Computer vision for landscape recognition

LEVY Daniel
CentraleSupelec student
Paris-Saclay
daniel.levy@student-cs.fr

PUJOL Corentin
CentraleSupelec student
Paris-Saclay
https://github.com/corentin-pujol/Landscape_recognition

Abstract

Safran Electronics & Defense is developing a multi-target detection, classification, tracking and localization video chain for land, sea and airborne defence applications to assist operators in their missions.

This video chain can be deployed in the context of directional sights, handheld cameras, panoramic surveillance sights, or distributed wide field sensors.

Safran Electronics & Defense wishes to optimize the valorization of its image & video database, which is heterogeneous, both in terms of content (type of campaign air, land, sea, type of sensor, associated metadata of navigation, image content, etc etc) and in terms of form (file format).

As part of this data enhancement, our team has implemented a context classification module. Thus, we studied different pre-trained classification models, which we re-trained for landscape and natural context recognition. This classification module is able to identify one of the following five landscapes in an image: Forest, Mountain, Ice, Coastal or Desert.

1. Introduction

As part of our annual INFONUM project, we are partnering with the company SAFRAN. Thus, within the framework of this project, we have to set up an architecture for extracting metadata each time a new file is added to the storage server. This architecture includes different bricks (extraction module, database, GUI, etc.) and notably AI bricks. Our partner gave us the freedom to choose what we would like to implement, and we chose to work on a Deep Learning module allowing us to classify the contexts to which the imported images could belong (aerial, rural, maritime, desert, mountainous context). Having seen that it was possible to couple the project of the Deep Learning module with the project of another course requiring a Deep Learning part, we thus made the choice to link these two projects. The team is composed of two students: LEVY Daniel and PUJOL Corentin. As I stated above, we want to implement and compare different Deep Learning approaches to extract the context of different images. The objective would be to implement different models (ResNet, MobileNet, VGG for example) and to study their hyperparameters in order to obtain the best possible results on context extraction. The data used will be those proposed by the partner. As the partner has not yet provided us with the data, we have found several sources allowing us to start working with a dataset obtained from the Kaggle platform. The approach we would like to take would be to implement different pre-trained models, then to perform Transfer Learning (with and without fine-tuning) in order to make these models perform well on our data.

2. State of the art

[1]https://arxiv.org/ftp/arxiv/papers/1804/1804.03928.pdf

Summary: This paper presents a multi-label image classification approach using a combination of a pre-trained convolutional neural network image representation and a multi-layer neural network structure. The proposed approach is applied to two multi-label image classification datasets, one for food classification and the other for bird classification. The results show that the proposed approach achieves high multi-label classification performance for both datasets. This approach has the potential to be applied to multi-label image classification problems in other domains.

[2]https://www.cse.ust.hk/~qyang/Docs/2009/tkde_transfer_learning.pdf

Summary: This paper presents a review of the literature on transfer learning, a machine learning technique that uses knowledge gained on one task to improve performance on another similar task. The authors present different approaches to transfer learning, including the use of pre-trained neural networks, domain adaptation techniques, transfer methods based on principal component analysis, and multi-task transfer strategies. The advantages and limitations of each approach are discussed, as well as the application areas of transfer learning. The authors conclude by outlining the current challenges in transfer of learning research and the prospects for the future.

[3]https://towardsdatascience.com/image-classification-transfer-learning-and-fine-tuning-using-tensorflow-a791baf9dbf3

Summary: The article discusses image classification using transfer learning and fine-tuning in TensorFlow. Transfer learning involves using a pre-trained model on a large dataset as a starting point for a new task. Fine-tuning involves training additional layers on top of the pre-trained model to further adapt it to the new task. The article discusses the benefits of transfer learning and fine-tuning, including faster convergence, better performance, and reduced training data requirements. It also describes the steps involved in implementing transfer learning and fine-tuning in TensorFlow, including loading the pre-trained model, modifying the top layers, and training the new model. The article concludes with a discussion of some of the challenges and limitations of transfer learning and fine-tuning.

