DIAGNOSTIC ET OPTIMISATION · DEEP LEARNING PAR LA PRATIQUE

Diagnostic et optimisation des modèles

Reconnaître ce qui ne va pas dans un entraînement, et savoir quoi corriger

Entraîner un réseau de neurones ne se résume pas à lancer un programme : il faut **lire** ce qui se passe, **nommer** le problème, puis choisir le bon remède. Ce document apprend à le faire, sur un cas pratique avec des contraintes réalistes (peu de données annotées, processeur seul, aucun téléchargement). Chaque technique est **mesurée** sur ce cas, avec plusieurs initialisations, et les résultats sont rapportés tels qu'ils sont, y compris quand une technique n'apporte presque rien. Trois questions y reçoivent une réponse :

- comment **reconnaître** un sous-apprentissage, un sur-apprentissage ou une convergence lente à partir des courbes (étapes 1 et 2) ;
- quelles **techniques** corriger chaque problème : régularisation, normalisation, batch normalization, écrêtage des gradients, fine-tuning (étapes 4 à 9) ;
- comment les **appliquer** sur un jeu de données précis, avec ses contraintes (étapes 3 et 10).

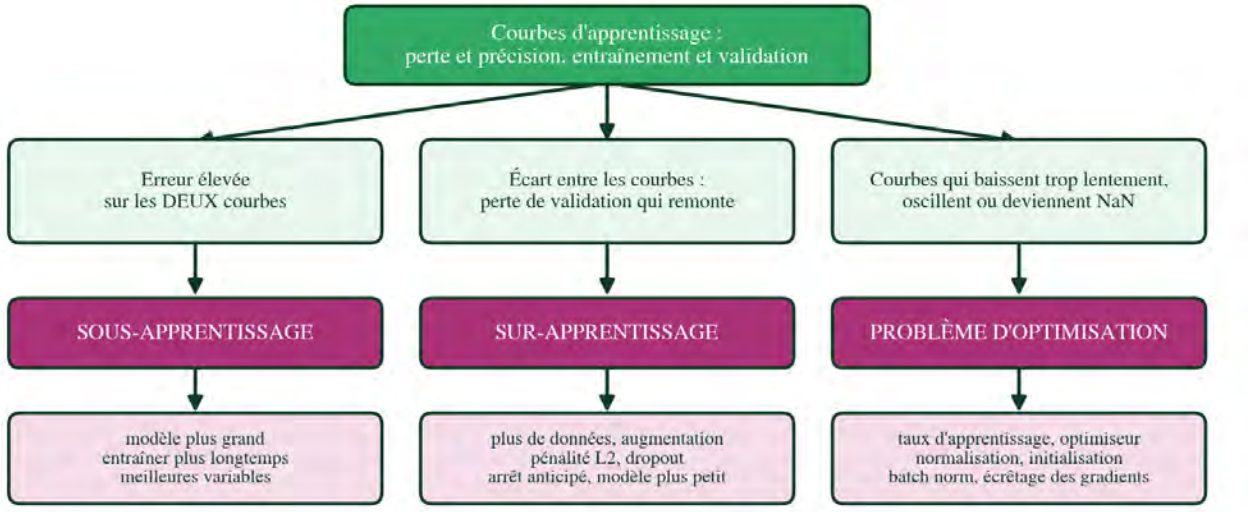

Figure 1 – La démarche : lire les courbes, nommer le symptôme, choisir un remède

> **Le cas pratique** : classer des chiffres manuscrits (images de 8 × 8 pixels, 10 classes, fournies avec la bibliothèque scikit-learn). Contrainte : **100 images annotées** pour l'entraînement. Tout le code est dans un seul script, `diagnostic_optimisation.py` (étape 11), qui s'exécute sur un processeur ordinaire.

Étape 1 – La démarche : cinq règles

- **Partir d'une référence simple.** Avant d'optimiser un réseau, on mesure ce que fait un modèle élémentaire (ici une régression logistique). Sans point de comparaison, on ne sait pas si un score est bon.
- **Un seul changement à la fois.** Si l'on modifie le taux d'apprentissage, l'architecture et la régularisation ensemble, on ne saura pas ce qui a marché.

- **Choisir sur la validation, regarder le test une seule fois.** Le jeu de test sert à la mesure finale ; l'utiliser pour décider d'un réglage le rend optimiste.
- **Répéter avec plusieurs graines aléatoires.** Un seul entraînement est trompeur : ici, l'écart-type d'un réglage à l'autre est de l'ordre de 0,5 point de précision. Une différence plus petite que cela ne prouve rien.
- **Contrôler avant d'optimiser.** La perte initiale d'un classifieur à 10 classes doit valoir environ ln(10) = 2,30 ; un réseau doit pouvoir surapprendre un tout petit lot. Si ce n'est pas le cas, le problème est un bogue, pas un réglage.

