INF473-V Project report : EfficientNet for object detection

Reda Belhaj-Soullami
reda.belhaj-soullami@polytechnique.edu

Amine Rabhi
mohamed.rabhi@polytechnique.edu

Abstract

In this project, we investigate a compound scaling method for deep convolutional neural networks called EfficientNet [6]. To demonstrate the relevance of compound scaling, we apply this scaling method to a ResNet baseline network [3] on the CIFAR-10 dataset. In a second part we use the EfficientNet method to produce a backbone network for a R-CNN network, for the task of object detection on a Kaggle dataset.

1 Description of the EfficientNet scaling method

In recent years, deeper and more complex networks have proven to achieve spectacular performance in computer vision tasks such as image classification and object detection. This is why scaling a network (i.e. augmenting its size) generally helps improving the model's performance. Yet, this comes with the cost of a longer training time. Therefore, a compromise needs to be made between complexity and training time. Usually, this is done manually by the practitioner via trial and error. Not only is this a tedious task, it is also suboptimal. The original paper [6] gives a general method for scaling deep CNNs efficiently.

1.1 The three ways to scale a CNN

There are three main ways to scale up a CNN. For simplicity, we will assume that the cost of training a CNN (i.e doing forward and backward passes) is dominated by the cost of the convolution operations.

- **Depth :** This means increasing the number of layers of the network. A deeper network will detect more abstract and more complex features. Note that multiplying by α the number of layers of a network will approximately multiply the numbers of operation per second (FLOPS) it does by a factor α.
- **Width :** This means increasing the number of channels of the convolutional layers of the network. This will help the network capture fine features of the image, while using a small network (with a small number of layers). Note that multiplying by β the numbers of channels in the network will multiply the number of FLOPS by β^2.
- **Resolution :** This means increasing the resolution of the images feeding the network. With a higher resolution, the network will be able to see more details in the image and therefore learn more detailed features. Note that multiplying by γ the input resolution will multiply the number of FLOPS by γ^2.

1.2 The efficient scaling method

The method described in [6] consists of combining the three ways of scaling. This is called compound scaling. The rule for scaling the network is the following :

- depth : $d = \alpha^\phi$

- width : $w = \beta^{\phi}$
- resolution : $r = \gamma^{\phi}$

ϕ is a user-specified coefficient that controls the amount of resource the user is willing to use. In order to control the amount of supplementary resource used, we set the constraint $\alpha \times \beta^2 \times \gamma^2 \simeq 2$. This way, when scaling using compound coefficient ϕ, we know we will use approximately 2^{ϕ} more FLOPS. To choose the α, β, γ coefficients, we do a small gridsearch on our basis network to find the coefficients that gives the best accuracy.

This is the final algorithm for finding α, β, γ :

Algorithm 1 Finding α, β, γ

Inputs : a baseline model with resolution r_0, width w_0, depth d_0
  best $= 1, 1, 1$, best accuracy$= 0$
  for α, β, γ in the search space * **do**
    Train the network with $d = \alpha d_0$, $w = \beta w_0$, $r = \gamma r_0$ and get its accuracy a
    if a $>$ best accuracy **then**
      best $= \alpha, \beta, \gamma$
      best accuracy $=$ a
    end if
  end for
  return best

* the search space is (a discretized version of) the set of numbers $\alpha, \beta, \gamma \geq 1$ such that $\alpha \times \beta^2 \times \gamma^2 \simeq 2$

1.3 The EfficientNet networks

In the article, the aforementioned method is applied to a particular baseline network called EfficientNet-B0 (which is similar to M-NASNet [5]) to produce EfficientNet-B1 to EfficientNet-B7 with the scaling method. EfficientNet-B7 achieves state-of-the art accuracy on ImageNet (while being 8 times smaller and 6 times faster than best CNNs), as well as on other datasets [6].

We used ResNet [3] as a baseline network because we are more familiar with it. Moreover, as the method is designed to work for any kind of convolutional networks, we should obtain good results as well.

