ConvMixer is all you need?

Andrea Alfarano[#1]

[1]alfarano.2026770@studenti.uniroma1.it

***Abstract*— Transformers have gained increasing popularity outside NLP environments, overcoming traditional CNN in several tasks. We are going to investigate ConvMixer, a simple CNN which uses similar patch based embedding in order to obtain superior performance with fewer parameters and computation resources. We will evaluate the possibility of use ConvMixer layer in custom simple architectures for object detection and object segmentation, introducing two novel architectures: ConvMixer-Unet and YOLO-ConvMixer. In order to evaluate superior ConvMixer performance we evaluate the novel ConvMixer-Unet against the classical Unet**

***Keywords*— ConvMixer image segmentation object detection.**

I. Introduction

CNNs aren't cool anymore. CNNs have become the dominant architecture for vision tasks for plenty of years. Although recently new Transformers based architectures, called Vision Transformers (ViTs), have proved to be competitive (often better) compared to state-of-the-art convolutional networks while requiring substantially fewer computational resources to train[1].

Applying transformers to images requires some kind of embedding: it's not possible to apply transformers directly to a pixel's image, due to quadratic cost of self attention layers which will lead to dramatic computational complexity. Several attempts have been made [2] to apply transformers to computer vision, until the introduction of embedding by dividing images in pixel patches.

Vision Transformers (ViT) have gained popularity becoming the standard the facto in computer vision.

New kid in town. Recently a novel proposed architecture, ConvMixer, tries to replicate transformer behavior using only Cnn. ConvMixer's key idea is that superior Transformers capabilities mainly depend on patch based embedding approaches.

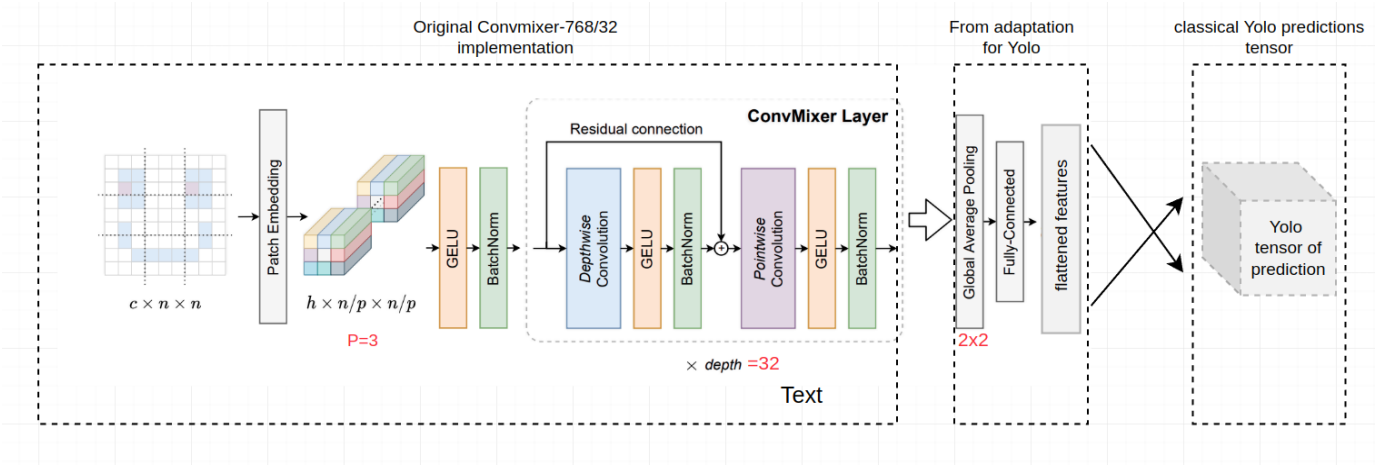
In the original paper it is shown how this approach allows to overcome classical object classification CNNs architectures, being competitive also with ViT dramatically less computational resources. ConvMixer is also so simple that it is possible to implement it in just 6 rows of compact code.

In order to extend ConvMixer outside original object classification tasks, I developed ConvMixer-Unet and YOLO-ConvMixer. Proposed models were developed as simply as possible in order to show ConvMixer capabilities without external factors.

Disclaimer. Both Yolo-ConvMixer and ConvMixer-Unet are in their first design iterations. Due to my few computational capabilities I couldn't try other iterations and refine those models with the gained experience.

