Image colourisation E3 project report

By: Elliot CAMBIER, Kérien HARTWEG, Hugo KOTHE,
Olivier WANG, Caull YANG
Tutor: Imen KACHOURI

Table of contents

Thank you

entire COLORIA team would like to thank: Imen KACHOURI

Éric LLORENS

ESIEE Paris 2024 Projects Day

Context

General introduction

Artificial intelligence (AI), and more specifically Deep Learning techniques, have revolutionised many areas of research and industry thanks to their ability to model complex tasks without direct human intervention. E3 project falls within this dynamic framework, with the ambition of exploring the various techniques linked to image colourisation. This technique has many applications, particularly in the restoration old images. This project will result in design a human-machine interface (HMI), a website that brings together various colourisation methods to enable users to choose the technique that best suits their needs.
This report initially explores the current state of colourisation technologies, providing a basis that can be understood even by the uninitiated. We then present four specific colouring methods: their operation, advantages and limitations are detailed and compared. Each method is evaluated not only theoretically but also through a practical implementation within our website developed for this project, which also addresses accessibility issues.
The structure of the report is as follows: after an introduction that contextualises our study, we present a state of the art of deep learning techniques applicable to colourisation. We then detail the specific technologies and architectures used, such as GANs, CNNs, and advanced architectures such as U-NET and ResNet. The next section is devoted to the exposition of our models, the description of the datasets used, and the theoretical and practical comparative results. The web site developed is presented in detail before concluding with an analysis of the results and the prospects for our work in future.

Definition artificial intelligence and Deep Learning

To begin with, let's define Deep , using the CNIL's definition: "Deep learning is an automatic learning process using neural networks with several layers of hidden neurons. As these algorithms have a very large number of parameters, they require a very large amount of data in order to be trained".
There are two main categories of deep learning, supervised learning and unsupervised learning:

Supervised learning
Data = observations + labels Knowledge -> input-
output relationship
Unsupervised learning Data
= observations Knowledge -> latent
structures

Supervised learning involves using labelled data to train a model, where the system acquires the ability to associate specific inputs with desired outputs. contrast, unsupervised learning is not based labelled data and seeks to capture the intrinsic structure of the data by spotting patterns or clusters without recourse to known output data.

Image colourisation

Deep Learning colourisation is a method for colourising images in shades of grey, using neural networks. It is popular method in the field of computer vision, as it can be used to bring to life old images/videos that have been stripped of all colour.
The procedure common to all methods is as follows:

- Model training: in this first stage, a neural network model is trained on a database greyscale and colour images. The aim is to teach the model to predict the colours of pixels based on the colours of surrounding pixels and other characteristics of the image, such as contours and textures.
- Image colouring: Using this trained model, we can colour greyscale images by predicting the colours of the missing pixels. To do this, the model takes the greyscale image as input and produces a corresponding coloured image.

Analysis of the existing situation

There are many colouring methods, some older than others. Manual colouring is one of the oldest and requires artistic skills to apply the colours accurately and realistically. Users can draw strokes of colour directly onto the image, which are then propagated using the contours of image to define the colour spread [8, 16]. This method, also known as

The scribble method for colouring greyscale images has proved complex because of its requirement for precision in the drawings. However, other researchers have improved this method by making it more accessible using a colour palette and a brush, while reducing the amount of scribbling required and the potential for errors [9]. A variant of this approach allows the user to provide a colour image as a reference, making the colourisation task easier by copying the colours of the corresponding areas [10,15]. This can be referred to as colour : the algorithm converts the reference image into greyscale, and compares it with the image to be colourised. Objects" / "pieces" of the image with the same luminosity or shade of grey will be colourised according to the associated colour in the reference image.
As technology has advanced, more sophisticated colourisation methods have been developed, including those using convolutional neural networks (CNNs). These models, which are deep networks typically used image , learn to predict colours a dataset containing black and white images and their colour matches [11,14]. The architecture of CNNs allows them to capture features at different scales and to understand the overall structure as well as the local details of images, which is crucial for the colourisation task.
An interesting architecture in this field is "Classification via Retrieval" (CvR), which combines colour learning with the extraction semantic representations of images. This method not only predicts the appropriate colours, but also seeks understand the context and content of the images in order to improve the accuracy of the predicted colours [12].
The colourisation method [13] uses a neural network model based on "transformers", trained with greyscale images and their colour equivalents. This model learns to interpret the semantic content of greyscale images and to establish a correspondence between grey pixels and their colours. It processes greyscale images through successive layers of 'transformers', generating a probability map for the colour of each pixel. Attention mechanisms are used to focus on key areas of the image in order to optimise colour prediction.
Image colourisation represents a major challenge due to its complex nature: the same object can be associated with different plausible colours. Deep Learning techniques, effective, require large amounts of annotated data and sometimes struggle with complex images containing multiple objects or a busy background. Detection and

accurate classification of objects is crucial for realistic colourisation. It is also essential to correctly manage multiple instances that may overlap, ensuring a clear distinction between objects background, which facilitates the synthesis and manipulation of visual features [17].

Advanced architectures

Convolutional Neural Networks (CNNs): Specially designed to process structured data such as images, CNNs use filters to capture spatial relationships and features at various scales.
Generative Adversarial Networks (GANs): Comprising two competing networks, a generator and a discriminator, GANs [are used to create new data similar to real data, offering significant advances in image and colourisation.
Scribbling techniques: Enable users to guide the process
by drawing colour hints on a greyscale image, which the model then uses to colour the rest of the image.
Example-based colourisation: Uses a colour image as a reference to transfer the colour scheme to a corresponding greyscale image.

U-NET architecture

The U-NET architecture is widely adopted in computer vision for tasks such as image segmentation. Originally designed in 2015 for medical image segmentation, U-NET is characterised by its U-shaped structure, comprising two main parts: the encoder (contraction path) and the decoder (expansion path).

Encoder: The encoder is composed of multiple layers of convolution and max pooling. This configuration progressively reduces the dimensionality of input while extracting and compacting the important features of the image. This reduction is essential to capture the global context of the image without overloading the model in terms of computation and memory.

Decoder: The decoder uses transposed convolutions to reconstruct the image from the features encoded by the encoder. This part of the network increases the resolution of the image by step and merges the high-resolution features from encoder using the skip connections. These connections are crucial for restoring the local detail and structure of the original image.

U-NET is particularly effective even with limited datasets, as it can generalise from few examples thanks to its efficient feature transfer mechanism between encoder and decoder. This makes it ideal applications where annotated data is scarce or expensive to obtain. [1]

ResNet-34 and ResNet-101 architectures

ResNet-34 and ResNet-101 are two types of ResNet, special neural network model that uses "residual connections". These connections help to avoid the problem of gradients that disappear when the network is very deep[2].
ResNet-34 has 34 layers and is good for simple tasks or when you don't have a lot of computing power. ResNet-101 has 101 layers and can understand more complex details, which is useful for tasks that require a lot of precision.
In short, if the task is simple or if you have power limits, ResNet-34 is often sufficient. For more detailed tasks, ResNet-101 is preferable. The idea behind ResNet is to make neural networks deeper without losing performance. This is made possible by connections that allow signals to skip certain layers, facilitating the passage of information.

By models such as ResNet with U-NET to colour images, these systems learn not only to add colours realistically but also to respect the original shape of the images. These methods, whether supervised or unsupervised, use these techniques to obtain highly realistic and effective results, depending their intended use.

The models

Introduction of models

For our project, we explored four state-of-the-art models image colourisation, each distinguished by its approach and underlying architecture. These models are based on the principle of the U-NET architecture, well known for its effectiveness in computer vision tasks such as image segmentation.
Two of the models, named "Artistic" and "Stable", are developed around the unsupervised approach with generative adversarial network (GAN) architectures. A third model called "BigColor" is also being developed around GAN, but not using the U-NET architecture.
The latest model is a simple CNN based on U-net created by Rich Zhang.

So why have we chosen these models from the enormous range available?

DeOldify "artistic" and "stable

DeOldify's "Artistic" and "Stable" models have been selected their ability to offer optimal flexibility and efficiency thanks to their foundation on technologies. These models use an unsupervised approach generative adversarial network (GAN) architectures and are based on the U-NET architecture, renowned for its accuracy computer vision tasks such as image segmentation. This choice therefore makes it possible to generate high-quality colourised images without requiring large sets of labelled data, which considerably reduces the costs and complexities associated with the model formation process.
The availability of two variants, "Artistic" and "Stable", further enriches the project, allowing it to be adapted to different aesthetic and practical requirements. While "Artistic" favours a freer, more expressively coloured approach, ideal for renderings where the artistic aspect is favoured, "Stable" offers a more precise and natural colour reproduction, suited to the need for high visual fidelity. These characteristics, combined with the proven performance of these models, which are widely used and appreciated in the community, fully justify their choice for a project requiring a high level of quality and realism in the colourisation images. They are also quite popular, as we can find numerous articles and reports on them, making our understanding of these models all the easier.

KIMGEONUNG (BigColor)

The Big Color model stands out for its ability to make effective use deep learning architectures to process images. Big Color employs a mechanism based on deep convolutional networks (DCNs) and deep learning , optimised to process high-resolution images without compromising fine detail or colour saturation. This approach is particularly advantageous for colouring images, as it enables large-scale processing while preserving the quality and accuracy of colour nuances.

Big Color's strength lies in its ability to incorporate recent advances in artificial intelligence, including the use of self-learning and colour refinement techniques to improve the accuracy of the final results. This technology is chosen not only for its high performance in terms of colour fidelity and resolution of processed images, but also for its flexibility in processing a wide range of image types and lighting conditions. By incorporating these innovations, Big is positioned as a robust and versatile tool, ideal for applications requiring high-quality visual reproduction and complex colour transformations.

Rich ZHANG

The model developed by Rich Zhang, known as "Colorful Image Colorization", uses an innovative approach to colouring black and images, based on convolutional neural networks (CNNs). The distinguishing feature of this model is the integration of object class classification into the colourisation process, using predictions based on millions of reference coloured images to guide colour distribution. This enables it to colour selectively and appropriately according to the features detected in the image, improving the realism and accuracy of the final results.

Zhang's method is particularly notable for its ability to automatically process complex images with a high degree of detail and nuance. In addition, it offers a simple user interface that allows users to actively participate in the colourisation process, by suggesting colours specific regions, which can then be adjusted and perfected by the model. This feature makes the model highly accessible and adaptable, suitable for both professional applications and personal projects where the user's creativity and control are valued. This makes Zhang's model an effective choice for those seeking a an image colouring solution combining innovation, interactivity and high visual quality.

After 3 weeks, we selected these 4 models in order to evaluate and compare their image colouring performance.

Supervised models are trained with data where each greyscale image is linked to its colour version, allowing the model to learn the correspondence between the two directly. Unsupervised models, on the other hand, learn to generate plausible colourations without direct matches, making colour fidelity imperfect.

Dataset :
Our models were trained on 100,000 images from the ImageNet dataset [4], a large dataset widely used in computer vision competitions and research. ImageNet is ideal for this type of task as it includes a wide variety of high-resolution images with many different categories, allowing the model to generalise the colourisation to a wide range of subjects and scenes.

Type training, Advantages and Disadvantages

We started by testing the 4 models and we also did some research into them to compare the most common opinions and reviews of other testers with our own, so here's a summary of the most common pros and cons of each model:

DeOldify : https://github.com/jantic/DeOldify

The **Artistic** and **Stable** models are part of the same type of model called **NoGan**. The main idea behind this type of model is you can enjoy the benefits of GAN training while spending as little time as possible on direct GAN training.

Artistic :
Benefits: The "Artistic" model is optimised to deliver high-quality image colouring, accentuating detail and dynamism. It uses a sophisticated architecture combining ResNet 34 and U-Net, with a focus on a deep decoder layer, enabling it to produce images coloured with great richness details and an impressive vividness .[1]

Cons: However, this model presents challenges in terms of stability when applied to more mundane tasks, such as colouring natural scenes and portraits. It also requires careful adjustment of settings and considerable time to fine-tune results, which may prove impractical for everyday use or for users seeking solutions more direct and less laborious.[1]

Stable :
Benefits: The "Stable" model is particularly effective at colouring landscapes and portraits, producing superior results in these categories. Thanks to its advanced architecture based on ResNet 101 and U-Net, this model is capable of colouring human faces in a more natural way, avoiding the grey complexion effect often encountered with other colouring techniques.[1]
Disadvantages: However, it offers greater colour fidelity and less strange discolouration, this model tends to produce less saturated images. This can be seen as a disadvantage in cases where brighter or more expressive colours are desired, as it may lack the visual intensity offered by models such as "Artistic"[1].

This structuring helps to clearly assess the strengths and limitations of each model, providing a solid basis for choosing the most appropriate model for the user's specific needs.

Type of training: In DeOldify, **unsupervised learning** means that models do not need specific pairs of black-and-white images and their colour correspondents for training. Instead, the model learns autonomously to generate plausible colourations from features learned unlabelled images. The process is enhanced by the use of GANs, where a generator attempts to create colourised images and a discriminator assesses whether these images are realistic (i.e. indistinguishable from real colour images)[2].

KIMGEONUNG (BigColor) : https://github.com/KIMGEONUNG/BigColor

Big Color
Benefits: Big Color excels at handling large images and retaining fine detail at high resolution, thanks to its optimised architecture that combines advanced deep learning techniques. This model excels in particular at colourising complex images and

detailed, such as photographs of urban or natural landscapes, where precision of detail and richness of colour are paramount[7].

Disadvantages: Although the model is effective for processing large images, it can require significant computational resources, which could limit its use in less equipped environments. In addition, colour saturation can sometimes be less intense, which may not be suitable for projects requiring extremely vibrant and expressive colourisations[7].

Type of training : Big Color a deep learning approach, mainly supervised, requiring datasets consisting of black and white images with their coloured versions associated. This method enables the model to learn precise colour associations, which is crucial for the fidelity of the final results. The use of a powerful architecture also makes it possible to process large images efficiently, ensuring high-performance generalisation over a wide range of images[7].

