

# Apprentissage Profond (Deep Learning)

L'apprentissage profond est une sous-discipline de l'intelligence artificielle qui se distingue par des empilements profonds de calculs, permettant de détecter des schémas complexes dans les données. Cette approche a permis des avancées majeures dans des domaines comme la traduction, la reconnaissance d'images et le jeu vidéo, où les modèles ont parfois atteint ou dépassé les performances humaines.

Les réseaux de neurones, composés de neurones interconnectés, sont au cœur de l'apprentissage profond. Bien que chaque neurone effectue un calcul simple, la puissance des réseaux réside dans la complexité des connexions entre ces neurones, ce qui leur permet de modéliser des relations hiérarchiques complexes.

## Le Neurone

Un neurone individuel dans un réseau de neurones fonctionne selon une équation linéaire simple :  $y = w * x + b$ .

Voici une explication du diagramme du neurone :

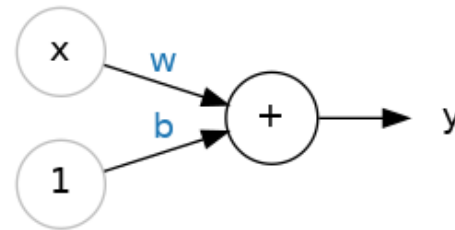
$x$  est l'entrée du neurone.

$w$  est le poids de la connexion associée à l'entrée. Il modifie l'importance de l'entrée  $x$  en la multipliant par  $w$ .

$b$ , appelé biais, est une constante qui permet de décaler la sortie du neurone, indépendamment des entrées.

Le neurone calcule la sortie  $y$  en ajoutant le produit  $w * x$  au biais  $b$ .

L'équation  $y = w * x + b$  est une simple équation de ligne droite, où  $w$  représente la pente, et  $b$  l'ordonnée à l'origine. Le neurone "apprend" en ajustant ces poids  $w$  et  $b$  pour mieux modéliser les données.



*The Linear Unit:  $y = wx + b$*

Dans cet exemple, nous utilisons un neurone individuel (ou unité linéaire) pour modéliser une relation linéaire entre deux variables : le sucre et les calories dans l'ensemble de données "80 Céréales".

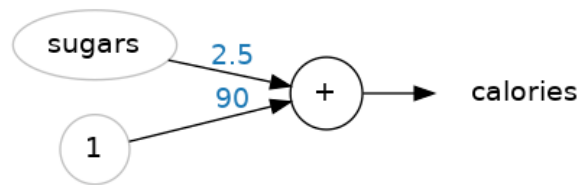
Le modèle prédit les calories en fonction de la quantité de sucre par portion, avec la formule  $y = w * x + b$ , où :

$x$  est la quantité de sucre,

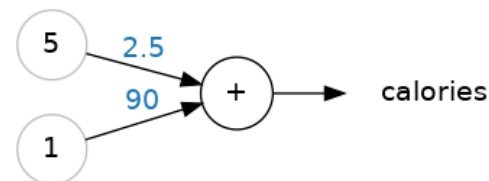
$w$  est le poids (ici, 2,5), qui indique l'impact du sucre sur les calories,

$b$  est le biais (ici, 90), qui ajuste la sortie indépendamment du sucre.

En appliquant la formule, pour une céréale avec 5 grammes de sucre :  $\text{calories} = 2.5 \times 5 + 90 = 102.5$  Cette valeur correspond aux calories estimées pour cette portion, démontrant comment un neurone simple peut modéliser une relation linéaire.

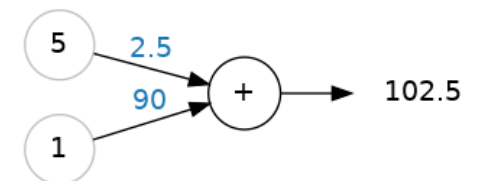


$w=2.5, b=90$



Input sugars=5

*Computing with the linear unit.*



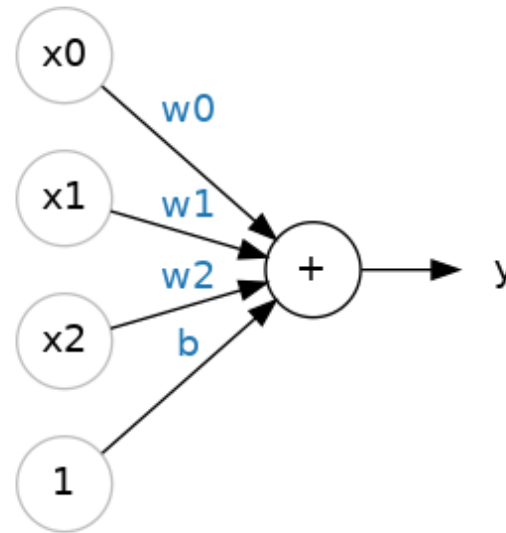
Multiply inputs by weights and sum.

Lorsque nous avons plusieurs caractéristiques d'entrée, comme la teneur en fibres ou en protéines en plus du sucre, nous pouvons étendre le modèle du neurone en ajoutant plus de connexions d'entrée, une pour chaque caractéristique supplémentaire.

Chaque connexion d'entrée est associée à un poids, et la sortie du neurone est calculée en multipliant chaque entrée par son poids respectif,

puis en additionnant les résultats avec le biais b. La formule devient :  $y = w_0x_0 + w_1x_1 + w_2x_2 + b$

Ici, chaque x représente une caractéristique différente (comme le sucre, les fibres, les protéines), et chaque w est le poids associé à cette caractéristique. Ce modèle permet de prendre en compte plusieurs facteurs pour prédire la sortie. Si le modèle a deux entrées, il ajustera un plan, et avec plus d'entrées, il ajustera un hyperplan dans un espace multidimensionnel.



