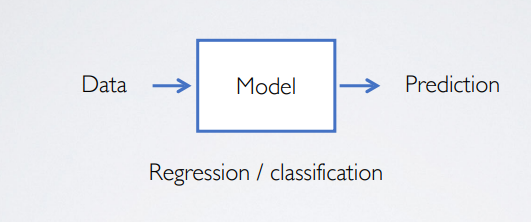
**Intelligence artificielle :**

**Compréhension et prise de notes**

**Le principe :**



**Type de données :**

Les variables mesurées dans une expérimentation peuvent être:

**• Qualitatives ou Catégorielles** (expriment une qualité comme le sexe, le métier ou le nom)

**• Quantitatives:**

• Discrètes (Ex: âge, nombre d'habitants)

• Continues (Ex: T°)

Les variables peuvent être aussi :

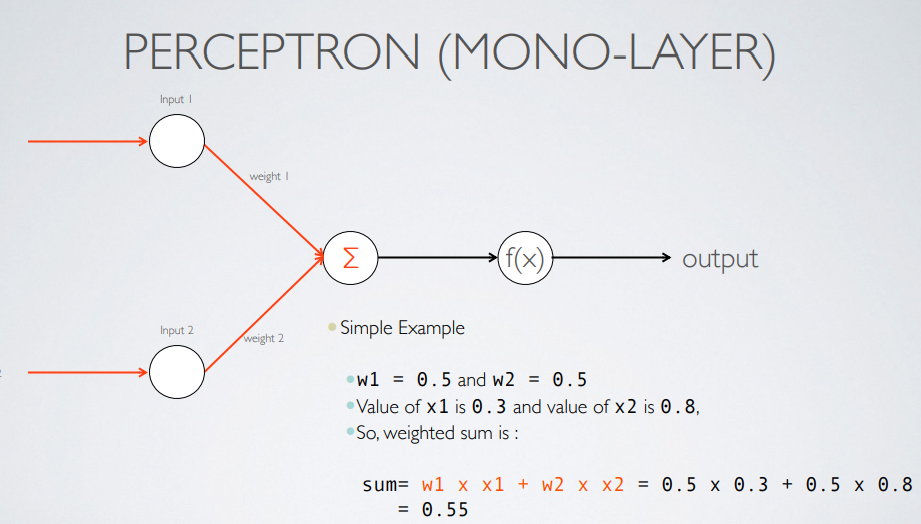
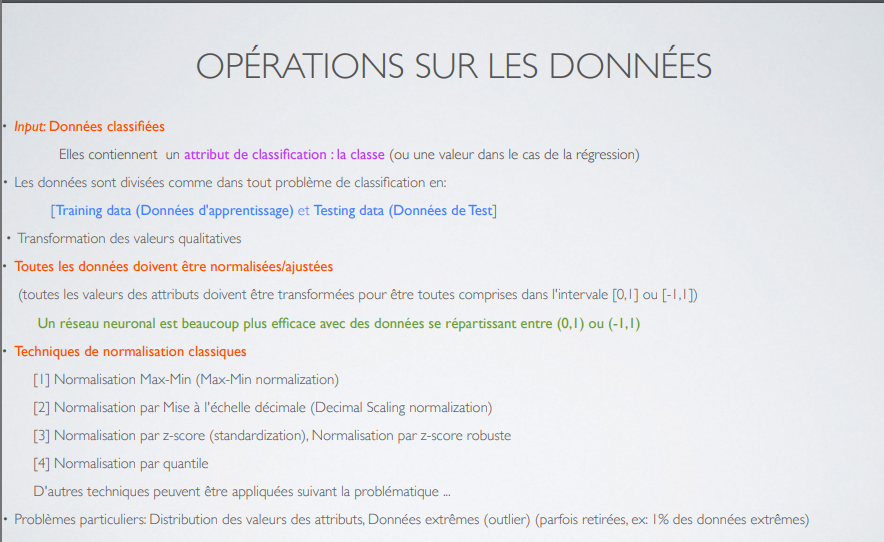
• Nominales (ex: homme ou femme, le nom de la voiture, une couleur, nom de journal)

