
Colorisation d'images
Rapport de projet E3



Par : Elliot CAMBIER, Kérien HARTWEG, Hugo KOTHE,
Olivier WANG, Caull YANG
Tutrice : Imen KACHOURI

Table des matières

Remerciement.....	3
Contexte.....	4
Introduction générale	4
Définition de l'intelligence artificielle et du Deep Learning.....	4
Colorisation d'image.....	5
Analyse de l'existant.....	5
Architectures avancées	7
Architecture U-NET	7
Architectures ResNet-34 et ResNet-101.....	8
Les modèles.....	G
Introduction des modèles.....	9
DeOldify « artistic » et « stable »	9
KIMGEONUNG (BigColor)	10
Rich ZHANG	10
Type d'entraînement, Avantages et Inconvénients	11
Comparaison théorique des modèles	14
Entraînement stable et artistic	14
Entraînement BigColor	15
Entraînement du model Zhang	36
Comparaison pratique des modèles.....	41
Le site Web	45
Backend	45
Frontend	48
Conclusion Site Web	52
Conclusion	54
IEEE	55

Remerciement

Toute l'équipe de COLORIA tient à remercier :

Imen KACHOURI

Eric LLORENS

Jour des Projets 2024

ESIEE Paris

Contexte

Introduction générale

L'intelligence artificielle (IA), et plus particulièrement les techniques de Deep Learning, ont révolutionné de nombreux domaines de la recherche et de l'industrie grâce à leur capacité à modéliser des tâches complexes sans intervention humaine directe. Notre projet de E3 s'inscrit dans ce cadre dynamique, avec l'ambition d'explorer les différentes techniques liées à la colorisation d'images. Cette technique trouve de nombreuses applications, notamment dans la restauration d'images anciennes. Ce projet aboutira à la conception d'une interface homme-machine (IHM), un site web qui regroupe diverses méthodes de colorisation pour permettre aux utilisateurs de choisir la technique la plus fidèle à leurs besoins.

Ce rapport explore initialement l'état actuel des technologies de colorisation, offrant une base compréhensible même pour les non-initiés. Nous présentons ensuite quatre méthodes spécifiques de colorisation : leur fonctionnement, leurs avantages et leurs limites sont détaillés et comparés. Chaque méthode est évaluée non seulement théoriquement mais aussi à travers une implémentation pratique au sein de notre site web développé pour ce projet, lequel répond également à des problématiques d'accessibilité.

La structure du rapport est la suivante : après une introduction qui contextualise notre étude, nous présentons un état de l'art des techniques de deep learning applicables à la colorisation. Nous détaillons ensuite les technologies et architectures spécifiques utilisées, telles que les GANs, les CNNs, et des architectures avancées comme U-NET et ResNet. La section suivante est consacrée à l'exposition de nos modèles, la description des datasets utilisés, ainsi que les résultats comparatifs théoriques et pratiques. Le site web développé est présenté en détail avant de conclure sur l'analyse des résultats ainsi que sur les perspectives de notre travail à l'avenir.

Définition de l'intelligence artificielle et du Deep Learning

Pour débuter, définissons le Deep Learning, grâce à la définition de la CNIL : « L'apprentissage profond est un procédé d'apprentissage automatique utilisant des réseaux de neurones possédant plusieurs couches de neurones cachées. Ces algorithmes possédant de très nombreux paramètres, ils demandent un nombre très important de données afin d'être entraînés. »

Dans l'apprentissage profond il y a deux grandes catégories, l'apprentissage supervisé et l'apprentissage non supervisé, On peut les opposer ainsi :

Apprentissage supervisé

Données = observations + étiquettes

Connaissance -> relation entrée-sortie

Apprentissage non-supervisé

Données = observations

Connaissance -> structures latentes

L'apprentissage supervisé consiste à utiliser des données étiquetées afin d'entraîner un modèle, où le système acquiert la capacité d'associer des entrées spécifiques à des sorties souhaitées.

Contrairement à cela, l'apprentissage non supervisé ne se base pas sur des données étiquetées et cherche à saisir la structure intrinsèque des données en repérant des patterns ou des clusters sans avoir recours à des données de sortie connues.

Colorisation d'image

La colorisation par Deep Learning est une méthode permettant de coloriser des images en nuances de gris, grâce à des réseaux neuronaux. Il s'agit d'une méthode populaire dans le domaine de la vision par ordinateur, en effet elle permet de donner vie à des images/vidéos anciennes qui sont dépossédées de toute couleur.

Le déroulement commun à toutes les méthodes est le suivant :

- Entrainement du modèle : dans cette première étape, un modèle de réseau de neurones est entraîné sur une base de données d'images en nuances de gris et en couleur. Le but est de faire apprendre au modèle à prédire les couleurs des pixels en se basant sur les couleurs des pixels environnants et d'autres caractéristiques de l'image, telles que les contours et les textures.
- Colorisation de l'image : Nous pouvons colorer grâce à ce modèle entraîné des images en nuances de gris en prédisant les couleurs des pixels manquants. Pour cela, le modèle prend en entrée l'image en nuances de gris et produit une image en couleur correspondante.

Analyse de l'existant

Il existe de nombreuses méthodes de colorisation, plus ou moins anciennes. La colorisation manuelle est l'une des plus anciennes et nécessite des compétences artistiques pour appliquer les couleurs de manière précise et réaliste. Les utilisateurs peuvent dessiner directement des traits de couleur sur l'image, qui seront ensuite propagés en utilisant les contours de l'image pour définir la diffusion des couleurs [8, 16]. Cette méthode, également appelée

gribouillage pour la colorisation d'images en niveaux de gris, s'avère complexe en raison de son exigence de précision dans les dessins. Cependant, d'autres chercheurs ont amélioré cette méthode en la rendant plus accessible grâce à une palette de couleurs et un pinceau, tout en réduisant le nombre de gribouillages nécessaires et les erreurs potentielles [9]. Une variante de cette approche permet à l'utilisateur de fournir une image en couleur comme référence, facilitant ainsi la tâche de colorisation en copiant les couleurs des zones correspondantes [10,15]. On peut parler de transfert de couleurs : l'algorithme convertit l'image de référence en niveaux de gris, et compare cette image avec celle à coloriser. Les « objets » / « morceaux » de l'image ayant la même luminosité ou nuance de gris seront colorisés selon la couleur associée dans l'image de référence.

Avec l'avancement de la technologie, des méthodes de colorisation plus sophistiquées ont été développées, notamment celles utilisant des réseaux de neurones convolutifs (CNN). Ces modèles, qui sont des réseaux profonds typiquement utilisés pour l'analyse d'images, apprennent à prédire les couleurs à partir d'un ensemble de données contenant des images en noir et blanc et leurs correspondances en couleur [11,14]. L'architecture des CNN leur permet de capturer des caractéristiques à différentes échelles et de comprendre la structure globale ainsi que les détails locaux des images, ce qui est crucial pour la tâche de colorisation.

Une architecture intéressante dans ce domaine est la "Classification via Retrieval" (CvR), qui combine l'apprentissage des couleurs avec l'extraction de représentations sémantiques des images. Cette méthode ne se contente pas de prédire les couleurs appropriées mais cherche également à comprendre le contexte et le contenu des images pour améliorer la précision des couleurs prédites [12].

La méthode de colorisation [13] utilise un modèle de réseau de neurones basé sur des « transformeurs », entraîné avec des images en niveaux de gris et leurs équivalents en couleur. Ce modèle apprend à interpréter le contenu sémantique des images en niveaux de gris et à établir une correspondance entre les pixels gris et leurs couleurs. Il traite les images en niveaux de gris à travers des couches successives de « transformateurs », générant une carte de probabilité pour la couleur de chaque pixel. Des mécanismes d'attention permettent de se concentrer sur les zones clés de l'image afin d'optimiser la prédiction des couleurs.

La colorisation d'images représente un défi majeur en raison de sa nature complexe : un même objet peut être associé à différentes couleurs plausibles. Les techniques de Deep Learning, bien qu'efficaces, exigent d'importantes quantités de données annotées et peinent parfois avec des images complexes contenant de multiples objets ou un arrière-plan chargé. La détection et la

classification précises des objets sont cruciales pour une colorisation réaliste. Il est également essentiel de gérer correctement plusieurs instances qui peuvent se chevaucher, en assurant une distinction nette entre les objets et l'arrière-plan, ce qui facilite la synthèse et la manipulation des caractéristiques visuelles [17].

Architectures avancées

Réseaux de Neurones Convolutionnels (CNN) : Spécialement conçus pour traiter des données structurées comme des images, les CNNs utilisent des filtres pour capturer les relations spatiales et les caractéristiques à diverses échelles.

Réseaux Adversaires Génératifs (GANs) : Composés de deux réseaux en compétition, un générateur et un discriminateur, les GANs [sont utilisés pour créer de nouvelles données semblables aux données réelles, offrant des avancées notables dans la génération d'images et la colorisation.

Techniques de Gribouillage : Permettent aux utilisateurs de guider le processus de colorisation en dessinant des indices de couleur sur une image en nuances de gris, que le modèle utilise ensuite pour coloriser le reste de l'image.

Colorisation Basée sur des Exemples : Utilise une image en couleur comme référence pour transférer le schéma de couleur à une image en nuance de gris correspondante.

Architecture U-NET

L'architecture U-NET est largement adoptée dans le domaine de la vision par ordinateur pour des tâches telles que la segmentation d'images. Conçue initialement en 2015 pour la segmentation des images médicales, U-NET se caractérise par sa structure en forme de "U", comprenant deux parties principales : l'encodeur (chemin de contraction) et le décodeur (chemin d'expansion).

Encodeur : L'encodeur est composé de multiples couches de convolution et de max pooling. Cette configuration permet de réduire progressivement la dimensionnalité de l'entrée tout en extrayant et en compactant les caractéristiques importantes de l'image. Cette réduction est essentielle pour capturer le contexte global de l'image sans surcharger le modèle en termes de calculs et de mémoire.

