Fast Bib Number Detection And Recognition – Deep Learning Implementation

Jean Luc ANOMA

Summary

Recognizing text in natural photographs is a hard problem due to the multiple sub-tasks that it entails : detecting and localizing the text, segmenting the text region and recognizing the characters. A few solutions have been implemented for this problem. Traditional solutions use morphological image processing to detect and segment the text region and pass this region proposal to an Optical Character Recognition tool. The more recent development of deep learning algorithms for computer vision led a few researchers to tackle this problem from a different angle. A unified solution fully based on machine learning has been developed to recognize bounded sequences of numbers in a natural scene. However, no performance details regarding the speed of the algorithm were provided. It appears that deep learning algorithms tend to be very challenging to deploy on consumer hardware. For that reason, building efficient and fast deep learning algorithms has recently been a very active area of research. In this work, we propose to tackle the problem of recognizing and reading bib numbers using and end-to-end deep learning algorithm which can run smoothly on a consumer hardware.

Keywords: Text detection, character recognition, Deep Learning, Convolutional Neural Networks.

Table des matières

1. Introduction

As expressed in [2], Recognizing arbitrary multi-character text in unconstrained natural photographs is a hard problem. Compared to scanned documents, the challenge is much harder due to added noise, interfering objects, occlusion of part of the text, illumination, shadows, rotation ect….

The problem is even much more complex when the image is a raw image which has not been cropped around the bib area and when there are multiple bibs to be deciphered is one image.

Nonetheless, this problem is linked to very concrete use cases : license plate number recognition, brand name recognition on clothes or shoes, automatic annotation of photo or video content in social media applications etc… In the frame of this work, we focus on race bib number recognition. Indeed, for photographers, tagging by hand photographs to be sent to the racers is very time consuming. A fast automated solution would be an important increase in productivity for them.

Capitalizing on recent significant advances in image processing based on deep learning algorithm, we are proposing a fully deep learning-based solution which is able to read multiple bibs in raw images, regardless the number of bibs in the image or the color of the bib.

2. Problem formulation

The final aim is to be able to read athletes' bib numbers in 'quasi' real-time for example from a camera placed on the finish line, or to be able to quickly tag large amounts of photographs of athletes with their bibs. So given an image, the task is to identify the bibs present in the image, as a set of sequences of digits of bounded length. Each bib number will have to be identified entirely and with the digits placed in the right order. A sequential implementation for this problem already exists and allows to recognize up to 9+ frames per seconds on a portable PC (2GHz @ 4 core CPU) using a multi-threaded application. However this work does not measure end-to-end performance, from the raw image to the fully tagged image. Indeed, the number recognition part (using Optical Character Recognition) is not included in the throughput measure. This work will come up with an end-to-end, deep-learning based, solution for this problem. The solution will have to be fast enough to run in a real-time situation and accurate enough in order to be relevant.

3 Literature review : related work and existing solutions

3.1 Methods based on pre-processed images

The text recognition methods which are based on pre-processed images are often composed of 3 main phases:

- Pre-processing of the image to increase uniformity
- Text spotting
- Optical Character Recognition

3.1.1 Image pre-processing methods

Those methods are usually based on **mathematical morphology**. As explained in [3], mathematical morphology is used as a tool for extracting image components that are useful in the representation and description of region shape. The basic mathematical operations, which are the building blocks for most complex operations, are **dilation** and **erosion**. Based on the latter, we can build **opening** and **closing** operations.

Dilation and erosion are set theory operations. Those operations transform a set *A* into a new set *A'.* The transformation of the set *A* uses another set, the **structuring element**, as a parameter. As pointed out in [3], the structuring elements are small sets of sub images used to probe an image for properties of interest. The structuring element has both a shape and an origin as shown in *figure 1*.

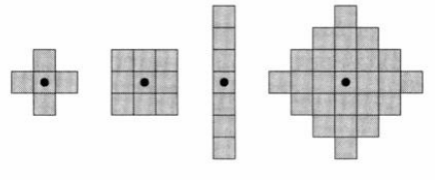


Figure 1: Examples of structuring elements

Dilation is used to enlarge the original set *A* while erosion is used to shrink it. Based on those basic operations, in the context of computer vision, the opening operation generally smooths the contours of an object and eliminates thin protrusions. On the other hand, closing fuses narrow breaks and long and thin gulfs and eliminates small holes and gaps in the contour ([3]).
As a result, Morphological Image Processing enhances degraded noisy images and can be used to fix incomplete or broken characters.

In addition, in [1], a filtering process is applied with a view to sharpening foreground/background transitions and reducing image complexity. Traditional filtering (i.e. order-statistic filters such as Gaussian, Median Blur etc) is not applied since it manages noise by using weighted averages of the neighborhood around each pixel. This approach increases image uniformity but also smear the image and lose definition around high-contrast boundaries. Instead, a Majority Vote Processor (MVP) is used, which consists in modifying the median blur filter to select the most common value in a pixel's neighborhood. This is done in two steps :

- First the image resolution is reduced to limit the range of available grays (in the case of a gray-scale image)
- Then an aperture is run across the image and the most common value is selected in the neighborhood of each pixel

After applying this MVP transformation to an athlete's bib image, the result is higher image uniformity together with sharp boundaries around the bib. According to [1], this MVP process enhances the speed of traditional Extremal Regions (ER) algorithms for text detection by forming blobs of gray which the ER algorithm will exploit.

To conclude on this part, experience has proved that the methods described above tend to ease the text detection process. However the main drawback is that some of them (filtering for instance) incur a high processing overhead [1], even though a subsequently more efficient text detection may compensate for part of or all of this overhead. In [1], this overhead is significantly reduced using a multi-threaded application.

3.1.2. Text detection

This part of the process is about detecting the presence of text in an image and locate this text by drawing boundaries around it. The end objective is to have a region which can be successfully submitted to an Optical Character Recognition tool [1]. Two categories of methods can be quoted:

- Character Region Method
- Sliding Window method

a. Character Region method

An example of Character Region method is the use of Extremal Regions. Extremal Regions are dark regions fully surrounded by a lighter region (maximum intensity region) or light region surrounded by a darker region (minimum intensity region) [6].

Extremal Regions can be built ‘pixel by pixel’. In [1], a quicker algorithm based on blobs is implemented and proves to be 50% quicker than ‘pixel by pixel’ methods. The algorithm is called “Building Extremal Regions using Blobs” (bERb). It uses the blobs of gray formed after applying the MVP algorithm. First the bERb algorithm groups adjacent pixels with the same shade of gray into the same blob using a linked list. Then, after initializing the Extremal Regions with the black blobs, existing Extremal Regions iteratively absorb the blobs in their neigbourhhod starting from the darker ones. After a few iterations the algorithm is stopped, as soon as the bib numbers are perfectly formed and can be successfully submitted for Optical Character Recognition.

In [1], experiments show that the use of MVP plus bERb algorithm saves significant processing time with negligible cost in terms of accuracy. However, bERb algorithm’s main inconvenient is that it requires pre-processed images that are highly uniform with the risk

of terminating with insufficient memory resources. The other inconvenient is that the algorithm is less robust to dysfunctional images than other ER-based methods.

Another method is the Stroke Width Transform (SWT) used in [15]. The method relies on the fact that text appears as strokes of constant width. Then the SWT is a local operator which computes the likely stroke width of each pixel of the image. By grouping pixels with stable stroke width, color and size, text regions can be effectively detected and then passed on to an Optical Character Recognition tool as for the previous algorithm.

b. Sliding window method

This family of methods consists in sliding a box around the image and applying a classifier to each image crop to decide whether it is containing text or not. The result is a text saliency map which can be used to draw boundaries around the region predicted to contain the text.

In [7], every image crop is analyzed using feature extraction and is injected as an input into a linear Support Vector Machine to detect the presence of humans in an image. This process has been reused by [8] in their end-to-end scene text recognition.

3.2 Deep Learning

3.2.1 Basic Deep feedforward neural network models

Computer vision has traditionally been one of the most active research areas for deep learning applications [2]. In this section, we will introduce Deep Learning and its applications to computer vision.

Traditional machine learning algorithms like support vector machines, k-nearest neighbors, decision trees..., have worked well on a wide variety of problems but have not succeeded in solving the central problems in AI, such as recognizing speech or recognizing objects. The main barrier has been what has been called “the Curse of Dimensionality”: those types of problems impose to learn complicated functions belonging to a very high dimensional space ([2]). This has made generalization to new examples very challenging. These challenges have laid the ground for the development of deep learning algorithms to tackle these classes of problems.

Deep feedforward networks, also often called feedforward neural networks, or multilayer perceptrons (MLPs) are the the typical models used in deep learning, the main other one being **Recurrent Neural Networks** (RNNs). In deep feedforward networks, the flow of operations goes from the input, through the intermediate computations and finally to the output. Unlike RNNs, there is no feedback connection in which outputs of the model are fed back to itself ([2]). A **convolutional neural network** is a specific type of deep feedforward network which is based on convolution operations on the input image.