*A linear unit with three inputs.*

## Exemple

```
In [1]: from tensorflow import keras
        from tensorflow.keras import layers

        # Create a network with 1 linear unit
        model = keras.Sequential([
            layers.Dense(units=1, input_shape=[3])
        ])
```

```
2024-12-18 08:52:56.505044: I external/local_xla/xla/tsl/cuda/cudart_stub.cc:32] Could not find cuda drive
rs on your machine, GPU will not be used.
2024-12-18 08:52:56.565231: I external/local_xla/xla/tsl/cuda/cudart_stub.cc:32] Could not find cuda drive
rs on your machine, GPU will not be used.
2024-12-18 08:52:56.623817: E external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:485] Unable to regis
ter cuFFT factory: Attempting to register factory for plugin cuFFT when one has already been registered
2024-12-18 08:52:56.686672: E external/local_xla/xla/stream_executor/cuda/cuda_dnn.cc:8454] Unable to regi
ster cuDNN factory: Attempting to register factory for plugin cuDNN when one has already been registered
2024-12-18 08:52:56.705740: E external/local_xla/xla/stream_executor/cuda/cuda_blas.cc:1452] Unable to reg
ister cuBLAS factory: Attempting to register factory for plugin cuBLAS when one has already been registere
d
2024-12-18 08:52:56.806656: I tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow binary is
optimized to use available CPU instructions in performance-critical operations.
To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow with the appropria
te compiler flags.
2024-12-18 08:52:58.332065: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could
not find TensorRT
/home/pnl/anaconda3/lib/python3.10/site-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not p
ass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input
(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Dans cet exemple, nous définissons deux arguments clés pour configurer un modèle dans Keras :

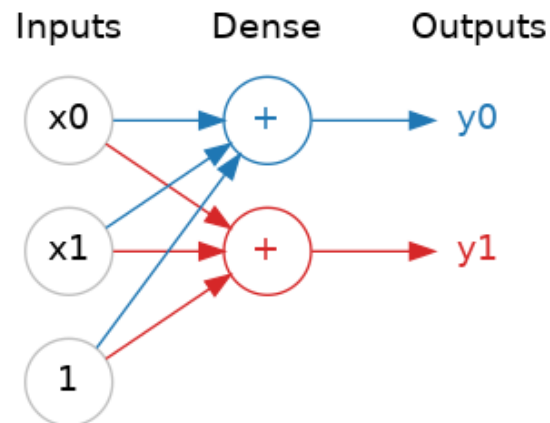
`units=1` : Cela spécifie le nombre de sorties du modèle. Ici, comme nous prédisons seulement les calories, nous utilisons 1 sortie.

`input_shape=[3]` : Cela définit la dimension des entrées. Ici, comme le modèle utilise trois caractéristiques (sucre, fibres et protéines), nous définissons `input_shape=[3]`, ce qui indique que le modèle accepte trois entrées.

## Couches (layers)

Les réseaux de neurones organisent leurs neurones en couches, où chaque couche effectue une transformation des données. Lorsque plusieurs unités linéaires reçoivent les mêmes entrées, elles forment une couche dense. Dans une telle couche, chaque neurone reçoit toutes les entrées et effectue un calcul pour produire une sortie.

Chaque couche réalise une transformation relativement simple des données, mais en empilant plusieurs couches, le réseau peut apprendre des transformations de plus en plus complexes. Cela permet au réseau de traiter les données de manière hiérarchique, où chaque couche affine un peu plus la solution finale.



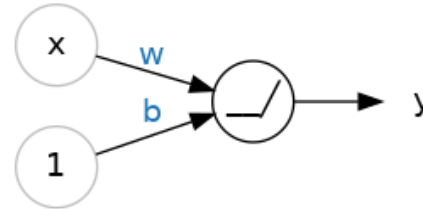
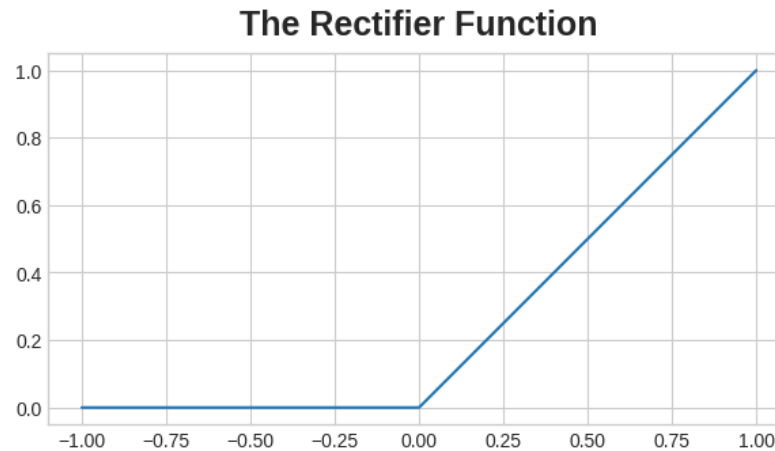
*A dense layer of two linear units receiving two inputs 1 bias.*

## La Fonction d'Activation

Dans un réseau de neurones, deux couches denses empilées sans fonction d'activation ne peuvent modéliser que des relations linéaires. Cela signifie que, même si on empile plusieurs couches, le réseau ne pourra apprendre que des lignes ou des plans. Pour capturer des relations non linéaires (comme des courbes), il est nécessaire d'introduire des fonctions d'activation.

Une fonction d'activation est appliquée à la sortie de chaque neurone d'une couche. La plus courante est la fonction ReLU (Rectified Linear Unit), définie par  $\max(0, x)$ . Elle "rectifie" les valeurs négatives à zéro, tout en conservant les valeurs positives inchangées. Cette fonction permet au réseau de modéliser des relations non linéaires, ce qui est essentiel pour résoudre des problèmes complexes.

Lorsqu'une fonction ReLU est appliquée à une unité linéaire, on obtient une unité linéaire rectifiée (ReLU). Cela signifie que la sortie du neurone devient  $\max(0, w * x + b)$ , ce qui introduit une courbure dans les données et permet au réseau de mieux capturer des relations plus complexes.

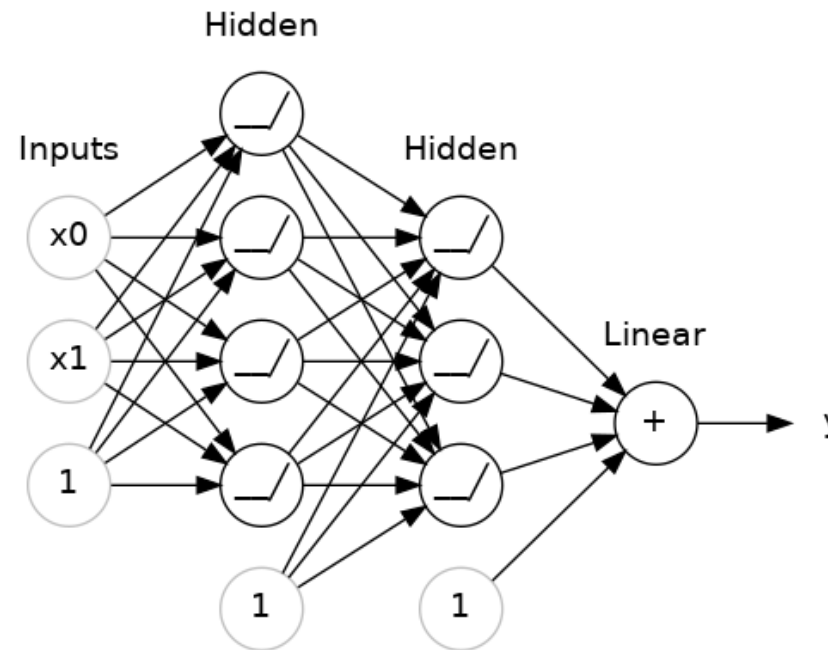


*A rectified linear unit.*

## Plusieurs Couches Denses

Lorsque l'on empile plusieurs couches denses, cela crée un réseau de neurones entièrement connecté, capable de réaliser des transformations de données complexes. Chaque couche intermédiaire, appelée couche cachée, reçoit les entrées de la couche précédente et applique ses transformations. Ces couches sont "cachées" car leurs sorties ne sont pas directement visibles, elles ne servent qu'à préparer les données pour la couche suivante.

La dernière couche, appelée couche de sortie, est ici une unité linéaire (sans fonction d'activation), ce qui en fait un choix adapté pour les tâches de régression, où l'objectif est de prédire une valeur numérique continue. En revanche, pour des tâches de classification, où l'on souhaite prédire une catégorie, une fonction d'activation (comme softmax ou sigmoïde) serait appliquée à la couche de sortie pour obtenir des probabilités.



## Construction de Modèles Séquentiels

Le modèle séquentiel que nous avons utilisé va connecter une liste de couches les unes après les autres, de la première à la dernière : la première couche reçoit l'entrée, et la dernière couche produit la sortie. Cela crée le modèle illustré dans la figure ci-dessus..

