Character Recognition using Convolutional Neural Networks

Paolo Filomarino, Nicola Salerno, Salvatore Pappalardo

Abstract
Recognizing handwritten characters in images is a challenging problem that has recently drawn a lot of attention in the machine learning community. This project aims to classify images of handwritten latin letters and digits in order to use them in digital format. We trained, evaluated and tested a Convolutional Neural Network on different datasets to achieve this goal.

Keywords
Convolutional — Neural — Network — Character — Recognition

Contents

1 Introduction 1
Introduction 1
2 Neural Network Architecture 1
3 Results and Discussion 2
3.1 Datasets . . . 2
3.2 Languages and frameworks used . . . 2
3.3 Dataset preprocessing . . . 2
3.4 Experimental results . . . 2
64x64 models • 32x32 models
4 Final Remarks 5
References 5

1. Introduction

Predicting the correct character contained in an image is not a new kind of task. Numerous works exist in this field, as shown in [1]. and [2], in which character recognition is used, along with unsupervised algorithm, to recognize text in the wild and handwritten sentences.
However, for this project we choose to take a step back and use Supervised Learning approach using a Convolutional Neural Network (often referred to as CNN). A CNN is an Artificial Network commonly used fields like object recognition (from images and videos), recommendation systems and text classification.
A CNN can handle hundreds of layers, it learns different kinds of relevant features from a given observation (which can be an image or a text) in each of them. In order to extract these features, a sequence of convolutional layers and pooling layers are applied to the input data, every convolutional layer applies different filters to the input data, subsequently, the output produced is commonly sent to the next layer, which is often a pooling layer, the latter is used to reduce the global size of the computed data.
In the images case the filters are used to learn different patterns, like borders, brightness and even more complex forms, all these patterns are useful for object identification. Using this kind of network, along with the right dataset, we can train a model which can later be used to classify and identify images with a certain grade of accuracy.

2. Neural Network Architecture

In this section we'll briefly describe the basic structure of the Convolutional Neural Network used to train and test the models for Character Recognition tasks (shown in figure 1).
We preprocess the data before feeding it into the network, every image is shrunk from 128x128 pixels to 32x32 and 64x64 (depending on the test case). Later, the dataset is shuffled (in order to feed random observations into the network) and divided in batch sets, each of which with 16 observations. After these preprocessing steps, the data goes into the first Convolutional layer, which uses 128 filters and a kernel size of 5x5. The activation function used, not only in this layer but throughout the network, is LeakyRelu, with the α hyper-parameter set to *0.1*. We chose this particular activation function because it is computationally efficient, it converges faster than other activation functions, like Sigmoid and the Hyperbolic Tangent, and it eventually smooths out the problem of "killing" too many neurons after numerous iterations, which may come up when using Relu (the latter sets to 0 any negative result, whereas Leaky Relu uses the α parameter to fix it). The result of this operation is a tensor with size 28x28x128.
In order to extract relevant features from data, we then apply a Max Pooling step, with a mask size of 4x4, resulting in a tensor with size 7x7x128. The output of the Max Pooling step is then fed into a second Convolutional