Rich ZHANG : https://github.com/richzhang/colorization

Rich Zhang model
Benefits: Zhang's model uses an innovative combination of convolutional neural networks deep learning to deliver high-fidelity colourisation. Through the use of object classification and reinforcement learning techniques, the model is able to accurately colourise complex areas such as faces and natural textures. It also provides an interactive experience, allowing users to influence the colourisation by suggesting colours for specific regions, which is ideal for applications requiring a high degree of personalisation and human intervention.[6]

Drawbacks: Despite its ability to produce accurate results, the model can sometimes produce images where colours appear less vivid compared to those generated by more artistic methods such as DeOldify's "Artistic" model. In addition, the need manual intervention to adjust colours can be a drawback for users looking for a fully automatic solution[6].

Type of training : The model uses a semi-supervised learning approach that combines classical supervisory methods with elements of unsupervised learning. This enables the model to learn a relatively small number labelled images while exploiting a large number of large quantities unlabelled images, maximising its efficiency and ability to generalise from diverse training data.[6]

Theoretical comparison of models

Theoretically, Bigcolor should be the best performing: in previous studies, researchers compared BigColor with DeOldify (Stable, Artistic)[3].
Artistic is probably the best for a variety of tasks. It uses GANs and works without supervision, which is ideal for creating or modifying images creatively.
Stable is highly effective for tasks where the accuracy and consistency of the images generated are crucial.
Zhang is the least efficient, using only CNN. Nevertheless, it is suitable for basic images.

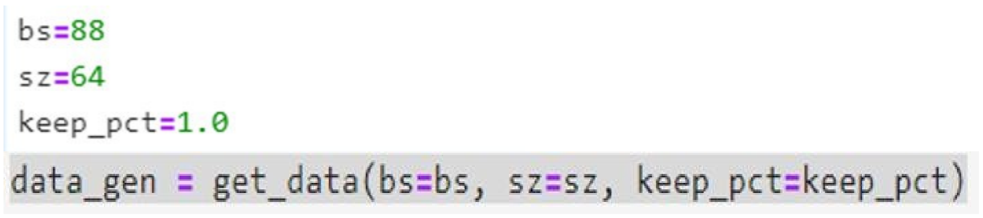
Model	BigColor	Zang	Stable	Artistic
U-net		x	x	x
Resnet 34				x
Resnet 101			x	
CNN		x		
GAN	x		x	x
Supervised		x		
No-supervised	x		x	x

The NoGan is a type of Gan whose aim is to minimise Gan training time and spend more time training the generators and the discriminator separately.
So during the short gan training period, the generator can colour correctly and almost without artefacts.
We're only going to look in detail at the drive one of the two models, because the two drives are in fact the same and the difference lies in the parameters.

Stable and artistic drive

First we need to prepare the DataSet we're going to use. We convert the 100,000 colourised images to greyscale to train the generator and discriminator.
The generator is driven and the aim is to push the generator drive as far as possible by varying the size of the image and the batch size.

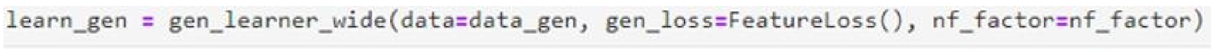
We start by defining the batch size, image size and the proportion of training data in order to load the data for the generator (see Figure 1):



```
bs=88
sz=64
keep_pct=1.0
data_gen = get_data(bs=bs, sz=sz, keep_pct=keep_pct)
```

Figure 1

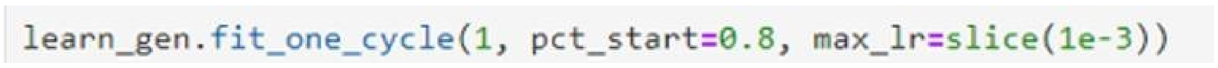
We then create our generator with the following parameters: the first parameter is used to pass our data to it, the second is used to define the loss function which compares the characteristics of the generated images with those of the generated images in order improve the generated images. The third indicates that the normalisation factor in this case is 2 (Figure 2).



```
learn_gen = gen_learner_wide(data=data_gen, gen_loss=FeatureLoss(), nf_factor=nf_factor)
```

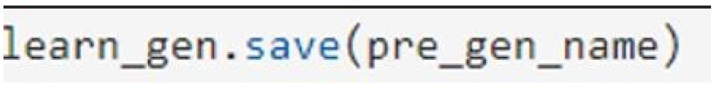
Figure 2

We start the generator training with a single epoch as parameter, 80% of the cycle the learning increases then decreases and we define a maximum learning rate of 1e-3(figure3). The current state of the generator is then saved (Figure 4).



```
learn_gen.fit_one_cycle(1, pct_start=0.8, max_lr=slice(1e-3))
```

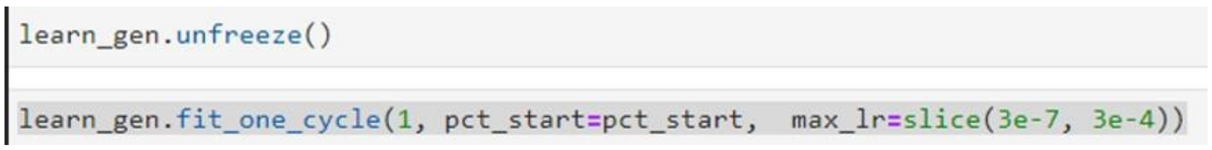
Figure 3



```
learn_gen.save(pre_gen_name)
```

Figure 4

We will unlock all layers of the model, which is blocked by default, in order to deepen the learning process (Figure 5). The parameters remain the same, except for the maximum learning , which varies between 3^{e}-7 and 3^{e}-4. The current state of the generator is then saved again (Figure 4).

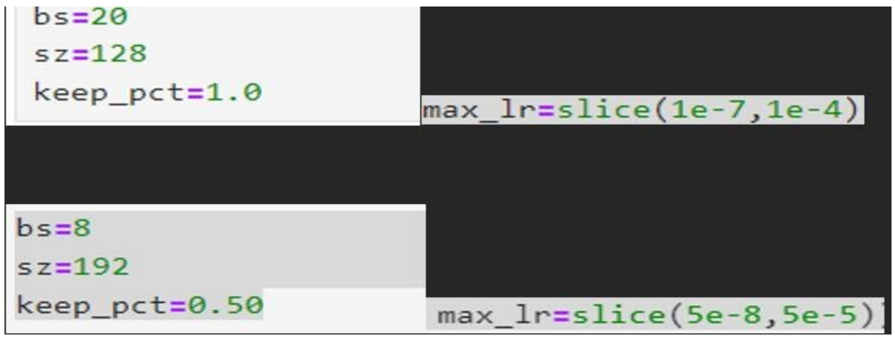


```
learn_gen.unfreeze()

learn_gen.fit_one_cycle(1, pct_start=pct_start,  max_lr=slice(3e-7, 3e-4))
```

Figure 5

We're going to re-train the generator in the same way, but with different parameters in order improve the colourisation of the images:

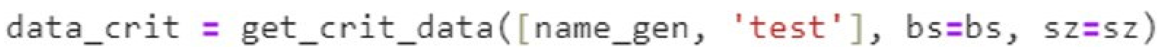


```
bs=20
sz=128
keep_pct=1.0
                        max_lr=slice(1e-7,1e-4)

bs=8
sz=192
keep_pct=0.50
                        max_lr=slice(5e-8,5e-5)
```

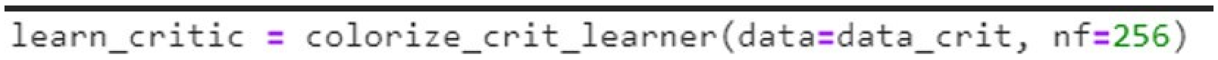
We're now going to move on to training the discriminator, and the aim is also to take the training as far as possible.

We start by loading the data with the batch size set to 64 and the image size set to 128 (Figure 6). Then we create the discriminator with the loaded data and specify number of filters we want to use, in this case 256(Figure 7).



```
data_crit = get_crit_data([name_gen, 'test'], bs=bs, sz=sz)
```

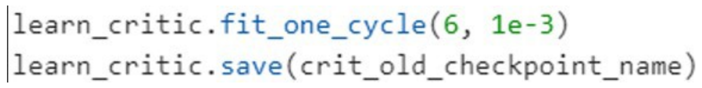
Figure c



```
learn_critic = colorize_crit_learner(data=data_crit, nf=256)
```

Figure 7

We run the training with 6 epochs and a maximum learning rate of 1^{e}-3 and save the current state of the model (Figure 8).

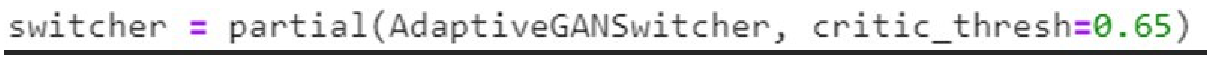


```
learn_critic.fit_one_cycle(6, 1e-3)
learn_critic.save(crit_old_checkpoint_name)
```

Figure 8

We will repeat the previous part several times, at least 5 times, but with a batch size of 16, 192 image sizes ,4 epochs and a learning rate of 1^{e}-4. The number of repetitions varies according to what we obtain when we repeat the Gan training.

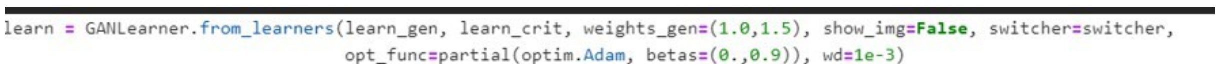
To configure the gan, we set the switcher parameter. A switcher alternates between driving the generator and the critic. The critic_tresh indicates the critical threshold of the discriminator, beyond the drive switches to the generator. (Figure 9)



```
switcher = partial(AdaptiveGANSwitcher, critic_thresh=0.65)
```

Figure S

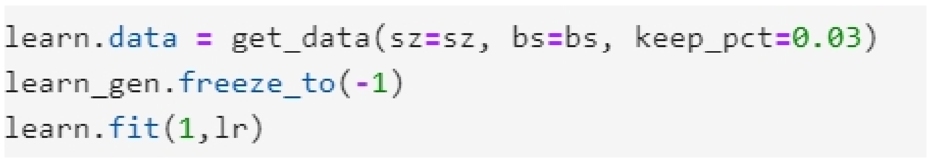
We then create the gan by loading the generator, the discriminator, the weights for the losses for the generator, optimisation function, the switcher, the optimiser function to update the model weights with the Adam optimiser and the hyper parameter, the weight decay avoid overlearning. (Figure 10)



```
learn = GANLearner.from_learners(learn_gen, learn_crit, weights_gen=(1.0,1.5), show_img=False, switcher=switcher,
                                 opt_func=partial(optim.Adam, betas=(0.,0.9)), wd=1e-3)
```

Figure 10

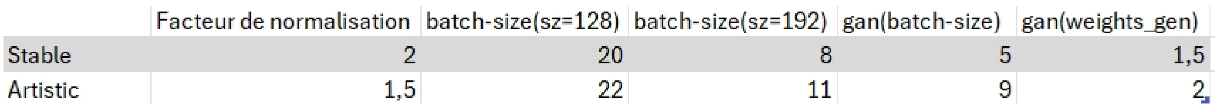
Training will only be carried out on 30% of the dataset and all the layers of the generator are frozen, except for the last layer, which is the output, in order speed up training. We launch the training with an epoch and a maximum learning rate of lr=2^{e}-5 (Figure 11).



```
learn.data = get_data(sz=sz, bs=bs, keep_pct=0.03)
learn_gen.freeze_to(-1)
learn.fit(1,lr)
```

Figure 11

To conclude, here is a table summarising the different parameters between Stable and Artistic that we will use (Figure 12).



	Facteur de normalisation	batch-size(sz=128)	batch-size(sz=192)	gan(batch-size)	gan(weights_gen)
Stable	2	20	8	5	1,5
Artistic	1,5	22	11	9	2

Figure 12

BigColor training

Let's now analyse the training code for the Big Color model, which was by far the longest, and organise our training process for this model into eight main stages.

We start with the **initial configuration**, where we define the essential parameters of the model. Next, **model definition** involves setting up the generator and discriminator architectures. We then move **to optimisation and planning**, where we configure the learning rate optimisers and planners. To ensure optimum efficiency across multiple GPUs, we prepare the system with **distributed training**.

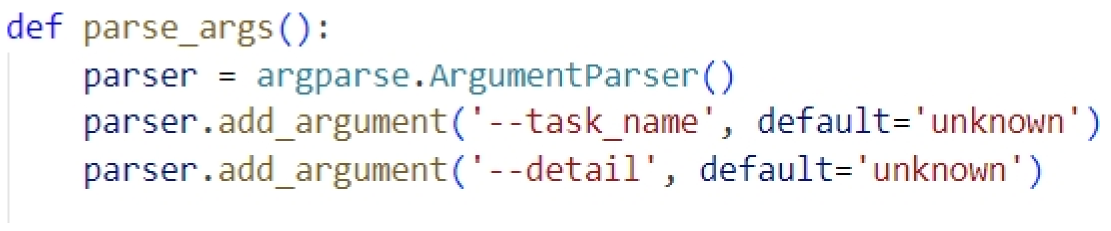
The training loop itself processes the data in batches, performing forward propagation and backpropagation. **Logging and tracking** are used to record progress and adjust parameters if necessary. The use of the **exponential moving average (EMA)** is crucial for stabilising model weights over time. Finally, **image processing** ensures that all inputs are correctly formatted and ready to be processed by the network.

These steps together form a complete process that guides our learning of the image colourisation model from start to finish.

1) **Initial configuration**

In the first phase of the training process, we focus on the initial configuration of the code. Here, the argparse library is used to manage the configuration options, allowing the user to specify a variety of parameters such as file paths, normalisation types and activation . Paths to pre-trained configuration files and log directories are also defined, ensuring meticulous organisation and complete preparation before training begins.

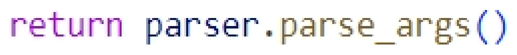
This fundamental step is encapsulated by the code's extensive parse_args() function, which plays a crucial role in customising and optimising all aspects of the model and its training environment.



```python
def parse_args():
    parser = argparse.ArgumentParser()
    parser.add_argument('--task_name', default='unknown')
    parser.add_argument('--detail', default='unknown')
```

...