[4]https://towardsdatascience.com/transfer-learning-with-convolutional-neural-networks-in-pytorch-dd09190245ce

Summary : This article discusses transfer learning with convolutional neural networks (CNNs) in PyTorch. It starts by introducing transfer learning and explaining why it is useful in deep learning. Then, it goes on to explain how to use transfer learning with CNNs using PyTorch. The article covers how to load pre-trained models, replace the classifier layer, and fine-tune the model on a new dataset. It also discusses some of the commonly used pre-trained models in PyTorch, such as VGG and ResNet. Finally, the article provides some tips and tricks for successful transfer learning with CNNs in PyTorch.

[5]https://medium.com/mlearning-ai/transfer-learning-and-convolutional-neural-networks-cnn-e68db4c48cca

Summary: The article "Transfer Learning and Convolutional Neural Networks (CNN)" discusses the concept of transfer learning and its application to convolutional neural networks (CNNs). The article starts by explaining the limitations of training a CNN from scratch and highlights the benefits of using transfer learning to reduce training time and improve accuracy. It then goes on to discuss different types of transfer learning, such as feature extraction and fine-tuning, and provides examples of popular pre-trained models, including VGG16 and ResNet. The article also includes code examples using PyTorch to demonstrate how to implement transfer learning for a classification task. Finally, the article concludes by discussing some of the limitations of transfer learning and the importance of selecting an appropriate pre-trained model for a specific task.

3. Dataset presentation

This is Landscape classification dataset. This data consists of 5 different classes. Each class representing a kind of landscape. These classes are:

- Coast: This class contains images belonging to coastal areas, or simply beaches.
- Desert: This class contains images of desert areas such as Sahara Thar, etc.
- Forest: This class is filled with images belonging to forest areas such as Amazon.
- Glacier: This class consists of some amazing white images; these images belong to glaciers. For example, the Antarctic.
- Mountains: This class shows you the world from the top i.e., the mountain areas such as the Himalayas.

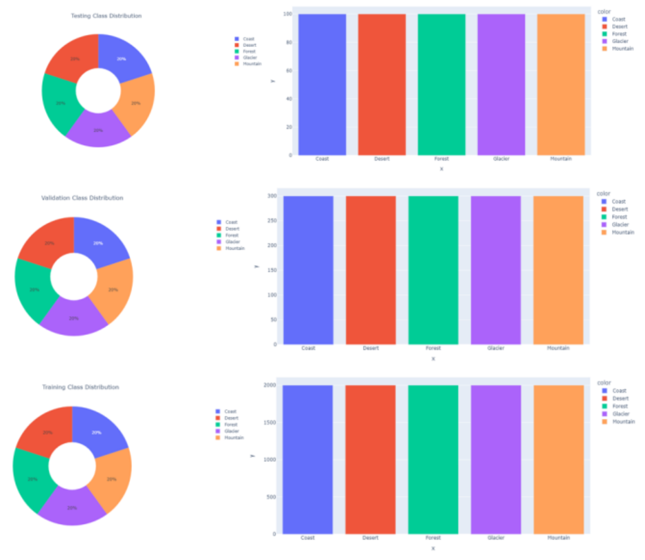
This data is first divided into 3 sub directories. These sub directories are the training, validation, and testing data directories.

Here is some images example from the dataset that we use, with the link below:

Link to get the dataset:
https://www.kaggle.com/datasets/utkarshsaxenadn/landscape-recognition-image-dataset-12k-images

<u>Class and distribution study</u>

The training, validation and testing data, all have equal class distribution. This make sure that the model will be trained, validated and tested. We can see this conclusion on the different graph and chart below:



4. Our approach to the problem

To succeed in determining the best performing model, we will load the infrastructures of four well-known models, already performing well on more global data sets. The objective is to re-train these models on the data of our problem.

We will divide the training phase into several steps in order to study each hyperparameter, understand how they work and finally choose the ones that will make the models perform best with respect to our initial problem.

After loading the pre-trained architectures, to avoid any fine-tuning of the first layers of the network, the gradients of the corresponding parameters must be frozen. Then, to perform the learning transfer, we will replace the last classification layer by the one appropriate to our problem. Thus, our prediction layer will be trained on our new data based on the performance of the previous layers already pre-trained. Once this step is completed, we will proceed to a fine-tuning step, in order to adjust the parameters of all the layers of our models.

