OCR Word Search Solver

Alexandre Joaquim Lima Salgueiro
alexandre-joaquim.lima-salgueiro

Hugo Guyennet
hugo.guyennet

Léa-Angélina Kolmerschlag
lea-angelina.kolmerschlag

Tom Huynh
tom.huynh

4 novembre 2025

Sommaire

1 Réseau de neurone pour la reconnaissance de caractères **3**
1.1 Représentation des données sous forme de tenseurs . . . 3
1.1.1 Représentation en Mémoire et Performance . . . 3

2 Architecture du réseau de neurones **4**
2.1 Couche linéaire . . . 4
2.1.1 Passe avant (Forward Pass) . . . 5
2.1.2 Passe arrière (Backward Pass) . . . 5
2.2 Couche de convolution 2D (Conv2D) . . . 5
2.2.1 Passe avant (Forward Pass) . . . 6
2.2.2 Passe arrière (Backward Pass) . . . 7
2.2.3 Optimisation algorithmique de la convolution . . . 7
2.2.4 Autres pistes d'optimisation . . . 8
2.3 Couche de Normalisation par Lots (Batch Normalization) . . . 8
2.3.1 Passe avant (Forward Pass) . . . 9
2.3.2 Passe arrière (Backward Pass) . . . 9
2.4 Couche de Dropout . . . 10
2.4.1 Passe avant (Forward Pass) . . . 11
2.4.2 Passe arrière (Backward Pass) . . . 11
2.5 Couche de Dropout 2D (Dropout2D) . . . 12
2.5.1 Passe avant (Forward Pass) . . . 12
2.5.2 Passe arrière (Backward Pass) . . . 12
2.6 Couche de Max-Pooling 2D (MaxPool2D) . . . 12
2.6.1 Passe avant (Forward Pass) . . . 13
2.6.2 Passe arrière (Backward Pass) . . . 13
2.7 Couche de mise en commun moyenne adaptative 2D (Adaptive Average Pooling 2D) . . . 13
2.7.1 Passe avant (Forward Pass) . . . 14
2.7.2 Passe arrière (Backward Pass) . . . 14
2.8 Fonction de perte : Entropie Croisée (Cross-Entropy Loss) . . . 15
2.8.1 La fonction Softmax . . . 15
2.8.2 Passe avant (Forward Pass) . . . 15
2.8.3 Passe arrière (Backward Pass) . . . 15

3 Explication de l'architecture utilisée **17**
3.1 Fonction d'activation : SiLU (Sigmoid Linear Unit) . . . 17
3.1.1 Choix par rapport à d'autres fonctions d'activation . . . 18
3.2 Initialisation des Poids . . . 18
3.3 Principes de Conception : Blocs Résiduels et Efficacité . . . 19
3.4 Détail de l'architecture . . . 20
3.4.1 Bloc Convolutif 1 . . . 20
3.4.2 Bloc Convolutif 2 . . . 21
3.4.3 Bloc Convolutif 3 . . . 21
3.4.4 Tête de Classification . . . 22

4 Optimisation du modèle : l'optimiseur AdamW **23**
4.1 Le rôle d'un optimiseur dans l'entraînement . . . 23
4.2 Adam (Adaptive Moment Estimation) . . . 23
4.3 Le problème de la régularisation L2 dans Adam . . . 24
4.4 AdamW : Le découplage pour une meilleure régularisation . . . 24
4.5 L'algorithme AdamW en détail . . . 24

5 Entraînement du Modèle **26**
5.1 Préparation des Données et Environnement . . . 26
5.2 Hyperparamètres . . . 27
5.3 Analyse de la Performance du Modèle . . . 28
5.3.1 Évolution par Époque . . . 28
5.3.2 Analyse au Niveau des Lots . . . 28
5.4 Stratégies d'Optimisation de l'Entraînement . . . 29
5.4.1 Ordonnancement du Taux d'Apprentissage (Learning Rate Scheduling) . . . 29
5.4.2 Prévention du Surapprentissage et Arrêt Précoce (Early Stopping) . . . 29
5.5 Résultats Finaux de l'Entraînement . . . 30
5.5.1 Métriques Finales . . . 30
5.6 Considérations sur le Matériel et le Temps d'Entraînement . . . 30
5.7 Défis Rencontrés et Analyse des Résultats sur Données Réelles . . . 30

6 Le Solveur : Une Approche par Appariement de Patrons Probabilistes **31**
6.1 Représentation Probabiliste . . . 31
6.2 Le Mécanisme de Notation : Entropie Croisée . . . 31
6.2.1 Log-Probabilités pour la Stabilité Numérique . . . 31
6.2.2 Score Lettre-à-Cellule via l'Entropie Croisée . . . 32
6.2.3 Score de Chemin . . . 32
6.3 Implémentation et Complexité de l'Algorithme . . . 32
6.3.1 Analyse de la Complexité . . . 33
6.3.2 Comparaison avec un solveur heuristique . . . 33

1 Réseau de neurone pour la reconnaissance de caractères

Pour la reconnaissance de caractères, nous utiliserons un réseau de neurones convolutif (CNN).
Un CNN est un type de réseau de neurones spécialement conçu pour le traitement d'images. Sa structure s'inspire du cortex visuel animal. Il utilise des couches de convolution pour extraire des caractéristiques hiérarchiques des images, comme les contours, les formes, puis des objets plus complexes. Ces couches sont suivies de couches de pooling, qui réduisent la taille des données pour en conserver les informations essentielles. Finalement, des couches entièrement connectées, similaires à celles d'un réseau de neurones classique, effectuent la classification finale pour identifier le caractère. Cette architecture rend les CNN particulièrement performants pour la reconnaissance de motifs dans les images.

Note:

Pour accélérer les calculs intensifs de notre réseau de neurones, nous exploitons les instructions SIMD (Single Instruction, Multiple Data). Le principe du SIMD est d'effectuer une seule opération sur plusieurs données simultanément. Les processeurs modernes disposent de registres spéciaux pouvant contenir des vecteurs de données (par exemple, 8 nombres flottants). Une seule instruction SIMD peut alors additionner ou multiplier tous ces nombres en un seul cycle d'horloge. L'utilisation de ces instructions, notamment via les intrinsèques AVX2, permet de paralléliser les calculs au plus bas niveau et d'obtenir des gains de performance considérables.

1.1 Représentation des données sous forme de tenseurs

Le **tenseur** est la structure de données fondamentale au cœur de notre réseau de neurones. Il s'agit d'une généralisation des vecteurs (tableaux à une dimension) et des matrices (tableaux à deux dimensions) à un nombre arbitraire de dimensions. Dans le cadre de ce projet, toutes les données manipulées, qu'il s'agisse des images d'entrée, des poids des couches ou des résultats intermédiaires, sont représentées sous forme de tenseurs.

1.1.1 Représentation en Mémoire et Performance

L'efficacité de la manipulation des tenseurs est intrinsèquement liée à leur représentation en mémoire. Au lieu de recourir à des structures de données imbriquées (comme des tableaux de tableaux), un tenseur stocke l'ensemble de ses valeurs dans un **bloc de mémoire contigu**, qui s'apparente à un simple tableau unidimensionnel.

La structure multidimensionnelle est donc une abstraction logicielle, définie par des méta-informations qui décrivent comment interpréter ce bloc de données. La plus importante de ces méta-informations est la **forme** (*shape*) du tenseur. La forme spécifie la taille de chaque dimension, permettant de projeter le tableau de données brutes en une grille multidimensionnelle. Par exemple, un tenseur de forme `(28, 28)`, représentant une image de 28x28 pixels, contiendra 784 éléments stockés séquentiellement en mémoire.

Cette organisation est déterminante pour les performances. Le stockage contigu des données garantit une excellente **localité spatiale**, ce qui signifie que les données accédées séquentiellement sont physiquement proches en mémoire. Cette propriété est exploitée par la mémoire cache du processeur pour réduire les temps de latence. De plus, cette disposition linéaire des données est idéale pour le traitement vectoriel, permettant d'exploiter efficacement les instructions **SIMD** des processeurs modernes pour paralléliser les calculs.

2 Architecture du réseau de neurones

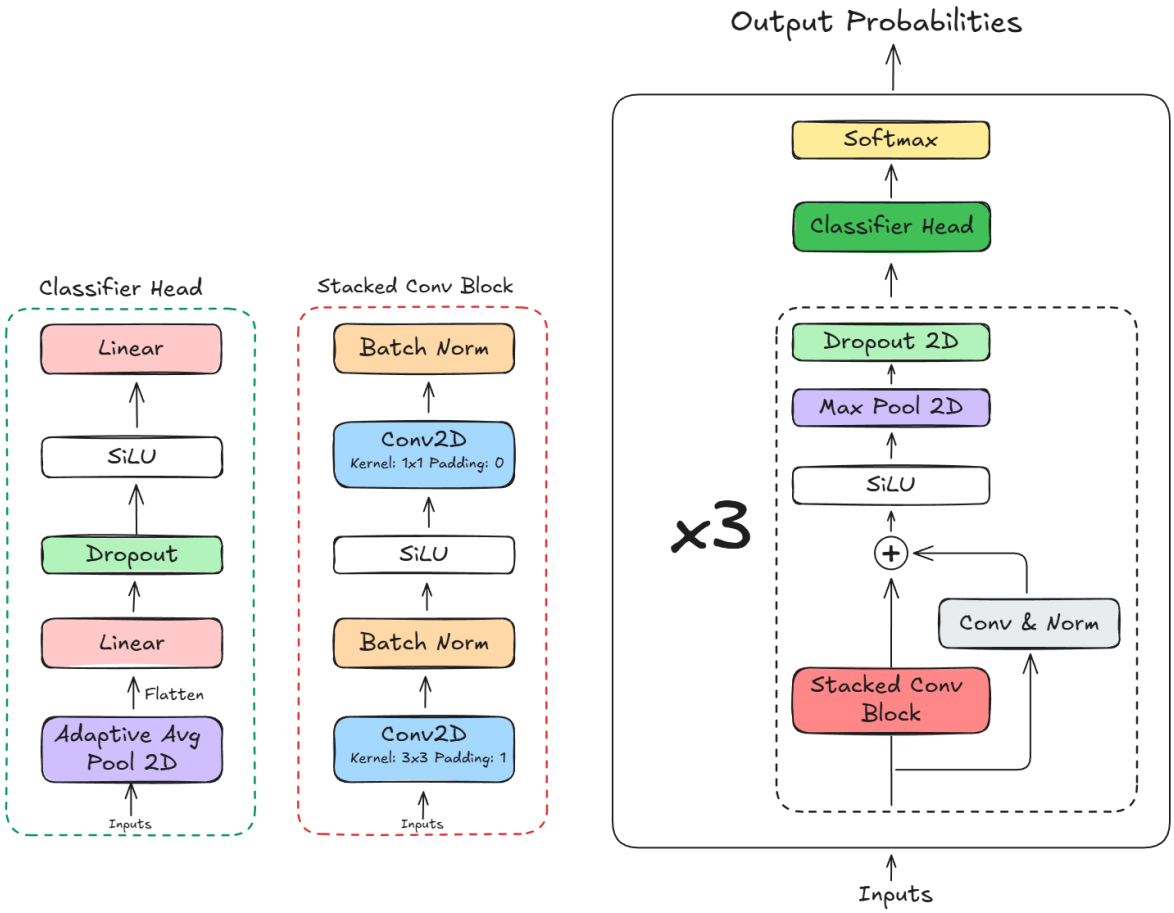


Figure 1: Diagramme simplifié de l'architecture du réseau de neurones

Avant de détailler l'architecture du réseau, il est important de comprendre le rôle de chaque couche qui le compose.

2.1 Couche linéaire

La couche linéaire, également connue sous le nom de couche entièrement connectée ou dense, est l'un des blocs de construction fondamentaux des réseaux de neurones. Son rôle est d'effectuer une transformation affine sur les données d'entrée.

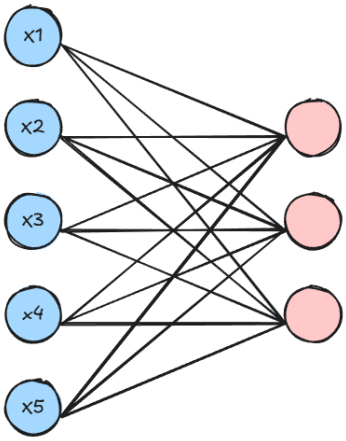


Figure 2: Couche linéaire

2.1.1 Passe avant (Forward Pass)

La passe avant d'une couche linéaire calcule la sortie $\mathbf{Y}$ en appliquant une transformation affine à l'entrée $\mathbf{X}$. Cette transformation est définie par :

$$\mathbf{Y} = \mathbf{X}\mathbf{W} + \mathbf{b}$$

où $\mathbf{W}$ est la matrice des poids et $\mathbf{b}$ le vecteur de biais, qui sont les paramètres apprenables de la couche. L'entrée $\mathbf{X}$ est mise en cache pour être utilisée lors de la passe arrière.