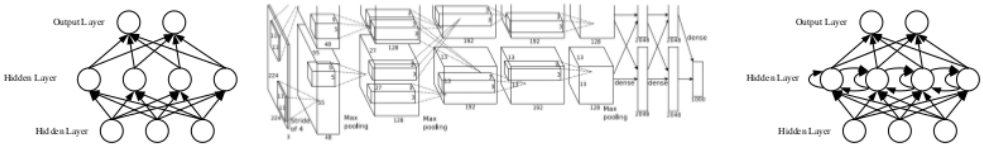


Figure 2: On the left, a fully connected deep feedforward neural network; in the middle a convolution neural network; on the right, a recurrent neural network

Typically, deep feedforward networks are composed of several layers, each of which represent a function to be learned. Those functions are then connected into a chain to produce the desired output([2]). The length of the chain gives the depth of the model. “Deep” models typically have at least two layers, the final one being the **output layer** and the previous ones being the **hidden layers**. The first layer, which contains the inputs, is called the **input layer**.

Each layer is composed of a certain number of units called **neurons** by analogy to the neurons of the brain in the sense that they receive inputs from many other units and compute their own activation value. To move forward from one layer *A* to the next layer *B,* the output of the neurons of layer *A* are transformed linearly to compute **scores :** $s=Wx$ (with x being the input to the layer and W the **weight matrix**, each weight being a parameter of the model to be optimized). Those scores are then used as input for a **non-linear activation function** which computes the outputs of the neurons of layer *B*. According to [2], in modern neural networks, the default recommendation for the non-linear activation function is to use the rectified linear unit or ReLU defined by *g(s) = max{0,s}*. The weight matrix *W* is the one which has to be learned during the training phase.

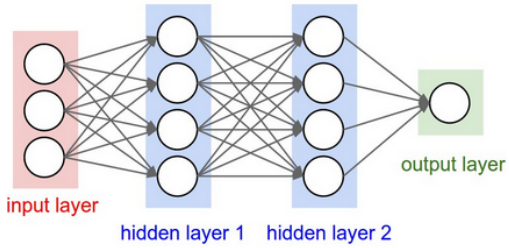


Figure 3: A 3-layer neural network with three inputs, two hidden layers of 4 neurons each and one output layer.

3.2.2 Training the model

Deep feedforward neural networks are **supervised learning** algorithms, meaning that the models are trained with a dataset containing features, but each example is also associated with a **label** or **target** ([2]). First, the dataset should be split between a training set, a validation set and a test set. The central challenge in machine learning is that the model must perform well on new, previously unseen inputs—not just those on which the model was trained([2]). We measure how well the model performs by setting a **loss function** that the model has to minimize. Hence we will have a training error, a validation error and a test error measured by this loss function. The objective is to have the lowest possible test error.

We will try to do so by minimizing the training error which should act as an estimator of the test error. To minimize the training error, nearly all of deep learning is powered by one very important algorithm : **stochastic gradient descent** or SGD ([2]) with **backpropagation**. SGD is an extension of the Gradient Descent algorithm. First we initialize the parameters of the model (the weight matrices W from the previous paragraph) randomly. Then, in order to minimize the loss function L according to the parameters W, we apply iteratively the following steps until the loss function is below a threshold or until a predefined number of iterations is reached:

- Sample a mini-batch of examples from the training dataset
- Input those examples into the model to do a forward pass and compute the average loss for those examples
- The partial derivatives $\frac{\partial L}{\partial W_i}$ of L with respect to the weight matrix W_i of layer i are calculated for each layer, using the back-propagation algorithm which allows to propagate backwards the differential of L using the chain rule:

$\frac{d}{dx}\left[f\left(u\right)\right]=\frac{d}{du}\left[f\left(u\right)\right]\frac{du}{dx}$

- For each layer i, update all the weights in W_i using the formula $\mathrm{W}_i = W_i - \lambda\frac{\partial L}{\partial W_i}$, where λ is the learning rate which is an hyperparameter of the model to be optimized manually.

In order to reduce the risk of overfitting due to the large capacity of the model, **regularization** should be applied. According to [2], regularization is any modification we make to a learning algorithm that is intended to reduce its test error but not its training error. Regularizing is as important as minimizing the loss function. It is usually done by adding a regularizing term to the loss function, for example in the case of weight decay, we will add the term $\alpha\mathbf{W}^\intercal\mathbf{W}$ to express a preference for the parameters with smaller L^2 norm, α being a value chosen ahead of time (another hyperparameter) that controls the strength of our preference for smaller values of the weights.

An application of these deep feedforward models for computer vision has been implemented for the MNIST dataset on the Tensorflow documentation website [4]. The MNIST dataset consists of images of handwritten digits. It is split into three parts: 55,000 data points of training data, 10,000 points of test data, and 5,000 points of validation data. Each image in MNIST has a corresponding label, a number between 0 and 9 representing the digit drawn in the image. The task consists in producing, for each image, a probability distribution on the set {0, 1, … 9} corresponding to the likelihood of each number being drawn on that image.

The proposed model in [4] is a fully connected feedforward neural network consisting of one input layer and one output layer followed by a softmax classifier. The model is called fully connected since all the neurons of the input layer are connected to all the neurons of the output layer through a weight.
The scores of the output layer are obtained by a matrix multiplication plus a bias term. In vectorial notation, if the input data is X and the weight matrix W, the operation can be written as follows:

$$\mathrm{score} = \mathrm{WX} + \mathrm{B}$$

The score of each neuron is injected as input into the softmax classifier to obtain a probability distribution over the {0,…,9} space. The softmax classifier is well suited for this task of building a probability distribution since its definition is :

$$\forall i \in \{0, .., 9\}, softmax(score_i) = \frac{exp(score_i)}{\sum_j exp(score_j)}$$

The output of this function is then a list of scalars between 0 and 1 for each class, which can be interpreted as probabilities.

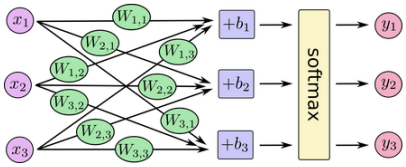


Figure 4 : Fully connected feedforward neural network implemented for the MNIST dataset

This very basic model achieves a test accuracy of 92%. However state of the art models achieve more than 99% accuracy on this dataset. Those models are based on convolutional networks.

3.2.3 Convolutional networks

One of the first models using only feedforward neural networks with backpropagation was the model developed by Lecun Y. and al who introduced their convolutional network applied to a zipcode number recognition problem [9].

According to [2] convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers. [9] explains further the motivation of such an operation. Indeed, a traditional fully connected feedfoward neural network with enough discriminative power for the task of recognizing the digits would have far too many parameters to be able to generalize correctly. Hence, the connections between the neurons have to be restricted. Inspired by the usual practice in computer vision to detect and combine local features when doing shape recognition, [9] made their model do the same by constraining the connections in the first few layers to be local. Having in mind that a feature detector which is useful on one part of the image, is likely to be useful on other parts of the image as well, they came up with the solution to scan the input image with a single neuron that has a local receptive field, and store the states of this neuron in corresponding locations in a layer called a **feature map.** This is equivalent to performing a convolution with a small kernel. A feature map is then a plane of neurons whose weights are constrained to be equal since they perform the same operation on different parts of the image. The number of feature maps generated is called number of **channels** of this convolution.

This trick reduced significantly the number of free parameters and allowed the model to learn, through backpropagation, to extract the relevant kinds features for each problem instead of doing it by hand. [9] stacked multiple feature maps in order to give flexibility to the model to extract different features from the same image. They then re-injected the resulting feature maps as inputs for another convolution operation in order to extract features of increasing complexity and abstraction.

Another remark was that higher level features required less precise coding of their location. Hence, each convolution operation leading to a feature map was followed by a another layer performing a local averaging and subsampling on this feature map in order to reduce its resolution, which increased the robustness of the model to distortions and translations. These types of layers are called pooling layers in recent models. Together with average-pooling, max-pooling is currently often implemented after a convolution layer [2].

At the end of the model was a series of traditional fully connected neuron layers connected lastly to a classifier.

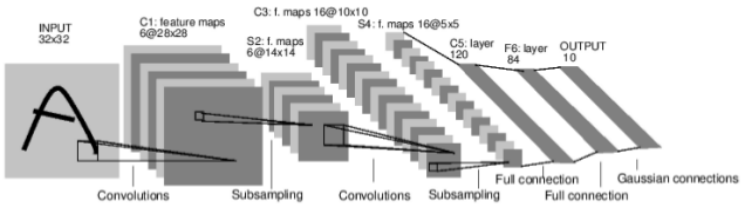


Figure 5 : LeNet-5 convolutional neural network, taken from LeCun et al., 1998

On the whole test set (2007 handwritten plus 700 printed characters), [9] achieved an error rate of 3.4% and all the classification errors occurred on handwritten characters.