Étape 2 – Lire les courbes d'apprentissage

À chaque époque, on enregistre la perte et la précision **sur l'entraînement et sur la validation**. Ces courbes sont l'outil de diagnostic de base. La figure 2 montre trois symptômes réels, relevés sur le cas pratique :

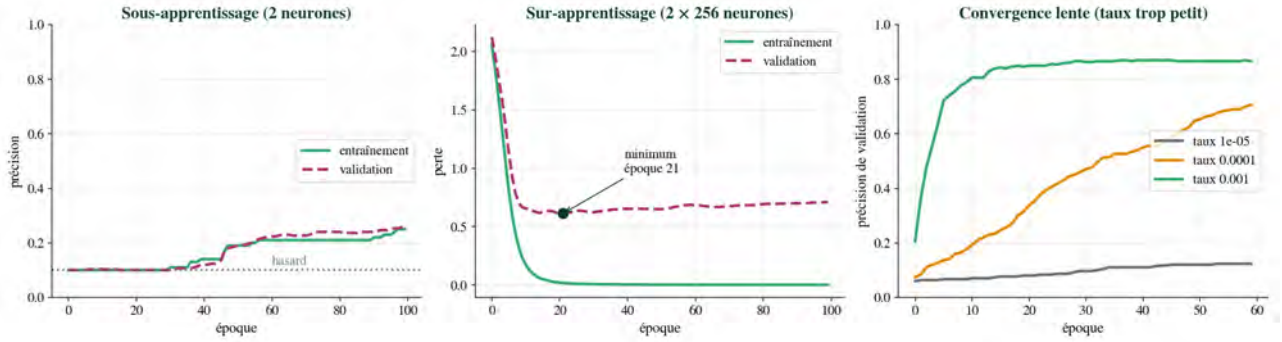

Figure 2 – Trois symptômes : sous-apprentissage, sur-apprentissage, convergence lente

Symptôme	Signature sur les courbes	Exemple mesuré sur le cas
Sous-apprentissage	erreur élevée sur l'entraînement et sur la validation ; les deux courbes sont proches	un réseau à 2 neurones cachés : précision de 0,28 à l'entraînement et 0,25 en validation, pour un hasard à 0,10
Sur-apprentissage	perte d'entraînement qui tend vers 0 ; perte de validation qui atteint un minimum puis remonte	un réseau 2 × 256 neurones : précision d'entraînement 1,000 contre 0,879 en validation ; perte de validation 0,62 au minimum, 0,77 en fin d'entraînement
Convergence lente	les courbes baissent mais trop lentement : l'entraînement s'arrête avant d'avoir fini	Adam avec un taux de 1e-4 : 0 entraînement sur 5 atteint 80 % de précision en 60 époques ; avec 1e-3, 80 % dès la 11ᵉ époque
Instabilité	perte qui oscille, bondit, ou devient NaN	SGD avec un taux de 1,0 : la perte devient NaN dans 5 entraînements sur 5

> **Un écart entre entraînement et validation n'est pas un problème en soi** : il existe toujours un peu. Ce qui compte, c'est la performance de **validation** et sa tendance. Un modèle qui surapprend peut encore être le meilleur disponible, tant que sa validation est bonne.

Étape 3 – Le cas pratique : contraintes et mise en place

Contrainte	Conséquence	Réponse dans le cas
Très peu d'exemples annotés (100)	risque élevé de sur-apprentissage	augmentation de données, régularisation, modèles modestes
Processeur seul, temps limité	pas de très gros modèles	petits réseaux denses et petit CNN ; le script complet tourne en une dizaine de minutes
Hors ligne, données sensibles	aucun jeu externe, aucun téléchargement	jeu de données fourni avec scikit-learn
Changement de domaine (nouveau scanner)	un modèle entraîné ailleurs se trompe	fine-tuning avec quelques exemples (étape 9)

Découpage des données : test 450 images (gardées pour la fin), validation 300, et 100 images tirées parmi les 1 047 restantes pour l'entraînement. Avoir plus d'exemples de validation que d'entraînement n'est pas réaliste ; on le fait ici pour que les mesures soient fiables. Dans un vrai projet, on utiliserait plutôt une validation croisée.

Points de référence : le hasard donne 0,10. Une **régression logistique** entraînée sur les mêmes 100 images donne 0,890 en validation et **0,851 sur le test**.

> **Premier enseignement** : avec 100 exemples, un modèle linéaire fait **au moins aussi bien** qu'un réseau de neurones sur la validation (0,890 contre 0,879 pour le réseau de référence de l'étape 5). Le réseau ne justifie sa complexité que si l'on arrive à le faire nettement mieux que cette référence.

Étape 4 – Sous-apprentissage

Un modèle sous-apprend quand il est **incapable d'ajuster même les données d'entraînement**. Sur le cas, on fait varier la taille du réseau (5 initialisations par réglage) :

Modèle	Précision d'entraînement	Précision de validation	Diagnostic
2 neurones cachés	0,280	0,251	sous-apprentissage net
16 neurones cachés	0,990	0,836	capacité suffisante, mais déjà un écart
2 × 256 neurones	1,000	0,880	sur-apprentissage (étape 5)

Avec 2 neurones, le réseau doit séparer 10 classes à travers un goulot de 2 valeurs : il n'en a pas la capacité, et la perte reste figée près de 2,0 (le hasard donne 2,30). **Les remèdes** :

- **augmenter la capacité** : plus de neurones, plus de couches ;
- **entraîner plus longtemps** ou vérifier le taux d'apprentissage (étape 6) ;
- **améliorer les variables** d'entrée (normalisation, variables plus informatives) ;
- **réduire la régularisation**, qui peut brider un modèle déjà trop simple.

