UNIVERSITÀ DEGLI STUDI DI MILANO

Academic year 2023/2024

Project of Algorithms for Massive Datasets

Deep learning based on convolutional neural network (CNN)

Antoine Guines

Contents

Introduction

The aim of this report is to give a detailed account and explanation of the final project of the Algorithms for massive dataset course given at the Università degli Studi di Milano by Mr Dario Malchiodi. The project I have chosen to carry out is the deep learning project. Therefore the goal of this project is to build a deep learning algorithms in order to determine the genre of a film from its poster. To do this, we're going to use convolutional neural networks (CNN) that we will train on a large dataset where each poster film is associated to one or several genre. Since the course focus on designing algorithms for massive databases, we will have to process the data and design the model in such a way that it can be used on very large dataset. This report is accompanied by a notebook containing the code of this project, and it is organised into 3 main parts, following the reasoning that was applied during the development of this project. We will go into the details of each of these parts later.

1 Version of Dataset

To carry out this project, we used the dataset named Letterboxd, available on kaggle at the following link : https://www.kaggle.com/datasets/gsimonx37/letterboxd/data. This dataset is under GPL 3 license, so it is a public dataset and we can access these data for free.

Moreover it has a usability score of 10, which is the best possible score. This means that the data are of good quality. More precisely, it tells us that the data are supposed to be complete, from a reliable source, with no licensing or access rights issues, and that it should be in an easy format to process. This dataset therefore seems ideal for our project.

The version of this dataset used to develop the code is the one available at this link on 23 May 2024. At that time, the dataset had been updated 4 months previously, and the expected update frequency is every 1 year. It should therefore not be modified in the coming weeks.

2 Organization of data

The dataset we were working on consisted of a folder called posters/ containing all the posters, and several .csv files containing various information relating to the films. First of all, it's important to point out that this dataset is supposed to contain, according to its description, more than 850,000 films. Moreover, the dataset as a whole is 23.3 GB in size. And as we might expect, the size of this dataset is almost entirely due to the folder containing the film posters, which accounts for 23GB, whereas the .csv files are comparatively very small, with an order of size of 10 or 100 MB. Clearly, it is the management of the images and the way in which the algorithm will process them that is likely to cause us problems.

Since the aim of our project is to determine the genre of a film from its poster, all we need from this dataset is the folder containing the posters and the file genres.csv. During the development of this project, we considered all the data contained in the *posters/* folder and the *genre.csv* file, but we were only running the algorithm on a small random sample of this dataset in order to reduce the calculation time and be able to compute it on my computer.

Personally, to work on this dataset, I loaded it locally onto my computer using the bash command provided at the beginning of the Jupyter notebook. I then kept only the posters/ directory and the genre.csv file, and I developed my code based on this organization.

As explained above, for this project we will only be using the genres.csv file and the posters/ folder containing all the posters. The genres .csv file is organised into 2 columns, one referring to the id, which is the number used to identify the film, and the other referring to the genre of the film. So on each line we have the number of the film and its associated genre. It is important to note that a film can be associated with several genres at the same time. In this case, we find the same film id number on several lines in a row, with the different genres on each line. There is therefore only one genre per line, and at least one line per film, or more if it is associated with several genres. Loading this dataframe with panda gives a total of 990,770 lines. But this does not correspond to the number of films. If we count the number of different ids, we end up with a dataset of 638632 films.

Regarding the posters/ directory, its organization is straightforward: each poster is in .jpg format and named with the associated movie's ID number. The movie ID is the key information that allows us to link the posters, which form the input X of our model, to the genres Y associated with these posters, serving as the labels. Additionally, a quick command executed in the terminal reveals that this directory contains 724,540 posters. Therefore this indicates that there are more posters than distinct movies, so we already know that we will need to clean the data in order to keep only one poster per movie and only the movies that get a poster. This is what we will be talking about in the next part.

3 Pre-processing techniques

As mentionned in the previous part, even if the dataset has a usability score of 10, it seems that we still need to pre-process and clean up the data.

To begin with, after downloading the dataset locally, we load the genres.csv file into memory as a pandas dataframe. We then begin the pre-processing steps. First we add a 'filename' column to our dataset containing the name of the poster associated with each film, which is id_film.jpg. Then, using the *delete_films_without_poster* function, we delete from the dataframe all the films whose filename is not associated with a poster in the directory of the same name. We then apply a function to delete any films whose filename is not associated with a genre. We also decided to remove from the dataframe any films associated with more than 3 genres, as we consider that this data is not necessarily reliable, and that it could provide too much information which could hinder the learning of our model.

