Compression d'image avec perte à base d'autoencodeurs

Romain Meynard
romain.meynard@student.ecp.fr
Venceslas des Déserts
venceslas.danguy-des-deserts@student.ecp.fr
Silvestre Perret
silvestre.perret@student.ecp.fr

CentraleSupélec, Avril 2018

Abstract

Le nombre d'images échangées sur Internet ne cesse d'augmenter, ainsi que leurs résolutions. La compression d'image redevient donc un sujet de recherche d'actualité. Les techniques d'apprentissage profond promettent des avancées dans ce domaine. Nous nous sommes intéressés aux autoencodeurs et en particulier à l'utilisation de fonctions de coût complexes (perceptuelle et texturale). Nous avons mesuré l'effet de l'utilisation de différentes fonctions de coût sur la qualité de la compression.

1. Introduction

Le problème de la compression d'image peut être vu comme deux problèmes distincts. Un premier algorithme, dit "de compression" doit prendre une image en entrée et retourner un fichier de poids inférieur. Un deuxième algorithme, dit "de décompression" doit pouvoir prendre ce fichier en entrée et retourner une image la plus semblable possible à l'image originale.

Il existe des techniques de compression sans perte, par exemple via un codage plus compact de l'information , ou avec perte, en supprimant une partie de l'information, l'image décompressée étant alors une version dégradée de l'image originale. Les approches avec perte présentent des taux de compression très supérieurs en contrepartie de la perte d'une partie de l'information contenue dans l'image. Pour ce type d'approche, l'enjeu clé est de sélectionner l'information la plus significative de l'image, i.e. celle qui permet d'obtenir une image décompressé peu dégradée. Des algorithmes tels que le JPEG mettent en œuvre une technique de compression avec perte avec des taux particulièrement intéressants. Les algorithmes de compression comportent généralement trois étapes : transformation, quantification et codage entropique.

Des travaux récents mettent en lumière les apports potentiels de l'apprentissage profond à la compression d'images. En effet le problème de la compression d'une image peut être vu comme l'apprentissage d'une fonction de compression et d'une fonction de décompression.

2. Etat de l'art

L'état de l'art de la compression d'image à base de réseaux profonds repose principalement sur des structures de réseaux récurrents (Toderici et al.)[7], d'autoencodeurs (Theis et al.)[6] et de réseaux adversariaux génératifs (Sajjadi et al.) (GANs) introduits par Goodfellow [1] . En effet, la structure de ces architectures en deux blocs opposés se prête particulièrement bien à la compression et décompression de l'image : le premier bloc aura pour charge de produire une version compressée de l'image que le deuxième bloc devra décompresser. En particulier Theis et al. ont développé un autoencodeur qui produit des résultats sur la compression comparable au JPEG 2000 [6]. Le problème de la compression d'image est également très proche du problème de super résolution qui consiste à augmenter la résolution d'une image. Dans ce domaine Sajjadi et al. ont produit EnhanceNet [4] en 2016, un GAN qui génère une version agrandie quatre fois d'une image initiale. Ce modèle repose en particulier sur la génération automatique de texture par le réseau et sur la prise en compte de la sémantique. Notre intuition est que certaines approches de super résolution peuvent être transposées à la compression d'image, en particulier la génération de texture.

3. Modèle

Notre travail repose principalement sur l'article [6] de Theis et al. dans lequel un autoencoder est utilisé.

Nous souhaitions mesurer les effets de la prise en compte de la perception et des textures. En effet ces fonctions sont utilisés dans l'état de l'art et les auteurs suggèrent ces pistes d'amélioration dans leur article.

3.1. Architecture

L'architecture de notre réseau est tirée de l'article de Theis et al. [6] qui propose une architecture d'autoencodeur pour la compression.

3.1.1 Les autoencodeurs

Les autoencodeurs sont un type particulier de réseaux de neurones multicouches ayant pour but d'apprendre une représentation, généralement de dimension réduite, de l'entrée. La sortie d'un autoencodeur est donc une reconstruction de l'image d'entrée à partir de cette représentation, le but étant d'avoir la reconstruction la plus fidèle possible. Pour éviter que l'autoencodeur n'apprenne la fonction identité, il faut que les couches cachées soient de taille inférieure à la couche d'entrée, ce qui oblige le réseau à "résumer" l'information.

