Lesson 8: Fully Connected Layers and Flattening.* URL: `https://medium.com/@nerdjock/convolutional-neural-network-lesson-8-fully-connected-layers-and-flattening-46638938721`.

To replicate this study

The original dataset can be downloaded from here and can be processed following the steps indicated here. Then, this Colab can be followed, after having uploaded the datasets in it.