2.1.2 Passe arrière (Backward Pass)

La passe arrière a pour but de calculer les gradients de la fonction de perte L par rapport aux entrées et aux paramètres de la couche. Ces calculs se basent sur le gradient de la perte par rapport à la sortie, $\frac{\partial L}{\partial \mathbf{Y}}$, qui est fourni par la couche suivante. En appliquant la règle de dérivation en chaîne, on obtient les gradients suivants :

- **Gradient par rapport à l'entrée** ($\frac{\partial L}{\partial \mathbf{X}}$): Ce gradient est propagé à la couche précédente.

$$\frac{\partial L}{\partial \mathbf{X}} = \frac{\partial L}{\partial \mathbf{Y}} \mathbf{W}^T$$

- **Gradient par rapport aux poids** ($\frac{\partial L}{\partial \mathbf{W}}$): Utilisé pour mettre à jour les poids.

$$\frac{\partial L}{\partial \mathbf{W}} = \mathbf{X}^T \frac{\partial L}{\partial \mathbf{Y}}$$

- **Gradient par rapport au biais** ($\frac{\partial L}{\partial \mathbf{b}}$): Utilisé pour mettre à jour les biais.

$$\frac{\partial L}{\partial \mathbf{b}} = \sum_{\text{batch}} \frac{\partial L}{\partial \mathbf{Y}}$$

2.2 Couche de convolution 2D (Conv2D)

La couche de convolution 2D est la pierre angulaire des réseaux de neurones convolutifs, spécialisée dans la détection de caractéristiques locales dans les données d'entrée, telles que les images. Elle

fonctionne en faisant glisser un ou plusieurs filtres (ou noyaux, *kernels*) sur l'image d'entrée. Chaque filtre est une petite matrice de poids apprenables.

À chaque position, la couche effectue un produit scalaire entre les poids du filtre et la région correspondante de l'image. Ce processus génère une carte de caractéristiques (*feature map*), qui indique la présence de la caractéristique que le filtre est conçu pour détecter (par exemple, des contours verticaux, des coins, ou des motifs de couleur). En utilisant plusieurs filtres, un CNN peut apprendre à extraire une riche hiérarchie de caractéristiques, des plus simples aux plus complexes.

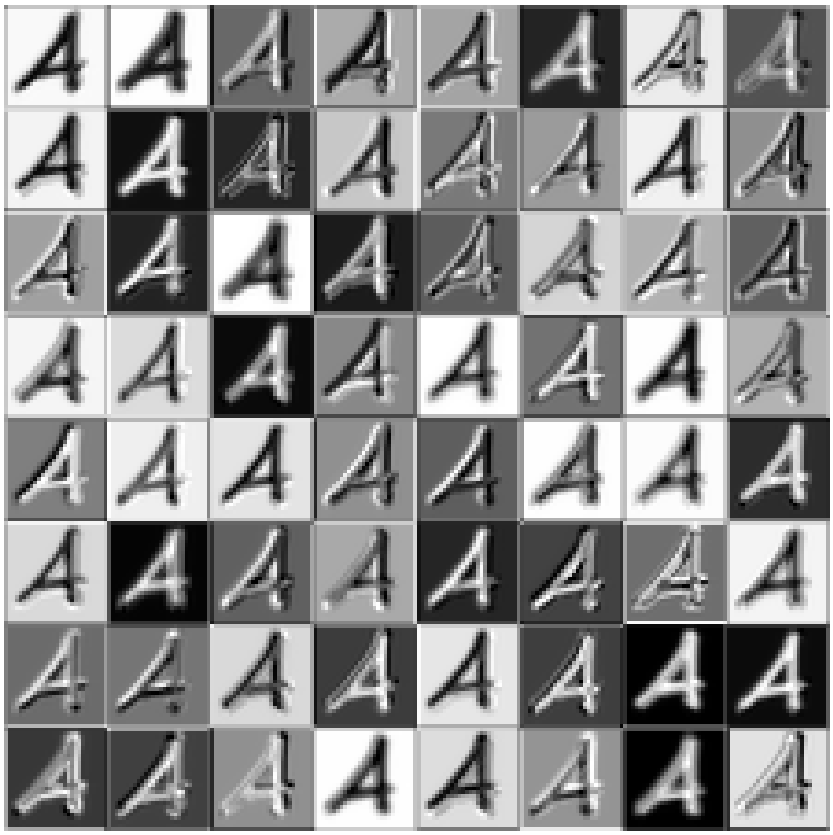


Figure 3: Exemple de carte de caractéristiques

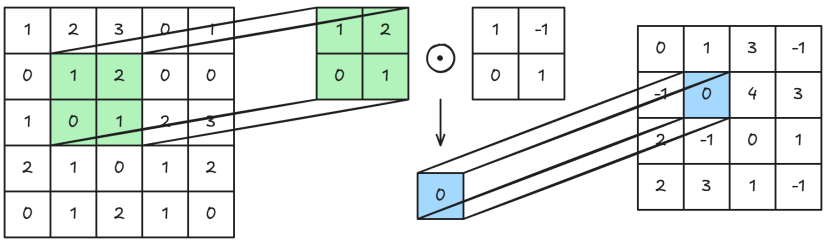


Figure 4: Exemple de convolution 2D

2.2.1 Passe avant (Forward Pass)

Soit une entrée $\mathbf{X}$ de forme (C_{in}, H_{in}, W_{in}) et une couche de convolution avec C_{out} filtres de taille (K_H, K_W), avec un pas (*stride*) S et un remplissage (*padding*) P. La sortie $\mathbf{Y}$ de forme $(C_{out}, H_{out}, W_{out})$

est calculée comme suit pour chaque carte de caractéristiques de sortie c_{out} aux coordonnées (i, j):

$$\mathbf{Y}_{c_{out},i,j} = \left(\sum_{c_{in}=0}^{C_{in}-1} \sum_{k_h=0}^{K_H-1} \sum_{k_w=0}^{K_W-1} \mathbf{X}'_{c_{in},i\cdot S+k_h,j\cdot S+k_w} \cdot \mathbf{K}_{c_{out},c_{in},k_h,k_w} \right) + \mathbf{b}_{c_{out}}$$

Où $\mathbf{X}'$ est l'entrée après application du padding, $\mathbf{K}$ est le tenseur des poids des filtres, et $\mathbf{b}$ est le vecteur de biais. Les dimensions de la sortie sont données par :

$$H_{out} = \left\lfloor \frac{H_{in} - K_H + 2P}{S} \right\rfloor + 1$$

$$W_{out} = \left\lfloor \frac{W_{in} - K_W + 2P}{S} \right\rfloor + 1$$

2.2.2 Passe arrière (Backward Pass)

La passe arrière calcule les gradients de la fonction de perte par rapport aux entrées et aux paramètres de la couche, en utilisant la règle de dérivation en chaîne et le gradient de la couche suivante, $\frac{\partial L}{\partial \mathbf{Y}}$.

- **Gradient par rapport au biais ($\frac{\partial L}{\partial \mathbf{b}}$)**: C'est la somme des gradients de la carte de caractéristiques de sortie.

$$\frac{\partial L}{\partial b_{c_{out}}} = \sum_{i=0}^{H_{out}-1} \sum_{j=0}^{W_{out}-1} \left(\frac{\partial L}{\partial \mathbf{Y}} \right)_{c_{out},i,j}$$

- **Gradient par rapport aux poids ($\frac{\partial L}{\partial \mathbf{K}}$)**: Ce gradient est obtenu en convoluant l'entrée $\mathbf{X}$ avec le gradient de la sortie $\frac{\partial L}{\partial \mathbf{Y}}$.

$$\left(\frac{\partial L}{\partial \mathbf{K}} \right)_{c_{out},c_{in},k_h,k_w} = \sum_{i=0}^{H_{out}-1} \sum_{j=0}^{W_{out}-1} \mathbf{X}'_{c_{in},i\cdot S+k_h,j\cdot S+k_w} \left(\frac{\partial L}{\partial \mathbf{Y}} \right)_{c_{out},i,j}$$

- **Gradient par rapport à l'entrée ($\frac{\partial L}{\partial \mathbf{X}}$)**: Ce calcul est plus complexe et correspond à une "convolution transposée" (parfois appelée déconvolution). Il s'agit d'une convolution entre les filtres $\mathbf{K}$ (pivotés de 180 degrés) et le gradient de sortie $\frac{\partial L}{\partial \mathbf{Y}}$, en tenant compte du pas et du remplissage.

Lorsque la technique `im2col` est utilisée, ces opérations de passe arrière se réduisent également à des multiplications de matrices, préservant ainsi l'efficacité des calculs.

2.2.3 Optimisation algorithmique de la convolution

Une implémentation naïve de la convolution avec des boucles imbriquées est extrêmement inefficace. Pour accélérer les calculs, notre approche se concentre sur l'optimisation directe de l'opération de convolution. L'objectif est de maximiser l'utilisation des calculs et d'exploiter efficacement la hiérarchie de la mémoire.

Les noyaux de convolution 1x1 et 3x3 étant les plus fréquents dans notre architecture, des stratégies d'optimisation spécifiques ont été développées pour ces deux cas. Ces stratégies visent à exploiter le parallélisme des données à un grain fin et à distribuer la charge de travail sur plusieurs cœurs de processeur.

Noyaux 1x1 : Pour les convolutions 1x1, l'opération se simplifie en une multiplication par accumulation sur les canaux d'entrée, où chaque pixel peut être traité indépendamment. Cette propriété permet un très haut degré de parallélisme. L'implémentation tire parti de cette indépendance pour traiter de grands blocs de pixels simultanément, ce qui réduit considérablement la surcharge liée aux boucles et maximise le débit de calcul.

Noyaux 3x3 : Les convolutions 3x3 sont plus complexes à optimiser en raison de la dépendance aux pixels voisins. Notre approche divise le problème :

- Le calcul pour les **pixels intérieurs** de l'image, où le noyau de convolution ne dépasse pas les limites, est effectué par un chemin de code hautement optimisé qui traite plusieurs pixels de sortie en parallèle.
- Le calcul pour les **pixels sur les bords**, qui nécessitent une gestion attentive du remplissage, est géré par un chemin de code distinct et plus général pour assurer l'exactitude.

Cette séparation permet de paralléliser massivement le traitement de la grande majorité des pixels (l'intérieur) tout en gérant correctement les cas particuliers des bords.

2.2.4 Autres pistes d'optimisation

Au-delà de l'implémentation directe, plusieurs algorithmes avancés existent pour accélérer les convolutions, chacun avec ses propres compromis.

im2col + GEMM : Une approche très répandue consiste à transformer les patchs de l'image d'entrée en colonnes d'une matrice (`im2col`). L'opération de convolution est alors réduite à une unique multiplication de cette matrice avec la matrice des poids du noyau, une opération connue sous le nom de GEMM (General Matrix-Matrix Multiplication) pour laquelle il existe des bibliothèques hautement optimisées. Cependant, cette méthode présente un inconvénient majeur : la transformation `im2col` duplique les données de l'image d'entrée, ce qui entraîne une consommation de mémoire significativement plus élevée. De plus, pour des noyaux de petite taille, comme 3x3, le coût de cette réorganisation de la mémoire peut parfois dépasser les gains de performance obtenus par la multiplication de matrices, rendant cette approche moins efficace que des algorithmes directs bien optimisés.

Algorithme de Winograd : L'algorithme de Winograd est une autre technique particulièrement efficace pour les convolutions avec de petits noyaux, notamment 3x3, qui sont omniprésents dans les architectures de réseaux de neurones modernes. Plutôt que de transformer l'opération en une grande multiplication de matrices, Winograd utilise une transformation linéaire pour réduire considérablement le nombre de multiplications arithmétiques requises. Par exemple, pour un noyau 3x3 sur une tuile de sortie de 2x2, il est possible de réduire le nombre de multiplications de 36 à 16, soit une accélération théorique de 2.25x. Cette réduction se fait au détriment d'un plus grand nombre d'additions et de transformations de données, mais comme les multiplications sont beaucoup plus coûteuses que les additions sur le matériel moderne, le gain de performance est substantiel.

