Fiche Méthodes

Neural Networks and Deep Learning

Module : ST2DLS8-DE

Un recueil de méthodes pour accompagner vos révisions

Doryan Denis

Version du document	**v1.0**
Première édition	17 juillet 2026
Dernière mise à jour	17 juillet 2026

Table des matières

1 Fondamentaux du perceptron et de l'entraînement **2**
1.1 Identifier les caractéristiques et le fonctionnement des réseaux de neurones profonds 2
1.2 Calculer la sortie d'un réseau de neurones par propagation avant (forward pass) 3

2 Rétropropagation et descente de gradient **5**
2.1 Calculer les gradients par rétropropagation (règle des deltas) 5
2.2 Mettre à jour les poids par descente de gradient (batch, stochastique, mini-batch) 7

3 Régularisation et optimisation **8**
3.1 Appliquer la régularisation L_2 (weight decay) à la mise à jour des poids 8

4 Réseaux de neurones convolutifs (CNN) **10**
4.1 Identifier les avantages des CNN pour la classification d'images 10
4.2 Calculer les dimensions de sortie et le nombre de paramètres d'un CNN, couche par couche . . . 11
4.3 Compléter une opération de convolution manuelle (calcul du feature map) 13

5 Modélisation générative profonde (Autoencodeurs, VAE, GAN) **15**
5.1 Identifier les caractéristiques de la modélisation générative profonde 15
5.2 Compléter une architecture d'autoencodeur convolutif (dimensions manquantes) 16
5.3 Compléter la fonction de perte GAN (objectif minimax) 17
5.4 Expliquer la fonction de perte VAE (ELBO, divergence KL) et contraster VAE / autoencodeur standard 18

6 Réseaux de neurones récurrents, attention et Transformers **20**
6.1 Compléter une architecture LSTM en Keras (cohérence des dimensions) 20
6.2 Expliquer pourquoi un SimpleRNN peine sur les dépendances longues et comment le LSTM y remédie 21
6.3 Expliquer la limite du LSTM qui motive le mécanisme d'attention 22
6.4 Identifier les propriétés du mécanisme d'auto-attention (self-attention) 22
6.5 Identifier les avantages du multi-head attention 23

1 Fondamentaux du perceptron et de l'entraînement

1.1 Identifier les caractéristiques et le fonctionnement des réseaux de neurones profonds

Méthode.
Un réseau de neurones profond (DNN) empile plusieurs couches de neurones, chacune appliquant une combinaison linéaire suivie d'une fonction d'activation non linéaire g :

$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i w_i\right)$$

Pour reconnaître un énoncé vrai/faux sur les DNN, il faut vérifier chaque proposition à l'aune des propriétés suivantes :

- **Extraction automatique de features** : contrairement aux méthodes classiques, un DNN apprend directement les caractéristiques pertinentes à partir des données brutes (pas de feature engineering manuel).
- **Non-linéarité** : chaque couche (sauf éventuellement la sortie) applique une activation non linéaire (sigmoïde, tanh, ReLU) — sans quoi l'empilement de couches resterait équivalent à un modèle linéaire.
- **Apprentissage par rétropropagation** : les poids sont ajustés via la descente de gradient et le taux d'apprentissage η.
- **Mini-batchs** : l'entraînement se fait par petits lots pour un compromis rapidité/qualité de l'estimation du gradient.
- **Régularisation** : dropout, arrêt précoce (early stopping), L_2, batch normalization limitent le surapprentissage.
- **Architectures profondes** : des connexions résiduelles (skip connections) peuvent faciliter l'entraînement de réseaux très profonds en évitant la disparition du gradient.

Exemple. Application n°1 — DE 2022/2023, Ex.1
Parmi les propositions suivantes, lesquelles sont des caractéristiques des réseaux de neurones profonds ?
(a) Extraient directement les features à partir des données
(b) Utilisent une activation non linéaire pour traiter des données complexes
(c) Utilisent la rétropropagation et l'ajustement du taux d'apprentissage pour optimiser les poids
(d) Utilisent des mini-batchs pour un entraînement plus rapide et une estimation plus précise des gradients
(e) Utilisent des techniques de régularisation (dropout, early stopping) pour généraliser
(f) Nécessitent un feature engineering manuel explicite

Solution.
On confronte chaque proposition à la méthode ci-dessus :
(a) Vrai — extraction automatique des features, propriété fondamentale des DNN.
(b) Vrai — la non-linéarité est indispensable pour traiter des données non linéairement séparables.
(c) Vrai — c'est le principe même de l'entraînement par descente de gradient.
(d) Vrai — les mini-batchs accélèrent l'entraînement et stabilisent l'estimation du gradient.
(e) Vrai — le dropout et l'early stopping sont des régularisations classiques.
(f) Faux — c'est l'inverse : les DNN évitent le feature engineering manuel.

Réponses correctes : (a), (b), (c), (d), (e)

Exemple. Application n°2 — DE 2024/2025, Ex.4 Q1
Parmi les propositions suivantes sur les DNN, lesquelles sont correctes ?
(a) Ils emploient plusieurs couches de neurones pour extraire automatiquement des représentations de plus en plus abstraites
(b) Ils peuvent intégrer des connexions résiduelles pour faciliter l'entraînement d'architectures très profondes
(c) Ils appliquent des techniques comme la régularisation L_2, le dropout et la batch normalization pour stabiliser l'entraînement et améliorer la généralisation
(d) Ils reposent fortement sur le feature engineering manuel
(e) Ils n'utilisent que des transformations linéaires, les non-linéarités n'étant appliquées qu'en sortie

Solution.
(a) Vrai — c'est le principe de la représentation hiérarchique en profondeur.
(b) Vrai — les connexions résiduelles (ResNet) limitent la disparition du gradient dans les réseaux très profonds.
(c) Vrai — ce sont des techniques de régularisation et de stabilisation usuelles (voir Chapitre 3).
(d) Faux — contraire au principe même des DNN.
(e) Faux — sans non-linéarité à chaque couche, l'empilement resterait équivalent à une seule transformation linéaire.

Réponses correctes : (a), (b), (c)

1.2 Calculer la sortie d'un réseau de neurones par propagation avant (forward pass)

Méthode.
La propagation avant consiste à calculer, couche par couche, la valeur d'activation de chaque neurone à partir des entrées :

1. Pour un neurone k recevant les sorties a_j des neurones de la couche précédente (ou les entrées x_j pour la première couche), on calcule d'abord la combinaison linéaire (sans biais si le réseau n'en a pas) :

$$z_k = \sum_j w_{j,k}\, a_j$$

2. On applique ensuite la fonction d'activation g imposée par l'énoncé :

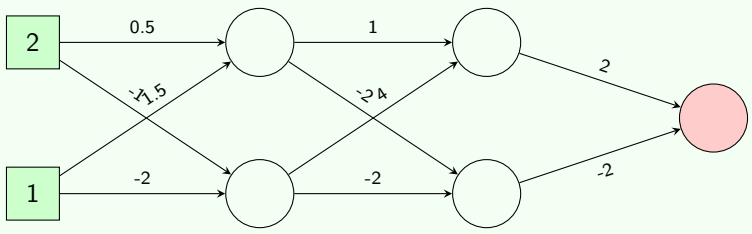
$$a_k = g(z_k)$$

3. On propage couche après couche jusqu'à la sortie.

Fonctions d'activation usuelles :

$$\sigma(z) = \frac{1}{1+e^{-z}} \qquad \mathrm{ReLU}(z) = \max(z, 0)$$

Exemple. Application n°1 — DE 2024/2025, Ex.2
On considère le réseau suivant (sans biais), d'entrée $x = [x_1, x_2] = [2, 1]$:



1. Propager l'entrée avec la fonction sigmoïde $\sigma(x) = \frac{1}{1+e^{-x}}$ comme seule activation, et donner la valeur finale de chaque neurone.
2. Refaire le calcul avec la fonction $\mathrm{ReLU}(x) = \max(x, 0)$ comme seule activation.

Solution.

1. Avec la sigmoïde. On note n_1, n_2 les neurones de la première couche cachée, n_3, n_4 ceux de la seconde, n_5 la sortie.