The task seen so far – MNIST dataset and Zipcode dataset- are all based on recognizing scanned handwritten digits. It would be interesting to document whether specific work has been done in deep learning for recognizing text in natural scenes.

3.2.4 Convolution networks for object detection

[16] states that 2 years ago, state-of-the-art object detection systems (R-CNN, Faster R-CNN…) were deep learning based systems which first hypothesize bounding boxes (region proposals), re-sample pixels or features for each box, and apply a deep learning image classifier. These systems have been successful accuracy-wise, but too

computationally expensive and too slow for real-time applications. For instance, according to [16], the fastest high-accuracy detector, Faster R-CNN operates at only 7 frames per second (FPS).

[16] proposes to get rid of the bounding box proposals and subsequent pixel or feature re-sampling stage. Instead, a small set of default bounding boxes is evaluated at each location of a given feature map. In a convolutional way, the model evaluates each cell of the feature map and predicts for each default box, both the shape offsets (relative to a fix-shaped default box) and the confidences for all object categories. The model loss is a weighted sum between localization loss and confidence loss. Then the final step is to eliminate non-relevant bounding boxes using a confidence threshold and reduce the number of overlapping bounding boxes per class.

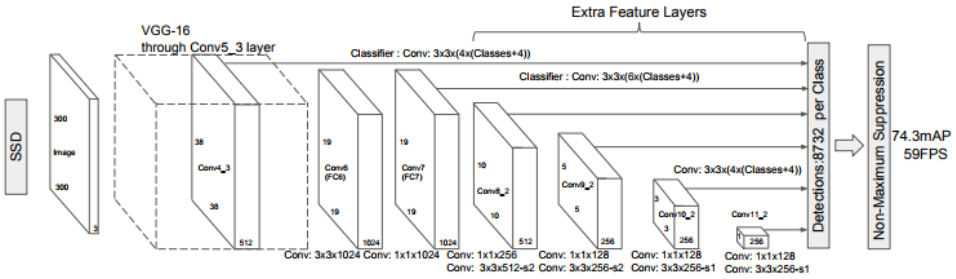


Figure 6: SSD network architecture

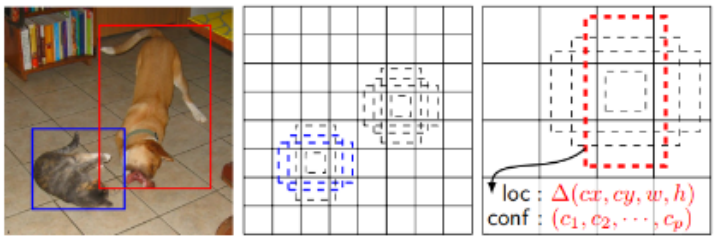


Figure 7: SSD framework. (a) SSD only needs an input image and ground truth boxes for each object during training. In a convolutional fashion, we evaluate a small set (e.g. 4) of default boxes of different aspect ratios at each location in several feature maps with different scales (e.g. 8×8 and 4×4 in (b) and (c)). For each default box, we predict both the shape offsets and the confidences for all object categories ($(c_1, c_2, \cdots, c_p)$). At training time, we first match these default boxes to the ground truth boxes. For example, we have matched two default boxes with the cat and one with the dog, which are treated as positives and the rest as negatives. The model loss is a weighted sum between localization loss (e.g. Smooth L1 [6]) and confidence loss (e.g. Softmax).

The SSD model allowed to multiply by more than 6 the inference speed of Faster R-CNN which was the fastest high-accuracy object detector, while maintaining a similar level of accuracy.

Method	mAP	FPS	batch size	# Boxes	Input resolution
Faster R-CNN (VGG16)	73.2	7	1	~ 6000	$\sim 1000 \times 600$
Fast YOLO	52.7	155	1	98	448×448
YOLO (VGG16)	66.4	21	1	98	448×448
SSD300	74.3	46	1	8732	300×300
SSD512	76.8	19	1	24564	512×512
SSD300	74.3	59	8	8732	300×300
SSD512	76.8	22	8	24564	512×512

Figure 8: comparative mAP and FPS results for different CNN architectures

The results above are measured using Titan X and cuDNN v4 with Intel Xeon E5-2667v3@3.20GHz. They are measured on the Pascal VOC2007 dataset. The mAP indicator is a measure of accuracy which takes into account the precision, recall and overlap of the bounding box with the true box (intersection over union or IoU). Precision measures how accurate the predictions are (the percentage of positive predictions which are correct); recall measures how good the model finds all the positives - true positive divided by (true positive + false negative); IoU is the area of intersection of true bounding box and predicted bounding box divided by the area of union.

Figure 9: IoU description
https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173

3.2.5 A unified convolution network for the natural text recognition task

[5] focuses on resolving the task of recognizing multi-digit numbers from Street View panoramas. While reminding the complexities of reading characters in a natural scene, [5] points out that for those reasons, traditional approaches have solved that problem by separating the overall task into 3 sub-tasks : localization, segmentation and recognition.

A year before [5], a step towards unifying those tasks had been done by [10] who proposed to approach the problem using a convolutional network as feature extractor for a system that performs object detection and localization. However, the system as a whole was larger than the neural network portion trained with backpropagation, and had special code for handling the task of coming up with region proposals.

[5] proposes a fully unified learned model to localize, segment and recognize multi-digit numbers from street level photographs, with no need for a separate component to propose region proposals or provide a higher level model of the image.

The base assumption of the model is that the number of digits is not greater than a constant N (equals 5 in the paper). For each image, the number to be identified is a sequence of digits, $\mathrm{s} = \mathrm{s}_1, s_2, ..., s_n$ plus an additional number l representing the length of the number. Hence, the output of the model is a conditional probability distribution of a vector S of N+1 random variables :

- L (the number of digits in the image with possible values ranging from 0 to N plus one additional value to account for the case where L is greater than N)
- S_1, S_2, .. , S_N, which is a sequence of N random variables, with S_i representing the i eth digit in the given image if it exists. Each of those random variables can take values ranging from *0* to *9.*

The random variables are assumed to be independent. They are discrete. The goal is then to learn a model of $P(S|X)$ by maximizing $logP(S|X)$ on the training set, X being the input image. The random variables being discrete, it is possible to use a separate softmax classifier for each of them and apply backpropagation separately for each of them.

For the inference part, at test time, the task is to predict

$$\mathrm{s} = (\mathrm{l}, \mathrm{s}_1, ..., s_l) = \mathrm{argmax}_{L,S_1,...,S_L} logP(S|X)$$

Given that the random variables are independent, this quantity can be maximized by maximizing separately the log-likelihood for each random variable. On public Street View House Numbers dataset, the best model in [5] has obtained a sequence transcription accuracy of 96.03%.

The results of the model also show a positive correlation between the depth of the model (number of layers) and its accuracy. This leads to the main issue concerning deep feedforward neural network models : resource consumption. Although the progress achieved in the hardware industry has led to the revival of deep learning with massive usage of GPUs, deep learning applications are still challenging to deploy into everyday devices like cellphones, personal computers, robots etc…

In terms of computational complexity, common convolutional network architectures like AlexNet requires about 2.5M operations per image, and close to 10 million operations for Resnet. Other famous architectures like VGG requires between 30 to 40 million operations per image, with 16 to 19 layers, more than 100 million parameters and 96MB/image of total memory needed to predict the class of only one image. Hence the need for fast and efficient convolutional networks.

3.2.6 Fast and efficient convolutional networks

The first way to speed up inference is the **use of GPU** and **parallelization** of the operations. In [11], the impact of running the deep learning algorithms on GPU and the impact of parallelization are measured during training time (number of batches of images treated per second). [11] carries out a benchmark on a few common deep learning algorithms for computer vision. The chosen algorithms were the following (input/output is the dimension of the input/output. An input of dimension 3072 correspond to a 32 by 32 pixels color image : 32x32x3) :

Networks		Input	Output	Layers	Parameters
FCN	FCN-R	784	10	5	~13 millions
CNN	AlexNet-R	3072	10	4	~81 thousands
	ResNet-56	3072	10	56	~0.85 millions
RNN	LSTM	10000	10000	2	~13 millions

Table 1 : Architecture of the deep learning models tested in [11]

As a comparison, the best algorithm obtained in [5] contained 11 hidden layers. In [11], the size of the images used as input were 54x54x3. For the AlexNet architecture running on Torch, the gain (with no parallelization) from desktop CPU (Intel i7-3820 CPU @ 3.60GHz)

to GPU (NVIDIA Telsa K80 @ 562MHz with Kepler architecture) is a 97% speedup of training time. In terms of parallelization, when AlexNet is run with Torch on 4 K80 GPUs, a speedup of 67% is observed compared to the usage of just one GPU. However those gains involve only training time but not inference time.