Le piège est de confondre sous-apprentissage et convergence lente : dans les deux cas, l'entraînement est mauvais. La différence est que, avec un sous-apprentissage, **même un entraînement très long ne suffirait pas**.

Étape 5 – Sur-apprentissage

5.1 Le constat

Le réseau de référence (2 couches de 256 neurones, aucune régularisation) atteint **1,000 de précision sur l'entraînement** mais 0,879 sur la validation et 0,862 sur le test. Sa perte de validation descend jusqu'à 0,62 puis remonte à 0,77 : le réseau se met à **mémoriser** les 100 images au lieu d'apprendre ce qui généralise.

5.2 Les techniques

- **Pénalité L2 (weight decay)** : on ajoute $\lambda \cdot \Sigma\, w^2$ à la perte, ce qui décourage les grands poids (voir le chapitre sur la régularisation).
- **Dropout** : à l'entraînement, chaque neurone est éteint au hasard avec une probabilité p ; le réseau ne peut plus compter sur un neurone isolé. Il faut passer en mode évaluation (`model.eval()`) pour l'inférence.
- **Arrêt anticipé** : on arrête quand la perte de validation cesse de s'améliorer, et on garde le meilleur modèle.
- **Augmentation de données** : on fabrique de nouveaux exemples valides à partir des existants. Ici, l'image et ses **4 décalages d'un pixel** (le chiffre garde le même sens), soit 500 exemples au lieu de 100.
- **Réduire le modèle** ou **ajouter des données annotées**.

5.3 Les résultats

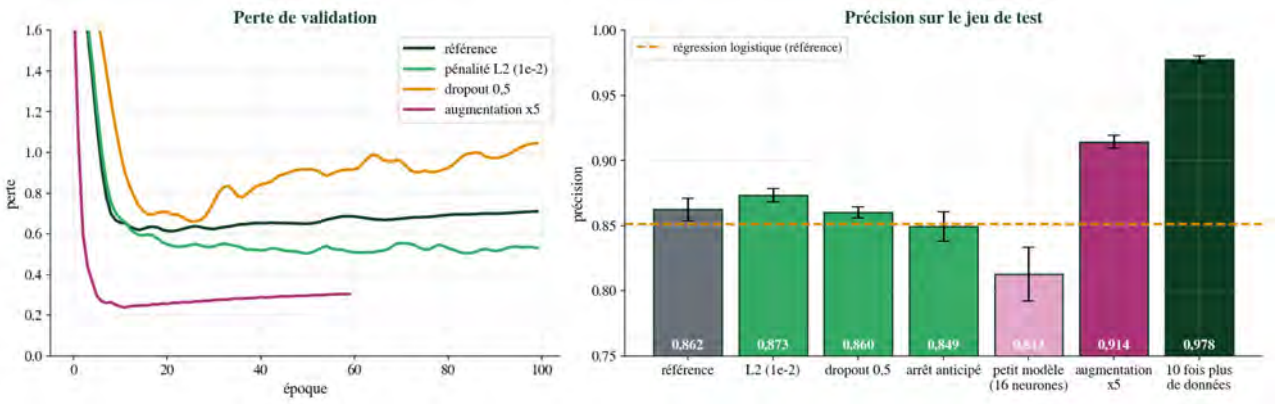

Figure 3 – À gauche : perte de validation ; à droite : précision sur le jeu de test (moyenne et écart-type sur 8 initialisations)

Réglage	Validation	Test	Perte de validation (minimum, fin)
référence (aucune régularisation)	0,879	0,862	0,62 → 0,77
pénalité L2 (1e-3)	0,885	0,868	0,61 → 0,63
pénalité L2 (1e-2)	0,888	0,873	0,52 → 0,56
dropout 0,5	0,884	0,860	0,64 → 0,99
arrêt anticipé (patience 8)	0,876	0,849	0,62 → 0,67
dropout 0,3 + L2 (1e-3)	0,882	0,865	0,64 → 0,73
modèle plus petit (16 neurones)	0,833	0,813	0,76 → 0,76
augmentation de données (x5)	0,908	0,914	0,26 → 0,33
augmentation + L2 (1e-2)	0,907	0,909	0,26 → 0,27
augmentation + dropout 0,3 + L2 (1e-3)	0,909	0,914	0,24 → 0,28
10 fois plus de données (1 047 exemples)	0,970	0,978	0,15 → 0,20

5.4 Comment lire ces résultats

- **L'augmentation de données est le remède le plus efficace à nombre d'annotations constant** : +5 points de test (0,862 → 0,914). C'est logique : elle injecte une connaissance du problème (un chiffre décalé d'un pixel reste le même chiffre).
- **Les régularisations classiques améliorent surtout la perte, peu la précision** : la pénalité L2 (1e-2) ramène la perte de validation de 0,77 à 0,56, mais la précision de test ne gagne qu'un point (0,862 → 0,873), de l'ordre du bruit. Le dropout 0,5 et l'arrêt anticipé ne changent pas la précision.
- **La perte et la précision ne disent pas la même chose** : avec le dropout 0,5, la perte de validation remonte jusqu'à 0,99 sans que la précision baisse. La perte pénalise la confiance excessive sur les erreurs ; la précision ne compte que les bonnes réponses.
- **Trop réduire le modèle le fait basculer en sous-apprentissage** : 16 neurones donnent 0,813 de test, moins que la référence.
- **Annoter davantage reste le meilleur remède** : avec 10 fois plus d'exemples, on atteint 0,978. Ce résultat donne une borne : si l'annotation est possible, c'est elle qu'il faut privilégier.