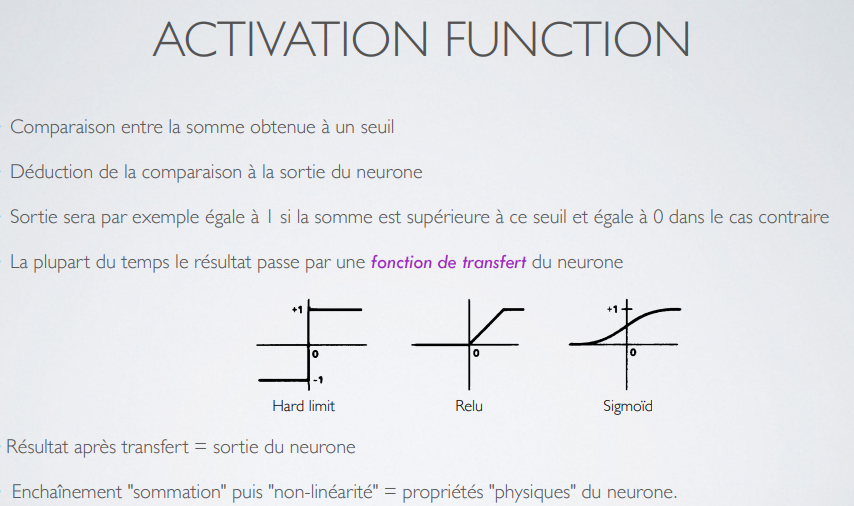
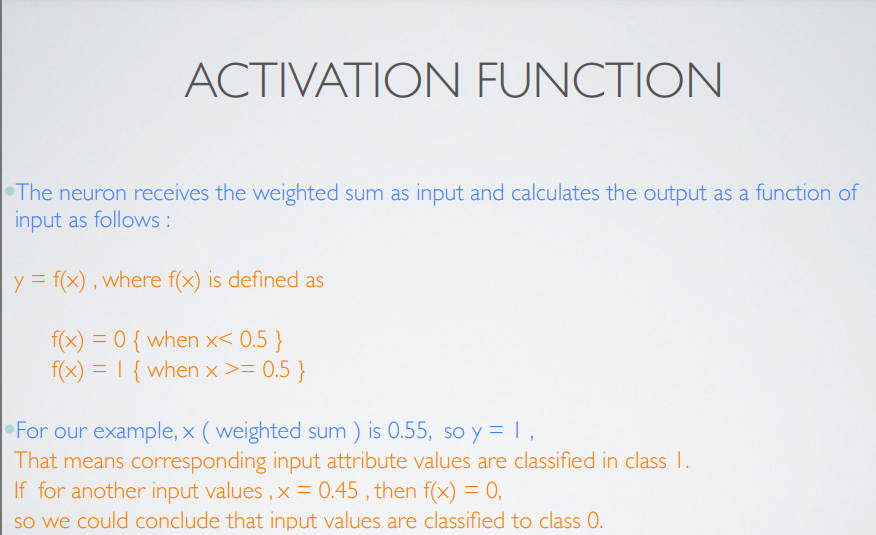
• Ordinales (catégories ordonnées, ex: échelle de Likert (Pas du tout, un peu, moyen, beaucoup, énormément, à la folie))

• D’intervalles (similaires aux variables ordinales sauf que l’intervalle entre deux catégories à toujours la même valeur. Ex: T°, date)

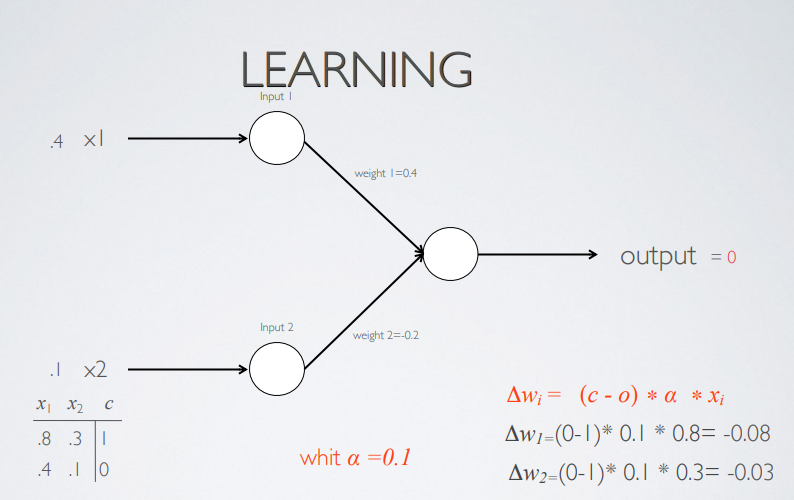
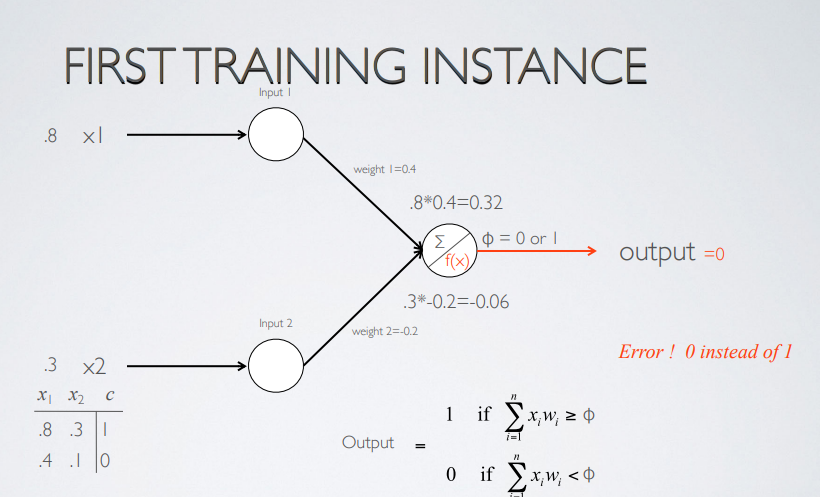
• De ratios (ou de rapport) (variables d’intervalles avec un zéro naturel. Ex: durée, distance, valeur)