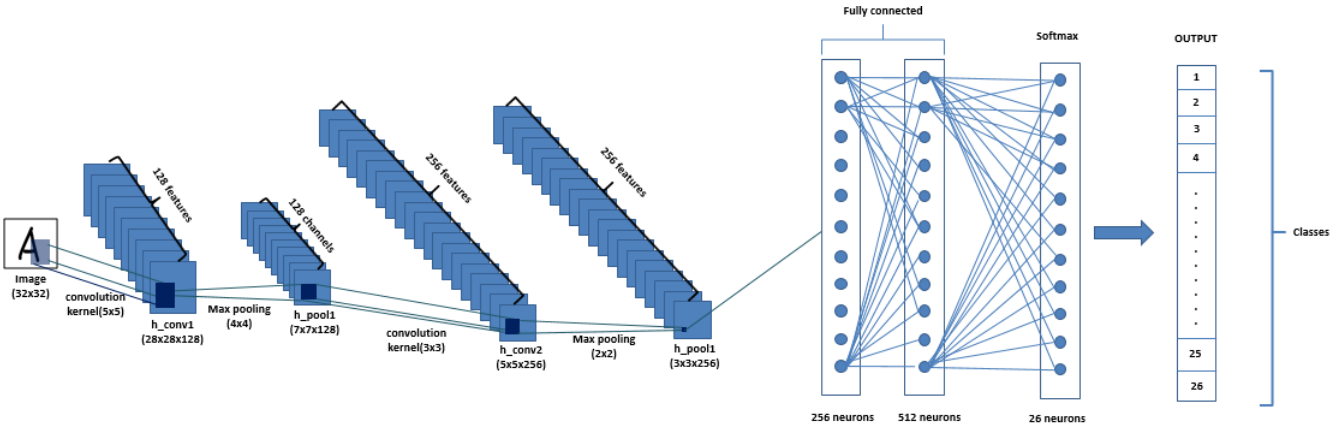


Figure 1. Convolutional Neural Network structure

layer having 3x3 kernel size and 256 filters, obtaining a tensor with shape 5x5x256; we then downsample the image again using another Max Pooling layer, this time having pool size 2x2, the outcome of this operation has shape 3x3x256.
We then convert the result into a one dimensional vector with 2304 elements, which is passed into a fully connected layer having 256 neurons. In order to avoid overfitting we then apply a Dropout operation (which is a regularization technique) to the outcome of the last layer, with a rate equal to *0.5*. We then add a second Fully Connected layer with 512 neurons and a second Dropout layer, this time with a rate of *0.25*.
Finally, we apply the last Dense layer, having 26 neurons (or 62 neurons, depending on the dataset we're using), we then use the Softmax activation function (commonly used for classification) to retrieve values between 0 and 1.

3. Results and Discussion

In this section we'll present and discuss the achievements obtained and the problems encountered while carrying out our project.

Datasets

We obtained our train and validation set from NIST Special Database 19. The original dataset contains roughly 1.500.000 images, it is organized by class and by writer, and it is filled with handwritten lowercase letters, uppercase letters and digits.
To test our model we used The Chars74K dataset, which contains 55 samples per class.

Languages and frameworks used

To build the Neural Network we used Keras. Keras is a library written in Python, it is an high level API that can run using different back-ends, such as Tensorflow, CNTK and Theano (in our case we used Tensorflow as back-end), its main purpose is to simplify the trainings and testing of neural networks, letting the users focus on its architecture rather than on its technical implementation. We used it to train our model exploiting the GPU of our machines, in order to compute results in (almost) reasonable times.
We used Tensorflow as a backend for Keras and to write and read TFRecords, which is a format developed and recommended by Tensorflow developers, it is used to handle large datasets in an optimized way.
To reshape the original dataset into TFRecords we developed two tools. The first one, written in Go (a language developed by Google), used to flatten the original directory structure, to divide it in test and train set and to obtain an equal number of observations for each class. The second one, written in Python does the actual conversion in TFRecord. Lastly, to plot and analyze our results we used R, in particular we used the ggplot2 package to do the plotting.

Dataset preprocessing

We used the Golang tool we've developed to reduce and reshape the original Nist19 dataset. We made two transformations, one resulted in a dataset containing 26 classes (uppercase letters), and the other in a dataset containing 62 classes (lowercase, uppercase and numbers), in each case we took 4133 observations by class. At the end of the 26-classes transformation, the size of the dataset has been reduced to a total of 107.458 images, whereas we obtained a dataset with 256.426 images with 62-classes transformation. In both cases we split the dataset in validation and train set, precisely reserving 75

Experimental results

In this subsection we'll describe and discuss the outcome of various experiments that we carried out.
We used the ADAM optimizer([3]) on a categorical cross entropy function.
We initially set a learning rate of 0.0003 and trained four

different models on 25 epochs with various configurations:

- 64x64 pixels and 26 classes
- 64x64 pixels and 62 classes
- 32x32 pixels and 26 classes
- 32x32 pixels and 62 classes

As we weren't completely satisfied with the results, in particular because we noticed that during the training phase the loss function was always increasing while the accuracy was decreasing, we decided to retrain the 32x32-pixels based models (because they performed better) setting the learning rate to 0.0001, obtaining slightly better results.