Convolution à base de FFT : Pour les noyaux de grande taille, la convolution peut être calculée de manière très efficace en utilisant la Transformée de Fourier Rapide (FFT). Selon le théorème de convolution, une convolution dans le domaine spatial est équivalente à une multiplication élément par élément dans le domaine fréquentiel. L'algorithme consiste donc à transformer l'image et le noyau dans le domaine fréquentiel via une FFT, à effectuer la multiplication, puis à retransformer le résultat dans le domaine spatial via une FFT inverse. Cependant, le coût des transformations FFT est élevé, ce qui rend cette méthode inefficace pour les petits noyaux.

2.3 Couche de Normalisation par Lots (Batch Normalization)

La normalisation par lots est une technique essentielle pour l'entraînement des réseaux de neurones profonds. Son objectif principal est de réduire le *changement de covariable interne* , un phénomène où la distribution des entrées de chaque couche change au fur et à mesure que les paramètres des couches précédentes sont mis à jour. En stabilisant cette distribution, la normalisation par lots permet d'accélérer la convergence, d'utiliser des taux d'apprentissage plus élevés et de régulariser le modèle.

Note:

Dans le contexte de l'apprentissage automatique, la convergence désigne l'état, lors de l'entraînement, où les performances du modèle sur l'ensemble de données d'entraînement cessent de s'améliorer. Ce phénomène s'observe généralement lorsque la valeur de la fonction de perte se stabilise, indiquant que le modèle a appris les motifs dans les données au mieux de ses capacités avec la configuration actuelle.

2.3.1 Passe avant (Forward Pass)

En mode entraînement : Pour un mini-lot de données $\mathcal{B} = \{x_1, \ldots, x_m\}$, la couche calcule d'abord la moyenne $\mu_\mathcal{B}$ et la variance $\sigma^2_\mathcal{B}$ de ce lot.

$$\mu_\mathcal{B} = \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma^2_\mathcal{B} = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2$$

Chaque entrée du lot est ensuite normalisée :

$$\hat{x}_i = \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma^2_\mathcal{B} + \epsilon}}$$

où ϵ est une petite constante ajoutée pour la stabilité numérique. Finalement, une transformation affine est appliquée à l'aide de deux paramètres entraînables, γ (échelle) et β (décalage), pour préserver la capacité de représentation du réseau :

$$y_i = \gamma\hat{x}_i + \beta$$

Durant l'entraînement, la couche maintient également une moyenne mobile de la moyenne et de la variance, qui seront utilisées pendant l'inférence.

En mode inférence : Au lieu de calculer la moyenne et la variance sur le lot courant, la couche utilise les moyennes mobiles accumulées durant l'entraînement pour normaliser les entrées. Cela garantit que la sortie est déterministe pour une entrée donnée.

Note:

Les moyennes mobiles sont une technique statistique permettant d'estimer de manière stable la moyenne et la variance globales sur l'ensemble des données d'entraînement. Plutôt que de simplement moyenner les statistiques de chaque lot, une moyenne mobile exponentielle est utilisée. À chaque nouveau lot, les moyennes mobiles sont mises à jour en intégrant une fraction des statistiques du lot courant, contrôlée par un facteur de momentum. Cette approche donne plus de poids aux lots récents tout en conservant une mémoire des lots passés, aboutissant à une estimation robuste qui est ensuite utilisée pour la normalisation en mode inférence.

2.3.2 Passe arrière (Backward Pass)

La passe arrière calcule les gradients de la fonction de perte par rapport aux paramètres γ et β, ainsi que par rapport à l'entrée de la couche x. Le calcul du gradient par rapport à l'entrée est complexe, car la sortie de chaque neurone dépend de l'ensemble des entrées du mini-lot à travers la moyenne et la variance. La règle de dérivation en chaîne est appliquée pour propager correctement les gradients à travers l'opération de normalisation.

En supposant que nous ayons le gradient de la perte par rapport à la sortie de la couche de normalisation par lots, $\frac{\partial L}{\partial y_i}$, les gradients sont calculés comme suit :

- **Gradient par rapport au décalage ($\frac{\partial L}{\partial \beta}$)**:

$$\frac{\partial L}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial L}{\partial y_i}$$

- **Gradient par rapport à l'échelle ($\frac{\partial L}{\partial \gamma}$)**:

$$\frac{\partial L}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial L}{\partial y_j} \hat{x}_i$$

- **Gradient par rapport à l'entrée ($\frac{\partial L}{\partial x_i}$)**:

$$\frac{\partial L}{\partial x_i} = \frac{\gamma}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \left(\frac{\partial L}{\partial y_i} - \frac{1}{m} \sum_{j=1}^{m} \frac{\partial L}{\partial y_j} - \frac{\hat{x}_i}{m} \sum_{j=1}^{m} \frac{\partial L}{\partial y_j} \hat{x}_j \right)$$

2.4 Couche de Dropout

Le *dropout* est une technique de régularisation puissante et simple pour les réseaux de neurones, conçue pour lutter contre le surapprentissage (*overfitting*).

Note: *Le surapprentissage, ou overfitting, se produit lorsque le réseau apprend "par cœur" les données d'entraînement, y compris leur bruit, au lieu d'apprendre à généraliser à de nouvelles données.*

Le dropout prévient ce phénomène en forçant le réseau à ne pas trop dépendre de neurones spécifiques.

L'idée centrale du dropout est de désactiver aléatoirement un certain nombre de neurones pendant la phase d'entraînement. À chaque itération, chaque neurone a une probabilité p (le *dropout rate*) d'être temporairement "abandonné", c'est-à-dire que sa sortie est mise à zéro.

2.4.1 Passe avant (Forward Pass)

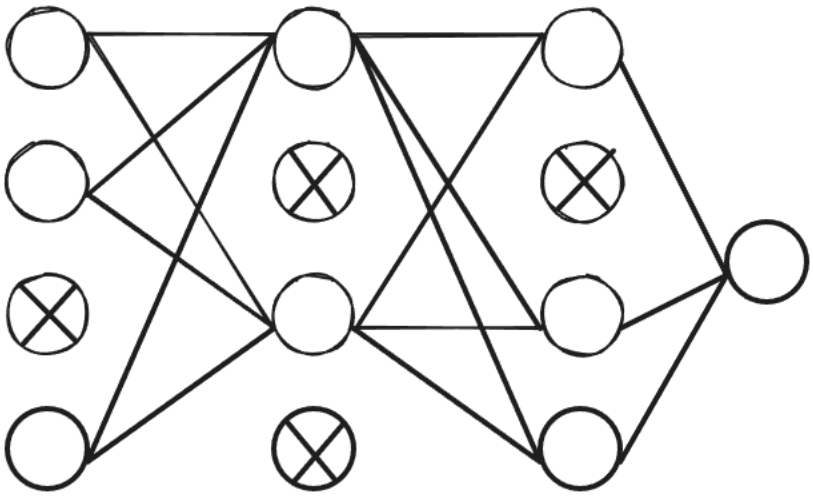


Figure 5: Exemple de dropout

En mode entraînement : Durant la passe avant, pour chaque lot de données, une "masque" de dropout est généré. Ce masque est un tenseur de la même forme que l'entrée, contenant des 0 et des 1. Un neurone est désactivé si la valeur correspondante dans le masque est 0. Pour compenser la désactivation d'une partie des neurones et conserver une espérance de sortie constante, les activations des neurones restants sont mises à l'échelle. La technique la plus courante, appelée *inverted dropout*, consiste à diviser les activations des neurones conservés par $1-p$. L'opération peut se résumer ainsi :

$$y_i = \begin{cases} 0 & \text{avec probabilité } p \\ \frac{x_i}{1-p} & \text{avec probabilité } 1-p \end{cases}$$

Où x_i est l'entrée d'un neurone et y_i sa sortie. Cette mise à l'échelle durant l'entraînement a l'avantage de ne nécessiter aucune modification durant la phase d'inférence.

En mode inférence : Lors de l'évaluation du modèle, le dropout est désactivé. Tous les neurones sont utilisés, et leurs sorties ne sont pas modifiées. Grâce à la technique de l'*inverted dropout*, la sortie du réseau a déjà la bonne échelle.

2.4.2 Passe arrière (Backward Pass)

Pendant la passe arrière, le gradient est simplement propagé à travers les neurones qui n'ont pas été désactivés. Les neurones "abandonnés" ne reçoivent aucun gradient et ne participent donc pas à la mise à jour des poids pour cette itération. Ceci est réalisé en appliquant le même masque de dropout ($\mathbf{M}$) utilisé lors de la passe avant au gradient de la sortie:

$$\frac{\partial L}{\partial \mathbf{X}} = \frac{\partial L}{\partial \mathbf{Y}} \odot \mathbf{M}$$

Où $\odot$ représente la multiplication élément par élément.

2.5 Couche de Dropout 2D (Dropout2D)

Le Dropout 2D est une variante de la technique de dropout spécifiquement adaptée aux couches de convolution. Alors que le dropout standard désactive des neurones individuels de manière indépendante, cette approche n'est pas toujours optimale pour les données d'image traitées par des CNNs. Dans les cartes de caractéristiques (*feature maps*) produites par les couches convolutives, les pixels adjacents sont souvent fortement corrélés. La désactivation de pixels individuels de manière aléatoire introduit un bruit qui peut être facilement compensé par les pixels voisins, réduisant ainsi l'efficacité de la régularisation.

Pour remédier à ce problème, le Dropout 2D désactive des cartes de caractéristiques entières. Au lieu de tirer un nombre aléatoire pour chaque pixel, un seul est tiré pour chaque canal de l'entrée pour chaque exemple dans le lot.

2.5.1 Passe avant (Forward Pass)

En mode entraînement : Pour une entrée de forme (B, C, H, W), où B est la taille du lot, C le nombre de canaux, H la hauteur et W la largeur, un masque de dropout est généré avec une forme $(B, C, 1, 1)$. Pour chaque exemple du lot et chaque canal, une décision est prise : soit le canal entier est mis à zéro (avec une probabilité p), soit il est conservé et mis à l'échelle par $1/(1-p)$. Ce masque est ensuite diffusé (*broadcasted*) sur les dimensions spatiales (hauteur et largeur) de l'entrée.

En mode inférence : Comme pour le dropout standard, la couche Dropout 2D est désactivée pendant l'inférence. L'entrée est transmise sans modification.

2.5.2 Passe arrière (Backward Pass)

La passe arrière suit le même principe que la passe avant. Le gradient est propagé uniquement à travers les canaux qui ont été conservés. Le même masque de dropout 2D ($\mathbf{M}$) est appliqué au gradient de la sortie avant de le propager à la couche précédente.

$$\frac{\partial L}{\partial \mathbf{X}} = \frac{\partial L}{\partial \mathbf{Y}} \odot \text{broadcast}(\mathbf{M})$$

2.6 Couche de Max-Pooling 2D (MaxPool2D)

La couche de Max-Pooling 2D est une opération de sous-échantillonnage (*downsampling*) généralement placée après une couche de convolution. Son rôle est de réduire la dimension spatiale des cartes de caractéristiques, ce qui présente plusieurs avantages :

- **Réduction de la complexité :** En diminuant la taille des données, elle réduit le nombre de paramètres et la charge de calcul dans les couches suivantes du réseau.
- **Invariance aux translations :** Elle rend le réseau plus robuste aux petites translations des caractéristiques dans l'image. En ne conservant que la valeur maximale dans une région, la position exacte de cette caractéristique devient moins importante.
- **Extraction des caractéristiques dominantes :** Elle aide à conserver les caractéristiques les plus saillantes (celles avec les activations les plus fortes) tout en écartant les informations moins pertinentes.

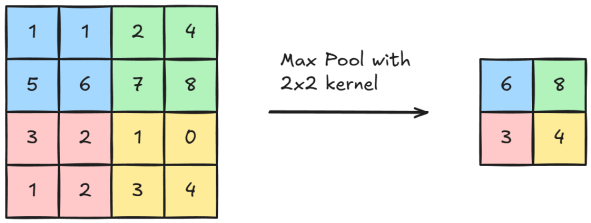


Figure 6: Exemple de Max-Pooling 2D avec un noyau 2x2 et un pas de 2

2.6.1 Passe avant (Forward Pass)

L'opération de Max-Pooling consiste à faire glisser une fenêtre de taille (K_H, K_W) sur chaque carte de caractéristiques d'entrée. Pour chaque position de la fenêtre, seule la valeur maximale des éléments qu'elle recouvre est conservée et transmise à la carte de caractéristiques de sortie.