Décodeur : Le décodeur utilise des convolutions transposées pour reconstruire l'image à partir des caractéristiques codées par l'encodeur. Cette partie du réseau augmente la résolution de l'image étape par étape et fusionne les caractéristiques de haute résolution à partir de l'encodeur grâce aux connexions

de saut (skip connections). Ces connexions sont cruciales pour restaurer les détails locaux et la structure de l'image originale.

U-NET est particulièrement efficace même avec des ensembles de données limités, car il peut généraliser à partir de peu d'exemples grâce à son mécanisme efficace de transfert de caractéristiques entre l'encodeur et le décodeur. Cela le rend idéal pour les applications où les données annotées sont rares ou coûteuses à obtenir. [1]

Architectures ResNet-34 et ResNet-101

ResNet-34 et ResNet-101 sont deux types de ResNet, un modèle spécial de réseau de neurones qui utilise des "connexions résiduelles". Ces connexions aident à éviter le problème des gradients qui disparaissent quand le réseau est très profond.[2]

ResNet-34 a 34 couches et est bon pour des tâches simples ou quand on n'a pas beaucoup de puissance de calcul. ResNet-101 a 101 couches et peut comprendre des détails plus complexes, ce qui est utile pour des tâches qui demandent beaucoup de précision.

En résumé, si la tâche est simple ou si on a des limites de puissance, ResNet-34 est souvent suffisant. Pour des tâches plus détaillées, ResNet-101 est préférable. L'idée derrière ResNet est de rendre les réseaux de neurones plus profonds sans perdre en performance. Cela est possible grâce aux connexions qui permettent aux signaux de sauter certaines couches, facilitant ainsi le passage de l'information.

En utilisant des modèles comme ResNet avec U-NET pour colorer des images, ces systèmes apprennent non seulement à ajouter des couleurs de manière réaliste mais aussi à respecter la forme originale des images. Ces méthodes, qu'elles soient supervisées ou non, utilisent ces techniques pour obtenir des résultats très réalistes et efficaces, selon l'usage qu'on veut en faire.

Les modèles

Introduction des modèles

Pour notre projet, nous avons exploré quatre modèles de pointe pour la colorisation d'images, chacun se distinguant par son approche et son architecture sous-jacente. Ces modèles sont basés sur le principe de l'architecture U-NET, bien connue pour son efficacité dans les tâches de vision par ordinateur telles que la segmentation d'images.

Deux des modèles, nommés "Artistic" et "Stable", sont développés autour de l'approche non supervisée avec des architectures de réseau antagoniste génératif (GAN). Un troisième modèle nommé "BigColor" est développé autour du GAN également mais pas selon l'architecture U-NET.

Le dernier modèle est CNN simple basé sur U-net créé par Rich Zhang.

Mais alors pourquoi avons-nous choisi ces modèles parmi l'énorme vastitude existante ?

DeOldify « artistic » et « stable »

Les modèles "Artistic" et "Stable" de DeOldify ont été sélectionnés en raison de leur capacité à offrir une flexibilité et une efficacité optimales grâce à leur fondement sur des technologies de pointe. Ces modèles utilisent une approche non supervisée avec des architectures de réseaux antagonistes génératifs (GAN) et sont basés sur l'architecture U-NET, reconnue pour sa précision dans les tâches de vision par ordinateur telles que la segmentation d'images. Ce choix permet donc de générer des images colorisées de haute qualité sans nécessiter de vastes jeux de données étiquetées, ce qui réduit considérablement les coûts et complexités associées au processus de formation des modèles.

La disponibilité des deux variantes, "Artistic" et "Stable", enrichit d'autant plus le projet, permettant de s'adapter à différentes exigences esthétiques et pratiques. Tandis que "Artistic" favorise une approche plus libre et expressivement colorée, idéale pour les rendus où l'aspect artistique est privilégié, "Stable" propose une restitution des couleurs plus précise et naturelle, adaptée aux besoins de fidélité visuelle élevée. Ces caractéristiques, combinées à la performance éprouvée de ces modèles largement utilisés et appréciés dans la communauté, justifient pleinement leur choix pour un projet exigeant un haut niveau de qualité et de réalisme dans la colorisation d'images, ils sont également assez populaires puisque nous trouvons de nombreux articles et rapport sur eux facilitant d'autant plus notre compréhension de ces modèles.

KIMGEONUNG (BigColor)

Le modèle Big Color se distingue par sa capacité à utiliser efficacement les architectures de deep learning pour traiter des images à grande échelle. Big Color emploie un mécanisme basé sur les réseaux convolutionnels profonds (CNN) et des méthodes d'apprentissage profond, optimisées pour traiter des images de haute résolution sans compromettre les détails fins ni la saturation des couleurs. Cette approche est particulièrement avantageuse pour la colorisation d'images, car elle permet de traiter de grandes dimensions tout en préservant la qualité et la précision des nuances de couleur.

La force de Big Color réside dans sa capacité à intégrer des avancées récentes en intelligence artificielle, incluant l'utilisation de l'auto-apprentissage et des techniques de raffinement des couleurs pour améliorer la précision des résultats finaux. Cette technologie est choisie non seulement pour sa haute performance en termes de fidélité des couleurs et de résolution des images traitées, mais également pour sa flexibilité dans le traitement d'un large éventail de types d'images et de conditions d'éclairage. En incorporant ces innovations, Big Color se positionne comme un outil robuste et polyvalent, idéal pour des applications nécessitant une restitution visuelle de haute qualité et des transformations colorimétriques complexes.

Rich ZHANG

Le modèle développé par Rich Zhang, connu sous le nom de "Colorful Image Colorization", utilise une approche innovante pour la colorisation d'images en noir et blanc, basée sur des réseaux de neurones convolutifs (CNN). Ce modèle se distingue par l'intégration de la classification des classes d'objets dans le processus de colorisation, utilisant des prédictions basées sur des millions d'images colorées de référence pour guider la distribution des couleurs. Cela lui permet de coloriser de manière sélective et pertinente selon les caractéristiques détectées dans l'image, améliorant ainsi le réalisme et la précision des résultats finaux.

La méthode de Zhang est particulièrement remarquable pour sa capacité à traiter automatiquement des images complexes avec un haut degré de détail et de nuance. En outre, elle offre une interface utilisateur simple qui permet aux utilisateurs de participer activement au processus de colorisation, en suggérant des couleurs pour des régions spécifiques, ce qui peut ensuite être ajusté et perfectionné par le modèle. Cette caractéristique rend le modèle très accessible et adaptable, adapté tant pour les applications professionnelles que pour les projets personnels où la créativité et le contrôle de l'utilisateur sont valorisés. Le modèle de Zhang est ainsi un choix efficace pour ceux qui recherchent une

solution de colorisation d'image alliant innovation, interactivité et haute qualité visuelle.

Au bout de 3 semaines nous avons donc sélectionné ces 4 modèles afin d'évaluer et comparer leur performance dans la colorisation d'images.

Les modèles supervisés sont entraînés avec des données où chaque image en nuances de gris est liée à sa version en couleur, permettant ainsi au modèle d'apprendre directement la correspondance entre les deux. En revanche, les modèles non supervisés apprennent à générer des colorations plausibles sans correspondances directes, ce qui rend la fidélité des couleurs imparfaite.

Dataset :

Nos modèles ont été entraînés sur 100 000 images du dataset ImageNet [4], un large ensemble de données largement utilisé dans les compétitions et recherches en vision par ordinateur. ImageNet est idéal pour ce type de tâche car il comprend une grande variété d'images en haute résolution avec de nombreuses catégories différentes, permettant ainsi au modèle de généraliser la colorisation à un large éventail de sujets et de scènes.

Type d'entraînement, Avantages et Inconvénients

Nous avons commencé par tester les 4 modèles et nous avons également fait des recherches à leurs sujets afin de comparer les avis et critiques les plus courants d'autres testeurs aux nôtres, voici alors un résumé des avantages et des inconvénients les plus souvent retrouvés de chaque modèle :

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

Les modèles **Artistic** et **Stable** font partie du même type de modèle appelé **NoGan**. L'idée principale derrière ce type de modèle est qu'on puisse bénéficier des avantages de la formation GAN tout en passant un minimum de temps à suivre une formation GAN directe.

Artistic

:

Avantages : Le modèle “Artistic” est optimisé pour offrir une coloration d'image de haute qualité, accentuant les détails et le dynamisme. Il utilise une architecture sophistiquée combinant ResNet 34 et U-Net, avec un focus sur une couche décodeur profonde, ce qui lui permet de produire des images colorisées avec une grande richesse de détails et une vivacité impressionnante.[1]

Inconvénients : Cependant, ce modèle présente des défis en termes de stabilité lorsqu'il est appliqué à des tâches plus ordinaires, comme la colorisation de scènes naturelles et de portraits. Il nécessite également un ajustement minutieux des paramètres et un temps considérable pour peaufiner les résultats, ce qui peut s'avérer impraticable pour une utilisation quotidienne ou pour des utilisateurs recherchant des solutions plus directes et moins laborieuses.[1]

Stable

:

Avantages : Le modèle "Stable" est particulièrement efficace dans la colorisation de paysages et de portraits, produisant des résultats supérieurs dans ces catégories. Grâce à son architecture avancée basée sur ResNet 101 et U-Net, ce modèle est capable de colorer de manière plus naturelle les visages humains, évitant ainsi l'effet de teint gris souvent rencontré avec d'autres techniques de colorisation.[1]

Inconvénients : Toutefois, bien qu'il offre une plus grande fidélité des couleurs et moins de décolorations étranges, ce modèle tend à produire des images moins saturées. Cela peut être perçu comme un désavantage dans des cas où des couleurs plus vives ou plus expressives sont souhaitées, car il peut manquer de l'intensité visuelle offerte par des modèles comme "Artistic".[1]

Cette structuration aide à évaluer clairement les forces et les limites de chaque modèle, fournissant une base solide pour choisir le modèle le plus approprié en fonction des besoins spécifiques de l'utilisateur.

