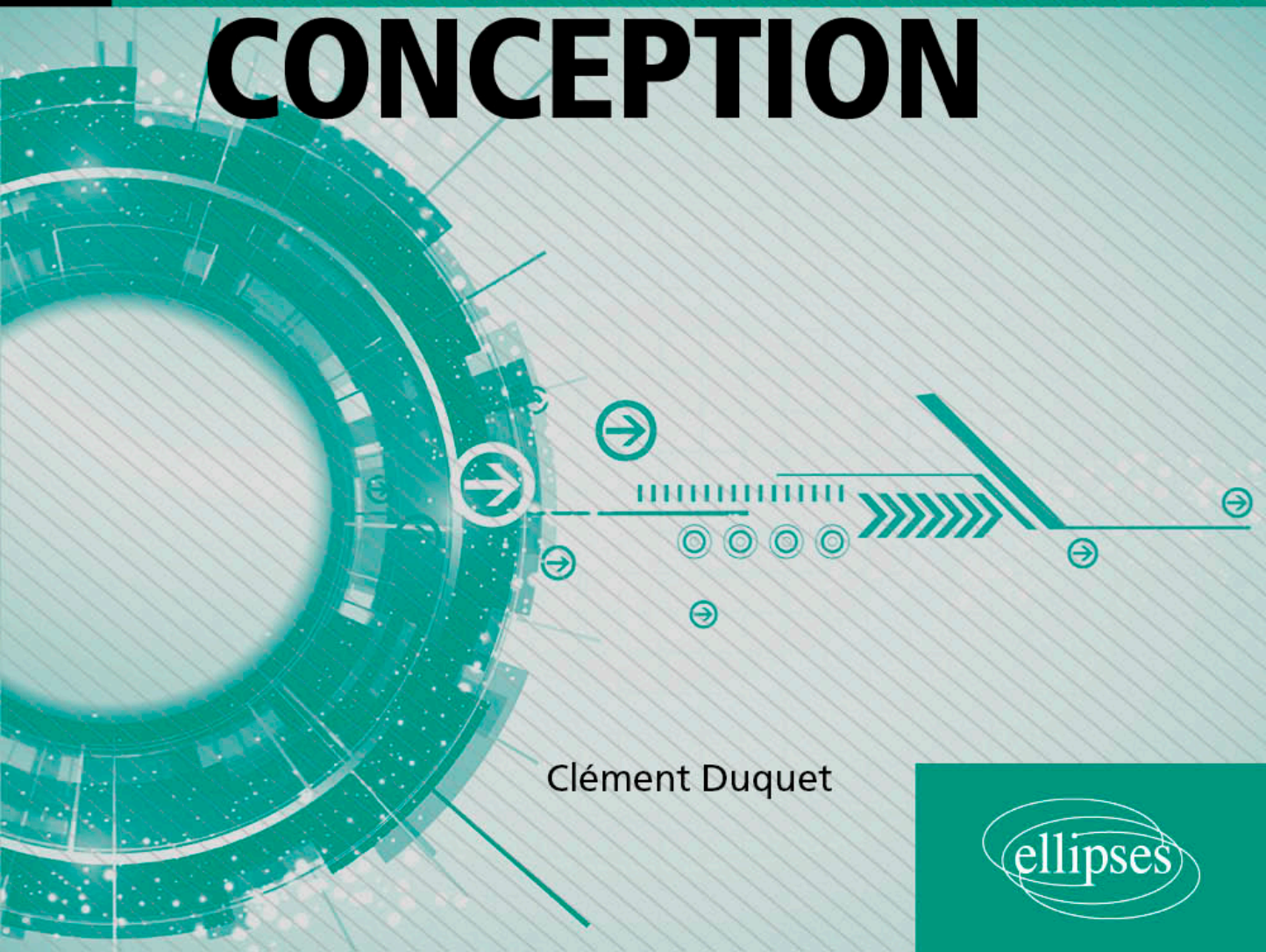
Formations & Techniques

# LES OUTILS DE L'INDUSTRIE 4.0 AU SERVICE DE L'ÉCO-CONCEPTION



Clément Duquet

ellipses

Chapitre 7

# L'optimisation topologique

## 7.1. L'intérêt de l'optimisation topologique

L'optimisation topologique est une discipline de l'ingénierie visant à minimiser la masse d'un produit ou système tout en conservant les propriétés mécaniques initiales ou celles définies par le cahier des charges. Le principe est donc d'éliminer les parties du volume total ne participant pas à l'efficacité de la pièce.