To do so, we will define the cost function adapted to the problem and we will first apply the gradient descent to the parameters of the newly defined fully connected layer only, with the different hyperparameters chosen. We will compare the performance of the model according to the chosen hyperparameters to determine which ones are the most suitable for training our models.

After training the new fully connected layer, we will untrain the upper layers of the model in the fine-tuning stage. We will follow the same approach as in the transfer learning step in order to make our models as efficient as possible.

4.1. Selected pre-trained models

VGG16:

VGG16 is a deep convolutional neural network (CNN) model widely used in computer vision. It was proposed by Simonyan and Zisserman in 2014 at the ImageNet Large Scale Visual Recognition Challenge (ILSVRC). The architecture of VGG16 consists of 16 layers, with convolutional layers stacked on top of each other in a simple and regular structure. Specifically, the VGG16 architecture consists of 13 convolutional layers and 3 fully connected layers (or FC), followed by a softmax function for classification. The convolutional layers use small filters (3x3) with a step size of 1, fills of 1 and ReLU activation functions, followed by a max clustering layer. The FC layers act as a final classification layer that takes the image features and predicts the corresponding object classes. The VGG16 architecture is often used for image classification in tasks such as object recognition and object detection. Its simple and regular structure makes it easy to understand and modify the architecture to accommodate different types of data and computer vision tasks.

ResNet-18:

The ResNet-18 model is a deep convolutional neural network (CNN) that was introduced in 2015 by He et al. in their paper "Deep Residual Learning for Image Recognition". ResNet-18 is a member of the ResNet family of models, which are known for their residual architecture that allows them to better handle vanishing gradient problems when training very deep neural networks. ResNet-18 has a relatively simple but very effective architecture. It has 18 convolution and pooling layers, followed by a fully connected layer for classification. The model takes as input images of size 224x224 and can classify these images into 1000 different categories. The special feature of ResNet's residual architecture is that it allows for a direct transition from the input information to the output layers, by adding skip connections that bypass certain layers. ResNet-18 is particularly useful for image classification, but can also be used for other computer vision tasks such as object detection or image segmentation. Because of its relative simplicity compared to other state-of-the-art models such as VGG or Inception, ResNet-18 is also faster to train and evaluate.

AlexNet:

AlexNet is a deep convolutional neural network model developed by Alex Krizhevsky, Ilya Sutskever and Geoffrey Hinton in 2012. It was the first model to win the ImageNet Large Scale Visual Recognition Challenge

(ILSVRC) in 2012, with a classification error of only 15.3%, a significant improvement over previous models. AlexNet's architecture consists of eight layers, including five convolution layers and three fully connected layers. The first layer is a convolution layer with an 11x11 core and a 4 stride, followed by a ReLU function and a 3x3 pooling layer. The following layers are similar, alternating convolution layers, ReLU functions and pooling layers. The last three layers are fully connected layers, with a first layer of 4096 neurons, followed by two layers of 4096 neurons each. The last layer is a classification layer with as many neurons as the number of classes to be predicted. AlexNet is often used for image classification tasks, especially in areas such as facial recognition, object detection, pattern recognition and image segmentation.

MobileNetV2:

The MobileNetV2 model is a convolutional neural network (CNN) model designed to be very efficient in terms of computational power and model size, while providing good image classification performance. It was introduced by Google in 2018. MobileNetV2's architecture features deep convolution blocks and bottleneck blocks that allow for high parameter reduction, while maintaining high classification accuracy. The model also uses group normalisation layers, deep weighting layers and residual connections to improve learning stability. MobileNetV2 is mainly used for image classification on tasks such as object recognition, object detection, semantic segmentation, etc.

4.2. Hyperparameters studied

Hyperparameters are adjustable parameters that let you control the model optimization process. Different hyperparameter values can impact model training and convergence rates.

We define the following hyperparameters for training:

- Number of Epochs: the number times to iterate over the dataset

- Batch Size: the number of data samples propagated through the network before the parameters are updated

- Learning Rate: how much to update models parameters at each batch/epoch. Smaller values yield slow learning speed, while large values may result in unpredictable behavior during training. We can adjust learning rate with a torch scheduler that could allow to manage the variation of the learning during the training steps automatically. For example we can use torch.optim.lr_scheduler.ReduceLROnPlateau, allowing dynamic learning rate reducing based on some validation measurements.