Couche 1 :

$$z_{n_1} = 0.5 \times 2 + 1.5 \times 1 = 2.5 \quad \Rightarrow \quad a_{n_1} = \sigma(2.5) \approx 0.924$$
$$z_{n_2} = (-1) \times 2 + (-2) \times 1 = -4 \quad \Rightarrow \quad a_{n_2} = \sigma(-4) \approx 0.018$$

Couche 2 :

$$z_{n_3} = 1 \times a_{n_1} + 4 \times a_{n_2} \approx 0.924 + 0.072 = 0.996 \quad \Rightarrow \quad a_{n_3} = \sigma(0.996) \approx 0.730$$

$$z_{n_4} = (-2) \times a_{n_1} + (-2) \times a_{n_2} \approx -1.848 - 0.036 = -1.884 \quad \Rightarrow \quad a_{n_4} = \sigma(-1.884) \approx 0.132$$

Sortie :

$$z_{n_5} = 2 \times a_{n_3} + (-2) \times a_{n_4} \approx 1.460 - 0.264 = 1.197 \quad \Rightarrow \quad \hat{y} = \sigma(1.197) \approx 0.768$$

$\boxed{\hat{y} \approx 0.768 \text{ (sigmoïde)}}$

2. Avec ReLU. Les combinaisons linéaires z sont identiques à celles calculées ci-dessus (elles ne dépendent pas de l'activation) ; seule l'activation change :

$$a_{n_1} = \text{ReLU}(2.5) = 2.5 \qquad a_{n_2} = \text{ReLU}(-4) = 0$$
$$z_{n_3} = 1 \times 2.5 + 4 \times 0 = 2.5 \quad \Rightarrow \quad a_{n_3} = \text{ReLU}(2.5) = 2.5$$
$$z_{n_4} = (-2) \times 2.5 + (-2) \times 0 = -5 \quad \Rightarrow \quad a_{n_4} = \text{ReLU}(-5) = 0$$
$$z_{n_5} = 2 \times 2.5 + (-2) \times 0 = 5 \quad \Rightarrow \quad \hat{y} = \text{ReLU}(5) = 5$$

$\boxed{\hat{y} = 5 \text{ (ReLU)}}$

Remarque. À retenir
Les combinaisons linéaires z_k ne dépendent que des poids et des sorties de la couche précédente : elles sont **indépendantes du choix de l'activation**. Seule l'activation appliquée ensuite change le résultat final — c'est pourquoi on recalcule uniquement les $a_k = g(z_k)$ d'une question à l'autre, sans refaire les produits $w_{j,k}\, a_j$.

2
Rétropropagation et descente de gradient

2.1 Calculer les gradients par rétropropagation (règle des deltas)

Méthode.
La rétropropagation calcule le gradient de la fonction de perte $\mathcal{L}$ par rapport à chaque poids, en propageant l'erreur de la sortie vers l'entrée grâce à la règle de dérivation en chaîne. Pour un réseau dont tous les neurones utilisent la sigmoïde $g = \sigma$ (avec $\sigma'(z) = \sigma(z)(1-\sigma(z)) = a(1-a)$) et une perte d'entropie croisée binaire, la méthode se déroule en trois temps :

1. Passe avant (forward pass) : calculer et conserver l'activation a_k de chaque neurone (cf. méthode 1.2).

2. Passe arrière (calcul des δ) :
- Pour le neurone de sortie o :

$$\delta_o = y - a_o$$

(simplification valable pour la combinaison sigmoïde/softmax + entropie croisée : $\delta = \hat{y} - y$ au signe près selon la convention retenue).
- Pour un neurone caché j, en fonction des δ_k des neurones k de la couche suivante auxquels il est connecté :

$$\delta_j = a_j(1-a_j)\sum_k w_{j,k}\,\delta_k$$

3. Mise à jour des poids : pour un poids $w_{i,j}$ reliant le neurone i au neurone j, avec taux d'apprentissage α :

$$w_{i,j}^{\text{new}} = w_{i,j}^{\text{old}} + \alpha\, a_i\, \delta_j$$

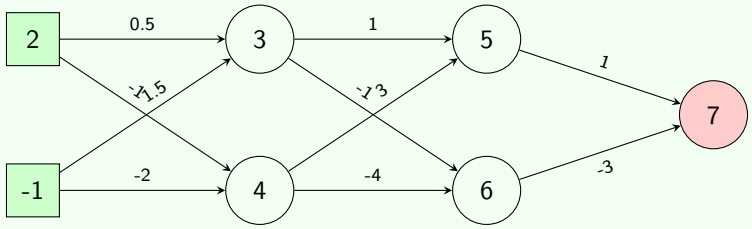
où a_i est l'activation du neurone amont (ou l'entrée x_i si i est un neurone de la couche d'entrée).

Propriété. Règle de dérivation en chaîne appliquée au réseau
Le gradient de la perte par rapport à un poids profond se décompose en un produit de dérivées locales :

$$\frac{\partial \mathcal{L}}{\partial w_{i,j}} = \frac{\partial \mathcal{L}}{\partial a_j} \cdot \frac{\partial a_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_{i,j}} = \delta_j \cdot a_i$$

C'est cette décomposition qui permet de calculer δ_j couche par couche, de la sortie vers l'entrée, sans jamais recalculer le gradient complet à chaque fois.

Exemple. Application — exemple du cours, Chapitre 2
On considère le réseau suivant (sans biais, activation sigmoïde partout), d'entrée $x = [x_1, x_2] = [2, -1]$ et de sortie cible $y = 1$:



Effectuer la passe avant, calculer les δ de chaque neurone, puis mettre à jour tous les poids avec un taux d'apprentissage $\alpha = 0.1$.

Solution.
Étape 1 — Passe avant.

$$a_3 = \sigma(0.5 \times 2 + 1.5 \times (-1)) = \sigma(-0.5) = 0.378$$
$$a_4 = \sigma((-1) \times 2 + (-2) \times (-1)) = \sigma(0) = 0.5$$
$$a_5 = \sigma(1 \times a_3 + 3 \times a_4) = \sigma(1.878) = 0.867$$
$$a_6 = \sigma((-1) \times a_3 + (-4) \times a_4) = \sigma(-2.378) = 0.085$$
$$a_7 = \sigma(1 \times a_5 + (-3) \times a_6) = \sigma(0.612) = 0.648$$

Étape 2 — Calcul des δ (couche par couche, de la sortie vers l'entrée).
Neurone de sortie :

$$\delta_7 = y - a_7 = 1 - 0.648 = 0.352$$

Neurones de la seconde couche cachée :

$$\delta_5 = a_5(1 - a_5) \cdot w_{5,7} \cdot \delta_7 = 0.867(1 - 0.867) \times 1 \times 0.352 = 0.041$$

$$\delta_6 = a_6(1 - a_6) \cdot w_{6,7} \cdot \delta_7 = 0.085(1 - 0.085) \times (-3) \times 0.352 = -0.082$$

Neurones de la première couche cachée :

$$\delta_3 = a_3(1 - a_3)(w_{3,5}\delta_5 + w_{3,6}\delta_6) = 0.378(1 - 0.378)(1 \times 0.041 + (-1) \times (-0.082)) = 0.029$$

$$\delta_4 = a_4(1 - a_4)(w_{4,5}\delta_5 + w_{4,6}\delta_6) = 0.5(1 - 0.5)(3 \times 0.041 + (-4) \times (-0.082)) = 0.113$$

Étape 3 — Mise à jour des poids ($w^{\text{new}} = w^{\text{old}} + \alpha\, a_i\, \delta_j$, avec $\alpha = 0.1$) :

$$w_{1,3} : 0.5 \to 0.5 + 0.1 \times 2 \times 0.029 = 0.506 \qquad w_{1,4} : -1 \to -1 + 0.1 \times 2 \times 0.113 = -0.977$$
$$w_{2,3} : 1.5 \to 1.5 + 0.1 \times (-1) \times 0.029 = 1.497 \qquad w_{2,4} : -2 \to -2 + 0.1 \times (-1) \times 0.113 = -2.011$$
$$w_{3,5} : 1 \to 1 + 0.1 \times 0.378 \times 0.041 = 1.002 \qquad w_{3,6} : -1 \to -1 + 0.1 \times 0.378 \times (-0.082) = -1.003$$
$$w_{4,5} : 3 \to 3 + 0.1 \times 0.5 \times 0.041 = 3.002 \qquad w_{4,6} : -4 \to -4 + 0.1 \times 0.5 \times (-0.082) = -4.004$$
$$w_{5,7} : 1 \to 1 + 0.1 \times 0.867 \times 0.352 = 1.031 \qquad w_{6,7} : -3 \to -3 + 0.1 \times 0.085 \times 0.352 = -2.997$$

Tous les poids se rapprochent, dans le bon sens, de la valeur qui réduit $\mathcal{L}$

Remarque. Pourquoi $\delta_o = y - a_o$?
Pour une sortie sigmoïde combinée à une perte d'entropie croisée binaire $\mathcal{L}(y, a) = -\big(y \log a + (1-y) \log(1-a)\big)$, les dérivées $\frac{\partial \mathcal{L}}{\partial a}$ et $\sigma'(z) = a(1-a)$ se simplifient exactement en $\delta = a - y$ (ou $y - a$ selon la convention de signe choisie pour la mise à jour). C'est cette simplification qui rend la formule de δ_o aussi compacte.

