

Compresser pour comprendre

Le goulot, l'espace latent et ce que l'on peut en tirer

goulot

espace latent

ACP

débruitage

VAE

diffusion

Forcer une image à passer par huit nombres, puis regarder ce qui a survécu au passage.

Un cadeau de fin de cours, à lire par curiosité : rien n'est à rendre. Le projet « Autoencodeur » reste décrit par sa fiche sur le site et par le README de son squelette ; ce fascicule raconte les idées qui l'entourent et ce qu'elles deviennent au-delà du cours.

1 Pourquoi ce fascicule

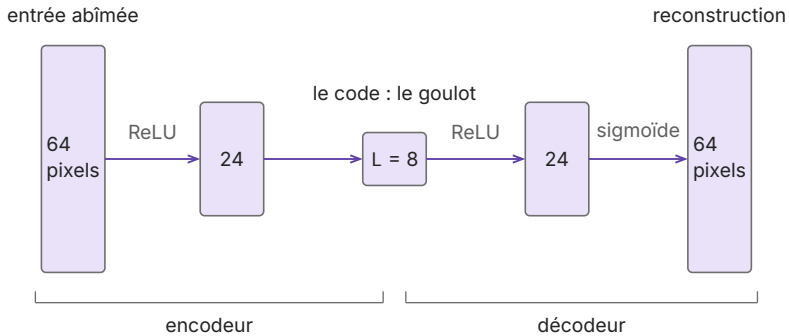
Dans le projet « Autoencodeur », vous écrivez avec Nx un encodeur qui ramène une image de huit pixels par huit à un code de quelques nombres, un décodeur qui tente de revenir aux pixels, une corruption qui efface des positions tirées au sort, et deux erreurs pour mesurer ce que le réseau répare. Le prolongement remplace le code par une distribution : c'est l'autoencodeur variationnel. Tout cela tient dans un projet Mix, sur CPU, avec les briques d'AP4 et d'AP5.

Ce fascicule raconte ce qu'il y a autour. D'où vient l'idée d'apprendre à recopier son entrée, et pourquoi elle n'a d'intérêt que si l'on empêche la copie ? Que calcule exactement un autoencodeur quand il est linéaire ? Que trouve-t-on dans l'espace latent, et pourquoi une interpolation entre deux codes produit-elle parfois des images qui n'existent pas ? Pourquoi le VAE remplace-t-il un point par une distribution, et comment dérive-t-on à travers un tirage au sort ? Enfin, comment le simple débruitage de 2008 est-il devenu, en 2020, le moteur des générateurs d'images actuels ?

Vous n'y trouverez aucune solution du projet : ni les cinq fonctions du socle, ni les deux du VAE. Les exemples de code sont de petits calculs voisins, écrits en Elixir sans dépendance, dont la sortie a été vérifiée. Rien n'est à rendre ; lisez ce qui vous intrigue.

2 Le goulot, ou ce qu'une copie contrainte apprend

Un *autoencodeur* est un réseau entraîné à reproduire son entrée. Il est fait de deux fonctions apprises ensemble : l'*encodeur* f envoie l'entrée x , ici 64 pixels, vers un *code* $c = f(x)$ de L nombres ; le *décodeur* g tente de revenir aux pixels, $\hat{x} = g(c)$. La perte est la MSE d'AP5 entre la reconstruction $\hat{x}$ et l'image : rien de nouveau dans la boucle d'entraînement, le gradient et Adam font le même travail qu'en séance. La seule nouveauté est que la cible est l'entrée elle-même : aucune étiquette n'est nécessaire.



L'architecture du projet. Les largeurs des colonnes suivent le nombre de valeurs à chaque étape : 64, 24, $L = 8$, 24, 64. L'encodeur est la moitié gauche, le décodeur la moitié droite ; le code est l'endroit le plus étroit.

Posons-nous la question naïve : si L vaut 64, qu'apprend le réseau ? Il peut recopier l'entrée, et c'est ce qu'il fait de mieux : perte nulle, représentation inutile. Avec $L = 8$, la copie exacte d'une entrée quelconque devient impossible : huit nombres, manipulés par des couches régulières, ne peuvent pas restituer 64 valeurs arbitraires. Le réseau doit donc choisir ce qu'il garde, et le critère de ce choix est la perte : il conserve ce qui, en moyenne sur les images d'apprentissage, permet de reconstruire le plus de pixels. C'est le *goulot*, en anglais *bottleneck*, qui transforme une tâche de copie en une tâche de représentation.

