Simple CRNN Implementation for OCR in PyTorch

Mirko Sommer
Department of Computational Linguistics
Heidelberg University, Germany
Hauptseminar: Introduction to Neural Networks and Sequence-to-Sequence Learning
GitLab: https://gitlab.cl.uni-heidelberg.de/sommer/pytorch-ocr/
sommer@cl.uni-heidelberg.de

Abstract

This project integrates a simple CRNN (Convolutional-Recurrent-Neural-Network) model for optical character recognition (OCR) in PyTorch[1]. Training and testing were performed on multiple datasets. The used dataset, implementation, results, and discussion of the experiments can be found in this report.

1 Introduction

The general goal of my project is to perform OCR in PyTorch.
OCR describes recognizing and transcribing typed, handwritten, or printed text in images into computer-readable text.
Early on in the research phase of my project, I stumbled upon the CRNN model architecture (for more detail see 3.2). I was intrigued by the good results that others were getting with this architecture, so I moved forward with it. At first sight, the implementation seemed pretty straightforward, but as I started programming one problem after another emerged.
To pull the project off I needed more concrete examples of how I could implement a CRNN model in PyTorch so I found a couple of projects 5 that tried to implement this architecture in PyTorch. Most of them were pretty complex, but they gave me a nudge in the right direction. However, it still took a long time to understand the code of the other projects, because the documentation was sometimes lacking or the implementation was unnecessarily complex.
My goal is to decrease the code's complexity and implement a more straightforward approach to implementing a CRNN model with a lot of documentation.

[1]https://pytorch.org/

2 Dataset

The datasets used in the experiments are the Captcha Dataset [6] and the IAM Handwriting word Dataset [3] (in the following referred to as IAM dataset). All samples of the captcha dataset were used, whereas only a chosen number of samples of the IAM Handwriting Dataset was used because training would have taken too long on the full dataset. A 70/20/10 train/test/val split ratio was used. Table 1 shows the data splits of both datasets.

Split	Samples Captcha	Samples IAM
Training	749	1750
Validation	214	500
Testing	107	250
Total	1070	2500

Table 1: Datset statistics

3 Implementation

Details of the model architecture, used criterion, implementation of the project, and problems I encountered during the project can be found in the following section.

3.1 Data Processing

As a first step, the raw datasets are split into training, testing, and validation datasets. The image files are now directly in every directory.
After that, the captcha dataset is loaded with the class *CaptchaLoader*. It loads all the images in a directory and transforms them into a tensor for each image, which will be used for training the model. The gold label of the text in the image is extracted from the file name of each image.

The IAM dataset is loaded with the class *IAMLoader*. It also loads all the images in a directory. Every image has a different size, but the model needs them to be all the same. They are all resized and padded to the same size so that each image starts at the left side of the image. As a next step, they are transformed into tensors. The gold label of the text in the images is extracted from a given information file, which is given from the IAM dataset.

Here are some samples of the captcha and the transformed IAM dataset. The blue frame is only to visualize the border.

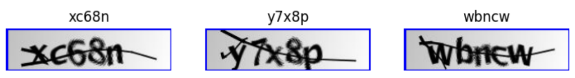


Figure 1: Samples captcha

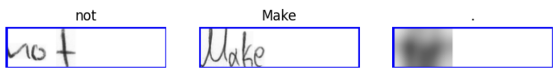


Figure 2: Samples IAM

3.2 Model Architecture: CRNN

The Convolutional Recurrent Neural Network (CRNN) architecture combines convolutional Neural Networks (CNNs) with Recurrent Neural Networks (RNNs). This architecture was first introduced by Shi et al. [5].

The CNN part of the model is responsible for extracting features from the input image. First, the image is fed into an input layer. After that convolutional layers apply filters to the image to extract features. Pooling layers are used in this step to downsample the feature maps. This is used to reduce the spatial dimensions while keeping important features.

I experimented with different CNN architectures:

- a simple CNN without explicit pooling layers, but using the stride parameter of the convolutional layers to effectively reduce the spatial dimensions of the feature map

- a more complex CNN; using the first three convolutional layers of the resnet18[2] model architecture with and without pre-trained weights. This architecture uses max-pooling in combination with the stride parameter to reduce the spatial dimensions of the feature map.