Figure 13: Start of the parse_args() function

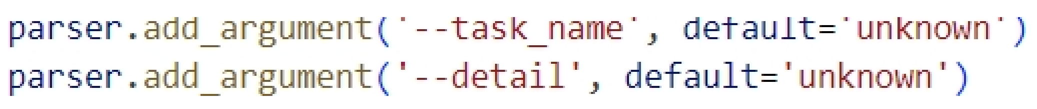


```python
return parser.parse_args()
```

*Figure 14: End of the **parse_args()** function*

We use this function to define and process command line arguments that we can specify to customise the execution of the training script. This is essential to make the script flexible and suitable for different training environments or model configurations without us needing to modify the source code. This function itself consists of 11 parts.

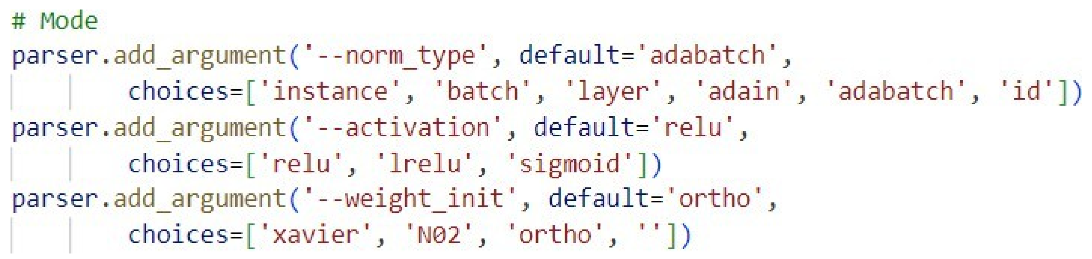
The `parse_args` function includes a general configuration part which identifies the different training tasks by a name (`task_name`) and allows us to provide additional `details` in order to better understand the objective or specificity of each task (Figure 15).



```python
parser.add_argument('--task_name', default='unknown')
parser.add_argument('--detail', default='unknown')
```

Figure 15

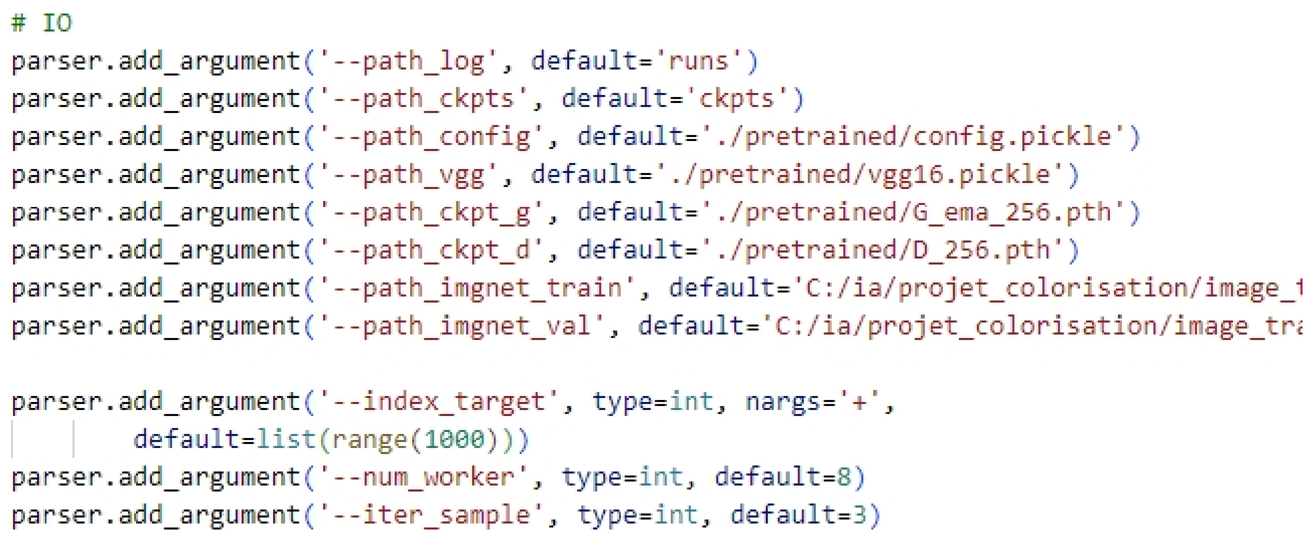
Next, we have an operating mode section which configures the internal behaviour of the model. This allows us to select the type of data normalisation (`norm_type`), the activation function used by the neurons (`activation`), and the weight initialisation method (`weight_init`). All these elements are essential for optimising the performance and convergence of the network. (Figure 16)



```
# Mode
parser.add_argument('--norm_type', default='adabatch',
        choices=['instance', 'batch', 'layer', 'adain', 'adabatch', 'id'])
parser.add_argument('--activation', default='relu',
        choices=['relu', 'lrelu', 'sigmoid'])
parser.add_argument('--weight_init', default='ortho',
        choices=['xavier', 'N02', 'ortho', ''])
```

Figure 1c

The next section, dedicated to inputs and outputs, defines the access paths to the files required training. Here we find the training data, the pre-trained models, as well as the save locations for logs and checkpoints.
This section plays a crucial role in the organisation and management of our project resources. (Figure 17)

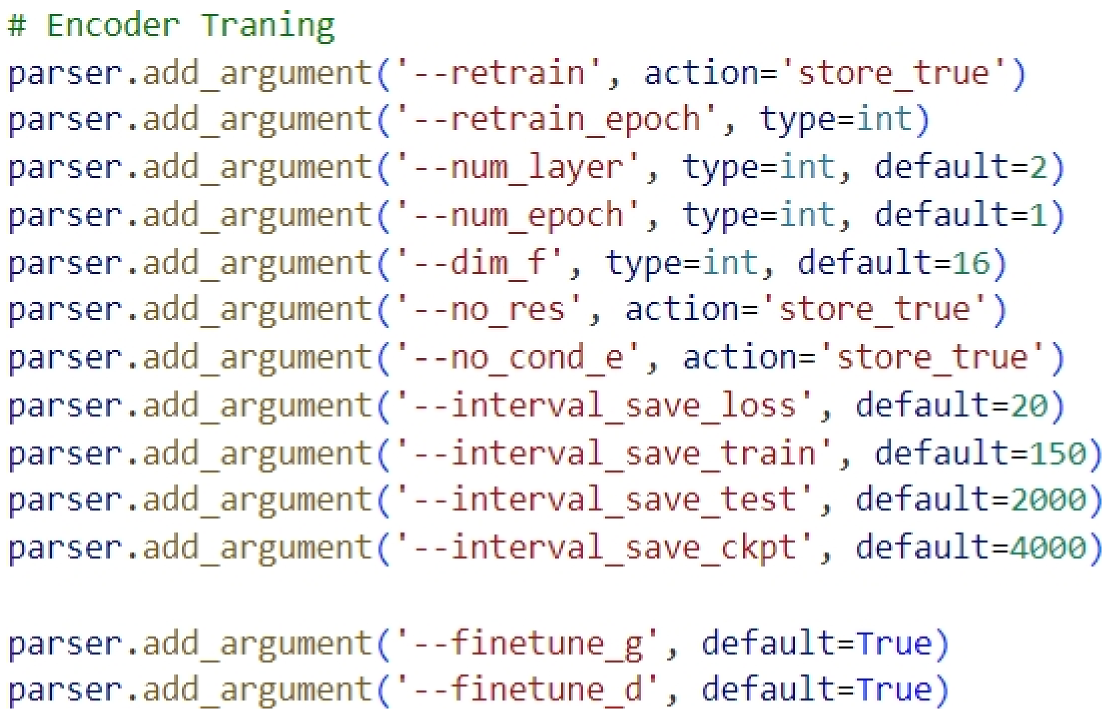


```
# IO
parser.add_argument('--path_log', default='runs')
parser.add_argument('--path_ckpts', default='ckpts')
parser.add_argument('--path_config', default='./pretrained/config.pickle')
parser.add_argument('--path_vgg', default='./pretrained/vgg16.pickle')
parser.add_argument('--path_ckpt_g', default='./pretrained/G_ema_256.pth')
parser.add_argument('--path_ckpt_d', default='./pretrained/D_256.pth')
parser.add_argument('--path_imgnet_train', default='C:/ia/projet_colorisation/image_t
parser.add_argument('--path_imgnet_val', default='C:/ia/projet_colorisation/image_tra

parser.add_argument('--index_target', type=int, nargs='+',
        default=list(range(1000)))
parser.add_argument('--num_worker', type=int, default=8)
parser.add_argument('--iter_sample', type=int, default=3)
```

Figure 17

The training section of the model is particularly interesting because it has a direct impact on the training process. Here we can determine the number of epochs (`num_epoch`), the number of layers in the model (`num_layer`), and other architectural parameters influencing the structure and complexity of the network. (Figure 18)

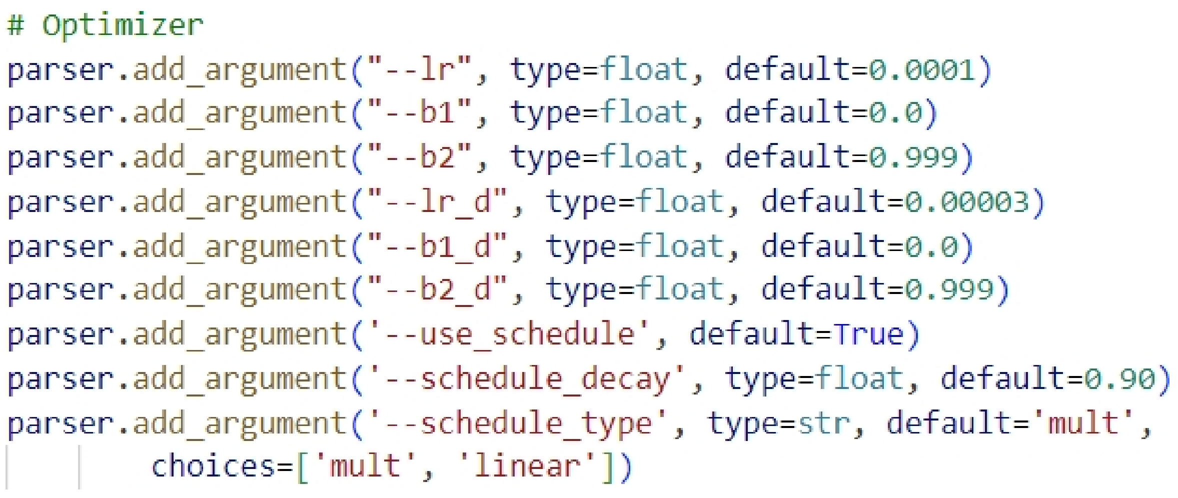


```python
# Encoder Traning
parser.add_argument('--retrain', action='store_true')
parser.add_argument('--retrain_epoch', type=int)
parser.add_argument('--num_layer', type=int, default=2)
parser.add_argument('--num_epoch', type=int, default=1)
parser.add_argument('--dim_f', type=int, default=16)
parser.add_argument('--no_res', action='store_true')
parser.add_argument('--no_cond_e', action='store_true')
parser.add_argument('--interval_save_loss', default=20)
parser.add_argument('--interval_save_train', default=150)
parser.add_argument('--interval_save_test', default=2000)
parser.add_argument('--interval_save_ckpt', default=4000)

parser.add_argument('--finetune_g', default=True)
parser.add_argument('--finetune_d', default=True)
```

Figure 18

As far as the optimisers are concerned, we start by defining parameters relating to optimisation, such as learning rate and moments. These parameters affect how the model adjusts its weights in response to the error calculated during training. We also have options to dynamically adjust these rates via a `scheduler`. (Figure 19)



```python
# Optimizer
parser.add_argument("--lr", type=float, default=0.0001)
parser.add_argument("--b1", type=float, default=0.0)
parser.add_argument("--b2", type=float, default=0.999)
parser.add_argument("--lr_d", type=float, default=0.00003)
parser.add_argument("--b1_d", type=float, default=0.0)
parser.add_argument("--b2_d", type=float, default=0.999)
parser.add_argument('--use_schedule', default=True)
parser.add_argument('--schedule_decay', type=float, default=0.90)
parser.add_argument('--schedule_type', type=str, default='mult',
            choices=['mult', 'linear'])
```

Figure 1S

The options part of verbose allows us to control the display of current configurations, which helps us to understand and debug the behaviour of the model during its execution. (Figure 20)

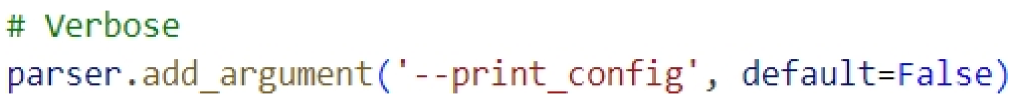

```
# Verbose
parser.add_argument('--print_config', default=False)
```

Figure 20

Pre-training management determines whether we should load pre-trained weights for the generator and discriminator. This approach can speed up training and initial model performance by providing a more advanced starting point (Figure 21).

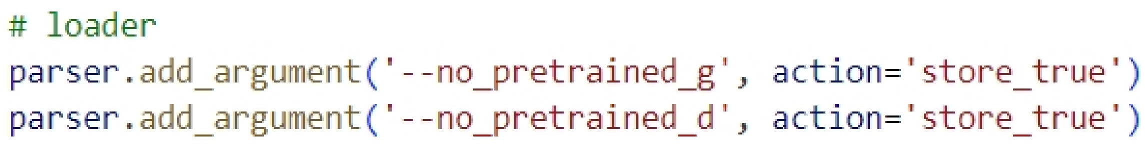

```
# loader
parser.add_argument('--no_pretrained_g', action='store_true')
parser.add_argument('--no_pretrained_d', action='store_true')
```

Figure 21

The configuration of loss functions defines the metrics by which we evaluate model error during training. We have different loss available to measure different facets of model performance (Figure 22).