Soit une entrée de forme (C, H_{in}, W_{in}), les dimensions de la sortie (C, H_{out}, W_{out}) sont calculées de la même manière que pour une couche de convolution, en fonction de la taille du noyau (K_H, K_W), du pas S et du remplissage P :

$$H_{out} = \left\lfloor \frac{H_{in} + 2P_H - K_H}{S_H} \right\rfloor + 1$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2P_W - K_W}{S_W} \right\rfloor + 1$$

Notre implémentation supporte également un mode *ceil*, qui arrondit au supérieur au lieu de l'inférieur, ce qui peut parfois être utile pour ne pas perdre d'informations sur les bords.

2.6.2 Passe arrière (Backward Pass)

La passe arrière du Max-Pooling est particulière car l'opération n'est pas une fonction linéaire continue. Le gradient de la couche de sortie est simplement propagé à l'élément qui a été sélectionné comme maximum lors de la passe avant. Tous les autres éléments de la fenêtre de pooling reçoivent un gradient de zéro, car ils n'ont pas contribué à la sortie.

Le processus est le suivant : pour chaque fenêtre de pooling de la passe avant, on identifie la position de la valeur maximale. Le gradient de la sortie correspondant est alors ajouté au gradient de l'entrée à cette position précise.

Pour optimiser ce processus, notre implémentation conserve en mémoire les indices des maxima lors de la passe avant. Cela évite d'avoir à recalculer leurs positions lors de la passe arrière, accélérant ainsi significativement l'entraînement.

2.7 Couche de mise en commun moyenne adaptative 2D (Adaptive Average Pooling 2D)

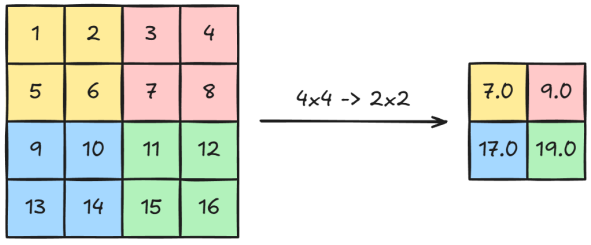


Figure 7: Example de mise en commun moyenne adaptive de 4x4 en 2x2

La couche de mise en commun moyenne adaptative 2D est une opération de sous-échantillonnage qui permet de redimensionner une carte de caractéristiques à une taille de sortie cible fixe, quelle que soit sa taille d'entrée. Contrairement au Max-Pooling standard, qui utilise une fenêtre de taille et de pas fixes, l'Adaptive Average Pooling calcule dynamiquement la taille des régions de mise en commun en fonction des dimensions de l'entrée et de la sortie souhaitée.

Cette propriété est particulièrement utile dans les architectures de réseaux de neurones qui doivent traiter des images de tailles variables. En garantissant une sortie de dimension fixe, cette couche permet de connecter de manière transparente les couches convolutives à des couches entièrement connectées, qui exigent une entrée de taille constante.

Dans notre cas, bien que nous traitions des images de taille prédéfinie, cette couche reste essentielle pour aplatir la sortie des blocs convolutifs en un vecteur de taille fixe pour les couches linéaires finales.

2.7.1 Passe avant (Forward Pass)

Pour redimensionner la carte de caractéristiques d'entrée à la taille de sortie souhaitée, la couche la divise conceptuellement en une grille de régions rectangulaires. Le nombre de régions dans cette grille est égal au nombre de pixels de la sortie désirée (par exemple, pour une sortie 2x2, l'entrée est divisée en 4 régions).

La valeur de chaque pixel de la carte de sortie est simplement la moyenne de toutes les valeurs contenues dans la région correspondante de l'entrée. La taille de ces régions est calculée dynamiquement pour couvrir toute l'image d'entrée.

Pour une carte d'entrée $\mathbf{X}$ de dimensions (H_{in}, W_{in}) et une sortie cible (H_{out}, W_{out}), la région de l'entrée correspondant au pixel de sortie (i_{out}, j_{out}) est définie par les indices de début et de fin :

$$h_{start} = \lfloor \frac{i_{out} \cdot H_{in}}{H_{out}} \rfloor, \quad h_{end} = \lceil \frac{(i_{out}+1) \cdot H_{in}}{H_{out}} \rceil$$

$$w_{start} = \lfloor \frac{j_{out} \cdot W_{in}}{W_{out}} \rfloor, \quad w_{end} = \lceil \frac{(j_{out}+1) \cdot W_{in}}{W_{out}} \rceil$$

La valeur du pixel de sortie $\mathbf{Y}_{i_{out},j_{out}}$ est alors la moyenne des valeurs de $\mathbf{X}$ dans cette région :

$$\mathbf{Y}_{i_{out},j_{out}} = \frac{1}{(h_{end}-h_{start}) \cdot (w_{end}-w_{start})} \sum_{h=h_{start}}^{h_{end}-1} \sum_{w=w_{start}}^{w_{end}-1} \mathbf{X}_{h,w}$$

2.7.2 Passe arrière (Backward Pass)

Lors de la passe arrière, le gradient de la fonction de perte par rapport à la sortie de la couche est propagé aux entrées qui ont contribué à cette sortie. Comme l'opération de la passe avant est une moyenne, le gradient d'un pixel de sortie est distribué uniformément à tous les pixels du bac d'entrée correspondant.

Pour un pixel de sortie donné $\mathbf{Y}_{i_{out},j_{out}}$, le gradient $\frac{\partial L}{\partial \mathbf{Y}_{i_{out},j_{out}}}$ est divisé par la taille du bac d'entrée, $N = (h_{end} - h_{start}) \cdot (w_{end} - w_{start})$. Ce gradient réparti est ensuite ajouté au gradient de chaque pixel $\mathbf{X}_{h,w}$ situé dans ce bac :

$$\frac{\partial L}{\partial \mathbf{X}_{h,w}} += \frac{1}{N} \frac{\partial L}{\partial \mathbf{Y}_{i_{out},j_{out}}}$$

Les gradients pour les pixels d'entrée qui n'ont pas contribué à cette sortie restent inchangés par cette opération. L'entrée de la passe avant est mise en cache pour retrouver facilement les dimensions des bacs lors de cette étape.

2.8 Fonction de perte : Entropie Croisée (Cross-Entropy Loss)

La fonction de perte d'entropie croisée est une mesure de performance fondamentale pour les tâches de classification en apprentissage automatique. Elle quantifie la différence entre deux distributions de probabilités : la distribution prédite par le modèle et la distribution réelle. Pour un problème de classification multi-classes, elle est presque toujours utilisée avec la fonction Softmax, qui convertit les scores bruts du réseau (`logits`) en une distribution de probabilités.

2.8.1 La fonction Softmax

La fonction Softmax prend en entrée un vecteur de scores $\mathbf{z}$ et le transforme en un vecteur de probabilités $\hat{\mathbf{y}}$, où chaque élément est compris entre 0 et 1 et la somme de tous les éléments est égale à 1. Pour un score z_i, la probabilité correspondante est calculée comme suit :

$$\hat{y}_i = \text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

où K est le nombre total de classes.

Stabilité numérique : Le calcul de e^{z_j} peut entraîner des problèmes de stabilité numérique si les scores z_j sont très grands (risque de débordement) ou très petits (risque de sous-dépassement). Pour pallier ce problème, une astuce courante consiste à soustraire la valeur maximale des scores à chaque score avant l'exponentiation :

$$\text{Softmax}(z_i) = \frac{e^{z_i - \max(\mathbf{z})}}{\sum_{j=1}^{K} e^{z_j - \max(\mathbf{z})}}$$

Cette transformation ne change pas le résultat final mais garantit que les arguments de la fonction exponentielle restent dans une plage de valeurs raisonnable.

2.8.2 Passe avant (Forward Pass)

Une fois les probabilités calculées via la fonction Softmax, la perte d'entropie croisée pour un unique exemple est définie comme le logarithme négatif de la probabilité prédite pour la classe correcte :

$$L_i = -\log(\hat{y}_{i,c})$$

où $\hat{y}_{i,c}$ est la probabilité prédite par le modèle pour l'échantillon i appartenant à la classe correcte c. La perte totale pour un lot de données est alors la moyenne des pertes de chaque exemple :

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i = -\frac{1}{N} \sum_{i=1}^{N} \log(\hat{y}_{i,c_i})$$

où N est la taille du lot et c_i est la classe correcte pour l'exemple i.

2.8.3 Passe arrière (Backward Pass)

L'un des avantages de combiner la fonction Softmax avec la perte d'entropie croisée est que le calcul du gradient par rapport aux scores d'entrée (avant Softmax) devient remarquablement simple et efficace. Le gradient de la perte L par rapport à un score d'entrée z_j est donné par :

$$\frac{\partial L}{\partial z_j} = \hat{y}_j - y_j$$

où $\hat{y}_j$ est la probabilité prédite par Softmax pour la classe j, et y_j est la vérité (typiquement 1 si j est la classe correcte, et 0 sinon, ce qui correspond à un encodage "one-hot").

Ce gradient a une interprétation intuitive : il correspond à la différence entre la probabilité prédite et la probabilité réelle. Si la prédiction est parfaite ($\hat{y}_j = y_j$), le gradient est nul. Sinon, le gradient indique la direction dans laquelle les scores doivent être ajustés pour réduire la perte. Le gradient final est ensuite moyenné sur la taille du lot pour rester cohérent avec la perte moyenne calculée lors de la passe avant.

3 Explication de l'architecture utilisée

Nous allons maintenant détailler l'architecture de notre réseau de neurones. Notre réseau est structuré en plusieurs blocs distincts pour un traitement hiérarchique de l'information. Il se compose de trois **blocs de convolution** suivis d'un **bloc de classification**.

Avant de décrire en détail ces blocs, nous allons d'abord présenter la fonction d'activation que nous avons choisie d'utiliser au sein de notre réseau : la fonction SiLU.

3.1 Fonction d'activation : SiLU (Sigmoid Linear Unit)

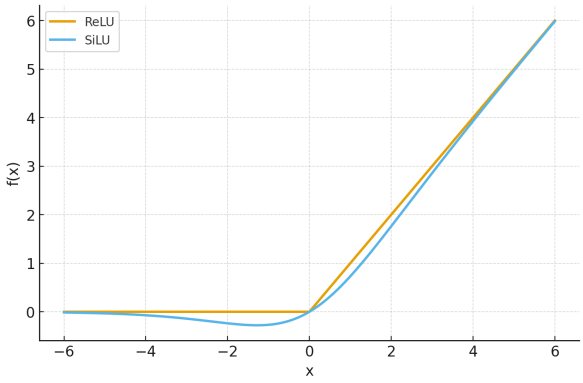


Figure 8: Comparaison entre SiLU et ReLU

La fonction d'activation SiLU, également connue sous le nom de Swish, est une fonction d'activation qui a récemment gagné en popularité en raison de ses performances supérieures à celles de fonctions plus traditionnelles comme ReLU dans de nombreuses architectures de réseaux de neurones profonds.

La fonction SiLU est définie par la formule :

$$\text{SiLU}(x) = x \cdot \sigma(x) = \frac{x}{1 + e^{-x}}$$

où $\sigma(x)$ est la fonction sigmoïde.

Quelques propriétés clés de SiLU expliquent son efficacité :

- **Non-monotonicité :** Contrairement à ReLU, qui est monotone, SiLU présente une légère baisse pour les valeurs négatives avant de converger vers zéro. Cette caractéristique permet à la fonction de produire des activations négatives, ce qui peut aider à la propagation du gradient et à une meilleure expressivité du modèle.
- **Continuité et dérivée lisse :** SiLU est une fonction lisse et continûment dérivable, ce qui la rend plus stable pendant l'optimisation par descente de gradient par rapport aux fonctions comme ReLU qui ont un point non-dérivable en zéro.
- **Auto-régulation :** La fonction sigmoïde agit comme une porte qui module le signal d'entrée x. Pour des valeurs de x très positives, $\sigma(x) \approx 1$ et la sortie est proche de x. Pour des valeurs de x très négatives, $\sigma(x) \approx 0$ et la sortie est proche de zéro. Cette modulation adaptative est considérée comme l'une des raisons de sa bonne performance.
- **Prévention de la "mort" des neurones :** Le fait que la dérivée de SiLU ne soit pas nulle pour les valeurs négatives (contrairement à ReLU) peut aider à atténuer le problème des "neurones morts", où un neurone cesse d'apprendre car son activation est toujours nulle.

3.1.1 Choix par rapport à d'autres fonctions d'activation

Le choix de SiLU se justifie par une comparaison avec d'autres fonctions d'activation couramment utilisées.