Another method to achieve gains during inference time is proposed in [12]: it is about **pruning the network**. It is based on the idea that in deep learning algorithms, many of the parameters end up to be redundant. The pruning method removes redundant connections and learns only important connections. After an initial training phase, [12] prunes the model by removing all connections whose weights are lower than a threshold. They then retrain the sparse network so that the remaining connections can compensate for the removed connections. The phases of pruning and retraining may be repeated iteratively to further reduce network complexity. Their results show that they are able to prune out up to 95% of the parameters before experiencing significant decrease in accuracy:

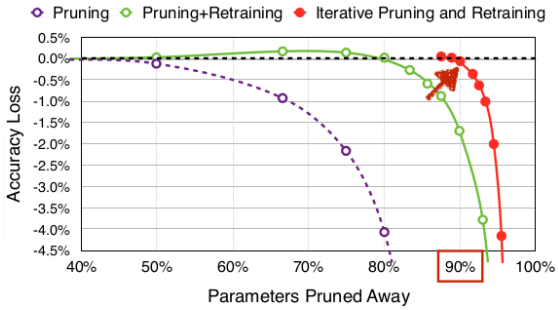


Figure 10 : Impact of pruning on a neural network

On top of pruning, [12] proposes methods to reduce the number of bits required to represent each weight. For instance, **weight sharing** proposes to group the weights into groups of similar value. Two weights will belong to the same group if and only if they differ only by a small precision number. Each group of weights will share only one single value. Thus for each weight, we need to store only a small index into a table of shared weights. Below is an illustration from [12] explaining the process. The groups of weights are materialized by their color. During the Stochastic Gradient Descent update, all the gradients are grouped by the color and summed together, multiplied by the learning rate and subtracted from the shared centroids from the previous iteration.

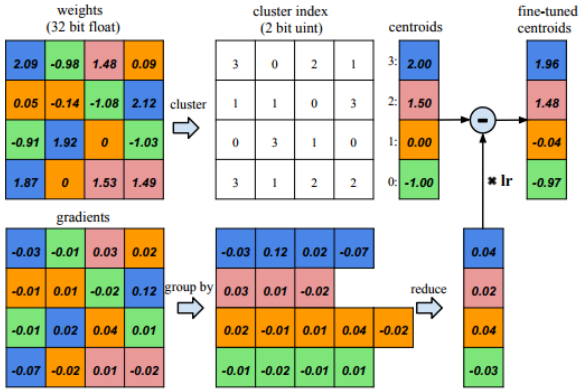


Figure 11 : Trained quantization by weight sharing (top) and centroids fine-tuning (bottom).

After implementing weight sharing, the weights become discrete, which mean that it is possible to use 4 bits to represent each weight without losing accuracy. When combining pruning and quantization, [12] shows that the network can be compressed down to 3% of its original size with no loss of accuracy.

In [13], another method called **low-rank approximation** is used for the convolution layers. It is based on the observation that the response at a position of a convolutional feature map approximately lies on a low-rank subspace. So if the response at one position of the feature map is $\mathrm{y} = \mathrm{Wx} + \mathrm{b}$, with $\mathrm{y} \in \mathbb{R}^d$, this convolution can be decomposed into two convolutions $\mathrm{y} = \mathrm{PW}'x + b'$ (*with* $\mathrm{W}'x \in \mathbb{R}^{d'}$ *and d > d')* which will have a lower combined complexity than the initial convolution. The matrix decomposition can be obtained by using Singular Value Decomposition. So this method entails to modify the architecture of the model by breaking down any low-rank convolution of filter size k and number of channels d into one convolution of same filter size and d' number of channels, plus another 1x1 convolution to scale up to d channels :

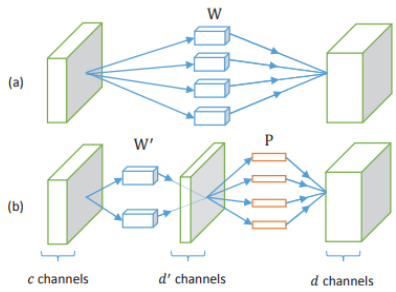


Figure 12 : Low rank approximation principle :
(a) An original layer with complexity *O(d.k².c)*
(b) An approximated layer with complexity reduced to *O(d'.k².c) + O(d.d')*

Finally, [14] has developed an algorithm based on **Recurrent Neural Networks** (RNNs) that is capable of extracting information from an image or video by adaptively selecting a sequence of regions or locations and only processing the selected regions at high resolution. It is called **Recurrent Attention Model**.

The attention problem is formulated as the sequential decision process of a goal-directed agent interacting with a visual environment. The system is trained using reinforcement learning methods which consist of adjusting the agent behavior based on a system of state/actions and rewards. Based on a given state, the agent takes an action and receives a reward for that action. Based on the magnitude of the reward, the agent modifies its behavior.

In this case, the state is the set glimpses taken so far in the image (the sub-regions already visited). The action consists in choosing the *(x,y)* coordinates (center of the glimpse) of where to look next in the image. The reward is positive if the image is correctly classified in the end otherwise it is zero.

On the MNIST dataset, the algorithm achieves accuracy results which are comparable to state of the art models. More interestingly, the overall capacity and the amount of computation of the recurrent attention model does not change from 60 × 60 images to 100 × 100 images, whereas the hidden layer of the standard convolutional networks which are connected to the linear layer grows linearly with the number of pixels in the input.

3.3 Deep Learning frameworks

In terms of tools, deep learning algorithms are usually developed on Python or R. However, a few frameworks have been developed to facilitate building models. The main ones are **TensorFlow** (from Google) and **PyTorch** (from Facebook). Those tools provide a more abstracted way of building complex deep learning models by easily stacking layers of different kinds (convolution layers, affine layers, easy choice of non-linear functions and of loss functions etc). On top of that, in the background, those tools build a computational graph corresponding to the model which is assembled, allowing them to automatically compute the gradient and perform the backpropagation. Those tools also take care of running everything efficiently on GPU.

The difference between those two tools is mainly that TensorFlow builds a static computational graph while PyTorch maintains a dynamic computational graph. Hence development on TensorFlow requires two phases: first, since the computational graph is static, we need to describe its nodes upfront and then in a second phase we run the graph multiple times using **sessions**. In PyTorch, the computational graph is built dynamically, so that there is no need to predefine the it upfront. Each forward pass rebuilds a new graph. The main advantage of static computational graphs is that they can be serialized, exported and run without access to the original code. Dynamic graphs cannot be separated from the code which generates them. However, they greatly facilitates coding many instructions like loops, conditional operations etc… Hence they make it easier to be creative in building models using RNNs, or combining together different existing models.

Besides, TensorFlow benefits from a larger community and has many high level libraries of top it (some of those libraries are Keras, TFLearn, Sonnet). In addition, TensorFlow has some pre-trained models (with the Keras or TF-Slim libraries) in order not to start from scratch, but from a model whose weights have already been learned from solving a similar problem. This process, called transfer learning, can speed up the training phase. However, it may be tricky to chose among all the available libraries. On its side, PyTorch is shipped with one high level library called *modules* which makes it unnecessary to use a third party library. This library also contains pre-trained models.

To wrap-up, we could say that PyTorch is more oriented towards research and allows to be creative and build more compact code. On the other hand, TensorFlow is very useful in deployment scenarios, in which we would need to make the graph run on different devices.

3.4 Existing end-to-end text recognition from raw pictures

Getting closer to our problem statement, this paragraph will investigate the existing solutions to the end to end problem of bib number recognition from raw pictures : going from a raw picture (amateur or professional photograph, frame from a video stream or webcam capture) to the annotation of the picture with bounding box around each bib of the picture labelled with the actual bib number.

As seen before, [1] used an MVP plus bERb algorithm to detect the characters and resorts to an OCR to read the characters. In the context of that paper, the algorithm was able to

recognize up to 9+ frames per seconds on a portable PC (2GHz @ 4 core CPU) using a multi-threaded application with a quality of match of 86%. However there is no mention on the dataset used for processing the images. Indeed, it is not clear whether or not the dataset is based on raw pictures possibly containing scenes with multiple bibs, multiple athletes and with possibly rich background components. In these harsh conditions, the definition of relevant extremal regions might prove to be challenging.

For instance, in [15], applying the text detection algorithm - in that case the stroke width transform (SWT) - to the entire image did not work reliably. Indeed, as pointed in that paper, the image background may be complex (urban scenes with vehicles, other humans, etc…). Besides, the raw pictures often contain irrelevant text (signs or billboard with printed text, text printed on people's clothes, license plates etc…). Finally, bibs usually cover a small area of the images. This led them to many false and missed detection. In order to address this issue, [15] built its algorithm in 3 phases, similar to the ones that we will use in this paper:

- First, they use a face detector to generate hypotheses regarding the bib location and scale (using the prior knowledge that the bib is located on a person's body)
- Then they apply the stroke width transform to detect the location of the tag, which is then processed
- Finally they feed the tag to an OCR engine

As shown in the figure below, this phased approach also allows them to avoid the false negative generated by irrelevant text which is often placed above or below the bib number (using an expected stroke width based on the size of the face). However in terms of throughput, no information was provided given that the focus was primarily on accuracy rather than speed.