Type d'entraînement : Dans le cadre de DeOldify, l'apprentissage non supervisé signifie que les modèles n'ont pas besoin de paires spécifiques d'images en noir et blanc et de leurs correspondants en couleur pour l'entraînement. Au lieu de cela, le modèle apprend de manière autonome à générer des colorations plausibles à partir de caractéristiques apprises dans les images non étiquetées. Le processus est renforcé par l'utilisation des GANs, où un générateur tente de créer des images colorisées et un discriminateur évalue si ces images sont réalistes (c'est-à-dire indiscernables des images en couleur réelles).[2]

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

Big Color

Avantages : Big Color se distingue par son excellence dans la manipulation de grandes images et la conservation des détails fins à haute résolution, grâce à son architecture optimisée qui combine des techniques de deep learning avancées. Ce modèle excelle notamment dans la colorisation d'images complexes et

détaillées, telles que des photographies de paysages urbains ou naturels, où la précision des détails et la richesse des couleurs sont primordiales.[7]

Inconvénients : Cependant, bien que le modèle soit efficace pour traiter de grandes images, il peut nécessiter des ressources computationnelles significatives, ce qui pourrait limiter son utilisation dans des environnements moins équipés. De plus, la saturation des couleurs peut parfois être moins intense, ce qui peut ne pas convenir à des projets nécessitant des colorisations extrêmement vibrantes et expressives.[7]

Type d'entraînement : Big Color adopte une approche d'apprentissage profond, principalement supervisé, nécessitant des jeux de données consistant en images noir et blanc auxquelles sont associées leurs versions colorées. Cette méthode permet au modèle d'apprendre des associations de couleurs précises, ce qui est crucial pour la fidélité des résultats finaux. L'utilisation d'une architecture puissante permet également de traiter efficacement des images de grandes dimensions, assurant une généralisation performante sur des images diverses.[7]

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

Modèle de Rich Zhang

Avantages : Le modèle de Zhang utilise une combinaison innovante de réseaux de neurones convolutifs et d'apprentissage profond pour offrir une colorisation de haute fidélité. Grâce à l'utilisation de la classification d'objets et des techniques d'apprentissage par renforcement, ce modèle parvient à coloriser avec précision des zones complexes telles que les visages et les textures naturelles. Il offre également une expérience interactive, permettant aux utilisateurs d'influencer la colorisation en proposant des couleurs pour des régions spécifiques, ce qui est idéal pour des applications demandant un haut degré de personnalisation et d'intervention humaine.[6]

Inconvénients : Malgré sa capacité à produire des résultats précis, le modèle peut parfois produire des images où les couleurs semblent moins vives comparées à celles générées par des méthodes plus artistiques comme le modèle "Artistic" de DeOldify. De plus, le besoin d'intervention manuelle pour ajuster les couleurs peut être un inconvénient pour les utilisateurs recherchant une solution entièrement automatique.[6]

Type d'entraînement : Le modèle utilise une approche d'apprentissage semi-supervisé qui combinent des méthodes classiques de supervision avec des éléments d'apprentissage non supervisé. Cela permet au modèle d'apprendre à partir d'un nombre relativement restreint d'images étiquetées tout en exploitant de

grandes quantités d'images non étiquetées, maximisant ainsi son efficacité et sa capacité à généraliser à partir de données d'entraînement diversifiées.[6]

Comparaison théorique des modèles

Bigcolor doit être théoriquement le plus performant, en effet au cours de travaux précédent des chercheurs ont comparé BigColor avec DeOldify (Stable, Artistic).[3]

Artistic est probablement le meilleur pour des tâches variées. Il utilise des GANs et fonctionne sans supervision, ce qui est idéal pour créer ou modifier des images de façon créative.

Stable est très efficace pour des tâches où la précision et la constance des images générées sont cruciales.

Zhang est le moins performant, utilisant uniquement CNN. Néanmoins, il peut convenir pour des images basiques.

Modèle	BigColor	Zang	Stable	Artistic
U-net		x	x	x
Resnet 34				x
Resnet 101			x	
CNN		x		
GAN	x		x	x
Supervisé		x		
Non-supervisé	x		x	x

Le NoGan est un type de Gan qui a pour but de minimiser le temps d'entraînement du Gan et de passer plus de temps sur l'entraînement des générateurs et du discriminateur mais séparément.
Donc durant le court entraînement de gan le générateur peut quasiment coloriser de manière correcte et presque sans artefact.
On va voir en détail juste l'entraînement d'un des deux modèles car les deux entraînements sont concrètement les mêmes et la différence se joue sur les paramètres.

Entraînement stable et artistic

Tout d'abord nous devons préparer le DataSet qu'on va utiliser. Sur les 100 000 images colorisées on les convertit en nuance de gris pour entraîner le générateur et le discriminateur.

On entraîne le générateur et le but est de pousser l'entraînement du générateur le plus loin possible en faisant varier la taille de l'image et le batch-size.

On commence par définir le batch-size, la taille de l'image et la proportion des données d'entraînement afin de charger les données pour le générateur (voir Figure 1) :

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

Figure 1

On crée ensuite notre générateur avec les paramètres suivants. Le premier paramètre on lui fait passer notre données, le second on définit la fonction de perte qui compare les caractéristiques des images générées et celles des images générées afin d'améliorer les images générées. Le troisième indique que le facteur de normalisation ici c'est 2 (Figure 2).

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

Figure 2

On lance l'entraînement du générateur avec comme paramètre un seul epoch, 80% du cycle le taux d'apprentissage augmente puis va décroître et on définit un taux d'apprentissage maximal de 1e-3(figure3). Puis on sauvegarde l'état actuelle du générateur (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

On va débloquer toutes les couches du modèle qui par défaut est bloqué afin d'approfondir l'apprentissage (figure 5), les paramètres restent les mêmes à part le taux d'apprentissage maximal qui varie entre 3^e-7 et 3^e-4. Puis on sauvegarde à nouveau l'état actuel du générateur (figure 4).

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

Figure 5

On va réentraîner de la même manière le générateur mais avec des paramètres différents afin d'améliorer la colorisation des 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)
```

On va passer maintenant à l'entraînement du discriminateur et le but est aussi de pousser le plus loin possible l'entraînement.

On commence par charger les données avec le batch-size à 64 et la taille de l'image à 128(Figure 6). Puis on crée le discriminateur avec les données chargées et on spécifie le nombre de filtres qu'on souhaite utilisés ici 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

On lance l'entraînement avec 6 epochs et un taux d'apprentissage maximum de $1e-3$ et on sauvegarde l'état actuelle du modèle (Figure 8)

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

Figure 8

On va répéter plusieurs fois la partie précédente au moins 5 fois mais avec un batch-size de 16, 192 tailles d'image ,4 epoch et un taux d'apprentissage de 1^e-4. Le nombre de répétitions varie selon ce qu'on obtient quand on répète l'entraînement du Gan.

Pour configurer le gan on détermine le paramètre du switcher, un switcher permet d'alterner entre l'entraînement du générateur et du critique. Le critic_thresh indique le seuil critique du discriminateur, passer le seuil l'entraînement passera au générateur. (Figure 9)

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

Figure S

Ensute on créer le gan en chargeant le générateur, le discriminateur, les poids lors des pertes pour le générateur, la fonction d'optimisation, le switcher, la fonction optimiseur pour mettre à jour le poids du modèle avec l'optimiseur Adam et l'hyper paramètre, la décroissance de poids afin d'éviter le surapprentissage. (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

L'entraînement sera seulement juste sur 30% du dataset et on gèle toutes les couches du générateur a par la dernière qui est la sortie afin d'accélérer l'entraînement. On lance l'entraînement avec un epoch et un taux d'apprentissage maximum de 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

Pour conclure voici un tableau récapitulatif des différents paramètres entre Stable et Artistic qu'on va utiliser plus tard. (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

Entraînement BigColor

Décortiquons analysons à présent le code d'entraînement du modèle Big Color qui était de très loin le plus long .Notre processus d'entraînement de ce modèle est alors organisé en huit étapes principales.

Nous commençons par la **configuration initiale** où nous définissons les paramètres essentiels du modèle. Ensuite, la **définition du modèle** implique la mise en place des architectures du générateur et du discriminateur. On passe ensuite à l'**optimisation et à la planification**, où nous configurons les optimiseurs et les planificateurs de taux d'apprentissage. Pour garantir une efficacité optimale sur plusieurs GPUs, on prépare le système avec l'**entraînement distribué**.

La **boucle d'entraînement** proprement dite traite les données par lots, effectuant la propagation avant et la rétropropagation. On utilise le **logging et le suivi** pour enregistrer les progrès et ajuster les paramètres si nécessaire. L'utilisation de la **moyenne mobile exponentielle (EMA)** est cruciale pour stabiliser les poids du modèle au fil du temps. Enfin, le **traitement d'image** assure que toutes les entrées sont correctement formatées et prêtes à être traitées par le réseau.

Ces étapes ensemble forment un processus complet qui guide notre apprentissage du modèle de colorisation d'images du début à la fin.

1) Configuration initiale

Dans la première phase du processus d'entraînement, nous nous concentrons sur la Configuration initiale du code. Ici, la bibliothèque argparse est utilisée pour gérer les options de configuration, permettant à l'utilisateur de spécifier une variété de paramètres

tels que les chemins des fichiers, les types de normalisation et les fonctions d'activation. On définit également les chemins vers des fichiers de configuration pré-entraînés et des répertoires de logs, assurant ainsi une organisation méticuleuse et une préparation complète avant le début de l'entraînement.

Cette étape fondamentale est encapsulée par la vaste fonction `parse_args()` du code, qui joue un rôle crucial dans la personnalisation et l'optimisation de tous les aspects du modèle et de son environnement d'entraînement.

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

...

Figure 13 : Début de la fonction `parse_args()`

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

Figure 14 : Fin de la fonction `parse_args()`

Nous utilisons cette fonction pour définir et traiter les arguments de ligne de commande que nous pouvons spécifier pour personnaliser l'exécution du script de formation. Cette méthode est essentielle pour rendre le script flexible et adapté à différents environnements de formation ou configurations de modèle sans que nous ayons besoin de modifier le code source. Cette fonction se compose elle-même de 11 parties.

La fonction `parse_args` inclut une partie de configuration générale qui identifie les différents travaux d'entraînement grâce à un nom (`task_name`) et nous permet de fournir des détails supplémentaires (`detail`) afin de mieux comprendre l'objectif ou la spécificité de chaque tâche.(Figure 15)

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

Figure 15

Ensuite, nous avons une section sur le mode de fonctionnement qui configure le comportement interne du modèle. Cela nous permet de sélectionner le type de

normalisation des données (`norm_type`), la fonction d'activation utilisée par les neurones (`activation`), et la méthode d'initialisation des poids (`weight_init`). Tous ces éléments sont essentiels pour optimiser les performances et la convergence du réseau. (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

La partie suivante, dédiée aux entrées et sorties, définit les chemins d'accès aux fichiers nécessaires pour l'entraînement. Nous y trouvons les données d'entraînement, les modèles pré-entraînés, ainsi que les lieux de sauvegarde des logs et des checkpoints. Cette section joue un rôle crucial dans l'organisation et la gestion des ressources de notre projet. (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_imnet_train', default='C:/ia/projet_colorisation/image_train')
parser.add_argument('--path_imnet_val', default='C:/ia/projet_colorisation/image_val')

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

La section d'entraînement du modèle est particulièrement intéressante car elle impacte directement le processus d'entraînement. Nous pouvons y déterminer le nombre d'époques (`num_epoch`), le nombre de couches du modèle (`num_layer`), et d'autres paramètres architecturaux influant sur la structure et la complexité du réseau. (Figure 18)