L'optimisation topologique représente une évolution majeure dans le processus de conception de produits, permettant aux ingénieurs de repenser intelligemment la manière de répartir les charges dans une structure. On peut ainsi créer des produits plus légers pour des performances égales ou supérieures.

On comprend alors vite que cette technique peut avoir un grand intérêt pour l'éco-conception en réduisant l'impact environnemental sur plusieurs des phases de son cycle de vie :

- Sur la phase d'extraction/fabrication de la matière première : si la masse est réduite, la quantité de matière à extraire/produire en est réduite d'autant.
- Sur la phase d'usage : un produit plus léger a donc une inertie plus faible et demande moins d'énergie pour le déplacer.
- Fin de vie : si le produit a une masse plus faible, la quantité de matière à recycler est plus faible.

## 7.2. Concepts théoriques fondamentaux

Les logiciels proposant des optimisations topologiques sont des logiciels de calcul par éléments finis. Le produit étudié sera donc discrétisé en petits éléments (maillage), comme cela a été décrit dans la partie *Simulations sur jumeaux numérique* du chapitre 6. On rappelle donc que la qualité du résultat dépendra, entre autres, de la qualité du maillage choisi.

La formulation des objectifs et des contraintes est au cœur de tout problème d'optimisation. Dans le contexte de l'optimisation topologique, les fonctions objectifs visent à minimiser la masse en maintenant certaines caractéristiques, telles que la rigidité ou la résistance (contraintes internes inférieures à une certaine valeur). La variable de conception est alors ici le volume de la pièce (on pourra retirer ou non chaque maille du maillage total, d'où l'intérêt de le définir à la bonne taille).

L'optimisation des variables est enfin basée sur des modèles mathématiques tels que la méthode SIMP (*Solid Isotropic Material with Penalization*), qui régularise les solutions en ajustant la densité des matériaux, ou encore la méthode *level-set* qui permet de représenter l'évolution de l'interface entre les zones pleines et vides. On effectuera ensuite différentes itérations de calcul à l'aide de méthodes telles que : la méthode du gradient, la méthode évolutionnaire ou encore la méthode de recherche directe. Ainsi, selon la méthode utilisée, les résultats obtenus peuvent différer. Il convient donc de toujours garder un regard critique sur les résultats obtenus.

## 7.3. Mise en œuvre pratique

De nombreux logiciel de FEM proposent des simulations d'optimisation topologique. On pourra alors citer les logiciels tel que Ansys, Altair Inspire, SolidWorks, Fusion360 ou encore CREO.

Pour l'optimisation de matière, on peut procéder de deux manières différentes. Soit partir d'une pièce surdimensionnée existante qui sera évidée pour être allégée, soit partir d'un gros bloc de matière (dont la seule contrainte est

l'encombrement total) qui sera évidé ensuite. Cette deuxième solution est à privilégier dans le cas où on conçoit un système non existant en partant de zéro afin d'être certain d'avoir la matière là où elle est la plus efficace.

Après avoir réalisé une modélisation CAO de la pièce à optimiser, on peut de manière générale décrire la méthode à suivre telle que :

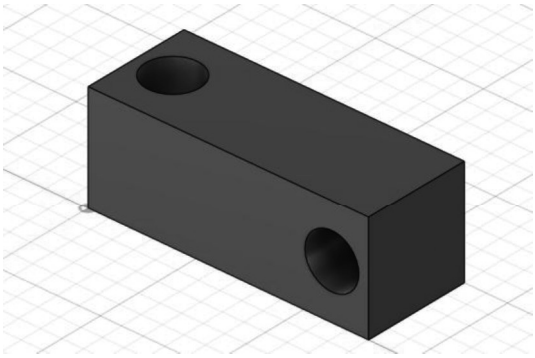
- Définir un maillage pour notre pièce qui soit assez fin pour obtenir un résultat fiable.
- On doit ensuite définir des objectifs de l'étude. On peut alors choisir différents objectifs tels que :
    - Réduire la masse de X% tout en maximisant la rigidité.
    - Minimiser la masse pour une limite de déplacement imposée.
    - Minimiser la masse pour une limite de fréquence imposée.
    - Minimiser la masse pour une limite contrainte/coefficient de sécurité imposée.
- Il faut ensuite définir les conditions aux limites du modèle, c'est-à-dire toutes les actions mécaniques qui s'exercent sur le modèle. Ces dernières peuvent être des forces, des pressions, un poids ou encore une charge. Il faudra également définir toutes les liaisons qui vont s'appliquer sur la pièce pour empêcher le déplacement de celle-ci. Une attention particulière devra être apportée au type de liaison choisi ou le nombre de mobilités bloquées. En effet, un encastrement imposera une déformation nulle en un point, ce qui ne serait par exemple pas le cas avec une liaison ponctuelle. Il convient donc de choisir une modélisation représentant au mieux le comportement réel.
- On pourra par la suite définir des zones à conserver. Ces zones seront l'ensemble des surfaces fonctionnelles pour lesquelles il ne faudra pas enlever de matière.
- On pourra également définir des symétries pour notre pièce, ce qui simplifiera les calculs et nous permettra de n'imposer des chargements que d'un côté de la pièce.
- Certains logiciels permettent également de choisir le moyen de fabrication qui sera utilisé pour la production de la pièce. Le logiciel prendra donc en compte les contraintes de fabrication telles que les conditions de démoulage pour les pièces de fonderie par exemple.
- Le logiciel peut à présent nous proposer un résultat. Il convient alors d'effectuer une analyse du résultat. Certaines modifications seront alors nécessaires et des simulations de comportement telles que présentées

