UNIVERSITÀ DI PISA

Department of Information Engineering
MSc Computer Engineering

Intelligent Systems

Optical Music Recognition

Federico Cristofani

Academic Year 2022/2023

Contents

1 Introduction **4**
- 1.1 Optical Music Recognition . . . 4
- 1.2 Dataset . . . 4
  - 1.2.1 Overview . . . 4
  - 1.2.2 Analysis . . . 5
- 1.3 Related Works . . . 6
- 1.4 Work environment . . . 8

2 Model **9**
- 2.1 Architecture . . . 9
  - 2.1.1 Convolutional Neural Network . . . 9
  - 2.1.2 Recurrent Neural Network . . . 10
  - 2.1.3 Transcription Layer . . . 10
- 2.2 Hyperparameters . . . 10
  - 2.2.1 Custom model . . . 10
  - 2.2.2 Pretrained model . . . 11

3 Model Training **12**
- 3.1 Data . . . 12
- 3.2 Training model . . . 13
- 3.3 Configuration . . . 13
- 3.4 Results . . . 14

4 Model Evaluation **15**
- 4.1 Metrics . . . 15
- 4.2 Results . . . 16

5 Conclusions **18**

List of Figures

1.1 Optical Music Recognition task 4
1.2 Incipit example 5
1.3 Incipit length distribution 5
1.4 Symbol distribution 6
1.5 CRNN architecture 7

3.1 Training model structure 13
3.2 Learning rate scheduling for custom model 14
3.3 Training and Validation Loss Over Epochs 14

4.1 SER-LD metrics comparison 15
4.2 Models evaluation scores 16
4.3 Error rate per Symbol custom model 17

List of Tables

2.1 Custom Model Hyperparameters 11
2.2 Pretrained Model Hyperparameters 11

1. Introduction

1.1 Optical Music Recognition

The Optical Music Recognition (OMR) is an application of recognition techniques to musical scores, to encode the musical symbols into a digital format. Basically, an image containing a score is fed into a deep neural network, which produces as output the sequence of symbols (notes, clefs ...) contained within the input image. The OMR problem although it has some similarities with the better known OCR (Optical Character Recognition) hides additional difficulties, since context information determines the meaning associated with each symbol. Context dependence means that even if two symbols are represented through the same glyph, their semantics may not be the same, e.g. a note depends on its position in the staff, so recognition technique must take this aspect into account. Note that the goal is limited to recognizing the sequence of symbols in the musical score, not locating those symbols within the image.

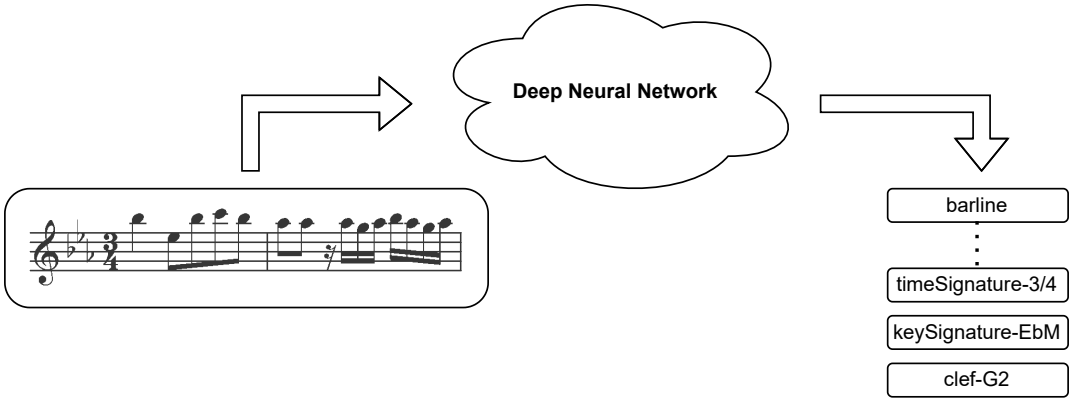


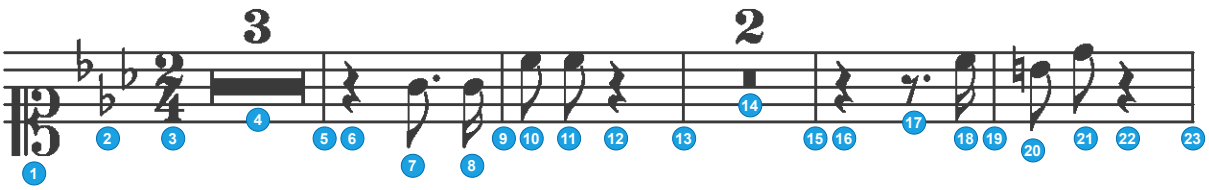
Figure 1.1: Optical Music Recognition task

1.2 Dataset

1.2.1 Overview

The data used came from the PrIMuS [2] (Printed Images of Music Staves) dataset, which consists of more than 80,000 incipits, i.e., the first bars of a musical work.
For each incipit the dataset provides a dedicated folder in which can be found several files, but the only ones of interests to this project are:

- **Graphical representation**, image with the rendered score.
- **Semantic encoding**, sequence of symbols in the score with their musical meaning, used as ground truth for the training process.