```

# 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

Concernant les optimiseurs, on commence par les définitions des paramètres relatifs à l'optimisation, tels que le taux d'apprentissage et les moments. Ces paramètres affectent la manière dont le modèle ajuste ses poids en réponse à l'erreur calculée pendant l'entraînement. Nous avons également des options pour ajuster dynamiquement ces taux via un programme (`scheduler`). (Figure 19)

```

# 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

La partie options de verbose nous permet de contrôler l'affichage des configurations actuelles, ce qui aide à comprendre et à déboguer le comportement du modèle pendant son exécution. (Figure 20)

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

Figure 20

La gestion des pré-entraînements détermine si nous devons charger des poids pré-entraînés pour le générateur et le discriminateur. Cette approche peut accélérer l'entraînement et améliorer la performance initiale du modèle en fournissant une base de départ plus avancée. (Figure 21)

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

Figure 21

La configuration des fonctions de perte définit les métriques selon lesquelles nous évaluons l'erreur du modèle pendant l'entraînement. Nous avons différentes fonctions de perte disponibles pour mesurer différentes facettes de la performance du modèle. (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

La moyenne mobile exponentielle (EMA) est configurée pour stabiliser les poids du générateur au fil du temps. Cela rend les résultats du modèle moins sensibles aux variations des dernières données traitées et plus robustes en général. (Figure 23)

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

Figure 23

Une autre partie comprend une variété d'autres configurations importantes, telles que la dimension de l'espace latent et la graine (seed) pour la génération aléatoire. Ces paramètres offrent un contrôle plus fin sur des aspects spécifiques de l'entraînement et de l'initialisation du modèle.(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

Enfin, la dernière partie est consacrée aux options GPU, nous permettant de configurer l'utilisation des ressources graphiques disponibles. Cela vise à maximiser l'efficacité de l'entraînement en exploitant un ou plusieurs GPUs, en fonction de la disponibilité et de la compatibilité de l'infrastructure matérielle.(Figure 25)

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

Figure 25

2) Définition du modèle

En ce qui concerne la définition du modèle, le code initialise deux types de modèles : un modèle Colorizer, destiné à la génération d'images, et un Discriminator utilisé pour le cadre de l'apprentissage adversarial. Cette étape inclut également la possibilité d'utiliser des modèles pré-entraînés en option, facilitant ainsi le fine-tuning des modèles pour améliorer leur précision et leur efficacité à partir de configurations déjà établies. Cette configuration est essentielle pour adapter et optimiser le modèle en fonction des besoins spécifiques et des caractéristiques des données traitées. (Figure 26)

```

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

Le code commence par configurer l'environnement, permettant un entraînement synchronisé sur plusieurs GPUs. Les modèles que nous utilisons, le Colorizer (générateur) et le Discriminator, sont initialisés selon les paramètres définis, et les configurations sont potentiellement pour utiliser des états pré-entraînés afin d'accélérer le processus et d'améliorer les performances initiales.

Vient ensuite en place des optimiseurs spécifiques pour chaque modèle, en ajustant les taux d'apprentissage et les paramètres de moment selon les besoins. C'est complété avec des planificateurs de taux d'apprentissage, si nécessaire, pour ajuster dynamiquement le taux en fonction de notre progression. Une attention particulière est

accordée à maintenir la stabilité et la fiabilité des résultats en utilisant la moyenne mobile exponentielle des poids du générateur au fil du temps.

Pour maximiser l'efficacité de l'entraînement parallèle, le code optimise le chargement et la distribution des données. Sa boucle d'entraînement principale traite les données par lots, colorisant les images, les évaluant, et ajustant nos modèles par rétropropagation des erreurs calculées à partir de diverses fonctions de perte. Le code assure également des points de contrôle réguliers pour suivre les performances et sauvegarder les états de nos modèles à des étapes cruciales.

En résumé, cette fonction représente un écosystème d'entraînement complet et rigoureusement structuré. Visant à optimiser et superviser le développement de modèles de colorisation d'images hautement performants et précis, adaptés à l'exécution sur des systèmes dotés de ressources de calcul distribuées.

3) Optimisation et à la planification

Dans la partie d'optimisation et de planification, on trouve la configuration des paramètres qui gouverneront le comportement de nos modèles pendant l'entraînement. Cette section établit les fondements pour une adaptation efficace des poids du réseau en fonction de notre progression.

Pour la Configuration des Optimiseurs, un optimiseur spécifique est défini pour chaque modèle. Le générateur (Colorizer) et le discriminateur ont leurs optimiseurs respectifs, utilisant l'algorithme d'Adam. On choisit Adam pour sa capacité à gérer efficacement des gradients épars et à s'adapter dynamiquement aux différents types de données. Le code configure les taux d'apprentissage et les moments beta (b_1 et b_2) en fonction de ses recommandations ou expériences antérieures, où b_1 contrôle la décroissance exponentielle du taux d'estimation du gradient et b_2 celle du carré du gradient, pour un ajustement plus fin et un comportement plus prévisible pendant l'apprentissage. (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

Dans la Planification des Taux d'Apprentissage, le code ajuste les taux d'apprentissage à chaque époque à l'aide d'un planificateur qui suit un programme prédéfini. Ce système de planification peut adopter différentes formes, telles que multiplicative (mult) ou

linéaire (linear), influençant la vitesse à laquelle le taux diminue. Cette stratégie vise à affiner l'apprentissage en réduisant progressivement le taux à mesure que le modèle converge, ce qui optimise les performances et stabilise le réseau. (Figure 28)

```
# Scheduler
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 schedule 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

En résumé, cette section du code assure que nos modèles bénéficient d'un environnement d'optimisation adaptatif qui favorise une convergence efficace et régulière. Cela permet de réduire les risques de blocage dans des minimums locaux ou de sur-ajustement aux données d'entraînement. Ces outils d'optimisation et de planification sont essentiels pour exploiter pleinement le potentiel des architectures complexes et obtenir les meilleurs résultats possibles.

4) Entraînement distribué

Dans la section Entraînement distribué, le code est spécifiquement conçu pour fonctionner sur plusieurs GPUs en utilisant DistributedDataParallel. Cette approche permet un entraînement parallèle et efficace sur des architectures multi-GPU. Le code intègre aussi des méthodes telles que autocast et GradScaler pour optimiser l'exécution en termes de mémoire et de calcul. En particulier, cela nous permettra d'utiliser des opérations en demi-précision (FP16), jouant ainsi un rôle crucial pour maximiser l'efficacité de l'entraînement. Cela réduit le temps nécessaire et augmente la capacité du modèle à traiter de grandes quantités de données sans saturer la mémoire disponible. (Figure 29 C 30)

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

Figure 29

```

# 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

Dans cette partie de la fonction train dédiée à l'entraînement distribué, est gérée l'exécution du modèle sur plusieurs unités de traitement graphique (GPU), ce qui est essentiel pour traiter efficacement de grands volumes de données et des architectures complexes dans des délais raisonnables.

Dans la phase d'Initialisation de l'Environnement Distribué, le code commence par configurer les connexions nécessaires pour la communication entre les multiples GPUs, en définissant les adresses et les ports requis. Cette étape inclut l'utilisation de `dist.init_process_group` avec `gloo` comme backend, facilitant ainsi les échanges interprocessus essentiels pour un entraînement synchronisé sur différents dispositifs.

Pour la Configuration de PyTorch pour la Distribution, chaque composant du modèle - que ce soit le générateur (EG), le discriminateur (D), ou l'outil de perception VGG - est intégré dans une architecture `DistributedDataParallel` (DDP). DDP, un outil de PyTorch, parallélise automatiquement les données et les calculs à travers les GPUs disponibles. Cette parallélisation garantit une distribution équilibrée du travail, une agrégation efficace des gradients, et optimise l'utilisation des ressources computationnelles pour un entraînement rapide et efficace.

La Gestion des Ressources GPU revêt également une importance critique, impliquant l'optimisation de la mémoire et des devices pour éviter tout conflit. Les fonctions comme `torch.cuda.set_device` et `torch.cuda.empty_cache` sont utilisées pour assigner spécifiquement chaque processus à un GPU dédié et libérer la mémoire non utilisée.

Cela prévient les problèmes courants de saturation mémoire rencontrés avec des modèles complexes.

Enfin, lors de l'Exécution de la Formation, une fois que l'environnement et les modèles sont prêts, la boucle d'entraînement démarre le traitement des données de manière efficace et parallèle. Les échantillonneurs distribués assurent que chaque instance de modèle traite une section unique du dataset, évitant ainsi les doublons et garantissant une couverture exhaustive à chaque époque d'entraînement. Cette coordination fine permet une utilisation optimale des ressources disponibles et une progression régulière vers la convergence du modèle.