Notre corpus rend ce choix presque lisible. Les 128 images viennent de huit glyphes placés à seize positions : pour désigner une image, il suffit de dire lequel des huit et laquelle des seize, soit sept bits d'information. Le réseau n'en sait rien ; il reçoit seulement des pixels. S'il trouve des coordonnées qui séparent les glyphes et qui suivent la position, le décodeur n'a plus qu'à redessiner. S'il n'y parvient que partiellement, les reconstructions mélangent deux glyphes ou tremblent d'une colonne. Les deux cas s'observent dans *reconstructions.svg*, et c'est précisément ce qui rend l'expérience intéressante.

PROPRIÉTÉ

Deux entrées qui reçoivent le même code reçoivent la même reconstruction. Tout ce que le décodeur doit distinguer doit donc avoir survécu à l'encodeur. Pourquoi c'est vrai : le décodeur est une fonction du code seul ; il ne voit ni l'entrée ni le masque. Chaque fois que vous examinez une reconstruction ratée, demandez-vous si l'information manquante pouvait encore être dans huit nombres.

2.1 Pourquoi les reconstructions sont grises

Un phénomène surprend souvent : plutôt que de dessiner un trait net au mauvais endroit, le réseau dessine deux traits pâles. Est-ce un défaut d'entraînement ? Non, c'est la MSE qui le demande. Prenons un trait d'un pixel qui, pour tout ce que le code en sait, se trouve en colonne 3 ou en colonne 4, à égalité. Si le décodeur parie sur la colonne 3, il a raison une fois sur deux, et l'autre fois il se trompe sur deux pixels : l'erreur quadratique attendue vaut $\frac{1}{2} \times 0 + \frac{1}{2} \times 2 = 1$. S'il dessine les deux colonnes à mi-gris, 0,5, il se trompe d'un demi-pixel sur chacune, à tous les coups : l'erreur attendue vaut $0,25 + 0,25 = 0,5$. Le gris gagne ! La MSE récompense la moyenne des possibilités, et un décodeur entraîné avec elle produit des images floues dès qu'il hésite. Retenez ce calcul : il explique aussi pourquoi les générateurs d'images modernes ont dû changer de perte, comme nous le verrons à la section 7.

PIÈGE

Avec `--noise 0.0`, la tâche est la reconstruction d'images intactes ; avec `--noise 0.35`, c'est la réparation. Une courbe plus basse dans le premier cas ne dit rien sur l'architecture : les deux problèmes n'ont pas la même difficulté. Comparez des recettes qui ne changent qu'un choix.

3 Un autoencodeur linéaire retrouve l'analyse en composantes principales

Retirons un instant les activations : l'encodeur et le décodeur deviennent deux matrices, et la reconstruction est une fonction linéaire de l'entrée. Ce cas se calcule entièrement, et le résultat éclaire le cas général.

Rappelons l'*analyse en composantes principales*, ou ACP, que vous avez peut-être rencontrée en statistiques : étant donné un nuage de points centré, on cherche la direction le long de laquelle les points s'étalent le plus, c'est-à-dire celle de variance maximale ; puis, parmi les directions orthogonales à la première, la suivante, et ainsi de suite. Par le théorème de Pythagore, la direction de variance maximale est aussi celle qui minimise la somme des carrés des distances entre chaque point et sa projection orthogonale. En clair, projeter sur les L premières directions est la meilleure façon de résumer le nuage par L coordonnées, au sens des moindres carrés.

Déroulons un exemple à la main, avec des images réduites à deux pixels : quatre points, $(2, 1)$, $(1, 2)$, $(-2, -1)$ et $(-1, -2)$, de moyenne nulle. Les sommes des carrés

valent $\sum x^2 = 10$, $\sum y^2 = 10$ et $\sum xy = 8$. Le long de la diagonale, de vecteur unitaire $(1, 1)/\sqrt{2}$, la coordonnée d'un point est $(x + y)/\sqrt{2}$ et la somme des carrés de ces coordonnées vaut $(10 + 10 + 16)/2 = 18$; le long de l'anti-diagonale $(1, -1)/\sqrt{2}$, elle vaut $(10 + 10 - 16)/2 = 2$. La diagonale est la direction principale. Un code à un seul nombre, la coordonnée le long de cette diagonale, reconstruit $(2, 1)$ en $(1,5 ; 1,5)$ avec une erreur de $(0,5 ; -0,5)$, de carré 0,5 ; c'est la même erreur pour les quatre points. Garder plutôt la coordonnée x reconstruirait $(2, 1)$ en $(2, 0)$, avec une erreur de carré 1.