2 Description of the R-CNN method for object detection

R-CNN [2] is an object detection method that uses the SelectiveSearch algorithm for region proposals. In this section we briefly explain how the R-CNN algorithm work. For technical details about the algorithm we refer to [2].

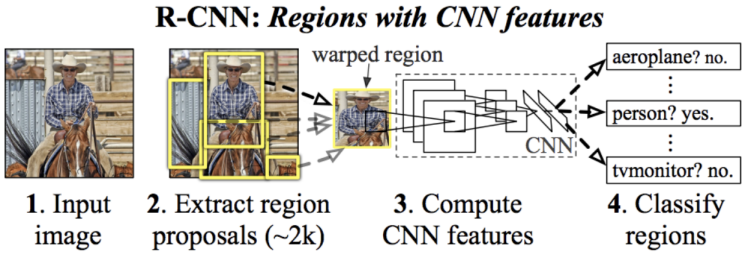

Figure 1: RCNN method

First, we get regions proposals from the Selective Search algorithm. Each region is passed through a pre-trained CNN (we use a scaled ResNet), giving a feature map with fixed format. The feature

map passes then through a bounding-box regressor and a classifier with 5 classes (4 corresponding to the actual classes and one class representing the background class). Only the boxes that do not correspond to the background are therefore displayed.

Now we briefly explain the training process.

- First, we get region proposals from the Selective Search algorithm.
- Then we construct our batch. Each batch consists of 100 positive regions (meaning that they have more than 0.5 IoU with a ground truth box) and 28 negative regions (they have less that 0.5 IoU with all ground truth boxes).
- For every region in the batch, we compute the loss.
  - If the region is positive, the loss in the sum of a classification loss (cross-entropy loss) and a bounding box regression loss (L1-Loss).
  - If the region is negative, the loss is the classification loss. In this case, the ground truth label is "background".

3 Description of our implementation

3.1 Application of the scaling method

3.1.1 The ResNet architecture

ResNet [3] is a deep CNN architecture developed in 2015 that enables to train deep CNNs (that are usually hard to train), using a technique called "identity shortcut connection" that skips layers. The main component of a ResNet is called residual block, which is a block of two convolutional layers that uses the skipping connections technique. This technique is illustrated in Figure 3

A ResNet consists of four stacks of various residual blocks. Between two stacks, resolution is divided by 2 and width is multiplied by two. The structure of a ResNet is illustrated in Figure 2.

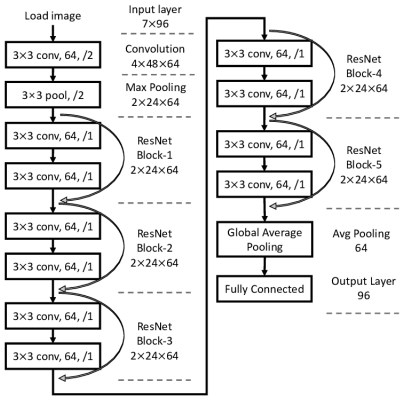

Figure 2: Structure of a ResNet

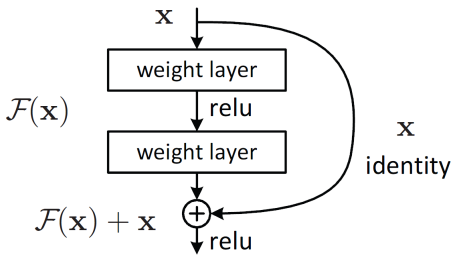

Figure 3: A skip connection

We used our own implementation of ResNet (with PyTorch). We could not use existing implementations mainly because they have a fixed depth (e.g ResNet50, ResNet152, etc...) and width. We implemented a class that enables us to choose the resolution, the width and the depth that we want. The depth is represented as an array of length 4, representing the number of blocks in each stack. The resolution that we will use are not necessarily powers of 2, so we had to be careful when computing the resolutions along the network. More precisely, we discovered that PyTorch's convolutional operation yields a new resolution of $r/2$ when r is even and $(1+r)/2$ when r is odd.

Finally, we coded a single class representing a ResNet with variable width, scale and resolution parameters. As we will need to access the model's resolution during training and testing (see paragraph 3.1.3), we provided the class with a `resolution` attribute.