2.2 Mettre à jour les poids par descente de gradient (batch, stochastique, mini-batch)

Méthode.
Une fois les gradients calculés par rétropropagation, plusieurs variantes de descente de gradient permettent de mettre à jour les poids W :

1. Descente de gradient (batch) : calcule le gradient moyen sur tout le jeu de données à chaque itération :

$$W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}, \qquad J(W) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\big(f(x^{(i)}; W), y^{(i)}\big)$$

Précis mais très coûteux sur de grands jeux de données.

2. Descente de gradient stochastique (SGD) : calcule le gradient sur un seul exemple tiré aléatoirement à chaque itération. Rapide mais très bruité.

3. Mini-batchs : calcule le gradient sur un petit lot B d'exemples :

$$\frac{\partial J(W)}{\partial W} = \frac{1}{B} \sum_{k=1}^{B} \frac{\partial J_k(W)}{\partial W}$$

Compromis standard en pratique : estimation plus stable que SGD, coût par itération bien inférieur au batch complet, et parallélisable sur GPU.

Remarque. Optimiseurs adaptatifs
Le taux d'apprentissage η peut être rendu adaptatif (dépendant de l'historique du gradient) plutôt que fixe. Optimiseurs courants : `SGD`, `Adam`, `RMSProp`, `Adagrad`, `Adadelta`. Un η trop petit converge lentement (risque de rester bloqué dans un minimum local) ; un η trop grand fait diverger l'optimisation.

3 Régularisation et optimisation

3.1 Appliquer la régularisation L_2 (weight decay) à la mise à jour des poids

Méthode.
La régularisation L_2 ajoute à la perte d'entraînement une pénalité proportionnelle au carré de la norme des poids, afin de décourager les poids de devenir trop grands (et donc de limiter le surapprentissage) :

1. Perte régularisée :

$$\mathcal{L}_{L_2}(\theta) = \mathcal{L}_{\text{train}}(\theta) + \frac{\lambda}{2}\|\theta\|_2^2, \qquad \|\theta\|_2^2 = \sum_j \theta_j^2$$

où $\lambda \geq 0$ contrôle l'intensité de la régularisation.

2. Gradient du terme de pénalité :

$$\nabla_\theta \left(\frac{\lambda}{2}\|\theta\|_2^2 \right) = \lambda\theta$$

3. Règle de mise à jour : en injectant ce gradient dans la descente de gradient classique,

$$\theta \leftarrow \theta - \eta\nabla_\theta \mathcal{L}_{L_2}(\theta) = (1 - \eta\lambda)\,\theta - \eta\nabla_\theta \mathcal{L}_{\text{train}}(\theta)$$

Chaque poids est donc **décru linéairement** vers zéro (facteur $(1 - \eta\lambda)$) à chaque mise à jour, en plus de la correction liée au gradient de la perte d'entraînement.

Exemple. Application
On entraîne un réseau avec $\lambda = 0.01$ et un taux d'apprentissage $\eta = 0.1$. Un poids vaut actuellement $\theta = 2.0$ et le gradient de la perte d'entraînement (hors pénalité) vaut $\nabla_\theta \mathcal{L}_{\text{train}} = 0.05$. Calculer la nouvelle valeur de θ après une itération de descente de gradient avec régularisation L_2.

Solution.
On applique directement la règle de mise à jour :

$$\theta \leftarrow (1 - \eta\lambda)\,\theta - \eta\,\nabla_\theta \mathcal{L}_{\text{train}} = (1 - 0.1 \times 0.01) \times 2.0 - 0.1 \times 0.05$$

$$= (1 - 0.001) \times 2.0 - 0.005 = 1.998 - 0.005 = 1.993$$

$\boxed{\theta \approx 1.993}$ Le poids diminue légèrement plus vite qu'avec une descente de gradient classique (qui aurait donné $2.0 - 0.1 \times 0.05 = 1.995$) : la différence $1.995 - 1.993 = 0.002$ correspond exactement à la décroissance linéaire $\eta\lambda\theta$ apportée par la pénalité L_2.

Remarque. Pourquoi ne régularise-t-on généralement pas les biais ?
Les biais ne multiplient aucune entrée : ils ne contribuent pas à la complexité du modèle de la même façon que les poids (qui déterminent la sensibilité aux variations des entrées). Pénaliser les biais apporterait donc peu de bénéfice en généralisation, tout en risquant de contraindre inutilement le décalage global du modèle. En pratique, on régularise uniquement les matrices de poids $W^{(\ell)}$, pas les vecteurs de biais $b^{(\ell)}$.

Récap.
Le Chapitre 1 a déjà recensé les grandes familles de régularisation testées sous forme de QCM (méthode 1.1) :
- **Dropout** : désactive aléatoirement une fraction p des neurones à chaque passe avant pendant l'entraînement (équivaut à entraîner un ensemble de sous-réseaux) ; on utilise le réseau complet à l'inférence.
- **Early stopping** : arrête l'entraînement dès que la perte de validation cesse de s'améliorer.
- **Data augmentation** : enrichit artificiellement le jeu de données par des transformations qui préservent le label.
- **Batch normalization** : normalise les activations intermédiaires par mini-batch, stabilisant et accélérant l'entraînement.
- **Régularisation L_2 (weight decay)** : détaillée ci-dessus.

4 Réseaux de neurones convolutifs (CNN)

4.1 Identifier les avantages des CNN pour la classification d'images

Méthode.
Par rapport à un réseau pleinement connecté (qui aplatit l'image en un vecteur et perd toute structure spatiale), un CNN exploite la structure spatiale des images grâce à trois idées clés :

- **Connectivité locale** : chaque neurone d'une couche de convolution ne « voit » qu'un petit patch de l'entrée (son champ récepteur), au lieu d'être connecté à tous les pixels.
- **Partage de poids (weight sharing)** : le même filtre (noyau) glisse sur toute l'image, ce qui réduit drastiquement le nombre de paramètres et permet de détecter un même motif où qu'il apparaisse dans l'image.
- **Hiérarchie de représentations** : les couches successives apprennent des motifs de plus en plus abstraits (contours $\rightarrow$ parties $\rightarrow$ objets), et le pooling réduit la résolution spatiale tout en augmentant le champ récepteur, diminuant le coût de calcul — mais il ne préserve pas l'intégralité de l'information spatiale (c'est justement une perte contrôlée d'information au profit de l'invariance).

Exemple. Application n°1 — DE 2022/2023, Ex.2
Parmi les propositions suivantes, lesquelles sont des avantages des CNN par rapport aux réseaux de neurones classiques pour la classification d'images ?
(a) Les CNN préservent l'information spatiale grâce aux couches de convolution
(b) Les CNN apprennent automatiquement des représentations hiérarchiques des features
(c) Les CNN réduisent la dimensionnalité tout en préservant l'invariance spatiale

Solution.
(a) Vrai — la convolution conserve l'agencement 2D des pixels, contrairement à un aplatissement en vecteur.
(b) Vrai — c'est le principe même de l'empilement de couches convolutives.
(c) Vrai — le pooling réduit la taille des cartes de caractéristiques tout en conservant l'invariance aux petites translations.