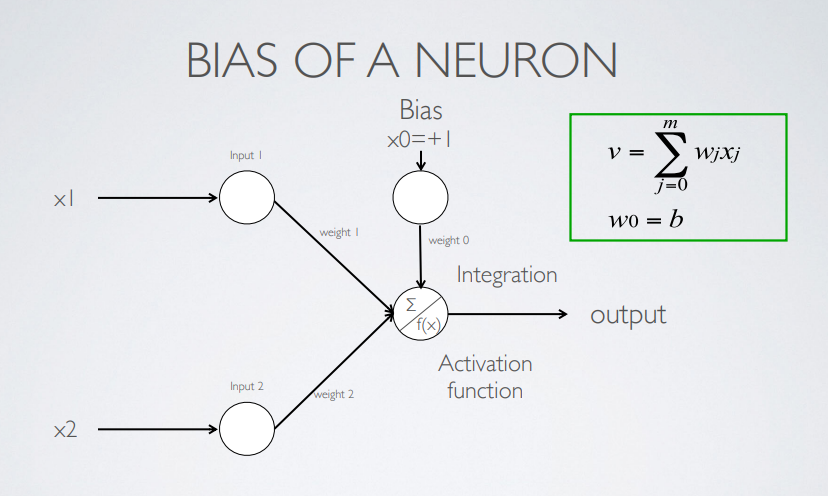
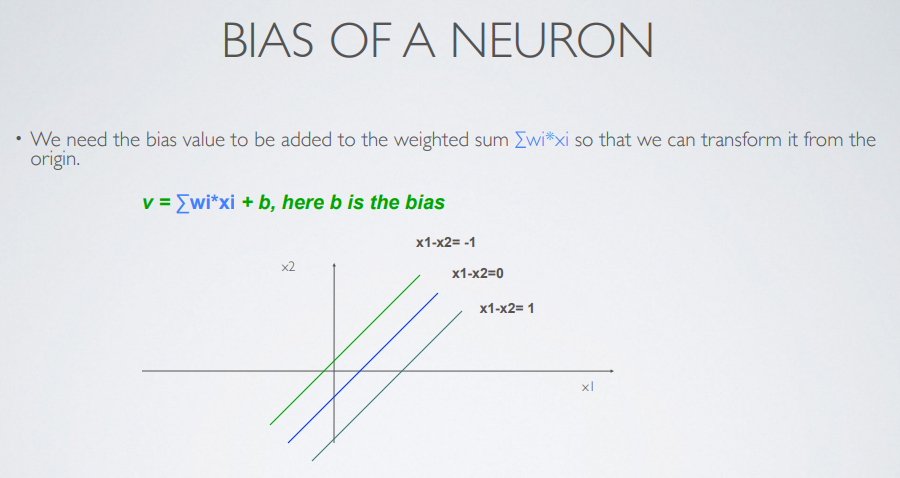
**IL FAUT FAIRE UN TRAVAIL D’ANALYSE DE DONNÉES AVANT DE COMMENCER, COMPRENDRE LES DONNÉES C’EST ESSENTIEL**