1. clef-C1
2. keySignature-EbM
3. timeSignature-2/4
4. multirest-3
5. barline
6. rest-quarter
7. note-G4_eighth.
8. note-G4_sixteenth
9. barline
10. note-C5_eighth
11. note-C5_eighth
12. rest-quarter
13. barline
14. multirest-2
15. barline
16. rest-quarter
17. rest-eighth.
18. note-C5_sixteenth
19. barline
20. note-B4_eighth
21. note-D5_eighth
22. rest-quarter
23. barline

Figure 1.2: Incipit example with relative semantic symbols

1.2.2 Analysis

Below some statistics inferred from the dataset are reported.

Incipit length Incipit length is quantified as the number of symbols appearing in the associated image, i.e. the length of the sequence according to the semantic encoding. For instance, the incipit's length in *Figure 1.2* amounts to 23 symbols. The maximum length of incipits is 58 and it's important to highlight because as will be explained in section 1.3 such value plays a role in the model definition. In addition, it is interesting to observe the average length of the incipits to spot any high skewness that might affect the final model score. The best method to observe this is to draw a simple histogram of the incipit lengths, from which it can be seen at a glance that the distribution appears quite symmetrical, as also stated by statistics reported in the plot, even if the spread around the mean is not negligible.

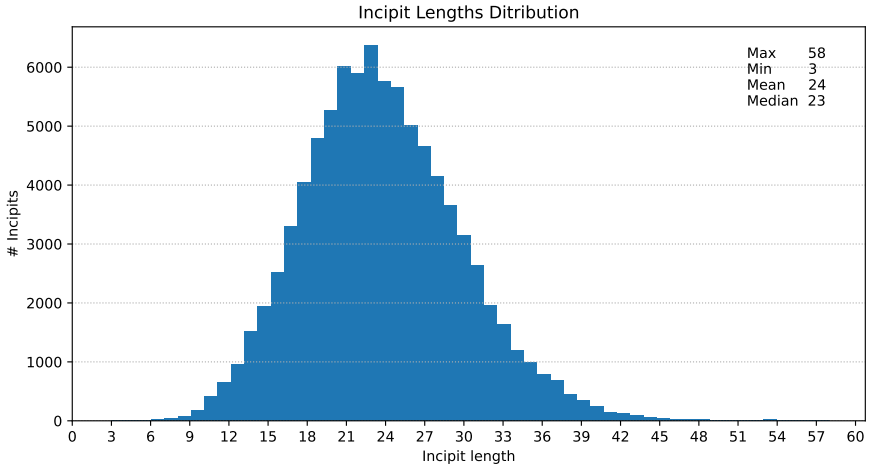


Figure 1.3: Incipit length distribution

Symbol occurrences The symbol occurrences determine how many times each symbol appears in the dataset. First, it is important to report the total number of symbols in the dataset, which is 1781.

Counting the number of occurrences for each symbols results in a very skew distribution, caused by many symbols with very few occurrences and other symbols much more frequent. Moreover one symbols, the *barline*, outperforms all the others in terms of occurrences. The distribution can be visualized in a box-plot having on the logarithmic y-axis the number of occurrences. The box-plot is built choosing as whiskers the 1^{st} and 99^{th} percentiles and the IQR, median and outliers for the other componenents.

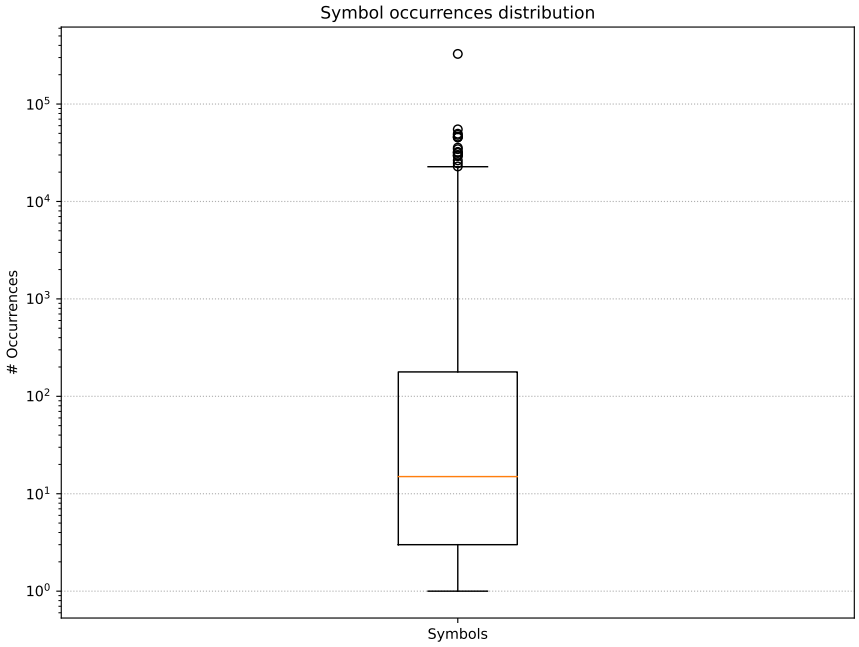


Figure 1.4: Symbol distribution

The plot spans over 5 order of magnitudes, demonstrating the high variance in the number of occurrences, that may lead to problems in recognizing "rare" symbols. The problem is related to the dataset and is hard to solve without adding more data to balance the occurrences. Moreover, such skewness may be entirely natural and thus present in real-world data as well. In any case, the penalties on overall scores should not be so relevant because the rare symbols are obviously contained in a few amount of incipits.