```
In [2]: from tensorflow import keras
from tensorflow.keras import layers

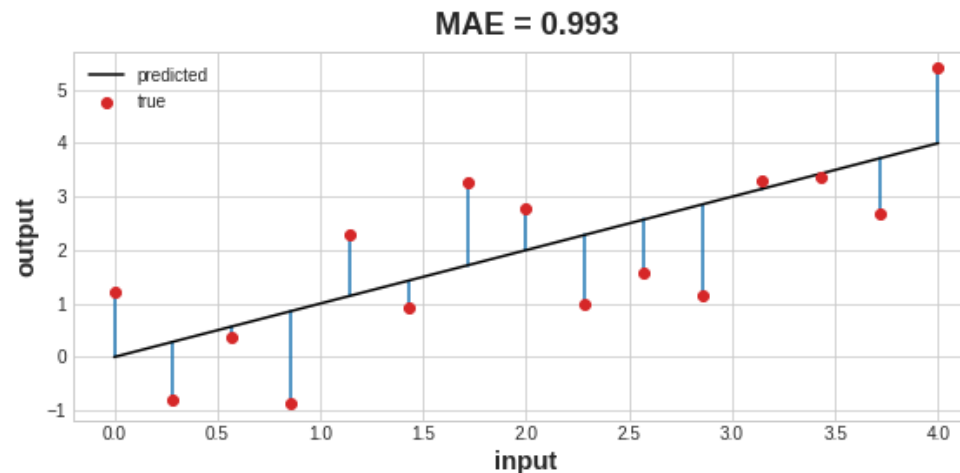
model = keras.Sequential([
    # the hidden ReLU layers
    layers.Dense(units=4, activation='relu', input_shape=[2]),
    layers.Dense(units=3, activation='relu'),
    # the linear output layer
    layers.Dense(units=1),
])
```

Assurez-vous de passer toutes les couches ensemble dans une liste, comme [couche, couche, couche, ...], au lieu de les passer en arguments séparés. Pour ajouter une fonction d'activation à une couche, il suffit de donner son nom dans l'argument activation.

## La Fonction de Perte (the loss function)

La fonction de perte joue un rôle crucial dans l'entraînement d'un réseau de neurones en mesurant l'écart entre les prédictions du modèle et les valeurs réelles de la cible. Elle sert de guide au modèle pour ajuster ses poids afin de minimiser cet écart.

Pour les tâches de régression (prédiction de valeurs numériques), une fonction de perte couramment utilisée est la MAE (Erreur Absolue Moyenne). La MAE calcule la différence absolue entre chaque prédiction ( $y_{\text{pred}}$ ) et la vraie valeur cible ( $y_{\text{true}}$ ), puis prend la moyenne de ces écarts. Plus la MAE est faible, meilleure est la performance du modèle.





# L'Optimiseur - Descente de Gradient Stochastique

L'optimiseur est l'algorithme qui ajuste les poids du réseau pour minimiser la perte. En apprentissage profond, l'optimisation est généralement effectuée via la descente de gradient stochastique (SGD). Cet algorithme itératif ajuste les poids par petites étapes, de la manière suivante :

Un échantillon de données d'entraînement est passé à travers le réseau pour faire des prédictions.

La perte entre les prédictions et les valeurs réelles est mesurée.

Les poids sont ajustés dans la direction qui réduit la perte, guidés par le gradient.

Le gradient indique comment ajuster les poids pour réduire la perte le plus rapidement possible, ce qui est appelé la "descente de gradient". Le terme stochastique (aléatoire) fait référence à l'utilisation d'échantillons aléatoires de données (appelés minibatches) à chaque étape d'entraînement.

Un epoch représente un passage complet sur toutes les données d'entraînement, tandis que chaque itération utilise un minibatch. À chaque minibatch, les poids du réseau sont ajustés. Au fil du temps, la perte diminue et les poids se rapprochent de leurs valeurs optimales, comme montré dans l'animation où le modèle ajuste progressivement la pente ( $w$ ) et l'ordonnée à l'origine ( $b$ ) pour mieux s'ajuster aux données.



# Taux d'Apprentissage et Taille des Batches (learning rate & batch size)

Lors de l'entraînement avec la descente de gradient stochastique (SGD), la taille des ajustements que le modèle fait à chaque itération est contrôlée par le taux d'apprentissage. Un taux d'apprentissage plus faible entraîne des ajustements plus petits, ce qui signifie que le réseau doit traiter davantage de minibatches avant que ses poids ne convergent vers leurs valeurs optimales.

Les deux paramètres les plus influents dans le processus d'entraînement sont :

- 1 Le taux d'apprentissage : détermine la vitesse à laquelle les poids sont ajustés.
- 2 La taille des minibatches : influence la fréquence des mises à jour des poids.

Trouver le bon équilibre entre ces paramètres peut être délicat, mais il existe des optimisateurs comme Adam, qui utilise un taux d'apprentissage adaptatif. Cela signifie qu'Adam ajuste automatiquement la vitesse d'apprentissage tout au long de l'entraînement, ce qui le rend adapté à la plupart des tâches sans besoin de réglage minutieux des hyperparamètres. Adam est donc souvent préféré pour son efficacité et sa capacité à s'auto-ajuster pour de bons résultats.

## La classification binaire

C'est un problème courant en apprentissage automatique où l'objectif est de prédire l'une des deux classes possibles. Par exemple, on peut vouloir prédire si un client va effectuer un achat ou non, si une transaction est frauduleuse, ou si un test médical révèle la présence d'une maladie.

Dans les données initiales, les classes peuvent être représentées par des chaînes comme "Oui"/"Non" ou "Chien"/"Chat". Avant d'entraîner un modèle, on attribue des étiquettes numériques aux classes, par exemple 0 pour une classe et 1 pour l'autre. Cela permet au réseau de neurones de traiter les données de manière efficace. Ce processus de conversion des classes en étiquettes numériques est essentiel pour que les modèles puissent effectuer la classification binaire.

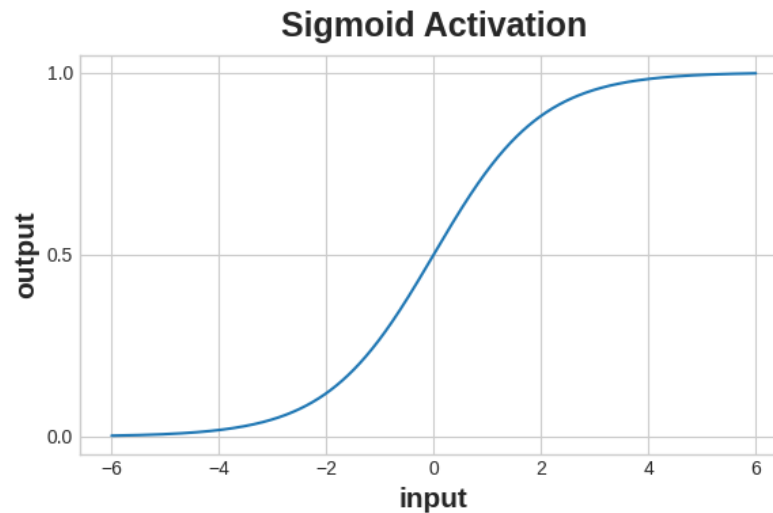
## Produire des probabilités avec la fonction Sigmoidale

La précision est une métrique utilisée en classification, définie comme le rapport entre les prédictions correctes et le nombre total de prédictions.

Un score de 1,0 signifie que toutes les prédictions sont correctes. Cependant, la précision est surtout utile lorsque les classes sont équilibrées.

Le problème avec la précision est qu'elle ne peut pas servir de fonction de perte, car elle ne change pas de manière continue, ce qui est nécessaire pour la descente de gradient stochastique (SGD). Pour les tâches de classification, on utilise plutôt l'entropie croisée, qui mesure la distance entre les probabilités prédites et les vraies probabilités.

Les fonctions d'entropie croisée et de précision nécessitent des probabilités comme entrées, c'est-à-dire des nombres allant de 0 à 1. Pour convertir les sorties à valeur réelle produites par une couche dense en probabilités, nous attachons un nouveau type de fonction d'activation, la fonction d'activation sigmoïde.



*The sigmoid function maps real numbers into the interval  $[0, 1]$ .*

Pour obtenir la prédiction finale de la classe, nous définissons une probabilité de seuil. Typiquement, ce seuil sera de 0,5, de sorte que l'arrondi nous donnera la classe correcte : en dessous de 0,5, cela signifie la classe avec l'étiquette 0, et à 0,5 ou au-dessus, cela signifie la classe avec l'étiquette 1. Un seuil de 0,5 est celui utilisé par défaut par Keras avec sa métrique [accuracy].

## Exemple de classification binaire avec keras

```
In [3]: # Loading Necessary Libraries
import pandas as pd
from IPython.display import display
from sklearn.impute import SimpleImputer