Figure 13 : Tag Selection: **(a)** The tag is searched in the green bounding box; **(b)** two tags are detected – marked in blue. **(c)** the face scale is used to limit the size and stroke of the tag, and filter out wrong candidates.

Finally, as explained in [15], text recognition from raw pictures have been addressed extensively in the context of licence plate recognition (LPR). However the paper highlights

that most LPR methods work under restricted conditions, such as fixed illumination, one vehicle per image, specific location in the image, specific font or color, and non-complex background. In addition, a license plate is a well-defined rigid rectangular shape while a bib is wrapped on the runner's body, which entails a non-rigid transformation of the surface of the bib. For those reasons, [15] considers the bib number recognition problem to be more challenging than the license plate recognition one.

Finally even though [5] proposes a fully unified learned model to localize, segment and recognize multi-digit numbers from street level photographs, the photographs extracted from the dataset used for the paper are not really raw picture with complex backgrounds. Indeed, as shown below, they are already highly zoomed around the number to be recognize. Hence the model might not work reliably on pictures with more complex backgrounds and in which the number to be recognized covers only a small area with respect to the overall image area.

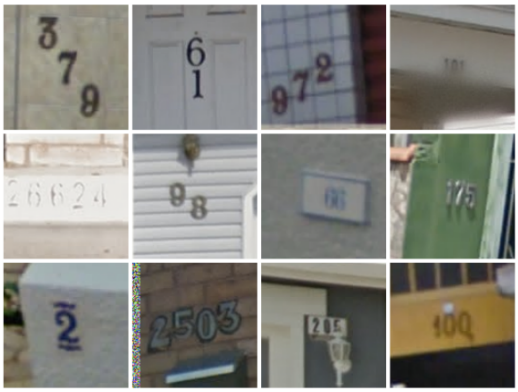


Figure 13 : Extract of the SVHN data set used by [5]

In conclusion, to the best of our knowledge there has not been yet any paper on an end-to-end, full-machine learning, bib number recognition model or on the processing time of an end-to-end bib number recognition algorithm.

4. Data

In order to propose a solution for the problem described earlier, we will adopt a full deep-learning-based algorithm. This paragraph will introduce the data which will be used to train this algorithm. The training data is based on the same image dataset as in [15]. The

dataset contains 217 color images and annotated bib numbers per image divided into three sets, each taken from a different race.

	Resolution	# of Images	# of RBNs
Set #1	342x512 - 480x720	92	100
Set #2	800x530 - 850x1260	67	77
Set #3	768x1024	58	113

Table 2 : Composition of our training dataset

The data used for testing the accuracy of the model is made of a few marathon images taken from google images and annotated using the labelImg tool. This tool allows to annotate images by drawing a bounding box around the object of interest (here the bib) and by adjoining a label to that box (here the bib number). The output is an XML file with sevaral entries, each one containing a reference to an image, the coordinates of the box and the label. We then convert XML file into a CSV file which is more convenient to import into a Python program. LabelImg is an open source tool available at https://github.com/tzutalin/labelImg

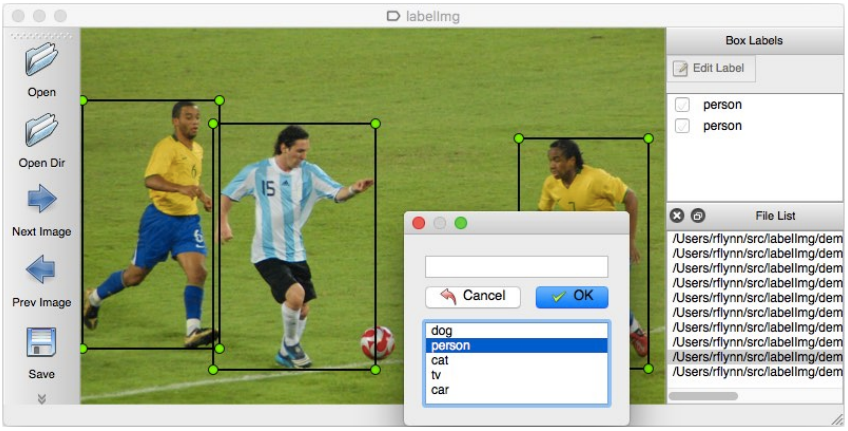


Figure 14 : Labelling tool LabelImg

Finally, in order to test the inference speed of our model, our algorithm was adapted to be tested on a video stream uploaded from youtube, representing a marathon finish line[1]. OpenCV for Python was used in order to read the video and convert it into a sequence of frames suitable to be served to our model. OpenCV was also used to visualize the output (annotated sequence of frame) as a video stream which can be saved as a standard video

1 *Link to the marathon video: https://www.youtube.com/watch?v=nKbKokxYgBs&t=601s&index=2&list=WL*

or displayed on screen. It is to be noted that OpenCV can be easily configured to read frames from a webcam which would allow to run the program in “real time”.

One of the components of our algorithm is a pretrained object detection API made available by Tensorflow [2]. As explained on the Tensorflow github page, the TensorFlow Object Detection API is an open source framework built on top of TensorFlow that makes it easy to construct, train and deploy object detection models. The models have been trained on the COCO dataset[3] containing 330,000 images (220,000 annotated images), 1.5 million object instances, 80 object categories and 5 captions per image. Out of these 80 object categories, we will only be using the “person” category.

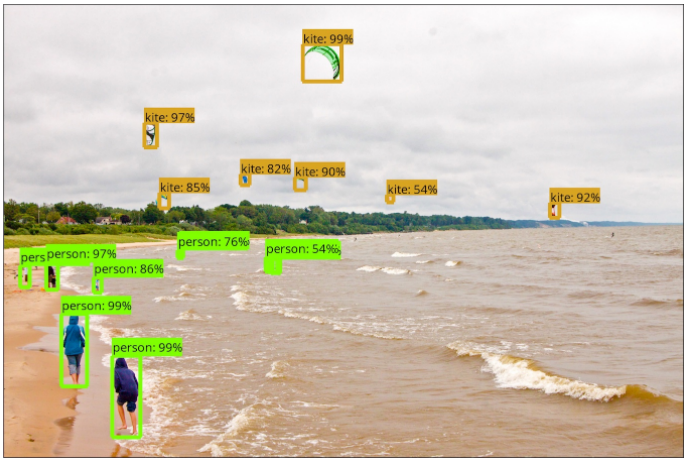


Figure 15 : example of an output from the object detection API

Finally, another component of our model is a deep learning algorithm which will eliminate the need to resort to an OCR tool. It is the algorithm described in [5] and in section 4.4, which has been train on street view images. The open source implementation of the model was made available on github[4] and was trained on the Street View House Numbers (SVHN) Dataset[5]. As displayed on the dataset webpage, SVHN is a real-world image dataset similar to MNIST (images are small cropped digits) but with significantly more image data (over 600,000 digit images) and coming from a significantly harder, unsolved, real world problem (recognizing digits and numbers in natural scene images). SVHN is obtained from house numbers in Google Street View images. As for MNIST, there are 10 classes (one for each digit). The volume of data is 73,257 digits for training, 26,032 digits

2 *https://github.com/tensorflow/models/tree/master/research/object_detection*
3 *http://cocodataset.org/#home*
4 *https://github.com/jiweibo/SVHN-Multi-Digit-Recognition*
5 *http://ufldl.stanford.edu/housenumbers/*

for testing, and 531,131 additional, somewhat less difficult samples, to use as extra training data.

Figure 16 : Another sample of the SVHN dataset

One of the inconvenient of the SVHN dataset is that it contains mainly street numbers with less than 5 digits. Hence the trained model performs very well on numbers with less than 5 digits (accuracy higher than 96% in the paper) but we get poor results on numbers with 5 digits or more. The main axis of enhancement for the model would be to enrich the dataset with numbers with more than 4 digits and retrain it.

5. Our contribution to the problem resolution

5.1 First approach : 2-module algorithm (interfacing a "re-trained" Object Detection API with the SVHN-trained algorithm) :

There already exists a deep learning algorithm (that we will call from now on SVHN algorithm) capable of reading a sequence of numbers from a natural image taken from Google Street view (based on paper [5]). A first simple approach is then to

(a) check that the SVHN-algorithm reads numbers accurately on images representing bibs (text recognition task)

(b) **Bib detection task:** build a deep learning module for the bib detection and segmentation tasks. It would

- take as an input a raw image (possibly containing several bibs)
- detect the bibs present in the image and draw a bounding box around each bib
- crop the image around the bounding boxes
- output the list of cropped images. Each cropped image would represent a bib and would serve as an input to the SVHN algorithm

(c) **Text recognition task :** feed the output images from (b) to the SVHN algorithm so that it can read the number on the bib and annotate it.