> **Précaution** : les valeurs de test sont affichées ici pour tous les réglages **à titre pédagogique**. Dans un projet réel, on choisit le réglage avec la validation, puis on ne regarde le test qu'**une seule fois**.

Étape 6 – Convergence lente et instabilité

6.1 Le taux d'apprentissage

C'est le réglage le plus influent. Trop petit, le réseau avance à peine ; trop grand, il oscille ou diverge. On a entraîné le même réseau (une couche de 128 neurones, 60 époques, 5 initialisations) avec plusieurs taux :

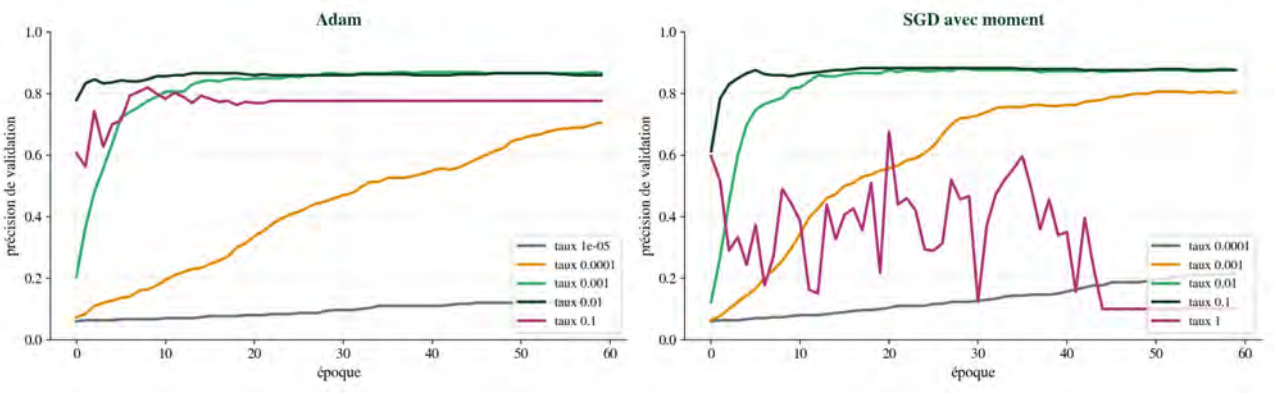

Figure 4 – Précision de validation selon le taux d'apprentissage (Adam à gauche, SGD avec moment à droite)

Optimiseur et taux	80 % atteint (sur 5)	Époques pour y arriver	Précision de validation finale
Adam 1e-5	0	jamais	0,149
Adam 1e-4	0	jamais	0,719
Adam 1e-3	5	11	0,875
Adam 1e-2	5	2	0,869
Adam 1e-1	4	10	0,782 (oscille)
SGD 1e-4	0	jamais	0,223
SGD 1e-3	5	51	0,821
SGD 1e-2	5	10	0,883
SGD 1e-1	5	3	0,873
SGD 1,0	0	NaN dans 5 cas sur 5	0,100

- **Trop petit** : la courbe monte lentement et l'entraînement s'arrête trop tôt (Adam 1e-5 et 1e-4).
- **Trop grand** : la courbe oscille (Adam 0,1 finit à 0,782 alors que des taux plus petits dépassent 0,86) ou diverge (SGD 1,0 : NaN).
- **Adam tolère mieux** une large plage de taux (de 1e-3 à 1e-2 ici) ; SGD y est plus sensible.
- **Diagnostic pratique** : faire croître le taux pas à pas en surveillant la perte, et retenir une valeur un peu en dessous de celle où la perte cesse de baisser. On peut aussi **réduire le taux au fil de l'entraînement** (décroissance par paliers, cosinus, échauffement au début). Ces deux méthodes ne sont pas mesurées ici.