# Load the dataset
red_wine = pd.read_csv('./winequality.csv')

# 'quality' is a binary classification where:
# "good" -> 1 and "bad" -> 0
# First, convert 'quality' to numeric representation

# We create a dictionary that maps the string values in the quality column to numbers
quality_mapping = {'good': 1, 'bad': 0}

# This replaces the string values in the quality column with the corresponding numeric values
red_wine['quality'] = red_wine['quality'].map(quality_mapping)

# Drop any rows where 'quality' is still NaN (because we can't train on rows without labels)
red_wine = red_wine.dropna(subset=['quality'])

# Now handle the feature columns (all except 'quality')
# Drops the quality column from the DataFrame because we don't want to apply transformations on the target
# Convert all other columns to numeric, forcing errors to NaN if any non-numeric values are found
# Converts all the remaining feature columns to numeric types.
# If there are any non-numeric values, they are converted to NaN
red_wine_numeric = red_wine.drop(columns=['quality']).apply(pd.to_numeric, errors='coerce')

# Impute missing values for the numeric columns (using mean imputation)
# This creates an imputer that fills missing values (NaN) in the DataFrame using the mean of each column
imputer = SimpleImputer(strategy='mean')

# This applies the imputer to the DataFrame. It computes the mean for each column and replaces the missing
# Converts the imputed data back into a DataFrame and ensures the column names remain the same.
red_wine_numeric_imputed = pd.DataFrame(imputer.fit_transform(red_wine_numeric), columns=red_wine_numeric.columns)

# Combine imputed numeric columns with the 'quality' column
# We need to do this because we had earlier separated quality from the feature columns.
# Ensures that the index of the quality column aligns with the imputed feature DataFrame.
red_wine_imputed = pd.concat([red_wine_numeric_imputed, red_wine['quality'].reset_index(drop=True)], axis=1)
```

```

# Create training and validation splits
df_train = red_wine_imputed.sample(frac=0.7, random_state=0)
df_valid = red_wine_imputed.drop(df_train.index)

# calculate the maximum and minimum values for each feature column (excluding quality) in the training set
max_ = df_train.drop(columns=['quality']).max(axis=0)
min_ = df_train.drop(columns=['quality']).min(axis=0)

# Feature scaling:
# Scale each feature to a range between 0 and 1 using min-max scaling.
# The formula  $(x - \min) / (\max - \min)$  is applied to each feature.
# This step ensures that all feature values are on a similar scale,
# which helps the neural network converge more effectively during training.
df_train.loc[:, df_train.columns != 'quality'] = (df_train.loc[:, df_train.columns != 'quality'] - min_) /
df_valid.loc[:, df_valid.columns != 'quality'] = (df_valid.loc[:, df_valid.columns != 'quality'] - min_) /

# Split features and target
X_train = df_train.drop('quality', axis=1)
X_valid = df_valid.drop('quality', axis=1)
y_train = df_train['quality']
y_valid = df_valid['quality']

# Display the result to ensure everything looks good
display(X_train.head(), y_train.head())

# Import TensorFlow and Keras
from tensorflow import keras
from tensorflow.keras import layers

# Define the model
# Eleven columns means eleven inputs.
# We've chosen a three-layer network with over 1500 neurons. This network should be capable
# of learning fairly complex relationships in the data.
# The Input layer specifies the shape of the input features : X_train.shape[1] (the number of feature columns)
# Dense layers are fully connected.
# Relu is used as activation function
# The final Dense layer has one neuron, with a sigmoid activation function (sigmoid).
# Sigmoid squashes the output between 0 and 1 in this case.
# Sigmoid is appropriate for binary classification problems.

```

```

model = keras.Sequential([
    layers.Input(shape=[X_train.shape[1]]), # Automatically infer input shape from the training data
    layers.Dense(512, activation='relu'),
    layers.Dense(512, activation='relu'),
    layers.Dense(512, activation='relu'),
    layers.Dense(1, activation='sigmoid'), # Since it's binary classification, use sigmoid activation
])

# Compile the model
# We use the Adam optimizer, which is a popular choice for training deep learning models.
# Since this is a binary classification problem, we use binary_crossentropy as the loss function.
# We care about classification accuracy, so we use accuracy as the evaluation metric
model.compile(
    optimizer='adam',
    loss='binary_crossentropy', # Use binary_crossentropy for binary classification
    metrics=['accuracy'] # We care about accuracy for classification
)

# Train the model
# This trains the model using the training data (X_train and y_train) for 10 epochs.
# We pass validation data, so the model evaluate its performance after each epoch.
# The model processes 256 samples at a time before updating the weights.
history = model.fit(
    X_train, y_train,
    validation_data=(X_valid, y_valid),
    batch_size=256,
    epochs=30,
)