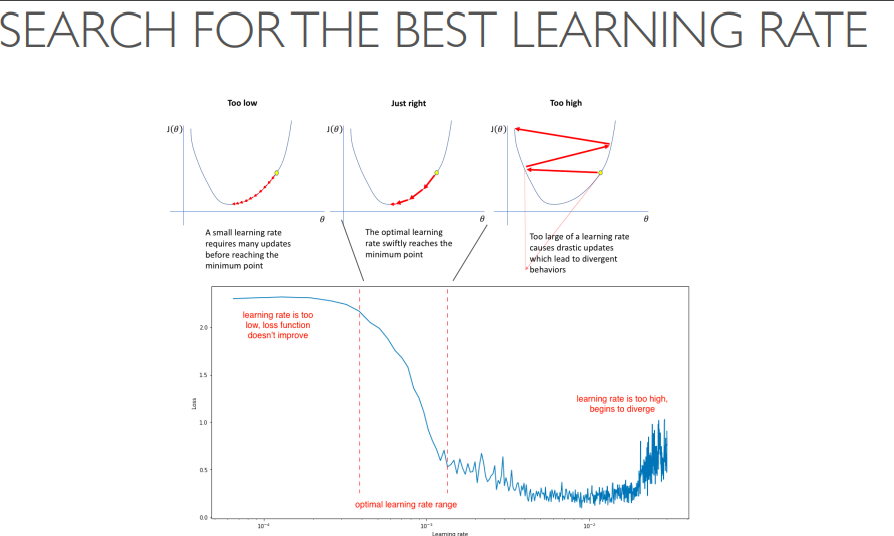
**Perceptron :**



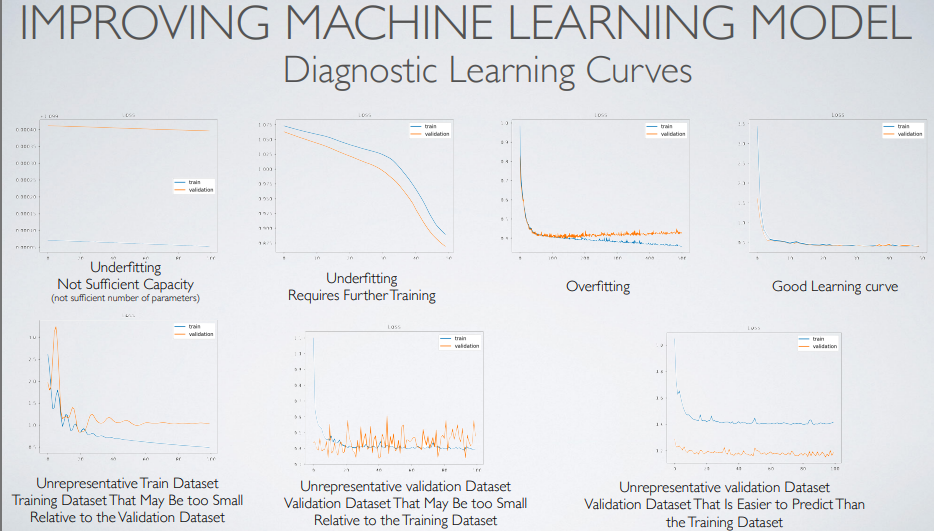


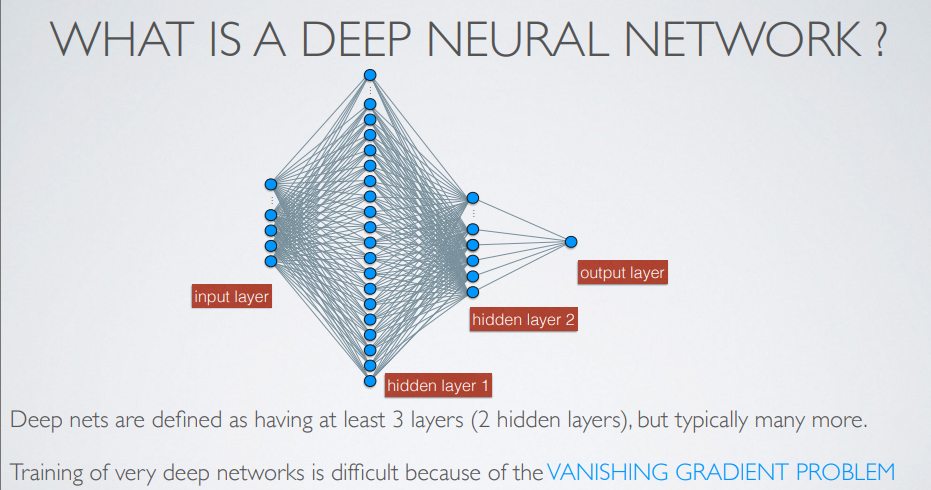
**PAS OUF CAR RÉSOUT UNIQUEMENT LES PB LINÉAIRES .**

Meilleur learning rate :



**Réseaux de neurones :**





Comment faire ?

# Importation des bibliothèques nécessaires

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

# Création d'un modèle séquentiel

model = Sequential()

# Ajout de la couche d'entrée

# L'argument 'input\_dim' spécifie la dimension d'entrée du réseau.

model.add(Dense(units=64, input\_dim=8, activation='relu'))

# - 'units': Le nombre de neurones dans cette couche (64 dans cet exemple).

# - 'input\_dim': La dimension d'entrée du réseau (8 dans cet exemple).

# - 'activation': La fonction d'activation ReLU est utilisée ici.

# Ajout d'une couche cachée

model.add(Dense(units=32, activation='relu'))

# - 'units': Le nombre de neurones dans cette couche (32 dans cet exemple).

# - 'activation': La fonction d'activation ReLU est également utilisée ici.

# Ajout de la couche de sortie

# La fonction d'activation dépendra de la nature de votre problème (par exemple, 'sigmoid' pour la classification binaire).

model.add(Dense(units=1, activation='sigmoid'))

# - 'units': Le nombre de neurones dans cette couche de sortie (1 pour un problème de classification binaire).

# - 'activation': Utilisez 'sigmoid' pour la classification binaire ou 'softmax' pour la classification multiclasse.

# Compilation du modèle

model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])

# - 'loss': La fonction de perte que vous souhaitez minimiser (utilisez 'binary\_crossentropy' pour la classification binaire).

# - 'optimizer': L'optimiseur qui ajuste les poids du réseau (utilisez 'adam' pour commencer).

# - 'metrics': Les métriques que vous souhaitez surveiller pendant l'entraînement (ici, 'accuracy' pour la précision).

# Résumé du modèle, affiche une description de la structure du réseau.

model.summary()

Voici une explication détaillée des différentes parties du code :

* Importation des bibliothèques : Nous importons les bibliothèques TensorFlow et Keras nécessaires pour construire notre modèle.
* Création du modèle : Nous utilisons Sequential() pour créer un modèle séquentiel, qui permet de définir un réseau de neurones couche par couche.
* Ajout de la couche d'entrée : Nous utilisons model.add(Dense(...)) pour ajouter une couche dense (complètement connectée) à notre réseau. Cette couche contient 64 neurones, et input\_dim est défini à 8, ce qui signifie que la dimension d'entrée du réseau est de 8. Nous utilisons l'activation ReLU avec activation='relu'.
* Ajout d'une couche cachée : Nous ajoutons une autre couche dense avec 32 neurones et l'activation ReLU.
* Ajout de la couche de sortie : La couche de sortie a 1 neurone (pour un problème de classification binaire) et utilise l'activation 'sigmoid'. Pour la classification multiclasse, vous pouvez utiliser 'softmax' à la place.
* Compilation du modèle : Nous compilons le modèle en spécifiant la fonction de perte (loss), l'optimiseur (optimizer), et les métriques (metrics) à surveiller pendant l'entraînement. Dans cet exemple, nous utilisons la perte 'binary\_crossentropy' pour la classification binaire et l'optimiseur 'adam'.
* Résumé du modèle : model.summary() affiche une description de la structure du réseau, y compris le nombre de paramètres dans chaque couche.

Vous pouvez personnaliser ce modèle en fonction de votre problème spécifique en ajustant le nombre de neurones, les fonctions d'activation, la fonction de perte, l'optimiseur, etc. Une fois le modèle défini, vous pouvez l'entraîner sur vos données en utilisant la méthode .fit().

En Keras, une "shape" (forme) se réfère généralement à la structure ou aux dimensions d'un tenseur (matrice multidimensionnelle) qui représente des données. La forme d'un tenseur est déterminée par le nombre de dimensions (axes) qu'il possède et la taille de chaque dimension. La forme d'un tenseur est souvent importante lors de la création et de la définition de modèles de réseaux neuronaux, car elle influence la manière dont les données sont traitées et comment les poids et les couches du modèle sont configurés.