II. Yolo-ConvMixer introduction

Why Yolo-ConvMixer? In Yolo-ConvMixer I substituted the original Yolov1 Cnn backbone (usually VGG or Resnet) with ConvMixer-768/32 pretrained on Imaginet1k by original paper authors. I have chosen Yolov1 because it is a classical object detection model, with good capabilities in real time object detection. Recently some authors have implemented Yolo style models using transformers as backbone. I guess that the high throughput of ConvMixer and the low weight of the models may allow faster not Cnn Yolo implementation, specially on the edge, where size and weight are crucial factors. In fact our proposed architecture is



surprising lightweight compared to classics object detections models (table 2, modified version of [2])

Yolo head: I changed 1x1 Global Average Pooling to 9x9, and added two linear layers and leakyRelu activation function between the two linear layers. I made some adaptation to final layers to deal with

Table 4
Performance comparison of various object detectors on MS COCO and PASCAL VOC 2012 datasets at similar input image size. Rows colored gray are real-time detectors (>30 FPS).

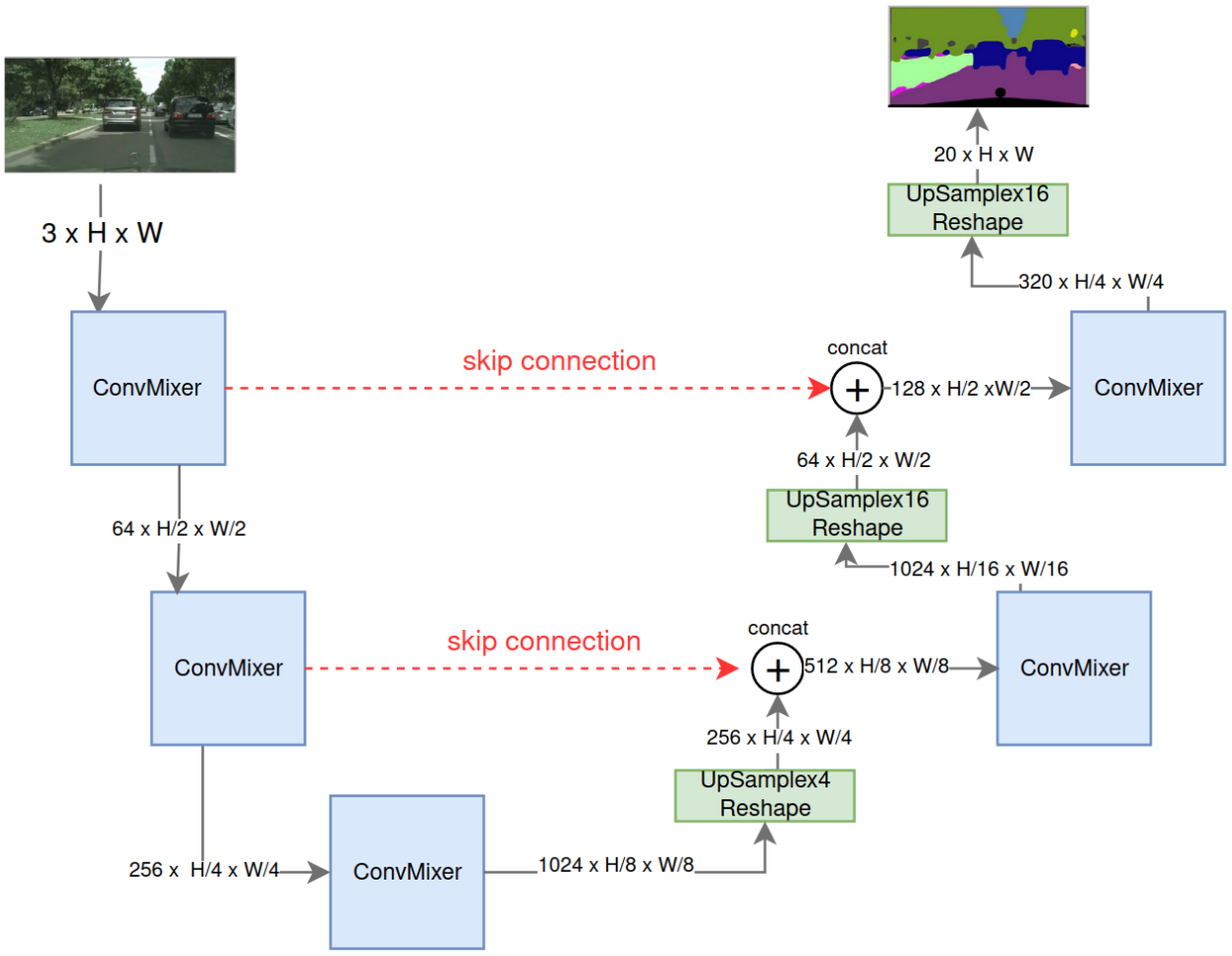
Model	Year	Backbone	Size	$AP_{[0.5:0.95]}$	$AP_{0.5}$	FPS
R-CNN*	2014	AlexNet	224	-	58.50%	~0.02
SPP-Net*	2015	ZF-5	Variable	-	59.20%	~0.23
Fast R-CNN*	2015	VGG-16	Variable	-	65.70%	~0.43
Faster R-CNN*	2016	VGG-16	600	-	67.00%	5
R-FCN	2016	ResNet-101	600	31.50%	53.20%	~3
FPN	2017	ResNet-101	800	36.20%	59.10%	5
Mask R-CNN	2018	ResNeXt-101-FPN	800	39.80%	62.30%	5
DetectoRS	2020	ResNeXt-101	1333	53.30%	71.60%	~4
YOLO*	2015	(Modified) GoogLeNet	448	-	57.90%	45
SSD	2016	VGG-16	300	23.20%	41.20%	46
YOLOv2	2016	DarkNet-19	352	21.60%	44.00%	81
RetinaNet	2018	ResNet-101-FPN	400	31.90%	49.50%	12
YOLOv3	2018	DarkNet-53	320	28.20%	51.50%	45
CenterNet	2019	Hourglass-104	512	42.10%	61.10%	7.8
EfficientDet-D2	2020	Efficient-B2	768	43.00%	62.30%	41.7
YOLOv4	2020	CSPDarkNet-53	512	43.00%	64.90%	31
DeTR	2020	ResNet-101	-	43.50%	63.80%	20
Swin-L	2021	HTC++	-	57.70%	-	-
YOLO-ConvMixer	2022	ConvMixer	49.7			

