<https://www.kaggle.com/datasets/fournierp/captcha-version-2-images> Dataset

<https://www.kaggle.com/code/mathieulareu/captcha-lm> Separa las letras

<https://github.com/JackonYang/captcha-tensorflow> Fa la pràctica

CaptchaSolver

A deep-learning model to solve textual captchas

# The dataset

We found

* 1070 PNG images each with their label
* 200 x 50 image size (W x H)

With this dataset, we faced three essential problems:

1. The dataset may be too small
2. The dataset is not balanced
3. The image size is not square

The implications and solutions for these problems are explained below.

## Dataset size

1070 images could seem like not enough to train a model.

To augment the size of the dataset, we could take three approaches (from easier to harder):

1. Traditional data augmentation: applying small rotations, scaling or image deformations.
2. Manual data generation: since this type of Captcha consists in just black characters on top of a grey gradient background, with some noise and lines on top, generating many samples like these automatically with Pillow could be an easy solution to get infinitely many samples.
3. Data generation with a conditional Generative Adversarial Network (GAN).

At the end, we found that we didn’t need to augment the dataset size as the models were already giving great results with the original dataset.

## Dataset distribution

There are

## Image size

Convolutional Neural Networks (CNNs) are traditionally applied over squared images.

We could design our CNN to work specifically with 200x50 images, but if we wanted to use transfer learning at some point, we would not be able to do so because pre-designed architectures use squared images.

We have two options for this:

1. Padding: adding pixels on top and below to have images of size 200x200.
2. Cropping: removing pixels from the left and the right to have images of size 50x50.

We decided to do both:

1. First we add padding by extending the pixels vertically to get 200x200 images.
2. Finally, we crop the image to get a 192x192 image.

We reduce the images to 192x192 because, this way, we can apply pooling to halve the image size 6 times at much to obtain a 3x3 image. With 200x20, we could only halve them 3 times to get a 50x50 size.

Nonetheless, from the documentation of Torch pre-trained modules, we have seen that they apply paddings and croppings to the input images automatically to fit the requirements of the architecture.

# The model

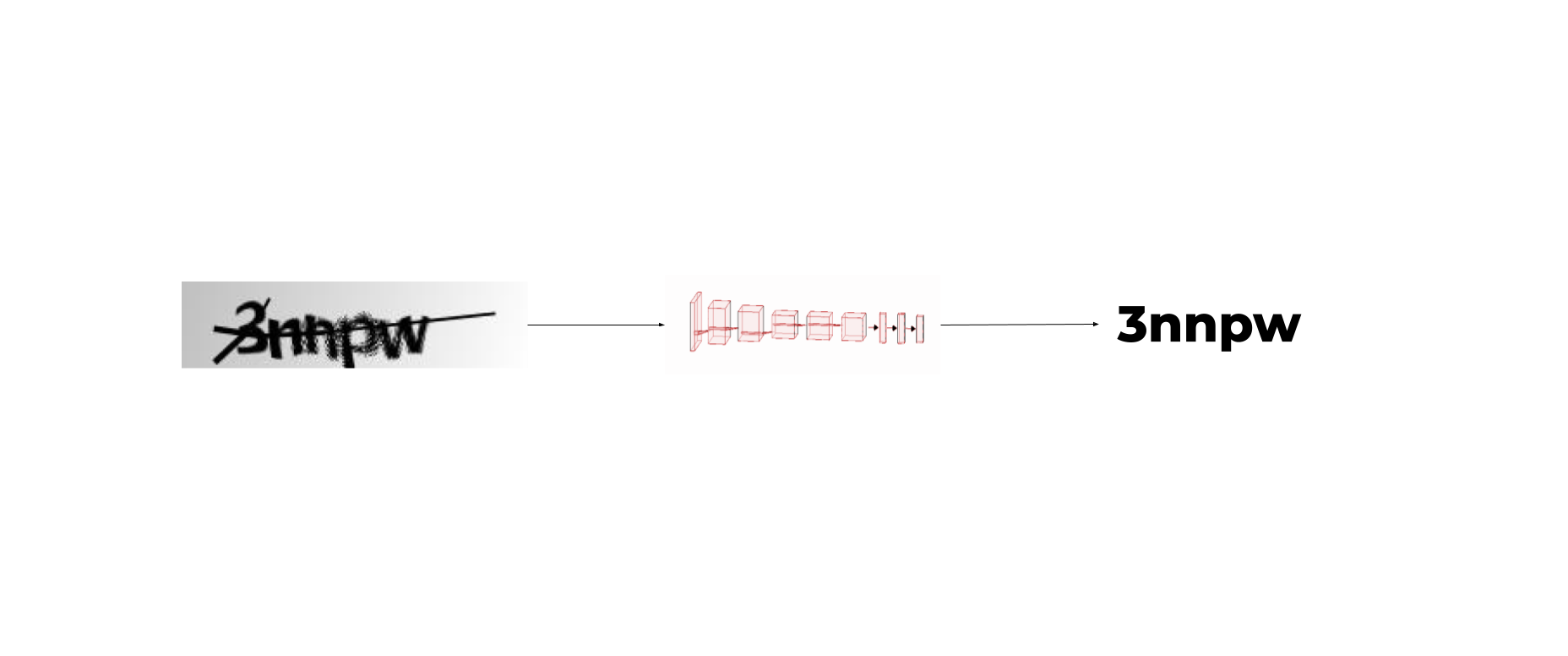
Regarding the model, we faced several challenges:

1. From a naive point of view, the output space (all the possible classes) is too large: with 26 letters and 10 possible numbers, the combinations of 5 of these characters implies +60,000,000 possible labels. So the output of the last fully connected layer should have a dimension of 60 million.
2. The input being an image suggests the use of a CNN. But we are trying to detect a sequence of characters in the image, so this could also suggest the use of a Recursive Neural Network (RNN). But at the same time, this sequence of characters does not have any sequential dependency (characters in a Captcha are randomly chosen). So an RNN would most probably be useless because there are no sequential relationships between the characters.

In the following sections we propose different architectures for the model and different embeddings for the output space and evaluate their performance.

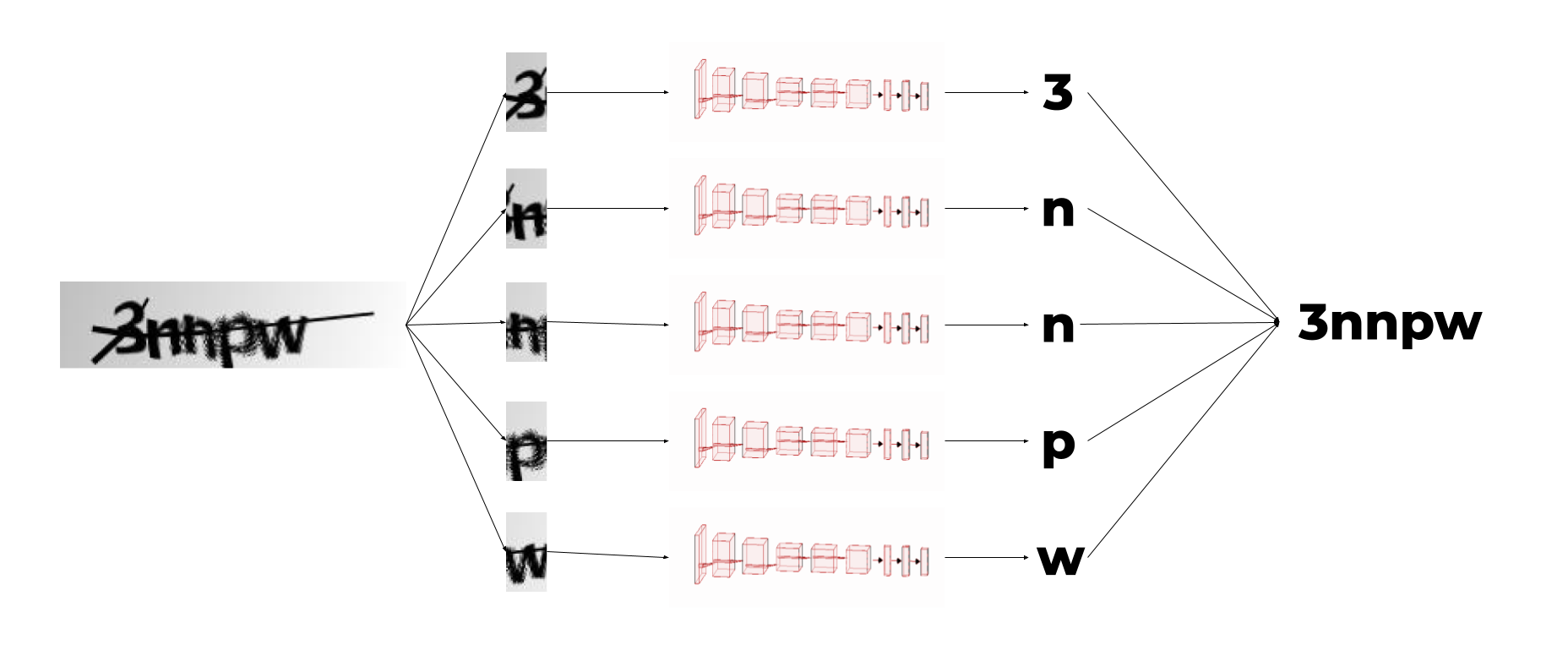
## The most naive approach

The first idea that comes to mind is to give the image directly to the CNN and make it predict the full label.