1.3 Related Works

The architecture that will be presented in the following is inspired by some previous works. In [7] the CRNN (*Convolutional Recurrent Neural Network*) network architecture is introduced to solve OCR problem, while in [1] such architecture is slightly modified and applied to OMR problem. The latter also introduce the *Primus* dataset, the one above described and used in this project. Some details about CRNN architecture are reported in order to understand how the process of sequence recognition from images works. Moreover, details about the training process will be given because it has an impact both on data preparation and model definition.

Architecture The CRNN architecture is a composition of a CNN used for feature extraction and a RNN used for sequence recognition. It is able to produce an output sequence starting from an input one in an end-to-end fashion, more precisely in the OMR case starting from the input image, considered as a sequence of columns, the sequence of output symbols is produced.

The architecture can be broken down into three main components:

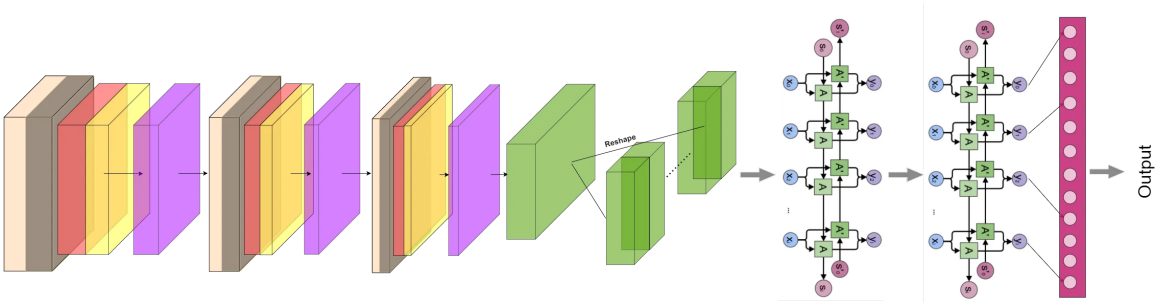


Figure 1.5: CRNN architecture [3]

- **Convolutional Neural Network**, the input image is fed into a CNN in order to extract relevant spatial features that will reshaped in such a way to build a sequence of feature vectors, scaling down from 3 dimensions to 2 dimensions.
- **Recurrent Neural Network**, the feature vectors represent the inputs to recurrent block that produces as output a sequence of symbol predictions. For each input vector a symbol prediction is generated an passed to the last network component.
- **Transcription layer**, the symbol predictions are finally transformed into probability distributions by a dense layer equipped with softmax activation function. The layer has many output neurons as the number of available symbols in the dataset, plus one that represents a "blank" character for those vector processed by recurrent block that doesn't contain any symbol. The sequence of probability distributions over the available symbols produced needs to be decoded to obtain the actual sequence of symbols.

The network has a constraint on the sequence it is able to generate, in particular is not possible to produce a sequence longer than the number of feature vectors. This places some limits on the CNN, because it must produce as output a volume suitable to be reshaped into at least 58 feature vectors, that is the maximum incipit length.

Training Both CNN and RNN can be trained via backpropagation algorithm and SGD (*Stochastic Gradient Descent*) optimizer. The peculiarity of CRNN training resides in the choice of CTC (*Connectionist Temporal Classification*) as loss function, that relives from the burden to provides positional information about each symbol in the training images. In principle the recurrent network would perform a per-frame classification to produce the output sequence, requiring a label for each frame, but it is in contrast with the initial assumption of producing sequence directly from the raw image.

Basically the CTC loss function provides a feasible mechanism to optimize, at least locally, the conditioned probability:

$$\hat{y} = argmax_{y \in \Sigma} P(y|x) \tag{1.1}$$

The above equation consists in find the $\hat{y}$ that maximizes the conditioned probability of obtaining the sequence y, given the input image x. In simpler terms objective is to maximized the probability to obtain the right sequence starting from the input image. The Σ represents the set of available symbols.

Note that the CTC is applied only during the training process, once trained the sequence is directly generated decoding the output from transcription layer. From a practical point of view the CTC loss function, aside the predicted and ground truth sequences, also requires both sequence lengths, requiring a slightly modification of the input to the model during the training.

More details about CTC loss function can be found in [6].