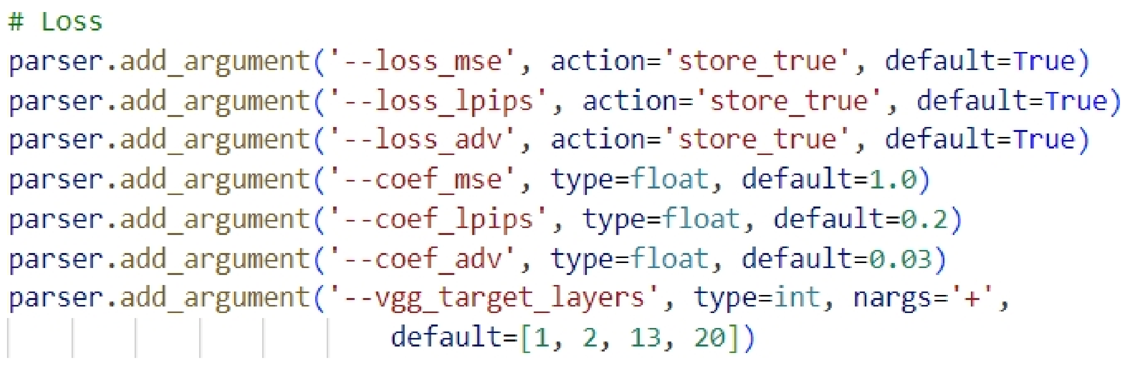

```
# Loss
parser.add_argument('--loss_mse', action='store_true', default=True)
parser.add_argument('--loss_lpips', action='store_true', default=True)
parser.add_argument('--loss_adv', action='store_true', default=True)
parser.add_argument('--coef_mse', type=float, default=1.0)
parser.add_argument('--coef_lpips', type=float, default=0.2)
parser.add_argument('--coef_adv', type=float, default=0.03)
parser.add_argument('--vgg_target_layers', type=int, nargs='+',
                    default=[1, 2, 13, 20])
```

Figure 22

The exponential moving average (EMA) is configured to stabilise generator weights over time. This makes the model results less sensitive to variations in the most recently processed data and more robust in general. (Figure 23)

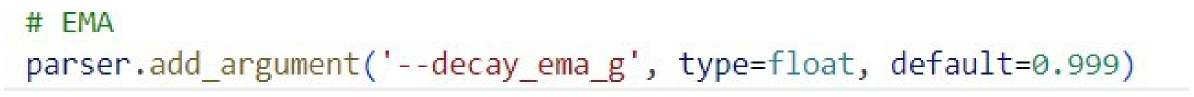
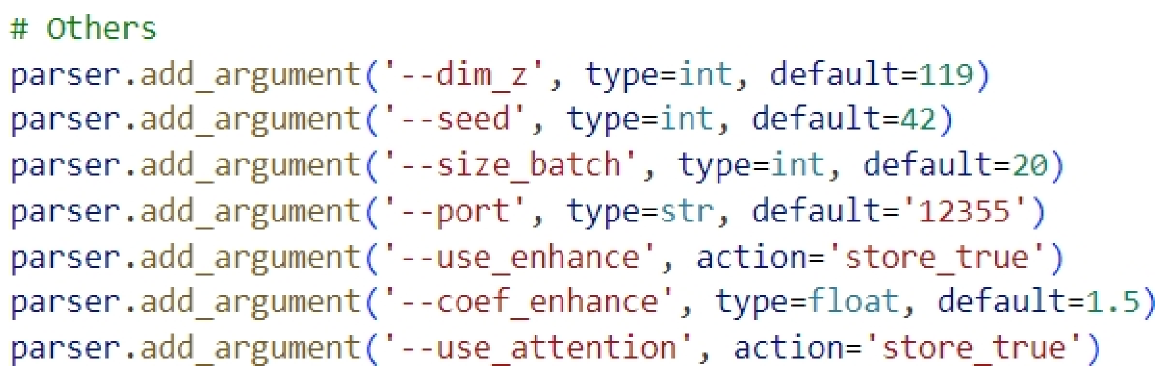

```
# EMA
parser.add_argument('--decay_ema_g', type=float, default=0.999)
```

Figure 23

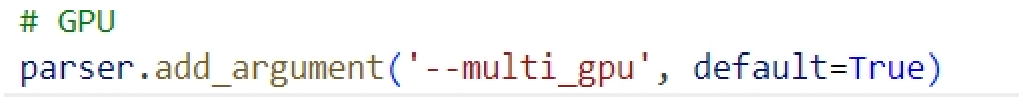
Another part includes variety of other important settings, such as the size of the latent space and the seed for random generation. These parameters offer finer control over specific aspects training and model initialization (Figure 24).



```
# Others
parser.add_argument('--dim_z', type=int, default=119)
parser.add_argument('--seed', type=int, default=42)
parser.add_argument('--size_batch', type=int, default=20)
parser.add_argument('--port', type=str, default='12355')
parser.add_argument('--use_enhance', action='store_true')
parser.add_argument('--coef_enhance', type=float, default=1.5)
parser.add_argument('--use_attention', action='store_true')
```

Figure 24

Finally, the last section is devoted to the GPU options, allowing us to configure the use of the available graphics resources. The aim is to training efficiency by using one or more GPUs, depending on the availability and compatibility of the hardware infrastructure (Figure 25).

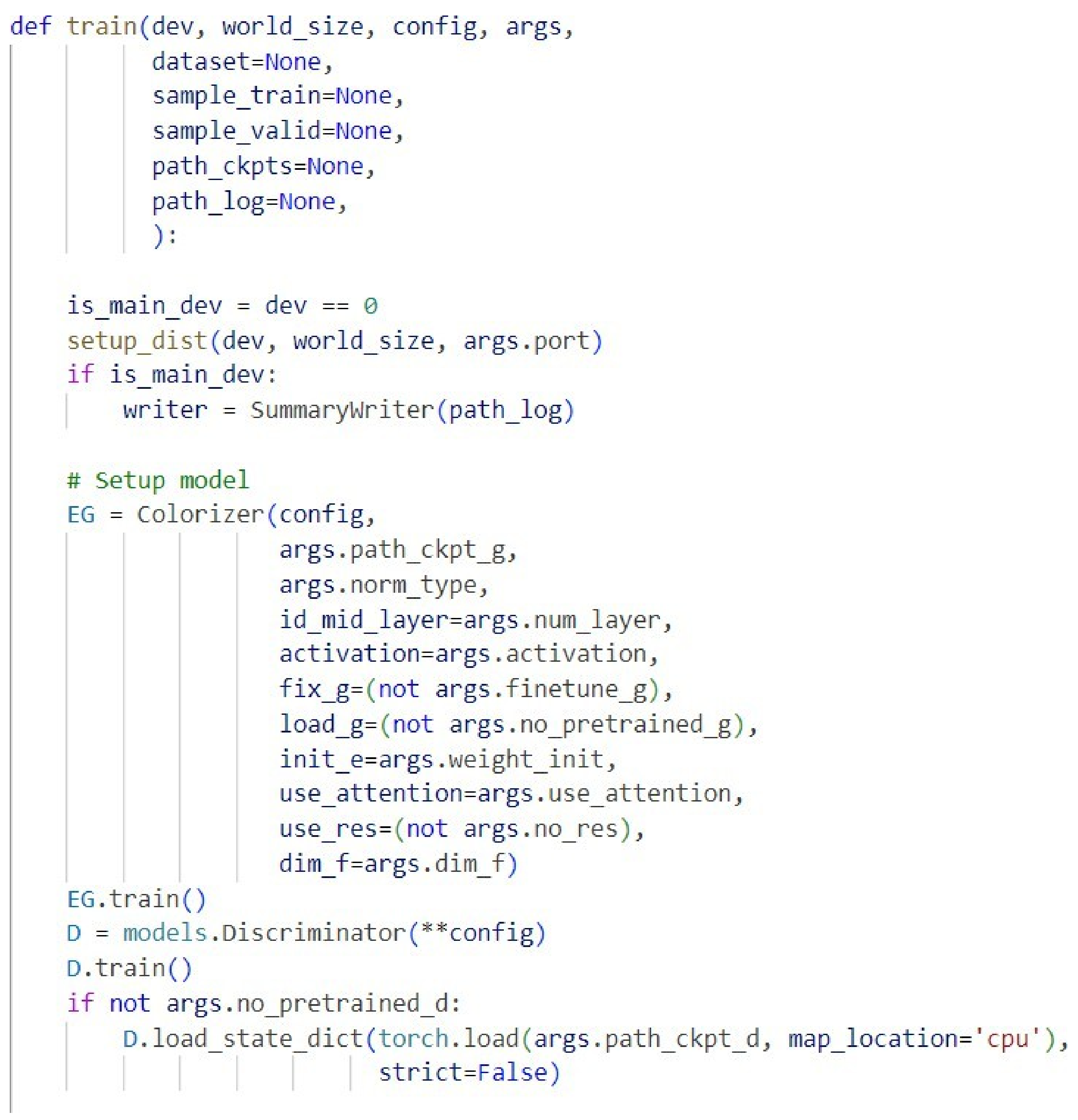


```
# GPU
parser.add_argument('--multi_gpu', default=True)
```

Figure 25

2) **Model definition**

As far as model definition is concerned, the code initializes two types of model: a Colorizer model, intended for image generation, and a Discriminator used adversarial learning. This step also includes the possibility using optional pre-trained models, making it easier to fine-tune models to improve their accuracy and efficiency from already established configurations. This configuration is essential for adapting and optimising the model according to the specific needs and characteristics of the data being processed. (Figure 26)



```python
def train(dev, world_size, config, args,
          dataset=None,
          sample_train=None,
          sample_valid=None,
          path_ckpts=None,
          path_log=None,
          ):

    is_main_dev = dev == 0
    setup_dist(dev, world_size, args.port)
    if is_main_dev:
        writer = SummaryWriter(path_log)

    # Setup model
    EG = Colorizer(config,
                   args.path_ckpt_g,
                   args.norm_type,
                   id_mid_layer=args.num_layer,
                   activation=args.activation,
                   fix_g=(not args.finetune_g),
                   load_g=(not args.no_pretrained_g),
                   init_e=args.weight_init,
                   use_attention=args.use_attention,
                   use_res=(not args.no_res),
                   dim_f=args.dim_f)
    EG.train()
    D = models.Discriminator(**config)
    D.train()
    if not args.no_pretrained_d:
        D.load_state_dict(torch.load(args.path_ckpt_d, map_location='cpu'),
                          strict=False)
```

Figure 2c

The code starts by configuring the environment, enabling synchronised training across multiple GPUs. The models we use, the Colorizer (generator) and the Discriminator, are initialized according to the defined parameters, and the configurations are potentially to use pre-trained states speed up the process and improve initial performance.

Specific optimisers are then put in place for each model, adjusting learning rates and moment parameters as required. This is completed with learning rate planners, if necessary, to dynamically adjust the rate as we progress. Particular attention is paid to to maintain the stability and reliability of the results by using the exponential moving average of the generator weights over time.

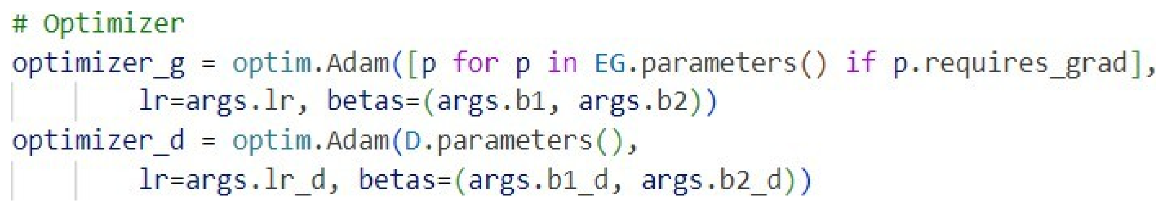
To maximise the efficiency of parallel training, the code optimises data loading and distribution. Its main training processes data in batches, colouring images, evaluating them, and adjusting our models by backpropagating errors calculated from various loss functions. The code also provides regular checkpoints to monitor performance and save the states of our models at crucial stages.

In short, this function represents a complete and rigorously structured training ecosystem. Aimed at optimising and supervising the development of high-performance, accurate image colourisation models, suitable execution on systems with distributed computing resources.

3) **Optimisation and planning**

In optimisation and planning section, we configure the parameters that will govern the behaviour of our models training. This section lays the foundations for effective adaptation of the network weights as we progress.

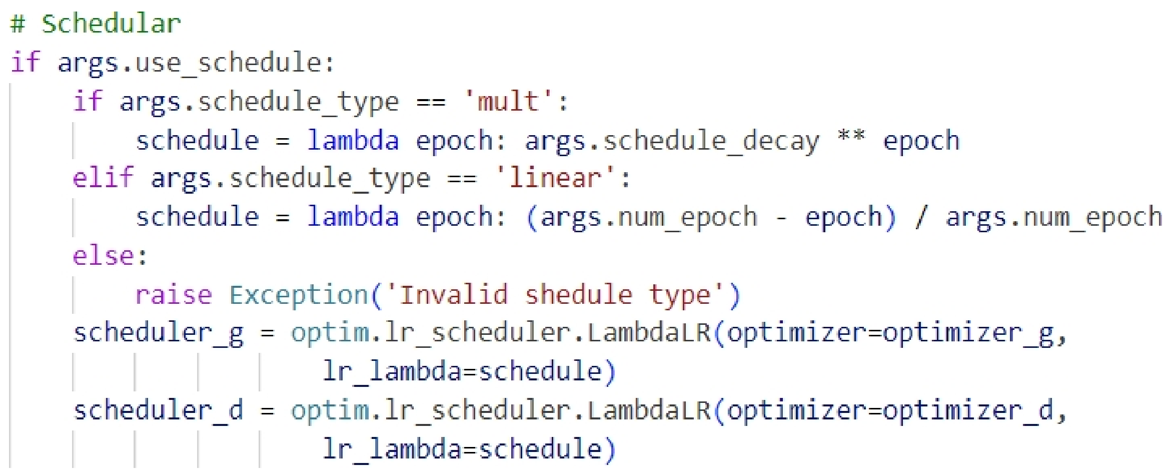
For the Optimizer Configuration, a specific optimizer is defined for each model. The generator (Colorizer) and the discriminator have their respective optimisers, using Adam's algorithm. Adam is chosen for its ability to handle sparse gradients efficiently and to adapt dynamically to different types of data. The code configures the learning rates and beta moments (b1 and b2) according its recommendations or previous experience, where b1 controls the exponential decay of the gradient estimation rate and b2 that of the square of the gradient, for a finer adjustment and more predictable behaviour during learning. (Figure 27)



```
# Optimizer
optimizer_g = optim.Adam([p for p in EG.parameters() if p.requires_grad],
        lr=args.lr, betas=(args.b1, args.b2))
optimizer_d = optim.Adam(D.parameters(),
        lr=args.lr_d, betas=(args.b1_d, args.b2_d))
```

Figure 27

In Learning Rate Scheduling, the code adjusts the learning rates for each epoch using a scheduler that follows a predefined programme. This planning system can take different forms, such as multiplicative (mult) or linear, influencing the speed at which the rate decreases. This strategy aims to refine learning by gradually reducing the rate as the model converges, which optimises performance and stabilises the network. (Figure 28)



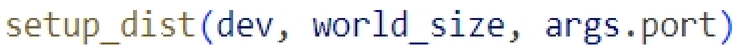
```python
# Schedular
if args.use_schedule:
    if args.schedule_type == 'mult':
        schedule = lambda epoch: args.schedule_decay ** epoch
    elif args.schedule_type == 'linear':
        schedule = lambda epoch: (args.num_epoch - epoch) / args.num_epoch
    else:
        raise Exception('Invalid shedule type')
    scheduler_g = optim.lr_scheduler.LambdaLR(optimizer=optimizer_g,
                                              lr_lambda=schedule)
    scheduler_d = optim.lr_scheduler.LambdaLR(optimizer=optimizer_d,
                                              lr_lambda=schedule)
```

Figure 28

In summary, this section of the ensures that our models benefit an adaptive optimisation environment that promotes efficient and smooth convergence. This reduces the risk of getting stuck in local minima or over-fitting to the training data. These optimisation and planning tools are essential for exploiting the full potential of complex architectures and obtaining the best results.