Réponses correctes : (a), (b), (c)

Exemple. Application n°2 — DE 2024/2025, Ex.4 Q2
Parmi les propositions suivantes sur les CNN appliqués aux images, lesquelles sont correctes ?
(a) Les couches convolutives utilisent des filtres glissants (noyaux) pour capturer des motifs spatiaux hiérarchiques
(b) Le partage de poids réduit drastiquement le nombre de paramètres et permet de détecter le même motif n'importe où dans l'image
(c) Le zero-padding est souvent utilisé pour contrôler la taille spatiale des cartes de caractéristiques
(d) Les couches de pooling réduisent la résolution spatiale, diminuant le calcul et augmentant le champ récepteur
(e) La réduction de dimensionnalité par pooling ou convolution à stride préserve toujours toute l'information spatiale

Solution.
(a) Vrai.
(b) Vrai.
(c) Vrai — le padding « same » ajoute des zéros en bordure pour contrôler la taille de sortie.
(d) Vrai.
(e) Faux — c'est l'inverse : pooling et strided convolution **compressent** l'information spatiale (c'est justement leur rôle), ils ne la préservent pas intégralement.

Réponses correctes : (a), (b), (c), (d)

4.2 Calculer les dimensions de sortie et le nombre de paramètres d'un CNN, couche par couche

Méthode.
Pour un réseau CNN décrit par son code Keras, on calcule successivement, couche par couche :

1. Dimension spatiale de sortie d'une couche de convolution ou de pooling :

$$O_n = \left\lfloor \frac{N - F}{S} \right\rfloor + 1 \qquad \text{(padding } \texttt{valid}\text{)}$$

où N est la dimension d'entrée, F la taille du filtre (ou de la fenêtre de pooling), S le stride. En padding `same`, la dimension spatiale de sortie est égale à la dimension d'entrée (le padding compense exactement la perte due au filtre).

2. Nombre de paramètres d'une couche de convolution (avec biais) :

$$P_n = \big[(m \times n \times C_i) + 1\big] \times C_o$$

où $m \times n$ est la taille du filtre, C_i le nombre de canaux d'entrée et C_o le nombre de filtres (canaux de sortie).

3. Couches de pooling : ne contiennent aucun paramètre entraînable ($P_n = 0$) — elles ne font qu'un calcul (max ou moyenne).

4. Couche Flatten : aplati la sortie 3D (h, w, c) en un vecteur de taille $h \times w \times c$, sans paramètre.

5. Couche Dense de u unités recevant un vecteur de taille d :

$$P_n = (d + 1) \times u$$

On peut vérifier l'ensemble du calcul en sommant tous les P_n et en comparant au total de paramètres donné dans l'énoncé.

Exemple. Application n°1 — DE 2022/2023, Ex.3
On entraîne, sur CIFAR-10 (images $32 \times 32 \times 3$), le modèle Keras suivant : `Conv2D(16, (3,3), relu)` → `MaxPool2D((2,2), strides=2)` → `Conv2D(32, (3,3), padding='same', relu)` → `MaxPool2D((2,2), strides=2)` → `Flatten()` → `Dense(128, relu)` → `Dense(10, softmax)`. Sachant

que tous les neurones ont un biais et que le total de paramètres est $207\,210$, compléter les dimensions de sortie et le nombre de paramètres de chaque couche.

Solution.

Couche	Dimension de sortie	Paramètres
Conv2D (16, 3×3, valid)	$(30, 30, 16)$	$(3 \times 3 \times 3 + 1) \times 16 = 448$
MaxPool2D (2,2, stride 2)	$(15, 15, 16)$	0
Conv2D (32, 3×3, same)	$(15, 15, 32)$	$(3 \times 3 \times 16 + 1) \times 32 = 4640$
MaxPool2D (2,2, stride 2)	$(7, 7, 32)$	0
Flatten	$(1568,)$	0
Dense (128)	$(128,)$	$(1568 + 1) \times 128 = 200832$
Dense (10)	$(10,)$	$(128 + 1) \times 10 = 1290$

On vérifie : $448 + 4640 + 200832 + 1290 = 207\,210$ ✓
Total $= 207\,210$ paramètres, conforme à l'énoncé

Exemple. Application n°2 — DE 2024/2025, Ex.3
On entraîne, sur des images RGB $59 \times 59 \times 3$, le modèle Keras suivant : `Conv2D_1(16, k=5, s=2, valid, relu)` → `Pool1(2,2)` → `Conv2D_2(32, k=3, s=1, same, relu)` → `Pool2(2,2)` → `Conv2D_3(64, k=3, s=1, valid, relu)` → `Flatten` → `Dense_1(64, relu)` → `Dense_2(5, softmax)`. Sachant que tous les neurones ont un biais et que le total de paramètres est $127\,141$, compléter les dimensions de sortie et le nombre de paramètres de chaque couche.

Solution.

Couche	Dimension de sortie	Paramètres
InputLayer	$(59, 59, 3)$	0
Conv2D_1 ($k = 5, s = 2$, valid)	$\left\lfloor \frac{59-5}{2} \right\rfloor + 1 = (28, 28, 16)$	$(5 \times 5 \times 3 + 1) \times 16 = 1216$
Pool1 (2,2)	$\left\lfloor \frac{28-2}{2} \right\rfloor + 1 = (14, 14, 16)$	0
Conv2D_2 ($k = 3, s = 1$, same)	$(14, 14, 32)$	$(3 \times 3 \times 16 + 1) \times 32 = 4640$
Pool2 (2,2)	$\left\lfloor \frac{14-2}{2} \right\rfloor + 1 = (7, 7, 32)$	0
Conv2D_3 ($k = 3, s = 1$, valid)	$\left\lfloor \frac{7-3}{1} \right\rfloor + 1 = (5, 5, 64)$	$(3 \times 3 \times 32 + 1) \times 64 = 18496$
Flatten	$(1600,)$	0
Dense_1 (64)	$(64,)$	$(1600 + 1) \times 64 = 102464$
Dense_2 (5)	$(5,)$	$(64 + 1) \times 5 = 325$

On vérifie : $1216 + 4640 + 18496 + 102464 + 325 = 127\,141$ ✓
Total $= 127\,141$ paramètres, conforme à l'énoncé

Remarque. Piège fréquent
Quand le `strides` d'un `MaxPooling2D` n'est pas précisé explicitement dans le code, Keras utilise par défaut `strides = pool_size` (donc `strides=(2,2)` si `pool_size=(2,2)`) — c'est le cas de `Pool1` et `Pool2` dans l'Application n°2 ci-dessus.

4.3 Compléter une opération de convolution manuelle (calcul du feature map)

Méthode.
Pour calculer la valeur de sortie d'une convolution en une position donnée, avec un volume d'entrée multi-canaux (par exemple $C_i = 3$ canaux), un filtre de même profondeur, un stride S et un padding P déjà appliqué à l'entrée :

1. Repérer la position (i, j) à calculer dans la carte de sortie, et localiser la fenêtre correspondante dans l'entrée : elle démarre à la ligne $i \times S$ et à la colonne $j \times S$, sur les $F \times F$ lignes/colonnes suivantes, pour chacun des C_i canaux.
2. Pour chaque canal, multiplier terme à terme la fenêtre de l'entrée par le filtre correspondant à ce canal, et sommer tous les produits.
3. **Sommer les contributions des C_i canaux**, puis ajouter le biais b du filtre : c'est la valeur de sortie à la position (i, j).
4. Répéter pour chaque filtre (chaque filtre produit sa propre carte de sortie / son propre canal de sortie) et pour chaque position, en faisant glisser la fenêtre selon le stride S.