Le petit programme suivant refait ce calcul pour trois directions : l'axe des x , la diagonale et l'anti-diagonale. Il n'utilise que des listes.

```

1 points = [{2.0, 1.0}, {1.0, 2.0}, {-2.0, -1.0}, {-1.0, -2.0}]
2
3 # erreur moyenne quand le code est la coordonnée
4 # le long d'une direction unitaire {dx, dy}
5 erreur = fn {dx, dy} ->
6   points
7   |> Enum.map(fn {x, y} ->
8     code = x * dx + y * dy
9     (code * dx - x) ** 2 + (code * dy - y) ** 2
10  end)
11   |> Enum.sum()
12   |> Kernel./(length(points))
13 end
14
15 r = :math.sqrt(0.5)
16 IO.inspect(erreur.({1.0, 0.0}), label: "axe des x")
17 IO.inspect(erreur.({r, r}), label: "diagonale")
18 IO.inspect(erreur.({r, -r}), label: "anti-diagonale")

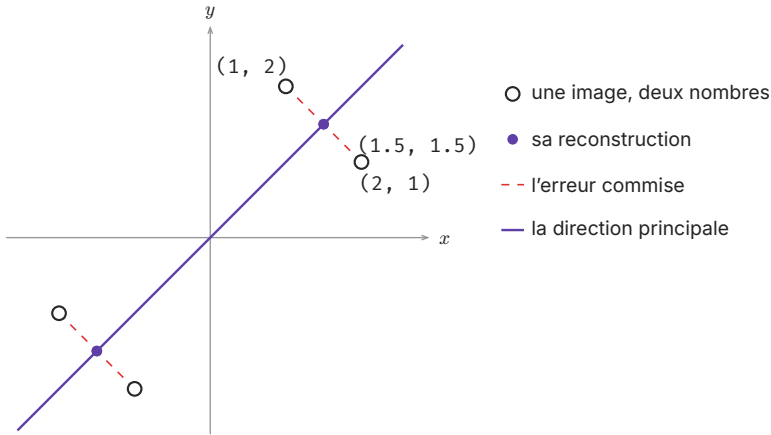
```

```

CONSOLE  axe des x: 2.5
         diagonale: 0.5
         anti-diagonale: 4.5

```

Remarquez que l'erreur de la diagonale, 0,5, est exactement la variance qui restait le long de l'anti-diagonale, $2/4$: ce que le goulot perd, c'est l'étalement des directions abandonnées. La figure ci-dessous montre les quatre points, la diagonale et les erreurs commises.



Quatre « images » de deux pixels et leur reconstruction par un code d'un seul nombre, la coordonnée le long de la direction principale. Les pointillés mesurent ce que le goulot a perdu.

Voici maintenant le résultat qui relie ce calcul aux réseaux.

PROPRIÉTÉ (BALDI ET HORNIK, 1989)

Pour un autoencodeur dont l'encodeur et le décodeur sont linéaires, entraîné avec la MSE, la perte atteint son minimum exactement lorsque la composition décodeur après encodeur est la projection orthogonale sur le sous-espace des L premières composantes principales. La perte minimale est la somme des variances des directions abandonnées, et il n'existe aucun autre minimum local : les autres points critiques sont des cols, que la descente de gradient traverse. Pourquoi c'est vrai, en esquisse : la composition de deux applications linéaires $64 \rightarrow L \rightarrow 64$ est une matrice de rang au plus L , et parmi les matrices de rang L , celle qui approche le mieux les données au sens des moindres carrés est la projection sur les L premières directions principales, un théorème d'Eckart et Young de 1936. Le travail de Baldi et Hornik est d'avoir montré que le paysage de la perte ne cache pas de faux minimum.