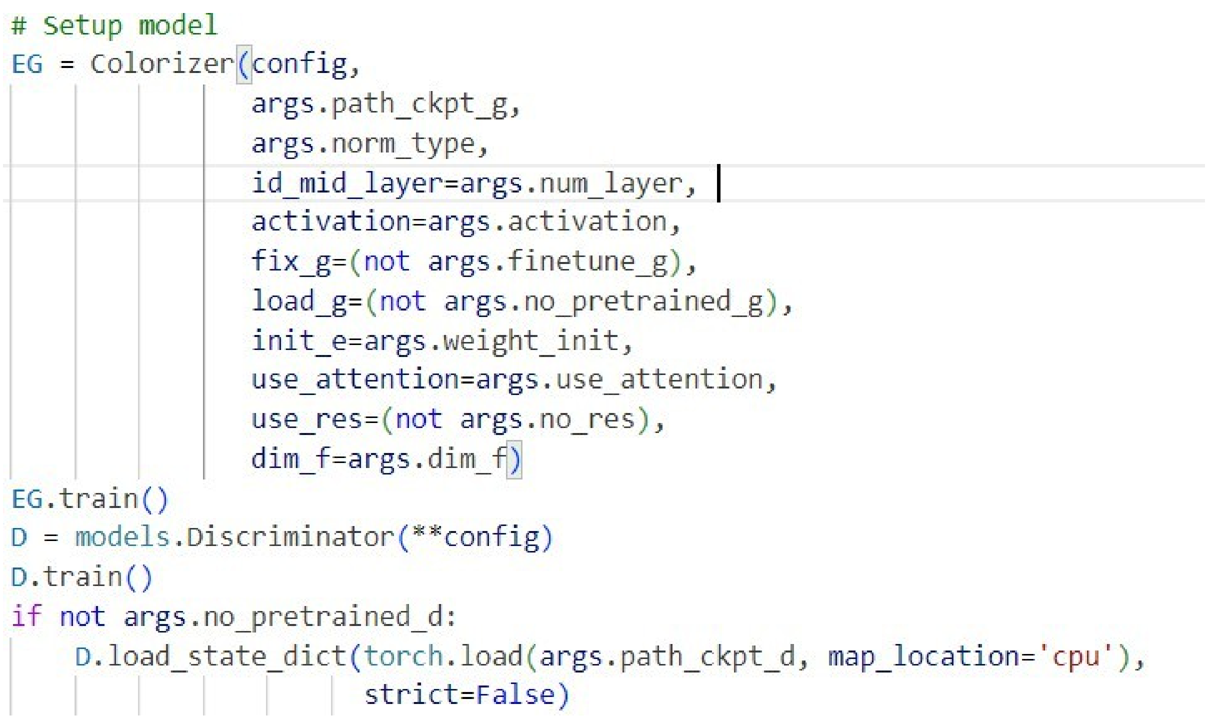
4) Distributed training

In the Distributed Training section, the code is specifically designed to run on multiple GPUs using DistributedDataParallel. This approach enables efficient parallel training on multi-GPU architectures. The code also methods such as autocast and GradScaler to optimise execution in terms of memory and computation. In particular, it will allow us use half-precision operations (FP16), playing a crucial role in maximising training efficiencyThis reduces the time required and increases the model's ability to process large amounts of data without saturating the available memory. (Figure 29 C 30)



```python
setup_dist(dev, world_size, args.port)
```

Figure 2S



```
# Setup model
EG = Colorizer(config,
               args.path_ckpt_g,
               args.norm_type,
               id_mid_layer=args.num_layer,
               activation=args.activation,
               fix_g=(not args.finetune_g),
               load_g=(not args.no_pretrained_g),
               init_e=args.weight_init,
               use_attention=args.use_attention,
               use_res=(not args.no_res),
               dim_f=args.dim_f)
EG.train()
D = models.Discriminator(**config)
D.train()
if not args.no_pretrained_d:
    D.load_state_dict(torch.load(args.path_ckpt_d, map_location='cpu'),
                      strict=False)
```

Figure 30

This part of the train function, dedicated to distributed training, manages the execution of the model on several graphics processing units (GPUs), which is essential for efficiently processing large volumes of data and complex architectures in reasonable timescales.

In the Distributed Environment Initialisation phase, the code begins by configuring the connections needed for communication between the multiple GPUs, defining the addresses and ports required. This step includes the use of dist.init_process_group with gloo as the backend, facilitating the inter-process exchanges essential for synchronised training on different devices.

For the PyTorch Configuration for Distribution, each component of the model - be it the generator (EG), the discriminator (D), or the perception tool VGG - is integrated into a DistributedDataParallel (DDP) architecture. DDP, a PyTorch tool, automatically parallelizes data and calculations across the available GPUs. This parallelization ensures a balanced distribution of work, efficient aggregation of gradients, and optimizes the use of computational resources for fast and efficient training.

GPU Resource Management is also of critical importance, involving the optimisation of memory and devices to avoid conflicts. Functions such as torch.cuda.set_device and torch.cuda.empty_cache are used to specifically assign each process to a dedicated GPU and free up unused memory.

This prevents the common problems of memory saturation encountered with complex models.

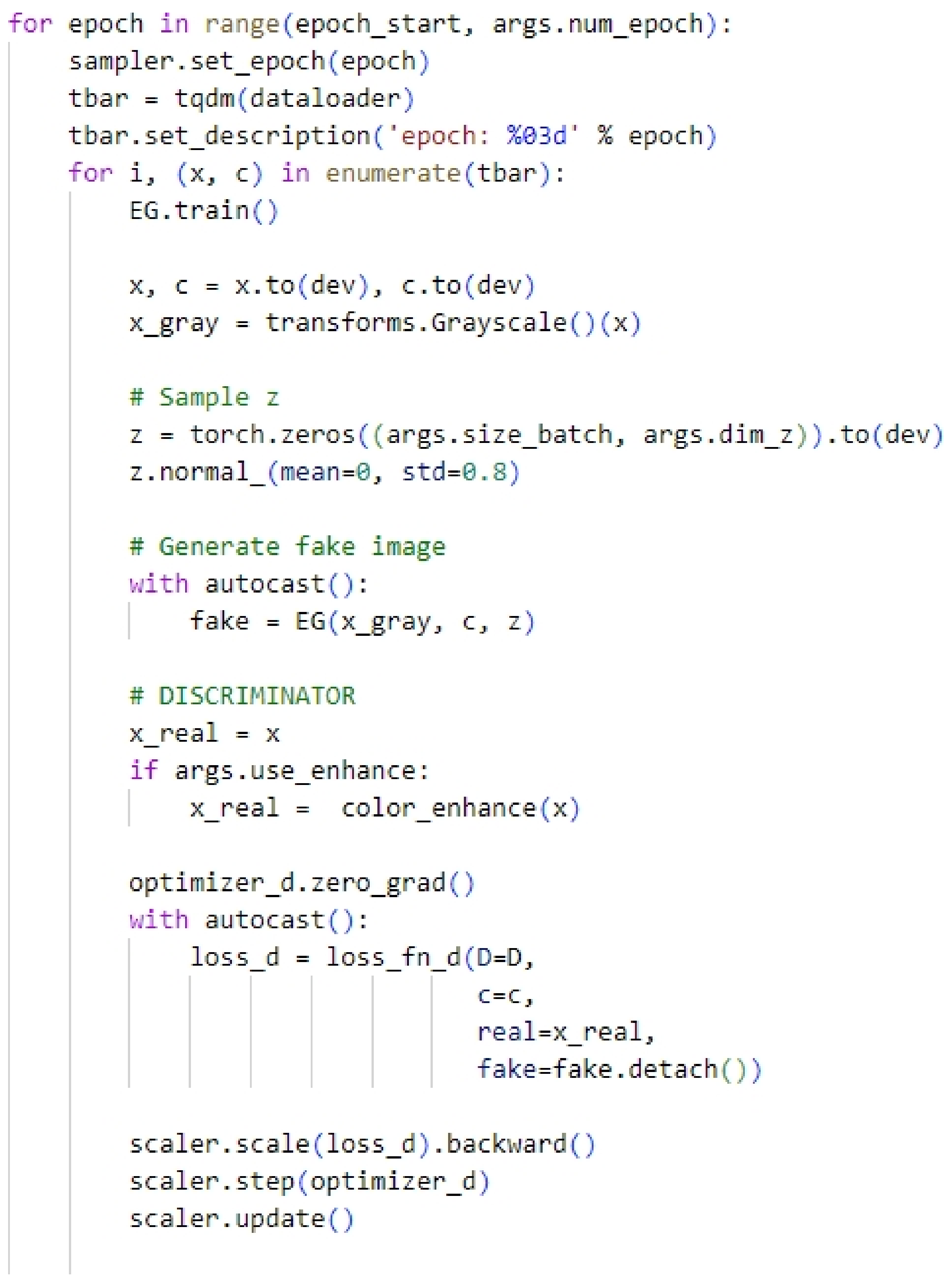
Finally, during Training Execution, once the environment and models are ready, the training loop starts processing the data efficiently and in parallel. Distributed samplers ensure that each model instance processes a unique section of the dataset, avoiding duplication and guaranteeing exhaustive coverage at each training epoch. This fine coordination ensures optimum use of available resources and steady progress towards model convergence.

In summary, this part organises and executes distributed training optimise computational performance and resource management. This enables large volumes of data to be processed efficiently with deep neural networks on parallel architectures, resulting in considerable time savings and maximum exploitation of available hardware capacity.

5) Drive loop

In the Training Loop, we iteratively process samples where greyscale images are transformed into colourised images by the generator. In parallel, we calculate the separate losses for the generator and the discriminator in order to evaluate their respective performance. This evaluation determines how well the generator is able to fool the discriminator with images that should appear realistic, while the discriminator learns to distinguish the generated images from those actually captured.

The code periodically saves the state of the model to enable training to be resumed in the event of interruption and to carry out evaluations on a validation set. These evaluations help us to measure the model's ability to generalise and to ensure that performance improvements are not limited solely to the data used training. (Figure 31)



```
for epoch in range(epoch_start, args.num_epoch):
    sampler.set_epoch(epoch)
    tbar = tqdm(dataloader)
    tbar.set_description('epoch: %03d' % epoch)
    for i, (x, c) in enumerate(tbar):
        EG.train()

        x, c = x.to(dev), c.to(dev)
        x_gray = transforms.Grayscale()(x)

        # Sample z
        z = torch.zeros((args.size_batch, args.dim_z)).to(dev)
        z.normal_(mean=0, std=0.8)

        # Generate fake image
        with autocast():
            fake = EG(x_gray, c, z)

        # DISCRIMINATOR
        x_real = x
        if args.use_enhance:
            x_real =  color_enhance(x)

        optimizer_d.zero_grad()
        with autocast():
            loss_d = loss_fn_d(D=D,
                               c=c,
                               real=x_real,
                               fake=fake.detach())

        scaler.scale(loss_d).backward()
        scaler.step(optimizer_d)
        scaler.update()
```

