This technical exercise asks us to create a binary classifier for a computer vision task. The task consists of distinguishing roads from fields.

1 Model

1.1 Problem analysis and possible solutions

Classification is a common task of computer vision. Historically, the first main attempts to tackle image classification appeared with "plain" neural networks. However, it is the apparition of Convolutional Neural Networks (CNN) that really allowed improvement in this task.

Since the apparition of LeNet-5 presented in 1998, which was able to identify written numbers, many other innovative CNNs were proposed, such as AlexNet, VGGs, ResNets, Inception, etc... The apparition of Transformers, mainly used in Natural Language Processing in the beginning, allowed the apparition of Vision Transformers (ViT). Those ViT, combined or not with CNN, reached new state-of-the-art results in image classification tasks.

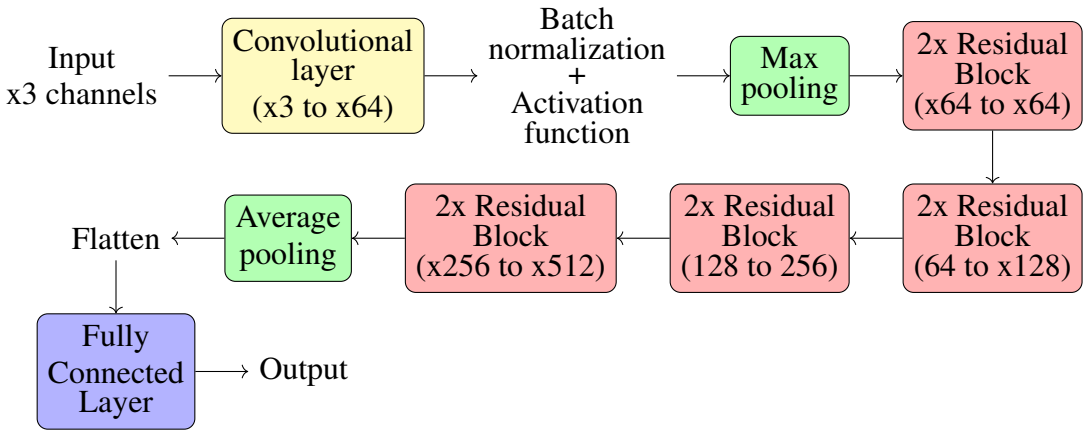
To create a binary road/field classifier using mainly Pytorch library and no pre-trained model, I choose an easy-to-implement ResNet architecture.

ResNets are CNNs that appeared in 2015 and proposed the novelty of skip connections. A skip connection consists of layer output "skipping" some layers to be fed to a deeper layer as input. This limited a "degradation" problem observed between shallow and deeper models, the firsts sometimes performing better than the latter.

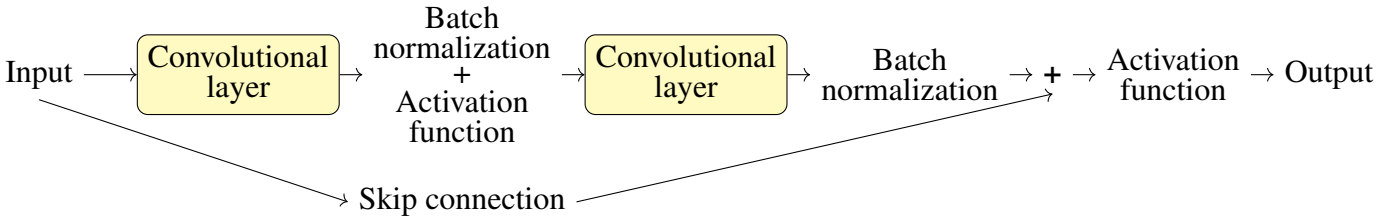
In the initial paper presenting ResNets, architectures with different depths (number of layers) were proposed. As my personal computer lacks GPU computational capability and to have the ability to train multiple models with different hyperparameters, I choose to implement an 18-layer ResNet model.

1.2 Architecture implemented

Here's the ResNet 18 architecture used :



With the skip connection between 2 Residual Block as follows:



If ReLU is commonly used in ResNet implementation, I replaced it by a LeakyReLU with a negative slope of 0.2. LeakyReLU could reduce potential vanishing gradient problems.

The model is implemented in *cnn_classifier.py* file.

1.3 Loss, optimizer, and scheduler

For a binary classification problem, the "logical" loss to use is binary cross-entropy loss. However, I created a modular classifier that could output classification results for a number of classes superior to 2 (though it is not used for this specific road/field classifier). I then use a (non-binary) cross-entropy loss on my model outputs.