Yolov1 head is linked to the last linear layer in order to make final detection.

III. ConvMixer-Unet introduction

Why ConvMixer-Unet? Unet was a very popular segmentation architecture, so I decided to extend ConvMixer to object segmentation following the original Unet idea of building an encoder-decoder model, with skip connection for dealing with dimensionality reduction. I tried to keep the model as simple as possible using only ConvMixers and tensor operation.

Model description. I used 2x2 patches, kernel size= 9 and deep=8 for all the blocks. I guess that a deeper model may achieve better results. ConvMixer block is great for encoding: itself embed input space, downsampling original HxW resolution into smaller H/p x W/p. We can apply



ConvMixers blocks directly in cascades performing deconvolutions. I increase hidden input at each stage at each encoding stage to compensate for resolution loss.

Decoding with ConvMixer is harder. ConvMixer patch embedding implies at each decoding stage some loss of resolution. So we need to perform a double increase of resolution: the first for compensating the embedding nature of ConvMixer and the second for achieving the decoder necessity of the nets. For this reason I designed the UpSample-reshape block. UpSample-reshape block takes as input a tensor CxHxW and gives back a n-higher resolution tensor with shape C/n^2 x Hn x Wn. So it basically reduces channels dimensions, concat together features for increasing resolution dimension. It is implemented using the einops library, for better readability.

IV. Yolo-ConvMixer setup and experiments

Dataset choice. Most popular object detection datasets are MS COCO, imaginet 1k and Pascal VOC. Pascal VOC has fewer classes than the other two, so due to lack of computational resources I decided to train and evaluate Yolo-ConvMixer on Pascal VOC 2012. This dataset is composed of 5,717 images, with 20 object classes. Every class belongs to 4 main clusters: people, animals, vehicles, and indoor.

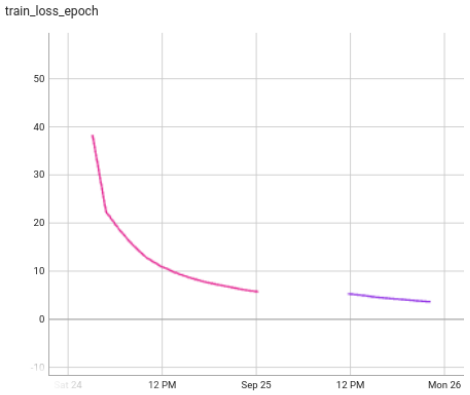
Dataset transformations. Every image is reshaped into a 448x448 pixel image and normalized. I didn't apply any augmentation technique which may increase performances.