3.1.2 Notre architecture

L'architecture proposée dans le papier de Theis et al. [6] est un autoencodeur convolutionnel. Il comporte des blocs résiduels tels que proposés par He et al. [2] en 2015. La spécificité de ce réseau tient de ce qu'il comporte des étapes non dérivables : une étape d'arrondi entre l'encodeur et le décodeur qui permet d'appliquer un codage de Huffman, et une étape de *clipping* en sortie qui permet d'avoir une sortie dont toutes les valeurs sont comprises entre 0 et 255, c'est à dire une sortie correspondant à une image.

Afin de pouvoir entraîner ce réseau, il est nécessaire de choisir une fonction faisant office de dérivée pour ces couches particulière. En pratique, l'article indique que la fonction identité fonctionne bien, tant pour des questions de facilité d'implémentation que pour des raisons de préservation du gradient.

Contrairement à l'article qui fait une normalisation sur la distribution des valeurs de chaque canal d'entrée, nous divisons simplement la valeur de tous les pixels d'entrée par 255 de façon à ce que le tenseur d'entrée prenne des valeurs comprises entre 0 et 1. Cette version simplifiée de normalisation suffit en pratique.

La compression est réalisée avec des couches

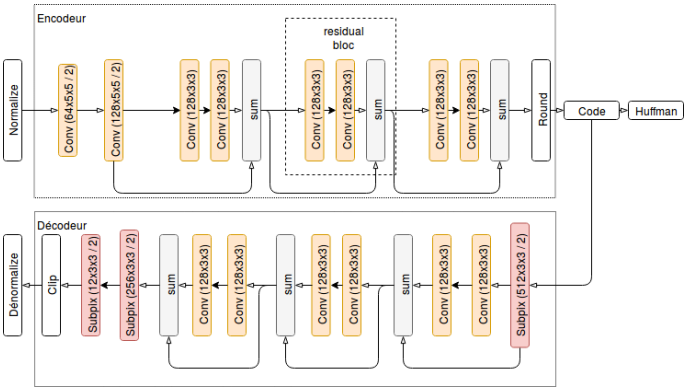

FIGURE 1. Architecture de notre modèle

convolutives de stride (pas) de 2 ce qui a pour effet de diviser la taille de la carte d'activation par 2. L'encodeur comporte également 3 blocs résiduels pour extraire l'information de l'image. Au lieu d'utiliser une couche de Mirror Padding et des paddings de type *valid* dans les convolutions comme le fait l'auteur, nous avons choisi d'utiliser simplement des convolutions avec un padding de type *same*. La réduction de dimension est alors uniquement assurée par des convolutions de stride 2 en entrée et en sortie de l'encodeur.

Le sur-échantillonnage dans le décodeur se fait via une opération de *subpixel* décrite par Shi et al. [5]. Elle consiste en une réorganisation des coefficients transformant un tenseur de taille $[x, y, z]$ en un autre tenseur de taille $[x \times k, y \times k, \frac{z}{k^2}]$, où k est un paramètre qui pilote la dilatation de la taille de la carte d'activation. Dans notre cas $k = 2$, par conséquent la carte d'activation est doublée à chaque opération de *subpixel* ce qui a l'effet inverse des couches de convolution avec une stride de 2.

L'architecture est visible sur la figure 1.

3.2. Codage entropique

En sortie de l'encodeur, une couche permet d'arrondir les activations à la valeur entière la plus proche. Cette quantification permet d'appliquer une technique de compression sans perte, également utilisée dans JPEG : l'encodage de Huffman. Il s'agit d'une technique de codage entropique, i.e. trouver la manière la plus économe en espace de coder la matrice obtenue après quantification.

On associe à chaque valeur différente présente dans la matrice un code, Les codes plus courts sont associés aux symboles les plus fréquents pour optimiser la taille du code.Puis on recode la matrice avec ce nouvel encodage. La matrice encodée est plus légère et peut être envoyé sur le réseau plus facilement. Pour pouvoir décoder la nouvelle matrice, il faut également transmettre la table de conversion.

3.3. Fonctions de coût

Nous utilisons trois fonctions de coût prenant en compte des similarités différentes :

— Similarité entre pixels
— Similarité entre features
— Similarités entre textures

Ainsi qu'une quatrième fonction de coût prenant en compte le taux de compression c'est à dire le poids nécessaire au transfert et au stockage de la représentation apprise.