ReLU (Rectified Linear Unit)

$$\text{ReLU}(x) = \max(0, x)$$

Bien que simple et efficace, ReLU souffre du problème des "neurones morts" (*dying ReLU*), où les neurones peuvent cesser d'apprendre si leurs entrées deviennent négatives. De plus, sa dérivée est discontinue en zéro. SiLU, en étant lisse et en ayant un gradient non nul pour les valeurs négatives, atténue ces problèmes.

Leaky ReLU

$$\text{Leaky ReLU}(x) = \max(0, x) + \alpha \min(0, x), \quad \alpha > 0$$

Cette fonction tente de résoudre le problème des neurones morts en introduisant une petite pente α pour les valeurs négatives. Cependant, SiLU se distingue par sa non-monotonicité et sa nature auto-régulée, qui peuvent souvent conduire à de meilleures performances en pratique.

Sigmoïde

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Cette fonction est sujette au problème de la saturation et de la disparition du gradient (*vanishing gradient*) dans les réseaux profonds, ce qui ralentit considérablement l'entraînement.

Tanh (Tangente Hyperbolique)

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

Tout comme la sigmoïde, Tanh est sujette à la saturation et à la disparition du gradient, bien que son centrage en zéro soit un avantage par rapport à la sigmoïde. SiLU, comme ReLU, ne sature pas pour les valeurs positives, ce qui facilite un apprentissage plus rapide et plus stable.

GELU (Gaussian Error Linear Unit)

$$\text{GELU}(x) = x \cdot \Phi(x)$$

où $\Phi(x)$ est la fonction de répartition de la loi normale centrée réduite. C'est une autre fonction d'activation moderne et performante, particulièrement populaire dans les modèles de type Transformer (notamment utilisé pour les LLMs). Comme SiLU, elle est lisse et non-monotone. Bien que leurs performances soient souvent très similaires, SiLU peut être légèrement plus efficace sur le plan computationnel car la fonction sigmoïde est moins coûteuse à calculer que la fonction de répartition gaussienne (ou ses approximations) utilisée par GELU.

L'utilisation de SiLU dans notre architecture vise à améliorer la capacité du réseau à apprendre des caractéristiques complexes et à accélérer la convergence lors de l'entraînement.

3.2 Initialisation des Poids

Pour initialiser les poids des couches de convolution et des couches linéaires, nous avons opté pour l'initialisation **Xavier Uniforme** (également connue sous le nom de Glorot Uniform).

Cette méthode tire les poids W d'une distribution uniforme définie comme suit :

$$W \sim \mathcal{U}\left[-\frac{\sqrt{6}}{\sqrt{n_{\text{in}}+n_{\text{out}}}}, \frac{\sqrt{6}}{\sqrt{n_{\text{in}}+n_{\text{out}}}}\right]$$

où n_{in} est le nombre de connexions entrantes et n_{out} le nombre de connexions sortantes de la couche.

L'objectif de cette stratégie est de maintenir la variance des activations et des gradients relativement constante tout au long du réseau. Cela permet d'éviter les problèmes de disparition ou d'explosion du gradient, facilitant ainsi la convergence du modèle, en particulier pour des fonctions d'activation linéaires ou sigmoïdes.

Cependant, il convient de noter qu'avec l'utilisation de la fonction d'activation **SiLU**, l'initialisation de **He** (Kaiming Initialization) aurait été théoriquement plus appropriée. En effet, SiLU partageant des propriétés avec ReLU (notamment le fait d'être non bornée pour les valeurs positives), l'initialisation de Xavier peut ne pas maintenir la variance de manière optimale.

L'initialisation de He (mode Uniforme) est définie par :

$$W \sim \mathcal{U}\left[-\sqrt{\frac{6}{n_{\text{in}}}}, \sqrt{\frac{6}{n_{\text{in}}}}\right]$$

Elle prend en compte la nature rectificatrice des fonctions comme ReLU ou SiLU.

Malgré ce choix sub-optimal, l'impact sur les performances reste limité. Notre réseau n'étant pas extrêmement profond, et grâce à l'utilisation de la normalisation par lots après chaque convolution, les effets de l'initialisation sont atténués et n'entravent pas l'apprentissage.

3.3 Principes de Conception : Blocs Résiduels et Efficacité

Notre architecture est fortement inspirée des principes des réseaux de neurones résiduels (ResNets).

Le Problème des Réseaux Profonds et la Solution Résiduelle Historiquement, l'augmentation de la profondeur d'un réseau de neurones (l'ajout de couches supplémentaires) se heurtait à un obstacle majeur : le problème de la **disparition du gradient** (*vanishing gradient*). Lors de la passe arrière, le gradient est multiplié à chaque couche ; dans un réseau très profond, ce gradient peut devenir si petit qu'il ne permet plus aux premières couches d'apprendre efficacement. Contre-intuitivement, cela menait à un phénomène de **dégradation**, où un réseau profond obtenait de moins bons résultats qu'un réseau moins profond.

Les **connexions résiduelles** (*residual connections* ou *shortcuts*) ont été introduites pour résoudre ce problème. L'idée est de créer un "raccourci" qui permet à l'information (et au gradient) de contourner une ou plusieurs couches.

Soit $\mathcal{H}(x)$ la transformation que l'on souhaite qu'un bloc de couches apprenne. Dans un réseau traditionnel, ces couches tenteraient d'approximer directement $\mathcal{H}(x)$. Dans un bloc résiduel, on reformule le problème. Les couches apprennent plutôt une fonction résiduelle $\mathcal{F}(x)$, et la sortie du bloc est définie comme :

$$\mathbf{y} = \mathcal{F}(\mathbf{x}) + \mathbf{x}$$

Ici, $\mathbf{x}$ est l'entrée du bloc, et $\mathcal{F}(\mathbf{x})$ représente la transformation effectuée par les couches du bloc (par exemple, une séquence de convolutions, de normalisations et d'activations). La connexion qui ajoute $\mathbf{x}$ est la connexion résiduelle.

Si la transformation identité ($\mathcal{H}(x) = x$) est optimale, il est beaucoup plus facile pour le réseau de pousser les poids de $\mathcal{F}(x)$ vers zéro que de forcer les couches à apprendre la fonction identité. Cette formulation facilite la propagation du gradient lors de la passe arrière. En appliquant la règle de dérivation en chaîne, le gradient de la perte L par rapport à $\mathbf{x}$ devient :

$$\frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{y}}\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{y}}\left(\frac{\partial \mathcal{F}(\mathbf{x})}{\partial \mathbf{x}} + 1\right)$$

Le terme '+1' garantit que le gradient peut toujours s'écouler à travers la connexion identité, même si le gradient provenant des couches $\mathcal{F}(\mathbf{x})$ devient très faible. Cela empêche la disparition du gradient et permet d'entraîner des réseaux beaucoup plus profonds sans dégradation des performances.

Choix Architecturaux pour la Performance Le design de notre réseau vise un équilibre entre précision, vitesse d'entraînement et capacité de généralisation.

- **Précision et Généralisation :** L'utilisation de blocs résiduels nous permet de construire un réseau suffisamment profond pour apprendre une hiérarchie de caractéristiques complexe, ce qui est essentiel pour atteindre une haute précision. Ces connexions agissent également comme une forme de régularisation implicite, améliorant la capacité du modèle à généraliser sur des données qu'il n'a jamais vues.
- **Vitesse et Efficacité :**
  - **Convolutions 1x1 :** L'emploi de convolutions 1x1 est une technique clé pour manipuler la profondeur des canaux de manière efficace. Elles permettent de réduire ou d'augmenter le nombre de cartes de caractéristiques avec un coût de calcul bien inférieur à celui des noyaux plus grands. Dans notre cas, elles servent à projeter l'entrée du bloc résiduel dans un espace de plus grande dimension pour correspondre à la sortie du chemin principal.
  - **Petits Noyaux (3x3) :** L'utilisation exclusive de petits noyaux de 3x3 est une pratique courante. L'empilement de deux couches 3x3 a un champ réceptif équivalent à une couche 5x5, mais avec moins de paramètres et une non-linéarité supplémentaire entre les deux.
  - **Global Average Pooling (GAP) :** Avant la classification finale, nous utilisons une couche de GAP au lieu d'aplatir directement les cartes de caractéristiques. Cette technique réduit considérablement le nombre de paramètres par rapport à une couche entièrement connectée traditionnelle, ce qui diminue fortement le risque de surapprentissage et allège le modèle.

3.4 Détail de l'architecture

Le réseau prend en entrée des lots d'images en niveaux de gris de taille `28x28`. La forme d'un tenseur d'entrée est donc `(B, 1, 28, 28)`, où `B` est la taille du lot (nombre d'images traitées simultanément), `1` représente le canal unique (niveaux de gris), et `(28, 28)` sont les dimensions spatiales de l'image.

3.4.1 Bloc Convolutif 1

Ce premier bloc a pour rôle d'extraire des caractéristiques de bas niveau, telles que les contours et les textures simples, directement depuis l'image d'entrée. Il augmente également la profondeur (nombre de canaux) de 1 à 64, tout en réduisant la dimension spatiale de moitié, passant de 28x28 à 14x14. Une connexion résiduelle est utilisée pour faciliter le flux de gradient.

- **Entrée :** `(B, 1, 28, 28)`
- **Chemin principal :**
  - `Conv2d` : 32 filtres, noyau 3x3, pas 1, remplissage 1. Sortie : `(B, 32, 28, 28)`
  - `BatchNorm2d` : sur 32 canaux.
  - `SiLU` : fonction d'activation.
  - `Conv2d` : 64 filtres, noyau 1x1, pas 1, remplissage 0. Sortie : `(B, 64, 28, 28)`
  - `BatchNorm2d` : sur 64 canaux.
- **Chemin résiduel (Shortcut) :**

- `Conv2d` : 64 filtres, noyau 1x1, pas 1, remplissage 0. Sortie : `(B, 64, 28, 28)`
- `BatchNorm2d` : sur 64 canaux.

- **Opérations post-résiduelles :**
  - Addition du chemin principal et du chemin résiduel.
  - `SiLU` : fonction d'activation.
  - `MaxPool2d` : noyau 2x2, pas 2. Sortie : `(B, 64, 14, 14)`
  - `Dropout2D` : taux de 10%.
- **Sortie :** `(B, 64, 14, 14)`

3.4.2 Bloc Convolutif 2

Ce deuxième bloc s'appuie sur les caractéristiques extraites par le premier pour apprendre des motifs plus complexes. Il double à nouveau la profondeur des canaux (de 64 à 128) et réduit encore la dimension spatiale de moitié, de 14x14 à 7x7. La structure est similaire au premier bloc, avec une connexion résiduelle pour préserver l'information.

- **Entrée :** `(B, 64, 14, 14)`
- **Chemin principal :**
  - `Conv2d` : 64 filtres, noyau 3x3, pas 1, remplissage 1. Sortie : `(B, 64, 14, 14)`
  - `BatchNorm2d` : sur 64 canaux.
  - `SiLU` : fonction d'activation.
  - `Conv2d` : 128 filtres, noyau 1x1, pas 1, remplissage 0. Sortie : `(B, 128, 14, 14)`
  - `BatchNorm2d` : sur 128 canaux.
- **Chemin résiduel (Shortcut) :**
  - `Conv2d` : 128 filtres, noyau 1x1, pas 1, remplissage 0. Sortie : `(B, 128, 14, 14)`
  - `BatchNorm2d` : sur 128 canaux.
- **Opérations post-résiduelles :**
  - Addition du chemin principal et du chemin résiduel.
  - `SiLU` : fonction d'activation.
  - `MaxPool2d` : noyau 2x2, pas 2. Sortie : `(B, 128, 7, 7)`
  - `Dropout2D` : taux de 10%.
- **Sortie :** `(B, 128, 7, 7)`

3.4.3 Bloc Convolutif 3

Le troisième et dernier bloc convolutif affine davantage les caractéristiques pour capturer des informations sémantiques de plus haut niveau. La profondeur est de nouveau doublée, passant à 256 canaux, tandis que la résolution spatiale reste à 7x7. Ce bloc n'inclut pas de couche de pooling, car la réduction finale des dimensions spatiales sera gérée par la couche de mise en commun moyenne adaptative.

- **Entrée :** `(B, 128, 7, 7)`
- **Chemin principal :**

- `Conv2d` : 128 filtres, noyau 3x3, pas 1, remplissage 1. Sortie : `(B, 128, 7, 7)`
- `BatchNorm2d` : sur 128 canaux.
- `SiLU` : fonction d'activation.
- `Conv2d` : 256 filtres, noyau 1x1, pas 1, remplissage 0. Sortie : `(B, 256, 7, 7)`
- `BatchNorm2d` : sur 256 canaux.