64x64 models

Table 1. Training results for 64x64 models

Classes	Loss	Train acc.	Val. acc.	Test acc.	Tr. time
26	3.3049	79.49%	84.75%	68.11%	7h56m
62	8.4626	79.49%	79.49%	33.26%	19h04m

In this experiment we didn't achieve remarkable results. We reached an 68% accuracy on the test set in the 26-classes model and only a 33.26% in the 62-class model. The plots containing the confusion matrix for both models are shown in figures 2 and 3.

Figure 2. Confusion matrix for 26-classes 64x64-pixels model

32x32 models

As shown in the summary table, for 26 classes, the training phase needed 3h28m to complete, achieving a 1.5712 loss value and a testing accuracy of 74.83%, while, regarding to 62 classes case, the training phase took 8h38m,

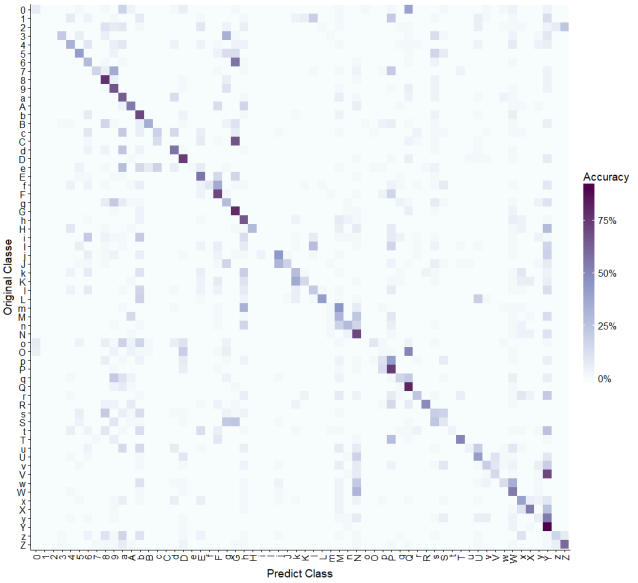


Figure 3. Confusion matrix for 62-classes 64x64-pixels model

Table 2. Training results for 32x32 models with learning rate set to 0.0003

Classes	Loss	Train acc.	Val. acc.	Test acc.	Tr. time
26	1.5712	90.25%	92.05%	74.83%	3h28m
62	0.5813	90.57%	80.28%	54.78%	8h38m

achieved a 0.5813 loss value and a 54.78% testing accuracy.
This table denoted a small improvement compared to the first set of experiments. The confusion matrices related to this experiment are shown in figures 4 in and 5

Table 3. Training results for 32x32 models with learning rate set to 0.0001

Classes	Loss	Train acc.	Val. acc.	Test acc.	Tr. time
26	0.0133	99.79%	97.61%	87.27%	3h45m
62	0.0957	96.97%	82.67%	57.51%	8h32m

As shown in the summary table 3, for the 32-class case, after a training period of 3h45m the testing accuracy reached 87.27% with a 0.0133 loss. Instead, the 62-classes case took 8h32m, the testing accuracy reached was 96.97% with a loss of 0.0957.

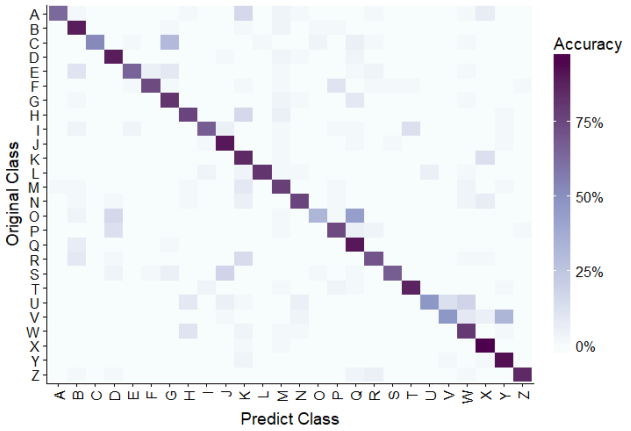


Figure 4. Confusion matrix for 26-classes 32x32-pixels model with learning rate 0.0003