3.3.1 Similarité entre pixels

La similarité entre pixel est implémenté à l'aide de l'erreur quadratique moyenne (Mean Squared Error - MSE) entre les images :

$$MSE(X, f(X))$$

3.3.2 Similarité perceptuelle

La fonction de coût perceptuelle vise à contrôler l'écart entre les images dans l'espace perceptuel, l'idée étant que plus qu'une ressemblance au niveau des pixels, on cherche une ressemblance au niveau de la sémantique de l'image. Or il s'avère que des réseaux convolutifs tels que VGG sont déjà entraînés à convertir une image dans l'espace perceptuel. Suivant cette idée, on peut implémenter une fonction de coût :

$$MSE(\phi_i(X), \phi_i(f(X)))$$

Où ϕ_i est la fonction qui renvoie la carte d'activation de la couche i d'un réseau de type VGG. En pratique, on prend la carte d'activation en sortie du bloc 2 et du bloc 5.

3.3.3 Similarité entre textures

La fonction de coût de texture telle qu'elle est introduite par [4] dans EnhanceNet vise à évaluer la similarité de texture sur des patchs des images.

$$MSE(Gram(\phi_i(X)), Gram(\phi_i(f(X))))$$

On utilise la matrice de $Gram$ appliquée aux mêmes sorties d'un VGG que précédemment.

$$Gram(F) = FF^T \in \mathbb{R}^{n \times n}$$

La matrice de Gram permet de calculer les corrélations entre les canaux de la carte d'activation du VGG. Elle s'applique sur des patchs de l'image de taille 8×8.

En effet, comme expliqué dans [6], si on l'appliquait sur toute l'image, on obtiendrait une sorte de représentation globale de toutes les textures présentes dans l'image, or nos images contiennent des textures variées. Afin de pouvoir générer différentes textures correspondant à celles que l'on avait en entrée, on effectue ce traitement patch par patch, l'idée étant que la taille du patch doit être bien choisie afin de pouvoir représenter une texture mais pas plusieurs.

Afin de pouvoir faire passer chaque patch dans le VGG lors de l'apprentissage, nous avons créé des *custom layers* Keras qui modifient le tenseur de données pour calculer les représentations des textures des patchs. En pratique, cela consiste à extraire les patchs de l'image via la fonction dédiée de Tensorflow puis à reformater le tenseur de sorte qu'il passe d'une taille *(batch_size, input_size_1, input_size_2, num_channels)* à une taille *(batch_size*num_patches_per_image, patch_size_1, patch_size_2, num_channels)*. Ce format permet de mettre le tenseur transformé directement en entrée du VGG, qui le traite comme s'il s'agissait d'un batch d'images classique. Une fois les features obtenues, on les mets sous la forme *(batch_size, num_patches_per_image, vgg_output_size_1, vgg_output_size_2, vgg_output_num_channels)* de sorte que pour chaque image, pour chaque patch, on a la sortie du VGG sur ce patch. On effectue ce même travail sur les images d'entrée, et on calcule la *Mean Squared Error* entre les deux.

3.3.4 Coût de stockage

Pour pénaliser la taille du codage de l'image, nous avons implémenté une fonction de coût qui mesure l'entropie du code.

$$Entropie = \sum p \times \log(p)$$

Le code résultant de l'image est généré après la couche de *Rounding*. À ce stade le nombre de paramètres du vecteur est divisé par deux par rapport à l'image originale. Nous avons alors étudié l'ajout de cette fonction de coût pour contraindre le réseau a utiliser moins d'information pour le codage de l'image. De plus le code de Huffman que nous utilisons ensuite est efficace seulement si l'entropie de l'image est faible.

4. Expériences

Faute d'implémentation existante pour notre cas d'usage, nous avons implémenté l'architecture en Keras et Tensorflow.

Le dataset utilisé est un sous-ensemble de Celeba, un dataset de visages rassemblés par Liu et al. [3]. Contrairement à Theis et al.[6], nous avons fait le choix de simplifier le problème en prenant un jeu de données spécifique. Le dataset comporte plus de 200 000 images de taille 210×178, mais nous nous sommes limités à un ensemble de 6 000 images redimensionnées en 64x64 pour l'exploration des hyperparamètres

pour des raisons de temps de calcul. En revanche nous avons pu faire une expérience avec 2000 images en 128x128 pour observer la qualité de la reconstitution.

Le dataset d'images 64×64 est divisé comme suit :

— Entraînement : 4200 images
— Validation : 1200 images
— Test : 600 images

Le dataset d'images 128×128 est divisé comme suit :

— Entraînement : 1400 images
— Validation : 400 images
— Test : 200 images

4.1. Entraînement

Dans notre modèle nous avons 5 fonctions de coût différentes et qui ont des comportements antagonistes. En effet une partie des fonctions de coût pénalisent l'erreur entre la prédiction et la vérité terrain tandis que la fonction d'entropie pénalise le nombre de valeurs uniques utilisées dans le code de l'image. Ce comportement s'observe également dans l'entraînement comme le montre la figure 2.