# convert the training history to a dataframe
history_df = pd.DataFrame(history.history)
# use Pandas native plot method
#history_df['loss'].plot();
#history_df['val_loss'].plot();

```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol
<b>1109</b>	0.548673	0.239726	0.544304	0.092308	0.237435	0.366197	0.212014	0.619193	0.291262	0.260606	0.369231
<b>1032</b>	0.309735	0.479452	0.000000	0.246154	0.105719	0.056338	0.028269	0.645088	0.475728	0.121212	0.184615
<b>1002</b>	0.398230	0.116438	0.417722	0.088462	0.050260	0.169014	0.074205	0.387662	0.378641	0.309091	0.507692
<b>487</b>	0.495575	0.359589	0.455696	0.069231	0.032929	0.056338	0.028269	0.619193	0.291262	0.054545	0.246154
<b>979</b>	0.672566	0.226027	0.620253	0.038462	0.071057	0.028169	0.000000	0.520183	0.252427	0.181818	0.307692

1109 1














1032 0

1002 1


487 1


979 0


Name: quality, dtype: int64


Epoch 1/30  
5/5  1s 56ms/step - accuracy: 0.5443 - loss: 0.6759 - val\_accuracy: 0.6812 - val\_loss: 0.6374  
Epoch 2/30  
5/5  0s 25ms/step - accuracy: 0.6833 - loss: 0.6294 - val\_accuracy: 0.7521 - val\_loss: 0.5518  
Epoch 3/30  
5/5  0s 26ms/step - accuracy: 0.7146 - loss: 0.5735 - val\_accuracy: 0.7563 - val\_loss: 0.5179  
Epoch 4/30  
5/5  0s 24ms/step - accuracy: 0.7327 - loss: 0.5449 - val\_accuracy: 0.6979 - val\_loss: 0.5562  
Epoch 5/30  
5/5  0s 25ms/step - accuracy: 0.7226 - loss: 0.5550 - val\_accuracy: 0.7604 - val\_loss: 0.5062  
Epoch 6/30  
5/5  0s 24ms/step - accuracy: 0.7418 - loss: 0.5338 - val\_accuracy: 0.7458 - val\_loss: 0.5091  
Epoch 7/30  
5/5  0s 25ms/step - accuracy: 0.7539 - loss: 0.5213 - val\_accuracy: 0.7458 - val\_loss: 0.5069  
Epoch 8/30  
5/5  0s 25ms/step - accuracy: 0.7463 - loss: 0.5149 - val\_accuracy: 0.7521 - val\_loss: 0.4996  
Epoch 9/30  
5/5  0s 25ms/step - accuracy: 0.7456 - loss: 0.5231 - val\_accuracy: 0.7500 - val\_loss: 0.4995  
Epoch 10/30  
5/5  0s 23ms/step - accuracy: 0.7604 - loss: 0.4966 - val\_accuracy: 0.7583 - val\_loss: 0.4894  
Epoch 11/30  
5/5  0s 26ms/step - accuracy: 0.7605 - loss: 0.5001 - val\_accuracy: 0.7333 - val\_loss: 0.5110  
Epoch 12/30  
5/5  0s 25ms/step - accuracy: 0.7555 - loss: 0.5022 - val\_accuracy: 0.7583 - val\_loss: 0.4886  
Epoch 13/30  
5/5  0s 25ms/step - accuracy: 0.7665 - loss: 0.4884 - val\_accuracy: 0.7292 - val\_loss: 0.5180  
Epoch 14/30





5/5  0s 26ms/step - accuracy: 0.7609 - loss: 0.4891 - val\_accuracy: 0.7521 - val\_loss: 0.4929  
Epoch 15/30


5/5  0s 27ms/step - accuracy: 0.7799 - loss: 0.4793 - val\_accuracy: 0.7604 - val\_loss: 0.4870  
Epoch 16/30


5/5  0s 26ms/step - accuracy: 0.7724 - loss: 0.4871 - val\_accuracy: 0.7375 - val\_loss: 0.5077  
Epoch 17/30


5/5  0s 25ms/step - accuracy: 0.7798 - loss: 0.4727 - val\_accuracy: 0.7708 - val\_loss: 0.4891  
Epoch 18/30


5/5  0s 23ms/step - accuracy: 0.7696 - loss: 0.4781 - val\_accuracy: 0.7563 - val\_loss: 0.5006  
Epoch 19/30


5/5  0s 26ms/step - accuracy: 0.7622 - loss: 0.4854 - val\_accuracy: 0.7167 - val\_loss: 0.5391  
Epoch 20/30


5/5  0s 26ms/step - accuracy: 0.7711 - loss: 0.4814 - val\_accuracy: 0.7563 - val\_loss: 0.5077  
Epoch 21/30


5/5  0s 25ms/step - accuracy: 0.7639 - loss: 0.4826 - val\_accuracy: 0.7375 - val\_loss: 0.5375  
Epoch 22/30


5/5  0s 26ms/step - accuracy: 0.7728 - loss: 0.4735 - val\_accuracy: 0.7667 - val\_loss: 0.4824  
Epoch 23/30

5/5  0s 40ms/step - accuracy: 0.7847 - loss: 0.4656 - val\_accuracy: 0.7708 - val\_loss: 0.4832  
Epoch 24/30


5/5  0s 25ms/step - accuracy: 0.7864 - loss: 0.4582 - val\_accuracy: 0.7604 - val\_loss: 0.5138  
Epoch 25/30

5/5  0s 25ms/step - accuracy: 0.7738 - loss: 0.4675 - val\_accuracy: 0.7750 - val\_loss: 0.4861  
Epoch 26/30


5/5  0s 26ms/step - accuracy: 0.7828 - loss: 0.4623 - val\_accuracy: 0.7312 - val\_loss: 0.5218  
Epoch 27/30

5/5  0s 26ms/step - accuracy: 0.7820 - loss: 0.4654 - val\_accuracy: 0.7708 - val\_loss: 0.4867


Epoch 28/30

5/5  0s 25ms/step - accuracy: 0.7847 - loss: 0.4393 - val\_accuracy: 0.7688 - val\_loss: 0.4834

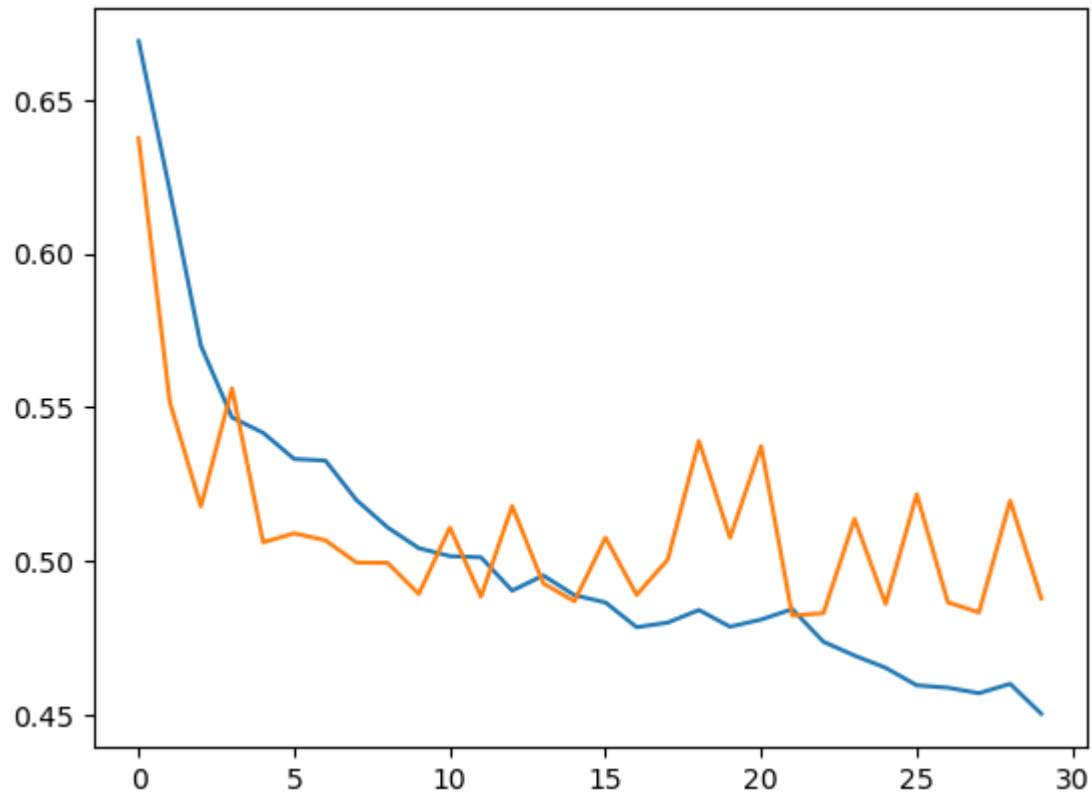
Epoch 29/30

5/5  0s 25ms/step - accuracy: 0.8039 - loss: 0.4496 - val\_accuracy: 0.7542 - val\_loss: 0.5197

Epoch 30/30

5/5  0s 25ms/step - accuracy: 0.7827 - loss: 0.4489 - val\_accuracy: 0.7604 - val\_loss: 0.4880