Then as mentionned above this project will be divided into 3 main experiments. Firstly, we will consider all films, including those associated with several genres (but no more than 3 !). Secondly, we will consider the simplified case of films associated with a single genre, which should be easier to predict, and with hope for better results. And finally, in a third part, we will consider only the films associated to the 6 more represented genres, and in addition we will balance the dataset in order to have the same magnitude of number of films associated to each genre.

We therefore need to create 3 different dataframes, each one with its own specificities. The first one including all films with less than 3 genres, the second one consisting solely of films associated with a single genre, and the third one considering only a part of the films associated to the 6 more represented genre. We therefore apply several function to the initial dataframe to end up with these 3 dataframe.

We use the *films_with_one_genre* function to build our dataframe composed of films associated to only one genre. And we use the *group_the_genre* function to build the dataframe composed of all our films. This last function gives us a dataframe consisting of just one line per film, with all the genres associated with each film grouped together in a list. We also apply it to the 2 other dataframe in order to obtain the same data structure as the full dataframe and be able to process them in the same way. It's also important to note that this function removes the id column, which we don't need any more as we just need to associate the name of the poster with its genres. This function also mixes the dataframe. To create our last dataframe we use the *process_select_genre* function that we apply to the dataframe containing one genre per films before applying the *group_the_genre* function.

Finally, now that our 3 dataframes are clean, mixed and contain only the necessary information, we can divide them into training, validation and test sets. We then save them in the dataframe/ directory in .csv format. This way, this part only needs to be run once, and later we can load only the training set or the test set, depending on our needs, and this will allow us not to reload the entire dataframe and have to separate it into these 3 sets each time we run our code.

The first pre-processing stage is complete, but to train our model we still need to modify the shape of the dataset. We now need to create a dataset that associates an image with the associated genres. For the moment, we only have the name of the image associated with the genres. To do this I've chosen to use Tensorflow, which offers different types of data adapted to our problem. So before we can train our model we're going to transform the training and validation sets into tensor. More specifically, we're going to create a data input pipeline from our pandas dataframe, to obtain a tensor of type tf.data.Dataset, which is a data type that is much better suited to managing large databases than pandas dataframes. Using this type of data will provide us with a lot of usefull methods that we can use for large database.

So thanks to the *transform_into_tf* function, we get this tf.data.Dataset inout pipeline for our dataset. And now that we have this symbolic tensor, we can apply various operations to it. To this end, the tf.data API offers the tf.data.Dataset.map transformation, which applies a function to each element of the input dataset. Then because each element can be treat separately these operations can be parallelized across multiple CPU, wich can be very usefull when dealing with large database. So in our case we're going to use this map function to apply the *parse_function* on each element. The *parse_function* will call the read_image function and the *one_hot_encoding* function. The *read_image* function will read the image from disk, decodes it as a JPEG, resizes it to 256x256 pixels, and normalizes the pixel values. This ensures that all images have a consistent size and range of values, which is important for training the CNN. After that the *one_hot_encoding* function is called and encodes the labels (the genre) as binary vector.

The one hot encoding allows us to transform the string representation of the genre into binary representation vector. Each genre is associated to a binary vector so in our case we have 19 differents genre so our binary vector will be compose of 19 elements. For instance

'Action' will be represented by : [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
Then :

'Adventure' will be represented by : [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
And if a film is associated with both of these 2 genres then its labels is such that :

['Action', 'Adventure'] will be represented by : [1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

So finally after performing these transformations on the input dataset, each element of the resulting tensor is a pair consisting of a numpy array representing the image with normalised values between 0 and 255, and a vector of 19 components made up of 0 and 1, acting as a label.

Finally, after applying the *map* method, which enables us to apply the *parse_function* to each element of our dataset (thereby calling the *read_image* and *one_hot_encode* functions), we obtain a dataset composed of one poster associated with a binary vector coding for the genre. We then apply the *batch* method to our dataset to group the elements by batches of size BATCH_SIZE. This ensures that when we run our model, the data will be loaded and fed to our model in batches of size BATCH_SIZE.

Thus, this dataset is in an acceptable form for use as input to our deep learning model.

4 Algorithms used

In this project, we used a convolutional neural network (CNN) in order to classify the genres of movie based on their posters. The CNN is a type of deep learning algorithm that is particularly good at processing image data. To create our CNN model we used the TensorFlow Keras library, which provides a lot of tools for creating and training deep learning models. Then let'see how we create it.