(d) **Vizualizaton task :** rebuild the initial image and adjoin the bounding boxes coming from (b) and the bib numbers obtained in (c)

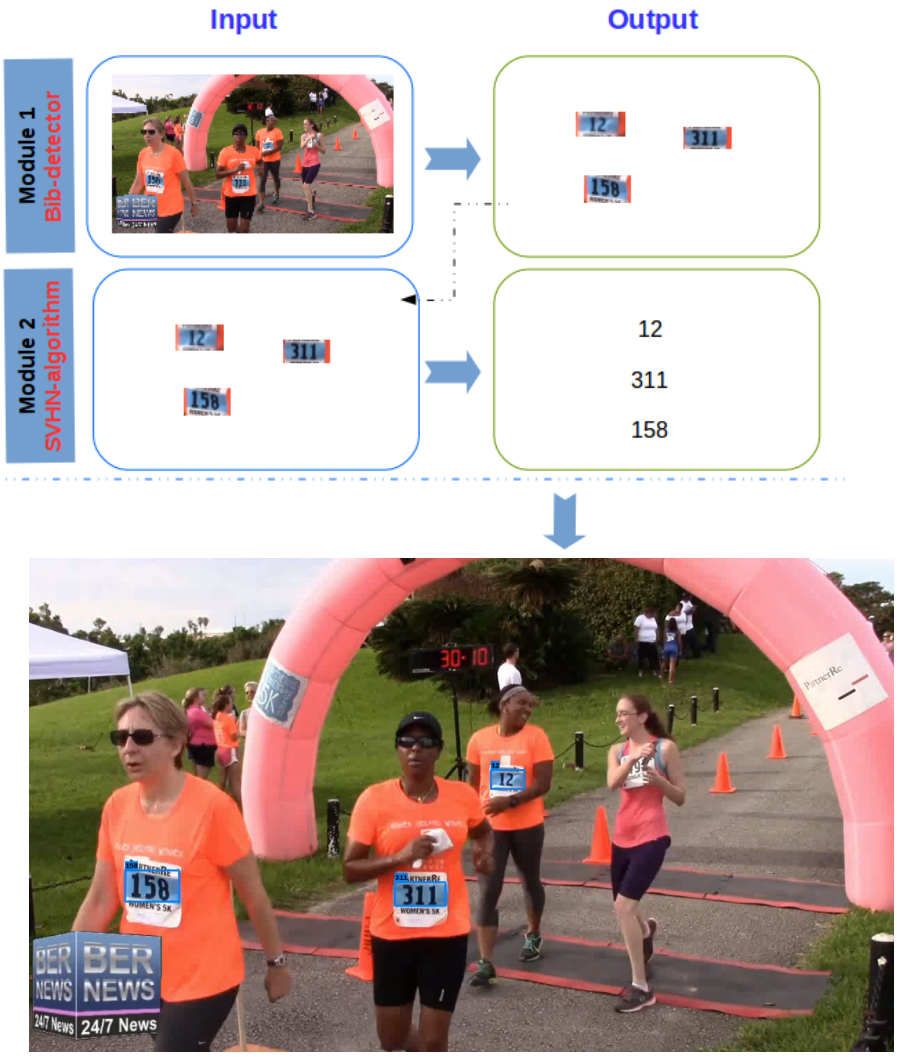
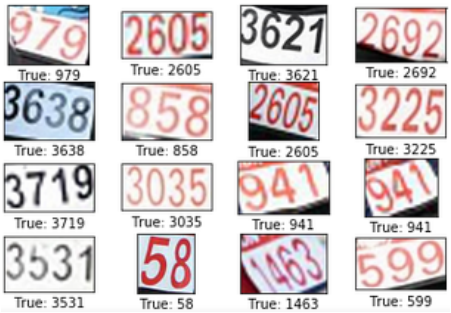


Figure 17 : Summary of the 2-phased approach

a) check that the algorithm performs well on images representing bibs:

In order to validate this point, we used the same dataset of marathon images taken from 3 different marathons. This dataset was already annotated with bounding box coordinates and bib numbers. The Python Imaging Library (PIL) was used to crop the images according to the bounding box coordinates in order to get single bib images like the ones below.



These bib images were then fed to the SVHN-algorithm. The main focus at this point was accuracy rather than speed since the objective was to assert the feasibility of using that algorithm to read bib numbers in various conditions. The accuracy reached 91% on the first set of marathon images. It scored only 77% when trained on all 3 sets of marathon images. This is explained by the fact that the other 2 sets contain bibs with more than 4 digits whereas the SVHN-algorithm has not seen a lot of 5 or more digit numbers during its training phase. Based on this result, we can conclude that the SVHN-algorithm is suitable to read cropped bib images which is not a surprising result given the similarity of those images with the ones from the SVHN dataset.

(**b) build a deep learning module for the bib detection and segmentation task:**

From the best of our knowledge, such a program, built with a deep learning framework, does not exist. As a result, we had to train a model to detect bibs and extract the regions containing bibs from the raw image.

In order to detect bibs in an image, we used as a basis the architecture of the Object detection API from Tensorflow. Tensorflow proposes several architectures for the object detection problem. Among them SSD Mobilenet architectures are those with the smallest inference delay. The SSD architecture has been covered in the literature review. Mobilenet is a light CNN architecture conceived to be working effectively on common devices like mobile phones. Inference time of Mobilenet is kept low for example by using 1x1 convolutions as seen in the literature review. This architecture is already so efficient and compact that the accuracy is very sensitive to pruning [17]. SSD-Mobilenet without pruning will be then selected as the architecture shell for our bib detection and segmentation component.

To be noted that other architectures with better mAP accuracy are available in Tensoflow, but the show significantly higher inference delay[6]:

Coco Model Name	Speed (ms)	mAP^[1]
SSD Mobilenet V1	30	21
SSD Mobilenet V2	31	22
SSD Lite mobilenet v2	27	22
SSD inception v2	42	24
Faster RCNN Inception v2	58	28
Faster RCNN ResNet 50	89	30
Faster RCNN ResNet 50 Low Proposals	64	30
RFCN-ResNet 101	92	30
Faster RCNN ResNet 101	106	32
Faster RCNN ResNet 101 Low Proposals	82	32

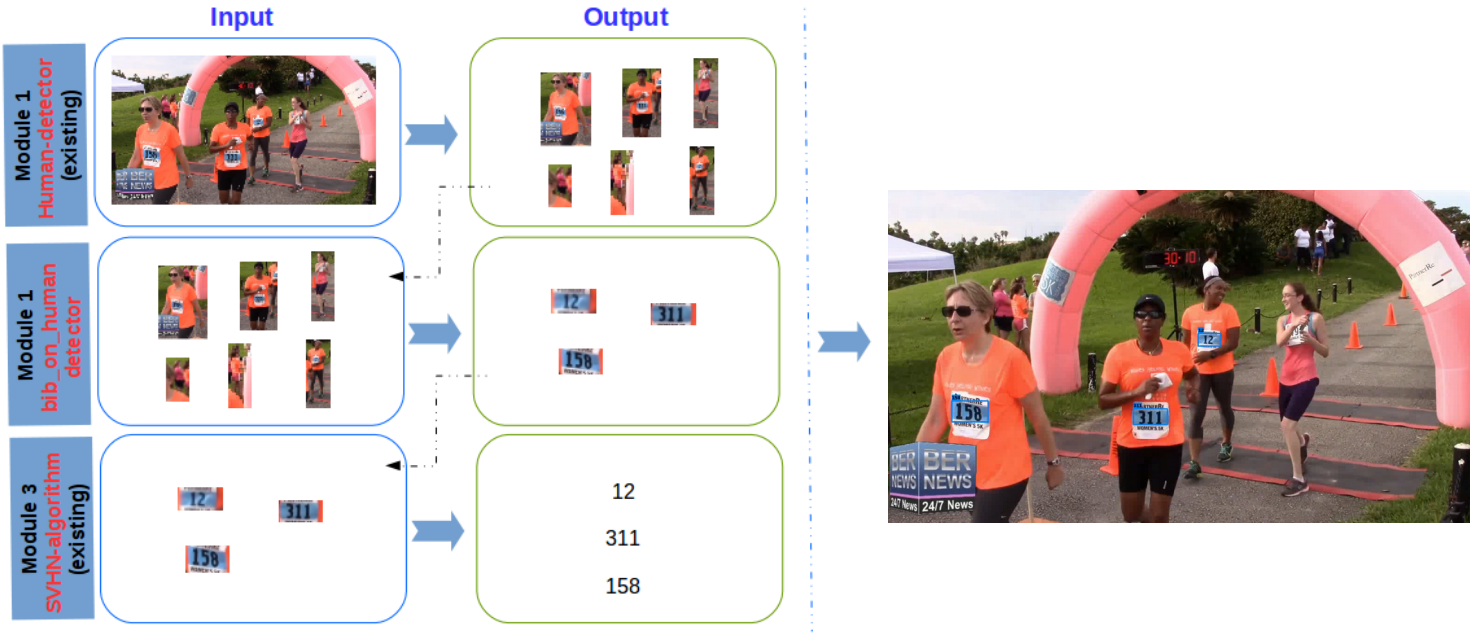
In addition, as noted in [18], an intensively trained Mobilenet model can equate the accuracy and precision on output object detection of a Faster RCNN model.