Par exemple, si vous avez un tenseur en Keras avec une forme (batch\_size, height, width, channels), cela signifie que :

- "batch\_size" est le nombre d'échantillons (exemples) dans un lot (batch) de données.

- "height" et "width" sont les dimensions spatiales des données, qui sont généralement utilisées pour les images. Par exemple, dans une image 2D de 64x64 pixels, "height" serait égal à 64 et "width" serait égal à 64.

- "channels" (canaux) fait référence aux différentes couches ou canaux dans les données. Dans le cas d'une image en noir et blanc (niveaux de gris), il n'y a qu'un seul canal. Dans le cas d'une image couleur RVB, il y a généralement trois canaux (rouge, vert, bleu).

**CNN :**

*Un réseau de neurones convolutionnels (CNN ou ConvNet) est une architecture de réseau de neurones spécialement conçue pour traiter des données de forme matricielle bidimensionnelle, telles que des images et des vidéos. Les CNN sont particulièrement efficaces pour extraire des caractéristiques pertinentes des données visuelles en utilisant des opérations de convolution et de sous-échantillonnage. Voici une explication détaillée de leur fonctionnement :*

*1. \*\*Structure de base d'un CNN\*\* :*

*- Un CNN est composé de plusieurs couches, généralement organisées en trois types principaux : les couches de convolution, les couches de sous-échantillonnage (ou de pooling), et les couches entièrement connectées (ou Dense layers).*

*2. \*\*Les couches de convolution\*\* :*

*- Les couches de convolution sont le cœur du CNN. Elles utilisent des filtres (kernels) pour effectuer des opérations de convolution sur les données d'entrée (généralement une image).*

*- Chaque filtre parcourt l'image en petits morceaux, appelés fenêtres ou noyaux, et effectue une multiplication scalaire et une sommation pour produire une nouvelle valeur dans une carte de caractéristiques (feature map).*

*- Les caractéristiques importantes, telles que les bords, les textures et les motifs, sont extraites à partir de l'image à l'aide de ces filtres.*

*- Vous pouvez spécifier le nombre de filtres dans chaque couche de convolution.*

*3. \*\*Les couches de sous-échantillonnage (pooling)\*\* :*

*- Les couches de sous-échantillonnage (ou de pooling) sont utilisées pour réduire la dimension des cartes de caractéristiques tout en préservant les caractéristiques importantes.*

*- Le sous-échantillonnage se fait généralement en utilisant des opérations comme le max-pooling, où la valeur maximale dans une fenêtre donnée est conservée, ou le moyennage, où la moyenne des valeurs dans la fenêtre est calculée.*

*- Cela réduit la quantité de calcul nécessaire et aide à éviter le surapprentissage en réduisant le nombre de paramètres du réseau.*

*4. \*\*Les couches entièrement connectées (Dense layers)\*\* :*

*- Les couches entièrement connectées sont similaires aux couches dans les réseaux neuronaux traditionnels (perceptrons multicouches). Elles sont responsables de la classification finale ou de la régénération des données.*

*- Les sorties des couches de convolution et de sous-échantillonnage sont aplatis en un vecteur unidimensionnel et alimentés dans ces couches entièrement connectées.*

*- La dernière couche dense est souvent suivie d'une fonction d'activation appropriée, telle que softmax pour la classification.*

*5. \*\*Apprentissage et rétropropagation du gradient\*\* :*

*- Comme d'autres réseaux neuronaux, les CNN apprennent à partir des données en utilisant l'optimisation itérative des poids des filtres et des couches entièrement connectées.*

*- La rétropropagation du gradient est utilisée pour ajuster les poids du réseau afin de minimiser une fonction de perte, généralement la fonction de perte logistique pour la classification.*

*- Les optimiseurs tels qu'Adam, SGD (descente de gradient stochastique), et RMSprop sont couramment utilisés pour cette tâche.*

*6. \*\*Utilisation des CNN\*\* :*

*- Les CNN sont couramment utilisés dans des tâches telles que la classification d'images, la détection d'objets, la segmentation d'images, la génération d'images, la super-résolution d'images, et bien d'autres.*

*- Ils ont révolutionné le domaine de la vision par ordinateur et ont permis des avancées significatives dans des applications telles que la reconnaissance faciale, la conduite autonome, la médecine et bien d'autres domaines.*

*Pour créer un CNN en utilisant Keras, vous pouvez empiler les couches de convolution, de sous-échantillonnage et de couches entièrement connectées en utilisant la classe `Sequential`. Chaque couche est ajoutée à l'aide de la méthode `add()`. Vous pouvez personnaliser l'architecture en ajustant le nombre de couches, le nombre de filtres, les fonctions d'activation, etc., en fonction de votre tâche spécifique. Une fois le modèle défini, vous pouvez l'entraîner sur vos données à l'aide de la méthode `fit()`.*

Les couches de convolution, telles que Conv1D et Conv2D, sont des composants clés dans les réseaux de neurones convolutionnels (CNN) utilisés pour le traitement des données, en particulier dans des domaines tels que la vision par ordinateur, le traitement du langage naturel et d'autres domaines où la reconnaissance de motifs est importante. La principale différence entre Conv1D et Conv2D réside dans la dimension des données sur lesquelles elles agissent :

1. \*\*Conv1D (Convolution 1D)\*\* :

- Conv1D est principalement utilisée pour le traitement de données unidimensionnelles, telles que des séquences temporelles, des signaux audio ou des séquences de texte.