Le code du goulot n'est pas forcément la liste des composantes principales elles-mêmes : toute base du même sous-espace convient, et l'entraînement en choisit une au hasard. En revanche, la reconstruction, elle, est unique. Pouvez-vous prédire ce que la descente de gradient va trouver sur nos quatre points ? Le programme suivant entraîne un autoencodeur linéaire à code d'un nombre, avec un encodeur w et un

décodeur v de deux poids chacun, par la règle de mise à jour d'AP5 : trois cents pas, taux 0,05.

```

1 points = [{2.0, 1.0}, {1.0, 2.0}, {-2.0, -1.0}, {-1.0, -2.0}]
2 n = length(points)
3
4 # un pas de descente : code = w . x, reconstruction = code * v
5 pas = fn {{w1, w2}, {v1, v2}}, taux ->
6   {gw1, gw2, gv1, gv2, perte} =
7     Enum.reduce(points, {0.0, 0.0, 0.0, 0.0, 0.0},
8       fn {x, y}, {a, b, c, d, l} ->
9         code = w1 * x + w2 * y
10        {rx, ry} = {code * v1 - x, code * v2 - y}
11        s = rx * v1 + ry * v2
12        {a + 2 * s * x / n, b + 2 * s * y / n,
13         c + 2 * code * rx / n, d + 2 * code * ry / n,
14         l + (rx * rx + ry * ry) / n}
15      end)
16    {{w1 - taux * gw1, w2 - taux * gw2},
17     {v1 - taux * gv1, v2 - taux * gv2}}, perte}
18 end
19
20 depart = {{0.3, -0.1}, {0.2, 0.4}}
21 {{_w, {v1, v2}}, perte} =
22   Enum.reduce(1..300, {depart, nil},
23     fn _, {params, _} -> pas.(params, 0.05) end)
24 norme = :math.sqrt(v1 * v1 + v2 * v2)
25 IO.inspect({Float.round(v1 / norme, 3), Float.round(v2 / norme, 3)},
26   label: "direction du décodeur")
27 IO.inspect(Float.round(perte, 4), label: "perte")

```

```

CONSOLE direction du décodeur: {0.707, 0.707}
perte: 0.5

```

Le décodeur a trouvé la diagonale, à la précision de l'affichage, et la perte est bien 0,5. Les dérivées écrites dans `pas` sont celles de la règle de la chaîne du chapitre 8 ; dans le projet, c'est `defn` et `grad` qui les calculent à votre place.

Que change la ReLU du projet ? Elle rend la reconstruction non linéaire : le réseau peut alors plier le sous-espace et suivre une surface courbe dans l'espace des pixels, là où l'ACP ne sait tracer qu'un plan. Pour des glyphes qui se déplacent, c'est décisif : traduire un trait n'est pas une opération linéaire sur les pixels. Mais le résultat

linéaire reste la bonne référence : si votre autoencodeur ne fait pas mieux que l'ACP à même L , ses activations ne servent à rien.

4 Une idée des années 1980, relancée en 2006

L'idée de faire passer de l'information par un étranglement est aussi vieille que la rétropropagation. Dans le chapitre de 1986 où Rumelhart, Hinton et Williams décrivent cet algorithme, un des problèmes d'essai est le « problème du codage », posé par Ackley, Hinton et Sejnowski en 1985 : huit motifs d'entrée doivent être reproduits en sortie après avoir traversé trois unités cachées. Le réseau doit inventer seul un code sur trois unités, un par motif ; le chapitre note même que la solution trouvée utilise des valeurs intermédiaires, et pas seulement 0 et 1. C'est déjà notre sablier, avec $L = 3$ et huit images.

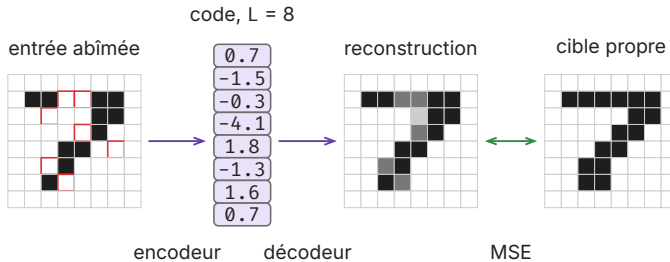
La théorie a suivi de près. Bourlard et Kamp, en 1988, puis Baldi et Hornik, en 1989, établissent le résultat de la section précédente : avec une seule couche cachée linéaire, l'autoencodeur ne fait rien de plus que l'ACP, et l'ajout d'une activation sur cette unique couche n'y change rien. Le résultat a souvent été lu comme une limite : à quoi bon un réseau, si l'ACP fait aussi bien ?

La relance vient de Hinton et Salakhutdinov, dans *Science* en 2006. Ils entraînent un autoencodeur profond sur les chiffres manuscrits de MNIST, avec des couches de 1 000, 500 et 250 unités jusqu'à un code de 30 nombres, puis le chemin inverse, soit plusieurs millions de paramètres. Le secret n'est pas l'architecture, mais la façon de l'initialiser : chaque couche est d'abord entraînée seule, comme une petite machine de Boltzmann, avant que l'ensemble soit affiné par rétropropagation. Avec un code de deux nombres, le dessin de l'espace latent sépare les dix chiffres bien plus nettement que les deux premières composantes de l'ACP. Cet article est l'un des points de départ de ce qu'on a appelé ensuite l'apprentissage profond.