- **Chemin résiduel (Shortcut) :**
  - `Conv2d` : 256 filtres, noyau 1x1, pas 1, remplissage 0. Sortie : `(B, 256, 7, 7)`
  - `BatchNorm2d` : sur 256 canaux.

- **Opérations post-résiduelles :**
  - Addition du chemin principal et du chemin résiduel.
  - `SiLU` : fonction d'activation.

- **Sortie :** `(B, 256, 7, 7)`

3.4.4 Tête de Classification

Cette partie finale du réseau prend les cartes de caractéristiques de haute dimension et les transforme en une prédiction finale. Elle agrège d'abord les informations spatiales, puis utilise des couches entièrement connectées pour effectuer la classification.

- **Entrée :** `(B, 256, 7, 7)`
- **Couches :**
  - `AdaptiveAvgPool2d` : taille de sortie cible 1x1. Sortie : `(B, 256, 1, 1)`
  - Aplatissement (*Flatten*) : transforme le tenseur en un vecteur. Sortie : `(B, 256)`
  - `Linear` (fc1) : 256 neurones d'entrée, 128 neurones de sortie. Sortie : `(B, 128)`
  - `SiLU` : fonction d'activation.
  - `Dropout` : taux de 25%.
  - `Linear` (fc2) : 128 neurones d'entrée, 26 neurones de sortie (un par lettre). Sortie : `(B, 26)`
- **Sortie Finale :** `(B, 26)`. Ce tenseur contient les scores bruts (*logits*) pour chaque classe, qui sont ensuite passés à la fonction de perte d'entropie croisée pour l'entraînement.

4 Optimisation du modèle : l'optimiseur AdamW

4.1 Le rôle d'un optimiseur dans l'entraînement

Une fois l'architecture du réseau définie et la fonction de perte calculée, le processus d'apprentissage du modèle commence. Ce processus est itératif : le réseau traite un lot de données, effectue une prédiction, et compare cette prédiction à la vérité via la fonction de perte. Cette perte quantifie l'erreur du modèle. L'objectif de l'entraînement est de minimiser cette erreur en ajustant les paramètres (poids et biais) du réseau.

C'est ici qu'intervient l'optimiseur. Un **optimiseur** est un algorithme qui modifie les attributs du réseau neuronal, tels que les poids et le taux d'apprentissage, afin de réduire les pertes. Il utilise le gradient de la fonction de perte par rapport à chaque paramètre (calculé lors de la passe arrière) pour déterminer la direction dans laquelle ajuster ces paramètres. Pour mieux comprendre son rôle, on peut décomposer le problème de la manière suivante :

- **L'espace des paramètres :** Un réseau de neurones contient des millions de paramètres (poids et biais). On peut imaginer un espace abstrait où chaque dimension correspond à un seul de ces paramètres. En raison du grand nombre de paramètres, cet espace est dit "de haute dimension".
- **Une configuration = un point :** Une configuration spécifique du modèle, c'est-à-dire une valeur donnée pour chaque poids et chaque biais du réseau, correspond à un unique point dans cet espace de haute dimension.
- **L'erreur pour chaque configuration :** Pour n'importe lequel de ces points (donc pour n'importe quelle configuration des poids), nous pouvons calculer l'erreur du modèle en lui faisant traiter les données d'entraînement et en évaluant la fonction de perte. Cette erreur est une valeur numérique unique.

Le but de l'optimiseur est donc d'explorer cet immense espace de configurations possibles pour trouver le point qui correspond à la plus faible erreur possible (le minimum de la fonction de perte). Le terme "paysage de perte" est une conceptualisation de cette idée : c'est une surface de haute dimension où la position est déterminée par les poids du modèle, et "l'altitude" à chaque position est déterminée par l'erreur. Le travail de l'optimiseur est de trouver la "vallée" la plus profonde de ce paysage.

Le choix de l'optimiseur est crucial car il détermine non seulement la vitesse de convergence du modèle, mais aussi sa performance finale. Pour notre projet, nous avons choisi **AdamW**, une évolution de l'optimiseur Adam, reconnue pour sa robustesse et son efficacité.

4.2 Adam (Adaptive Moment Estimation)

L'optimiseur Adam est une méthode de descente de gradient stochastique populaire due à son efficacité. Il combine deux techniques pour améliorer la convergence :

- **Momentum :** Cette technique vise à accélérer la convergence. Au lieu de se baser uniquement sur le gradient du lot actuel pour mettre à jour les poids, le momentum accumule une moyenne mobile exponentielle des gradients passés. Ce vecteur de "momentum" pousse les mises à jour dans une direction persistante à travers les itérations, ce qui aide à progresser plus rapidement dans les directions de gradient stable et à amortir les oscillations dans les directions où le gradient change fréquemment.
- **Taux d'apprentissage adaptatif (inspiré de RMSprop) :** Cette technique ajuste le taux d'apprentissage pour chaque paramètre individuellement. Elle maintient une moyenne mobile du carré des gradients. Les paramètres qui reçoivent des gradients de grande magnitude verront leur taux d'apprentissage effectif réduit, tandis que ceux avec des gradients de faible magnitude auront un taux d'apprentissage plus élevé. Cela permet de prendre des mesures prudentes pour les paramètres sensibles et des mesures plus audacieuses pour les paramètres moins sensibles.

Adam fusionne ces deux concepts en maintenant une estimation du premier moment (la moyenne, comme le momentum) et du second moment (la variance non centrée, comme RMSprop) des gradients pour calculer des mises à jour de poids efficaces et bien adaptées.

4.3 Le problème de la régularisation L2 dans Adam

Décroissance de poids et régularisation L2. Pour éviter le surapprentissage, on utilise des techniques de régularisation. La **régularisation L2** consiste à ajouter un terme de pénalité à la fonction de perte, proportionnel au carré de la magnitude des poids :

$$L_{total} = L_{original} + \frac{\lambda}{2} \sum_i w_i^2$$

où λ est le coefficient de régularisation. Ce terme a pour effet d'ajouter une composante λw_i au gradient de chaque poids, ce qui les pousse à décroître vers zéro. C'est ce qu'on appelle la **décroissance de poids** (*weight decay*). Cela favorise les modèles avec des poids plus petits, qui sont souvent plus simples et généralisent mieux.

Le couplage indésirable. Pendant longtemps, la régularisation L2 et la décroissance de poids ont été traitées comme interchangeables. Cependant, dans les optimiseurs adaptatifs comme Adam, cette équivalence est rompue. Lorsque la régularisation L2 est ajoutée à la fonction de perte, son gradient (λw_i) est inclus dans le calcul global du gradient. Par conséquent, il est normalisé par le second moment (le terme RMSprop) de l'optimiseur Adam.

Le résultat est que la force de la décroissance de poids devient dépendante de l'historique des gradients d'un poids. Les poids qui ont eu des gradients élevés par le passé se voient appliquer une décroissance de poids *plus faible* que ceux qui ont eu de faibles gradients. Cet effet de couplage affaiblit l'efficacité de la régularisation, car elle n'est plus appliquée uniformément comme prévu.

4.4 AdamW : Le découplage pour une meilleure régularisation

L'optimiseur AdamW (Adam with Decoupled Weight Decay) a été proposé pour corriger cette interaction non désirée. L'idée fondamentale est que la décroissance de poids est une forme de régularisation qui doit être indépendante du processus d'optimisation du gradient.

La solution consiste à découpler les deux mécanismes. La décroissance de poids est retirée du calcul du gradient (le terme de régularisation L2 n'est pas ajouté à la perte). Elle est appliquée directement lors de l'étape de mise à jour des poids :

$$w_{t+1} \leftarrow w_t - \eta \cdot \text{update}_t - \eta \lambda w_t$$

Dans cette formulation, l'étape de mise à jour de l'optimiseur Adam (update_t) et la décroissance de poids (λw_t) sont deux termes distincts. En découplant les deux processus, AdamW garantit que la décroissance de poids agit comme une force de régularisation constante et prévisible, telle qu'elle a été conçue à l'origine. Cette approche s'est avérée plus efficace, menant souvent à une meilleure généralisation du modèle et à de meilleures performances finales.

4.5 L'algorithme AdamW en détail

Le pseudocode ci-dessous illustre le mécanisme interne d'AdamW, mettant en évidence la séparation entre la mise à jour des poids basée sur le gradient et l'étape de décroissance de poids.

Algorithm 1 Pseudocode de l'optimiseur AdamW.

Require: η (Taux d'apprentissage global)
Require: λ (Taux de décroissance de poids)
Require: $\beta_1, \beta_2 \in [0, 1)$ (Facteurs d'amortissement des moments)
Require: θ_0 (Paramètres initiaux)

Initialisation :
$m_0 \leftarrow 0$ ▷ Vecteur du premier moment
$v_0 \leftarrow 0$ ▷ Vecteur du second moment
$t \leftarrow 0$ ▷ Compteur de pas de temps

while θ_t n'a pas convergé **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta L_t(\theta_{t-1})$ ▷ Obtenir le gradient
    $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$ ▷ Mise à jour du premier moment
    $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$ ▷ Mise à jour du second moment
    $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ ▷ Correction du biais
    $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ ▷ Correction du biais
    $\theta_t \leftarrow \theta_{t-1} - \eta \lambda \theta_{t-1}$ ▷ **Décroissance de poids découplée**
    $\theta_t \leftarrow \theta_t - \eta \left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \right)$ ▷ Mise à jour des poids
end while
return θ_t ▷ Les paramètres optimisés

Note: *Le pseudocode présenté applique la décroissance de poids avant la mise à jour du gradient. Dans certaines implémentations, l'ordre est inversé. Bien que cela puisse avoir un impact subtil sur le taux d'apprentissage effectif pour les poids, le concept fondamental du découplage est préservé.*

5 Entraînement du Modèle

Le processus d'entraînement est l'étape cruciale où le réseau de neurones apprend à reconnaître les caractères en ajustant ses millions de paramètres de manière itérative.

5.1 Préparation des Données et Environnement

La qualité et la préparation des données sont fondamentales pour le succès de l'entraînement. Plutôt que d'utiliser un jeu de données standard tel que EMNIST dans sa forme brute, nous avons opté pour une approche plus robuste en générant synthétiquement notre propre ensemble de données à l'aide d'un script dédié. L'objectif était de créer un corpus d'images plus riche et plus difficile, simulant les imperfections du monde réel que l'on trouve dans les documents imprimés et numérisés.

Le processus de génération a commencé par le rendu de caractères alphabétiques (minuscules et majuscules) à partir d'une vaste collection de polices de caractères courantes (Serif, Sans-serif, Monospace, ...). Cette étape initiale a permis de garantir une grande diversité dans les formes de base des lettres. Par la suite, une chaîne d'augmentations sophistiquées a été appliquée à chaque image de base pour simuler une variété de dégradations :

- **Effets de Papier et d'Impression :** Pour imiter le support physique et les imperfections de l'impression.
  - *Froissement et Plis :* Simulation de papier froissé avec des ombres pour créer un effet de relief et des déformations locales.
  - *Jaunissement du Papier :* Application d'une teinte jaune subtile pour simuler le vieillissement naturel du papier.
  - *Bavure d'Encre :* Dilatation légère des contours des caractères pour imiter l'encre s'étalant sur les fibres du papier.
  - *Grain de Papier :* Ajout d'un bruit de fond à basse fréquence pour recréer la texture inhérente au papier.
- **Distorsions Géométriques :** Pour simuler des numérisations imparfaites et des documents mal positionnés.
  - *Rotation :* Inclinaison de l'image entière pour simuler un document mal aligné sur le scanner.
  - *Perspective :* Déformation de l'image comme si elle était photographiée sous un angle, et non à plat.
  - *Transformations Affines :* Combinaison de mises à l'échelle (zoom avant/arrière), de translations, de rotations et de cisaillements pour une grande variété de distorsions linéaires.
- **Bruit et Flou :** Pour simuler des conditions de numérisation de faible qualité ou des problèmes d'équipement.
  - *Bruit Gaussien et Multiplicatif :* Ajout de bruit aléatoire pour simuler les imperfections des capteurs électroniques du scanner.
  - *Flou Gaussien et de Mouvement :* Application de flous pour simuler des problèmes de mise au point ou un léger mouvement du document pendant la numérisation.
  - *Défocalisation :* Imitation d'un effet de profondeur de champ ou d'une mauvaise mise au point générale.
- **Déformations Avancées et Artefacts :** Pour simuler des dégradations plus complexes du papier ou de l'optique.

– *Distorsion en Grille et Élastique :* Déformations non linéaires qui étirent et contractent localement l'image pour simuler du papier ondulé ou déformé.