## Cropping each character and classifying it independently with a CNN

Our following intuitive idea was to crop each input image in five different images so that we had each one of the five characters in a separate image, and then use a CNN to predict the character that each image shows.



We found this option very efficient for many reasons:

1. Now, instead of 1070 images, we have 5350 images in our dataset.
2. We need to train a CNN for a very simple task: identifying a single character.
3. The same CNN can be reused to detect each of the five characters.
4. The output space of the CNN is 36 (the number of possible characters) and we would only need to aggregate the results of the predictions of the five characters.

So, using this approach reduces complexity and dimensionality.

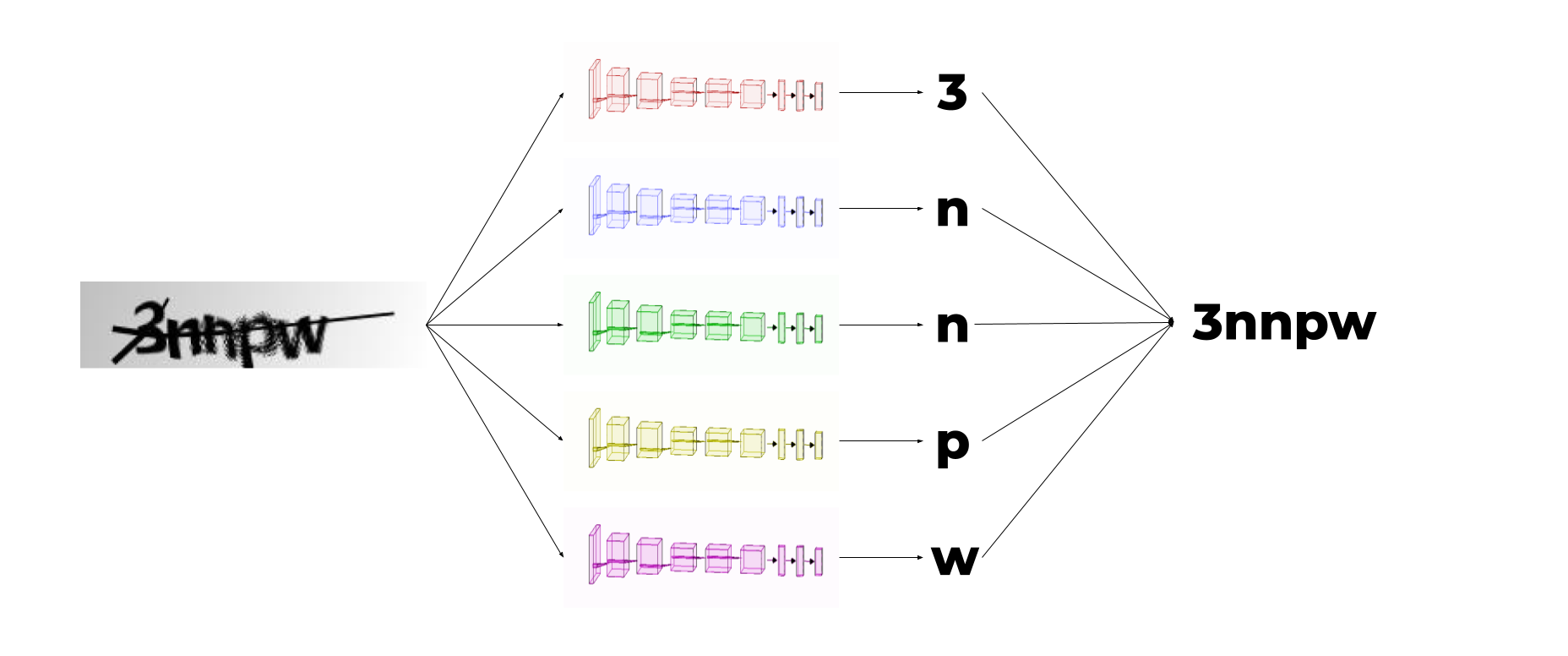
But we also thought that removing the context from the analysis can lead to less accuracy. For instance, analysing a character independently, we can classify it as a B, but when looking at the context we could see that it is clearly an R but with a line in the bottom crossing the full string.

Nevertheless, we thought that classifying the characters independently could help with the rest of the process. For instance, by classifying them independently we may doubt if a character is a B or an R or a P, but at least we know it is clearly not a W.

In other words, without context, we can efficiently rule out some output options and introducing later the context we can finally make a definitive guess.

## Using a different CNN to identify each of the five characters

So we thought that maybe context was a key piece in our problem. This means that we have to train whatever CNN with the full image, not with croppings.



(Another alternative solution: same CNN with different FC for each character)