En 2008, Vincent, Larochelle, Bengio et Manzagol proposent, à la conférence ICML, l'*autoencodeur de débruitage* : on abîme l'entrée et on demande l'image propre. Leur corruption de référence est exactement celle du projet, le *masquage* qui met à zéro une fraction des entrées tirée au sort. L'intérêt est double. D'une part, la copie devient impossible même sans goulot étroit, puisque l'entrée ne contient plus tout ce qu'il faut restituer. D'autre part, pour remplir un trou, le réseau doit utiliser les pixels voisins, et les régularités qu'il découvre ainsi sont plus robustes que celles d'une simple compression. Les auteurs empilent ensuite ces couches pour construire des réseaux profonds, dans un article de 2010.

À RETENIR

L'autoencodeur n'est pas né en 2006, mais c'est en 2006 que l'on a montré qu'un réseau profond pouvait apprendre une représentation meilleure que l'ACP, et en 2008 que le débruitage était une façon d'apprendre sans étiquettes. Votre projet reproduit ces deux idées à petite échelle.



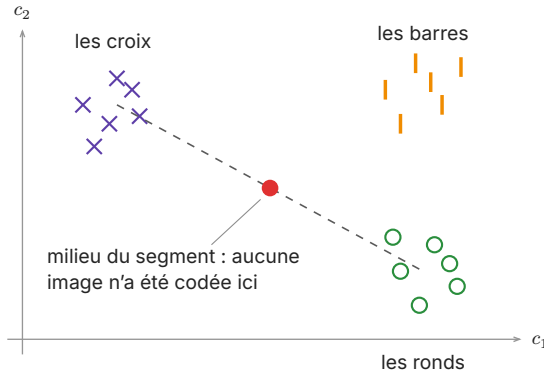
Le débruitage dans le projet. Les cases cerclées de rouge sont des pixels mis à zéro par la corruption ; le réseau ne voit que des zéros, sans savoir lesquels sont des trous. Le décodeur dessine en gris ce dont il n'est pas sûr. La cible propre ne sert qu'à mesurer l'erreur.

Le masque de la figure comprend une case qui était déjà blanche : le tirage décrit ce qui a été sélectionné, pas seulement ce qui a changé. La reconstruction montre deux comportements que vous retrouverez : des trous remplis en gris, et un pixel voisin légèrement modifié alors qu'il était intact. C'est pour cela que le projet mesure deux erreurs, sur tous les pixels et sur les seules positions masquées : un réseau qui restaure des traits tout en abîmant un peu le reste peut faire mieux que l'entrée abîmée sur la seconde et moins bien sur la première. Une seule courbe ne raconte pas cette histoire.

5 L'espace latent : interpoler, et pourquoi cela peut dérailler

L'ensemble des codes possibles est l'*espace latent* : avec $L = 8$, un espace à huit dimensions dans lequel chaque image devient un point. Le fichier `codes.csv` du projet contient ces points pour les images de validation ; avec `--latent 2`, on peut les dessiner. On observe en général des amas, un par glyphe, dont la forme suit les positions.

Une tentation vient aussitôt : prendre deux codes c_A et c_B , parcourir le segment $c(t) = (1 - t)c_A + tc_B$ pour t de 0 à 1, et décoder chaque point. C'est l'*interpolation* que dessine `interpolation.svg`. Entre deux positions d'un même glyphe, elle donne souvent un glissement convaincant. Entre deux glyphes différents, elle produit des fantômes : une croix qui s'efface pendant qu'un rond apparaît, des traits à moitié dessinés. Pourquoi ?



Un espace latent à deux dimensions, schématique : trois familles de glyphes forment trois amas. Le segment entre une croix et un rond traverse une région vide. Le décodeur n'y a jamais été entraîné et y dessine ce que ses poids lui dictent.

Trois raisons se combinent. D'abord, le décodeur n'a été entraîné que sur des codes d'images réelles ; entre les amas, il extrapole, et rien ne garantit que son extrapolation ressemble à une image. Ensuite, rien ne force les codes à remplir l'espace : la perte ne regarde que les points d'apprentissage, et les trous entre amas sont libres. Enfin, même si le décodeur se comportait au mieux, le milieu entre deux images n'est pas une image : c'est une superposition, et la MSE, nous l'avons vu, pousse à dessiner les superpositions en gris. Un code tiré au hasard dans l'espace latent cumule les trois soucis : il tombe presque sûrement loin de tout amas.