```
In [4]: history_df['loss'].plot();  
history_df['val_loss'].plot();
```



```
In [5]: X_train.shape[1]
```

```
Out[5]: 11
```

Remarquez comment la perte se stabilise au fil des époques. Lorsque la courbe de perte devient horizontale de cette manière, cela signifie que le modèle a appris tout ce qu'il pouvait et qu'il n'y a plus de raison de continuer avec des époques supplémentaires.

## GRADIENT et Descente de Gradient

Le gradient est un concept fondamental en optimisation, largement utilisé dans les réseaux de neurones et divers algorithmes d'apprentissage automatique. Il indique la direction et l'ampleur des changements d'une fonction (comme la fonction de perte) par rapport à ses paramètres. Dans les réseaux de neurones, les gradients sont essentiels pour la rétropropagation, où l'erreur est propagée en arrière pour ajuster les poids afin de minimiser la perte. Des optimisateurs comme SGD, Adam, et RMSprop utilisent ces gradients pour ajuster les poids itérativement.

Le gradient est également central dans des algorithmes comme le gradient boosting, où les modèles sont entraînés séquentiellement pour corriger les erreurs en fonction du gradient négatif de la perte. Des outils comme XGBoost et LightGBM exploitent cette technique. En régression linéaire et logistique, la descente de gradient est utilisée pour minimiser les erreurs, et dans les SVMs, elle aide à trouver l'hyperplan optimal pour séparer les classes.

En résumé, les gradients sont essentiels à la fois dans le deep learning et dans des méthodes d'apprentissage automatique traditionnelles pour optimiser les modèles en minimisant leurs fonctions de perte respectives.

## Interprétation des courbes d'apprentissage

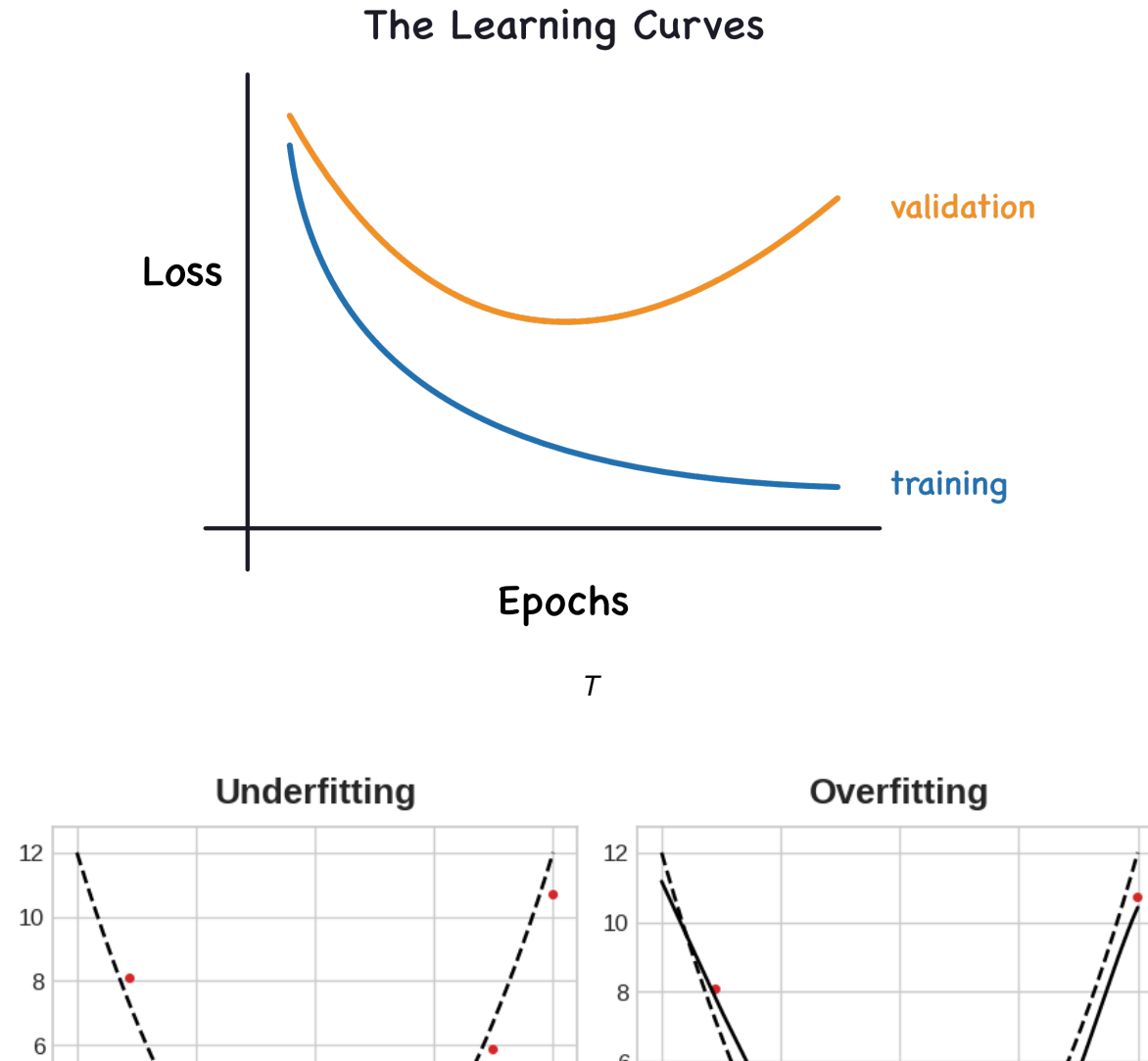
Les données d'entraînement contiennent à la fois du signal et du bruit. Le signal représente les informations généralisables qui permettent au modèle de faire des prédictions sur de nouvelles données, tandis que le bruit correspond à des fluctuations aléatoires ou des motifs non informatifs spécifiques aux données d'entraînement.

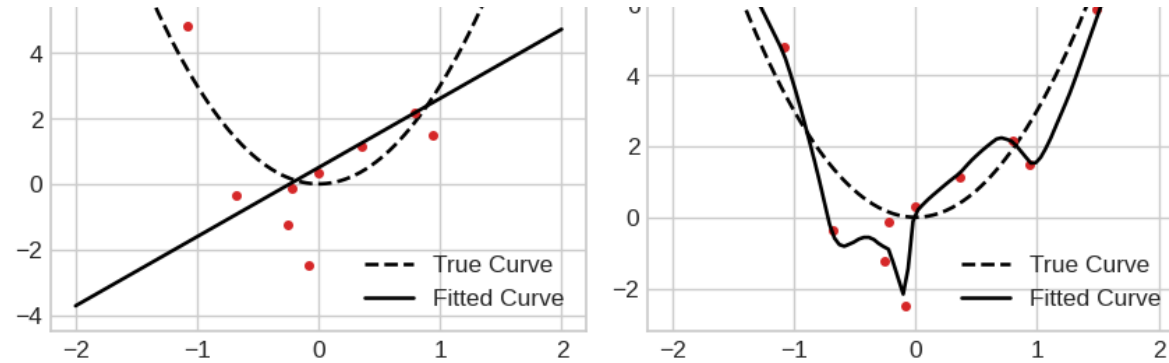
Pendant l'entraînement, le modèle ajuste ses poids pour minimiser la perte sur les données d'entraînement. Cependant, pour évaluer correctement les performances du modèle, on utilise des données de validation. En traçant la perte sur les données d'entraînement et de validation, on obtient des courbes d'apprentissage. Ces courbes montrent que :

- 1 La perte d'entraînement diminue lorsque le modèle apprend à partir du signal ou du bruit.
- 2 La perte de validation ne diminue que lorsque le modèle apprend du signal.

Lorsque le modèle commence à apprendre du bruit, un écart apparaît entre les deux courbes, indiquant un surajustement (overfitting). L'objectif est de trouver un compromis où le modèle apprend le maximum de signal avec un minimum de bruit. Si le modèle n'apprend pas assez de signal, il y a sous-ajustement (underfitting), et s'il apprend trop de bruit, il y a surajustement.

L'entraînement optimal consiste à trouver le bon équilibre entre le signal et le bruit pour éviter ces deux problèmes.





## Capacité

La capacité d'un modèle désigne la quantité et la complexité des motifs qu'il peut apprendre. Dans les réseaux de neurones, la capacité est principalement déterminée par le nombre de neurones et la manière dont ils sont connectés. Si un modèle sous-ajuste (n'apprend pas suffisamment), cela peut indiquer qu'il a une capacité insuffisante.

Pour augmenter la capacité, vous pouvez soit :

- 1 Élargir le réseau en augmentant le nombre de neurones dans chaque couche (rendre le réseau "plus large"),  
ce qui aide à apprendre des relations plus linéaires.
- 2 Approfondir le réseau en ajoutant plus de couches (rendre le réseau "plus profond"),  
ce qui permet d'apprendre des relations plus non linéaires.

Par exemple, un réseau plus large aurait plus d'unités dans chaque couche, tandis qu'un réseau plus profond ajouterait des couches supplémentaires.

Exemples ci-après en keras

```
In [6]: model = keras.Sequential([
        layers.Dense(16, activation='relu'),
        layers.Dense(1),
    ])
#modèle de base
```

```
In [7]: wider = keras.Sequential([
        layers.Dense(32, activation='relu'),
        layers.Dense(1),
    ])
#modèle plus large
```