– *Distorsion Optique :* Simulation des aberrations de lentille (comme un léger effet "fisheye") qui peuvent se produire avec des appareils photo ou des scanners de mauvaise qualité.

– *Netteté et Masque de Flou :* Application de filtres pour accentuer ou adoucir artificiellement les contours, simulant différents types de post-traitement logiciel.

– *Artefacts de Scanner :* Simulation de pertes d'information locales en supprimant des zones rectangulaires ou en grille de l'image (*dropout*).

Ce processus a permis de générer un ensemble de données final composé de **164,424 images pour l'entraînement** et **29,016 pour les tests**.

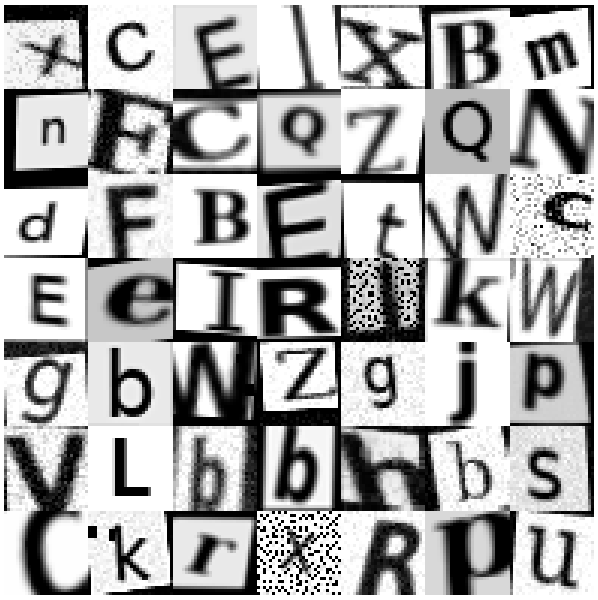


Figure 9: Un aperçu des images du jeu de données d'entraînement synthétique.

Enfin, avant d'être injectées dans le réseau, les images subissent une étape de **normalisation**. Les valeurs des pixels, initialement comprises dans l'intervalle [0, 1], sont transformées pour être dans l'intervalle [-1, 1]. Cette normalisation centre les données autour de zéro, ce qui aide à stabiliser et accélérer la convergence du modèle.

La durée totale de l'entraînement pour atteindre les performances optimales a été de **4 heures, 39 minutes et 44 secondes**.

5.2 Hyperparamètres

- **Époques (Epochs) :** L'entraînement a été effectué sur **12 époques**. Une époque correspond à un passage complet sur l'intégralité du jeu de données d'entraînement.
- **Taille de Lot (Batch Size) :** Nous avons utilisé une taille de lot de **64**. Le traitement des données par lots permet une estimation plus stable du gradient tout en étant efficace sur le plan computationnel.
- **Optimiseur :** Comme détaillé dans la section précédente, nous avons utilisé **AdamW**, avec des paramètres standards pour les moments ($\beta_1 = 0.9$, $\beta_2 = 0.999$) et une petite constante $\epsilon = 1e-8$ pour la stabilité numérique.

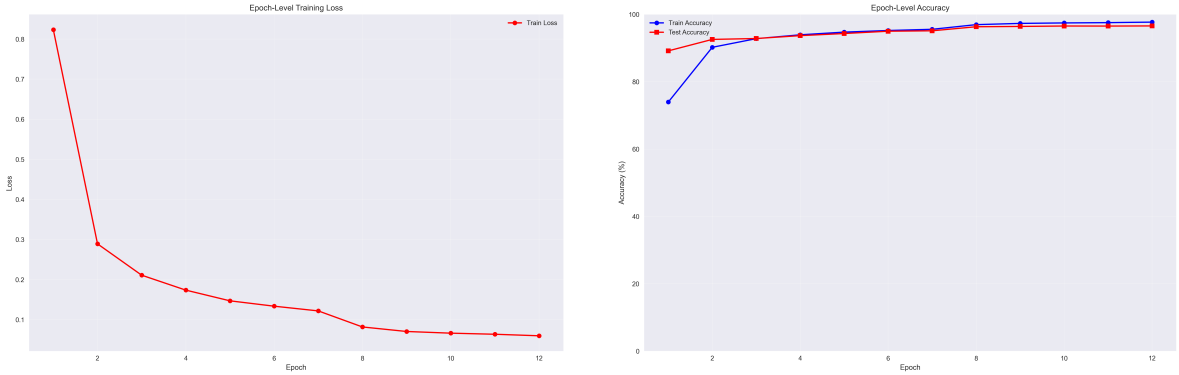
- **Taux d'Apprentissage Initial :** Le taux d'apprentissage a été initialisé à **0.001**. C'est une valeur de départ courante qui offre un bon compromis entre une convergence rapide et le risque de dépasser un minimum local.
- **Décroissance de Poids (Weight Decay) :** Une valeur de **0.0001** a été utilisée pour la décroissance de poids découplée d'AdamW.

5.3 Analyse de la Performance du Modèle

Le suivi de la perte (loss) et de la précision (accuracy) est essentiel pour évaluer la convergence du modèle.

5.3.1 Évolution par Époque

Une vue d'ensemble de l'apprentissage est obtenue en analysant les performances à la fin de chaque époque.

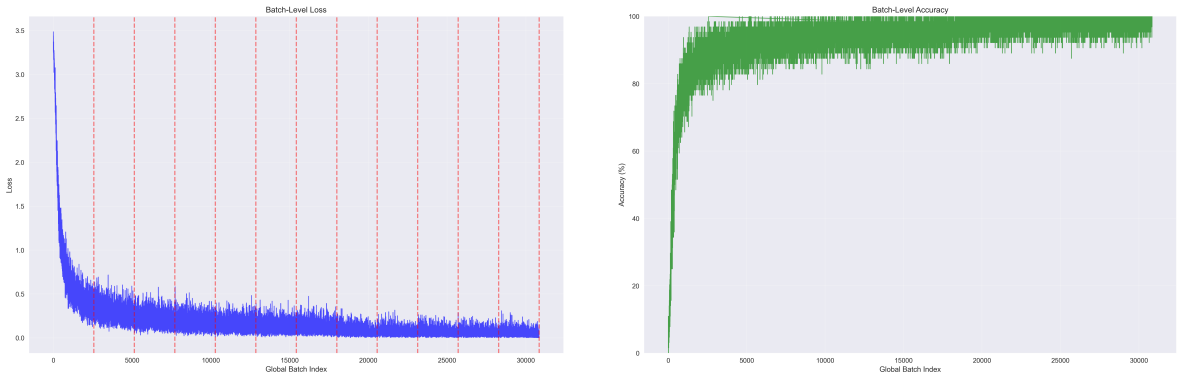


(a) Évolution de la perte par époque. (b) Évolution de la précision par époque.

Figure 10: Performance du modèle au fil des époques.

5.3.2 Analyse au Niveau des Lots

Une analyse plus fine peut être menée en observant les métriques au niveau des lots (batches).



(a) Perte par lot. (b) Précision par lot.

Figure 11: Performance du modèle au niveau des lots.

5.4 Stratégies d'Optimisation de l'Entraînement

Au-delà du choix de l'optimiseur, des stratégies supplémentaires ont été mises en œuvre pour améliorer la robustesse et l'efficacité de l'entraînement.

5.4.1 Ordonnancement du Taux d'Apprentissage (Learning Rate Scheduling)

Le taux d'apprentissage est l'un des hyperparamètres les plus importants. Un taux trop élevé peut empêcher la convergence, tandis qu'un taux trop faible peut la ralentir considérablement. Un **ordonnanceur de taux d'apprentissage** est une technique qui ajuste dynamiquement cette valeur au cours de l'entraînement.

Nous avons utilisé un ordonnanceur de type **StepLR**. Cette stratégie maintient le taux d'apprentissage constant pendant un certain nombre d'époques, puis le réduit d'un facteur multiplicatif. Dans notre configuration, le taux d'apprentissage était réduit d'un facteur de **0.1** (gamma) toutes les **7 époques**. L'avantage de cette approche est double :

- Au début, un taux d'apprentissage relativement élevé (0.001) permet au modèle de converger rapidement vers une bonne région du paysage de perte.
- Plus tard, un taux d'apprentissage plus faible permet au modèle d'explorer cette région plus finement pour se rapprocher d'un minimum optimal, sans "rebondir" hors de la vallée.

Le pseudo-code suivant illustre le mécanisme de mise à jour du taux d'apprentissage par l'ordonnanceur StepLR.

Algorithm 2 Pseudo-code de l'ordonnanceur StepLR

Require: $lr_{initial}$ (Taux d'apprentissage initial)
Require: $step\_size$ (Fréquence de la mise à jour, en époques)
Require: γ (Facteur de réduction du taux d'apprentissage)
function GetLearningRate($epoch$)
    $power \leftarrow \lfloor (epoch - 1)/step\_size \rfloor$
    $lr_{new} \leftarrow lr_{initial} \times \gamma^{power}$
    return lr_{new}
end function

5.4.2 Prévention du Surapprentissage et Arrêt Précoce (Early Stopping)

Le surapprentissage est un risque majeur où le modèle mémorise les données d'entraînement au lieu d'apprendre à généraliser. Pour contrer cela, nous avons mis en place une stratégie d'**arrêt précoce (Early Stopping)**.

Cette technique consiste à surveiller les performances du modèle sur un ensemble de données de validation (dans notre cas, l'ensemble de test) à la fin de chaque époque. Si les performances sur cet ensemble n'ont pas montré d'amélioration pendant un certain nombre d'époques consécutives (un paramètre appelé **patience**, que nous avons fixé à 10), l'entraînement est interrompu. Le modèle final retenu est celui qui a obtenu les meilleures performances sur l'ensemble de validation, et non nécessairement celui de la toute dernière époque. Cette méthode offre deux avantages majeurs :

- Elle agit comme une forme de régularisation en empêchant le modèle de trop se spécialiser sur les données d'entraînement.
- Elle peut économiser un temps de calcul considérable en terminant l'entraînement dès que le modèle cesse de s'améliorer.

5.5 Résultats Finaux de l'Entraînement

5.5.1 Métriques Finales

L'entraînement s'est terminé avec succès après **12 époques**, atteignant des performances remarquables sur l'ensemble de validation. Le modèle a démontré une capacité d'apprentissage exceptionnelle avec les métriques finales suivantes :

Table 1: Métriques finales du modèle à l'époque 12

Métrique	**Valeur**
Perte d'entraînement (Loss)	0.059935
Précision d'entraînement	97.7230%
Précision de validation	96.5950%

Le modèle a atteint son pic de performance à l'**époque 12**, avec une précision de validation de **96.60%**.

5.6 Considérations sur le Matériel et le Temps d'Entraînement

L'entraînement du modèle a été réalisé sur un processeur **Intel Xeon Platinum 8488C**. Le choix de ce matériel de qualité serveur n'est pas anodin : il a été motivé par la recherche d'une stabilité maximale durant les longs processus de calcul. En effet, une durée d'entraînement de plus de 4 heures est considérable et s'explique en grande partie par la complexité inhérente à l'optimisation d'un réseau de neurones. Malgré de nombreuses optimisations, le temps de traitement par lot (batch) se situait autour de **500 ms**.

À titre de comparaison, des tests préliminaires sur des processeurs grand public (**I7 11800H**) ont montré des temps par lot non seulement plus élevés, oscillant entre 700 et 1000 ms, mais aussi beaucoup plus instables, ce qui aurait pu compromettre allonger le temps d'entraînement.

5.7 Défis Rencontrés et Analyse des Résultats sur Données Réelles

Malgré des métriques de validation impressionnantes, les performances du modèle sur des données réelles se sont avérées quelque peu décevantes. Nous suspectons que cette baisse de performance est due à une implémentation perfectible des couches de convolution (*Conv2D*), qui ne parviendraient pas à extraire les caractéristiques les plus pertinentes des images de manière aussi efficace que prévu.

Cette hypothèse est renforcée par une comparaison avec un prototype développé sous **PyTorch**, un framework d'apprentissage automatique de référence. Avec une architecture strictement identique, la version PyTorch a montré une bien meilleure généralisation sur des données réelles, en particulier sur la distinction de caractères notoirement difficiles comme les paires **O**/**Q** ou le groupe **I**/**T**/**L**. Or, notre jeu de données synthétique avait été spécifiquement enrichi et affiné pour forcer le modèle à bien différencier ces cas ambigus. Le fait que le modèle en C éprouve des difficultés là où la version PyTorch réussit suggère que le problème ne réside ni dans l'architecture ni dans les données, mais bien dans les détails de l'implémentation bas-niveau des opérations de convolution.