EN PRATIQUE

Avant de conclure sur une interpolation, regardez où tombent les codes intermédiaires par rapport aux codes de validation. Une image fantôme au milieu d'une région vide n'est pas un bug du décodeur : c'est une question posée là où il n'y a pas de réponse.

6 Un point devient une distribution : le VAE

L'*autoencodeur variationnel*, ou VAE, proposé par Kingma et Welling fin 2013, s'attaque précisément aux trous de l'espace latent. Son idée tient en une phrase : chaque image ne reçoit plus un code, mais une petite distribution de codes, et c'est un tirage dans cette distribution que le décodeur reconstruit. L'encodeur produit deux vecteurs de taille L , une moyenne μ et un logarithme de variance $\log \sigma^2$; le code est tiré selon la loi normale $\mathcal{N}(\mu, \sigma^2)$, coordonnée par coordonnée. Une même image donne donc des codes différents d'un passage à l'autre, et le décodeur doit bien reconstruire pour toute la région que couvre sa distribution, pas pour un seul point. Les trous se comblerent.

Si c'était tout, l'entraînement apprendrait vite à rendre σ minuscule pour retrouver un autoencodeur ordinaire. La perte du VAE ajoute donc un *terme de rapprochement* : la *divergence de Kullback-Leibler* entre la distribution de l'image et la loi normale centrée réduite $\mathcal{N}(0, 1)$, la même pour toutes les images. Cette divergence mesure à quel point une distribution diffère d'une autre ; elle est toujours positive ou nulle, et nulle seulement si les deux coïncident. Pour deux lois normales à coordonnées indépendantes, elle a une formule fermée :

$$\frac{1}{2} \sum_{j=1}^L (\mu_j^2 + \sigma_j^2 - 1 - \ln \sigma_j^2).$$

Lisons-la terme à terme. Le μ_j^2 pénalise une moyenne qui s'éloigne de zéro : toutes les images sont tirées vers le même centre, où leurs distributions se recouvrent. La quantité $\sigma_j^2 - 1 - \ln \sigma_j^2$ est positive ou nulle et s'annule seulement pour $\sigma_j = 1$, parce que $t - 1 \geq \ln t$ pour tout $t > 0$; elle pénalise aussi bien une distribution trop large qu'une distribution trop étroite, et rendre σ_j minuscule coûte $-\ln \sigma_j^2$, qui tend vers l'infini. Quelques valeurs pour une seule coordonnée : avec $\mu = 0$ et $\sigma^2 = 2$, le terme vaut environ 0,153 ; avec une variance de 0,25, environ 0,318 ; avec $\mu = 2$ et $\sigma^2 = 1$, exactement 2. La perte totale est la reconstruction plus β fois cette divergence, et le coefficient β arbitre : trop petit, on retrouve un autoencodeur avec ses trous ; trop grand, chaque distribution se confond avec $\mathcal{N}(0, 1)$, le code ne porte plus d'information et le décodeur dessine l'image moyenne. Ce dernier cas porte un nom, l'*effondrement postérieur*, et vous le verrez peut-être avec `--beta 0.1`.

PIÈGE

La divergence rapproche chaque distribution de $\mathcal{N}(0, 1)$; elle ne garantit pas que le nuage des moyennes μ soit lui-même normal. Tirer un code dans $\mathcal{N}(0, 1)$ puis le décoder donne des formes plausibles seulement dans la mesure où les distributions des images d'apprentissage recouvrent bien cette région. Les taches floues de `samples.svg` sont un résultat à expliquer, pas à cacher.

6.1 Dériver à travers un tirage au sort

Reste un souci technique que Kingma et Welling ont résolu d'une pirouette. La perte dépend de μ et de σ à travers un code tiré au hasard. Or, « tirer un nombre selon $\mathcal{N}(\mu, \sigma^2)$ » n'est pas une opération que l'on sait dériver : le tirage est une boîte noire pour la règle de la chaîne. L'*astuce de reparamétrisation* consiste à tirer d'abord un bruit ε selon $\mathcal{N}(0, 1)$, indépendant de tout paramètre, puis à calculer $z = \mu + \sigma\varepsilon$. Le résultat suit bien la loi voulue, mais le hasard est maintenant une *entrée* du calcul, comme le sont les pixels, et le chemin de μ et σ vers z est une formule ordinaire que Nx dérive sans peine. Le projet fournit ε en argument pour cette raison : cela rend aussi la fonction testable avec un bruit choisi.