3.1.2 Scaling the three dimensions of a ResNet

First, we should determine how to scale up depth, accuracy and resolution of a ResNet network. More precisely, for any given α, β, γ such that $\alpha \times \beta^2 \times \gamma^2 \simeq 2$, we need to find a method that scales the networks depth by a factor α, width by a factor β and resolution by a factor γ. As the original paper did not use ResNet as its basis network, we had to figure out the details ourselves.

- Scaling the resolution is pretty straightforward : we set our first layer to the desired resolution.
- To scale the depth, we have to multiply the number of layers by α. We decided to keep 4 piles and to add blocks in each pile until the number of blocks is scaled by a factor α. This way we ensure that in total we scale the total number of layers by a factor α.
- To scale the width, we have to scale the total number of channels of our network by factor β. In a ResNet architecture, the number of channels is controlled by the numbers of channels of the first layer, as the number of channels is doubled at every new stack. Hence we only need to scale the number of channels of the first layer.

3.1.3 Dealing with different resolutions

Training models with different width and depth is fairly easy as there is nothing to modify in the training and test set images. However, when using models with different resolutions, it can be tricky. In order to avoid re-downloading the dataset for every new model, we use bilinear interpolation on the fly during training : we load the data from a single dataloader (that yields images of a fixed resolution) and interpolate the image to the desired resolution before feeding the image to the network. We use the same technique at test time. Note that we used a PyTorch function for bilinear interpolation.

3.1.4 Dealing with GPU memory

We used GPUs for training our models. GPUs have limited memory, so we had to make sure we never exceed the maximum memory of the GPU. To do so, during training, we make sure to always use the same amount of memory for all models. This means that for a bigger model, we have to use a smaller batch size. More precisely, if the batch size used for our baseline model is $b_0 (= 220$ in our case), when using a network scaled with parameters α, β, γ we use a batch size of $\frac{b_0}{\alpha \times \beta^2 \times \gamma^2}$. Generally, to speed-up the training of our networks, we tried to free GPU memory as much as possible by deleting the Python objects when we do not need them anymore. There is however a drawback from using GPUs, namely the fact that GPUs sometimes fail to produce reproducible results. We address this problem in section 3.1.5.

3.1.5 Training the models

When using the whole CIFAR-10 dataset, training the models that we used took a lot of time, even with GPU power. One epoch of training takes from 7 minutes (baseline network) to 1 hour (most scaled network). This was a major issue for our experiments. Indeed, as mentioned in the original article, the effects of scaling properly are small (usually only a few points of accuracy). In order to properly see those effects, our training must be stable enough.

Improving the stability of our models To make sure that our training is stable, the training must be done until convergence. Unfortunately, making large networks converge usually takes a lot of time (Fig. 4). For instance, a basic ResNet18 will take four hours to converge when training on the whole CIFAR-10 dataset (50000 images). A larger network could take more hours or even days. Consequently, we had no choice but to use a smaller dataset to reach convergence more quickly (we used only 1500 to 2000 images for training). We also used a smaller baseline network that ResNet18, namely a network with 95×95 resolution images, a width of 12. The depth is however the same as

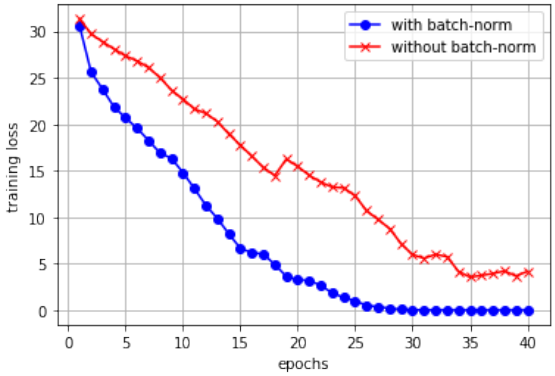

Figure 4: Training loss of ResNet18, using 7000 training images with and without batch-normalization

ResNet18, as we wanted to keep at least two blocks per layer in our network.