Figure 31

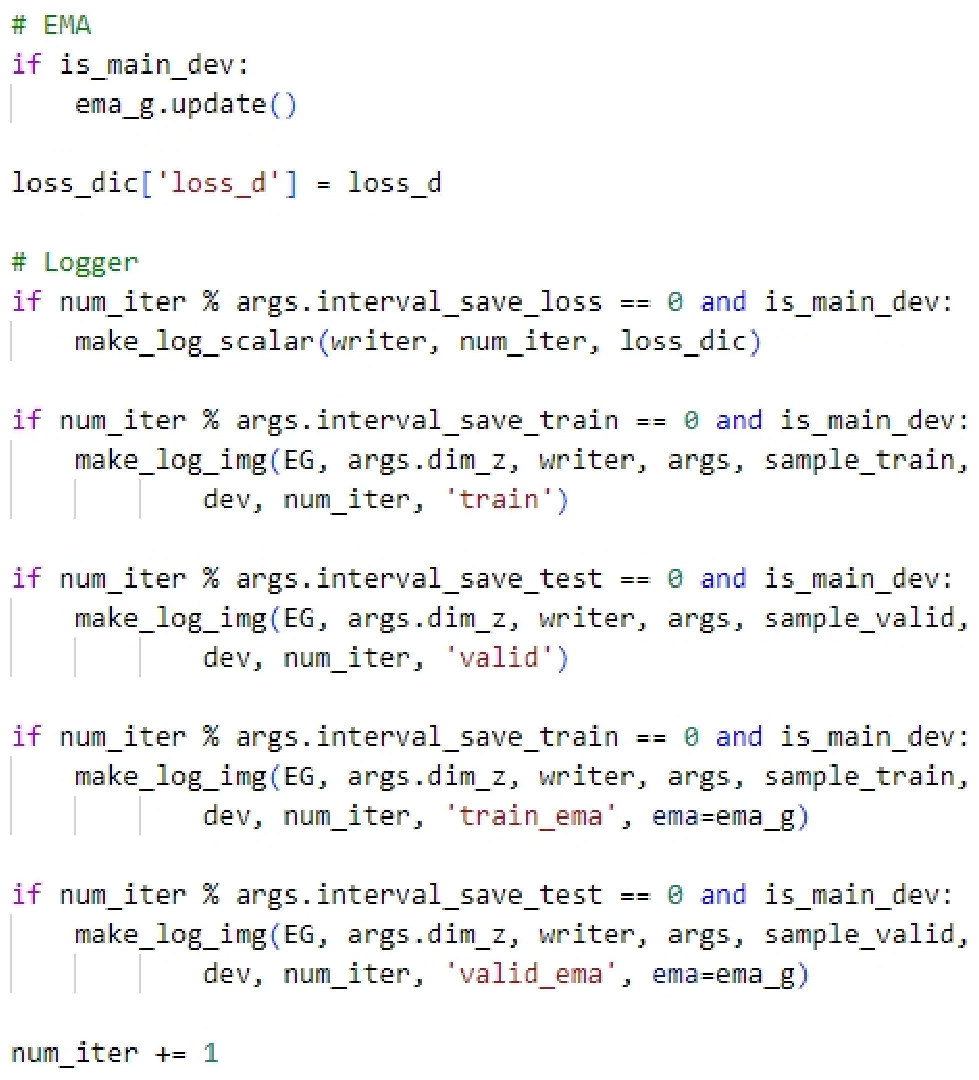
In the Training Loop, the process begins by iterating over the total number of defined epochs, with each epoch representing a complete pass over the training data set. The code uses a distributed sampler to ensure that each process handles a fair share of the data, maximising the use of all available GPU resources. Data is loaded in batches via a DataLoader at each epoch, and the network supports the supplied greyscale images for the generator to attempt to transform into colourised images.

Forward Propagation and Loss Calculation then come into play: the generator creates colourised images from the inputs, and the discriminator evaluates these images against the real images. The loss functions evaluate the errors of the generator and the discriminator, measuring the quality of the colourisation and the discriminator's ability to detect fakes. In Optimisation: Backpropagation and phase, the code calculates gradients from these losses and uses them to update the network weights via the pre-configured optimisers, improving the accuracy of the model predictions over time.

Monitoring and Backup play an essential role, with mechanisms in place to record training progress, periodic backups are made of generated images and loss metrics, and the code retains checkpoints at predefined intervals, making it easy resume training and evaluate saved models. Memory Management and Synchronisation are also critical, with memory clean-ups performed at each iteration to optimise the use of GPU resources, and synchronisations to ensure that all GPUs complete their tasks consistently before moving on to the next batch. This section effectively coordinates data management, parallel computations, and algorithmic optimisations, orchestrating efficient, high-quality learning for the colourisation model.

6) **Logging and monitoring**

In the Logging and Tracking section, the code systematically creates logs and uses TensorBoard to visualise training progress. This functionality includes the recording of important metrics such as losses and provides previews of the images generated. This allows us not only to monitor the progress and improvements of the model over time, but also to quickly identify any anomalies or areas for improvement in the training process. By providing an intuitive visual interface, TensorBoard helps users to gain an in-depth understanding of the behaviour and effectiveness of their model (Figure 32).



```python
# EMA
if is_main_dev:
    ema_g.update()

loss_dic['loss_d'] = loss_d

# Logger
if num_iter % args.interval_save_loss == 0 and is_main_dev:
    make_log_scalar(writer, num_iter, loss_dic)

if num_iter % args.interval_save_train == 0 and is_main_dev:
    make_log_img(EG, args.dim_z, writer, args, sample_train,
                dev, num_iter, 'train')

if num_iter % args.interval_save_test == 0 and is_main_dev:
    make_log_img(EG, args.dim_z, writer, args, sample_valid,
                dev, num_iter, 'valid')

if num_iter % args.interval_save_train == 0 and is_main_dev:
    make_log_img(EG, args.dim_z, writer, args, sample_train,
                dev, num_iter, 'train_ema', ema=ema_g)

if num_iter % args.interval_save_test == 0 and is_main_dev:
    make_log_img(EG, args.dim_z, writer, args, sample_valid,
                dev, num_iter, 'valid_ema', ema=ema_g)

num_iter += 1
```

Figure 32

This part of the train function focuses on logging and monitoring training, playing a crucial role assessing model performance and adjusting parameters if necessary.

In the Logging and Monitoring section, great emphasis is placed on the regular recording metrics training to assess model performance. This includes continuous monitoring of generator and discriminator losses, providing a detailed view of the evolution of the model and signalling the need for adjustments to optimise training. The use of TensorBoard facilitates data visualisation, allowing the tracking of various metrics such as loss curves and generated images. This gives users the ability to visually detect potential problems such as over-fitting or under-performance of the training.

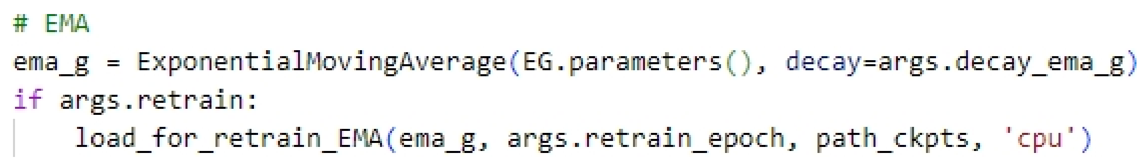
We regularly save samples of the images generated for a qualitative assessment of the model's performance. These images are used assess the quality of the colourisation and the relevance of the results compared with the original greyscale images. Control points, including generator and discriminator weights and optimizer states, are recorded at predefined intervals. This method is essential for resuming training after interruptions, replicating experiments, or deploying models in production.

The logging and monitoring system also plays a crucial role in dynamic management of training, enabling rapid adjustments to be made to parameters in response unexpected observations. For example, if the logs show a plateau in model improvement, we can adjust the learning rate or other hyperparameters to encourage continued progress. This flexibility is essential for maintaining and optimising the performance of the colourisation model.

In summary, this section is dedicated to the collection, analysis and visualisation of data generated during training. Effective monitoring is crucial for navigating the complexity of the deep model training process, providing the insights needed to improve performance and ensure the production of results.

7) **Exponential moving average (EMA)**

In the Exponential Moving Average (EMA) section, the code applies an exponential moving average technique to the generator weights improve their stability over time. This method is crucial for smoothing out variations in weights observed over several training iterations, which helps to mitigate the impact of abrupt fluctuations due to particularly atypical batches or noisy data. Incorporating the EMA makes the fitted weights less sensitive to these isolated variations, promoting a more predictable and robust convergence behaviour of the model. This approach is particularly useful in the advanced phases of training, where minor adjustments can have a significant impact on the overall performance of the model. In short, the use of the exponential moving average helps to stabilise and improve the consistency of the generator weights throughout the training process, thereby enhancing the reliability and efficiency of image colourisation model. (Figure 33)



```
# EMA
ema_g = ExponentialMovingAverage(EG.parameters(), decay=args.decay_ema_g)
if args.retrain:
    load_for_retrain_EMA(ema_g, args.retrain_epoch, path_ckpts, 'cpu')
```

Figure 33

The EMA part of the train function focuses on the use of the exponential moving average (EMA) to manage the generator weights in the training process. EMA is an essential technique for smoothing fluctuations in model weights over time, helping to stabilise training and reduce variance in the final results.

The main objective of the EMA is to maintain an average version of the generator weights that incorporates a proportion of the previous weights. This helps to stabilise the variations in weights observed from one epoch to the next, especially in the context of long and complex drives typical of deep networks. The implementation of EMA begins by initializing an ExponentialMovingAverage object with a specified decay rate. This rate determines how quickly historical weights are forgotten in favour of new weights.

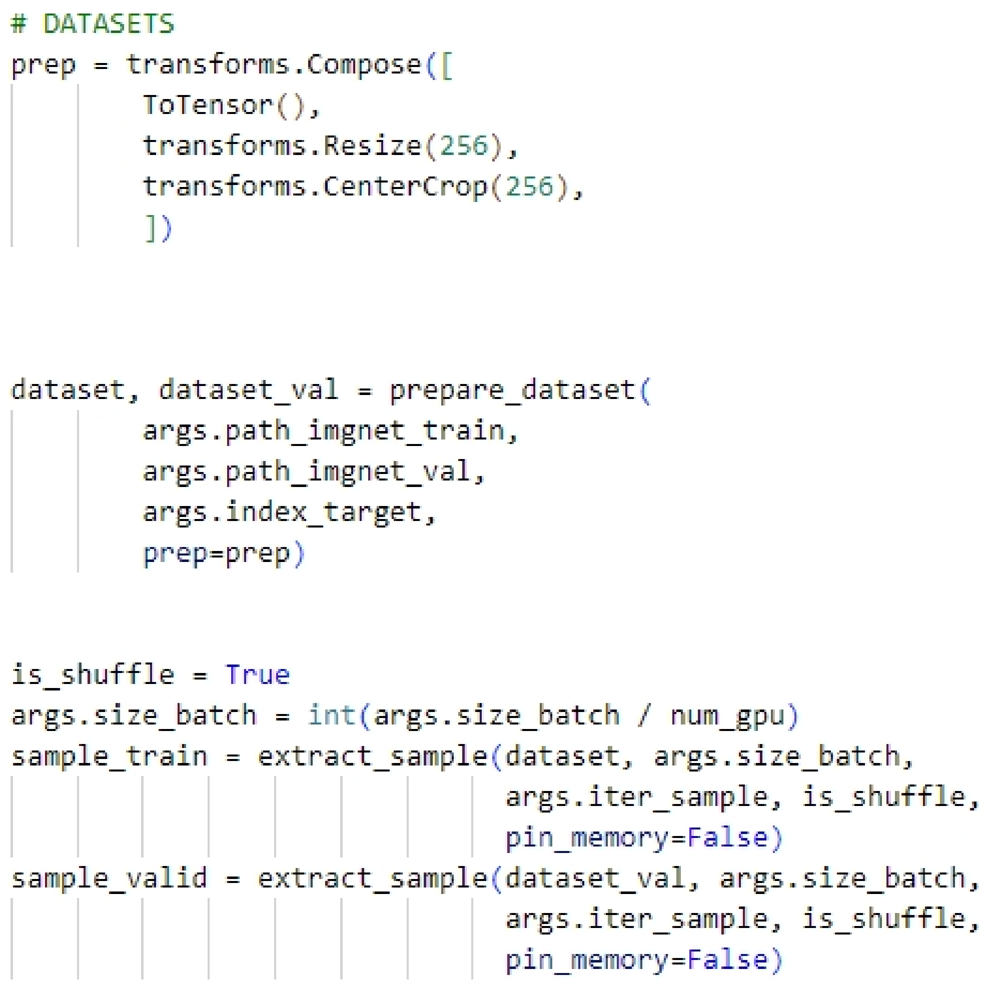
At each training iteration, after the generator weights have been updated by backpropagation, the EMA is updated to reflect the new weights. This ensures that the average weights used for predictions remain stable and consistent over time. The benefits of EMA are significant, as this method helps to achieve a more robust and powerful final model. Mean weights that are less sensitive to extreme fluctuations and sudden variations improve the generalisation of the model, making it more reliable in real-life scenarios.

Integrating EMA into the training loop ensures that even small but constant adjustments to model weights are taken into account in a way that promotes long-term stability and performance. This continuous approach is crucial for optimising the efficiency and usefulness of Deep Learning models trained on large datasets and over extended periods of time.

In summary, the EMA section of the train function enriches the generator's weight management by enabling smoother and more reliable convergence of the model throughout training. This method is essential for enhancing the efficiency Deep Learning models, particularly in contexts where stability and long-term performance are priorities.

8) **Image**

In the image section, image transformations are applied to prepare the data, including tensor conversion, resizing and cropping (Figure 34).



```
# DATASETS
prep = transforms.Compose([
        ToTensor(),
        transforms.Resize(256),
        transforms.CenterCrop(256),
        ])



dataset, dataset_val = prepare_dataset(
        args.path_imgnet_train,
        args.path_imgnet_val,
        args.index_target,
        prep=prep)


is_shuffle = True
args.size_batch = int(args.size_batch / num_gpu)
sample_train = extract_sample(dataset, args.size_batch,
                              args.iter_sample, is_shuffle,
                              pin_memory=False)
sample_valid = extract_sample(dataset_val, args.size_batch,
                              args.iter_sample, is_shuffle,
                              pin_memory=False)
```

Figure 34

This part of the train function focuses on image processing, a crucial step in preparing the data before it is used in the neural network training. This process ensures that the images are in an appropriate format and scale to be processed efficiently by the colourisation model.

In the Image Processing section, a number of transformations are applied to prepare the data. This often includes resizing, normalising and other specific operations needed to meet model requirements. To manage these transformations in a structured way, torchvision.transforms.Compose is generally used, which allows you to create a pipeline of transformations. This

The pipeline can include converting images into PyTorch tensors, resizing them, cropping them to centre, and possibly normalising pixel values, making them easier to manipulate by the network.

Data Preparation ensures essential consistency between the training and validation phases. The same transformations must be applied to training and validation data to essential consistency. Data is generally organised into distinct sets, with paths specified by the function arguments, ensuring that each image is treated identically before being fed into the model.

Efficient data loading is optimised using a PyTorch DataLoader. This component enables data to be loaded in parallel using multiple workers (num_workers), optimising memory use and speeding up the process. This is particularly advantageous in systems with multiple GPUs, reducing bottlenecks and maximising efficiency of data processing during training.