Vérifions au moins que la formule produit la distribution annoncée. Dix mille tirages de $1 + 0,5\varepsilon$ devraient avoir une moyenne proche de 1 et un écart-type proche de 0,5 :

```
1 :rand.seed(:exsss, 2026)
2 tirages = for _ <- 1..10_000, do: 1.0 + 0.5 * :rand.normal()
3 moyenne = Enum.sum(tirages) / 10_000
4 ecart = :math.sqrt(
5   Enum.sum(Enum.map(tirages, &((81 - moyenne) ** 2))) / 10_000)
6 IO.inspect({Float.round(moyenne, 2), Float.round(ecart, 2)})
```

CONSOLE {1.0, 0.5}

Un dernier mot sur la validation du projet, qui décode la moyenne μ sans bruit : c'est une mesure déterministe et pratique, mais ce n'est ni la perte stochastique de l'entraînement ni l'espérance de la reconstruction. Les deux chiffres ne sont pas comparables terme à terme ; dites-le dans votre compte rendu.

7 Au-delà du projet : du débruitage à la diffusion

Les autoencodeurs produisent des images floues, et les VAE, souvent, des images un peu floues aussi. La génération d'images a d'abord contourné le problème avec les réseaux antagonistes de Goodfellow et ses coauteurs, en 2014, puis l'a résolu, vers 2020, en prenant au sérieux une idée que vous avez déjà programmée : le débruitage.

Un *modèle de diffusion*, dans la forme popularisée par Ho, Jain et Abbeel en 2020 d'après un travail de Sohl-Dickstein et ses coauteurs de 2015, définit un processus qui ajoute à une image un peu de bruit gaussien, puis encore un peu, en un millier d'étapes, jusqu'à ce qu'il ne reste que du bruit. Un réseau est ensuite entraîné à faire un pas en arrière : étant donné une image bruitée et le niveau de bruit, prédire le bruit qui a été ajouté, avec une MSE. C'est un autoencodeur de débruitage sans goulot, entraîné à tous les niveaux de corruption à la fois. Pour générer, on part d'un bruit pur et l'on débruite pas à pas ; à chaque étape, le réseau ne dessine qu'une petite correction, et la moyenne grise que la MSE impose ne porte que sur ce petit pas, pas sur l'image entière. C'est ainsi que le calcul de la section 2 cesse d'être un obstacle.

La *diffusion latente* de Rombach, Blattmann, Lorenz, Esser et Ommer, en 2022, ajoute la seconde moitié de votre projet. Débruitez des images de 512 pixels de côté, mille fois par image, coûte cher ; ils entraînent donc d'abord un autoencodeur qui réduit l'image d'un facteur 4 ou 8 dans chaque direction, puis font tourner la diffusion dans son espace latent. Le décodeur, appelé une seule fois à la fin, rend les pixels. Leur autoencodeur n'est pas entraîné avec une MSE seule, précisément pour éviter le gris : la perte compare des caractéristiques perceptives et ajoute un critère antagoniste local. Les générateurs d'images ouverts les plus répandus depuis 2022 reposent sur cette architecture : un espace latent et un décodeur, avec quelques centaines de millions de paramètres de plus que les vôtres.

7.1 La chasse aux anomalies

Une autre lignée utilise l'erreur de reconstruction comme un détecteur. Entraînez un autoencodeur sur des exemples normaux, par ex. des pièces sans défaut ou les relevés de capteurs d'un système sain. Il apprend à reconstruire ce qu'il a vu. Présentez-lui un exemple anormal : le code n'a pas de place pour ce qu'il contient, et l'erreur de reconstruction grimpe. Un seuil sur cette erreur, choisi sur des données de validation, sépare le normal de l'anormal sans qu'on ait jamais montré une anomalie au réseau. L'idée est ancienne : Japkowicz, Myers et Gluck l'ont présentée en 1995 à la conférence IJCAI, sous le nom de *détection de nouveauté*. Elle a les limites que vous connaissez déjà : un réseau trop généreux reconstruit aussi les anomalies, un réseau

trop pauvre reconstruit mal le normal, et une erreur n'a de sens que comparée à une référence, comme vos deux lignes de base.