Exemple. Application — DE 2022/2023, Ex.4
On dispose d'un volume d'entrée $7 \times 7 \times 3$ (image $5 \times 5 \times 3$ avec un padding $P = 1$ déjà appliqué), de deux filtres W_0 et W_1 de taille $3 \times 3 \times 3$ (avec biais $b_0 = 1$ et $b_1 = 0$), et d'un stride $S = 2$:

$$x[:,:,0] = \begin{pmatrix} 0&0&0&0&0&0&0\\ 0&1&1&0&0&2&0\\ 0&0&2&1&0&2&0\\ 0&2&0&1&2&2&0\\ 0&2&0&2&1&2&0\\ 0&0&0&1&1&2&0\\ 0&0&0&0&0&0&0 \end{pmatrix} \quad x[:,:,1] = \begin{pmatrix} 0&0&0&0&0&0&0\\ 0&0&2&1&0&2&0\\ 0&0&2&1&2&2&0\\ 0&0&0&2&1&2&0\\ 0&1&0&0&0&2&0\\ 0&1&2&0&1&2&0\\ 0&0&0&0&0&0&0 \end{pmatrix} \quad x[:,:,2] = \begin{pmatrix} 0&0&0&0&0&0&0\\ 0&1&2&1&2&0&0\\ 0&1&0&2&2&1&0\\ 0&1&0&0&0&1&0\\ 0&1&0&0&1&1&0\\ 0&0&2&1&0&1&0\\ 0&0&0&0&0&0&0 \end{pmatrix}$$

$$W_0[:,:,0] = \begin{pmatrix} 0&1&1\\ 0&1&0\\ 0&1&0 \end{pmatrix} \quad W_0[:,:,1] = \begin{pmatrix} 1&0&-1\\ -1&0&0\\ 1&-1&1 \end{pmatrix} \quad W_0[:,:,2] = \begin{pmatrix} 0&0&-1\\ 0&-1&-1\\ 0&-1&-1 \end{pmatrix}$$

$$W_1[:,:,0] = \begin{pmatrix} 0&0&0\\ 1&0&1\\ -1&0&-1 \end{pmatrix} \quad W_1[:,:,1] = \begin{pmatrix} 0&0&1\\ 1&-1&-1\\ 1&-1&0 \end{pmatrix} \quad W_1[:,:,2] = \begin{pmatrix} 1&1&0\\ -1&0&0\\ 1&-1&1 \end{pmatrix}$$

On obtient les deux cartes de sortie 3×3 partiellement complétées :

$$o_1[:,:,0] = \begin{pmatrix} 0&-4&4\\ 2&?&4\\ 1&1&3 \end{pmatrix} \qquad o_1[:,:,1] = \begin{pmatrix} -4&-1&-3\\ 1&3&1\\ -2&0&? \end{pmatrix}$$

Calculer les deux valeurs manquantes.

Solution.
Valeur manquante de $o_1[:,:,0]$, position $(1,1)$ (indices à partir de 0). Avec $S = 2$, cette position correspond à la fenêtre $x[2{:}5,\ 2{:}5,\ :]$:

$$\text{fenêtre}_0 = \begin{pmatrix} 2&1&0\\ 0&1&2\\ 0&2&1 \end{pmatrix} \quad \text{fenêtre}_1 = \begin{pmatrix} 2&1&2\\ 0&2&1\\ 0&0&0 \end{pmatrix} \quad \text{fenêtre}_2 = \begin{pmatrix} 0&2&2\\ 0&0&0\\ 0&0&1 \end{pmatrix}$$

On multiplie terme à terme chaque fenêtre par le filtre W_0 du canal correspondant, puis on somme les trois contributions et le biais $b_0 = 1$:

$$\underbrace{(0{+}1{+}0)}_{\text{canal } 0} + \underbrace{(1{+}0{+}1{-}0)}_{\text{canal } 0\text{, suite}} \ldots$$

Plus précisément, canal par canal :

canal 0 : $(2{\times}0) + (1{\times}1) + (0{\times}1) + (0{\times}0) + (1{\times}1) + (2{\times}0) + (0{\times}0) + (2{\times}1) + (1{\times}0) = 1 + 1 + 2 = 4$

canal 1 : $(2{\times}1) + (1{\times}0) + (2{\times}{-}1) + (0{\times}{-}1) + (2{\times}0) + (1{\times}0) + (0{\times}1) + (0{\times}{-}1) + (0{\times}1) = 2 - 2 = 0$

canal 2 : $(0\times0)+(2\times0)+(2\times-1)+(0\times0)+(0\times-1)+(0\times-1)+(0\times0)+(0\times-1)+(1\times-1) = -2-1 = -3$

$$o_1[1,1,0] = \underbrace{4}_{\text{canal } 0} + \underbrace{0}_{\text{canal } 1} + \underbrace{(-3)}_{\text{canal } 2} + \underbrace{1}_{b_0} = 2$$

$\boxed{o_1[1,1,0] = 2}$

Valeur manquante de $o_1[:,:,1]$, position $(2,2)$. Fenêtre $x[4:7, 4:7, :]$, filtre W_1, biais $b_1 = 0$: par la même méthode (fenêtre $\times$ filtre, canal par canal, puis somme des trois canaux et du biais), on obtient

$$\text{canal } 0 = 1 \qquad \text{canal } 1 = -1 \qquad \text{canal } 2 = 2$$

$$o_1[2,2,1] = 1 + (-1) + 2 + 0 = 2$$

$\boxed{o_1[2,2,1] = 2}$

Remarque. Un filtre par canal de sortie
Chaque filtre (W_0, W_1, ...) produit un seul canal de la sortie, quel que soit le nombre de canaux de l'entrée : on somme toujours les contributions de tous les canaux d'entrée en une seule valeur par position. C'est pourquoi 2 filtres $3\times3\times3$ donnent ici une sortie à 2 canaux ($3\times3\times2$), et non $3\times2=6$.

5 Modélisation générative profonde (Autoencodeurs, VAE, GAN)

5.1 Identifier les caractéristiques de la modélisation générative profonde

Méthode.
Contrairement aux modèles discriminatifs (qui apprennent $p(y \mid x)$), les modèles génératifs cherchent à capturer la distribution des données elles-mêmes, $p(x)$, afin de pouvoir :

- **Estimer une densité** : approcher la distribution de probabilité qui a généré les données observées.
- **Générer de nouveaux échantillons** : produire des données inédites mais réalistes, similaires à celles du jeu d'entraînement.
- **Découvrir la structure latente** : mettre au jour des facteurs explicatifs sous-jacents aux données (représentations compressées).
- **Apprentissage non supervisé** : ces modèles n'ont en général pas besoin de labels — ils exploitent directement la structure des données.

Ils ne sont en revanche pas conçus par défaut pour modéliser des dépendances temporelles (cela nécessite des architectures spécialisées comme les RNN/LSTM ou les Transformers, voir Chapitre 6), et ne sont pas entraînés de façon strictement supervisée avec labels explicites.

Exemple. Application n°1 — DE 2022/2023, Ex.6
Parmi les propositions suivantes, lesquelles sont des caractéristiques de la modélisation générative profonde ?
(a) Apprentissage non supervisé
(b) Estimation de densité
(c) Génération d'échantillons
(d) Mise au jour de features sous-jacentes dans un jeu de données
(e) Capture de dépendances temporelles

Solution.
(a) Vrai.
(b) Vrai.
(c) Vrai.
(d) Vrai.
(e) Faux — la capture de dépendances temporelles nécessite une architecture séquentielle dédiée (RNN, LSTM, attention), pas la modélisation générative en tant que telle.

Réponses correctes : (a), (b), (c), (d)

Exemple. Application n°2 — DE 2024/2025, Ex.4 Q3
Concernant les modèles génératifs profonds, lesquelles des propositions suivantes sont correctes ?
(a) Ils réalisent une estimation de densité pour apprendre la distribution des données
(b) Ils peuvent échantillonner de nouvelles données réalistes, similaires à celles du jeu d'entraînement
(c) Ils exploitent des variables latentes pour découvrir la structure sous-jacente des données
(d) Ils modélisent efficacement des données séquentielles (séries temporelles) sans architecture spécialisée
(e) Ils sont entraînés selon un paradigme strictement supervisé, avec labels explicites

Solution.
(a) Vrai.
(b) Vrai.
(c) Vrai.
(d) Faux — sans mécanisme séquentiel dédié (RNN/LSTM/attention), un modèle génératif « générique » ne capture pas nativement l'ordre temporel.
(e) Faux — l'apprentissage est typiquement non supervisé (pas de labels requis).