To make the accuracy results even more stable, we used k-fold cross-validation, which consists in repeating the training on k independent subsets of the dataset, and using the mean of the accuracies as an accuracy measure. We also used batch-normalization to increase stability. The effects of using batch-normalization are presented in Figure 4. We can clearly see on this figure that batch-normalization makes the training smoother. It also increases the quality of the training, as the training loss converges better.

At this point, to improve stability and consistency, we could use more training examples, which increases the training time. Therefore, there is a trade-off between good stability and long training time. Due to our limited resources, we could not get very stable results. To take into account this difficulty, we used error bars on our accuracy graphs, that represent the standard deviation of the list of accuracies obtained when executing cross-validation.

Using a smaller training set (7000 training examples instead of 50000), we applied the scaling algorithm to find the α, β, γ coefficients. We used those same coefficients to produce a single scaled classifier trained with the whole dataset (50000 training examples). To justify this method, we make the assumption that the best scaling coefficients α, β, γ should only depend on the architecture of the network and the specifications of the dataset, and not on the number of training examples we use at each training, provided the training remains stable enough. Of course we cannot prove this but it seems fairly reasonable, especially when looking for a universal way of scaling CNNs. Moreover, we do not really have a choice here because if we decide to train the models properly on the whole dataset, the execution of the gridsearch algorithm would probably take weeks.

3.2 Implementation of R-CNN

We implemented the algorithm in PyTorch, using our own network as the pre-trained backbone CNN.

3.2.1 The pre-trained CNN

We trained our backbone model on a Kaggle dataset [1], composed of images of 4 classes : buffalos, rhinos, elephants and zebras. The dataset contains only 2000 images, so we used data augmentation (random horizontal flip, rotation, and color jittering, using PyTorch's `torchvision.transforms`. We also used a re-sampling technique to get a balanced set of the 4 classes.

3.2.2 The R-CNN implementation

The R-CNN was represented as two `nn.Module` classes (the classifier and the regressor). The forward pass consists in passing the image through a pre-trained CNN and then passing the result

[1] `https://www.kaggle.com/biancaferreira/african-wildlife`

to the classifier or the regressor. The only trained weights during training, are the one of the classifier and the bounding-box regressor. Unlike the original architecture [2], we did not use one bounding-box for each regressor class, but only one global regressor.

The training loop is different from the one of a classification task. First, each batch has to be sampled manually, because of the proportions of positive regions and negative regions needed. We did not use the proportions specified in the article, because our SelectiveSearch algorithm did not sample as much negative regions as the one in the article.

The loss is calculated on each region of the batch before doing a step of optimization. The learning rate has to be smaller for this task, and according to the article [2], L1 regularization should be used.

We used PyTorch losses (Cross-Entropy Loss and Smooth L1 Loss) in order to compute the gradients and backpropagate automatically. Most of the training time is due to batch-sampling. For these operations, using a GPU does not really help. For this reason, the training time is still slow even though only a few layers are trained.

Our implementation differs from the implementation described in [2] in numerous points. For example, we trained the regressor and the classifier at the same time. We trained the regressor using L1 loss, whereas in the article they used SVM regression, which has a closed-form expression. We also did not incorporate various tricks mentioned in the article, mainly because details about their implementation were omitted. Those tricks are vital for achieving good results, but they depend a lot on the implementation.

4 Results

4.1 Scaling

We compared the results of scaling with respect to resolution, width and depth. The baseline network had a 95×95 resolution, width $= 12$, and depth = $[2, 2, 2, 2]$. It was trained for 40 epochs on 7000 training examples, using batch-normalization and cross-validation. The overall results are presented in Figure 5. Detailed results, with error bars representing the standard deviation of the results obtained during cross validation, are presented in Figures 6, 7, 8. 'Relative flops' means that we measure flops relatively to the baseline network : relative flops $= 2$ means the networks has twice more flops that the baseline network.