7.2 Le lien avec le cours

Tout le projet est fait des briques des chapitres 7 et 8. Une couche dense écrite avec `Nx.dot` et un biais est celle qu'`Axon.dense` construisait en AP5 ; la forme `{B, 64}` qui devient `{B, L}` suit la convention d'AP4, un exemple par ligne ; la MSE, le gradient et Adam n'ont pas changé. Ce qui est nouveau est le protocole : la cible est l'entrée propre, la corruption de validation est fixée par une graine, et deux lignes de base, l'entrée abîmée rendue telle quelle et l'image moyenne de l'apprentissage, donnent un sens aux chiffres. Un réseau qui bat ces deux références sur les pixels masqués, mais pas sur l'ensemble, est un résultat honnête et intéressant ; le chapitre 8 vous a appris à l'écrire ainsi.

POUR EN PARLER

Trois affirmations que vous devriez pouvoir défendre après cette lecture : un autoencodeur linéaire ne fait pas mieux que l'ACP, et sa perte minimale est la variance abandonnée ; la MSE pousse un décodeur hésitant vers le gris ; la reparamétrisation déplace le hasard vers une entrée pour rendre le tirage dérivable.

Pour aller plus loin

Learning representations by back-propagating errors, D. Rumelhart, G. Hinton et R. Williams (1986).

<https://doi.org/10.1038/323533a0>

Trois pages dans *Nature* : la rétropropagation telle qu'elle a été présentée au monde. Le chapitre long de la même année contient le problème du codage.

Neural networks and principal component analysis: learning from examples without local minima, P. Baldi et K. Hornik (1989).

[https://doi.org/10.1016/0893-6080\(89\)90014-2](https://doi.org/10.1016/0893-6080(89)90014-2)

La preuve que le paysage d'un autoencodeur linéaire n'a pas de faux minimum. Court et lisible après un cours d'algèbre linéaire.

Reducing the dimensionality of data with neural networks, G. Hinton et R. Salakhutdinov (2006).

<https://www.cs.toronto.edu/~hinton/absps/science.pdf>

L'article de *Science* qui a relancé les autoencodeurs profonds ; regardez la figure où les codes à deux dimensions séparent les chiffres.

Extracting and composing robust features with denoising autoencoders, P. Vincent, H. Larochelle, Y. Bengio et P.-A. Manzagol (2008).

<https://www.cs.toronto.edu/~larocheh/publications/icml-2008-denoising-autoencoders.pdf>

L'origine de la corruption par masquage de votre projet, avec l'argument géométrique qui explique pourquoi débruiteur apprend de meilleures représentations.

An introduction to variational autoencoders, D. Kingma et M. Welling (2019).

<https://arxiv.org/abs/1906.02691>

Les auteurs du VAE réexpliquent leur idée six ans après, avec le temps de dérouler la perte et la reparamétrisation ; plus abordable que l'article de 2013.

Denoising diffusion probabilistic models, J. Ho, A. Jain et P. Abbeel (2020).

<https://arxiv.org/abs/2006.11239>

La diffusion dans sa forme devenue standard ; l'algorithme d'entraînement tient en cinq lignes et ressemble beaucoup à votre boucle de débruitage.

High-resolution image synthesis with latent diffusion models, R. Rombach, A. Blattmann, D. Lorenz, P. Esser et B. Ommer (2022).

<https://arxiv.org/abs/2112.10752>

Pourquoi diffuser dans l'espace latent d'un autoencodeur plutôt que dans les pixels, et comment cet autoencodeur est entraîné pour ne pas être flou.

Deep Learning, chapitre 14 « Autoencoders », I. Goodfellow, Y. Bengio et A. Courville (2016).

<https://www.deeplearningbook.org/contents/autoencoders.html>

Le chapitre de référence, en accès libre : goulot, débruitage et autoencodeurs contractifs.

Une question pour la route

Un seul nombre réel possède une infinité de décimales : en théorie, un code à $L = 1$ pourrait numéroté nos 128 images, et un décodeur parfait les restituer toutes. Pourquoi la descente de gradient ne trouve-t-elle jamais cette solution ? Et quelle est la plus petite valeur de L pour laquelle un décodeur *continu*, où un petit déplacement du code donne un petit changement de l'image, pourrait réussir sur huit glyphes placés sur une grille de quatre positions par quatre ? Dessinez l'espace latent avant de répondre. Pour toute question, vous pouvez me contacter par email avec la commande, l'erreur et le plus petit exemple qui reproduit le souci.