Réponses correctes : (a), (b), (c)

5.2 Compléter une architecture d'autoencodeur convolutif (dimensions manquantes)

Méthode.
Un autoencodeur convolutif est composé de deux parties symétriques :

1. Encodeur : une succession de couches `Conv2D` et `MaxPooling2D` qui réduit progressivement la résolution spatiale tout en augmentant (ou conservant) la profondeur, jusqu'à obtenir la représentation latente (« bottleneck »).

2. Décodeur : symétrique de l'encodeur, il alterne `Conv2D` et `UpSampling2D` pour restaurer progressivement la résolution spatiale d'origine, jusqu'à obtenir une sortie de même forme que l'entrée.

Règle pratique pour retrouver les dimensions manquantes :
- Chaque `MaxPooling2D((2,2))` divise la résolution spatiale par 2 (le nombre de canaux ne change pas, il est fixé par le `Conv2D` qui précède).
- Chaque `UpSampling2D((2,2))` multiplie la résolution spatiale par 2.
- La dernière couche du décodeur doit retrouver exactement les dimensions spatiales et le nombre de canaux de l'image d'entrée (ex. $(28, 28, 1)$ pour du MNIST), avec une activation `sigmoid` si les pixels sont normalisés dans $[0, 1]$.

Exemple. Application — DE 2022/2023, Ex.8
On entraîne l'autoencodeur suivant sur MNIST $(28 \times 28 \times 1)$:

```
input_img = Input(shape=(28, 28, 1))
x = Conv2D(16, (3,3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2,2), padding='same')(x)
x = Conv2D(8, (3,3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2,2), padding='same')(x)
# à ce stade, la représentation est (7, 7, 8)
x = Conv2D(8, (3,3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2,2))(x)
x = Conv2D(16, (3,3), activation='relu', padding='same')(x)
x = UpSampling2D((2,2))(x)
decoded = Conv2D(1, (3,3), activation='sigmoid', padding='same')(x)
```

Justifier la cohérence des dimensions à chaque étape, en particulier celles du décodeur.

Solution.
Encodeur (padding `same`, donc seule la résolution change avec le pooling) :
$(28, 28, 1) \xrightarrow{\text{Conv2D 16}} (28, 28, 16) \xrightarrow{\text{Pool}} (14, 14, 16) \xrightarrow{\text{Conv2D 8}} (14, 14, 8) \xrightarrow{\text{Pool}} (7, 7, 8)$
On retrouve bien la représentation latente donnée dans l'énoncé : $(7, 7, 8)$.

Décodeur (symétrique, avec `UpSampling2D` qui double la résolution à chaque étape) :
$(7, 7, 8) \xrightarrow{\text{Conv2D 8}} (7, 7, 8) \xrightarrow{\text{UpSampling}} (14, 14, 8) \xrightarrow{\text{Conv2D 16}} (14, 14, 16) \xrightarrow{\text{UpSampling}} (28, 28, 16) \xrightarrow{\text{Conv2D 1, sigmoid}} (28, 28, 1)$
On retrouve bien la forme de l'image d'entrée $(28, 28, 1)$: le nombre de filtres de la dernière couche doit être égal au nombre de canaux de l'image d'origine (ici 1, image en niveaux de gris), et l'activation `sigmoid` produit des valeurs dans $[0, 1]$ comparables aux pixels normalisés.

5 valeurs à restituer : $16,\ 8,\ 8,\ 16,\ 1$ (nombres de filtres successifs)

5.3 Compléter la fonction de perte GAN (objectif minimax)

Méthode.
Un GAN oppose deux réseaux :

- Le **générateur** G transforme un bruit $z \sim \mathcal{N}(0, 1)$ en donnée synthétique $G(z)$, en essayant de tromper le discriminateur.
- Le **discriminateur** D reçoit une donnée (réelle x ou générée $G(z)$) et estime la probabilité qu'elle soit réelle.

Les deux réseaux s'entraînent de façon adverse selon l'objectif minimax :

$$\arg\min_G \max_D \ \mathbb{E}_x\big[\log D(x)\big] + \mathbb{E}_z\big[\log\big(1 - D(G(z))\big)\big]$$

- D cherche à maximiser cet objectif : bien classer le réel ($D(x) \to 1$) et le faux ($D(G(z)) \to 0$).
- G cherche à minimiser cet objectif : tromper D en le poussant à croire que $G(z)$ est réel ($D(G(z)) \to 1$).

À l'optimum global, G reproduit exactement la distribution des données réelles.

Exemple. Application — DE 2022/2023, Ex.7
Compléter les opérateurs « min » et « max » manquants dans l'expression suivante de la perte GAN :

$$____G \ ____D \Big[\mathbb{E}_z \log D(G(z)) + \mathbb{E}_x \log\big(1 - D(x)\big)\Big]$$

Solution.
En reprenant la méthode ci-dessus : le générateur G minimise l'objectif, le discriminateur D le maximise.

$$\boxed{\min_G \max_D \Big[\mathbb{E}_z \log D(G(z)) + \mathbb{E}_x \log\big(1 - D(x)\big)\Big]}$$

Remarque. Attention à la lecture des termes
Selon la convention utilisée dans l'énoncé (rôles de x et z échangés par rapport à l'écriture « standard » du cours), on retrouve toujours la même logique : le terme en $D(\cdot)$ appliqué à une donnée **réelle** doit être maximisé par D, et le terme en $D(G(\cdot))$ (donnée **générée**) doit pousser G à tromper D. Ne pas se fier uniquement à la position de x et z dans la formule : identifier d'abord qui est réel et qui est généré.

5.4 Expliquer la fonction de perte VAE (ELBO, divergence KL) et contraster VAE / autoencodeur standard

Méthode.
Un autoencodeur variationnel (VAE) remplace l'encodage déterministe $x \to z$ d'un autoencodeur classique par un encodage **probabiliste** : l'encodeur produit une distribution $q_\phi(z \mid x)$ (typiquement une gaussienne de moyenne μ et variance σ^2), dont on échantillonne z.

1. Rôle des deux termes de la perte :

$$\mathcal{L}(\phi, \theta, x) = \underbrace{\mathbb{E}_{q_\phi(z|x)}\big[\log p_\theta(x \mid z)\big]}_{\text{reconstruction}} - \underbrace{D_{KL}\big(q_\phi(z \mid x) \,\|\, p(z)\big)}_{\text{régularisation}}$$

- $q_\phi(z \mid x)$: distribution approchée du code latent *sachant* l'entrée x (sortie de l'encodeur).
- $p(z)$: la distribution a priori choisie pour l'espace latent, en général $\mathcal{N}(0, I)$.

2. Rôle de la divergence KL : $D_{KL}\big(q_\phi(z \mid x) \| p(z)\big)$ mesure l'écart entre la distribution latente inférée et le prior imposé. La minimiser force les codes latents à rester proches d'une gaussienne centrée réduite.

3. Propriétés recherchées sur l'espace latent :
- **Continuité** : deux points proches dans l'espace latent doivent se décoder en contenus similaires.
- **Complétude** : tout point échantillonné dans l'espace latent doit se décoder en un contenu « plausible » (pas de trous vides de sens).

4. Contraste avec un autoencodeur standard : l'autoencodeur classique apprend un mapping déterministe $x \to z \to \hat{x}$ (pas de notion de distribution), il est simple et stable à entraîner mais n'est pas génératif : on ne peut pas échantillonner un z « au hasard » et obtenir un contenu cohérent, car l'espace latent n'est pas régularisé. Le VAE, lui, rend l'espace latent structuré et échantillonnable, au prix d'une perte composite plus délicate à équilibrer (poids relatif reconstruction / KL) — et peut être étendu à l'apprentissage semi-supervisé ou au clustering via sa représentation latente.

Exemple. Application n°1 — DE 2024/2025, Ex.4
1. Dans le terme de régularisation de la perte VAE, que représentent $q_\phi(z \mid x)$ et $p(z)$?
2. Que mesure la divergence KL $D_{KL}\big(q_\phi(z \mid x) \| p(z)\big)$ dans le contexte d'un VAE ?
3. Citer deux propriétés ou comportements souhaitables que la minimisation de ce terme impose à la distribution latente apprise.