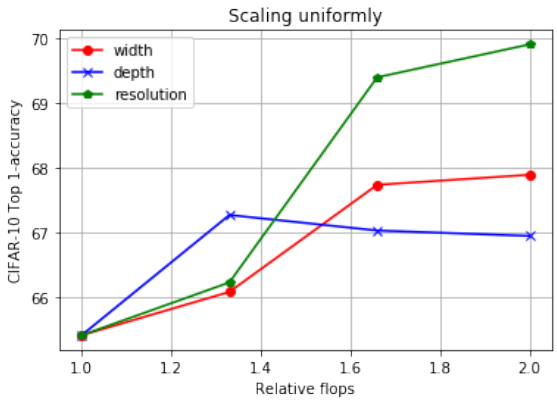

Figure 5: Overall scaling results

Overall, the results we got are reasonable. Indeed, for width and resolution, the accuracy increases with width and resolution. The results of depth scaling are a bit more surprising. Increasing depth

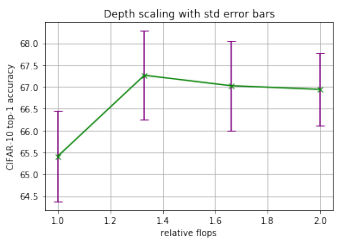

Figure 6: Depth scaling

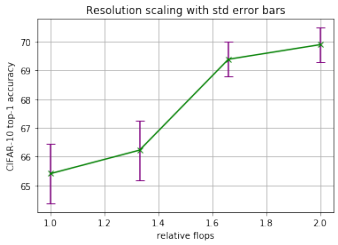

Figure 7: Resolution scaling

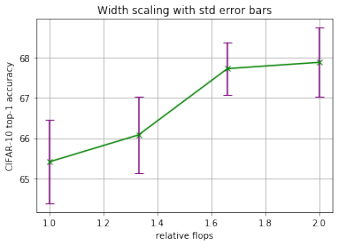

Figure 8: Width scaling

only does not seem to help a lot. We can explain this result with the fact that deeper networks can be harder to train. We did not used a learning rate decay schedule, for example.

Then, we used the compound scaling algorithm to find the best scaling coefficients α, β, γ. We got the following results :

$$\alpha = 1.17 \;\; \beta = 1.09 \;\; \gamma = 1.21$$

We scaled the baseline network according to these coefficients. Figure 10 show how the compound scaling method compares to the other ones, and Figure 9 shows detailed results of compound scaling. Figure 10 shows that compound scaling beats the other uniform methods. We also note that the difference only occurs with larger flops. This is reasonable, because the drawbacks of scaling uniformly will be more visible when the network is scaled a lot.

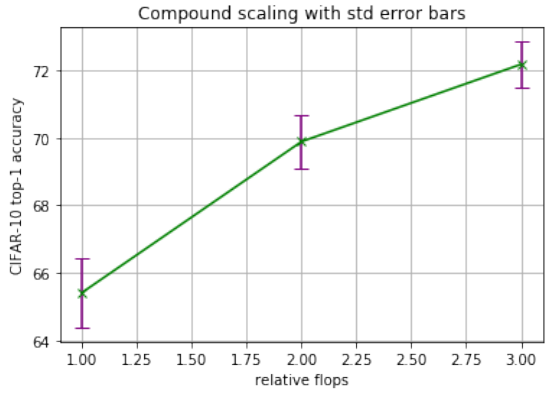

Figure 9: Results obtained with compound scaling

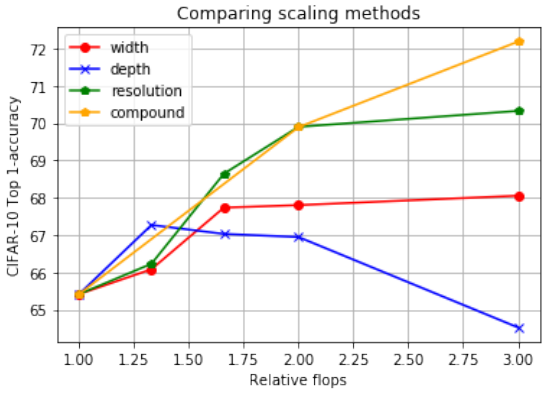

Figure 10: Comparison between all scaling methods

4.2 Detection

4.2.1 Training the backbone classifier

Our scaled ResNet classifier achieves 98.8 accuracy on the classification task. The majority of the mistakes are due to the dataset (misclassified images and poor box labeling). In figure 11 we show a few of the actual mistakes made by the classifier