1.4 Work environment

The work environment deserve a special mentioning due to the limited available resources that had an impact on some decision taken during the design and training of the neural network models. The project has been developed on a standard Google Colab environment that offers the following resource:

- Dual-core CPU
- About 12 GB of main memory
- About 78 GB of secondary storage
- Nvidia Tesla T4 GPU for limited time-sessions
- Only interactive sessions, no background operations admitted

The time-sessions GPU usage and only-interactive operations limit represented the main issues that prevented the possibility to run long operations needed during model design and training.

The deep learning framework adopted is Keras along with Tensorflow as the back-end.

2. Model

2.1 Architecture

The models built for this project propose two different implementations of the CRNN architecture above described, specifically two different approaches for feature extraction by CNN:

- **Custom CNN**, built and trained from scratch
- **Pretrained CNN**, tweaked via fine-tuning

2.1.1 Convolutional Neural Network

In the original work [1] only the height of input images is fixed, while the width is variable. Although this approach may improve the capacity of the network in the extraction of relevant features, the control on the size of training-set is limited and this represent a risk for a resource-limited environment. Therefore, the input shape for the models developed for this project is fixed a priori and depends on the CNN in use.

Custom model

The CNN built for the custom model presents a straightforward structure, composed of five sequential blocks, each consisting of five layers ordered as follows:

- **Convolutional layer**, an increasing number of filters are applied, doubling at each level from 32 up to 512 in order to increase the level of feature abstraction.
- **Batch normalization layer**, the normalization introduces many benefits, in particular is able to improve the training convergence and stability, speeding up the entire process.
- **Activation layer**, the activation function is applied after normalization as general rule.
- **Max pooling layer**, the maximum is chosen as pooling operation because of its ability to maintain most relevant features and patterns during the downsampling process.
- **Dropout layer**, the dropout allows to prevent overfitting and stabilize the training process, maintaining at the same time a good network capacity needed to address the problem.

The custom CNN accepts input shape of 32x256x1, i.e. greyscale images having a width of 256 pixels and a height of 32 pixels. In principle, pooling layers should halve the size of feature maps, but starting from a 32x256x1 image the final size would be 1x8x512, thus ignoring the constraint on the final features shape. To guarantee the respect of such constraint only the first two pooling layers halves both dimensions, while the remaining present an asymmetric pool-size of 2x1, halving only the first dimension. The introduction of this approach allows to have a final features size of 1x128x512, that can be reshape into 128 feature vectors each of 512 elements.

Pretrained model

The CNN used by the pretrained model is the well known VGG-19, trained on ImageNet [5], a dataset built to address object recognition tasks. The CNN is a variant of the family *Visual Geometry Group*, introduced in 2014 [8], composed of 19 layers. The layers include 3 fully connected ones that are removed for this project, along with the last max pooling layer to avoid excessive reduction in the length of the final features. The input images are sized to 32x1024x3, a shape that derives from the following considerations:

- The VGG-19 model, at least in the adopted framework, accepts only RGB images having a size at least 32x32.
- The size must be increased to prevent the final volume of features from being too small.

The adopted size allows to obtain a final feature map of 2x64x512 that can be reshaped into 128 feature vectors each of 512 elements, as the custom CNN.

2.1.2 Recurrent Neural Network

The recurrent network follows the guidelines specified in [7], adopting a stacked network of bidirectional LSTM(Long-Short Term Memory) blocks. The choices are justified as follows:

- **LSTM blocks**, compared with classical RNN, LSTM have the ability to address the *"vanishing gradient"* problem, which limits the ability to model long-term dependencies. It is worth mentioning that such dependencies are common in the OMR problem, for example, all notes on the staff are strictly dependent on the initial clef.
- **Bidirectional property**, the motivation comes from the fact that the sequence recognition, especially from images, obtains benefits by using information from both past and future contexts.
- **Stacked network**, as stated in [7], stacking multiple blocks is possible to build a deep LSTM able to guarantee higher level of abstraction, resulting in better performance.

2.1.3 Transcription Layer

In both models the transcription component is a simple fully connected layer, composed by the required number of neurons (1782), equipped with softmax activation function to transform the symbol predictions produced by recurrent component into probability distributions over the available symbols. The final decoding, not included in the models themselves, consists of taking the most probable symbol for each distribution, omitting any empty symbols. More in details, the CTC loss function allows to perform a greedy decoding, i.e. when the symbol predicted from a feature vector is the same as the previous one, it is assumed that they represent the same instance and only one is concatenated to the final sequence. Two identical symbols can be concatenated only if they are separated by the "blank" symbol.

2.2 Hyperparameters

The models present many configurable hyperparameters and the configuration has a significant impact on the final results. In principle the best configuration could be obtained via a rigorous approach, adopting one of the available hyperparameter optimization techniques, such as *"hyperband"*. Unfortunately, as already mentioned, the project is developed in a environment with limited resources, so even if the developed code supports the tuning of hyperparameters, due to problems related with lack of resources the configuration is established following heuristic approach based on trial and error and experimental observation. In the following tables a resume of the main hyperparameters is reported for both model. Note that for the omitted hyperparameters the default value provided by the implementation framework is to be considered.