```
In [8]: deeper = keras.Sequential([
        layers.Dense(16, activation='relu'),
        layers.Dense(16, activation='relu'),
        layers.Dense(1),
    ])
#modèle plus profond
```

## Arrêt anticipé (early stopping)

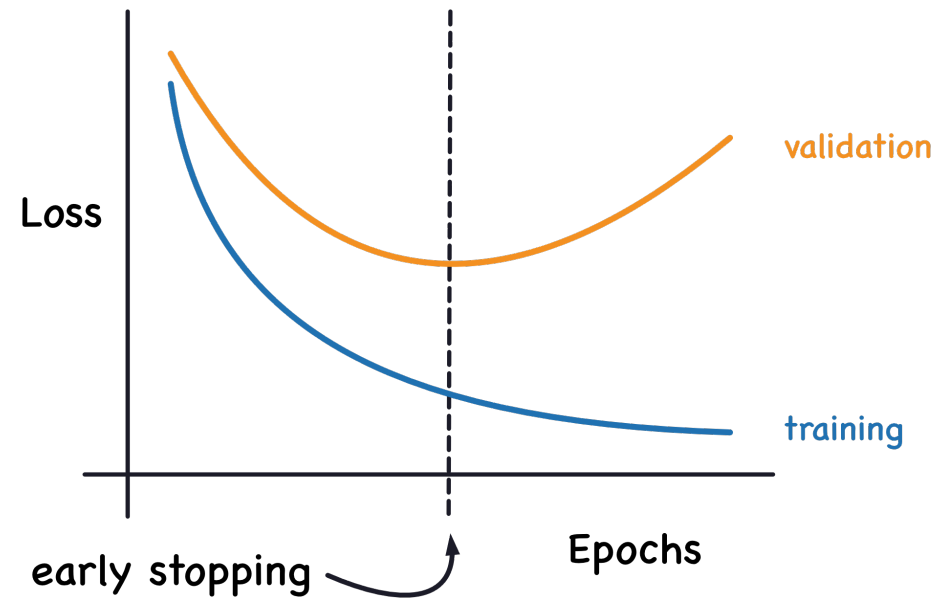
L'arrêt anticipé (early stopping) est une technique utilisée pour éviter le surajustement lors de l'entraînement d'un modèle. Lorsque la perte de validation commence à augmenter, cela indique que le modèle apprend trop de bruit plutôt que du signal. L'arrêt anticipé consiste à interrompre l'entraînement dès que la perte de validation cesse de diminuer, évitant ainsi de poursuivre l'apprentissage du bruit.

En pratique, cela signifie que l'entraînement est arrêté au point où la perte de validation est minimale, et les poids du modèle sont réinitialisés à ce moment optimal. Cela empêche le modèle de continuer à surajuster les données, tout en évitant d'arrêter l'entraînement trop tôt (ce qui causerait un sous-ajustement).

L'avantage de l'arrêt anticipé est qu'il équilibre le risque de sous-ajustement (interrompre trop tôt) et de surajustement (entraînement trop long). Dans Keras, vous pouvez ajouter un callback pour l'arrêt anticipé qui surveille la perte de validation après chaque époque et arrête l'entraînement au bon moment.

Underfitting

Overfitting



```
In [9]: from tensorflow.keras.callbacks import EarlyStopping

early_stopping = EarlyStopping(
    min_delta=0.001, # minimum amount of change to count as an improvement
    patience=5, # how many epochs to wait before stopping
    restore_best_weights=True,
)
```

```
In [10]: # Loading Necessary Libraries
import pandas as pd
from IPython.display import display
from sklearn.impute import SimpleImputer

# Load the dataset
red_wine = pd.read_csv('./winequality.csv')

# 'quality' is a binary classification where:
# "good" -> 1 and "bad" -> 0
# First, convert 'quality' to numeric representation

# We create a dictionary that maps the string values in the quality column to numbers
quality_mapping = {'good': 1, 'bad': 0}

# This replaces the string values in the quality column with the corresponding numeric values
red_wine['quality'] = red_wine['quality'].map(quality_mapping)

# Drop any rows where 'quality' is still NaN (because we can't train on rows without labels)
red_wine = red_wine.dropna(subset=['quality'])

# Now handle the feature columns (all except 'quality')
# Drops the quality column from the DataFrame because we don't want to apply transformations on the target
# Convert all other columns to numeric, forcing errors to NaN if any non-numeric values are found
# Converts all the remaining feature columns to numeric types.
# If there are any non-numeric values, they are converted to NaN
red_wine_numeric = red_wine.drop(columns=['quality']).apply(pd.to_numeric, errors='coerce')

# Impute missing values for the numeric columns (using mean imputation)
# This creates an imputer that fills missing values (NaN) in the DataFrame using the mean of each column
imputer = SimpleImputer(strategy='mean')

# This applies the imputer to the DataFrame. It computes the mean for each column and replaces the missing
# Converts the imputed data back into a DataFrame and ensures the column names remain the same.
red_wine_numeric_imputed = pd.DataFrame(imputer.fit_transform(red_wine_numeric), columns=red_wine_numeric.columns)

# Combine imputed numeric columns with the 'quality' column
# We need to do this because we had earlier separated quality from the feature columns.
# Ensures that the index of the quality column aligns with the imputed feature DataFrame.
red_wine_imputed = pd.concat([red_wine_numeric_imputed, red_wine['quality'].reset_index(drop=True)], axis=1)
```



```

# Create training and validation splits
df_train = red_wine_imputed.sample(frac=0.7, random_state=0)
df_valid = red_wine_imputed.drop(df_train.index)

# calculate the maximum and minimum values for each feature column (excluding quality) in the training set
max_ = df_train.drop(columns=['quality']).max(axis=0)
min_ = df_train.drop(columns=['quality']).min(axis=0)

# Feature scaling:
# Scale each feature to a range between 0 and 1 using min-max scaling.
# The formula  $(x - \min) / (\max - \min)$  is applied to each feature.
# This step ensures that all feature values are on a similar scale,
# which helps the neural network converge more effectively during training.
df_train.loc[:, df_train.columns != 'quality'] = (df_train.loc[:, df_train.columns != 'quality'] - min_) /
df_valid.loc[:, df_valid.columns != 'quality'] = (df_valid.loc[:, df_valid.columns != 'quality'] - min_) /

# Split features and target
X_train = df_train.drop('quality', axis=1)
X_valid = df_valid.drop('quality', axis=1)
y_train = df_train['quality']
y_valid = df_valid['quality']

# Display the result to ensure everything looks good
display(X_train.head(), y_train.head())

# Import TensorFlow and Keras
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.callbacks import EarlyStopping # Import EarlyStopping

# Define the model
model = keras.Sequential([
    layers.Input(shape=[X_train.shape[1]]), # Automatically infer input shape from the training data
    layers.Dense(512, activation='relu'),
    layers.Dense(512, activation='relu'),
    layers.Dense(512, activation='relu'),
    layers.Dense(1, activation='sigmoid'), # Since it's binary classification, use sigmoid activation
])

# Compile the model

```

```

model.compile(
    optimizer='adam',
    loss='binary_crossentropy', # Use binary crossentropy for binary classification
    metrics=['accuracy'] # We care about accuracy for classification
)

# Define EarlyStopping callback
# patience=5 means training will stop if validation loss does not improve for 5 consecutive epochs.
# restore_best_weights=True will reset the model to the epoch with the best validation loss.
early_stopping = EarlyStopping(
    monitor='val_loss', # Monitor validation loss
    min_delta=0.001,
    patience=20, # Stop training if val_loss doesn't improve for 5 epochs
    restore_best_weights=True # Restore model to the best weights after early stopping
)

# Train the model
history = model.fit(
    X_train, y_train,
    validation_data=(X_valid, y_valid),
    batch_size=256,
    epochs=200,
    callbacks=[early_stopping], # Include the early stopping callback
    # verbose=0, # turn off training log
)