Epochs and fine tuning. I trained the model for 20 epochs. I chose Convmixer-768-32 (768 is the embedding features size, 32 is the model depth) because it is the most convenient as performance/cost ratio. Infact Convmixer-768-32 achieved 80.16% on top 1 Immagine 1k, with only 21.1M parameters. Convmixer-1536/20 archive a slight 2% of increment for 30M more parameters.

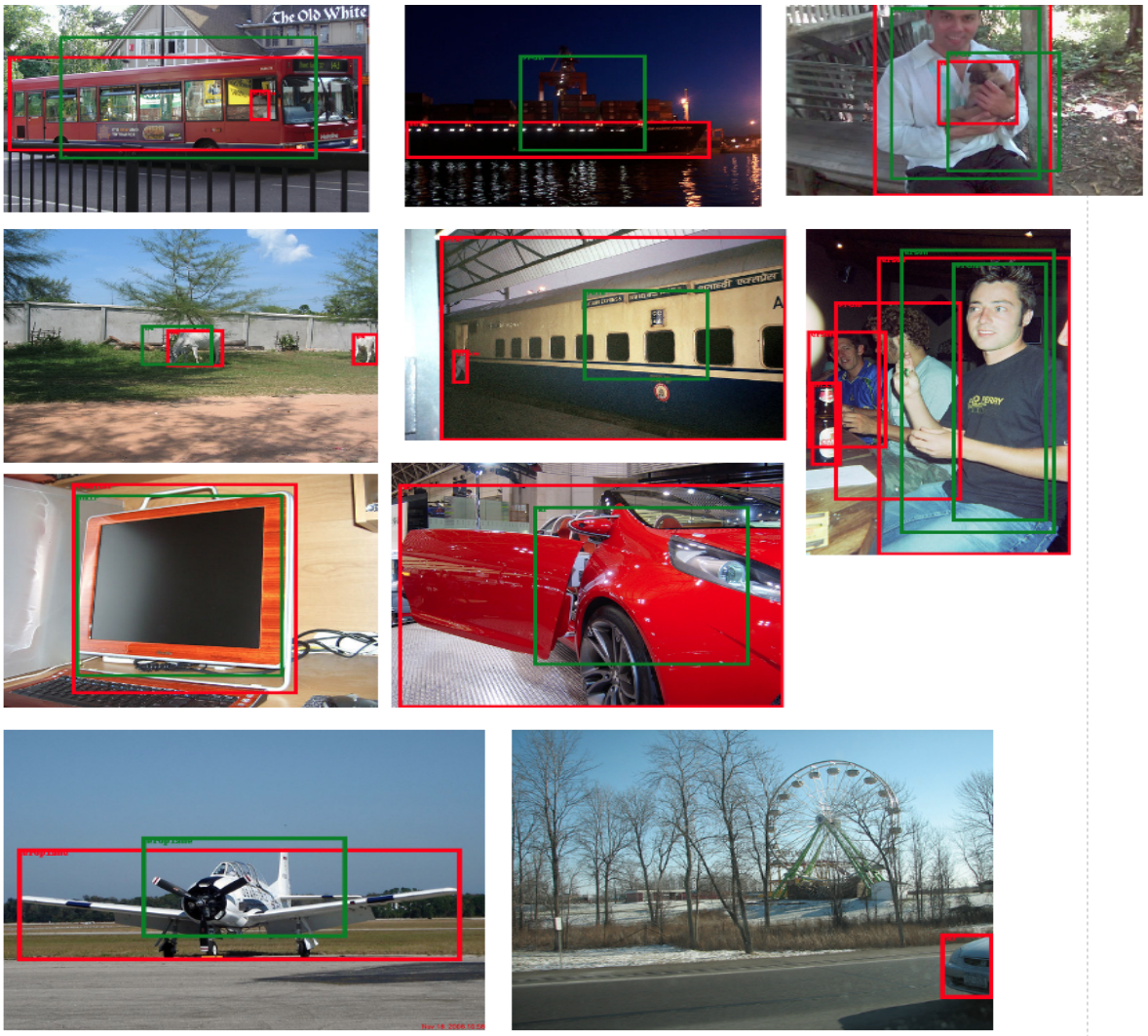
From Convmixer-768-32 pre-trained by original paper authors on imaginet-1k, I removed the last 3 layers used for object classification and I added two linear layers with leakyRelu in between. I also tried to use only one linear layer directly but I achieved poor results.