Subsequently, we kept the same model architecture and re-trained the model to detect bibs from our raw images. For the training process, we used the data from [15] described in the Data section of this document. The model was trained for about 12 hours on a Tesla K40 GPU. The overall loss plateaued at an average of 1.5 but with very high variance from one batch to another going from 0.8 to 3. Indeed the resulting annotated images showed poor performance of the algorithm. Indeed, close to 8 images out of 10 showed completely misplaced bounding boxes, indicating the presence of bibs on elbows, on billboards and signs containing text etc…

These results reflect well the point made by [15] that trying to recognize bibs directly in the image is not recommended due to the richness of the background and the small area covered by a bib with respect to the overall image area. Hence another approach had to be conceived to ease the work of the object detection API. This is where we introduced the 3-module algorithm in place of the 2-module algorithm.

6 running time are provided in ms per 600x600 image (including all pre and post-processing) ; these timings were performed using an Nvidia GeForce GTX TITAN X card

5.2 Second approach: 3-module algorithm ("pre-trained" Human Detection API - "re-trained" bib detection API - SVHN-trained algorithm)



The workflow for this second approach is as follows:

(a) **Human detection task** : detect the humans present in the image .The areas of the picture which contain a human will be then cropped out and saved as unique images for later use

(b) **Bib-on-human detection task** : detect the bibs present on the human pictures resulting from step (b). Crop the images around the bounding boxes and save each bib image in a single file

(c) **Text recognition task** : as in the previous approach, feed the output images from (b) to the SVHN algorithm to read the number on each bib image

(d) **Visualization task** : rebuild the initial image with annotations, using the bounding box coordinates from (a) and (b) and the bib numbers obtained in (c)

(a) Human detection task

There is already an existing module to solve this task. Indeed, the pre-trained version of Tensorflow Object Detection API has the "human" category in one of the 80 object categories it was trained to recognized. The output of this module is bounding boxes around the areas containing humans with a confidence factor of the presence of humans.

Since we are expecting marathon runners to be well defined on the marathon pictures we are using, we consider define a threshold on the confidence factor. This threshold is tuned empirically, with a final value at 50% for our experiment.

We also build an interface using the PIL library to crop the images around the bounding boxes and save each detected human in a separate image.

As a result of this step, we get a new database of images containing single bib racers with little to no background. This will considerably ease the work of our bib detector. This step is similar to the approach used by [15], where they first recognize human faces before looking for bibs. Hence this step turned out to be of high importance for the success of the overall algorithm despite the fact that it will consume a significant amount of resources.

(b) Bib-on-human detection task

We had to build this module since there is no existing module allowing to recognize the presence of a bib on a human body.

In order to specialize this module and make it more accurate, it will be train on images zoomed onto a single marathon racer. This allows us to capitalize on step (a).

We reuse the images of racers resulting from step (a). We annotate each racer image using the LabelImg tool to draw a bounding box around the bib and convert the resulting XML file into CSV in order to use it for our learning algorithm.
As for our previous approach, we train the bib-recognition learning algorithm on the same Tesla K40 GPU chip. This time, the loss converges quickly to an average of around 1 in less than 2 hours. We let the algorithm train for another one hour to get an average loss lower than one with much less variance from one batch to another compared to the previous approach.

Unlike the previous approach, this phased approach produces a reliable algorithm which consistently detects bibs and accurately draws bounding boxes around them.

Finally we build an interface with the text recognition module. Indeed, for each image, if there is a detected bib, this bib is cropped out around the bounding box and converted into a separate image which will be passed to the text recognition module.

(c)Text recognition task

We are using the existing open-source implementation of the algorithm conceived in [5] and presented in the state of the art. As explained before, the algorithm was trained on highly zoomed images and is not suitable for images containing a rich background around the text to be read. Here, as inputs, we will be using the bib images resulting from (b) which are zoomed around the bib. We were then able to reproduce the good result that we achieved with the algorithm in step (a) of the previous approach with more than 91% accuracy for the first set of images.

(d)Visualization task

After having implemented this 3-step approach, we need to check that, for our overall task of reading bibs in rich-background images containing multiple racers, we consistently get good results. For this we need to visualize the results on these images.

In order to visualize the result, at each step, we kept the relative coordinates of the bounding boxes resulting from that step. We then use these relative coordinates to draw the bounding boxes back in the raw image, and annotate each one with its bib number The process is as follows:

- from the raw image, we save the coordinates of each human detected in step (a). We then apply the algorithm that we conceived in step (b) concurrently to each human image extracted from the raw image
- from each human image, we save the coordinates of one bib (if there exists one) relative to this human image dimensions
- We then apply the algorithm from step (c) concurrently to each bib image resulting from each human image and collect the bib number
- finally we calculate the coordinates of each bib of the raw image based on the human coordinates and the bib coordinates relative to each human image. We draw a bounding box around each bib in the raw image and annotate it with its corresponding bib number.

The same succession of steps is used to annotate a video stream. We simply annotate the video stream frame by frame. In Python, this is made possible by using the OpenCV library for Python in order to convert the video into a sequence of images and to convert the annotated images back into video format.

6 Inference time improvement

All the inference test we run from now on are run on Google Cloud instances with the following setup:

- 4 CPUs Intel Xeon E5 2.2GHz
- 1 NVIDIA Tesla P100 GPU

In order to get an average throughput over a large number of images, we run the model on a video stream and measure the processing time over each frame of the image.

The processing time is measured from the moment the image is loaded into a Python program in a well formatted array to the moment we produce the Python dictionary containing the bib number with the right coordinates in the raw image. Hence the time to load the image in the program and the time to produce the visualization in the form

of an annotated image or a video is not included since it depends on the framework and interface used with the video recorder or screen. In addition, optimizing those steps is out of the scope of this study.

Besides, we also break down the total processing time of a frame into 3 pieces, each one representing the separate processing time of each of the 3 phases of our approach.

6.1 Base results

Below are the first results obtained with our 3-phase model, without any additional improvement.
The average time to process one image (over the entire video) is 9.65 seconds. The inference time spreads from 17 seconds for a few images to 3 seconds for other images. So with these results we are far from real time inference. A lot of fine tuning is required.

6.1 Optimizing session openings

The standard object detection API proposed in Tensorflow is build for single image processing. It means that if the program has to process 300 images, it will open 300 new Tensorflow sessions and reload the inference graph and the variables 300 times. The graph is the file in which the structure of the model is saved. The value of the parameters of the model are also saved in a separate file and need to be restored for each new session. As explained in the Tensorflow website, a Tensorflow session is a class which represent a connection between the client program (here our Python program) and the C++ runtime. A session object provides access to devices in the local machine and allows to actually run the operations of the Tensorflow graph.q

The same goes for the SVHN algorithm, for which the graph is reloaded for every new image as we can see in the screenshot below:

```
3 : 5.752 sec
3 : 5.707 sec
3 : 5.734 sec
3 : 5.453 sec
3 : 5.503 sec
INFO:tensorflow:Restoring parameters from /home/jlma/models/research/object_detection/SVHN/checkpoints/model.ckpt
-328000
3 : 9.431 sec
3 : 5.502 sec
INFO:tensorflow:Restoring parameters from /home/jlma/models/research/object_detection/SVHN/checkpoints/model.ckpt
-328000
3 : 8.964 sec
INFO:tensorflow:Restoring parameters from /home/jlma/models/research/object_detection/SVHN/checkpoints/model.ckpt
-328000
3 : 9.400 sec
```

Opening a new Tensorflow session is an expensive operation. It is then obvious that the first improvement to make is to limit the number of session opening as much as possible. The ideal would be to open one session for each module to use for the entire video.

We modify the Tensorflow Object Detection API and our main program so that instead of opening a new session for each new image, we open 3 sessions once for all:
- one session for the human detection graph
- one session for the bib detection graph
- one session for the number recognition graph

For each of those graphs, the saved instances of the model parameters are restored once and the same set of sessions is reused during the whole video stream processing. Below are the results compared to our previous instance of the model.

Average inference times	No improvement	adding session optimization
Phase 1 (sec)/frame		0,114
Phase 2 (sec)/frame		0,061
Phase 3 (sec)/frame		0,023
Total (sec)/frame	9,650	0,200
Equivalent number of fps	0,10	5,00

This improvement was the low hanging fruit which provided a significant improvement. The additional improvements to make will be less obvious.