Finally, the Extraction of Validation Samples plays a crucial role in periodically evaluating the model. These samples, used for regular validations, enable the progress of the model to be monitored and the parameters to be adjusted according to the performance observed on data not seen during regular training. This ensures that the model remains efficient and adaptive, even in the face of new or unexpected situations.

In short, this part of the train function effectively orchestrates image pre-processing, ensuring that the colourisation model learns optimally. By standardising and optimising the data format upstream, this approach reduces the risks associated with inappropriate data formats or inefficient loading, while making it easier to fine-tune the model's performance on the processed data.

Conclusion Big Color

To conclude on the training of the Big Color model, its script uses a GAN network image colourisation, optimised to run efficiently on multiple GPUs. This advanced method combines a generator that produces colourised images and a discriminator that assesses their authenticity, optimising the visual quality of the results.

The flexibility of the script allows for precise adjustments thanks to configurable parameters and the use of pre-trained weights, which improves the convergence and fidelity of the images. The training process is supported by real-time monitoring and periodic back-ups via TensorBoard, facilitating ongoing evaluation and training management.

Despite its complexity and computational costs, this method is well suited to producing realistic colourisation results and is beneficial for research and practical applications.

It is nevertheless the most rigorous of our 4 models.

Training the Zhang model

Zhang's model training code deals with the preparation of image data training a convolutional neural network (CNN) model.

This code is divided into 6 main parts, including Data Separation, Data Reshape, Downsampling and Upsampling Layer Definition, Model Definition, Model Compilation and Training, and Model Evaluation and Saving.

```
train_gray_image=  gray_img[:38950]

train_color_image= color_img[:38950]


test_gray_image=  gray_img[38950:]

test_color_image= color_img[38950:]

print(len(train_gray_image))
```

In our study of the training code for Zhang's model, we first analysed the separation of the data into training and test sets.

The code begins by defining the variables `train_gray_image` and `train_color_image`, which contain the greyscale and colour images for the training respectively, selecting the first 38,950 images.
Next, the variables `test_gray_image` and `test_color_image` are defined to contain the remaining images for testing.

To divide the images, the code uses specific indices and then displays the size of the training set, confirming that it contains 38,950 images. This methodology ensures a clear and consistent separation of the data, which is essential for the validity of the training and model testing phases.

```
# reshaping
train_g = 
np.reshape(train_gray_image,(len(train_gray_image),SIZE,SIZE,3
))
train_c= np.reshape(train_color_image,
(len(train_color_image),SIZE,SIZE,3))
print('Train color image shape:',train_c.shape)



test_gray_image=
np.reshape(test_gray_image,(len(test_gray_image),SIZE,SIZE,3))
test_color_image= np.reshape(test_color_image,
(len(test_color_image),SIZE,SIZE,3))
print('Test color image shape',test_color_image.shape)
```

```
38950
Train color image shape: (38950, 160, 160, 3)
Test color image shape (50, 160, 160, 3)
```

We then looked at the second part, which involves resizing the images.

The code resizes the `train_g` and `train_c` variables to give them the shape `(number_of_frames, size, 3)`. This structure includes three colour channels, even for greyscale images, to maintain consistency with colour images.

After resizing, the code displays the new shapes of the training and test image sets. This step is essential to ensure that all images, whether in greyscale or colour, are processed consistently by the model.

```
from keras import layers

def down(filters , kernel_size, apply_batch_normalization=
True):

    downsample= tf.keras.models.Sequential()

    downsample.add(layers.Conv2D(filters,kernel_size,padding=
'same', strides=  2))

    if apply_batch_normalization:

        downsample.add(layers.BatchNormalization())

    downsample.add(keras.layers.LeakyReLU())

    return downsample



def up(filters, kernel_size, dropout=  False):

    upsample=  tf.keras.models.Sequential()

    upsample.add(layers.Conv2DTranspose(filters,
kernel_size,padding= 'same', strides= 2)) if

    dropout:

        upsample.dropout(0.2)

    upsample.add(keras.layers.LeakyReLU())

    return upsample
```

We looked at the third part, where two utility functions for the model are defined: `down` and `up`.

The `down` function creates a downsampling layer by applying a convolution, optionally followed by batch normalization and a

LeakyReLU activation. The `up` function creates upsampling layer using transposed convolution, optionally followed a dropout and LeakyReLU activation.

These two functions are crucial to the construction of the model, allowing resolution of the images to be reduced and increased in a controlled way while maintaining the important characteristics of the data thanks to the activations and normalisations applied.

```
def model():
    inputs= layers.Input(shape= [160,160,3])
    d1=  down(128,(3,3),False)(inputs)
    d2= down(128,(3,3),False)(d1)
    d3= down(256,(3,3),True)(d2)
    d4= down(512,(3,3),True)(d3)

    d5= down(512,(3,3),True)(d4)
    #upsampling
    u1= up(512,(3,3),False)(d5)
    u1= layers.concatenate([u1,d4])
    u2=  up(256,(3,3),False)(u1)
    u2= layers.concatenate([u2,d3])
    u3=  up(128,(3,3),False)(u2)
    u3= layers.concatenate([u3,d2])
    u4=  up(128,(3,3),False)(u3)
    u4= layers.concatenate([u4,d1])
    u5=  up(3,(3,3),False)(u4)
    u5= layers.concatenate([u5,inputs])
    output= layers.Conv2D(3,(2,2),strides= 1, padding= 'same')(u5)
    return tf.keras.Model(inputs= inputs, outputs= output)
   model= model()
```

```
model.summary()
```

In our study of Zhang's model training code, we analysed the fourth part, which concerns the definition of the neural network model.

The input layer, `layers.Input`, is configured to accept 160x160 images with 3 colour channels. The model includes several downsampling (d1 to d5) and upsampling (u1 to u5), where layers are concatenated to preserve information throughout the network. The `output` layer is a convolution designed to generate the coloured image.

The model is then compiled using the Adam optimizer with a learning rate of 0.001. The loss function used mean absolute , and precision is chosen as the evaluation metric. This structure enables the model to process and colour images efficiently and accurately.

```
model.compile(optimizer=
tf.keras.optimizers.Adam(learning_rate= 0.001), loss=
'mean_absolute_error',
              metrics= ['acc'])
```

```
model.fit(train_g, train_c, epochs= 50,batch_size= 60,verbose
= 0)
```

The fifth part focuses training the model.

The `model.fit` method is used to train the model on the training data. The training runs over 50 epochs with a batch size of
60. To keep the interface clean, drive details are not displayed using the `verbose=0` parameter.

This approach allows the model to learn from the training data over a specified number epochs, gradually adjusting its weights and parameters to error and improve image colourisation performance.

```
model.evaluate(test_gray_image,test_color_image)

model.save('my_model.h5') model.save('my_model.keras')
```

The sixth part focuses evaluating and saving the model.

First, the model is evaluated on the test data to measure its performance, using the appropriate metrics defined when the model was compiled.

Then, to ensure that the trained model is preserved, it is saved in two different formats: HDF5 with the `.h5` extension (named `my_model.h5`) and the native Keras format with the `.keras` extension (named `my_model.keras`). Please note that a warning specifies that the HDF5 format is considered obsolete, and it is recommended to use the native Keras format for the model backup in order guarantee future compatibility and backup stability.

This last step not only ensures a rigorous evaluation of the model's performance on independent data, but also a safe and durable backup of the trained model for future use or retraining.

Conclusion of the Zhang model

In conclusion, this code proposes a complete workflow for data preparation, definition, compilation, training, evaluation and saving a convolutional neural network model dedicated to image . It begins with the preparation of training and test sets, the resizing of data for compatibility with the model, and the use of utility functions for downsampling and upsampling. The model is built downsampling and upsampling layers, compiled with Adam as the optimizer, and then trained on the training data. After evaluation on the test data, the model saved in HDF5 and native Keras formats, the latter being recommended for future backup.

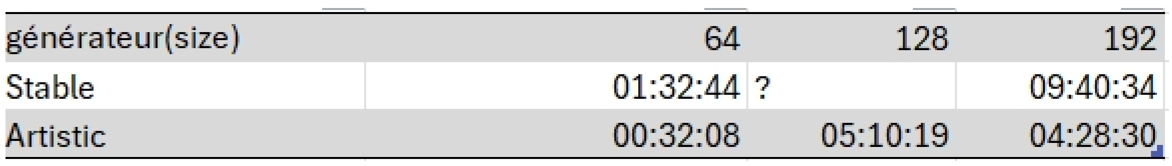
Practical comparison of models

We'll start by comparing the training time of each model. First, we'll analyse the Stable and Artistic models, because their processes are fundamentally similar, with the exception of certain parameters. However, training time varies significantly at certain stages of training.

Firstly, the pre-training of the generator for the two models shows significant differences at certain points, as shown in the table in Figure 1. The parameters for the part where the image size is 64 for the Stable and Artistic models are the same. The difference in training time can therefore be explained by the difference in scale factor, as shown in Figure 12. This is because the scale factor determines the number of filters in each convolutional layer, which increases the complexity of the generator and therefore the training time.

For an image size of 128 (sz=128), we were unable to run the training for the Stable model due to the limited capacity of our computer and the time available. However, we were able to run the training for the sz=128 part of the Artistic model, bearing in mind that Artistic uses a larger batch size than Stable (Figure 12). In theory, we could run the training for Stable if we didn't take the scaling factor into account.

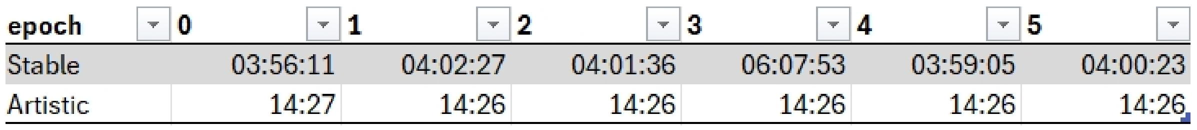
For an image size of 192 (sz=192), the batch size is also larger for Artistic (Figure 12), but Artistic takes almost half as long as Stable to complete its training. This clearly shows that the scaling factor plays an important role in the generator's training time.



générateur(size)	64	128	192
Stable	01:32:44	?	09:40:34
Artistic	00:32:08	05:10:19	04:28:30

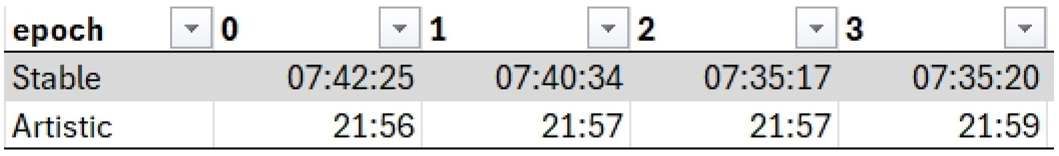
Figure 1: Table showing generator pre-drive times

During the pre-training of the discriminator, it appears that the scale factor also has a significant influence on training time, as shown Figure 2. The discriminator parameters are identical between the Artistic and Stable models, and we can clearly see a significant difference in the training time. We observe almost the same difference for the pre-training of the discriminator with a different batch-size and size (Figure 3).



epoch	0	1	2	3	4	5
Stable	03:56:11	04:02:27	04:01:36	06:07:53	03:59:05	04:00:23
Artistic	14:27	14:26	14:26	14:26	14:26	14:26

Figure 2: Table showing the time taken to pre-train the discriminator (batch-size=64 and sz=128)



epoch	0	1	2	3
Stable	07:42:25	07:40:34	07:35:17	07:35:20
Artistic	21:56	21:57	21:57	21:59

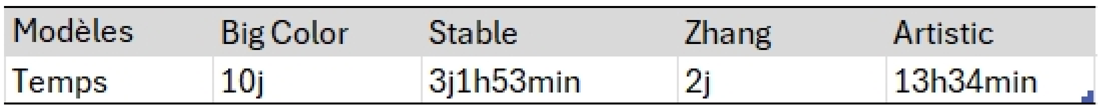
Figure 3: Table showing the time taken to pre-train the discriminator (batch-size=16 and sz=192)

The Gan training times for Stable and Artistic are roughly the same, at around one hour. So if we add up the total time for each of the two models, we have 3 days:1h:53min for Stable and only 13 h:34 min for Artistic.

For Big Color, we decided on 8 epochs, 8 for the size of the layers, a bach-size of 17 so that we could run our machine, and 8 epochs because the training for a single epoch lasts 30 hours, so that's about 10 days training.

The Cnn model was set to 60 epoch, 30 batch-size and 128 image size, and took around 2 days to train.

The table below (Figure 4) summarises the situation:



Modèles	Big Color	Stable	Zhang	Artistic
Temps	10j	3j1h53min	2j	13h34min

Figure 4: Total drive time for each model

We took a sample of 100 varied images and then converted them to greyscale so that the models could process them. To compare performance, we used the CIEDE 2000 formula, which calculates the difference between two colours (Figure 5). We chose this formula because it takes into account several factors such as luminosity, chroma and hue. We will apply this formula between the images produced by the models and the original images, then take an average. The closer DeltaE00 is to 0, the better the result.

$$\Delta E_{00}^{12} = \Delta E_{00}(L_1^*, a_1^*, b_1^*; L_2^*, a_2^*, b_2^*)$$

$$= \sqrt{\left(\frac{\Delta L'}{k_L S_L}\right)^2 + \left(\frac{\Delta C'}{k_C S_C}\right)^2 + \left(\frac{\Delta H'}{k_H S_H}\right)^2 + R_T \left(\frac{\Delta C'}{k_C S_C}\right)\left(\frac{\Delta H'}{k_H S_H}\right)}$$

ΔL': The difference in luminosity

ΔC**'**: The difference in chroma.

ΔH**'**: The difference in hue.

SL, SC, SH: Weighting functions for luminosity, chroma and hue.

RT: A rotation term to manage the interaction between chroma and hue differences.

Figure 5: General formula [18]

Let's start with the big colour model, which has the advantage of saving a model at each epoch. In the following table (Figure 6) we can see that the results do not improve at each epoch if we use the formula, but if we look directly at the images generated at each epoch the results improve with each epoch.

epoch	1	2	3	4	5	6	7	8
deltaE00	12,82	11,28	9,47	12,36	10,36	11,22	11,1	12,49

Figure 6: average delta value for 100 images at each big colour epoch

If we look at the overall performance of the models (Figure 7) we can see that Artistic is the best and Big-color-8(epoch) is last.Stable should have been the best in terms of performance but as I said earlier we had to skip part of the training.However the results of big-color-8 are much better than those of cnn if we look directly at the images generated.

modèles	artistic	stable	big-color-8	cnn
deltaE00	7,46	7,87	12,49	9,42

Figure 7: average delta value for 100 images for each model

In conclusion, in terms of time and performance Artistic and Stable are much better. You can see the images generated by each model[20] to have your own opinion.