- Elle applique des opérations de convolution sur une seule dimension (généralement le temps ou une séquence) en utilisant un filtre 1D (noyau 1D) qui se déplace le long de la séquence.

- Conv1D est souvent utilisée pour extraire des motifs séquentiels ou temporels, par exemple, dans la reconnaissance de la parole ou l'analyse de séquences de mots.

2. \*\*Conv2D (Convolution 2D)\*\* :

- Conv2D est principalement utilisée pour le traitement de données bidimensionnelles, telles que des images en niveaux de gris ou en couleurs.

- Elle applique des opérations de convolution bidimensionnelle en utilisant un filtre 2D (noyau 2D) qui parcourt l'image.

- Conv2D est couramment utilisée dans les CNN pour extraire des motifs spatiaux à partir d'images, ce qui est essentiel pour des tâches telles que la classification d'images, la détection d'objets, etc.

**Réseau récurents :**

Un réseau de neurones récurrents (RNN) est un type de réseau de neurones artificiels conçu pour traiter des séquences de données, telles que des séquences temporelles, des séquences de texte, des signaux audio, etc.

Le principal composant des RNN est une couche de cellules récurrentes. Chaque cellule récurrente prend en entrée une donnée de la séquence courante et l'état caché (ou la mémoire) de la cellule précédente. L'état caché évolue à mesure que la séquence est traitée, en incorporant les informations des états cachés précédents et les données actuelles. Cela permet aux RNN de capturer les dépendances séquentielles à différents niveaux d'abstraction.

*Voici un aperçu du fonctionnement des RNN :*

*1. \*\*Entrée de données séquentielles\*\* : Les données séquentielles sont introduites dans le réseau une étape à la fois, avec chaque étape représentant une unité de temps ou une position dans la séquence.*

*2. \*\*Cellules récurrentes\*\* : À chaque étape, les données actuelles et l'état caché de la cellule précédente sont combinés pour calculer l'état caché de la cellule courante. Les cellules récurrentes ont des poids partagés pour toutes les étapes de la séquence.*

*3. \*\*Propagation de l'information\*\* : L'information est propagée d'une étape à l'autre à mesure que la séquence est traitée. L'état caché évolue à chaque étape en prenant en compte les informations passées.*

*4. \*\*Sortie\*\* : À la fin de la séquence ou à chaque étape, selon la tâche, une sortie est générée en fonction de l'état caché courant.*

**Pour créer et entraîner un réseau de neurones récurrents en utilisant Keras (une bibliothèque de deep learning populaire), voici les étapes de base :**

1. \*\*Importez les bibliothèques nécessaires\*\* :

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import SimpleRNN, Dense

2. \*\*Créez le modèle RNN\*\* :

**model = Sequential()**

**# Ajoutez une couche récurrente SimpleRNN au modèle**

**model.add(SimpleRNN(units=64, input\_shape=(timesteps, features), activation='relu'))**

- `units` : Le nombre de neurones dans la couche récurrente.

- `input\_shape` : La forme de l'entrée (nombre d'étapes de temps, nombre de caractéristiques).

- `activation` : L'activation à utiliser (par exemple, 'relu').

3. \*\*Ajoutez d'autres couches si nécessaire\*\* :

Vous pouvez ajouter d'autres couches Dense (entièrement connectées) ou d'autres couches récurrentes selon votre tâche.

4. \*\*Compilez le modèle\*\* :

model.compile(loss='mean\_squared\_error', optimizer='adam')

- `loss` : La fonction de perte appropriée pour votre tâche.

- `optimizer` : L'optimiseur que vous souhaitez utiliser.

5. \*\*Entraînez le modèle\*\* :

model.fit(X\_train, y\_train, epochs=epochs, batch\_size=batch\_size)

- `X\_train` : Vos données d'entraînement.

- `y\_train` : Les étiquettes correspondantes.

- `epochs` : Le nombre d'époques d'entraînement.

- `batch\_size` : La taille des mini-lots.

6. \*\*Évaluez le modèle\*\* :

loss, accuracy = model.evaluate(X\_test, y\_test)

7. \*\*Faites des prédictions\*\* :

predictions = model.predict(X\_new)

N'oubliez pas d'adapter ces étapes à votre tâche spécifique, en ajustant la structure du modèle, la fonction de perte et l'optimiseur en fonction de vos besoins. Les RNN sont couramment utilisés pour des tâches telles que la prédiction de séries temporelles, la génération de texte, la traduction automatique et bien d'autres.

**Transformers :**

transformer des mots en vecteur: liens entre mots par exemple dans une phrase.

encoder/decoder. Matrice mult ++

Les transformers sont une architecture de réseau de neurones qui a révolutionné le domaine du traitement du langage naturel (NLP) et qui s'est avérée très efficace pour une variété de tâches de traitement du langage naturel. Ils ont été introduits pour la première fois par Vaswani et al. en 2017 dans l'article "Attention Is All You Need" et depuis lors, ils sont devenus la pierre angulaire de nombreuses applications de NLP et d'autres domaines de l'apprentissage automatique.