2.2.1 Custom model

The custom model hyperparameters configuration is resumed in the following table:

Component	Hyperparameter	Value
CNN	Activation function	ReLU
	# Filters	32 - 512
	Kernel-size	3x3
	Dropout	0.30
LSTM	Number of Units	256
	Dropout	0.45
Fully connected	Activation function	Softmax
	# Neurons	1782

Table 2.1: Custom Model Hyperparameters

2.2.2 Pretrained model

The pretrained model hyperparameters configuration is resumed in the following table:

Component	Hyperparameter	Value
LSTM	Number of Units	256
	Dropout	0.35
Fully connected	Activation function	Softmax
	# Neurons	1782

Table 2.2: Pretrained Model Hyperparameters

For the CNN hyperparameters refer to VGG-19 model [8].

3. Model Training

3.1 Data

The data in the Primus dataset need preliminary preparation before being fed into the models.

Loading The first step is to load the images and associated sequences used as ground truth in the training process. The large amount of sample prevent from the possibility to maintain in main memory all the images, so the solution is to load the images on-demand during the training process, paying for a longer training time. The initial operation is to retrieve, in random order, all the image paths along with the symbol sequences and their lengths. The sequences must also be encoded into a numerical format to be processed by the network. The encoding and decoding process is based on a dictionary structure that maintains the correspondence between symbol and code (number), initialized randomly once all sequences are loaded into memory. The random initialization prevents the possibility of generating patterns that may interfere with network training. Once all the sequence are encoded they must be padded to obtain a common length, equal to the one of the longest sequence (58). The padding value is the higher code plus one. The final result is a shuffled list of tuples composed by image path, padded-encoded sequence and original sequence length.

Splitting Once the initial list is ready, it is split into 3 sets: training set, validation set and test set. The split operation is carried out before any further preprocessing because the operation on training/validation sets are different from the ones for test set. The sets, encoder/decoder structure, along with other information are dumped into disk to allow a faster data preparation in future training sessions.

Preprocessing The sets, once prepared or loaded from disk, are fed into a pipeline, consisting of a sequence of operations that transform the data into a format suitable for the models. The pipeline is built using the Tensorflow Dataset API, which allows lazy evaluation, performing operations only when needed, i.e. during training or evaluation processes. Regarding training and validation sets the sequence of operations is responsible to produce all the data needed to apply CTC loss function, composed of four inputs and one output:

Inputs:

1. Image
2. Padded-encoded sequence
3. Feature vector length
4. Original sequence length

Output:

1. Target CTC loss value

The first stage of the pipeline loads, decodes, and resizes the image and returns it along with the encoded-padded sequence, number of features vectors (fixed to 128 by model construction), original sequence length, and the target CTC loss value (always 0). Next stages optimize the processing time pre-fetching and caching the data and batching them. The training set is also shuffled at each iteration to improve training process. Regarding the test set the initial stage is simpler because only needs to return processed image and padded-encoded sequence, i.e. input and target output of the model.

Note that the two models have different input size, so they needs also different pipelines.

3.2 Training model

The models are built to accept images as input, but the CTC loss functions, as mentioned above, needs other values as well. To overcome this issue a special training model is developed to train the original models. It is composed of:

1. Model to train
2. Input layer accepting padded-encoded sequences, characterized by many input neurons as the longest sequence (58)
3. Input layer composed by a single neuron accepting the number of feature vectors (always 128)
4. Input layer composed by a single neuron accepting the original sequence length

The training model is able to accept all the values required by the loss function, that are passed to a Lambda layer that implements the CTC loss. The output of Lambda layer is itself the loss value, so it can be directly compared with the target value that is obviously zero.

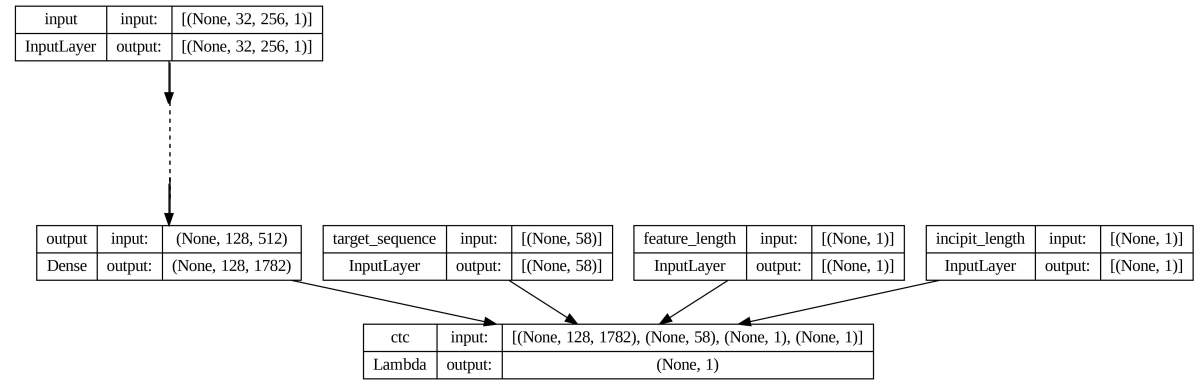


Figure 3.1: Training model structure

3.3 Configuration

The training process is slightly different from the one described in [1] because the ADAM (*Adaptive Moment Estimation*) optimizer is adopted. It is a variation of the classical SGD, based on adaptive estimation of first-order and second-order moments that allows faster and more stable convergence. Moreover to obtain the maximum trade-off between convergence stability and speed the learning rate is dynamically decreased during the training following a predefined scheduling, based on exponential decay. The scheduling is slightly different for the two models:

- **Custom model**, the learning rate decay is initially very slow, but after a few epochs the decay increases significantly.
- **Pretrained model**, the learning rate is initially fixed at an high level to speed-up the training of the custom network attached to the pretrained one, but when both parts are jointly trained the learning rate starts to decay after each epoch.

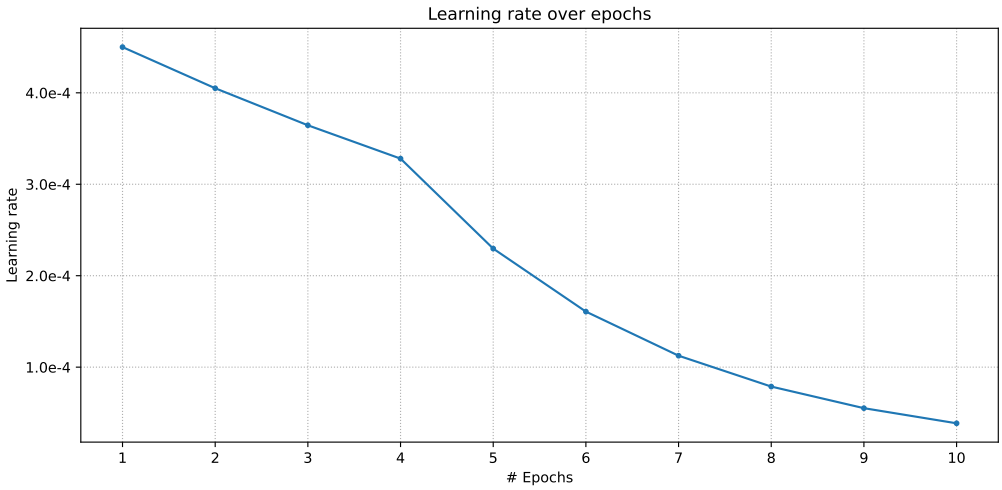


Figure 3.2: Learning rate scheduling for custom model

The overfitting problem is monitored adopting an early-stopping mechanism and the entire process is tracked in order to save partial results and the best model. Finally, both models were trained for a total of 10 epochs, although a larger number of epochs could produce better results, resource limitations prevent the possibility of a longer training process.

The training of pretrained model deserves additional attention due to the already trained CNN, even if using another dataset and for other purposes. To obtain good results without destroying the acquired capability of VGG-19 a fine-tuning technique is adopted. The model is first trained for some epochs by keeping the CNN layers frozen, and then for the remaining epochs all network parameters are jointly updated.

3.4 Results

Both models were able to learn for 10 epochs without overfitting and producing better results after each epoch. The training and validation loss over epochs is reported for both models in the following plots.

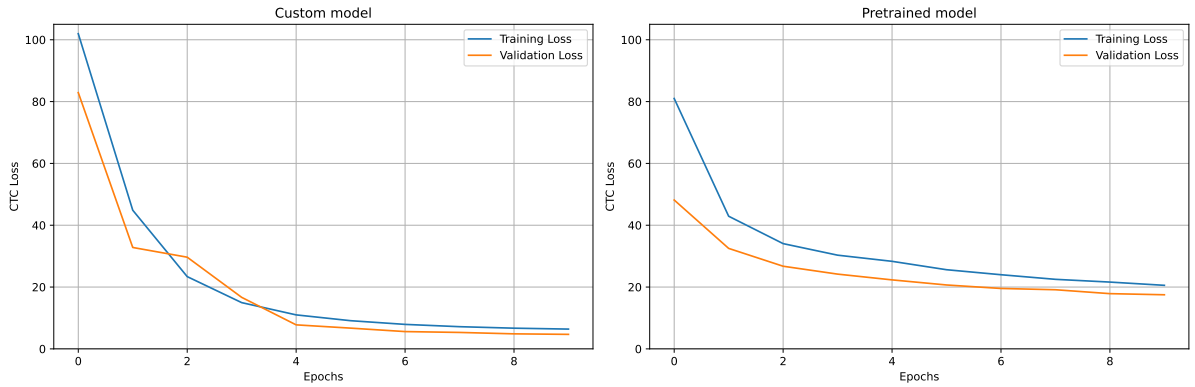


Figure 3.3: Training and Validation Loss Over Epochs

From the plots is possible to see at a glance that the custom model converges much faster than the pretrained model. However to draw more accurate results is necessary to evaluate the models computing adequate metrics over the test set.