Solution.
1. $q_\phi(z \mid x)$ est la distribution latente **approchée** produite par l'encodeur pour une entrée x donnée ; $p(z)$ est la distribution **a priori** (prior) fixée sur l'espace latent, en général $\mathcal{N}(0, I)$.

2. Elle mesure à quel point la distribution latente inférée $q_\phi(z \mid x)$ s'écarte du prior $p(z)$: c'est une pénalité qui empêche l'encodeur de « tricher » en dispersant arbitrairement les codes latents loin du prior.

3. Deux propriétés imposées :
- **Continuité** de l'espace latent (des points proches se décodent en contenus similaires) ;
- **Complétude** de l'espace latent (l'échantillonnage depuis $p(z)$ produit du contenu cohérent, sans « trous » non décodables).

Minimiser $D_{KL} \Rightarrow$ espace latent continu et complet, exploitable pour la génération

Exemple. Application n°2 — DE 2024/2025, Ex.4 Q4
Comparer VAE et autoencodeur standard : lesquelles des propositions suivantes sont correctes ?
(a) Les deux sont également efficaces pour générer des images photo-réalistes en haute résolution
(b) L'autoencodeur standard est généralement plus difficile à entraîner, avec une perte composite à régler finement
(c) Le VAE fournit une formulation probabiliste pour encoder les observations dans l'espace latent, alors que l'autoencodeur standard utilise un mapping déterministe
(d) Le VAE excelle dans la modélisation de distributions de données, permettant de générer de nouveaux échantillons similaires
(e) L'autoencodeur standard est surtout utilisé pour la réduction de dimensionnalité, tandis que le VAE est mieux adapté à la génération
(f) Le VAE peut être étendu à l'apprentissage semi-supervisé et au clustering via ses représentations latentes

Solution.
(a) Faux — ni l'un ni l'autre ne rivalise nativement avec un GAN ou un modèle de diffusion en photo-réalisme ; le VAE produit typiquement des images plus floues.
(b) Faux — c'est l'inverse : c'est le **VAE** qui a une perte composite (reconstruction + KL) à équilibrer ; l'autoencodeur standard n'a qu'un terme de reconstruction, plus simple à entraîner.
(c) Vrai.
(d) Vrai.
(e) Vrai.
(f) Vrai — la structure probabiliste du latent VAE s'y prête naturellement.

Réponses correctes : (c), (d), (e), (f)

6 Réseaux de neurones récurrents, attention et Transformers

6.1 Compléter une architecture LSTM en Keras (cohérence des dimensions)

Méthode.
Pour retrouver des valeurs manquantes dans une architecture séquentielle Keras (Embedding → LSTM → Dense), on s'appuie sur des contraintes de **cohérence dimensionnelle** plutôt que sur un calcul direct :

1. **Taille du vocabulaire** : le nombre d'entrées de la couche `Embedding` (`input_dim`) doit correspondre au nombre de symboles distincts que le modèle peut lire et produire. Si la couche de sortie `Dense` prédit un symbole parmi V possibles, elle doit avoir exactement V unités — et c'est en général la même valeur V que celle utilisée dans l'`Embedding`.

2. **Cohérence de la taille de batch avec un RNN** `stateful` : lorsque `stateful=True`, Keras conserve l'état caché d'un batch au batch suivant ; il faut donc fixer une taille de batch explicite et constante, via `batch_input_shape=[batch_size, ...]` en entrée du modèle — cette taille de batch doit être la même à chaque appel de `fit`/`predict`.

3. **Dimension de la représentation cachée** : le nombre d'unités de la couche `LSTM` (`units`) fixe la taille du vecteur d'état caché h_t, indépendante de la taille du vocabulaire ou du batch.

Exemple. Application — DE 2022/2023, Ex.5
On étudie le code Keras suivant, décrivant un LSTM appliqué à une base de données musicale (prédiction de la note suivante parmi un vocabulaire de notes) :

```
lstm_model = tf.keras.Sequential([
  tf.keras.layers.Embedding(83, 256, batch_input_shape=[32, None]),
  tf.keras.layers.LSTM(units=100, return_sequences=True,
            recurrent_activation='sigmoid', stateful=True),
  tf.keras.layers.Dense(83)
])
```

Deux valeurs sont nécessaires au bon fonctionnement du réseau : lesquelles, et pourquoi ?

Solution.
On applique la méthode ci-dessus :

1. Taille du vocabulaire (83). Le modèle prédit, à chaque pas de temps, une distribution sur les notes possibles : la couche `Dense` finale doit donc avoir autant d'unités que de notes distinctes dans le vocabulaire. Cette même valeur (83) doit être utilisée comme `input_dim` de l'`Embedding`, puisque c'est le nombre de symboles distincts que le modèle peut recevoir en entrée. Les deux valeurs « 83 » (Embedding et Dense) ne sont donc pas indépendantes : elles désignent toutes deux la taille du même vocabulaire.

2. Taille de batch fixe (32). Comme `stateful=True`, Keras doit connaître à l'avance une taille de batch fixe pour pouvoir conserver l'état caché h_t d'un batch au batch suivant — d'où le `batch_input_shape=[32, None]` (32 séquences par batch, longueur de séquence libre).

Cohérence à respecter : Embedding(83,...) ↔ Dense(83) ; batch_size fixe (32) car stateful=True

Remarque. Nombre de paramètres d'un LSTM
Un LSTM possède 4 portes (oubli, entrée, cellule candidate, sortie), chacune avec sa propre matrice de poids de taille $\text{shape}(h) \times [\text{shape}(h) + \text{shape}(x)]$. Le nombre de paramètres (hors biais) est donc $4 \times \text{shape}(h) \times [\text{shape}(h) + \text{shape}(x)]$ — soit 4 fois plus qu'un `SimpleRNN` de même taille d'état caché.

6.2 Expliquer pourquoi un SimpleRNN peine sur les dépendances longues et comment le LSTM y remédie

Méthode.

1. Le problème du SimpleRNN : l'état caché est mis à jour par $h_t = \tanh(W \cdot [h_{t-1}, x_t])$. Pour évaluer l'influence d'une entrée lointaine x_1 sur un état tardif h_T (avec $T \gg 1$), on doit calculer $\frac{\partial h_T}{\partial x_1}$, qui s'obtient par un produit en chaîne de $T-1$ facteurs (dérivées de $\tanh$ et de W) :

$$\frac{\partial h_T}{\partial x_1} = \prod_{t=2}^{T} \frac{\partial h_t}{\partial h_{t-1}}$$

Comme chaque facteur $\frac{\partial h_t}{\partial h_{t-1}}$ a une norme typiquement < 1 (la dérivée de $\tanh$ est ≤ 1), ce produit tend **exponentiellement vers 0** quand T grandit : c'est le **problème du gradient qui s'évanouit** (*vanishing gradient*). Le réseau devient alors incapable d'apprendre des dépendances entre éléments éloignés dans la séquence.

2. La solution du LSTM : le LSTM introduit une **cellule mémoire** C_t (le « tapis roulant », *conveyor belt*) qui traverse le temps par une liaison quasi directe :

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$

où $f_t = \sigma(W_f \cdot [h_{t-1}, x_t])$ est la **porte d'oubli** (décide ce qu'on efface de C_{t-1}), $i_t = \sigma(W_i \cdot [h_{t-1}, x_t])$ la **porte d'entrée** et $\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t])$ la valeur candidate. Comme la mise à jour de C_t est **additive** (et non plus une composition purement multiplicative comme pour h_t), le gradient peut circuler sur de longues distances sans s'annuler systématiquement, à condition que la porte d'oubli f_t reste proche de 1 quand l'information doit être conservée.

Exemple. Application — DE 2024/2025, Ex.5 Q1

Expliquer mathématiquement pourquoi un SimpleRNN peine à capturer les dépendances à long terme, et expliquer quel mécanisme le LSTM utilise pour surmonter ce problème.

Solution.

Le problème. Dans un SimpleRNN, $\frac{\partial h_T}{\partial x_1}$ s'écrit comme un produit de $T-1$ jacobiennes $\frac{\partial h_t}{\partial h_{t-1}}$, chacune impliquant la dérivée de $\tanh$ (bornée par 1) multipliée par la matrice de poids récurrente W. Si les valeurs propres de W (combinées à la saturation de $\tanh$) sont < 1 en norme, ce produit décroît exponentiellement avec T : le gradient **s'évanouit**, et l'état h_{100} devient presque totalement déconnecté de x_1 — le réseau ne peut plus apprendre cette dépendance.