Our model is created as a sequence of layers, so we start by instantiate the sequetial() class provided by keras. Then we can create and add the layer to our model.

The first layer in our model is a 2D convolutional layer with 16 filters, each of size 3x3. This layer scans the input image (256x256 pixels with 3 color channels) and detects simple features like edges and textures. The activation function used here is ReLU (Rectified Linear Unit), which introduces non-linearity to help the network learn more complex patterns.

After the convolutional layer, we use a batch normalization layer. Batch normalization normalizes the output of the previous layer by adjusting and scaling the activations. This helps to stabilize and speed up the training process by reducing the internal covariate shift, which means the distribution of each layer's inputs remains more stable as training progresses.

Next, we use a MaxPooling layer with a pool size of 2x2. This layer reduces the spatial dimensions of the feature maps by taking the maximum value from each 2x2 block. This helps to reduce the computational cost and to make the features more robust to small translations.

We then add another convolutional layer with 32 filters, followed by another batch normalization layer and a MaxPooling layer. This process is repeated with a third convolutional layer with 64 filters, another batch normalization layer, and another MaxPooling layer. As we go deeper into the network, the number of filters increases, allowing the model to learn more complex features from the images.

After the convolutional and pooling layers, we flatten the feature maps into a one-dimensional vector. This vector is then passed through a fully connected (Dense) layer with 64 neurons and a ReLU activation function. This layer helps the model to combine the features learned from the previous layers and make more abstract representations.

We also add a Dropout layer with a rate of 0.5. Dropout is a regularization technique that randomly sets half of the activations to zero during training, which helps to prevent overfitting.

Finally, we add another fully connected layer with a number of neurons equal to the number of genres we are trying to predict. Then the activation function used for the final layer depends on whether we are dealing with a multi-label classification problem or a multi-class classification problem. For multi-label classification, we use the sigmoid activation function. This is suitable because the output can have multiple labels, with each neuron representing the probability of a particular genre being present in the poster. For multi-class classification, we use the softmax activation function. In this case, the sum of the probabilities over the output vector should equal one, and the class with the highest probability is chosen as the predicted class.

5 Scale up the solution with data size

In this project, we handle a large dataset of movie posters and their associated genres. To manage and process this data efficiently, we use TensorFlow's data pipeline capabilities. Below, I will explain how this approach can be scale up with an increase size of data.

As mentionned previously we first read the training and validation datasets from .csv files using Pandas. Then, in order to be able to run the code in a reasonable time on my computer, I'm only taking a sample of these dataframes. But if we had a computer powerful enough to run the model on all the data in a reasonable time then we could take everything.

We then transform this data sample into a data type tf.data.Dataset and we create a data input pipeline, which is just a sequence of operations designed to efficiently load, preprocess, and feed data into our model. The source of our data input pipeline is the sample composed of our filename and our genre. Then as explained previously we will creates the final TensorFlow dataset by applying the *parse_function* to each element of the dataset thanks to the *map* method. But now let's really see why this approach is so interesting for managing large datasets.

First by creating a data input pipeline, data will be read and preprocessed sequentially and on-demand, which helps manage datasets that don't fit entirely in memory. Indeed we're not going to load all the images into memory and then feed the model with all these data, we're going to load them by batch, and we will feed the model batch by batch. So for the handling of large datasets, we will avoid memory overload by loading data in small chunks (or batches). You can easily choose the size of these batch applying the method *batch* to your dataset and choose the BATCH_SIZE.

Moreover, if you dispose of a distributed computing environment, then you can distribute data loading across multiple nodes thanks to the *map* method. Indeed you can choose the value of the parameter *num_parallel_ calls* in this method, and so if you have 20 computation nodes at your disposal then you can put it to 20 in order to parallelize the loading of the images over your 20 nodes.

In my case I just put the value *tf.data.experimental.AUTOTUNE* which is a value that automatically optimizes the number of threads that your computer will use for this method, in order to maximize resource utilization.

Finally we also use the *prefetch* method on our dataset, which is still a method from the tf.data.Dataset type, and which is a method employed to load the next batches while the model processes the current batch, reducing wait times and improving efficiency.

So with this approach we can easily handle dataset that doesn't fit in the main memory, since they are not loaded all at the same time. Therefore this allows us to scale up to large datasets without any problem. Moreover if you have a distributed computing environment you should be able to process all your data in a fast and efficient way.