# convert the training history to a dataframe
history_df = pd.DataFrame(history.history)
# use Pandas native plot method
#history_df['loss'].plot();
#history_df['val_loss'].plot();

```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol
<b>1109</b>	0.548673	0.239726	0.544304	0.092308	0.237435	0.366197	0.212014	0.619193	0.291262	0.260606	0.369231
<b>1032</b>	0.309735	0.479452	0.000000	0.246154	0.105719	0.056338	0.028269	0.645088	0.475728	0.121212	0.184615
<b>1002</b>	0.398230	0.116438	0.417722	0.088462	0.050260	0.169014	0.074205	0.387662	0.378641	0.309091	0.507692
<b>487</b>	0.495575	0.359589	0.455696	0.069231	0.032929	0.056338	0.028269	0.619193	0.291262	0.054545	0.246154
<b>979</b>	0.672566	0.226027	0.620253	0.038462	0.071057	0.028169	0.000000	0.520183	0.252427	0.181818	0.307692

1109 1














1032 0


1002 1


487 1


979 0


Name: quality, dtype: int64


Epoch 1/200  
5/5  1s 61ms/step - accuracy: 0.5674 - loss: 0.6819 - val\_accuracy: 0.7271 - val\_loss: 0.6246  
Epoch 2/200  
5/5  0s 25ms/step - accuracy: 0.6849 - loss: 0.6209 - val\_accuracy: 0.7521 - val\_loss: 0.5445  
Epoch 3/200  
5/5  0s 26ms/step - accuracy: 0.7394 - loss: 0.5522 - val\_accuracy: 0.7542 - val\_loss: 0.5101  
Epoch 4/200  
5/5  0s 42ms/step - accuracy: 0.7359 - loss: 0.5392 - val\_accuracy: 0.7646 - val\_loss: 0.5034  
Epoch 5/200  
5/5  0s 22ms/step - accuracy: 0.7441 - loss: 0.5311 - val\_accuracy: 0.7167 - val\_loss: 0.5367  
Epoch 6/200  
5/5  0s 24ms/step - accuracy: 0.7469 - loss: 0.5326 - val\_accuracy: 0.7458 - val\_loss: 0.5153  
Epoch 7/200  
5/5  0s 24ms/step - accuracy: 0.7449 - loss: 0.5449 - val\_accuracy: 0.7229 - val\_loss: 0.5216  
Epoch 8/200  
5/5  0s 29ms/step - accuracy: 0.7474 - loss: 0.5192 - val\_accuracy: 0.7646 - val\_loss: 0.4973  
Epoch 9/200  
5/5  0s 33ms/step - accuracy: 0.7515 - loss: 0.5115 - val\_accuracy: 0.7437 - val\_loss: 0.5017  
Epoch 10/200  
5/5  0s 25ms/step - accuracy: 0.7524 - loss: 0.4962 - val\_accuracy: 0.7625 - val\_loss: 0.4932  
Epoch 11/200  
5/5  0s 26ms/step - accuracy: 0.7507 - loss: 0.5130 - val\_accuracy: 0.7604 - val\_loss: 0.4894  
Epoch 12/200  
5/5  0s 25ms/step - accuracy: 0.7553 - loss: 0.5112 - val\_accuracy: 0.7458 - val\_loss: 0.4985  
Epoch 13/200  
5/5  0s 26ms/step - accuracy: 0.7659 - loss: 0.4936 - val\_accuracy: 0.7667 - val\_loss: 0.4902  
Epoch 14/200


5/5  0s 27ms/step - accuracy: 0.7628 - loss: 0.4949 - val\_accuracy: 0.7542 - val\_loss: 0.5043  
Epoch 15/200


5/5  0s 26ms/step - accuracy: 0.7684 - loss: 0.4897 - val\_accuracy: 0.7437 - val\_loss: 0.5032  
Epoch 16/200


5/5  0s 26ms/step - accuracy: 0.7520 - loss: 0.4986 - val\_accuracy: 0.7667 - val\_loss: 0.4924  
Epoch 17/200


5/5  0s 26ms/step - accuracy: 0.7760 - loss: 0.4838 - val\_accuracy: 0.7375 - val\_loss: 0.5174  
Epoch 18/200


5/5  0s 25ms/step - accuracy: 0.7620 - loss: 0.4814 - val\_accuracy: 0.7688 - val\_loss: 0.4873  
Epoch 19/200


5/5  0s 26ms/step - accuracy: 0.7723 - loss: 0.4767 - val\_accuracy: 0.7542 - val\_loss: 0.5061  
Epoch 20/200


5/5  0s 24ms/step - accuracy: 0.7704 - loss: 0.4778 - val\_accuracy: 0.7708 - val\_loss: 0.4884  
Epoch 21/200


5/5  0s 27ms/step - accuracy: 0.7873 - loss: 0.4580 - val\_accuracy: 0.7708 - val\_loss: 0.4890  
Epoch 22/200


5/5  0s 25ms/step - accuracy: 0.7782 - loss: 0.4742 - val\_accuracy: 0.7708 - val\_loss: 0.5000  
Epoch 23/200

5/5  0s 24ms/step - accuracy: 0.7962 - loss: 0.4463 - val\_accuracy: 0.7604 - val\_loss: 0.5103  
Epoch 24/200


5/5  0s 26ms/step - accuracy: 0.7913 - loss: 0.4622 - val\_accuracy: 0.7542 - val\_loss: 0.5221  
Epoch 25/200

5/5  0s 25ms/step - accuracy: 0.7834 - loss: 0.4612 - val\_accuracy: 0.7729 - val\_loss: 0.4967  
Epoch 26/200


5/5  0s 26ms/step - accuracy: 0.7903 - loss: 0.4562 - val\_accuracy: 0.7729 - val\_loss: 0.4969  
Epoch 27/200

5/5  0s 40ms/step - accuracy: 0.8116 - loss: 0.4330 - val\_accuracy: 0.7333 - val\_loss: 0.5455


Epoch 28/200

5/5  0s 28ms/step - accuracy: 0.7820 - loss: 0.4655 - val\_accuracy: 0.7729 - val\_loss: 0.5003


Epoch 29/200

5/5  0s 32ms/step - accuracy: 0.7936 - loss: 0.4388 - val\_accuracy: 0.7542 - val\_loss: 0.5119


Epoch 30/200

5/5  0s 29ms/step - accuracy: 0.7914 - loss: 0.4445 - val\_accuracy: 0.7667 - val\_loss: 0.5072


Epoch 31/200

5/5  0s 33ms/step - accuracy: 0.7943 - loss: 0.4328 - val\_accuracy: 0.7375 - val\_loss: 0.5684


Epoch 32/200

5/5  0s 31ms/step - accuracy: 0.7915 - loss: 0.4459 - val\_accuracy: 0.7792 - val\_loss: 0.5089


Epoch 33/200

5/5  0s 35ms/step - accuracy: 0.8029 - loss: 0.4344 - val\_accuracy: 0.7500 - val\_loss: 0.5321


Epoch 34/200

5/5  0s 28ms/step - accuracy: 0.7988 - loss: 0.4579 - val\_accuracy: 0.7250 - val\_loss: 0.5818


Epoch 35/200

5/5  0s 30ms/step - accuracy: 0.7908 - loss: 0.4430 - val\_accuracy: 0.7729 - val\_loss: 0.5095


Epoch 36/200

5/5  0s 29ms/step - accuracy: 0.8088 - loss: 0.4283 - val\_accuracy: 0.7771 - val\_loss: 0.5109

Epoch 37/200

5/5  0s 36ms/step - accuracy: 0.8073 - loss: 0.4275 - val\_accuracy: 0.7563 - val\_loss: 0.5374

Epoch 38/200

5/5  0s 33ms/step - accuracy: 0.8163 - loss: 0.4172 - val\_accuracy: 0.7354 - val\_loss: 0.5359

```
In [11]: history_df = pd.DataFrame(history.history)
history_df.loc[:, ['loss', 'val_loss']].plot();
print("Minimum validation loss: {}".format(history_df['val_loss'].min()))
```

Minimum validation loss: 0.48733803629875183