*Voici les principales caractéristiques et concepts des transformers :*

*1. \*\*Auto-attention (Self-Attention)\*\* :*

*- L'élément central des transformers est le mécanisme d'auto-attention, qui permet au modèle d'accorder une attention différente à différentes parties de la séquence d'entrée.*

*- L'auto-attention permet au modèle de capturer des dépendances à longue distance dans la séquence de manière efficace, ce qui était un défi pour les architectures précédentes.*

*2. \*\*Architecture sans récurrence ni convolution\*\* :*

*- Contrairement aux réseaux de neurones récurrents (RNN) et aux réseaux de neurones convolutionnels (CNN), les transformers n'ont pas de couches récurrentes ni de couches de convolution.*

*- Ils se composent de plusieurs couches d'attention empilées les unes sur les autres.*

*3. \*\*Mécanisme d'encodage et de décodage\*\* :*

*- Les transformers sont souvent utilisés dans des tâches de séquence-à-séquence, telles que la traduction automatique. Ils utilisent une architecture d'encodeur-décodeur, où l'encodeur capture des informations de la séquence source et le décodeur génère la séquence cible.*

*4. \*\*Positional Encoding\*\* :*

*- Les transformers n'ont pas de notion intrinsèque d'ordre dans les données séquentielles. Pour incorporer des informations sur la position des éléments dans la séquence, ils utilisent l'encodage positionnel.*

*- L'encodage positionnel est ajouté aux embeddings de chaque élément de la séquence pour indiquer sa position relative.*

*5. \*\*Multi-Head Attention\*\* :*

*- Dans les transformers, l'auto-attention est souvent implémentée sous la forme de "multi-head attention". Cela signifie que plusieurs mécanismes d'auto-attention fonctionnent en parallèle, ce qui permet au modèle de capturer différentes relations entre les éléments de la séquence.*

*6. \*\*Residual Connections et Layer Normalization\*\* :*

*- Pour faciliter la formation de réseaux profonds, les transformers utilisent des connexions résiduelles et la normalisation de couche (Layer Normalization).*

*7. \*\*Fonctions d'activation ReLU et GELU\*\* :*

*- Les fonctions d'activation ReLU (Rectified Linear Unit) et GELU (Gaussian Error Linear Unit) sont couramment utilisées dans les transformers.*

*8. \*\*Modèles pré-entraînés\*\* :*

*- Les transformers sont souvent pré-entraînés sur de grands corpus de texte non annoté, puis fine-tunés sur des tâches spécifiques. Les modèles pré-entraînés, tels que BERT, GPT-2, et d'autres, ont atteint des performances impressionnantes sur une gamme de tâches de NLP.*

*9. \*\*Applications\*\* :*

*- Les transformers sont utilisés dans une variété de tâches de NLP, notamment la classification de texte, la génération de texte, la traduction automatique, la compréhension de texte, etc. Ils ont également été adaptés à d'autres domaines, comme la vision par ordinateur.*

Les transformers ont considérablement amélioré la capacité des modèles à comprendre et à générer du texte de manière cohérente et contextuellement précise. Ils ont également été adaptés avec succès à d'autres types de données séquentielles et tabulaires, ce qui en fait une architecture polyvalente dans le domaine de l'apprentissage automatique.

**Autoencoders :**

Les autoencodeurs sont une classe de réseaux neuronaux utilisés en apprentissage automatique et en intelligence artificielle pour la compression de données, la réduction de dimensionnalité, la génération de données et d'autres tâches liées à la représentation des données. Ils sont principalement utilisés dans le domaine de l'apprentissage non supervisé.

La structure d'un autoencodeur est composée de deux parties principales : l'encodeur et le décodeur. Voici comment fonctionnent les autoencodeurs :

*1. \*\*Encodeur (Encoder)\*\* :*

*- L'encodeur prend en entrée des données brutes ou des caractéristiques et les transforme en une représentation de dimension réduite, généralement appelée "espace latent" ou "représentation latente".*

*- L'objectif de l'encodeur est de capturer les caractéristiques essentielles des données tout en réduisant leur dimensionnalité.*

*2. \*\*Décodeur (Decoder)\*\* :*

*- Le décodeur prend la représentation latente produite par l'encodeur et tente de la reconstruire en une forme similaire aux données d'origine.*

*- L'objectif du décodeur est de générer des données de sortie qui sont aussi proches que possible des données d'entrée.*

*3. \*\*Fonction de coût (Loss Function)\*\* :*

*- La fonction de coût mesure l'écart entre les données reconstruites par le décodeur et les données d'entrée. L'objectif est de minimiser cette erreur de reconstruction.*

*- Les fonctions de coût couramment utilisées incluent la moyenne des moindres carrés (MSE) pour les données continues ou la perte de log vraisemblance pour les données binaires.*

*4. \*\*Entraînement\*\* :*

*- Pendant l'entraînement, l'autoencodeur ajuste les poids de l'encodeur et du décodeur pour minimiser la fonction de coût.*

*- À mesure que le modèle s'entraîne, il apprend à extraire des caractéristiques utiles de la représentation latente, ce qui peut être utile pour des tâches telles que la compression d'images, la réduction de bruit, la détection d'anomalies, etc.*

*5. \*\*Utilisations\*\* :*

*- Les autoencodeurs sont utilisés dans diverses applications, notamment la réduction de dimensionnalité pour la visualisation de données, la génération d'images, la détection d'anomalies (anomaly detection), la suppression de bruit, la génération de texte, etc.*