4. Model Evaluation

4.1 Metrics

Choosing the correct metrics during the evaluation is essential to obtain an index that actually reflects the goodness of the model. Moreover it is important to select metrics that can be easily interpreted in order to asses if a model is generally good in absolute terms and not only if is better than another. Focusing on the OMR problem, the evaluation can be reduced to a comparison between sequences, i.e. between the sequence inferred from the image and the ground truth provided by the test set.

The first metric to be discussed is the accuracy, expressed as the number of correct sequences predicted over the total sequences in the test set. It may seem a good choice due to its trivial interpretability, however it tends to produce misleading results. The problem with accuracy arise from the fact that a sequence is composed by many symbols and each of them is a prediction provided by the network. If at least one of the symbols is wrong, the entire sequence will be wrong and the accuracy will drop very quickly.

Note that in more rigorous scenarios, even applied for OMR, strict metrics may considered necessary. For this project such strictness is not taken as requirement, so more relaxed metrics are considered. The ones taken into account are:

- **Symbol Error Rate (SER)**, trivially computed as the number of correct predicted symbols, also considering the position, over the sequence length.
- **Levenshtein distance (LD)**, sometimes referred as *Edit distance*, is defined as the number of basic operations (insertions, deletions or substitutions) needed to transform a sequence into another, thus representing the closeness between two sequences.

The SER still have some rigidity because in case of missing/extra symbol all the subsequents will be considered wrong, while on contrary the LD is much more robust to these type of errors. The LD metric deserves further observation because the score produced is not normalized as SER, i.e. a percentage, but extends over a range between 0 (no errors) and the length of the truth sequence (all wrong symbols). To produce a more interpretable result, each score is normalized by dividing it by the length of the truth sequence, resulting in values in the range [0,1].

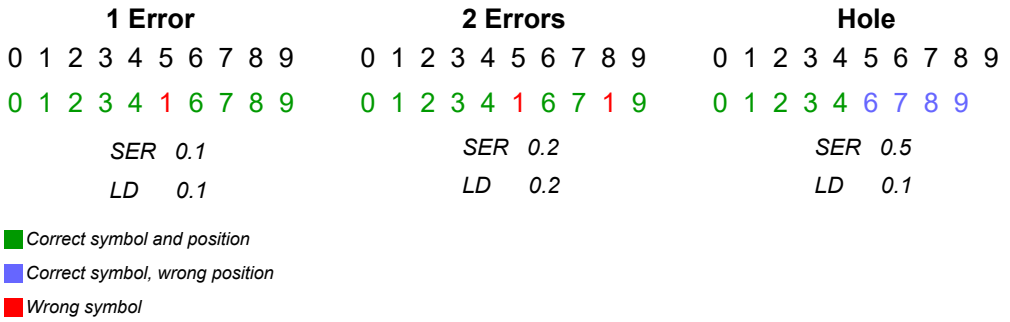


Figure 4.1: SER-LD metrics comparison

Note that normalizing the metrics by considering the length of the ground truth sequence as reference may produce scores higher than one in the case of a predicted sequence longer than the correct one. This situation is not likely enough to influence the overall result, so it is simply ignored.

4.2 Results

The results achieved computing the selected metrics over the test-set mirrors training results, i.e. the custom model outperform the pretrained model. With reference to SER the custom model achieve a score of 0.048, instead the pretrained model only achieve a score of 0.197. The same consideration are applied to LD, for which the custom model produce a score of 0.073, while the pretrained model a score of 0.333. As expected the LD has a better score in both models, due to its flexibility. The scores are plotted on a bar chart to appreciate the difference between the two models, the blue bars represent SER and LD of custom model, the orange ones the metrics of the pretrained model.

Figure 4.2: Models evaluation scores

The metric scores are a good index to summarize the goodness of the models, however, can be interesting analysing more in details the errors produced. In particular, some conclusions can be drawn observing the error rate per symbol (ERPS), computed as the number of times a symbol is incorrectly recognized out of the total occurrences of such symbol in the test-set. The following analysis is reported only for the custom model because is the one that deserves better attention due to the good results achieved.