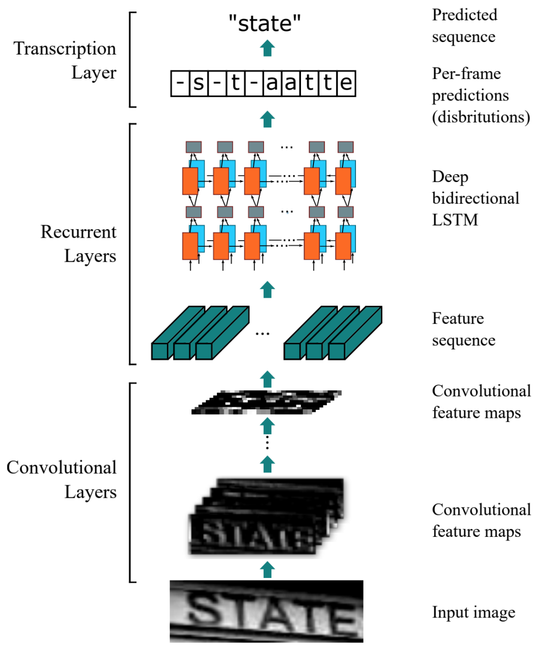


Figure 3: The network architecture. Figure from Shi et al. [5]

The RNN part is designed to handle sequential data. After the CNN extracts the feature map, this map is reshaped into a sequence. Each vector in this sequence corresponds to a column of the feature map and represents features at a particular position in the text. After that, they are passed through recurrent layers. These layers are bidirectional Gated Recurrent Units (GRUs) that process the sequence of feature vectors. GRUs capture temporal dependencies in the sequence, which is crucial for recognizing characters in context.

The final part of the CRNN is the transcription layer, which converts the output of the RNN into a readable text sequence (see 3.3).

To reduce variance in the model dropout is used in the implementation.

For more detail please look at the implementation in the file *crnn.py*.

3.3 Criterion: CTC Loss

The Connectionist Temporal Classification Loss (CTC Loss) [1] is designed for tasks where align-

ment between sequences is difficult. It is especially useful for OCR tasks. In my model it is used as the transcription layer (see figure 3).
The challenge with OCR is that the number of characters in the image (input sequence) and the length of the text (output sequence) can vary. The exact alignment between the input and the output is unknown.
In a nutshell, CTC works as follows: In this case, the input sequence is a series of feature vectors extracted from the image using a CRNN. The target output sequence is the gold text. To handle varying lengths CTC introduces an intermediate representation that includes a special blank token (denoted as "-"). This allows the network to output a sequence that can be longer than the target sequence, which allows the possibility of repeating characters and blanks. CTC calculates the loss by considering all alignments between input and output sequences. For each alignment, CTC calculates the probability of the input sequence producing that alignment. Mathematically if the input sequence is $x = (x_1, x_2, \dots x_n)$, where n is the number of time steps and the target sequence is $y = (y_1, y_2, \dots, y_k)$, where k is the number of characters in the output text, the CTC loss can be calculated like this:

$$L_{CTC} = -logP(y|x)$$

The Probability $P(y|x)$ is computed by summing over all possible alignments that can result in y.

3.3.1 Problem: NaN CTC Loss

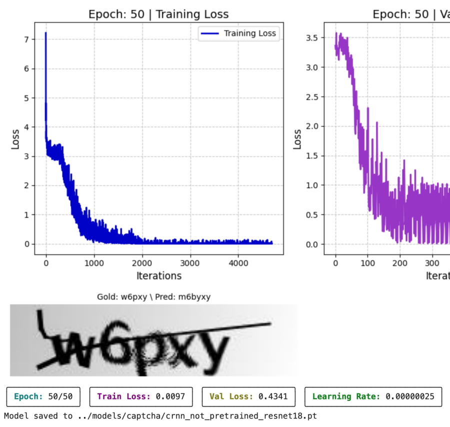
As awesome as the CTC Loss is, it resulted in one of the greatest challenges in my project. Sometimes the CTC Loss becomes infinity. Therefore the model only predicts empty outputs. However, after a lot of trial and error, changing the hyperparameters, changing the optimizer, and following the practices mentioned in this forums post[2], I was able to solve the NaN problem.
In the end, the following points helped me solve the problem:

- setting the zero_infinity parameter to True *nn.CTCLoss(..., zero_infinity=True)*
- using gradient clipping
- lowering the learning rate
- trying different optimizer parameters and for IAM changing the optimizer to AdamW

[2]https://discuss.pytorch.org/t/best-practices-to-solve-nan-ctc-loss/151913

3.4 Model Trainer

The class *CTCModelTrainer* is used to train the models. It trains the model, and after each epoch prints the progress by; validating the model, plotting the training loss and validation loss, printing a sample image with the corresponding gold label and prediction, the current learning rate, and the progress of training. See figure 4 for the beautiful output the trainer generates.

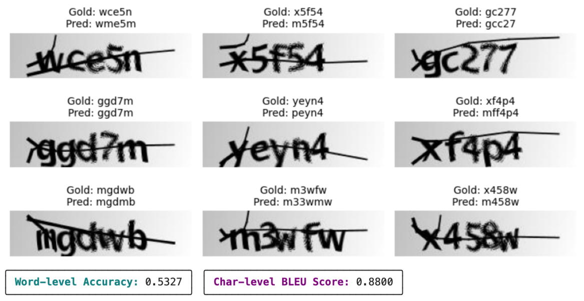

Epoch: 50 | Training Loss
Training Loss
Loss
Iterations
Epoch: 50 | Validation Loss
Validation Loss
Loss
Iterations
Gold: w6pxy \ Pred: m6byxy

```
Epoch: 50/50   Train Loss: 0.0097   Val Loss: 0.4341   Learning Rate: 0.00000025
Model saved to ../models/captcha/crnn_not_pretrained_resnet18.pt
```

Figure 4: Example of trainer output

3.5 Model Tester and Evaluator

The class *CTCModelTesterEvaluator* tests the model on the testing dataset. After that it calculates the word-level accuracy (only true if gold and prediction are an exact match) and the char-level BLEU score [4]. A sample grid of images with corresponding labels, the word-level accuracy, and the char-level BLEU score are printed beautifully 5. Optionally a confusion matrix is also printed.



```
Word-level Accuracy: 0.5327   Char-level BLEU Score: 0.8800
```

Figure 5: Example of evaluator output

Model	Word-level accuracy	Char-level BLEU score
Pretrained resnet18	53.27	88.00
Not pretrained resnet18	49.07	85.75
Simple CNN	0.00	2.75

Table 2: Results of the Captcha experiments

Model	Word-level accuracy	Char-level BLEU score
Pretrained resnet18	27.00	57.81
Not pretrained resnet18	21.60	47.53
Simple CNN	19.00	41.13

Table 3: Results of the IAM experiments

4 Results and Discussion

All the raw results can be found in my GitLab Repository[3]. The used Hyperparameters can be found in the corresponding Jupyter Notebook of the experiment.

4.1 Captcha dataset

In table 2 you can see the performance of the different CNN architectures. As you can see my simple CNN model performed very poorly and I have no idea why. I tried a lot of different hyperparameters and model architectures, but all the models that I implemented performed very poorly on the captcha task. That is also why I tried a different, more complex architecture. Resnet18 performed better with a word-level accuracy of around 50% and a char-level accuracy of around 85%. The accuracy and BLEU score increased another 3% if the pre-trained model was used.

4.2 IAM word datset

In table 3 you can see the performance of the different CNN architectures. This time my simple CNN model performed worse than the not pretrained resnet18 model, but only by around 2% accuracy-wise and around 6% BLEU-wise. The pre-trained resnet18 model performed significantly better than the model that has not been pre-trained. It has a 5% higher word-level accuracy and a 10% higher char-level BLEU score.
By looking at the example image grid you can see that only simple words (like ”and”, ”the”, etc.) were predicted completely right. The model predicted some chars, for more complex words, right, but often made a few character errors. See figure 6 to see by yourself.