Lr and optimizer. Original ConvMixer paper uses AdamW with a triangular learning rate scheduler. I find a lot of divergence problems[table n] with AdamW, also with very low (1e-6) learning rate. I finally found good convergence with classical Adam with B1=0.9 and B2=0.999. I trained the model for only 30 epochs and I actively monitored training status, so I decided to not use any learning rate scheduler.

Computing configuration and batch size.. I used 3 AMD R9 nano on a 4 core cpu. Computation was parallelized using GLOO as backend. All the GPUS have only 4 GB of RAM. So I needed to use batch size = 3. Training process Took 33 hours for this configuration.



Testing and consideration. When I reached the training plateau I stopped model learning and I tested it on a testing test. Surprisingly the model has dramatically overfitted data, although it was trained for only 20 epochs.



Labeled data form Yolo-ConvMixer

(green label are prediction from the model, red are labels)

V. ConvMixer-Unet setup and experiments

Dataset choice. There are plenty of datasets for data segmentation purposes. I decided to use cityscape, a popular urban dataset with 5000 annotated frames and 20000 weakly annotated frames. I thought that higher throughput of Convmixer could be useful in tasks where there is demand for fast response, and a dataset affine to autonomous driving looks like the right choice. Scenes come from 50 different cities, and there are 20 kinds of labeled objects.

Dataset transformations. Every image is reshaped into a 256x512 pixels image and normalized. I apply just random horizontal flipping.

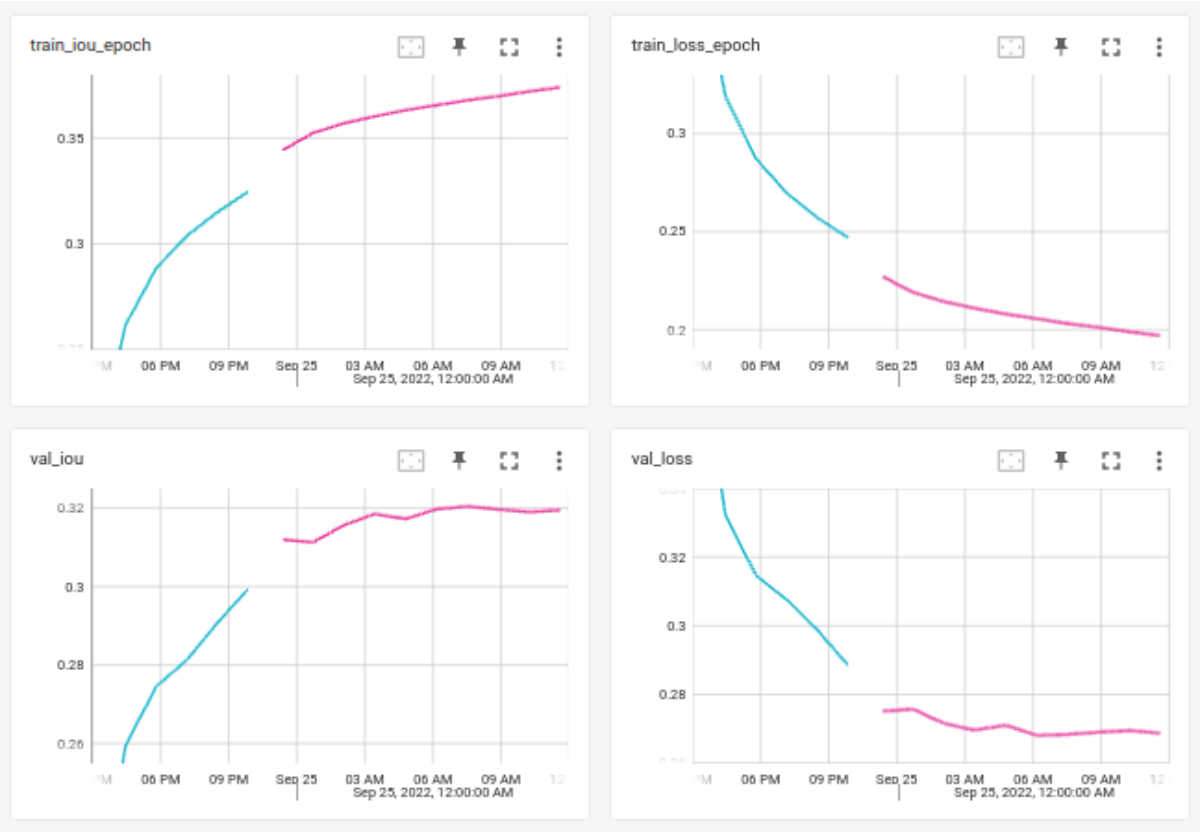
Epochs and fine tuning. I trained the model for 10 epochs. After the 5th i had an issue (maybe a driver issue or a power loss) and the training was restarted from a backup. For this reason plots are splitted in two. I cannot use pre trained weights for ConMixer blocks on my net.