La solution du LSTM. Le LSTM ajoute une cellule mémoire C_t mise à jour de façon additive : $C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$. Cette liaison directe entre C_{t-1} et C_t (sans passer par une composition non linéaire à chaque pas) permet au gradient de se propager sur de longues distances sans s'annuler, tant que la porte d'oubli f_t reste proche de 1 — le réseau apprend alors *quand* garder ou effacer l'information, plutôt que de la perdre systématiquement.

SimpleRNN : produit de jacobiennes → gradient qui s'évanouit / LSTM : mise à jour additive de C_t → gradient préservé

6.3 Expliquer la limite du LSTM qui motive le mécanisme d'attention

Méthode.
Même avec ses portes, un LSTM (ou tout RNN encodeur-décodeur classique) doit compresser **toute** l'information utile d'une séquence, aussi longue soit-elle, dans un unique vecteur d'état caché de taille fixe (le dernier h_T transmis au décodeur). C'est un **goulot d'étranglement (bottleneck)** : plus la séquence est longue, plus il est difficile de tout retenir dans ce vecteur unique, et les informations les plus anciennes tendent à être écrasées par les plus récentes malgré les portes.

Le **mécanisme d'attention** lève ce goulot d'étranglement : au lieu de ne transmettre que le dernier état caché, on autorise le décodeur à « regarder » directement, à chaque pas de temps, l'ensemble des états cachés passés $h_1, \ldots, h_T$, en leur affectant des poids $\alpha_1, \ldots, \alpha_T$ (calculés dynamiquement selon leur pertinence pour l'étape courante) afin de former un vecteur de contexte $c = \sum_i \alpha_i h_i$. Aucune information n'a donc plus besoin d'être compressée dans un vecteur unique.

Exemple. Application — DE 2024/2025, Ex.5 Q2
Expliquer la limite restante des modèles séquentiels à base de LSTM qui motive l'usage de l'attention, et décrire comment le mécanisme d'attention y répond.

Solution.
La limite. Un LSTM (même dans une architecture encodeur-décodeur) doit résumer toute une séquence d'entrée dans un vecteur d'état caché de taille fixe, transmis au décodeur. Ce goulot d'étranglement dégrade la qualité de la modélisation à mesure que la séquence s'allonge : le modèle « oublie » progressivement les éléments les plus anciens, faute de pouvoir tout compresser fidèlement dans un vecteur de dimension constante.

La réponse de l'attention. Le mécanisme d'attention permet au décodeur d'accéder directement à tous les états cachés de l'encodeur $h_1, \ldots, h_T$ à chaque pas de décodage, via un vecteur de contexte $c_j = \sum_i \alpha_{ij} h_i$ où les poids α_{ij} mesurent la pertinence de chaque état passé h_i pour l'étape courante j. Plus aucune information n'est donc forcée de transiter par un unique vecteur de taille fixe : le modèle peut « revenir consulter » directement n'importe quelle portion de la séquence d'entrée.

Goulot d'étranglement du vecteur unique → accès direct pondéré à tous les états passés

6.4 Identifier les propriétés du mécanisme d'auto-attention (self-attention)

Méthode.
Le mécanisme d'auto-attention permet à chaque élément (token) d'une séquence d'interagir directement avec tous les autres éléments de la même séquence, sans passer par une récurrence pas à pas.

1. Projections Query / Key / Value : chaque token produit trois vecteurs par des projections linéaires apprises :

$$q_{:j} = W_Q\, x_j \qquad k_{:i} = W_K\, x_i \qquad v_{:i} = W_V\, x_i$$

- *Query* : « que cherche ce token ? »
- *Key* : « qu'est-ce que ce token propose, en résumé ? »
- *Value* : « que donne effectivement ce token si on le sélectionne ? »

2. Poids d'attention : on mesure la compatibilité entre chaque requête et chaque clé, normalisée par $\sqrt{d_k}$ (pour éviter que les produits scalaires ne saturent le softmax en grande dimension) :

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V$$

Le softmax est appliqué le long de l'axe des *clés* : pour une requête (un token) donnée, les poids d'attention sur tous les tokens auxquels elle peut prêter attention somment à 1.

3. Portée : chaque token peut ainsi interagir directement avec n'importe quel autre token de la séquence, quelle que soit leur distance — contrairement à un RNN où l'information doit transiter pas à pas. C'est ce qui permet à l'auto-attention de capturer efficacement des dépendances à longue portée.

Exemple. Application — DE 2024/2025, Q5
Parmi les propositions suivantes sur le mécanisme d'auto-attention, lesquelles sont correctes ?
(a) L'auto-attention ne considère que les tokens immédiatement voisins dans la séquence
(b) L'auto-attention permet à chaque token de la séquence d'interagir avec tous les autres tokens
(c) Dans l'auto-attention, chaque token est projeté en vecteurs query, key et value
(d) L'auto-attention capture efficacement les dépendances à longue portée dans une séquence

Solution.
(a) Faux — c'est précisément l'inverse : l'auto-attention regarde toute la séquence, pas seulement les voisins immédiats (c'est un comportement plutôt associé à la convolution).
(b) Vrai.
(c) Vrai.
(d) Vrai.

Réponses correctes : (b), (c), (d)

6.5 Identifier les avantages du multi-head attention

Méthode.
Une seule tête d'attention ne peut apprendre qu'une seule notion de pertinence (par exemple, uniquement la syntaxe, ou uniquement la proximité thématique). Le **multi-head attention** exécute plusieurs opérations d'attention en parallèle, chacune avec ses propres projections W_Q, W_K, W_V dans un sous-espace de dimension réduite, puis concatène les résultats :

- Chaque tête apprend à se spécialiser dans un type de relation différent entre les tokens (une tête peut capter la syntaxe, une autre la référence pronominale, etc.), augmentant la capacité du modèle à représenter des relations variées et complexes.
- Les têtes sont calculées en parallèle, ce qui permet un calcul efficace sur GPU (pas de dépendance séquentielle entre les têtes).
- Avec h têtes de dimension $d_k = d_{\text{model}}/h$ chacune, la capacité totale (nombre de paramètres et dimension effective traitée) reste comparable à une seule tête de dimension d_{model} — le multi-head **ne réduit pas** significativement le nombre de paramètres, il répartit la capacité entre plusieurs sous-espaces de représentation plutôt que de la multiplier ou de la diminuer drastiquement.

Exemple. Application — DE 2024/2025, Q6
Quels avantages le multi-head attention apporte-t-il par rapport à une attention à une seule tête ?
(a) Il permet au modèle de porter son attention sur plusieurs sous-espaces de représentation simultanément
(b) Il augmente la capacité du modèle à capturer des features et relations variées
(c) Il réduit significativement le nombre total de paramètres du mécanisme d'attention
(d) Il permet un calcul parallèle des têtes d'attention, pour un apprentissage plus robuste

Solution.
(a) Vrai.
(b) Vrai.
(c) Faux — avec h têtes de dimension d_{model}/h chacune, la capacité totale reste comparable à une seule tête de dimension d_{model} ; le multi-head ne réduit pas notablement le nombre de paramètres, il diversifie les sous-espaces appris.
(d) Vrai.

Réponses correctes : (a), (b), (d)

Remarque. Repère architectural — le Transformer
Le Transformer (Vaswani et al., 2017) est un modèle séquence-à-séquence qui abandonne totalement la récurrence : il empile uniquement des blocs d'auto-attention (multi-head) et de couches denses, avec un encodage positionnel pour réinjecter l'information d'ordre (perdue par l'auto-attention, qui traite les tokens de façon permutation-invariante). C'est ce qui lui permet de paralléliser entièrement l'entraînement (contraire au RNN, intrinsèquement séquentiel) et d'atteindre de meilleures performances sur de grands jeux de données textuelles, à condition de disposer d'assez de données pour compenser l'absence de biais inductif (contrairement au RNN qui suppose « l'ordre compte », ou au CNN qui suppose « les pixels voisins comptent »).