- Loss function: The loss (or loss function) is used to measure how far your model predictions are from the true labels of your training data. It is calculated by comparing the outputs of your model with the true labels for a given training set.
  The loss is then used to calculate the gradient of the cost function with respect to the model weights, which are then updated via the optimiser to minimise the loss. Thus, the lower the loss, the better the performance of your model.
  To resolve our classification task, we choose the Cross Entropy Loss which is the most appropriate. Indeed, Cross Entropy Loss is a cost function commonly used for classification problems. It is based on the principle of maximising the log-likelihood for the correct class. It measures the distance between the probability distribution predicted by the model and the actual distribution of classes.
  Indeed, the Cross Entropy Loss function is more sensitive to erroneous predictions, which can help to speed up convergence and achieve better classification performance.
  You could also point out that Cross Entropy Loss is easy to compute and optimise using standard error back-propagation methods and is implemented in many Deep Learning frameworks, which allows for easy and fast implementation.
- Optimizer: Optimization is the process of adjusting model parameters to reduce model error in each training step. Optimization algorithms define how this process is performed. All optimization logic is encapsulated in the optimizer object. Here, we use the SGD optimizer; additionally, there are many different optimizers available in PyTorch such as ADAM and RMSProp, that work better for different kinds of models and data.
  We initialize the optimizer by registering the model’s parameters that need to be trained, and passing in the learning rate hyperparameter.
  The different optimizers can be divided in two families: gradient descent optimizers and adaptive optimizers. This division is exclusively based on an operational aspect which forces you to manually tune the learning rate in the case of Gradient Descent algorithms

while it is automatically adapted in adaptive algorithms — that’s why they have this name.

- Regularization: Regularization techniques are used to prevent overfitting by adding a penalty to the model's loss function. Some common regularization techniques include weight decay (L2 regularization), dropout, and batch normalization.

Furthermore, we could also decided to use data augmentation, if it can help us to reach better result.

5. Experiences

We will start by loading our data, and applying some transformations to increase the performance of our models:

First of all, we perform the resizing mentioned above, in 256 by 256.

Next, we transform the image into a tensor so that the data can be manipulated by the PyTorch library.

Finally, the Normalize transformation is used to normalise the pixel values of the images. mean and std are the mean and standard deviation of the pixel values of the images on the training dataset. These two values are used to centre the data around zero and scale it. To determine the values of mean and std, we need to calculate the mean and standard deviation of the pixel values of all the images in our training dataset.

We chose to set the batch_size at 40, but if our device have not enough memory, we will put down to allow the models training.

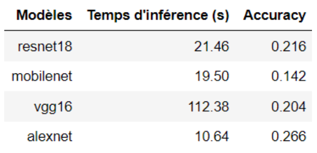
5.1. First performance without Transfer Learning

To begin our experiences, we will try to use directly the differents pretrained models, in order to watch their performance without transfer learning. Thus, we could see the effect of the different transfer learning steps and see how the models improve their ability to classify landscapes.

We put the four models into evaluation mode in order to make all predictions on the test dataset.

In our case, all classes are balanced and each class is represented equally as we have seen in the presentation of our data. Thus accuracy, which represents the number of correct predictions of the model divided by the total number of predictions made, is an appropriate metric to evaluate the performance of our model. Indeed, it will measure the ability of the model to correctly predict the different classes equally. If the classes were not balanced, it would have been necessary to use other metrics such as precision, recall or F1-score to better assess the performance of the model.

Here is thus, the models performance without any training on the test set containing 500 images (100 images of each class):



Modèles	Temps d'inférence (s)	Accuracy
resnet18	21.46	0.216
mobilenet	19.50	0.142
vgg16	112.38	0.204
alexnet	10.64	0.266

The accuracies are very low but it's normal, the models have a new prediction layer which has not been trained before. Then, despite the models have usually high performance, we have to make transfer learning in order to improve and take advantage of the pretrained models performance.

We also notice that the VGG model has a much longer inference time than the other models. This is due to its very large number of parameters, much larger than those of the other models.

5.2. Transfer Learning

We are going now to begin the Transfer learning with fine tuning step and the experiences in studying the listed above hyperparameters.