6 Description of experiments

First it is important to say that the notebook provided with this report has been organised in the way I did my reasoning. So we carried out 3 experiments.

The first one on the dataframe made up of all the films, we directly start by trying to solve the multi-label classification problem. In this section we have simply applied the model described in the previous section to our dataset. We use the Adam (or we also tried with sgd) optimizer, which is an efficient gradient-based optimization algorithm. The loss function used is binary cross-entropy, which seems to be the most appropriate one in this case. And as metrics we finally only used accuracy that will give us the number of true prediction over the the number of prediction. We then evaluate the model on the test set and eventually print some prediction to better understand how it works and where it fails.

Then as the results were not convincing, we tried, in a second part to simplify the problem by keeping only films associated with a single genre. So we apply the same approach to the dataset made up of films associated with a single genre. We just use a model with a sigmoid activation function on the last layer instead of softmax. This time the loss function used is categorical crossentropy because there can only be one correct class, and the metric used is categorical accuracy for the same reason.

But still the results does not seems good. So finally, in the last part, we move to an even simpler case. Now not only the films we process are associated with a single genre in the dataset, but we have also reduced the number of genres the model has to choose from. In fact, this dataset only contains the 6 genres that were the most represented in the dataset in part 2. We have also limited the number of films for the most represented genres to 25,000, in order to achieve a balanced dataset. With all these simplifications we hope to obtain better results. Otherwise we apply the same approach that in the second part with the same model.

7 Comments and discussion on the experimentals results

Finally in this last part, let's talk about the experimentals results.

Clearly, in general, the experimental results are rather disappointing. Having no previous experience of machine learning or deep learning, I naively thought that the algorithm would be able to correctly classify the genre of the film from the poster. But obviously this wasn't the case and I quickly realised that.

For the first case, which is by far the most complex, this was perhaps to be expected. Classifying images into 19 classes, knowing that some images may belong to more than one class, is by no means easy. In the end, the result is that in around 35% to 40% of cases we find that among the 3 genres with the highest probability in our predictions, one of these genres is also the film genre. This is obviously not a good result. However, it is easily explained by at least 2 things: - We trained our model on a small sample so as not to take too long (4000 films for the training set and 1000 for the validation set). - But above all, the dataset is far too unbalanced. And so it's likely that even with large samples our model would be biased due to the excessive imbalance in the dataset. On the other hand, as the test set is also unbalanced, this shouldn't put it at that much of a disadvantage either. So at first I thought that since the test set was also unbalanced, and since I wanted the best possible results from the test set, it wasn't a big deal. But in the end, this imbalance probably has too much impact on the learning process, preventing the algorithm from learning the characteristics of the least represented genres. To overcome this, we can under-sample, as I did in the third part, but in this case we lose a lot of data. I also thought of 2 other solutions that I haven't had time to implement. The first is simply to associate weights to each of the classes, so as to weight the learning, and pay more attention to the less represented genres. The second would be to use a tool such as tf.keras.preprocessing.image.ImageDataGenerator, which could allow us to increase our data and therefore perhaps rebalance this dataset a little.

In the second part, I attempted to simplify the problem by considering only movies associated with a single genre, assuming that these films might be more characteristic of their respective genres. However, this approach did not significantly improve the model's performance. While the categorical accuracy improved to around one-third of the cases, it was still not satisfactory. Upon closer inspection, it became apparent that the model tended to predict only the most prevalent genres in the dataset, namely documentaries or dramas.

To address this issue, in the third part, I decided to reduce the number of genres by keeping only the most represented ones and balancing the dataset by undersampling the overrepresented genres. While this approach led to a change in predictions, the categorical accuracy remained around 30%. It is worth noting that training the model on the entire dataset could potentially improve results.

Finally, experimenting with different CNN architectures by adjusting the number of layers, filter sizes, or dense layers did not yield significant improvements in performance. Additionally, the choice of optimizer, such as Adam or SGD, and normalization techniques like batch normalization, could play crucial roles in handling large datasets across multiple epochs.

In summary, exploring various data preprocessing techniques, dataset balancing strategies, and model architectures, along with optimizing hyperparameters, are crucial steps in improving the performance of the classification model on large-scale datasets.

Conclusion

Therefore, this project concludes without a significantly compelling result despite the various attempts made. However, it has provided me with valuable learning experiences in a domain that was previously unfamiliar to me. In particular, I discovered the powerful tools offered by TensorFlow, which can be very usefull in handling large databases effectively.