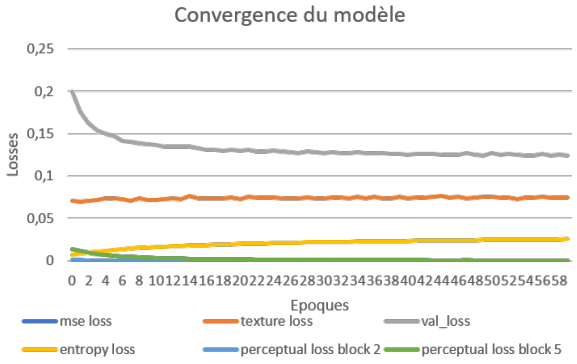

FIGURE 2. Évolution des fonctions de coût pendant l'entraînement

Sur le graphe on voit que la fonction de coût *MSE* converge rapidement ainsi que les fonctions de coûts perceptuelles. En revanche on observe que la fonction de coût entropique augmente. On peut l'expliquer par le fait que le réseau apprend : au début la couche de sortie de l'encodeur (rounding) ne contient que des zéros dus à l'initialisation. Le réseau va ensuite tenter de produire un code permettant de reconstruire l'image. Ce faisant, il utilise de plus en plus de symboles, ce qui fait augmenter l'entropie du code et provoque la hausse de la fonction de coût d'entropie.

Le modèle doit donc trouver un compromis entre le nombre de symboles utilisés dans le code et la qualité de la reconstruction. Cela nous impose également des entraînements plus long avant d'avoir une stabilité des fonctions de coûts.

Au niveau des temps d'entraînement et de la capacité mémoire, il est important de noter que l'ajout de la fonction de coût de texture impose la génération de patchs qui sont envoyés à travers un réseau VGG. Cela rallonge considérablement le temps de calcul et impose de réduire la taille des batchs (au plus 16 pour des images de 64x64 pixels sur une GPU Tesla K80). Nous avons utilisé des instances Azure et AWS pour pouvoir avoir des temps d'entraînement corrects.

5. Résultats

5.1. Exploration des coefficients des fonctions de coût

Nous avons cherché quels étaient les meilleurs poids à donner aux différentes fonctions de coût. Notre choix s'est finalement arrêté sur des poids produisant des valeurs proches de, mais légèrement inférieures à, la fonction de coût principale (similarité entre pixels), afin d'être certain de privilégier une reconstruction fidèle à l'image d'origine. Nous avons ensuite exploré à partir de cette référence en essayant de désactiver certains fonctions de coûts (poids nul) et d'explorer différentes valeurs de poids.

Nous nous sommes servis de 3 mesures afin de juger de la performance d'un modèle. La MSE entre image reconstruite et image originale, le peak signal-to-noise ratio ($PSNR$) qui estime la ressemblance au niveau signal entre l'image

d'entrée et l'image reconstruite, et le bit-per-pixel (BPP) qui donne le nombre moyen de bits nécessaire pour coder un pixel de l'image.

L'entraînement étant assez long, nous avons choisi de tester différentes configurations de poids sur 30 époques. Un tel entraînement pour une configuration prend environ une heure sur une instance AWS munie d'une Tesla K80. A partir de ces premiers résultats, nous avons choisi les paramètres donnant, sur l'ensemble de validation, les meilleurs résultats, et avons prolongé l'entraînement jusqu'à 60 époques. Nous ne présentons ci-dessous que les expériences que nous avons jugées prometteuses et menées jusqu'à 60 époques.

5.2. Configuration des expériences

À travers nos expériences nous avons fait varier les coefficients des fonctions de coût pour les optimiser.

No	Entr	MSE	Percep2	Percep5	Texture
1	0	1	0	0	0
2	0.01	1	10^{-4}	0.1	0
3	0.01	0.1	10^{-4}	0.01	0
4	0.01	1	10^{-4}	0.1	10^{-6}

Autres hyperparamètres :

— Optimizer : Adam avec $lr = 10^{-4}$ et un clip de norme
— Early stopping avec un delta minimum de 10^{-4} et une patience de 20
— 60 époques
— Taille de batch : 16

5.3. Résultats des expériences

No	MSE	PSNR	BPP
1	93.5	29.1	0.340
2	258.2	24.2	0.196
3	961.3	18.4	0.113
4	98.6	28.9	0.346

Au travers des expériences nous nous sommes aperçus que l'expérience avec uniquement la *MSE* avait les meilleurs résultats sur un entraînement de 60 époques. Nous pensons que le fait d'ajouter des fonctions de coûts complique l'apprentissage. Il faudrait peut-être entraîner sur plus d'itérations pour que le réseau trouve un optimum.