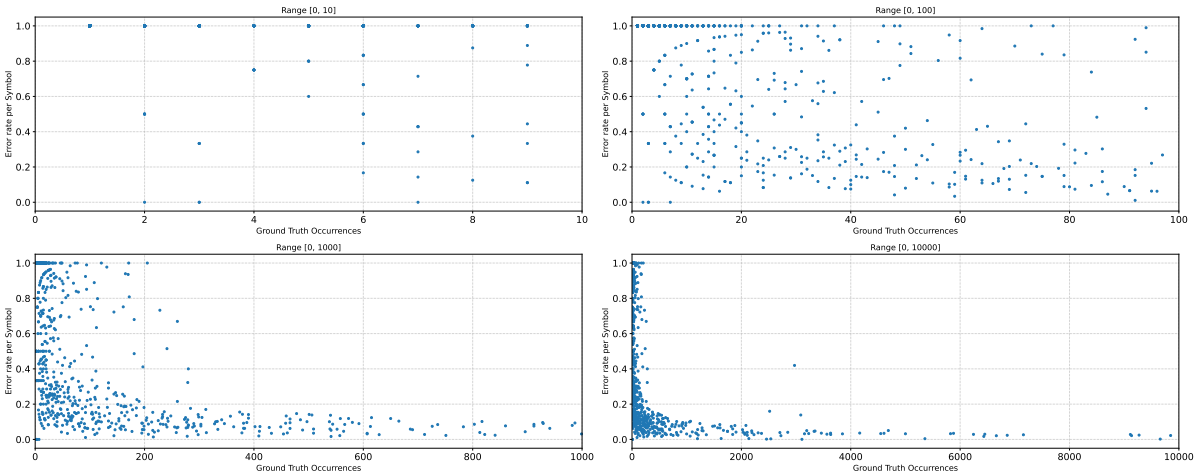


Figure 4.3: Error rate per Symbol custom model

The above scatter-plots show the ERPS for each symbol in the dataset, ordered by occurrences, i.e. each symbol is represented by point of coordinates [*occurrences, ERPS*]. To appreciate the trend a progressive zoom is reported, starting from left to right. The trend is quite visible and shows that the error rate drops quickly with the number of occurrences of the symbols, thus symbols with a low number of occurrences are those with the highest probability of being recognized incorrectly. Moreover from the first zoom is possible to notice that the low-occurrences symbols are randomly predicted, from the fact that all the possible y-axis position are occupied. Starting from the second zoom the trend is visible and in the second and third zooms the long flat tail is proof of such trend. This result is quite obviously because the low-occurrences symbols appears few times, or even none, in the training-set so the network is not able to learn about their recognition. The problems is not in the model itself, but in the skewness of the symbol distribution of the original dataset.

5. Conclusions

The developed models although implementing the same high-level architecture (CRNN) present very different results. The custom model converges to acceptable results very quickly, while the pretrained model is much more tricky to train. This leads to poor scores for the pretrained model, even if it is build on top of VGG-19, a robust CNN pretrained on a large dataset. The advantage given by the custom CNN derives from the fact that is built keeping in mind both the expected input and the needed output, i.e. feature vectors of a predefined shape. Moreover custom CNN has a smaller size compared to the VGG-19 (about 1.5M vs 20M) and includes some dropout layers, so the learning process tend to be easier with the moderate amount of available data.

In general may be possible to achieve better results reviewing some forces choice:

- **Epochs limit**, training the models on more epochs may guarantee better results, especially for the pretrained model. Some tuning may be necessary to exploit such possibility, but the absence of overfitting during the training suggest to tries longer training processes.
- **Fixed number of feature vectors**, fixing the width of input images may be not a good choice. First of all the symbols may be distorted and this hardly recognized, moreover no relation between features shape and the number of symbols in the images (typically directly proportional with the image length) is maintained.
- **Hyperparameter tuning**, a more accurate selection of the hyperparameters may improve the model results. The adoption of more rigorous techniques on an adequate searching space might produce a different set of hyperparameters respect to the one fixed in this project, improving the final results.
- **Larger dataset**, a larger dataset may produce better results at the cost of more resources needed. In particular may be exploited some data augmentation techniques, not implemented for this work, artificially increasing the amount of data without requiring other dataset to be merged with the one used.

Although there are some limits the adoption of CRNN for OMR problem represent a cost-effective solution, allowing to reach acceptable results without adopting very complex solution that may not fit in some limited resource environment.

All the code implementing the described models is freely available in the following repository [4].

Bibliography

[1] Jorge Calvo-Zaragoza and David Rizo. "End-to-End Neural Optical Music Recognition of Monophonic Scores". In: *Applied Sciences* 8.4 (2018). ISSN: 2076-3417. DOI: 10.3390/app8040606. URL: https://www.mdpi.com/2076-3417/8/4/606.

[2] Jorge Calvo-Zaragoza and David Rizo. *Primus Dataset*. https://grfia.dlsi.ua.es/primus/. 2018.

[3] Chandra Churh Chatterjee. *An Approach Towards Convolutional Recurrent Neural Networks*. https://towardsdatascience.com/an-approach-towards-convolutional-recurrent-neural-networks-a2e6ce722b19. 2019.

[4] Federico Cristofani. *OpticalMusicRecognition*. https://github.com/federico-cristofani/OpticalMusicRecognition. 2023.

[5] Jia Deng et al. "Imagenet: A large-scale hierarchical image database". In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255.

[6] Alex Graves et al. "Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural 'networks". In: vol. 2006. Jan. 2006, pp. 369–376. DOI: 10.1145/1143844.1143891.

[7] Baoguang Shi, Xiang Bai, and Cong Yao. "An End-to-End Trainable Neural Network for Image-Based Sequence Recognition and Its Application to Scene Text Recognition". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39.11 (2017), pp. 2298–2304. DOI: 10.1109/TPAMI.2016.2646371.

[8] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015. arXiv: 1409.1556 [cs.CV].