More precisely, I use the *CrossEntropyLoss* module of Pytorch, which performs a log softmax function on the logits output by the model before using cross-entropy loss.

To optimize gradient descent, I use an Adam optimizer. The learning rate is reduced when the loss on a validation set stops improving using *ReduceLROnPlateau*.

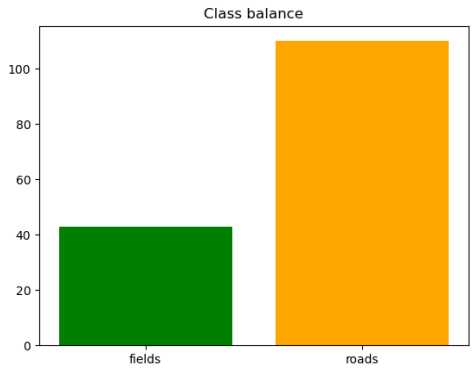
2 Data

2.1 Data analysis

Before training our model, a quick analysis of the training set is mandatory. The script *data_analysis.py* shows random examples of training and testing sets, the size of both sets, and the proportion of each class in the training set.

Thanks to the examples displayed, two mislabeled examples were identified in our training set. Two pictures labeled as "field" are actually road pictures (*3.jpg* and *5.jpg*). Because we have only 153 images in the training set, those mislabelled examples could greatly harm our classifier performances. Their labels were corrected.

Our dataset contains images of various sizes. Resizing would be necessary.



Finally, our training set has a severe class imbalance, with at least twice as many road pictures as field pictures. This could harm our classifier performances on the only 10 images of the testing set. To reduce the impact of this class imbalance, I choose to use *WeightedRandomSampler* object of Pytorch (see *ccn_trainer.py* file). When creating batches for training, it will oversample the field images.

2.2 Data pre-processing

For the testing set, examples are classified to evaluate performance later on. Using *torchvision transforms*, a resizing and normalization are performed. The first is necessary for the model to process images of different sizes. After testing different image sizes, a size 224 for height and width was chosen as it conserves enough image quality for correct classification while keeping computation time reasonable. The second can help CNN's performance achieve better classification results.

The original training data were split using a 90%/10% cut to obtain training and validation sets. The same resizing and normalization as the training set are performed on the validation set.

The remaining testing set is duplicated. This duplication is made because we have little data to train our model on. This is not a perfect option because one image may be seen twice (or more if it's a minority class image with *WeightedRandomSampler*), but it can help using only our available data. The same transformations as testing and validation sets are performed on one version of the training set. On the second, data augmentation is performed. This data augmentation aims to augment the size of our initial data (while avoiding overfitting) and create less ideal photos than what's initially available. I choose a combination of multiple "Random" transformations in *torchvision transforms*.

3 Training and Evaluation

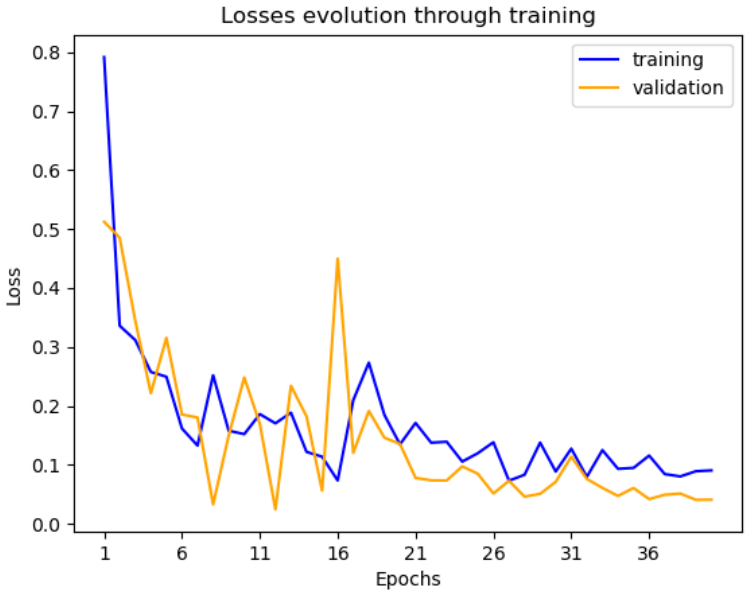
3.1 Losses evolution

Losses values on training and validation sets were saved at each epoch and displayed after training.

"Playing" on learning rate for Adam optimizer, I choose a starting value of $3e^{-4}$ as lesser or bigger values could take longer to converge (I tested values between $1e^{-5}$ - $1e^{-3}$ range).

Without an early system method implemented and "playing" with the total number of epochs performed, I stopped the model training at 40 epochs, where loss validation seems to stop improving (I tested values between 10 - 200 epochs range).