4.2.2 Training the R-CNN network

As mentioned in paragraph 3.2.2, our implementation differs a lot from the one of [2]. For this reason, we could not use the same hyper-parameters for training. We had to find the right hyper-parameters for our implementation. To do so, we looked at different metrics during training. Note that these metrics do not involve bounding box regression. We focused primarily on classification. Because of the training oscillations, we used early stopping to chose the best model.

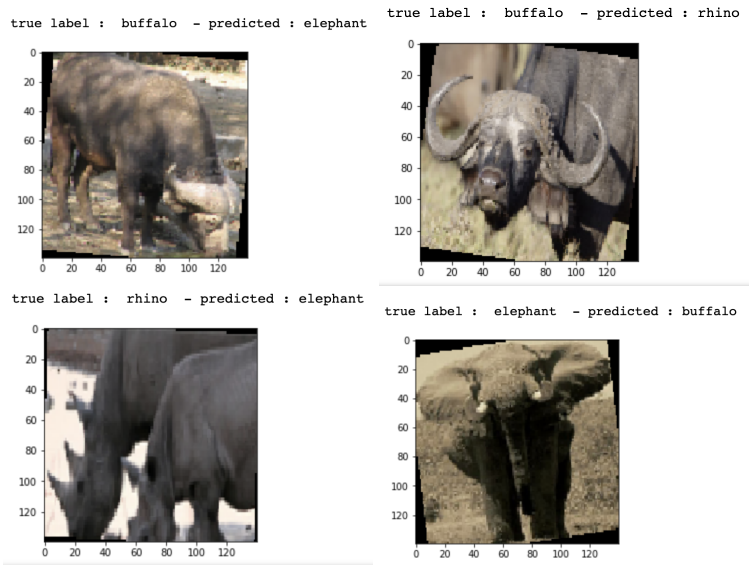

Figure 11: Some wrong classifications

- Training loss on one batch (128 regions)
- Classification accuracy : on a classification dataset, the proportion of accurate predicted labels;
- Object detection mistakes : proportion of object labelled as "background";
- Background recall : proportion of background regions labeled as "background".

The main hyper-parameters to find were the learning rate and the proportion of positive images in each batch. Finally we chose SGD optimization with a learning rate of 10^{-2}, and 79% positive images. We obtained the following results :

Classification accuracy	Object detection mistakes	Background recall
91%	<1%	82%

In table 1 we show detailed per-class performance of our trained classifier.

	Precision	Recall
Buffalos	94 %	98 %
Elephants	86 %	90 %
Rhinos	97 %	80 %
Zebras	94%	97 %

Table 1: Per-class performance of the classifier

Clearly, the training could be done better : the table shows that the performance is highly heterogeneous among the different classes.

As for the bounding box regressor, we used for training a smooth L1-Loss as described in various tutorials about R-CNN. In the article, the regressor was calculated exactly (it is a linear regression with inputs from the pre-trained network) and there was one regressor per class. We only used one regressor and trained it with SGD.

Finally we display a few examples of good detection done by our model in Figure 12.