Freeze Early layers

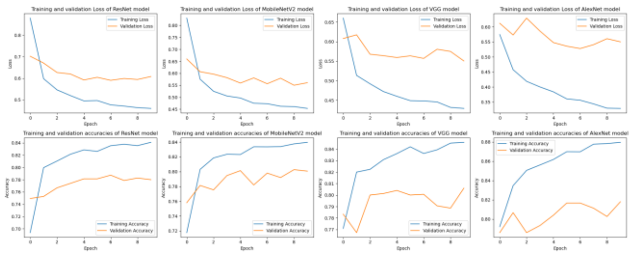
We freeze all of the existing layers in the network by setting requires_grad to False, and set the new classifying layer to adapt the different models to our problem.To begin, the new classifying layer is just a linear layer.

Optimizer

We will compare the first step of tranfer learning on two basics optimizer :

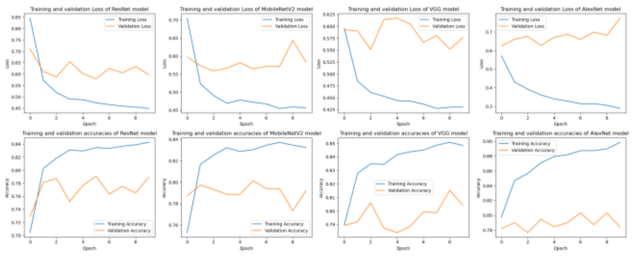
One gradient descent and one adaptative which will be SGD and Adam. We leave the default settings in order to study how the training works and to see afterwards what happens and if it is possible to improve the training with some modifications.

- SGD



```
       Modèles  Accuracy  Inference time  Total number of parameters
0       ResNet  0.787333      372.136650                    11179077
1  MobileNetV2  0.802667      374.301679                     2230277
2          VGG  0.806000      856.364979                   134281029
3      AlexNet  0.816667      365.326119                    57024325
```

- Adam



```
       Modèles  Accuracy  Inference time  Total number of parameters
0       ResNet  0.790667      370.796744                    11179077
1  MobileNetV2  0.793333      374.063680                     2230277
2          VGG  0.806000      855.969936                   134281029
3      AlexNet  0.782000      370.303783                    57024325
```

From this first training, we can note many things. Indeed, we can observe on the different graphs that the validation and training curves separate quite quickly, which is characteristic of overfitting. It is quite normal that the new classification layer overfits quickly during the learning transfer. This is due to the fact that the new layer is initially untrained, so it quickly learns the specific features of the new data set.

To avoid overfitting, we will use regularisation techniques such as L1/L2 regularisation, dropout or data augmentation.

Despite the overfitting, we can however point out that the ResNet18 and MobileNetV2 models seem to perform the best, especially with the SGD optimizer, even though the accuracy values are slightly lower than the other models that overfit completely.

We also observe slight stability problems as the values oscillate quite easily.

In the rest of our experiments, we will have to solve the two problems identified here:

1) Management of overfitting
2) Management of the stability of the training

<u>Management of overfitting</u>

Modifying the new classification layer can help avoid overfitting, as it is the last layer in the network and is responsible for the final prediction. Here are some changes we can make to the classification layer to avoid overfitting:

Add a regularisation: Adding an L1 or L2 regularisation to the classification layer to reduce the large weights in the layer. This can help reduce overfitting by limiting the complexity of the model.

Add Dropout: Adding Dropout to the classification layer to prevent the model from overfitting the training data. Dropout randomly disables neurons in the classification layer during training, preventing the model from overfitting to the training data.

Use a different activation function: Using of a different activation function for the classification layer. For example, we could try to use a ReLU activation function or a LeakyReLU activation function rather than a sigmoid activation function.

<u>Management of the stability</u>

To improve the stability of the training we can try the following manipulations:

Adjust the learning rate: If the learning rate is too high, the model parameters may "jump" from one optimal region to another, leading to large variations in loss and accuracy. In this case, you can try to reduce the learning rate to stabilise the training.

Use a scheduler: A scheduler allows you to adjust the learning rate as the training progresses. For example, the learning rate can be reduced each time the loss stagnates or the accuracy does not improve for a number of epochs. This can help to avoid unstable variations.