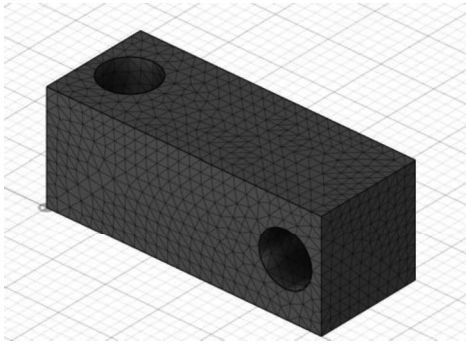
dans le chapitre *6* pourront être réalisées afin de valider l'ensemble des performances de la nouvelle pièce.

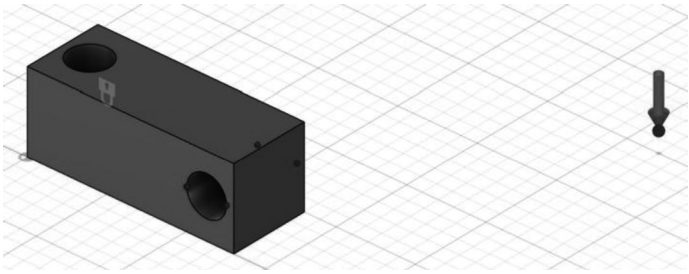
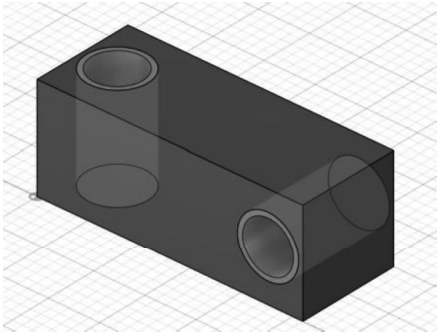
## 7.4. Exemple d'utilisation de l'optimisation topologique

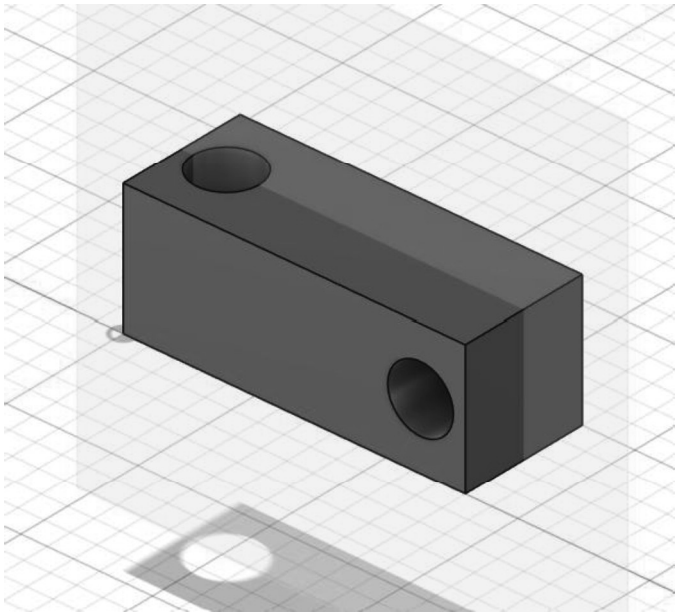
On peut alors appliquer la méthode précédente à l'exemple suivant à l'aide du logiciel Fusion 360 :