6.2 La normalisation des entrées

Quand les variables ont des échelles très différentes, la descente de gradient est dominée par celles de grande échelle. Le jeu « vin » (178 vins, 13 mesures chimiques) en est un bon exemple : l'alcool varie de 11 à 15, la proline de 278 à 1 680. Même réseau (32 neurones), SGD, 8 initialisations :

Entrées	90 % atteint (sur 8)	Époques pour y arriver	Précision finale
valeurs brutes	0	jamais	0,389
valeurs standardisées	8	2	0,998

La standardisation (soustraire la moyenne, diviser par l'écart-type, **calculés sur l'entraînement seul**) fait passer le réseau de l'échec à 99,8 %. Sur les chiffres manuscrits, en revanche, l'effet est faible : tous les pixels ont déjà une échelle comparable (0 à 16). Dans un essai préliminaire non repris dans le script, les pixels bruts convergeaient aussi vite que les pixels standardisés. **La normalisation est donc décisive quand les échelles diffèrent, et superflue quand elles sont déjà comparables.**

> **Attention aux pixels constants** : un pixel qui vaut toujours 0 dans l'entraînement a un écart-type nul. Diviser par une valeur proche de 0 faisait exploser les valeurs de validation (pertes de l'ordre de 10 000 lors de mes premiers essais). Le script remplace ces écarts-types nuls par 1.

6.3 L'initialisation

Une mauvaise initialisation (poids trop petits ou trop grands) fait évanouir ou exploser les signaux dès le départ. Les initialisations de Xavier (activations tanh, sigmoïde) et de He (ReLU), vues au chapitre sur les poids, ajustent l'échelle des poids au nombre de connexions. PyTorch les applique par défaut avec des valeurs raisonnables pour les couches denses ; on ne les modifie que si le diagnostic l'impose. Cette technique n'est pas mesurée ici.

Étape 7 – La batch normalization

7.1 Le principe

Pour chaque neurone, la **batch normalization** normalise ses valeurs **sur le mini-lot** en cours, puis applique deux paramètres appris (un facteur γ et un décalage β) :

> $\hat{x} = (x - \mu_lot) / \sqrt{(\sigma^2_lot + \varepsilon)}$ puis $y = \gamma \cdot \hat{x} + \beta$

Exemple pour un neurone qui reçoit, sur un lot de 4 exemples, les valeurs (2 ; 4 ; 6 ; 8) : la moyenne vaut 5 et l'écart-type 2,236, donc $\hat{x}$ = (−1,342 ; −0,447 ; 0,447 ; 1,342). Les valeurs ont maintenant une moyenne de 0 et un écart-type de 1, quelles que soient les couches précédentes. En évaluation, on remplace la moyenne et l'écart-type du lot par des moyennes glissantes enregistrées pendant l'entraînement (d'où l'importance de `model.eval()`).

À ne pas confondre avec la normalisation par couche (utilisée dans les Transformers) : la batch normalization calcule ses statistiques **sur le lot**, pour chaque neurone ; la normalisation par couche les calcule **sur les neurones d'un même exemple**, donc sans dépendre de la taille du lot.

7.2 L'expérience

On entraîne un réseau de 6 couches de 64 neurones (SGD avec moment, taux 0,05, 5 initialisations), avec et sans batch normalization, pour trois activations :

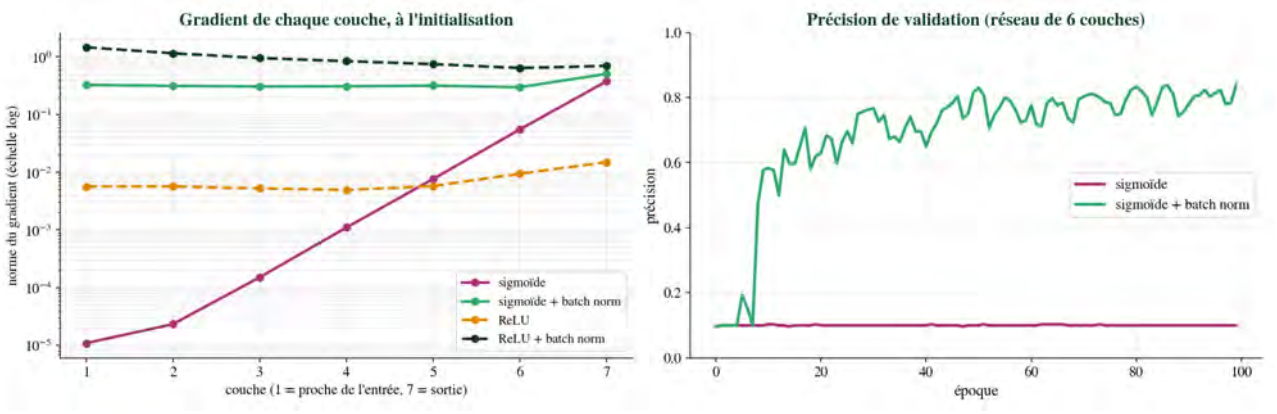

Figure 5 – À gauche : norme du gradient de chaque couche à l'initialisation ; à droite : précision de validation

Réseau (6 couches)	80 % atteint (sur 5)	Époques	Précision finale
sigmoïde	0	jamais	0,100
sigmoïde + batch norm	5	38	0,809
ReLU	2 (et 1 NaN)	83	0,589
ReLU + batch norm	3	66	0,754
tanh	1	17	0,774
tanh + batch norm	3	38	0,743

- **Pour la sigmoïde, l'effet est décisif** : sans batch normalization, aucun entraînement n'apprend. Le gradient de la première couche est d'environ 10^{-5}, contre 0,4 pour la dernière (un facteur de 30 000) : c'est l'**évanouissement du gradient**. Avec la batch normalization, toutes les couches reçoivent un gradient de l'ordre de 0,3.
- **Pour la ReLU**, elle stabilise l'entraînement (plus de NaN, 3 réussites sur 5 au lieu de 2) mais ne le rend pas fiable à elle seule.
- **Pour la tanh, le gain n'est pas net** : plus de réussites (3 sur 5 contre 1), mais une précision finale un peu plus basse.
- **Un réseau profond n'est pas meilleur ici** : les réseaux de 6 couches (0,59 à 0,81) sont **en dessous** du réseau d'une seule couche de 128 neurones (0,875 avec Adam 1e-3). Avec 100 exemples, ajouter de la profondeur complique l'optimisation sans rien apporter.