6 Le Solveur : Une Approche par Appariement de Patrons Probabilistes

Le cœur du solveur est un algorithme de recherche de mots mêlés qui est une variation de l'algorithme d'appariement de patrons probabilistes (*probabilistic template matching, (PTM)*). Dans cette approche, chaque mot à trouver est un patron, mais ni le patron ni la grille de recherche ne sont définis de manière déterministe. Ils sont représentés par des distributions de probabilités sur l'alphabet. Cette méthode est cruciale pour gérer l'incertitude inhérente à la reconnaissance de caractères effectuée par le Réseau de Neurones Convolutif (CNN).

6.1 Représentation Probabiliste

Les entrées du solveur ne sont pas de simples tableaux de caractères, mais des tenseurs qui capturent les probabilités pour chaque caractère, telles que déterminées par le réseau de neurones.

- **La Grille** : La grille de mots mêlés est représentée par un tenseur tridimensionnel de forme `(hauteur, largeur, 26)`. Pour chaque cellule à la position (r, c), il y a un vecteur de 26 probabilités, où le k-ième élément représente la probabilité que le caractère dans cette cellule soit la k-ième lettre de l'alphabet. Soit $G_{r,c,k}$ cette probabilité, alors pour toute cellule donnée (r, c):

$$\sum_{k=0}^{25} G_{r,c,k} = 1$$

- **Les Mots (Patrons)** : De même, chaque mot à trouver est représenté comme un patron probabiliste sous la forme d'un tenseur bidimensionnel de forme `(longueur_mot, 26)`. Pour la i-ème lettre du mot, il y a un vecteur de 26 probabilités. Soit $W_{i,k}$ la probabilité que la i-ème lettre soit le k-ième caractère de l'alphabet.

$$\sum_{k=0}^{25} W_{i,k} = 1$$

Ce cadre probabiliste permet au solveur d'exploiter la sortie complète du CNN, plutôt que seulement sa prédiction la plus confiante (l'`argmax`), ce qui conduit à une recherche plus robuste et précise.

6.2 Le Mécanisme de Notation : Entropie Croisée

Pour trouver le meilleur emplacement pour un patron (un mot), il nous faut une mesure pour quantifier la qualité de la correspondance entre une séquence de cellules de la grille et le patron. L'algorithme utilise une méthode de notation basée sur l'entropie croisée, un concept fondamental de la théorie de l'information.

6.2.1 Log-Probabilités pour la Stabilité Numérique

Pour calculer la probabilité totale qu'un mot corresponde, il faudrait multiplier les probabilités individuelles de chaque lettre. Cependant, la multiplication répétée de nombres inférieurs à 1 (les probabilités) engendre un résultat si minuscule qu'il risque de dépasser la précision de l'ordinateur et d'être interprété comme zéro. Ce phénomène est connu sous le nom de sous-passement numérique (*underflow*). Pour éviter cela, l'algorithme travaille dans le domaine logarithmique. La somme des log-probabilités est utilisée à la place du produit des probabilités :

$$\log(P_1 \times P_2 \times \cdots \times P_L) = \sum_{i=1}^{L} \log(P_i)$$

Cette transformation est numériquement stable et efficace.

6.2.2 Score Lettre-à-Cellule via l'Entropie Croisée

Le cœur de la notation est le calcul d'un score qui compare la distribution de probabilité d'une lettre du mot-patron avec celle d'une cellule de la grille. Pour ce faire, l'algorithme s'appuie sur le concept d'entropie croisée de la théorie de l'information.

En théorie de l'information, la "surprise" est une mesure formelle de l'improbabilité d'un événement. Un événement à haute probabilité est peu surprenant, tandis qu'un événement rare est très surprenant. Dans notre contexte, l'entropie croisée évalue à quel point les probabilités de la grille sont "surprenantes" du point de vue des probabilités attendues du mot. Une bonne correspondance est une correspondance où la surprise est faible.

Par exemple, si le patron est quasi certain que la lettre est un 'A', on s'attend à ce que la probabilité du 'A' dans la cellule de la grille soit également élevée. Si c'est le cas, la surprise est faible, et le score est élevé. Inversement, si la probabilité du 'A' dans la grille est très faible, la surprise est grande, ce qui résulte en un score très bas.

Le score $S(r, c, i)$ pour la i-ème lettre du mot à la position (r, c) de la grille est donc défini comme :

$$S(r, c, i) = \sum_{k=0}^{25} W_{i,k} \cdot \log(G_{r,c,k}) \tag{1}$$

Cette approche est particulièrement robuste aux données bruitées issues du CNN. Plutôt que de prendre une décision binaire basée sur la lettre la plus probable (un 'argmax'), le score prend en compte l'ensemble de la distribution de probabilité. Ainsi, même si la lettre correcte n'est pas la plus probable mais qu'elle a tout de même une probabilité non négligeable, elle contribuera positivement au score. Cela rend l'algorithme résistant aux erreurs occasionnelles et au manque de confiance du modèle de reconnaissance.

Ici, $W_{i,k}$ agit comme la distribution "vraie" (celle de notre patron) et $\log(G_{r,c,k})$ est la distribution que nous évaluons. Le score est maximisé lorsque les probabilités $G_{r,c,k}$ sont élevées pour les mêmes lettres k où les probabilités $W_{i,k}$ sont également élevées.

6.2.3 Score de Chemin

Un placement potentiel d'un mot est un chemin de cellules dans la grille. Le score total pour un mot le long d'un chemin est la somme des scores individuels lettre-à-cellule pour chaque lettre du mot le long de ce chemin, ce qui est équivalent à la log-probabilité de la séquence complète.

$$S_{\text{chemin}} = \sum_{i=0}^{L-1} S(r_i, c_i, i) \tag{2}$$

où (r_i, c_i) est la coordonnée de la grille correspondant à la i-ème lettre du mot pour le chemin donné.

6.3 Implémentation et Complexité de l'Algorithme

1. **Pré-calcul des Scores** : Pour optimiser la recherche, l'algorithme pré-calcule d'abord un tenseur `letter_scores` de forme `(hauteur, largeur, longueur_mot)`. Ce tenseur stocke le score $S(r, c, i)$ (de l'Équation 1) pour chaque combinaison d'une cellule de grille (r, c) et d'un indice de lettre de mot i. Cette étape évite les calculs redondants. Les log-probabilités de la grille sont également calculées une seule fois.

2. **Recherche Exhaustive** : L'algorithme effectue une recherche exhaustive sur tous les placements de mots possibles. Un placement est défini par une position de départ et une direction.

    - Il itère sur chaque cellule (r, c) de la grille, considérant chacune comme un point de départ potentiel pour le mot.

- Depuis chaque cellule, il cherche dans les 8 directions possibles (horizontale, verticale, et diagonale).

3. **Évaluation et Sélection** : Pour chaque chemin potentiel, l'algorithme vérifie sa validité (s'il reste dans la grille) et calcule son score total en sommant les scores pré-calculés. Il conserve en mémoire la correspondance ayant le score le plus élevé trouvé jusqu'à présent.

Après avoir testé toutes les possibilités, la fonction renvoie le placement du mot avec le score global le plus élevé.

6.3.1 Analyse de la Complexité

Soit H la hauteur de la grille, W sa largeur, L la longueur du mot à chercher, et A la taille de l'alphabet (26).

- Le pré-calcul du tenseur `letter_scores` a une complexité de $\mathcal{O}(H \cdot W \cdot L \cdot A)$, car pour chaque cellule, chaque lettre du mot, nous calculons un produit scalaire sur l'alphabet.
- La phase de recherche a une complexité de $\mathcal{O}(H \cdot W \cdot D \cdot L)$, où D est le nombre de directions (8). Pour chaque point de départ et chaque direction, nous sommons L scores pré-calculés.

La complexité totale est donc dominée par l'étape de pré-calcul : $\mathcal{O}(H \cdot W \cdot L \cdot A)$. Cette approche est efficace car le coût de l'appariement de l'alphabet est payé une seule fois.

6.3.2 Comparaison avec un solveur heuristique

Notre algorithme est un **solveur exact**. Il est comparable à une recherche par force brute : il est plus lent mais garantit de trouver la réponse correcte en évaluant le score final de chaque chemin possible.

Une approche alternative courante pour ce genre de problème utiliserait un arbre de préfixes (Trie) combiné à un algorithme de **recherche en faisceau (Beam Search)**. Cette méthode est un **solveur heuristique** : elle est beaucoup plus rapide, mais n'offre aucune garantie de trouver la meilleure solution. Pour notre projet, ce serait un mauvais choix.

Le problème fondamental est que la recherche en faisceau est un algorithme "glouton", et les algorithmes gloutons peuvent échouer de manière catastrophique lorsque les données d'entrée sont, comme dans notre cas, extrêmement bruitées.

Scénario d'échec de l'approche gloutonne Une recherche en faisceau construit les chemins pour tous les mots potentiels simultanément, lettre par lettre. À chaque nouvelle lettre, elle élague (supprime) les chemins qui semblent les moins prometteurs en se basant sur leur score partiel.

Imaginons que nous cherchons les mots "PYTHON" et "FLAMINGO", et que notre CNN, peu performant, a produit les scores suivants pour le chemin correct de "PYTHON" :

- P (Lettre 1) : Bonne correspondance. Score = -1.5
- Y (Lettre 2) : Correspondance extrêmement bruitée. Score = -10.0
- T (Lettre 3) : Bonne correspondance. Score = -1.2
- H (Lettre 4) : Bonne correspondance. Score = -1.0
- O (Lettre 5) : Bonne correspondance. Score = -1.4
- N (Lettre 6) : Bonne correspondance. Score = -1.1

Le score total, et donc correct, pour "PYTHON" est la somme de ces scores : **-16.2**.

Comment la recherche en faisceau échoue

Étape 1 : Chemins d'une lettre. La recherche commence et trouve plusieurs chemins, qu'elle classe par score. Disons qu'elle conserve les 10 meilleurs :

- Chemin("P") : Score = -1.5 (semble prometteur)
- Chemin("F") : Score = -2.0
- Chemin("A") : Score = -2.2
- ...

Étape 2 : Extension aux chemins de deux lettres. L'algorithme étend les 10 chemins conservés :

- Chemin("FL") : Score("F") + Score("L") = -2.0 + -2.1 = **-4.1** (semble excellent)
- Chemin("AV") : Score("A") + Score("V") = -2.2 + -2.5 = **-4.7** (semble très bon)
- ...
- Chemin("PY") : Score("P") + Score("Y") = -1.5 + (-10.0) = **-11.5** (semble très mauvais)

L'élagage. L'algorithme examine tous ses chemins de deux lettres et décide de ne conserver que les 10 meilleurs. Les chemins "FL" (-4.1) et "AV" (-4.7) sont conservés. Le chemin "PY", avec son score de -11.5, n'est pas dans le top 10. Il est donc élagué et **définitivement supprimé**.

Conclusion La recherche en faisceau ne trouvera jamais "PYTHON". Elle a été trop "gloutonne" : en voyant le mauvais score partiel de "PY", elle a incorrectement supposé que le chemin entier serait mauvais, sans jamais voir que les lettres suivantes ("T", "H", "O", "N") constituaient une excellente correspondance qui aurait mené au véritable meilleur score.

Pourquoi notre algorithme est robuste au bruit La force de notre solveur exact réside dans sa robustesse face à des données bruitées, ce qui est notre cas principal en raison des imperfections du CNN.

L'algorithme n'est pas "glouton" ; il n'abandonne jamais un chemin prématurément. Au lieu de cela, il évalue le score *complet* de chaque chemin candidat avant de prendre une décision :

- Il calcule le score final pour "PYTHON", incluant la lettre bruitée : -16.2.
- Il calcule le score final pour "FLAMINGO" (par exemple) : -25.0.
- Ce n'est qu'à la toute fin qu'il compare les scores finaux et choisit "PYTHON".

L'échec ponctuel et très bruité sur la lettre "Y" est compensé par les bonnes correspondances des autres lettres. En considérant l'évidence dans sa totalité, l'algorithme permet aux signaux forts de l'emporter sur le bruit. C'est cette capacité à intégrer toute l'information, sans jugement prématuré, qui le rend particulièrement adapté à notre problématique.

Bibliographie

References

[1] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft, 2006.

[2] Stefan Elfwing, Eiji Uchibe, and Kenji Doya. Sigmoid-weighted linear units for neural network function approximation in reinforcement learning. In *Neural Networks*, volume 107, pages 3–11. Elsevier, 2018.

[3] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[5] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning (ICML)*, pages 448–456, 2015.

[6] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4013–4021, 2016.

[7] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[8] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations (ICLR)*, 2019.