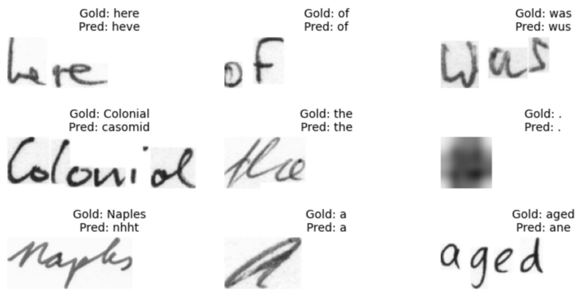


Figure 6: Sample image grid from trained pre-trained resnet18 model on the test dataset.

Interestingly, when you look at the confusion matrix 7, the models often predicted only lowercase letters. There could be multiple reasons why this is happening: the model might have seen more lowercase letters during training, leading to a bias towards them. Alternatively, the architecture might have too few layers, resulting in an inability to predict a broader range of characters.

5 Conclusion

The goal of this project was to integrate a simple CRNN model for OCR using PyTorch. Although the basic CNN model did not achieve good results on the captcha dataset, it performed comparably to the more complex ResNet18 architecture on the IAM task. The ResNet18 model, especially when pre-trained, significantly outperformed the non-pretrained version in both the captcha and IAM experiments. However, it is somewhat disappointing that the IAM model only correctly predicts simple words. To achieve even better results, a more complex and specialized architecture would be necessary, which is beyond the scope of this

[3]Link to results

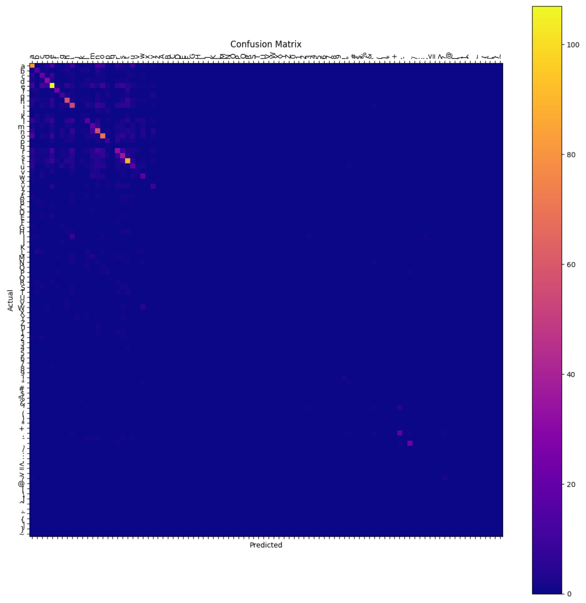


Figure 7: Sample confusion matrix from trained pre-trained resnet18 model on the test dataset. For a bigger image please look at the GitLab Repository.

project. I hope this work provides a solid foundation, complete with thorough documentation and visually appealing outputs, for beginners like me to further explore the CRNN architecture for OCR in PyTorch.

References

[1] Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural 'networks. *ICML 2006 - Proceedings of the 23rd International Conference on Machine Learning*, 2006:369–376, 01 2006.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[3] U-V Marti and Horst Bunke. The iam-database: an english sentence database for offline handwriting recognition. *International Journal on Document Analysis and Recognition, 5(1):39–46*, 2002. Available at `https://fki.tic.heia-fr.ch/databases/iam-handwriting-database`.

[4] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In Pierre Isabelle, Eugene Charniak, and Dekang Lin, editors, *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics.

[5] 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, 2015.

[6] Rodrigo Wilhelmy and Horacio Rosa. captcha dataset, July 2013. Available at `https://www.kaggle.com/datasets/fournierp/captcha-version-2-images/data`.

Code References

These references inspired my code structure, model architecture, and output formatting in some way.

1. https://medium.com/analytics-vidhya/resnet-understand-and-implement-from-scratch-d0eb9725e0db

2. https://github.com/GabrielDornelles/pytorch-ocr/tree/main

3. https://github.com/carnotaur/crnn-tutorial/tree/master

4. https://www.kaggle.com/code/kowalskyyy999/captcha-recognition-crnn-ctcloss-using-pytorch

5. https://deepayan137.github.io/blog/markdown/2020/08/29/bu
ocr.html