Using a training mini-batch size of 32, here are the curves of losses evolution through training epochs:



3.2 Results

My experimentations stopped when I reached perfect accuracy on the validation set (16 images in total). I could have installed an early stop on this metric to stop training when the best accuracy possible (lower or perfect) was reached on this validation set.

Coincidentally, the obtained model also reached a perfect accuracy on the testing set.

More details are presented in the following table:

Set	**#images**	**Accuracy**	**Recall**	**Precision**
Train(augmented)	274	97%	96%	98%
Val	16	100%	100%	100%
Test	10	100%	100%	100%

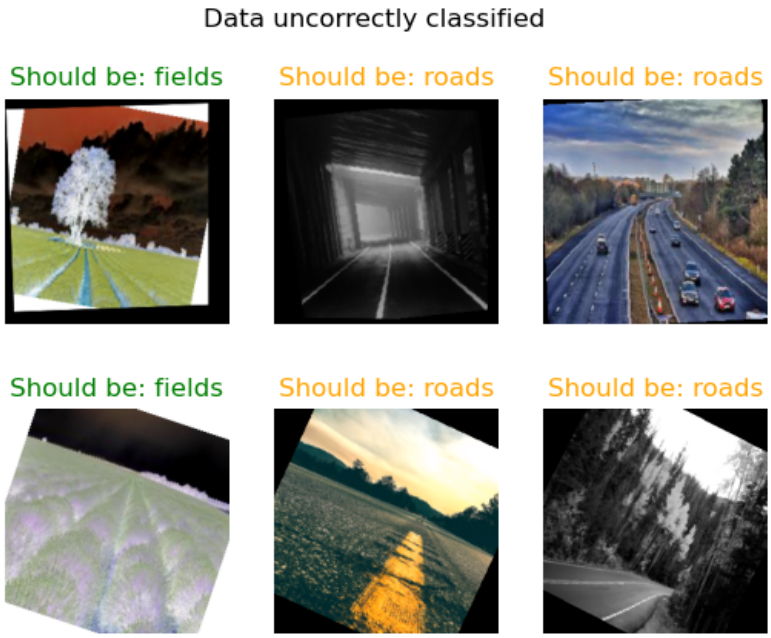
Table 1: Best model's performances on different datasets. Recall and precision are macro-average.

4 Additional: Visualization

4.1 Errors visualization on training set

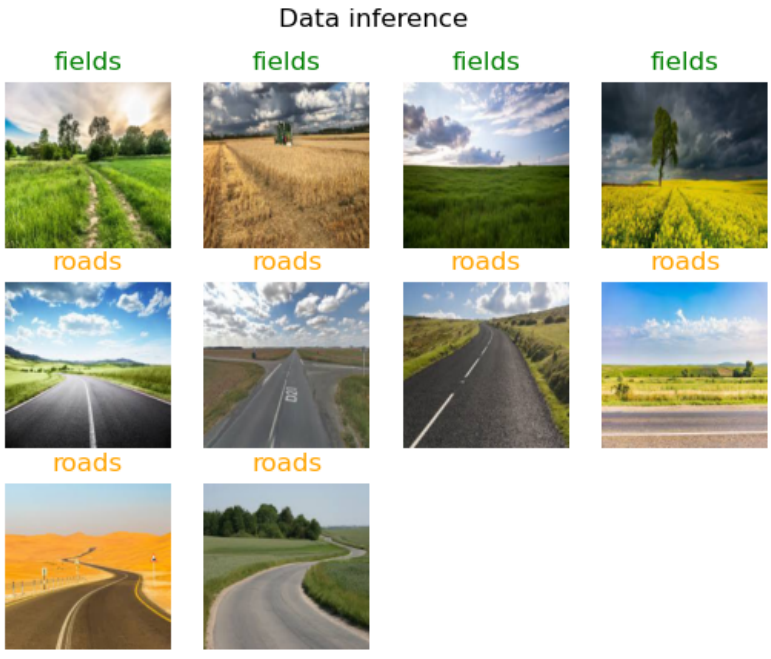
For a more in-depth analysis of our model results, here are the mislabeled images on our training set:

All those images are augmented images. However, despite the augmentation, it is still easy to classify them for a human evaluator. Though we obtained perfect accuracy on our validation and testing sets, improvement is still possible.



4.2 Inference on testing set

For a more in-depth analysis of our model results, here are the inference results on our testing set:



5 Short note

A shallower, more "conventional" CNN could have probably obtained similar results with a good set of hyperparameters. Similarly, a pretrained existing model - adapted or fine-tuned - could have also obtained excellent results, probably with less data processing.