The website

Backend

1.1 Introduction to the Backend

COLORIA project backend plays a crucial role in managing HTTP requests, processing images uploaded by users and communicating with the frontend. Built with Node.js and Express.js, it offers a lightweight, high-performance solution for server-side operations. Node.js server-side JavaScript execution, Express.js provides a flexible structure for robust authoring.

1.2 Backend Project Structure

The project backend is organised in such a way as to clearly separate the various responsibilities, thus facilitating maintenance and scalability. Here's an overview of the project's file tree:

1.3 Server configuration

The 'server.js' file configures the Node.js server to handle image upload, processing and requests.

Installation and start-up

- Installing dependencies: To install all the necessary dependencies in the root folder, use :
-`npm install`
-`npm install -g nodemon`
-`npm install multer`
-`npm install express`

- Starting the server: Use `nodemon server.js` to start the server, which enables it to be restarted automatically when changes are made.

Middleware used

- Request logging : Each request is logged with the time, IP address, request type and URL. This makes it easier to track activity and debug problems.
- CORS (Cross-Origin Resource Sharing): Allows server resources to be accessed from different domains, essential for separate frontend-backend applications.
- Serving static files: Static files (HTML, CSS, JS) are served from the `public` directory, and images downloaded from the directory `uploads`.

1.4 Image download management

The server uses Multer to manage uploaded files. Multer stores files in the `uploads` directory and names them uniquely to avoid conflicts. Files are named using a combination of the file field, timestamp and a unique identifier.

1.5 Routes and Endpoints

The server manages several main routes:

- /upload: This route allows images to be uploaded. Users can upload up to 5 images at a time. For the "All_Models" option, only one image is authorised.
- /download: This route allows processed images to be downloaded.
Users can download images via a direct link.
- /test: A simple test route to check that POST requests are received correctly.

1.6 File Management

Uploaded files are stored in the `uploads` directory. To save disk space, the original and processed files are automatically deleted after 10 minutes. This feature is implemented using JavaScript's `setTimeout` function, which schedules the deletion of files after the specified time.

1.7 Security and Validation

The server performs several validations to ensure security and integrity of the files:

- File validation: Checks that uploaded files are indeed images with the file header '/image' and limits the number of files to 5 per upload (or 1 for the "All_Models" option).
- Clear error messages: In the event of a problem, detailed error messages are returned to inform users and the server of the exact nature of the problem.

1.8 Optimisation and best practice

The server is optimised for performance using a number of techniques and best practices:

- Use of middleware: Middleware is used for logging, error management and CORS, which improves the modularity and maintainability of the code.
- Asynchronous script execution: External scripts (such as the image processing Python script) are executed asynchronously to avoid blocking the server.
- Centralised error management: Errors are managed centrally to provide consistent and informative error .
- Automatic file clean-up: downloaded and processed files are automatically deleted after a certain period to save disk space.

Conclusion of I

The COLORIA project backend is designed to be high-performance, secure and maintainable. It efficiently manages downloads and image processing, provides clear endpoints for the main operations and follows best practice in backend development. These features ensure a fluid and reliable user experience, while facilitating the maintenance and scalability of the project.

Frontend

2.1 Introduction to Frontend

The frontend of the COLORIA project is responsible for the user interface. It allows users to upload images, select treatment models and view results in an intuitive and interactive way. It also provides information on how to use the site and how to contact team. The frontend uses mainly HTML, CSS and JavaScript to provide a fluid and pleasant user experience.

2.2 Frontend Project Structure

The frontend is organised to provide clear navigation and efficient file management. Here is the structure of the frontend files and directories:

2.3 HTML pages

site.html

site.html' is the main page where users can upload images, select a processing model and view a before/after comparison of images.
Features :
- Drop frame: Allows users to drag and drop images or select them using a button.
- Template selection: Allows users to choose the processing template to be applied to images.
- Send button: Sends the selected images to the server for processing.
- Comparison slider: Allows you to visually compare an image before and after processing.
- Logo and favicon: a logo and favicon file have been set up to ensure the site's integrity across all browsers.

result.html
`result.html` displays the processed images and allows them to be downloaded. Features :
- Image list: Displays processed images with download buttons for each image.
- Back button : Returns you to the home page.

all_models_result.html
`all_models_result.html` displays the images processed by all the available models.
Features :
- Image list: Displays processed images with download buttons for each image, sorted by model.
- Back button : Returns you to the home page.

about_us.html
about_us.html` provides contact information for the COLORIA service.
Features :
- E-mail address: Provides a contact e-mail for the COLORIA team.

operation.html
How it works.html` explains how the COLORIA service works. Features :
- Instructions: Describes how to use the service to download and process images.

modelX.html

`modelX.html` 4 pages explaining how the different models work. Features :

- Model descriptions: Provides information on the different processing models available (Stable/StableX, Artistic/ArtistiX, BigColor/BigColorX, CNN).

-The models with an X are the pre-trained models, the others have been trained by us.

2.4 Design and Styles

The `styles.css` file contains the styles for all the HTML pages, with a simple, intuitive design. The colours match the logo, and Media Queries are used to make the site responsive, ensuring a pleasant user experience.

Main style elements :

- Buttons: Styles for the download, delete and back buttons.
- Image containers: Styles image containers and the images themselves.
- Error : Styles for error messages displayed to users.
- Comparison slider: Styles for the slider used to compare an image before and after processing.

2.5 JavaScript scripts

scripts.js

scripts.js manages the uploading and display of images on `site.html`, as well as the management of the slider to compare an image before and after processing. Features :

- Image download management: Adds event listeners to manage the download and display of images and resizes it to 1024x1024px to optimise the transfer.
- Updated slider: Allows you to visually compare images before and after processing.
- Error handling: Displays error messages if there are any problems with downloading or processing images.

result.js

`result.js` manages the display of processed images on `result.html` and their download.
Features :
- Display processed images: Retrieves the paths of processed images and displays them on the page.
- Image download: Allows processed images to be downloaded via download buttons.

all_models_result.js
`all_models_result.js` manages the display and downloading of images processed by all models on `all_models_result.html`.
Features :
- Display images processed by all templates: Retrieves the paths of images processed by all templates and of the original image, then displays them on the page with the template name.
- Image download: Allows processed images to be downloaded via download buttons.

2.6 User Experience (UX)

The user interface is designed to be simple and intuitive, with interactive elements such as buttons and a slider to enhance user experience.
Key UX considerations :
- Intuitive navigation: users can easily navigate between the different pages of the site via a navigation menu.
- Clear error messages: Error messages are clearly displayed in red to inform users of any problems.
- Responsive interface: The site is designed to be responsive, offering a fluid user experience and adapting to different devices and screen sizes.

2.7 Frontend security

The frontend includes validations for user input and protection against XSS attacks.
Safety measures :
- File validation: Checks that uploaded files are indeed images before sending them to the server with the file header '/image' and a valid extension list.

- Protection against XSS attacks: Cleans user input to limit XSS attacks.
- Error handling: Displays clear, informative error in the event of a problem.

2.8 Optimisation and best practice

The JavaScript code is optimised for performance, with efficient DOM manipulation and event management.
Main improvements :
- Efficient DOM manipulation: Use of modern techniques to manipulate the DOM efficiently.
- Event management: Add event listeners in an optimal way to avoid memory leaks and improve performance.
- Clean, maintainable code: Follow front-end development best practice to ensure clean, maintainable code.

The front end of the COLORIA project is designed to offer a pleasant and intuitive user experience. It allows users to upload images, select treatment models and view results interactively. The code is optimised for performance and security, and follows best practice in frontend development.

Conclusion Website

The COLORIA project offers an innovative online image colourisation service, enabling users to transform their greyscale images into high-quality colourised versions. This report has detailed the technical aspects of the site and server, focusing specifically on the backend and frontend.

Backend
The project's backend, developed using Node.js and Express.js, is responsible for managing requests, processing uploaded images and communicating with the frontend. Thanks to the use of middleware for logging, CORS management and file processing, the backend ensures efficient and secure management of operations. File validation, error management performance optimisation guarantee a smooth, reliable user experience.

Frontend

The frontend, built with HTML, CSS, and JavaScript, offers an intuitive and interactive user interface. Users can easily upload images, select treatment models, and visualise results using interactive elements such as buttons and a slider. Responsive design and clear error messages contribute to a pleasant user experience. Optimised JavaScript scripts ensure efficient DOM handling and high-performance event management.

Final thoughts

The COLORIA project, with its well-defined architecture and robust functionality, demonstrates an effective approach to offering online image processing services. The high-performance, secure backend, combined with an intuitive, responsive frontend, ensures a high-quality user experience. By following best development practices and using modern technologies, COLORIA is well positioned to evolve and adapt to future needs.

Suggestions for improvement :

To take further, a number of improvements can be envisaged:
- Add new features: Integrate additional features, such as image filters or advanced customisation options.
- Performance optimisation: Continue to optimise the code to improve processing speed and reduce loading times.
- Enhanced security: Implement additional security measures to protect user data and prevent potential attacks.
- User experience: carrying out user tests to identify areas for improvement and make the interface even more intuitive.

In conclusion, the COLORIA project illustrates a successful combination of backend and frontend technologies to deliver an innovative, high-performance online service. The solid foundations laid by this project make it possible envisage numerous extensions and improvements, thus ensuring future sustainability and success.

Conclusion

To conclude this report, the Coloria project has enabled us to tackle the technical challenges in the field of image colouring based artificial intelligence. Our work revolved around research, analysis and implementation of different Deep Learning models for colourising black and images. We were able to train them, compare them and integrate them into a website accessible to everyone.

In the course of our research, we first defined the key concepts of artificial intelligence and Deep Learning, which are fundamental to understanding the mechanisms underlying image colouring. We then analysed existing solutions, reviewing architectures such as U-NET and ResNet-34/101. This exploration led us to select and experiment with several cutting-edge models, namely DeOldify, KIMGEONUNG (BigColor), and Rich ZHANG's model.

Training these models revealed specific advantages and disadvantages for each, enabling us identify strengths and weaknesses in terms of performance and the quality of the results produced. Our theoretical comparison was complemented by a practical evaluation, highlighting the differences in training approaches and the impact on colourisation quality.

The project culminated   creation a website. This web integration not only provides a concrete demonstration of the models' capabilities, but also makes them accessible to a wider audience.

In conclusion, this project has been an enriching and formative experience. It has enabled us to combine our theoretical knowledge with a practical application in the field of artificial intelligence. The knowledge gained throughout this process will undoubtedly be invaluable for our future careers as engineers, preparing us to take on similar technical challenges and actively contribute to the evolution of AI-based technologies.

IEEE

Front page image: generated by chatgpt 4

[1]J. Zhang, "UNet Line by Line Explanation," Medium, Oct. 18, 2019. https://towardsdatascience.com/unet-line-by-line-explanation-9b191c76baf5

[2]P. Ruiz, "Understanding and visualizing ResNets," Medium, Oct. 08, 2018. https://towardsdatascience.com/understanding-and-visualizing-resnets-442284831be8

[3]G. Kim et al, "BigColor: Colorization using a Generative Color Prior for Natural Images." Accessed: Jun. 11, 2024. [Online]. Available: https://kimgeonung.github.io/assets/bigcolor/bigcolor_main.pdf

[4]"ImageNet Object Localization Challenge," kaggle.com. https://www.kaggle.com/competitions/imagenet-object-localization-challenge/data

[5]J. Antic, "jantic/DeOldify," GitHub, Mar. 10, 2020.

https://github.com/jantic/DeOldify

[6]R. Zhang, "richzhang/colorization," GitHub, May 29, 2024. https://github.com/richzhang/colorization

[7] Comar, "KIMGEONUNG/BigColor," GitHub, Jun. 11, 2024. https://github.com/KIMGEONUNG/BigColor (accessed Jun. 11, 2024).

[8] "Colorization using Optimization", Anat Levin, Dani Lischinski, Yair Weiss. ACM Transactions on Graphics, 2002.

[9] "Fast Colorization Using Edge and Gradient Constrains" , Yao Li, Lizhuang Ma, Wu Di, 2007

[10] "Variational Exemplar-Based Image Colorization", Aurélie Bugeau, Vinh Thong Ta, Nicolas Papadakis. IEEE Transactions on Image Processing, 2014.

[11] "Deep Colorization", Zezhou Cheng, Qingxiong Yang, Bin Sheng. International Conference on Computer Vision, 2015.

[12] "Learning Representations for Automatic Colorization", Gustav Larsson, Michael Maire, Gregory Shakhnarovich. 2016.

[13] "Deep Exemplar-based Colorization", Mingming He, Dongdong Chen, Jing Liao, Pedro V. Sander, Lu Yuan. ACM Transactions on Graphics (Proc. SIGGRAPH), 2018.

[14] R. Zhang, "richzhang/colorization," GitHub, May 29, 2024.

[15] Raj Kumar Gupta, Alex Yong-Sang Chia, Deepu Rajan, Ee Sin Ng, and Huang Zhiyong. "Image colorization using similar images",2012.

[16] Qing Luan, Fang Wen, Daniel Cohen-Or, Lin Liang, YingQing Xu, and Heung-Yeung Shum. "Natural image colorization", 2007.

[17] Shuang Ma, Jianlong Fu, Chang Wen Chen, and Tao Mei. Da-gan: "Instance-level image translation by deep attention generative adversarial networks" , 2018.

[18] The CIEDE2000 Color-Difference Formula:

"https://hajim.rochester.edu/ece/sites/gsharma/ciede2000/ciede2000noteCRNA.pdf"

[19] python formula function: https://github.com/lovro-i/CIEDE2000

[20] images generated by models:

"https://drive.google.com/drive/folders/13RwD5zHVGzV7nh0raskqAVlTqDJ_S67K"