En résumé, cette partie organise et exécute l'entraînement distribué pour optimiser les performances computationnelles et la gestion des ressources. Cela permet de traiter efficacement de grands volumes de données avec des réseaux de neurones profonds sur des architectures parallèles, ce qui se traduit par un gain de temps considérable et une exploitation maximale des capacités matérielles disponibles.

5) Boucle d'entraînement

Dans la Boucle d'entraînement, nous traitons itérativement des échantillons où des images en nuances de gris sont transformées en images colorisées par le générateur. En parallèle, on trouve le calcul des pertes distinctes pour le générateur et le discriminateur afin d'évaluer leur performance respective. Cette évaluation détermine comment le générateur réussit à tromper le discriminateur avec des images qui doivent sembler réalistes, tandis que le discriminateur apprend à distinguer les images générées de celles qui sont réellement captées.

Le code sauvegarde périodiquement l'état du modèle pour permettre la reprise de l'entraînement en cas d'interruption et pour effectuer des évaluations sur un ensemble de validation. Ces évaluations nous aident à mesurer la capacité du modèle à généraliser et à nous assurer que les améliorations de performance ne se limitent pas uniquement aux données utilisées pour l'entraînement. (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增强(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

Dans la Boucle d’entraînement, commence le processus en itérant sur le nombre total d’époques définies, chaque époque représentant un passage complet sur l’ensemble des données d’entraînement. Le code utilise un échantillonneur distribué pour que chaque processus traite équitablement une partie des données, maximisant ainsi l’utilisation de toutes les ressources GPU disponibles. Les données sont chargées par lots via un DataLoader à chaque époque, et le réseau prend en charge les images en niveaux de gris fournies pour que le générateur tente de les transformer en images colorisées.

La Propagation Avant et le Calcul de la Perte interviennent ensuite : le générateur crée des images colorisées à partir des entrées, et le discriminateur évalue ces images par rapport aux images réelles. Les fonctions de perte évaluent les erreurs du générateur et du discriminateur, mesurant ainsi la qualité de la colorisation et la capacité du discriminateur à détecter les faux. Dans la phase d'Optimisation : Rétropropagation et Mise à Jour, le code calcule les gradients à partir de ces pertes et les utilise pour mettre à jour les poids du réseau via les optimiseurs préconfigurés, améliorant ainsi la précision des prédictions du modèle au fil du temps.

Le Suivi et la Sauvegarde jouent un rôle essentiel, avec des mécanismes en place pour enregistrer les progrès de l'entraînement. Une sauvegarde périodique est faite pour les images générées et les métriques de perte, et le code conserve des points de contrôle à des intervalles prédéfinis, facilitant ainsi la reprise de l'entraînement et l'évaluation des modèles sauvegardés. La Gestion de la Mémoire et la Synchronisation sont également critiques, avec des nettoyages de mémoire effectués à chaque itération pour optimiser l'utilisation des ressources GPU, et des synchronisations pour assurer que tous les GPUs terminent leurs tâches de manière cohérente avant de passer au lot suivant. Cette section coordonne efficacement la gestion des données, les calculs parallèles, et les optimisations algorithmiques, orchestrant un apprentissage efficace et de qualité pour le modèle de colorisation.

6) Logging et suivi

Dans la section Logging et suivi, le code crée systématiquement des logs et utilise TensorBoard pour visualiser la progression de l'entraînement. Cette fonctionnalité inclut l'enregistrement de métriques importantes comme les pertes et fournit des aperçus des images générées. Ce suivi nous permet non seulement de monitorer les avancées et les améliorations du modèle au fil du temps, mais aussi d'identifier rapidement les éventuelles anomalies ou points d'amélioration dans le processus d'entraînement. En fournissant une interface visuelle intuitive, TensorBoard aide les utilisateurs à comprendre en profondeur le comportement et l'efficacité de leur modèle. (Figure 32)

```

# 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

Cette partie de la fonction train se concentre sur le logging et le suivi de l'entraînement, jouant un rôle crucial dans l'évaluation de la performance du modèle et l'ajustement des paramètres si nécessaire.

Dans la section Logging et suivi, une grande importance est accordée à l'enregistrement régulier des métriques pendant l'entraînement pour évaluer la performance des modèles. Cela inclut la surveillance continue des pertes du générateur et du discriminateur, fournissant une vue détaillée de l'évolution du modèle et signalant les nécessités d'ajustement pour optimiser l'apprentissage. L'utilisation de TensorBoard facilite la visualisation des données, permettant de suivre des métriques variées telles que les courbes de perte et les images générées. Cela donne aux utilisateurs la capacité de détecter visuellement des problèmes potentiels comme le sur-ajustement ou une sous-performance de l'entraînement.

Nous sauvegardons régulièrement des échantillons d'images générées pour une évaluation qualitative des performances du modèle. Ces images permettent d'évaluer la qualité de la colorisation et la pertinence des résultats par rapport aux images en niveaux de gris d'origine. Les points de contrôle, comprenant les poids du générateur et du discriminateur ainsi que les états des optimiseurs, sont enregistrés à des intervalles prédéfinis. Cette méthode est essentielle pour la reprise de l'entraînement après des interruptions, la réPLICATION des expériences, ou le déploiement des modèles en production.

Le système de logging et de suivi joue également un rôle crucial dans la gestion dynamique de l'entraînement, permettant des ajustements rapides des paramètres en réponse à des observations imprévues. Par exemple, si les logs montrent un plateau dans l'amélioration du modèle, nous pouvons ajuster le taux d'apprentissage ou d'autres hyperparamètres pour favoriser une progression continue. Cette flexibilité est essentielle pour maintenir et optimiser les performances du modèle de colorisation.

En résumé, cette section est dédiée à la collecte, à l'analyse et à la visualisation des données générées pendant l'entraînement. Un suivi efficace est crucial pour naviguer dans la complexité du processus d'entraînement des modèles de deep learning, fournissant les insights nécessaires pour améliorer les performances et garantir la production de résultats de haute qualité.

7) Moyenne mobile exponentielle (EMA)

Dans la section Moyenne mobile exponentielle (EMA), le code applique une technique de moyenne mobile exponentielle aux poids du générateur afin d'améliorer leur stabilité au fil du temps. Cette méthode est cruciale pour lisser les variations des poids observées sur plusieurs itérations d'entraînement, ce qui aide à atténuer l'impact des fluctuations abruptes dues à des lots particulièrement atypiques ou à des données bruitées. L'intégration de l'EMA rend les poids ajustés moins sensibles à ces variations isolées, favorisant ainsi un comportement de convergence plus prévisible et robuste du modèle. Cette approche se révèle particulièrement utile dans les phases avancées de l'entraînement, où des ajustements mineurs peuvent avoir des impacts significatifs sur la performance globale du modèle. En somme, l'utilisation de la moyenne mobile exponentielle contribue à stabiliser et à améliorer la cohérence des poids du générateur tout au long du processus d'entraînement, renforçant ainsi la fiabilité et l'efficacité du modèle de colorisation d'images. (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

La partie EMA de la fonction train se concentre sur l'utilisation de la moyenne mobile exponentielle (EMA) pour la gestion des poids du générateur dans le processus d'entraînement. L'EMA est une technique essentielle pour lisser les fluctuations des poids du modèle au fil du temps, ce qui contribue à stabiliser l'apprentissage et à réduire la variance des résultats finaux.

L'objectif principal de l'EMA est de maintenir une version moyenne des poids du générateur qui intègre une proportion des poids précédents. Cela permet de stabiliser les variations des poids observées d'une époque à l'autre, surtout dans le cadre d'entraînements longs et complexes typiques des réseaux profonds. L'implémentation de l'EMA commence par l'initialisation d'un objet ExponentialMovingAverage avec un taux de déclin spécifié. Ce taux détermine la vitesse à laquelle les poids historiques sont oubliés au profit des nouveaux poids.

À chaque itération d'entraînement, après la mise à jour des poids du générateur par la rétropropagation, l'EMA est mise à jour pour refléter les nouveaux poids. Cela garantit que les poids moyens utilisés pour les prédictions restent stables et cohérents au fil du temps. Les avantages de l'EMA sont significatifs, car cette méthode aide à obtenir un modèle final plus robuste et performant. Les poids moyens moins sensibles aux fluctuations extrêmes et aux variations brusques améliorent la généralisation du modèle, le rendant plus fiable dans des scénarios réels.

Intégrer l'EMA dans la boucle d'entraînement assure que même les ajustements mineurs mais constants des poids du modèle sont pris en compte de manière à favoriser la stabilité et la performance à long terme. Cette approche continue est cruciale pour optimiser l'efficacité et l'utilité des modèles de Deep Learning formés sur de vastes ensembles de données et sur des périodes prolongées.

En résumé, la section EMA de la fonction train enrichit la gestion des poids du générateur en permettant une convergence plus douce et plus fiable du modèle tout au long de l'entraînement. Cette méthode est essentielle pour renforcer l'efficacité des modèles de Deep Learning, en particulier dans des contextes où la stabilité et la performance à long terme sont des priorités.

8) Traitement d'images

Dans la partie traitement d'images , des transformations d'images sont appliquées pour préparer les données, incluant la conversion en tenseurs, le redimensionnement et le recadrage.(Figure 34)

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

dataset, dataset_val = prepare_dataset(
    args.path_imnet_train,
    args.path_imnet_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

Cette partie de la fonction train se concentre sur le traitement des images, une étape cruciale dans la préparation des données avant leur utilisation dans le réseau de neurones pour l'entraînement. Ce processus garantit que les images sont dans un format et une échelle appropriée pour être traitées efficacement par le modèle de colorisation.

Dans la section Traitement des Images, plusieurs transformations sont appliquées pour préparer les données. Cela inclut souvent la redimension, la normalisation et d'autres opérations spécifiques nécessaires pour répondre aux exigences du modèle. Pour gérer ces transformations de manière structurée, on utilise généralement `torchvision.transforms.Compose`, qui permet de créer un pipeline de transformations. Ce

pipeline peut comprendre la conversion des images en tenseurs PyTorch, leur redimensionnement, le recadrage centré, et éventuellement la normalisation des valeurs de pixels, facilitant ainsi leur manipulation par le réseau.

La Préparation des Données assure une cohérence essentielle entre les phases d'entraînement et de validation. Les mêmes transformations doivent être appliquées aux données d'entraînement et de validation pour maintenir une uniformité indispensable. Les données sont généralement organisées en ensembles distincts, avec les chemins spécifiés par les arguments de la fonction, garantissant ainsi que chaque image est traitée de manière identique avant d'être introduite dans le modèle.

Le Chargement Efficace des Données est optimisé à l'aide d'un DataLoader de PyTorch. Ce composant permet un chargement parallèle des données grâce à plusieurs travailleurs (num_workers), ce qui optimise l'utilisation de la mémoire et accélère le processus. Cela est particulièrement avantageux dans les systèmes équipés de plusieurs GPUs, réduisant les goulots d'étranglement et maximisant l'efficacité du traitement des données pendant l'entraînement.

Enfin, l'Extraction d'Échantillons de Validation joue un rôle crucial pour évaluer périodiquement le modèle. Ces échantillons, utilisés pour des validations régulières, permettent de suivre la progression du modèle et d'ajuster les paramètres en fonction de la performance observée sur des données non vues pendant l'entraînement régulier. Cela garantit que le modèle reste performant et adaptatif, même face à des situations nouvelles ou inattendues.

En résumé, cette partie de la fonction train orchestre efficacement le prétraitement des images, assurant que le modèle de colorisation apprend de manière optimale. En standardisant et en optimisant le format des données en amont, cette approche réduit les risques liés à des formats de données inappropriés ou à des chargements inefficaces, tout en facilitant un ajustement précis des performances du modèle sur les données traitées.

Conclusion Big Color

Pour conclure sur l'entraînement du modèle Big Color, son script utilise un réseau GAN pour la colorisation d'images, optimisé pour fonctionner efficacement sur plusieurs GPU. Cette méthode avancée combine un générateur qui produit des images colorisées et un discriminateur qui évalue leur authenticité, optimisant la qualité visuelle des résultats.

La flexibilité du script permet des réglages précis grâce à des paramètres configurables et l'utilisation de poids pré-entraînés, ce qui améliore la convergence et la fidélité des

images. Le processus d'entraînement est soutenu par un suivi en temps réel et des sauvegardes périodiques via TensorBoard, facilitant l'évaluation continue et la gestion de l'entraînement.

Malgré sa complexité et ses coûts de calcul, cette méthode est bien adaptée pour produire des résultats de colorisation réalistes et est bénéfique pour la recherche et les applications pratiques.

Cet entraînement n'en reste pas moins le plus rigoureux de nos 4 modèles.

Entraînement du modèle Zhang

Le code d'entraînement du modèle de Zhang traite de la préparation des données d'image pour l'entraînement d'un modèle de réseau de neurones convolutifs (CNN).

Ce code est séparé en 6 grandes parties, elles comprennent la Séparation des données, la Reshape des données, la Définition des couches de downsampling et d'upsampling, la Définition du modèle, la Compilation et entraînement du modèle et enfin l'Évaluation et sauvegarde du modèle.