Optimisation topologique d'une potence de vélo :

| | |
|---|---|
| **Modélisation de pièce :**<br><br>Pour la modélisation de la pièce, on choisit de réaliser un gros bloc de matière afin de garantir que la matière restant à la fin sera à la place la plus optimale. On réalise également les surfaces fonctionnelles qui ne devront pas être modifiées à la fin de l'étude. On définit enfin le matériau qui sera de l'aluminium. |  |

| | |
|---|---|
| **Maillage de la pièce :**<br><br>Pour le maillage de la pièce on choisit un maillage moyennement fin dans le but d'avoir un résultat plutôt fiable avec un temps de simulation raisonnable. |  |
| **Définition de l'objectif :**<br><br>L'objectif est ici de réduire la masse de 70% tout en maximisant la rigidité. | |

| | |
|---|---|
| **Définition des conditions aux limites :**<br><br>La potence étant encastrée sur le pivot de direction nous allons fixer la surface de l'alésage.<br><br>L'utilisateur s'appuie quant à lui à l'extrémité de son guidon. Nous allons donc modéliser le cas extrême où tout son poids serait placé à une extrémité. On applique donc une force déportée de 1200 N. |  |
| **Définition des zones à conserver :**<br><br>Les surfaces fonctionnelles sont les surfaces de mise en position avec le guidon et le pivot de direction. On choisit donc de les définir comme zones à conserver. |  |

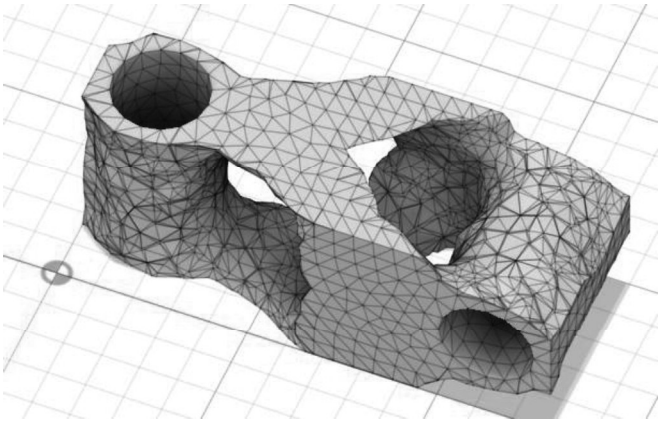
| | |
|---|---|
| **Définition des symétries :**<br><br>On a ici un plan de symétrie qu'on définit normal à l'axe du guidon. |  |

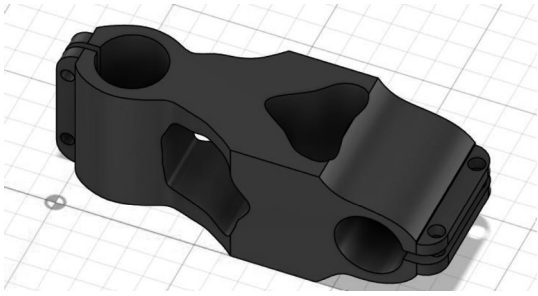
**Analyse des résultats :**

On obtient alors une proposition de conception pour notre potence.

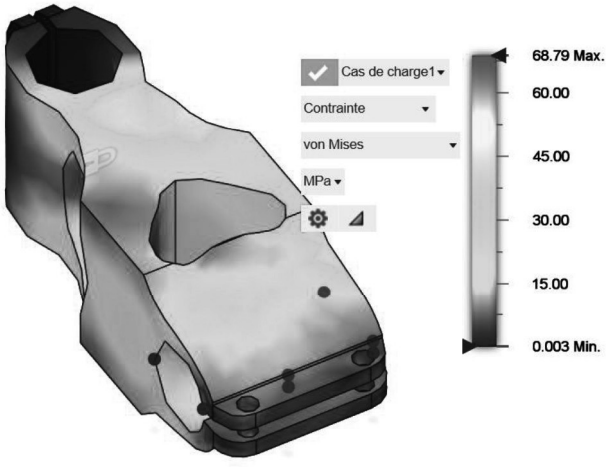
On peut donc modifier le modèle initial en nous inspirant de la solution proposée et en s'assurant de réaliser une pièce usinable en 2 phases avec la fraiseuse 3 axes qu'on aurait à disposition.