Précautions : la batch normalization devient bruitée avec de très petits lots ; ses statistiques doivent être placées en mode évaluation à l'inférence ; avec les séquences, on préfère la normalisation par couche.

Étape 8 – L'écrêtage des gradients

8.1 Le principe

Quand le gradient devient très grand (explosion), un seul pas de mise à jour peut propulser les poids très loin et ruiner l'entraînement. L'**écrêtage** plafonne la **norme** du gradient à une valeur c, en conservant sa direction : si $\|g\| > c$, alors $g \leftarrow g \times c / \|g\|$.

> **Exemple** : un gradient de norme 5 avec c = 1 est multiplié par 1/5 = 0,2. Sa direction ne change pas, seule sa longueur est ramenée à 1. Si sa norme est inférieure à c, il n'est pas modifié.
>
> En PyTorch : `torch.nn.utils.clip_grad_norm_(modele.parameters(), 1.0)`, entre `perte.backward()` et `optimiseur.step()`.

8.2 L'expérience : un RNN qui lit les chiffres pixel par pixel

Pour provoquer l'explosion, on fait lire chaque image **pixel par pixel** à un RNN simple (64 pas de temps), avec un taux d'apprentissage élevé (0,2), 400 exemples, 6 initialisations. On teste aussi des poids récurrents multipliés par 3 (échelle 3), qui rendent l'explosion plus probable :

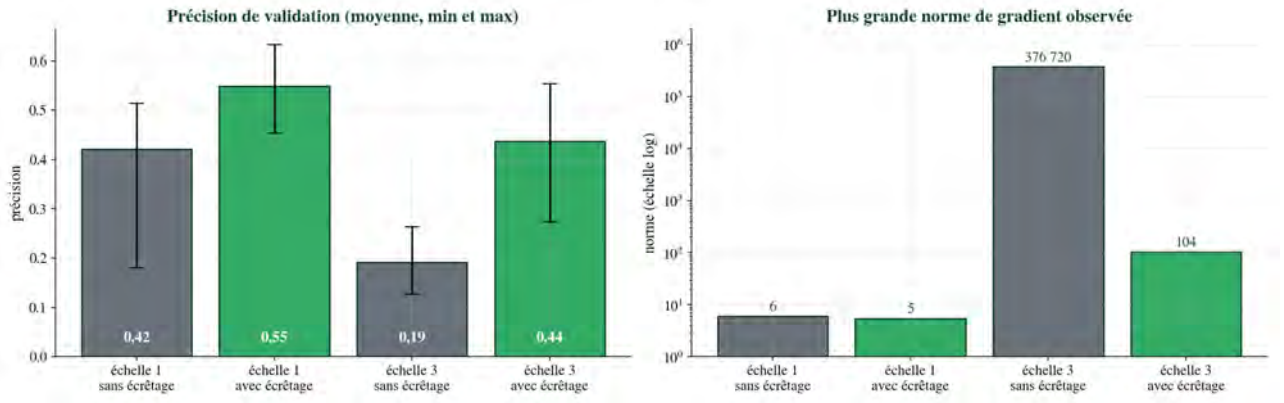

Figure 6 – Précision de validation (à gauche) et plus grande norme de gradient observée (à droite, échelle logarithmique)

Réglage	Précision de validation (moyenne)	Min – max	Plus grande norme de gradient
échelle 1, sans écrêtage	0,421	0,18 – 0,51	6
échelle 1, avec écrêtage	0,548	0,45 – 0,63	5
échelle 3, sans écrêtage	0,191	0,13 – 0,26	376 720
échelle 3, avec écrêtage	0,437	0,27 – 0,55	104

- **Là où le gradient explose** (échelle 3, norme de 376 720 observée), l'écrêtage rend l'apprentissage possible : 0,19 contre 0,44. La norme affichée dans la ligne avec écrêtage est mesurée **avant** l'écrêtage.
- **Là où il explose peu** (échelle 1), le gain est plus modeste (0,42 contre 0,55), et les scores restent bas : lire 64 pas avec un RNN simple est difficile. Le remède structurel est une architecture à portes (LSTM, GRU).
- **L'écrêtage est un filet de sécurité, pas un remède universel** : dans un essai préliminaire avec un RNN qui lit seulement 8 lignes (8 pas), il n'apportait aucun gain et ralentissait l'apprentissage. On l'active quand on **observe** des pics de gradient ou de perte, pas systématiquement.