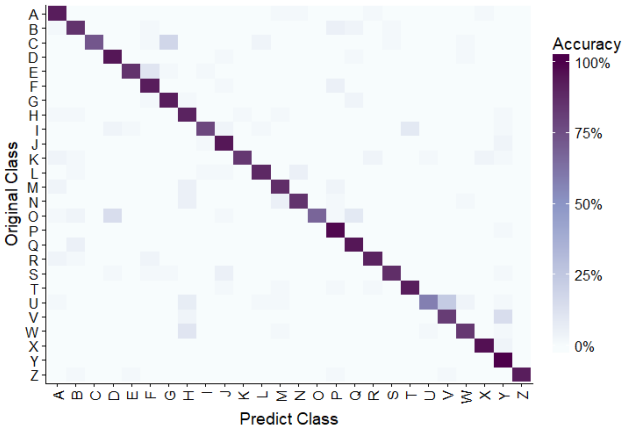


Figure 6. Confusion matrix for 26-classes 32x32-pixels model with learning rate 0.0001

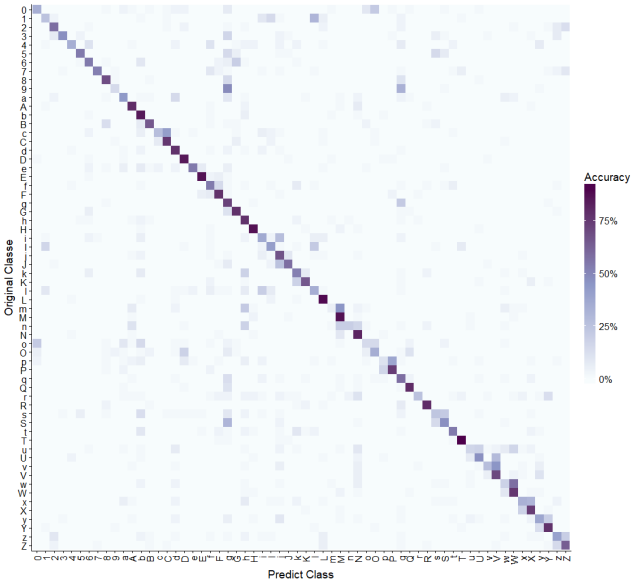


Figure 5. Confusion matrix for 62-classes 32x32-pixels model with learning rate 0.0003