Increase the amount of data: If the amount of training data is low, the model may overfit the training data, leading to instability in loss and accuracy. In this case, you can try to increase the amount of training data by using data augmentation techniques, such as rotation, translation or symmetry.

<u>Modifying of the classification layer by adding some elements to adjust the management of overfitting and learning stability</u>

As a first step, we decided to change our classification layer which was too simple (it was a single nn.Linear layer). We replaced this classification layer with the following structure:

```
nn.Linear(num_ftrs, 256),
nn.ReLU(),
nn.Dropout(p=dropout_prob),
nn.Linear(256, num_classes)
```

This is a first linear layer that takes as input num_ftrs (the output number of the previous layer of the network) and outputs 256 features. In the context of deep learning, these features can be thought of as learned attributes that represent important characteristics of the data.

We then add a non-linear ReLU activation layer. This function takes as input the results of the previous linear layer and applies a non-linear function to produce non-linear outputs. This helps the model learn non-linear representations of the data.

nn.Dropout(p=dropout_prob): As mentioned earlier, we also add a dropout layer which is a technique to prevent overlearning. This layer randomly removes some of the neurons from the previous layer with a probability of dropout_prob, which forces the model to learn more robust features that are less dependent on certain neurons. In our features, we chose a probability of 0.3 to determine whether a neuron would be disabled or not. Then the structure ends with another linear layer that takes as input the outputs of the previous layer and produces final outputs corresponding to the number of classes to be predicted (num_classes).

Overall, this sequence creates a so-called "fully connected" layer (fc) that takes the features learned by the previous model and transforms them into an output that corresponds to the classes to be predicted.

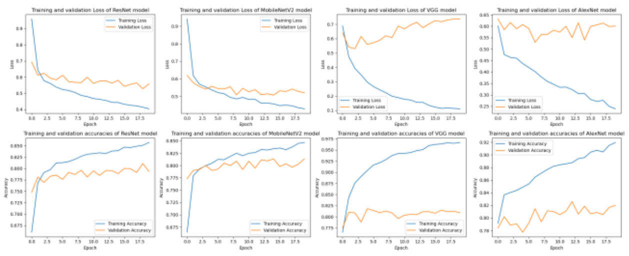
In addition, we add L2 regularisation to all parameters of the fully connected layer (fc) of our models.

We recall that the L2 regularisation consists in adding an extra term to the loss function which penalises the large weights of the model. This penalty, as mentioned above, limits the complexity of the model and prevents overfitting.

Finally we choose a ReduceLROnPlateau scheduler in order to adjust automatically the learning rate. The ReduceLROnPlateau scheduler is a learning rate scheduler in PyTorch that automatically adjusts the learning rate when the validation loss of the model stops improving. The scheduler monitors the validation loss and, when it stagnates for a certain number of epochs, it decreases the learning rate by a certain factor. In this way, it finds an optimal learning rate for a given model.

Our new classification layer being more complex than the previous one, we encountered memory problems when training the four models (Out of memory error). One way of dealing with this problem is to reduce the batch_size, which was initially set at 40. We will first reduce it to 20 in order to observe if the memory problems are solved.

After applying all these changes, we get these results:



```
      Modèles  Accuracy  Inference time  Total number of parameters
0      ResNet  0.810667     1134.308365                    11309125
1  MobileNetV2  0.813333     1041.530416                     2553093
2         VGG  0.808667     3495.495873                   135310661
3     AlexNet  0.814667     1159.483064                    58053957
```

We observe that by modifying the classification layer and giving it more parameters the results are much better, especially for the ResNet and MobileNet models. The large number of epochs chosen here completely overfits the other two models.

Looking at the total number of parameters of the different models, we see that the VGG and AlexNet models, which have respectively 5 times to 10 times more parameters than the ResNet model, seem to be complex architectures that do not solve the task we are interested in. These architectures can perform on 5-class classification but they were designed to handle large-scale image classification tasks. Being relatively large models with a large number of parameters, they may require significant computational resources to be trained and used effectively.