Bonne pratique : enregistrer la norme du gradient à chaque époque (le script le fait) et regarder si elle présente des pics.

Étape 9 – Le fine-tuning

9.1 Le principe

Le **fine-tuning** consiste à partir d'un modèle déjà entraîné sur une tâche proche (la tâche **source**) plutôt que de repartir de zéro. On remplace la dernière couche (la tête) et on choisit ce que l'on réentraîne :

- **Caractéristiques gelées** : on fige le corps du réseau et on n'entraîne que la tête. C'est rapide, adapté à très peu de données.
- **Fine-tuning complet** : on réentraîne tous les poids, avec un **taux d'apprentissage petit** pour ne pas détruire ce qui a été appris.
- **Variantes** : dégeler progressivement les couches, ou utiliser un taux plus petit pour les premières couches que pour la tête.

9.2 L'expérience : un nouveau scanner

Un petit CNN est entraîné sur 1 347 images propres (précision 0,916). Puis le « scanner » change : les images arrivent **décalées d'un pixel et bruitées**. Le modèle source, sans adaptation, tombe à **0,315**. On dispose de seulement 2, 5 ou 10 images annotées par classe dans ce nouveau domaine (4 tirages, 3 modèles pré-entraînés) :

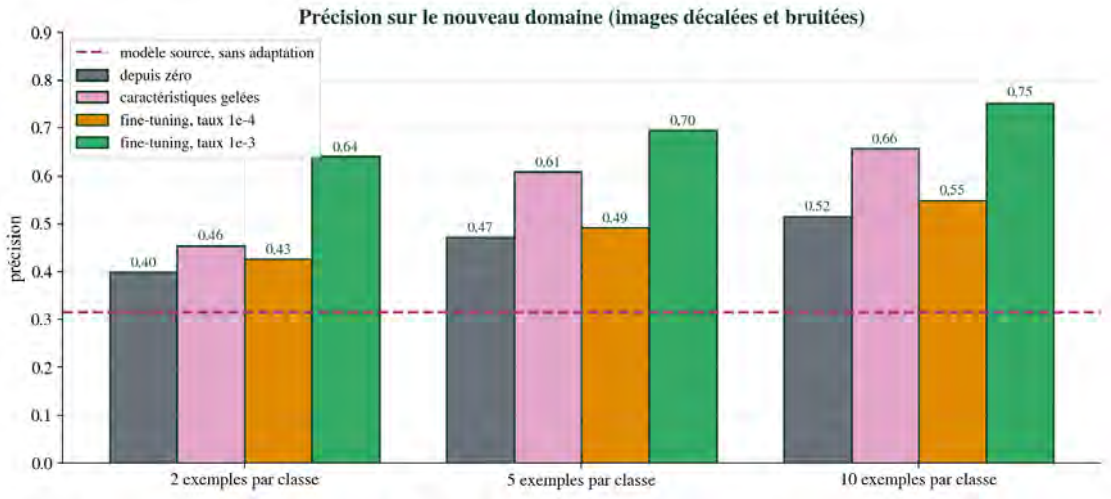

Figure 7 – Précision sur le nouveau domaine selon la méthode et le nombre d'exemples annotés par classe

Méthode	2 par classe	5 par classe	10 par classe
depuis zéro	0,40	0,47	0,52
caractéristiques gelées (tête seule)	0,46	0,61	0,66
fine-tuning, taux 1e-4	0,43	0,49	0,55
fine-tuning, taux 1e-3	0,64	0,70	0,75

- **Le fine-tuning avec un taux adapté gagne environ 22 à 24 points** par rapport à l'entraînement depuis zéro, pour le même nombre d'exemples.
- **Un taux trop petit (1e-4) bouge à peine** en 40 époques : le fine-tuning n'est pas « sans réglage ».
- **Geler le corps aide aussi**, mais moins ici, car le domaine a changé (décalage et bruit) : les caractéristiques apprises sur des images propres ont besoin d'être ajustées.

> **Quand le fine-tuning ne marche pas** : dans des essais préliminaires (non repris dans le script), un réseau dense sur pixels bruts, pré-entraîné sur les chiffres 0 à 4 puis adapté aux chiffres 5 à 9, ou adapté à des images décalées, **n'a pas fait mieux** que l'entraînement depuis zéro. Pour qu'il serve, il faut des **caractéristiques réutilisables** (un CNN capte des motifs qui se déplacent) et une tâche source **proche** de la cible.

Étape 10 – Bilan du cas et tableau de décision

10.1 Ce que l'on a gagné

Étape	Précision de test	Ce qui a changé
hasard	0,100	—
régression logistique (référence)	0,851	modèle linéaire, 100 exemples
réseau 2 × 256, sans régularisation	0,862	+1,1 point : le réseau n'apporte presque rien
+ augmentation de données	0,914	+5,2 points : la vraie amélioration
avec 10 fois plus d'exemples annotés	0,978	la borne à viser si l'on peut annoter

Le réglage retenu est l'**augmentation de données seule** : sur la validation, ses variantes avec pénalité L2 ou dropout font aussi bien (0,907 à 0,909 contre 0,908), donc rien ne justifie de les ajouter, et le modèle le plus simple l'emporte. Face à un changement de domaine, on poursuit avec un **fine-tuning** (étape 9), qui donne 0,64 à 0,75 avec 2 à 10 exemples par classe.