Lr and optimizer. Original ConvMixer paper uses AdamW. I decided to use the same optimizer to obtain good convergence. I used lr= 1e-3 for the first 5 epochs and 2e-4 for the last 5.

Loss. I used Dice loss for training loss and Intersection over Union (IoU) as evaluation loss. Both the choices are quite standard for image segmentation.

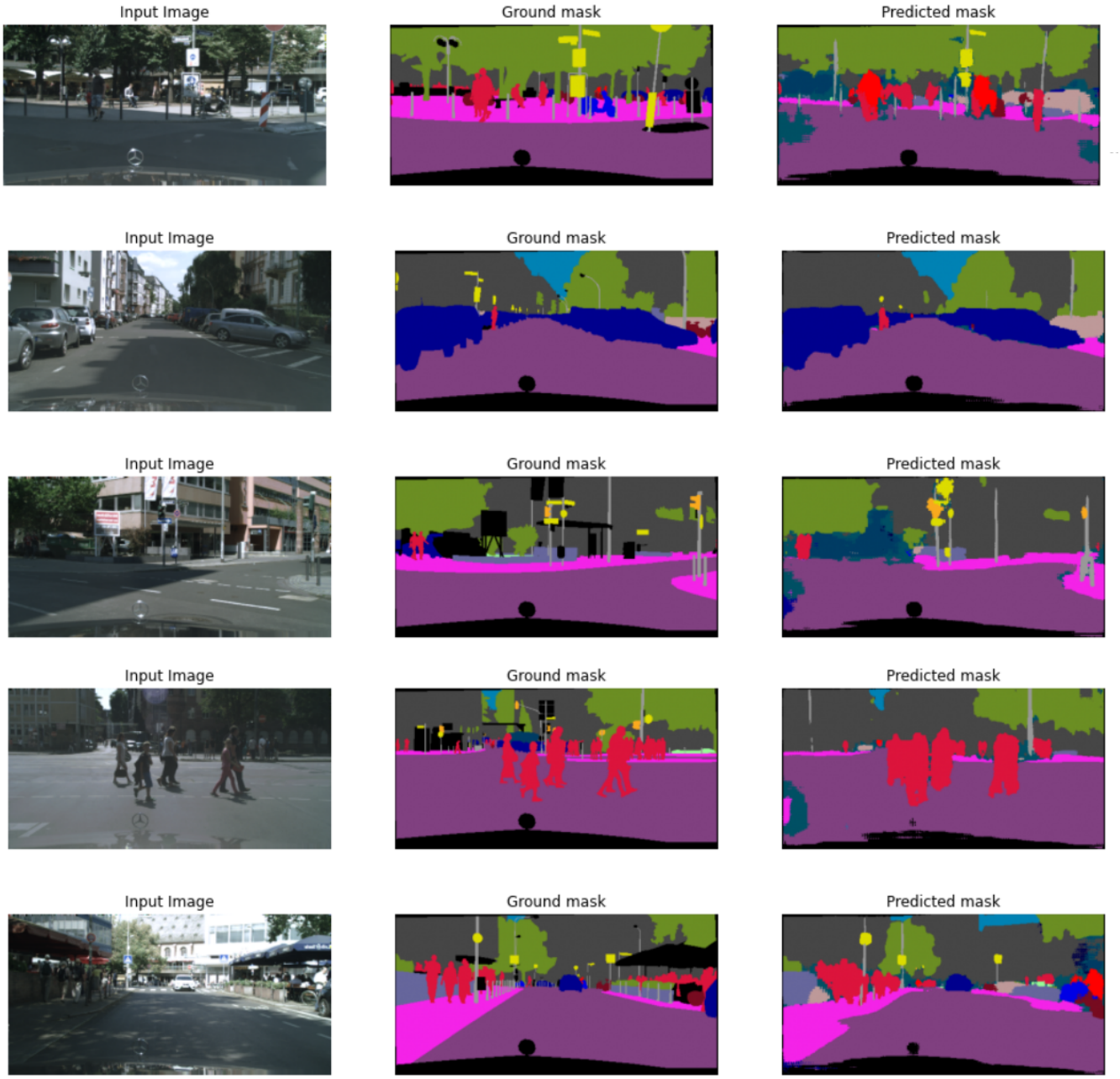
Computing configuration and batch size.. I used 2 AMD R9 nano on a 4 core cpu. Computation was parallelized using GLOO as backend. Both GPUS have only 4 GB of RAM. So i needed to use batch size = 2