It is therefore more appropriate here to use a ResNet or MobileNet model. These are lighter and faster models that work particularly well on a limited number of classes. In addition, the ResNet model contains residual blocks to improve the performance of the model despite its four times higher number of parameters

The residual blocks help to circumvent the "vanishing gradient" problem by introducing "jump" or "shortcut" connections between layers, which allow information to jump from intermediate layers. This allows information to flow more easily through the network and facilitates the formation of deep neural networks.

In the following experiments, we will focus on the use of the ResNet and MobileNet models, which are more suitable for solving our task.

Data augmentation

We decided to use data augmentation in order to avoid overfitting and to reach better performance after training on the two models that we kept to continue our experiences.

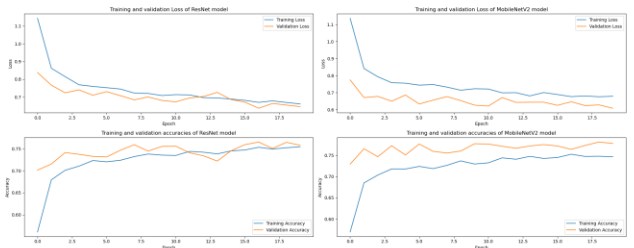
Here is a list of the transformations used:

Here is a brief description of the different transformations we do on our training set:

- transforms.Resize: Resizes images to a size of 256x256 pixels for training and validation.
- transforms.RandomCrop: Performs random cropping to achieve more diverse images during training.
- transforms.RandomHorizontalFlip: Flips images horizontally with a probability of 0.5 to increase the variations.
- transforms.RandomRotation: Randomly rotates images within a range of 15 degrees.
- transforms.ColorJitter: Randomly adjusts the brightness, contrast, saturation, and hue of the image to increase variations.
- transforms.ToTensor: converts images to PyTorch tensors.
- transforms.Normalize: Normalizes image tensors using means and standard deviations calculated from training data.

These transformations are only applied to the training set, because it is during this step that the parameters of the model are brought to evolve.

Then, we obtained better results as it is shown below:



```
       Modèles  Accuracy  Inference time  Total number of parameters
0       ResNet  0.765333     1005.882210                    11309125
1  MobileNetV2  0.778000     1053.336244                     2553093
```

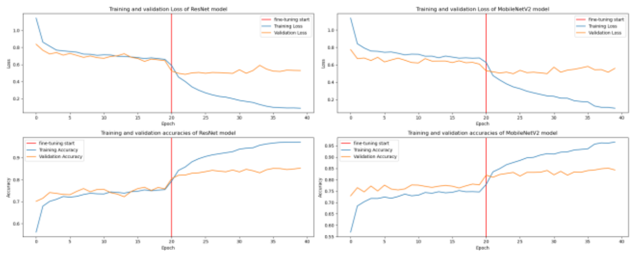
Indeed, we obtain very good results for the training of the classification layer since we observe a phenomenon of convergence between the curves of training and validation of the loss and accuracy. The curves are also very close between validation and training, which means that the models do not overfit the training data. This suggests that they can generalise correctly on the validation data.

We also observe that the accuracy reaches 76.5% for the ResNet18 model and 77.8% for the MobileNet model. These are very good performances as only the last classification layer was trained in this stage of our experiments.

We will now proceed to the fine-tuning stage in order to adjust the set of parameters of our models to the landscape recognition task.

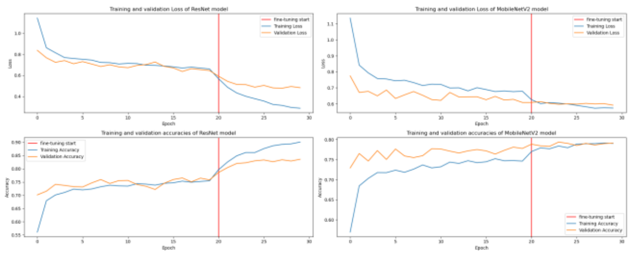
5.3. Fine tuning to train all the other layers of our models in order to fully adapt all the models to the landscape recognition specific task

We then unfroze all the top layers of our pre-trained models, while blocking the classification layer that we already trained in the previous steps:



```
       Modèles  Accuracy  Inference time  Total number of parameters
0       ResNet  0.821333     1043.991682                    11309125
1  MobileNetV2  0.832667     1056.663262                     2553093
```

We observe overfitting at the fine-tuning stage, as all the parameters fit the data except that as there are thousands of parameters the models fit too much to the data which explains the separations and the non convergence of the training and validation curves. In order to solve this overfitting problem at the fine-tuning level, we first consider re-training only some layers of the global model. By unfreezing only some specific layers of the model, we hope to achieve convergence. As the two models are different, we will choose the layers of the global models in different ways.