10.2 Tableau de décision

Symptôme	À vérifier d'abord	Premiers remèdes à essayer
Erreur élevée partout (sous-apprentissage)	taux d'apprentissage, durée, bogue (perte initiale ≈ ln K ?), pouvoir surapprendre un petit lot	modèle plus grand, entraîner plus longtemps, meilleures variables, moins de régularisation
Entraînement parfait, validation moyenne (sur-apprentissage)	fuite de données, écart-type entre graines, perte de validation qui remonte	plus de données, augmentation, pénalité L2, dropout, arrêt anticipé, modèle plus petit
Courbes trop lentes	taux d'apprentissage, normalisation des entrées, optimiseur	augmenter le taux (test de plage), Adam, standardiser, initialisation adaptée
Perte qui oscille, bondit ou devient NaN	taux trop grand, pics de la norme du gradient, valeurs aberrantes	réduire le taux, écrêtage des gradients, normalisation, vérifier les données
Réseau profond qui n'apprend pas	norme du gradient par couche (évanescence ?), activation sigmoïde	batch normalization, activation ReLU, initialisation, moins de couches
Peu d'annotations dans un nouveau domaine	existe-t-il un modèle proche pré-entraîné ?	fine-tuning (tête seule, puis complet avec petit taux), augmentation

10.3 Liste de contrôle

- Découpage entraînement / validation / test fait **avant** toute normalisation ; statistiques calculées sur l'entraînement seul.
- Référence simple mesurée (modèle linéaire, ou règle majoritaire).
- Perte initiale ≈ ln K ; le réseau surapprend un petit lot.
- Courbes de perte **et** de précision, sur entraînement **et** validation, enregistrées à chaque époque.
- Norme du gradient surveillée (pics, évanescence).
- Plusieurs graines ; une différence plus petite que l'écart-type n'est pas un résultat.
- Un changement à la fois, consigné dans un journal d'expériences.
- Test regardé **une seule fois**, à la fin.

Étape 11 – Pour le reproduire

Le script `diagnostic_optimisation.py` contient tout le cas pratique : outils communs (découpage, standardisation, augmentation, boucle d'entraînement qui enregistre les courbes et la norme du gradient) et sept sections correspondant aux étapes de ce document. Il utilise PyTorch, scikit-learn et NumPy, tourne sur un processeur et ne télécharge rien :

> **Version TP** : le notebook `TP_diagnostic_optimisation.ipynb` reprend ces expériences sous forme de **TP guidé** (six exercices à compléter avec cellules de vérification, neuf questions, onze graphiques). Il est plus léger que le script (3 graines au lieu de 5 à 8) et s'exécute en 5 à 8 minutes. Une version corrigée, avec ses résultats, est fournie pour le formateur.

```
python diagnostic_optimisation.py                # toutes les sections
python diagnostic_optimisation.py --section 2    # une seule section (0 à 6)
```

Section	Contenu	Durée (mon environnement)
0	points de référence (hasard, régression logistique)	moins d'une seconde
1	sous-apprentissage (taille du réseau)	12 secondes
2	sur-apprentissage et remèdes	environ 3 minutes
3	taux d'apprentissage et normalisation (chiffres et vin)	26 secondes
4	réseau profond, gradients par couche, batch normalization	47 secondes
5	écrêtage des gradients (RNN pixel par pixel)	environ 1 minute 40
6	fine-tuning après changement de domaine	environ 4 minutes

Chaque section enregistre ses résultats dans le dossier `resultats/` (fichiers .json) : courbes d'apprentissage, précisions, normes de gradient. Les durées dépendent du processeur ; elles sont données à titre indicatif. Les valeurs peuvent varier de quelques millièmes selon la version des bibliothèques, mais les ordres de grandeur et les conclusions se retrouvent.

À retenir

1. Un diagnostic commence par les **courbes** (perte et précision, entraînement et validation) et par une **référence simple**.

2. Sous-apprentissage : erreur élevée partout ; **sur-apprentissage** : validation qui décroche ; **convergence lente ou instable** : problème d'optimisation. Chacun a ses remèdes.

3. Avec peu de données, l'**augmentation** et l'**annotation** pèsent plus que la régularisation : ici +5 points pour l'augmentation, contre environ +1 point pour la pénalité L2.

4. Le **taux d'apprentissage** est le premier réglage à contrôler ; la **normalisation des entrées** est décisive quand les échelles diffèrent (0 contre 99,8 % sur le jeu vin).

5. La **batch normalization** rend un réseau profond sigmoïde entraînable (0 sur 5 contre 5 sur 5) ; l'**écrêtage** protège les réseaux récurrents des explosions de gradient.

6. Le **fine-tuning** gagne beaucoup quand le modèle source est proche et ses caractéristiques réutilisables (+22 à 24 points ici), et rien quand ce n'est pas le cas.

7. Mesurer avec plusieurs graines, changer une chose à la fois, ne regarder le test qu'une fois.