```
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))
```

Dans notre étude du code d'entraînement du modèle de Zhang, nous avons analysé en premier lieu la séparation des données en ensembles d'entraînement et de test.

Le code commence par définir les variables `train_gray_image` et `train_color_image`, qui contiennent respectivement les images en niveaux de gris et en couleur pour l'entraînement, en sélectionnant les premières 38 950 images. Ensuite, les variables `test_gray_image` et `test_color_image` sont définies pour contenir les images restantes destinées au test.

Pour diviser les images, le code utilise des indices spécifiques, puis affiche la taille de l'ensemble d'entraînement, confirmant qu'il contient bien 38 950 images. Cette méthodologie assure une séparation claire et cohérente des données, essentielle pour la validité des phases d'entraînement et de test du modèle.

```
# 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)
```

Nous avons ensuite examiné la deuxième partie, qui consiste à redimensionner les images.

Le code redimensionne les variables `train_g` et `train_c` pour leur donner la forme (`nombre_d'images, taille, taille, 3`). Cette structure inclut trois canaux de couleur, même pour les images en niveaux de gris, afin de maintenir une uniformité avec les images en couleur.

Après ce redimensionnement, le code affiche les nouvelles formes des ensembles d'images d'entraînement et de test. Cette étape est essentielle pour s'assurer que

toutes les images, qu'elles soient en niveaux de gris ou en couleur, soient traitées de manière cohérente par le modèle.

```
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
```

Nous avons examiné la troisième partie, où deux fonctions utilitaires pour le modèle sont définies : `down` et `up`.

La fonction `down` crée une couche de downsampling en appliquant une convolution, suivie optionnellement de la normalisation par lot (batch normalization) et d'une

activation LeakyReLU. Quant à la fonction `up`, elle crée une couche d'upsampling en utilisant une convolution transposée, suivie optionnellement d'un dropout et d'une activation LeakyReLU.

Ces deux fonctions sont cruciales pour la construction du modèle, permettant de réduire et d'augmenter la résolution des images de manière contrôlée tout en maintenant les caractéristiques importantes des données grâce aux activations et aux normalisations appliquées.

```
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()
```

Dans notre étude du code d'entraînement du modèle de Zhang, nous avons analysé la quatrième partie, qui concerne la définition du modèle de réseau de neurones.

La couche d'entrée, `layers.Input`, est configurée pour accepter des images de taille 160x160 avec 3 canaux de couleur. Le modèle inclut plusieurs couches de downsampling (d1 à d5) et d'upsampling (u1 à u5), où des couches sont concaténées pour préserver l'information tout au long du réseau. La couche de sortie, `output`, est une convolution destinée à générer l'image colorisée.

Ensuite, le modèle est compilé en utilisant l'optimiseur Adam avec un taux d'apprentissage de 0.001. La fonction de perte utilisée est l'erreur absolue moyenne (mean absolute error), et la précision est choisie comme métrique d'évaluation. Cette structure permet au modèle de traiter et de coloriser les images de manière efficace et précise.

```
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)
```

La cinquième partie se concentre sur l'entraînement du modèle.

La méthode `model.fit` est utilisée pour entraîner le modèle sur les données d'entraînement. L'entraînement se déroule sur 50 époques avec une taille de batch de 60. Pour maintenir la propreté de l'interface, les détails de l'entraînement ne sont pas affichés grâce au paramètre `verbose=0`.

Cette approche permet au modèle d'apprendre à partir des données d'entraînement sur un nombre d'époques spécifié, en ajustant progressivement ses poids et ses paramètres pour minimiser l'erreur et améliorer la performance de la colorisation des images.

```
model.evaluate(test_gray_image,test_color_image)

model.save('mon_modele.h5')