Training summary and consideration. Blu lines represent the first 5 epochs during training and evaluation. Red line the other 5. The decrease on the slope between the two color curves are due the reduction of lr. Considering the limited deep of the model, we can achieve a good convergence on few epochs. Use a deeper Convmixer blocks on the network, ad adopt augmentation techniques may dramatically increase model capabilities. We can notice a plateau on val_loss after the reboot of the model. I guess that 1e-3 could be a better learning



rate and the adoption of a learning rate scheduler could lead to better training.

Unfortunately I couldn't use the original dataset evaluation metric, due to script incompatibility with my machine and lack of documentation. Anyway, I reported on this paper the first 10 images of the test dataset, elaborated by the model. We can appreciate good comprehension of the scene, especially keeping in mind that the model was trained only for 10 epochs, without fine tuning and or data augmentation tecniche.




Input Image
Ground mask
Predicted mask
Input Image
Ground mask
Predicted mask
Input Image
Ground mask
Predicted mask
Input Image
Ground mask
Predicted mask
Input Image
Ground mask
Predicted mask
Input Image
Ground mask
Predicted mask

VI. CLASSICAL UNET VS NOVEL CONVMIXERUNET: A FAIR COMPARISON

I decided to test the proposed ConvMixerUnet in relation to the classical Unet, in order to measure if the ConvMixer block can improve a popular segmentation model.
Unet is the most popular image segmentation algorithm, initially designed for medical image segmentation, it has found a large range of applications in general computer vision, as well general image segmentation. I used the original Unet implementation, imported using torch.hub, in order to avoid possible mistakes in my implementation. In order to obtain a fair comparison i needed to test both the architecture under the same conditions:

1. augmentation equality: both the architectures are trained using the same augmentation tecquinches.
2. no pre training: both the architecture are trained without using pretrained weight, to avoid externals factors
3. same loss function: both the architecture are trained using dice loss (original Unet loss)
4. same optimizer: both the architectures use adamW with same parameters
5. same numbers of epochs: both nets were trained using 20 epochs
6. same batch size: i used 24 batch size during the training of both architectures

Results:

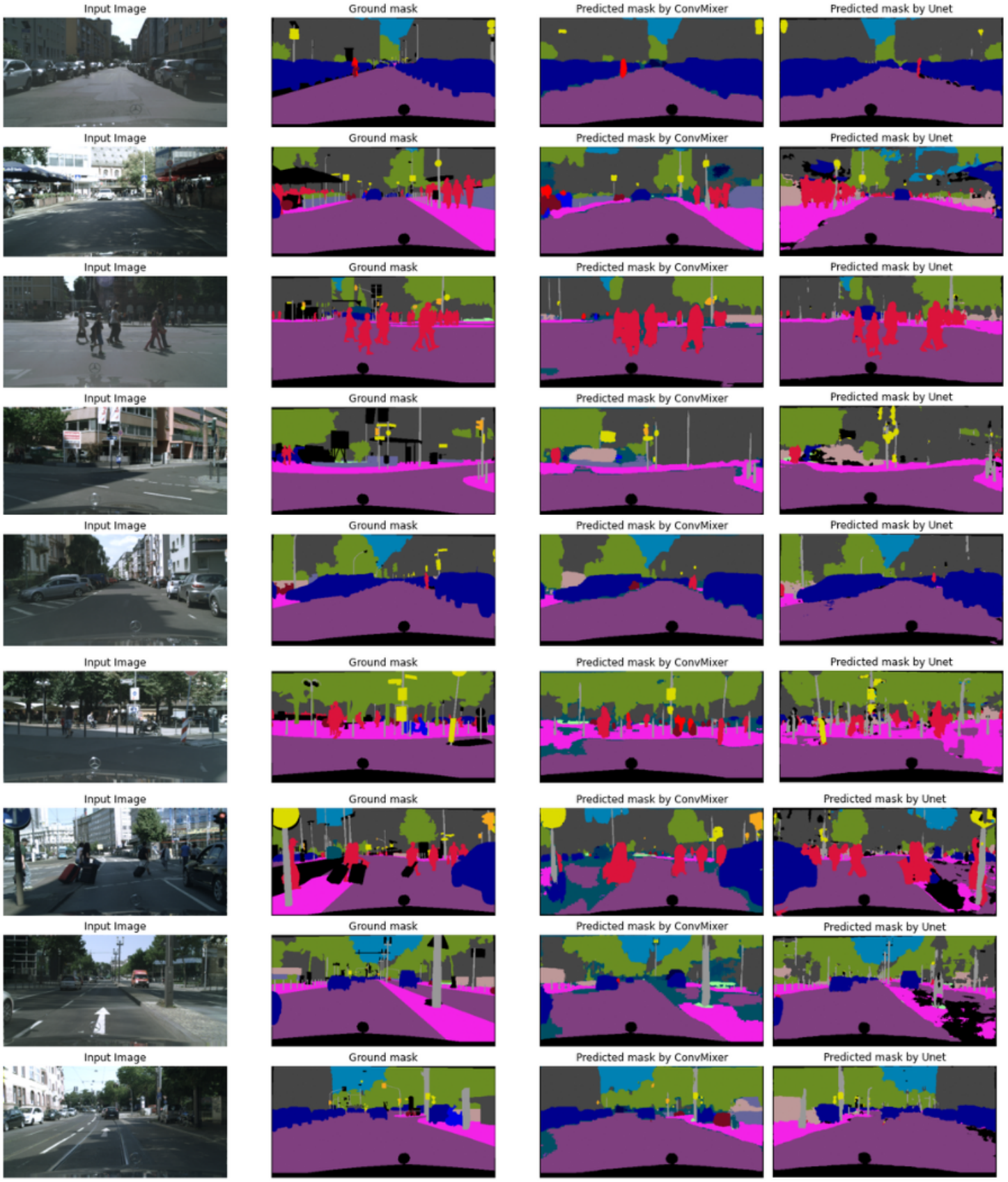
ConvMixer-Unet confirm its low weight nature: ConvMixer-Unet uses just 23.4M parameters, 50% less then Unet's 31M parameters.

ConvMixer-Unet is slower to train: on the same computational conditions it uses an average of 58 minutes to conclude each epoch. Unet uses just 23 minutes for epochs, resulting significantly faster during training

ConvMixer-Unet is has higher inference time respect Unet: using a batch of 500 pictures, ConvMixer-Unet uses 114,5 seconds to process all pictures, while Unet uses just 27,7 seconds

ConvMixer-Unet performs better than Unet in both Dice loss and IOU (intersection over unions) on the evaluation set

	ConvMixer_unet	Unet
#params	23.4M	31M
minute/ epoch	58 minute	23 minute
inference	4,3 picture for second	18 picture for second
Dice loss	0.2628	0.5976
IOU	0.3288	0.3035


Input Image
Ground mask
Predicted mask by ConvMixer
Predicted mask by Unet
Input Image
Ground mask
Predicted mask by ConvMixer
Predicted mask by Unet
Input Image
Ground mask
Predicted mask by ConvMixer
Predicted mask by Unet
Input Image
Ground mask
Predicted mask by ConvMixer
Predicted mask by Unet
Input Image
Ground mask
Predicted mask by ConvMixer
Predicted mask by Unet
Input Image
Ground mask
Predicted mask by ConvMixer
Predicted mask by Unet
Input Image
Ground mask
Predicted mask by ConvMixer
Predicted mask by Unet
Input Image
Ground mask
Predicted mask by ConvMixer
Predicted mask by Unet
Input Image
Ground mask
Predicted mask by ConvMixer
Predicted mask by Unet