On peut également observer que la fonction de coût d'entropie résulte en un bpp (bit per pixel) plus faible dans l'expérience 3. Cette fonction de coût produit donc bien l'effet souhaité mais au dépend d'une forte dégradation de la qualité.

Dans l'expérience 4, on introduit la fonction de coût de texture. On observe alors des résultats très proche de l'expérience 1 quoiqu'un peu plus lisses : il faudrait probablement jouer sur la taille de patch ou le coefficient de la fonction de coût.

Nous avons également souhaité observer les résultats sur des images de plus grande taille (128x128). Dans ce cas le training prend une durée d'environ une journée ce qui a limité nos expériences. On observe néanmoins un rendu visuel de bonne qualité sur les photos.

5.4. Rendu des images décompressées

Voir les figures 3 et 4.

6. Discussion

Les résultats obtenus sont très dépendants des mesures utilisées pour noter la performance des modèles. Ainsi notre choix de la MSE et du $PSNR$ n'est pas sans conséquence. Il est possible que les modèles incluant des fonctions de coûts plus complexes (comme la fonction de coûts de texture) aient obtenu de meilleurs résultats si nous avions pu intégrer un mesure de fidélité plus globale, avec par exemple les appréciations d'une groupes d'arbitres humains.

Un autre point clé qui a pu favoriser les modèles dominés par la fonction de coût MSE, est le faible nombre d'époques durant nos entraînements. Nous avons fait nos différentes explorations sur moins de 100 époques, là où l'article[6] de Theis et al. montre des modèles entraînés beaucoup plus longuement. Nos observations montrent que le réseau converge beaucoup plus vite lorsqu'il n'est soumis qu'à une

(a) Truth (b) Truth (c) Truth

(d) No 1 Pred (e) No 1 Pred (f) No 1 Pred

(g) No 2 Pred (h) No 2 Pred (i) No 2 Pred

(j) No 3 Pred (k) No 3 Pred (l) No 3 Pred

(m) No 4 pred (n) No 4 pred (o) No 4 pred

FIGURE 3. Images 64x64

(a) Original (b) Prédiction

FIGURE 4. Image 128x128

seule fonction de coût de type MSE. Ainsi les réseaux utilisant les autres fonctions de coûts n'ont toujours pas fini de converger à la fin de l'entraînement du modèle.

Une autre conséquence du faible nombre d'époques est le très fort taux de compression (de l'ordre de 30) obtenu même sur les moins bons modèles. La fonction de coût d'entropie, conçue pour limiter la taille de fichier a une influence très forte au début de l'apprentissage et les différents réseaux testés n'ont pas le temps d'atteindre une stabilisation de la fonction de coût d'entropie, favorisant ainsi une compression importante, au détriment de la fidélité de la reconstruction obtenue.

Références

[1] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative Adversarial Networks. *arXiv :1406.2661 [cs, stat]*, June 2014. arXiv : 1406.2661.

[2] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. *arXiv :1512.03385 [cs]*, Dec. 2015. arXiv : 1512.03385.

[3] Z. Liu, P. Luo, X. Wang, and X. Tang. Deep Learning Face Attributes in the Wild. *arXiv :1411.7766 [cs]*, Nov. 2014. arXiv : 1411.7766.

[4] M. S. M. Sajjadi, B. Schölkopf, and M. Hirsch. EnhanceNet : Single Image Super-Resolution Through Automated Texture Synthesis. *arXiv :1612.07919 [cs, stat]*, Dec. 2016. arXiv : 1612.07919.

[5] W. Shi, J. Caballero, F. Huszar, J. Totz, A. P. Aitken, R. Bishop, D. Rueckert, and Z. Wang. Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network. pages 1874–1883. IEEE, June 2016.

[6] L. Theis, W. Shi, A. Cunningham, and F. Huszár. Lossy Image Compression with Compressive Autoencoders. *arXiv :1703.00395 [cs, stat]*, Mar. 2017. arXiv : 1703.00395.

[7] G. Toderici, D. Vincent, N. Johnston, S. J. Hwang, D. Minnen, J. Shor, and M. Covell. Full Resolution Image Compression with Recurrent Neural Networks. pages 5435–5443. IEEE, July 2017.