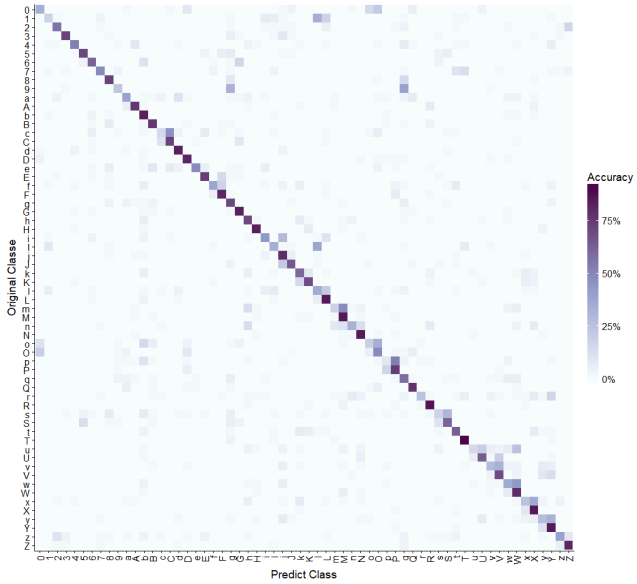
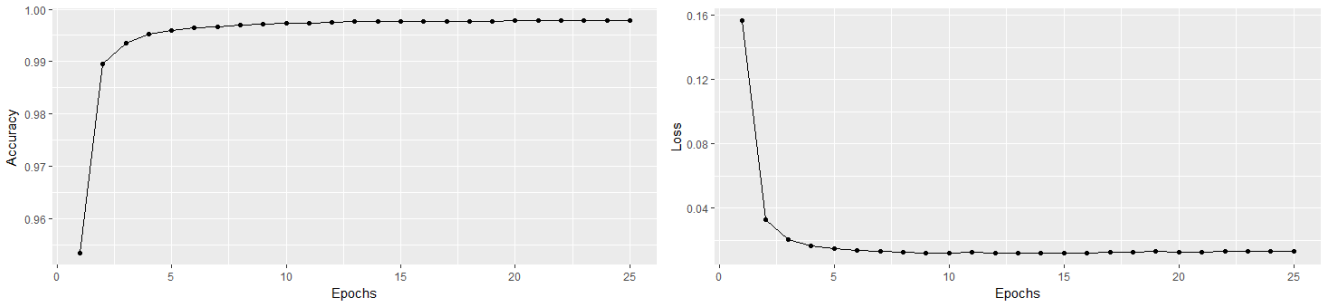
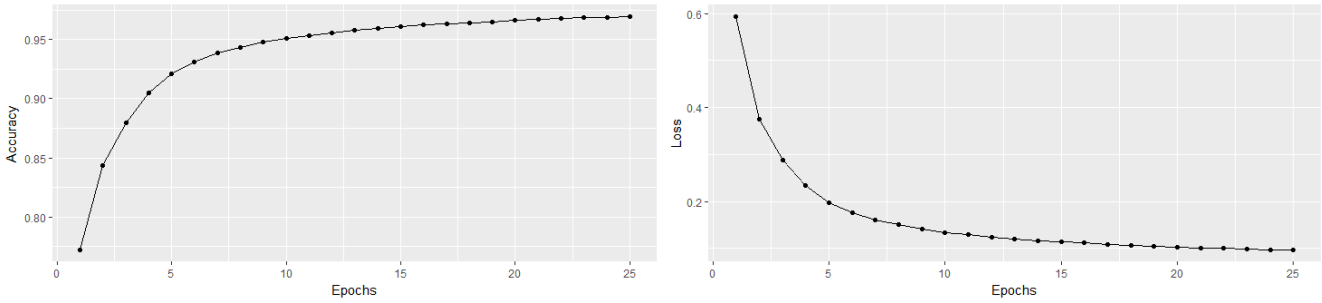


Figure 7. Confusion matrix for 62-classes 32x32-pixels model with learning rate 0.0001

The confusion matrix related to 26-classes model is shown in figure 6, while the confusion matrix related to the 62-class case is shown in figure 7. Plots for loss and accuracy are shown in figure 8. From the resulting graphs we note that the accuracy has increased while the loss has decreased in both cases, respecting the characteristics of a classic neural network



(a) loss and accuracy for the 26-classes 32x32-pixels model with learning rate 0.0001



(b) loss and accuracy for the 62-classes 32x32-pixels model with learning rate 0.0001

Figure 8

4. Final Remarks

The results produced by the network that we trained shows us that it works better with 32x32 shaped images and with the learning rate set to 0.0001. The less efficient performance was obtained using 64x64 resized images with the learning rate set to 0.0003, to improve the results of these we may stack another layer on top of this model, or tweak the hyper-parameters so that it adapts better the size of the input data. Regarding possible future developments, the model can be reused for the realization of an algorithm for handwritten word recognition, or can be adapted to train a model whose task is to detect text in natural images.

References

[1] A. Coates, B. Carpenter, C. Case, S. Satheesh, B. Suresh, T. Wang, D. J. Wu, and A. Y. Ng. Text detection and character recognition in scene images with unsupervised feature learning. pages 440–445, Sept 2011.

[2] Adam Coates Andrew Y. Ng Tao Wang, David J. Wu. End-to-end text recognition with convolutional neural networks. 2012.

[3] Jimmy Ba Diederik P. Kingma. Adam: A method for stochastic optimization. 2014.