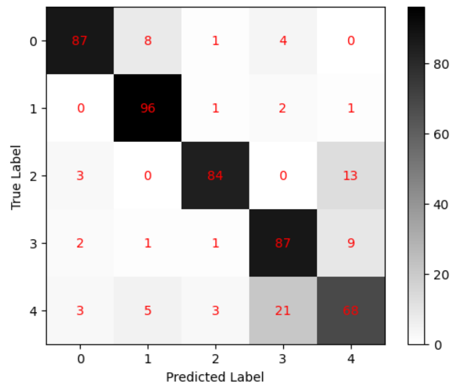
We have chosen to train only the deepest layers of the model in order to reduce model overfitting. Indeed, the deepest layers are used to capture the most specific features of each image. The results are not perfect, but relatively good. We will therefore test our two models on the test data. Nevertheless, it can be observed that fine-tunning has slightly more effects on the ResNet model. We wanted to obtain a double descent in the fine-tunning stage marking the learning progression of the model and here it is only very slight.

6. Test of our trained models on the test datasets

Here is the confusion matrix and the various associated metrics for the ResNet18 and MobileNetV2 models:

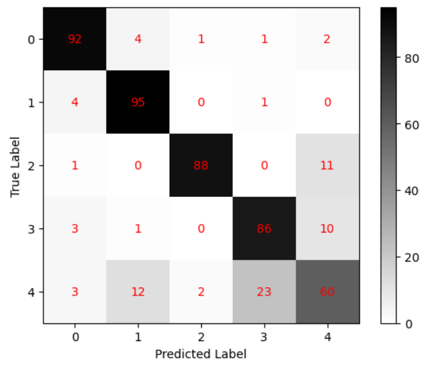
ResNet18

```
    Model  Accuracy  Precision  Recall  F1 score
0  ResNet     0.844   0.846452   0.844  0.843186
```



MobileNetV2:

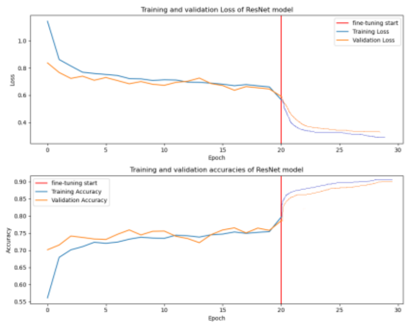
```
         Model  Accuracy  Precision  Recall  F1 score
0  MobileNetV2     0.842   0.841223   0.842     0.839
```



7. Conclusion

In conclusion, we have seen that by studying the different aspects of the hyperparameters chosen during the different stages of the training of the models, it was possible to significantly improve the quality and performance of the latter. Moreover, we had the opportunity to learn a lot about transfer learning and that its use could be very efficient provided that we initially choose the model(s) that are the most adapted to solve the studied task. Indeed, in our study we could see that very heavy architectures did not fit and tended very easily to overtake the training data because of their too large number of parameters.

However, we were finally able to obtain very efficient models, but they are not perfect and it would be possible to make them even more efficient if we had been able to make a more precise study of the layers to be trained during the fine-tuning. This study and time to work on this training step could allow us to obtain a great double descent as we can see on this example:



We encountered most of our difficulties on this part because the models tended to over-adjust our training data very quickly due to their complexity.

References

[1] https://arxiv.org/ftp/arxiv/papers/1804/1804.03928.pdf

[2] https://www.cse.ust.hk/~qyang/Docs/2009/tkde_transfer_learning.pdf

[3] https://towardsdatascience.com/image-classification-transfer-learning-and-fine-tuning-using-tensorflow-a791baf9dbf3

[4] https://towardsdatascience.com/transfer-learning-with-convolutional-neural-networks-in-pytorch-dd09190245ce

[5] https://medium.com/mlearning-ai/transfer-learning-and-convolutional-neural-networks-cnn-e68db4c48cca