model.save('mon_modele.keras')
```

La sixième partie se concentre sur l'évaluation et la sauvegarde du modèle.

Tout d'abord, le modèle est évalué sur les données de test pour mesurer sa performance, en utilisant les métriques appropriées définies lors de la compilation du modèle.

Ensuite, pour assurer la préservation du modèle entraîné, celui-ci est sauvegardé dans deux formats différents : HDF5 avec l'extension `.h5` (nommé `mon_modele.h5`) et le format natif Keras avec l'extension `.keras` (nommé `mon_modele.keras`). Il est noté qu'un avertissement spécifique que le format HDF5 est considéré comme obsolète, et il est recommandé d'utiliser le format natif Keras pour la sauvegarde du modèle afin de garantir la compatibilité future et la stabilité de la sauvegarde.

Cette dernière étape assure non seulement une évaluation rigoureuse de la performance du modèle sur des données indépendantes, mais aussi une sauvegarde sûre et durable du modèle entraîné pour une utilisation future ou la reprise de l'entraînement.

Conclusion du modèle de Zhang

Pour conclure ce code propose un workflow complet pour la préparation des données, la définition, la compilation, l'entraînement, l'évaluation et la sauvegarde d'un modèle de réseau de neurones convolutifs dédié à la coloration d'images. Il commence par la préparation des ensembles d'entraînement et de test, le redimensionnement des données pour la compatibilité avec le modèle, et l'utilisation de fonctions utilitaires pour le downsampling et l'upsampling. Le modèle est construit avec des couches de downsampling et d'upsampling, compilé avec Adam comme optimiseur, puis entraîné sur les données d'entraînement. Après évaluation sur les données de test, le modèle est sauvegardé dans les formats HDF5 et natif Keras, ce dernier étant recommandé pour la sauvegarde future.

Comparaison pratique des modèles

Nous allons commencer par comparer le temps d'entraînement de chaque modèle. Dans un premier temps, nous analyserons les modèles Stable et Artistic, car leurs processus

d'entraînement sont fondamentalement similaires, à l'exception de certains paramètres. Cependant, le temps d'entraînement varie de manière significative à certaines étapes de l'entraînement.

Tout d'abord, le pré-entraînement du générateur des deux modèles présente des écarts significatifs à certains endroits, comme le montre le tableau de la Figure 1. Les paramètres pour la partie où la taille de l'image est de 64 pour les modèles Stable et Artistic sont les mêmes. L'écart de temps d'entraînement s'explique donc par la différence de facteur d'échelle, d'après la Figure 12. En effet, le facteur d'échelle détermine le nombre de filtres dans chaque couche convolutionnelle, ce qui augmente la complexité du générateur et donc le temps d'entraînement.

Pour une taille d'image de 128 (sz=128), nous n'avons pas pu lancer l'entraînement pour le modèle Stable en raison des capacités limitées de notre ordinateur et du temps imparti. Cependant, nous avons pu lancer l'entraînement pour la partie sz=128 du modèle Artistic, sachant qu'Artistic utilise un batch size plus grand que Stable (Figure 12). En théorie, nous pourrions lancer l'entraînement de Stable si nous ne tenions pas compte du facteur d'échelle.

Pour une taille d'image de 192 (sz=192), le batch size est également plus important pour Artistic (Figure 12), mais Artistic prend presque deux fois moins de temps que Stable pour compléter son entraînement. Cela montre clairement que le facteur d'échelle joue un rôle important dans le temps d'entraînement du générateur.

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

Figure 1: Tableau représentant le temp pour le pré-entraînement du générateur

Lors du pré-entraînement du discriminateur, il apparaît que le facteur d'échelle a également une influence significative sur le temps d'entraînement, comme le montre la figure 2. Les paramètres du discriminateur sont identiques entre les modèles Artistic et Stable, et l'on peut clairement observer une différence notable dans le temps d'entraînement. On observe quasiment la même différence pour le pré-entraînement du discriminateur avec un batch-size différent et size différent (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: Tableau représentant le temp pour le pré-entraînement du discriminateur(batch-size=64 et 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: Tableau représentant le temp pour le pré-entraînement du discriminateur(batch-size=16 et sz=192)

Le temps d'entraînement du Gan entre Stable et Artistic ont à peu près la même durée d'environ une heure. Donc si on additionne le temp totale pour chacun des deux modèles on a 3 jours:1h:53min pour Stable et seulement 13 h:34 min pour Artistic.

Pour Big Color, on est parti sur 8 epoch, 8 pour la taille des couches,taille de batch-size à 17 pour qu'on puisse tourner sur notre machine.On a mis à 8 epochs car l'entraînement d'une seule epoch dure 30 h donc cela fait environ 10 jours d'entraînement.

Le modèle Cnn on a pris comme paramètre 60 epoch,30 batch-size et 128 comme taille d'image cela a pris environ 2 jours pour son entraînement.

Donc on peut résumer par le tableau ci-dessous(Figure 4):

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

Figure 4: Temp total des entraînement de chaque modèles

Nous avons pris un échantillon de 100 images variées puis de les convertir en nuances de gris afin que les modèles puissent les traiter. Pour comparer les performances nous allons utiliser la formule CIEDE 2000 qui calcule la différence entre deux couleurs(Figure 5). Nous avons choisi cette formule car il tient en compte plusieurs facteurs tels que la luminosité,le chroma et la teinte. Nous allons appliquer cette formule entre les images produits par les modèles et les images originales puis faire une moyenne.Plus DeltaE00 est proche de 0 meilleur est le résultat.

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

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

$\Delta L'$: La différence de luminosité

$\Delta C'$: La différence de chroma.

$\Delta H'$: La différence de teinte.

SL, SC, SH : Fonctions de pondération pour la luminosité, le chroma et la teinte.

RT : Un terme de rotation pour gérer l'interaction entre les différences de chroma et de teinte.

Figure 5: Formule générale [18]

Commençons par le modèle big color qui a pour avantage de sauvegarder un modèle à chaque epoch. Dans le tableau suivant(Figure 6) nous pouvons voir qu'à chaque epoch le résultat ne s'améliore si on se base sur la formule.Si on regarde directement les images générées à chaque epoch les résultats s'améliorent au fur et à mesure des epochs.

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: valeur moyen du delta pour 100 images à chaque epoch du big color

Si on regarde les performances globaux des modèles(Figure7) on peut voir qu'Artistic est le meilleur et en dernier Big-color-8(epoch).Stable aurait dû être le meilleur en terme performe mais comme j'avais dit précédemment on a dû sauter une partie de l'entraînement.Cependant les résultats de big-color-8 est bien mieux que celle de cnn si on regarde directement les images générées.

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

Figure 7: valeur moyen du delta pour 100 images pour chaque modèle

En conclusion, en termes de temps et de performance Artistic et Stable sont bien mieux. Mais cependant avec plus de temp big color en théorie doit être le meilleur. Vous pouvez voir les images générées par chaque modèle[20] pour avoir votre propres avis.

Le site Web

Backend

1.1 Introduction au Backend

Le backend du projet COLORIA joue un rôle crucial en assurant la gestion des requêtes HTTP, le traitement des images téléchargées par les utilisateurs et la communication avec le frontend. Construit avec Node.js et Express.js, il offre une solution légère et performante pour les opérations côté serveur. Node.js permet l'exécution de JavaScript côté serveur, tandis qu'Express.js fournit une structure flexible pour une création robuste.

1.2 Structure du Projet Backend

Le backend du projet est organisé de manière à séparer clairement les différentes responsabilités, facilitant ainsi la maintenance et l'évolutivité. Voici une vue d'ensemble de l'arborescence des fichiers du projet :

1.3 Configuration du Serveur

Le fichier 'server.js' configure le serveur Node.js pour gérer les requêtes de téléchargement, de traitement et de téléchargement d'images.

Installation et démarrage

- Installation des dépendances : Dans le dossier racine, pour installer toutes les dépendances nécessaires utilisez :
 - `npm install`
 - `npm install -g nodemon`
 - `npm install multer`
 - `npm install express`
- Démarrage du serveur : Utilisez `nodemon server.js` pour démarrer le serveur, ce qui permet une relance automatique lors de modifications.

Middlewares utilisés

- Journalisation des requêtes : Chaque requête est enregistrée avec l'heure, l'adresse IP, le type de requête et l'URL. Cela permet de suivre les activités et de déboguer les problèmes plus facilement.
- CORS (Cross-Origin Resource Sharing) : Permet aux ressources du serveur d'être accessibles depuis des domaines différents, essentiel pour les applications frontend-backend séparées.
- Servir des fichiers statiques : Les fichiers statiques (HTML, CSS, JS) sont servis depuis le répertoire `public`, et les images téléchargées depuis le répertoire `uploads`.

1.4 Gestion des Téléchargements d'Images

Le serveur utilise Multer pour gérer les fichiers téléchargés. Multer stocke les fichiers dans le répertoire `uploads` et les nomme de manière unique pour éviter les conflits. Les fichiers sont nommés en utilisant une combinaison du champ du fichier, de l'horodatage et d'un identifiant unique.

1.5 Routes et Endpoints

Le serveur gère plusieurs routes principales :

- /upload : Cette route permet le téléchargement des images. Les utilisateurs peuvent télécharger jusqu'à 5 images à la fois. Pour l'option "Tous_les_Modeles", une seule image est autorisée.
- /download : Cette route permet le téléchargement des images traitées. Les utilisateurs peuvent télécharger les images via un lien direct.
- /test : Une route de test simple pour vérifier la réception correcte des requêtes POST.

1.6 Gestion des Fichiers

Les fichiers téléchargés sont stockés dans le répertoire `uploads`. Pour économiser de l'espace disque, les fichiers d'origine et les fichiers traités sont automatiquement supprimés après 10 minutes. Cette fonctionnalité est implémentée à l'aide de la fonction `setTimeout` de JavaScript, qui planifie la suppression des fichiers après le délai spécifié.

1.7 Sécurité et Validation

Le serveur effectue plusieurs validations pour assurer la sécurité et l'intégrité des fichiers :

- Validation des fichiers : Vérifie que les fichiers téléchargés sont bien des images avec le header du fichier '/image' et limite le nombre de fichiers à 5 par upload (ou 1 pour l'option "Tous_les_Modeles").
- Messages d'erreur clairs : En cas de problème, des messages d'erreur détaillés sont renvoyés pour informer les utilisateurs et le serveur de la nature exacte du problème.

1.8 Optimisations et Bonnes Pratiques

Le serveur est optimisé pour la performance grâce à plusieurs techniques et meilleures pratiques :

- Utilisation de middlewares : Les middlewares sont utilisés pour la journalisation, la gestion des erreurs et le CORS, ce qui améliore la modularité et la maintenabilité du code.
- Exécution asynchrone des scripts : Les scripts externes (comme le script Python de traitement d'image) sont exécutés de manière asynchrone pour éviter de bloquer le serveur.
- Gestion centralisée des erreurs : Les erreurs sont gérées de manière centralisée pour fournir des messages d'erreur cohérents et informatifs.
- Nettoyage automatique des fichiers : Les fichiers téléchargés et traités sont automatiquement supprimés après un certain délai pour économiser de l'espace disque.

Conclusion du I

Le backend du projet COLORIA est conçu pour être performant, sécurisé et maintenable. Il gère efficacement les téléchargements et les traitements d'images, fournit des endpoints clairs pour les opérations principales et suit les meilleures pratiques de développement backend. Ces fonctionnalités assurent une expérience utilisateur fluide et fiable, tout en facilitant la maintenance et l'évolutivité du projet.

Frontend

2.1 Introduction au Frontend