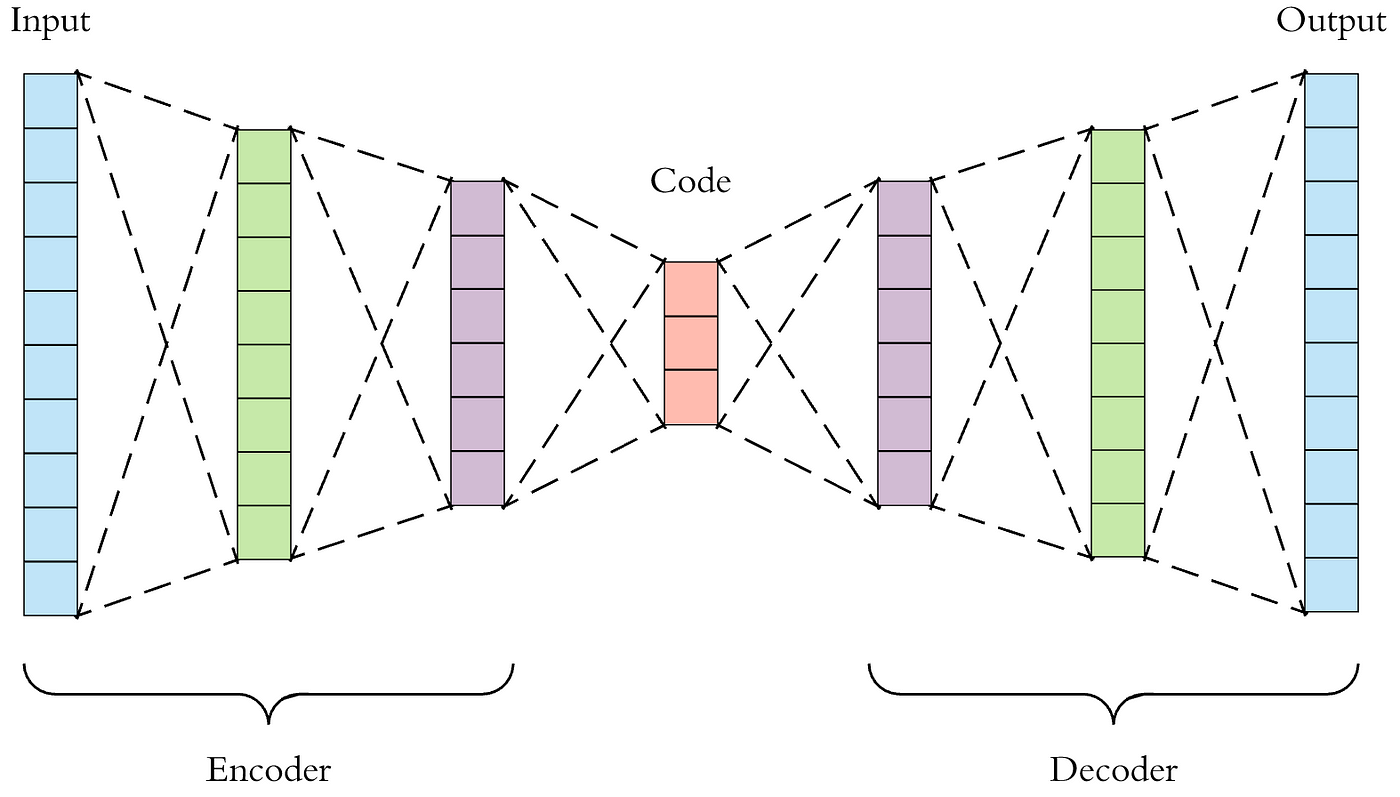
*- Ils sont également utilisés comme composants de base dans d'autres architectures de réseaux neuronaux, comme les réseaux générateurs adverses (GAN) et les variational autoencoders (VAE).*

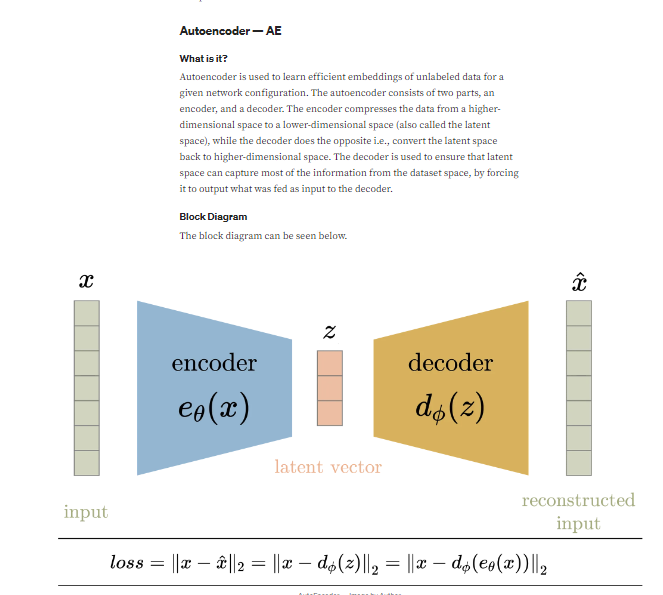
*6. \*\*Variations\*\* :*

*- Il existe plusieurs variations d'autoencodeurs, notamment les autoencodeurs convolutionnels (CAE) pour les données d'image, les autoencodeurs variationnels (VAE) qui introduisent une structure probabiliste dans la représentation latente, et d'autres.*

Les autoencodeurs sont des outils puissants pour l'apprentissage non supervisé et la génération de données, et ils sont souvent utilisés pour extraire des représentations significatives des données, ce qui peut être utile dans de nombreux domaines de l'intelligence artificielle et de l'apprentissage automatique.

La loss s’exerce sur des données latentes.





https://towardsdatascience.com/difference-between-autoencoder-ae-and-variational-autoencoder-vae-ed7be1c038f2