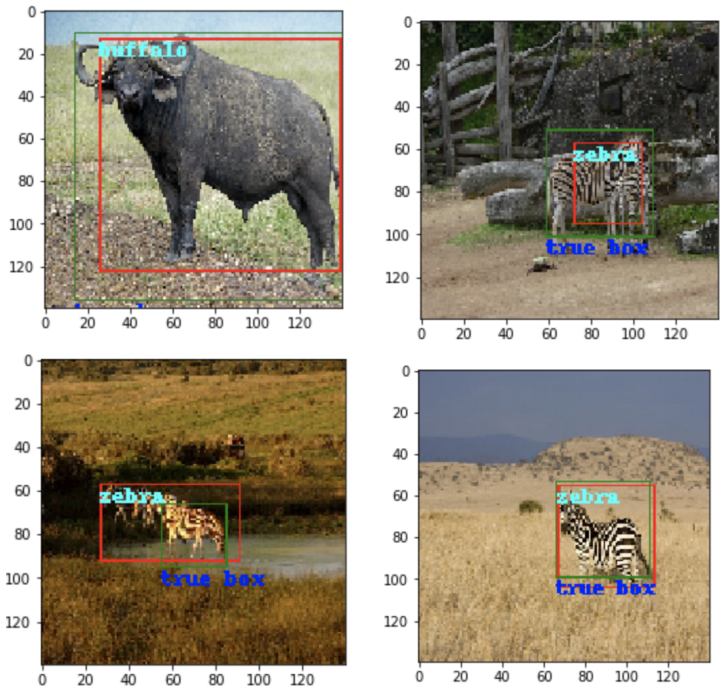

Figure 12: Some detection examples. The red box is the prediction, the green box is the ground-truth. The blue text is the predicted class.

5 Conclusion and discussion

5.1 About the scaling method

In this project, we showed that the compound scaling method described in [6] gives significantly better results that uniform scaling methods. However, we feel that this method cannot be used systematically.

- To use this method on a particular baseline network, one should be able to train the corresponding class of models (with various width, resolution and depth) with the same method. This is actually non trivial, because sometimes, some hyper-parameters should be adapted, as well as the optimization method. For example, a deeper network might require more training examples to properly converge, or a different learning rate adaptation technique.

- To find the best α, β, γ parameters, the results of the class of models should be stable enough, but they should also be fast to obtain. This makes this method hard to use if the user does not have the infrastructure (or the resources) to train large networks for a long time. Here, we could manage to get satisfying results with only one basic GPU, but we had to sacrifice a bit of accuracy, and it took a lot of trial and error to figure out the right trade-off between training time and stability.

That being said, if the user meets the aforementioned specifications, this method will certainly give great results, and is applicable in a very wide range of cases, namely with models where width, resolution and depth scaling can be identified like in paragraph 3.1.2.

The grid-search algorithm is an optimization algorithm, falling into the category of black-box optimization. This suggests the use of other, more sophisticated algorithms for finding the best scaling coefficients. However, the search space is fairly small here (for example, we only tested 20 values), so the benefits of using a more sophisticated algorithm will certainly be insignificant.

5.2 About the R-CNN method

We used the scaling method to build a classifier on the African animals dataset. For the detection task, we used R-CNN. In reality, our implementation was a bit different from the actual one. We were able to produce a satisfying detection model. However, this model could be improved in numerous ways.

- Optimization : training the classifier and the regressor using SGD is not an easy task. A better way would be to use an SVM as a classifier (which gives exactly the optimal solutions) and a linear regression model for the regressor (again, this gives the optimal solution). Using SGD for training these two components results in a lot of optimization oscillations.
- Batch sampling : In the original article [2], the proportion of positive images was chosen with trial and error. Indeed, this proportion is related to the proportion of positive images given by the region proposal algorithm. We first used the recommended proportions, and changed the proportions afterwards. A better way to find the right sampling proportions is necessary for a better training.
- Positive and negative regions : Again, the authors chose the IoU threshold for classifying regions to positive and negative ones by trial and error. We used the same one as theirs, but the best IoU threshold might be different for our implementation.

The main downside of R-CNN is the slow test time. For us, it was not really an issue, as there was only 4 classes to predict. To speed-up predictions (for example for real-time object detection), variants of this algorithm were designed : Fast R-CNN (2015) [1], Faster R-CNN (2016) [4].

References

[1] R. Girshick. Fast r-cnn, 2015.

[2] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *CoRR*, abs/1311.2524, 2013.

[3] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[4] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks, 2015.

[5] M. Tan, B. Chen, R. Pang, V. Vasudevan, and Q. V. Le. Mnasnet: Platform-aware neural architecture search for mobile. *CoRR*, abs/1807.11626, 2018.

[6] M. Tan and Q. V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *CoRR*, abs/1905.11946, 2019.