Le frontend du projet COLORIA est responsable de l'interface utilisateur. Il permet aux utilisateurs de télécharger des images, de sélectionner des modèles de traitement et de visualiser les résultats de manière intuitive et interactive. Il communique également des informations quant à l'utilisation du site et au contact de l'équipe. Le frontend utilise principalement HTML, CSS et JavaScript pour offrir une expérience utilisateur fluide et agréable.

2.2 Structure du Projet Frontend

Le frontend est organisé de manière à offrir une navigation claire et une gestion efficace des fichiers. Voici la structure des fichiers et répertoires du frontend :

2.3 Pages HTML

site.html

`site.html` est la page principale où les utilisateurs peuvent téléverser des images, sélectionner un modèle de traitement et visualiser une comparaison avant/après traitement des images.

Fonctionnalités :

- Cadre de dépôt : Permet aux utilisateurs de glisser-déposer des images ou de les sélectionner via un bouton.
- Sélection de modèle : Permet aux utilisateurs de choisir le modèle de traitement à appliquer aux images.
- Bouton d'envoi : Envoie les images sélectionnées au serveur pour traitement.
- Slider de comparaison : Permet de comparer visuellement une image avant et après traitement.
- Logo et favicon : un logo et un dossier de favicon permettant une intégrité sur tous les navigateurs ont été mis en place pour un esthétique soigné du site.

result.html

`result.html` affiche les images traitées et permet leur téléchargement.

Fonctionnalités :

- Liste des images : Affiche les images traitées avec des boutons de téléchargement pour chaque image.
- Bouton de retour : Permet de revenir à la page d'accueil.

all_models_result.html

`all_models_result.html` affiche les images traitées par tous les modèles disponibles.

Fonctionnalités :

- Liste des images : Affiche les images traitées avec des boutons de téléchargement pour chaque image, triées par modèle.
- Bouton de retour : Permet de revenir à la page d'accueil.

a_propos_de_nous.html

`a_propos_de_nous.html` fournit les informations de contact pour le service COLORIA.

Fonctionnalités :

- Adresse électronique : Fournit un mail de contact pour l'équipe COLORIA.

fonctionnement.html

`fonctionnement.html` explique le fonctionnement du service COLORIA.

Fonctionnalités :

- Instructions : Décrit comment utiliser le service pour télécharger et traiter les images.

modelX.html

`modelX.html` 4 pages expliquant le fonctionnement des différents modèles.

Fonctionnalités :

- Descriptions des modèles : Fournit des informations sur les différents modèles de traitement disponibles (Stable/StableX, Artistic/ArtistiX, BigColor/BigColorX, CNN).
- Les modèles avec un X sont les modèles pré-entraînés, les autres ont été entraînés par nos soins.

2.4 Design et Styles

Le fichier `styles.css` contient les styles pour l'ensemble des pages HTML, avec un design simple et intuitif. Les couleurs harmonisent avec le logo, et des Media Queries sont utilisées pour rendre le site responsive, assurant ainsi une expérience utilisateur agréable.

Principaux éléments de style :

- Boutons : Styles pour les boutons de téléchargement, de suppression, et de retour.
- Conteneurs d'images : Styles pour les conteneurs d'images et les images elles-mêmes.
- Messages d'erreur : Styles pour les messages d'erreur affichés aux utilisateurs.
- Slider de comparaison : Styles pour le slider permettant de comparer une image avant et après traitement.

2.5 Scripts JavaScript

scripts.js

`scripts.js` gère le téléchargement et l'affichage des images sur `site.html`, ainsi que la gestion du slider pour comparer une image avant et après traitement.

Fonctionnalités :

- Gestion des téléchargements d'images : Ajoute des écouteurs d'événements pour gérer le téléchargement et l'affichage des images et le redimensionne en 1024x1024px pour optimiser le transfert.
- Mise à jour du slider : Permet de comparer visuellement les images avant et après traitement.
- Gestion des erreurs : Affiche des messages d'erreur en cas de problème avec le téléchargement ou le traitement des images.

result.js

`result.js` gère l'affichage des images traitées sur `result.html` et leur téléchargement.

Fonctionnalités :

- Affichage des images traitées : Récupère les chemins des images traitées et les affiche sur la page.
- Téléchargement des images : Permet le téléchargement des images traitées via des boutons de téléchargement.

all_models_result.js

`all_models_result.js` gère l'affichage des images traitées par tous les modèles sur `all_models_result.html` et leur téléchargement.

Fonctionnalités :

- Affichage des images traitées par tous les modèles : Récupère les chemins des images traitées par tous les modèles et de l'image originale, puis les affiche sur la page avec le nom du modèle.
- Téléchargement des images : Permet le téléchargement des images traitées via des boutons de téléchargement.

2.6 Expérience Utilisateur (UX)

L'interface utilisateur est conçue pour être simple et intuitive, avec des éléments interactifs tels que des boutons et un slider pour améliorer l'expérience utilisateur.

Principales considérations UX :

- Navigation intuitive : Les utilisateurs peuvent facilement naviguer entre les différentes pages du site via un menu de navigation.
- Messages d'erreur clairs : Les messages d'erreur sont affichés en rouge de manière claire pour informer les utilisateurs de tout problème.
- Interface réactive : Le site est conçu pour être réactif, offrant une expérience utilisateur fluide et s'adapte en fonction des différents appareils et tailles d'écran.

2.7 Sécurité Frontend

Le frontend inclut des validations pour les entrées utilisateur et des protections contre les attaques XSS.

Mesures de sécurité :

- Validation des fichiers : Vérifie que les fichiers téléchargés sont bien des images avant de les envoyer au serveur avec le header du fichier '/image' et une liste d'extension valides.

- Protection contre les attaques XSS : Nettoie les entrées utilisateur pour limiter les attaques de type XSS.
- Gestion des erreurs : Affiche des messages d'erreur clairs et informatifs en cas de problème.

2.8 Optimisations et Bonnes Pratiques

Le code JavaScript est optimisé pour la performance, avec une manipulation efficace du DOM et une gestion des événements.

Principales optimisations :

- Manipulation efficace du DOM : Utilisation de techniques modernes pour manipuler le DOM de manière performante.
- Gestion des événements : Ajout d'écouteurs d'événements de manière optimale pour éviter les fuites de mémoire et améliorer la performance.
- Code propre et maintenable : Suivi des meilleures pratiques de développement frontend pour assurer un code propre et maintenable.

Le frontend du projet COLORIA est conçu pour offrir une expérience utilisateur agréable et intuitive. Il permet aux utilisateurs de télécharger des images, de sélectionner des modèles de traitement et de visualiser les résultats de manière interactive. Le code est optimisé pour la performance et la sécurité, et suit les meilleures pratiques de développement frontend.

Conclusion Site Web

Le projet COLORIA offre un service innovant de colorisation d'images en ligne, permettant aux utilisateurs de transformer leurs images en nuances de gris en versions colorisées de haute qualité. Ce rapport a détaillé les aspects techniques du site et du serveur, en se concentrant spécifiquement sur le backend et le frontend.

Backend

Le backend du projet, développé avec Node.js et Express.js, est responsable de la gestion des requêtes, du traitement des images téléchargées, et de la communication avec le frontend. Grâce à l'utilisation de middlewares pour la journalisation, la gestion des CORS, et le traitement des fichiers, le backend assure une gestion efficace et sécurisée des opérations. La validation des fichiers, la gestion des erreurs, et l'optimisation des performances garantissent une expérience utilisateur fluide et fiable.

Frontend

Le frontend, construit avec HTML, CSS, et JavaScript, offre une interface utilisateur intuitive et interactive. Les utilisateurs peuvent facilement télécharger des images, sélectionner des modèles de traitement, et visualiser les résultats grâce à des éléments interactifs tels que des boutons et un slider. Le design réactif et les messages d'erreur clairs contribuent à une expérience utilisateur agréable. Les scripts JavaScript optimisés assurent une manipulation efficace du DOM et une gestion performante des événements.

Réflexions Finales

Le projet COLORIA, avec son architecture bien définie et ses fonctionnalités robustes, démontre une approche efficace pour offrir des services de traitement d'images en ligne. Le backend performant et sécurisé, associé à un frontend intuitif et réactif, assure une expérience utilisateur de haute qualité. En suivant les meilleures pratiques de développement et en utilisant des technologies modernes, COLORIA est bien positionné pour évoluer et s'adapter aux besoins futurs.

Suggestions d'Amélioration :

Pour aller plus loin, plusieurs améliorations peuvent être envisagées :

- Ajout de nouvelles fonctionnalités : Intégrer des fonctionnalités supplémentaires, telles que des filtres d'image ou des options de personnalisation avancées.
- Optimisation des performances : Continuer à optimiser le code pour améliorer la vitesse de traitement et réduire les temps de chargement.
- Sécurité renforcée : Mettre en œuvre des mesures de sécurité supplémentaires pour protéger les données des utilisateurs et prévenir les attaques potentielles.
- Expérience utilisateur : Effectuer des tests utilisateurs pour identifier les points d'amélioration et rendre l'interface encore plus intuitive.

En conclusion, le projet COLORIA illustre une combinaison réussie de technologies backend et frontend pour offrir un service en ligne innovant et performant. Les bases solides posées par ce projet permettent d'envisager de nombreuses extensions et améliorations, assurant ainsi sa pérennité et son succès futurs.

Conclusion

Pour conclure ce rapport, le projet Coloria nous a permis de mettre les défis techniques du domaine de la colorisation d'images basé sur l'intelligence artificielle. Notre travail a été fait autour de la recherche, de l'analyse et de l'implémentation de différents modèles de Deep Learning pour la colorisation d'images en noir et blanc. Nous avons ainsi pu les entraîner, les comparer et les intégrer au sein d'un site web accessible à tous.

Au cours de nos recherches, nous avons d'abord défini les concepts clés de l'intelligence artificielle et du Deep Learning, éléments fondamentaux pour comprendre les mécanismes sous-jacents à la colorisation d'images. Nous avons ensuite procédé à une analyse des solutions existantes, en passant en revue des architectures telles que U-NET et ResNet-34/101. Cette exploration nous a conduit à sélectionner et expérimenter avec plusieurs modèles de pointe, à savoir DeOldify, KIMGEONUNG (BigColor), et le modèle de Rich ZHANG.

L'entraînement de ces modèles a révélé des avantages et des inconvénients spécifiques à chacun, nous permettant d'identifier les forces et les faiblesses en termes de performance et de qualité des résultats produits. Notre comparaison théorique a été complétée par une évaluation pratique, mettant en œuvre les différences dans les approches d'entraînement et l'impact sur la qualité de la colorisation.

La création d'un site web a été l'aboutissement de ce projet. Cette intégration web permet non seulement de démontrer concrètement les capacités des modèles mais aussi de les rendre accessibles à un public plus large.

En conclusion, ce projet a été une expérience enrichissante et formatrice. Il nous a permis de combiner nos connaissances théoriques avec une application pratique dans le domaine de l'intelligence artificielle. Les connaissances acquises tout au long de ce processus seront sans nul doute précieuses pour notre future carrière d'ingénieurs, en nous préparant à relever des défis techniques similaires et à contribuer activement à l'évolution des technologies basées sur l'IA.

IEEE

Image page de garde : généré par 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 Constraints" , 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"](https://hajim.rochester.edu/ece/sites/gsharma/ciede2000/ciede2000noteCRNA.pdf)
- [19] python fonction formule: <https://github.com/lovro-i/CIEDE2000>
- [20] images générées par les modèles:
["https://drive.google.com/drive/folders/13RwD5zHVGzV7nh0raskqAVlTqDJ_S67K"](https://drive.google.com/drive/folders/13RwD5zHVGzV7nh0raskqAVlTqDJ_S67K)