6.3 Custom task allocation between CPU and GPU

The allocation of operations between CPU and GPU is not optimal in the Tensorflow Object Detection graph. As explained on the following github page[7], we can analyse the running time of each node of the standard Object Detection graph. The result is that CNN-related operations (the first part of the graph before the fully connected layers) run much faster on GPU. However, the post-CNN operations run much faster on CPU.

This is probably due to the fact that GPUs are very good at repetitive tasks which can be parallelized, which is the case for convolutional operations. However, fully connected layers offer much less parallelization opportunities. They rather have sequential operations. So the expensive transfer of data between the GPU memory and system memory are not compensated by the gains in operation executions.

As a result, we manually split each of our detection graphs (human detection graph and bib detection graphs) into two sub-graphs at a specific node after the CNN convolutions. We use the Tensorboard API to visualize the nodes of the graph and appropriately select the separating nodes. Thanks to the Tensorflow API, it is possible to assign a graph to a specific device (CPU or GPU) in just one line of code. Hence, we assign the first sub-graph operations to GPU devices and the second sub-graph operations to CPU devices. Below are the results following this change:

	Incremental optimization operations from left to right		
Average inference times	No improvement	adding session optimization	adding CPU/GPU custom allocation for phase 2
Phase 1 (sec)/frame		0,114	0,057
Phase 2 (sec)/frame		0,061	0,060
Phase 3 (sec)/frame		0,023	0,023
Total (sec)/frame	9,650	0,200	0,139
Equivalent number of fps	0,10	5,00	7,19

This improvement was made only for the phase 1 module. It did not work for the phase 2 module since we used a different object detection version which I lighter than for the

7 *https://github.com/tensorflow/models/issues/3270*

phase 1 but a little bit less accurate. This choice was made since phase 1 is executed only once per image while phase 2 can be called multiple times per image. So one axis of improvement for those results would be to find an optimal custom allocation for phase 2 object detection algorithm. In this work, the split points we found did not work properly (see results below).

	Incremental optimization operations from left to right			
Average inference times	No improvement	adding session optimization	adding CPU/GPU custom allocation for phase 2	adding CPU/GPU custom allocation for phase 3
Phase 1 (sec)/frame		0,114	0,057	0,057
Phase 2 (sec)/frame		0,061	0,060	0,069
Phase 3 (sec)/frame		0,023	0,023	0,023
Total (sec)/frame	9,650	0,200	0,139	0,148
Equivalent number of fps	0,10	5,00	7,19	6,78

Regarding the SVHN algorithm, the implementation code already contains a custom allocation between CPU and GPU which seems to be sensible.

6.4 Parallelization

Coming back to our overall approach, for a specific image, we first detect all the humans in the image having a confidence factor over our predefined threshold (50% here); then we run the bib detection algorithm (phase 2) and SVHN agorithm (phase 3) on each human's picture. Hence we conclude that, for a specific picture, if we recognize 10 people in the picture, it is possible to run phase 2 and phase 3 in parallel over the 10 human pictures. This is what we are trying in this section. We are using Python's Threading library to encapsulate phase 2 into Thread class. In this experiment we are creating 3 more threads for phase 2. The results are below :

	Incremental optimization operations from left to right			
Average inference times	No improvement	adding session optimization	adding CPU/GPU custom allocation for phase 2	Adding phase 2 parallelization
Phase 1 (sec)/frame		0,114	0,057	0,058
Phase 2 (sec)/frame		0,061	0,060	0,051
Phase 3 (sec)/frame		0,023	0,023	0,022
Total (sec)/frame	9,650	0,200	0,139	0,129
Equivalent number of fps	0,10	5,00	7,19	7,78

The inference time is lower than previously. The improvement is not significant because we have to keep in mind that Python's Threading library does not allow GPU programming. Hence, the resulting phase 2 threads are all run on cpu.

If we include phase 3 in separate threads as well, we are lowering the inference time even further:

	Incremental optimization operations from left to right				
Average inference times	No improvement	adding session optimization	adding CPU/GPU custom allocation for phase 2	Adding phase 2 parallelization	Adding phase 3 parallelization
Phase 1 (sec)/frame		0,114	0,057	0,058	0,058
Phase 2 (sec)/frame		0,061	0,060	0,051	0,064
Phase 3 (sec)/frame		0,023	0,023	0,022	
Total (sec)/frame	9,650	0,200	0,139	0,129	0,120
Equivalent number of fps	0,10	5,00	7,19	7,78	8,33

Increasing the number of threads does not improve the inference time. Indeed, a raw image seldom contains more than 4 humans with 50% confidence factor, at least in the video we are using.

We are reminding that, the inference times presented above are average inference time throughout the full video we are using. Variance is quite high since in some frames there is no runner on the image (so no time lost in phase 2 and 3) while in other frames, we have several runners identified for whom we need to locate and read the bibs. So inference time goes anywhere from 0.04 second per image to 0.4 second per image.

7 Conclusion and perspectives

Our aim with this work was to build an end-to-end bib recognition and reading algorithm which could work in real time situations. We showed that deep learning models are powerful enough to take care of the end-to-end process with reasonable accuracy. Most previous work in the area or in close areas (license plate recognition) performed the bib region detection phase separately and fed the resulting characters to an OCR tool.

We also showed that, with a series of improvements, it is possible to make significant gains in inference time with those powerful models. We optimized the number of session openings in our models, enhanced the allocation of operations between CPU and GPU and used parallelization to speed up the process. The result is an average 8+ frames per second processing on a finish line marathon video. We did not find comparable results since most publish results in the area only focus on the region detection task and do not include the downstream operations (ordering the characters, passing the characters to an OCR) or upstream operations (going from a rich-background image to zoom onto a specific runner). Should we consider only our bib detection phase (phase 2), we get a 19+ frames per second rate. Should we consider only the bib detection phase starting from a raw image (phase 1 and phase 2), we get a 9+ frames per second rate.

our overall rate of 8+ frames per second is still lower than the expected 10 frames per second in order to comfortably run in real time situations. However, this number can be improved following a few of the suggestions below:
- improve the allocation of tasks between CPU and GPU for phase 2
- include the GPU in the parallel programming

Another axis of enhancement for our model would be to enrich the SVHN dataset with numbers with more than 4 digits and retrain it in order to get a more generalized algorithm.

8 References

1 '*Fast Detection of Text in a Natural Scene using Blobs*'.

2 Goodfellow I., Benjio Y., Courville A., *Deep Learning*, , MIT Press, 2016.

3 Rane M.A., '*Fast Morphological Image Processing on GPU using CUDA* ', *IEEE Transactions on Communications*, 40 (8) : College of Engineering, Pune , 2013.

4 www.tensorflow.org, MNIST For ML Beginners, 2017.

5 Goodfellow I., Bulatov Y., Ibarz J., Arnoud S., Shet V. '*Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks'*, ICLR, 2014.

6 Matas J., Chum O., Urban M, Pajdla T., *'Robust Wide Baseline Stereo from Maximally Stable Extremal Regions',* BMVC, 2002.

7 Dalal N., Triggs B., *'Histograms of Oriented Gradients for Human Detection', CVPR,* 2005.

8 Wang K., Babenko B. and Belongie S., *'End-to-End Scene Text Recognition', ICCV,* 2011.

9 Lecun Y., Boser B., Denker J. S., Henderson D., Howard R. E., Hubbard W., Jackel L. D., *'Backpropagation applied to Handwritten Zip Code Recognition',* 1989.

10 Girshick, R., Donahue, J., Darrell, T., and Malik, J. *'Rich feature hierarchies for accurate object detection and semantic segmentation'*. Technical report, arXiv:1311.2524, 2013.

11 Shaohuai S., Qiang W., Pengfei X., Xiaowen C.,Benchmarking *'State-of-the-Art Deep Learning Software Tools',* arXiv:1608.07249v7, 2017.

12 Han S., *'Efficient Methods And Hardware For Deep Learning',* Stanford University, 2017.

13 Zhang X., Zou J., Ming X., He K., Sun J., '*Efficient and Accurate Approximations of Nonlinear Convolutional Networks'*, CVPR 2015.

14 Mnih V., Graves N. H. A., Kavukcuoglu K., 'Recurrent Models of Visual Attention', Google Deep Mind, ArXiv: 1406.6247, 2014.

15 Ben-Ami I., Basha T., Avidan S., 'Racing bib number recognition', School of Electrical Engineering – Tel Aviv Uniersity,2016.

16 Liu W., Anguelov D., Erhan D., Szegedy C., 'SSD: Single Shot MultiBox Detector', ArXiv: 1512.02325, 2016.

17 He Y., Han S., 'ADC: Automated Deep Compression and Acceleration with Reinforcement Learning', Google Brain, Xi'an Jiaotong University, 2018.

18 Fleury D., Fleury A., 'Implementation of Regional-CNN and SSD Machine Learning Object Detection Architectures for the Real Time Analysis of Blood Borne Pathogens in Dark Field Microscopy', John Marshall, University of Minnesota, 2018.