On sait que ce type de produit est soumis à des contraintes en fatigue, on va donc avec une simulation en statique s'assurer d'avoir un coefficient de sécurité suffisant, soit supérieur à 5 (cf. simulation statique dans le chapitre 6)









Chapitre 8

# L'intelligence artificielle

L'utilisation de l'intelligence artificielle, IA, (ou *artificial intelligence*, *AI* en anglais) dans l'éco-conception présente plusieurs avantages qui contribuent à la création de produits et de services plus durables. Cette technologie peut par exemple être utilisée pour traiter des données (*big data*) afin d'optimiser ou automatiser un processus en temps réel ou en prévision. Cette technologie peut également vous permettre de réaliser des innovations dans la conception avec par exemple des conceptions génératives.

Il existe différents types d'IA qui peuvent fonctionner sur des technologies différentes. Selon votre problème, vous pourrez trouver une IA spécifiquement correspondante pour vous aider. Ce domaine est en constante évolution et les progrès en feront un outil de plus en plus efficace et donc indispensable à terme.

Il n'est pas nécessairement obligatoire de savoir coder une IA pour s'en servir mais il faut néanmoins en connaître les grands principes pour savoir interpréter les résultats obtenus.

## 8.1. Les grands principes de l'intelligence artificielle

Le terme « intelligence artificielle », créé par John McCarthy, est souvent abrégé IA (ou *AI* en anglais). Il est défini par l'un de ses créateurs, Marvin Lee Minsky, comme « la construction de programmes informatiques qui s'adonnent à des tâches qui sont, pour l'instant, accomplies de façon plus satisfaisante par des êtres humains car elles demandent des processus mentaux de haut niveau tels que : l'apprentissage perceptuel, l'organisation de la mémoire et le raisonnement critique. »

L'intelligence artificielle englobe le *machine learning* qui lui-même englobe le *deep learning* :

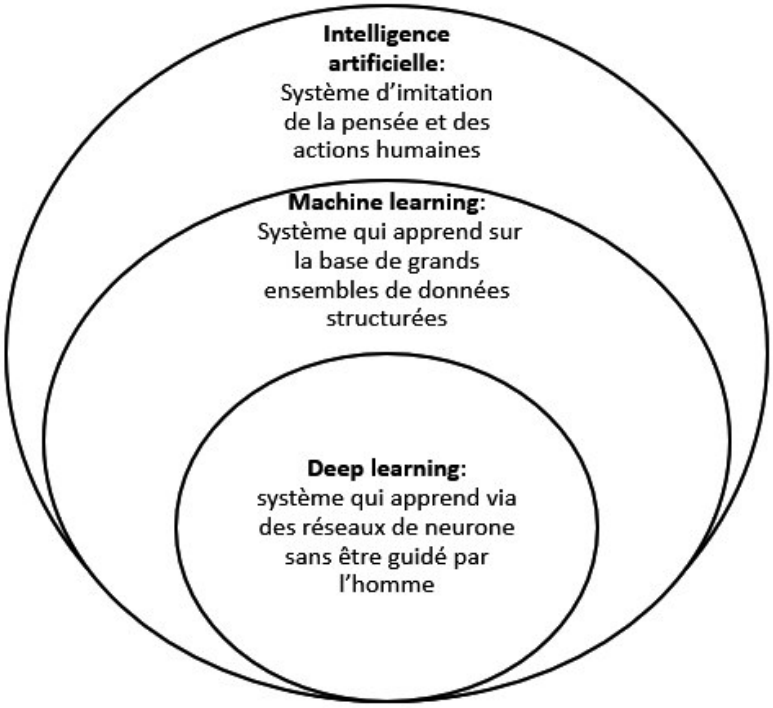



*Figure 46 : ensembles de l'IA*

## 8.1.1. Le machine learning

Pour le *machine learning*, on programme l'algorithme pour qu'il apprenne un modèle à partir de données. L'exemple le plus simple est la régression linéaire où on trouve un modèle de la forme $f(x) = a.x + b$ à partir de données observées/mesurées de sorte à minimiser les écarts entre le calcul et la mesure.

Pour chaque modèle un algorithme d'optimisation est à privilégier :

• pour les modèles linéaires : la descente de gradients

• pour les arbres de décisions : l'algorithme CART

• pour les supports *Vectors machines* : la marge maximum

• …

Un modèle de *machine learning* doit être entraîné, il existe différents types d'apprentissage qui permettent de réaliser cela :

- L'apprentissage supervisé : dans le jeu de données, les informations sont classées par catégorie (on dit qu'elles sont « labélisées »). Durant le processus d'apprentissage, les données labellisées sont présentées à l'algorithme, qui est supposé à la fin de l'apprentissage pouvoir prédire le label d'une nouvelle donnée. Par exemple, trouver le nom de l'animal sur la photo en connaissant le nom des autres photos :

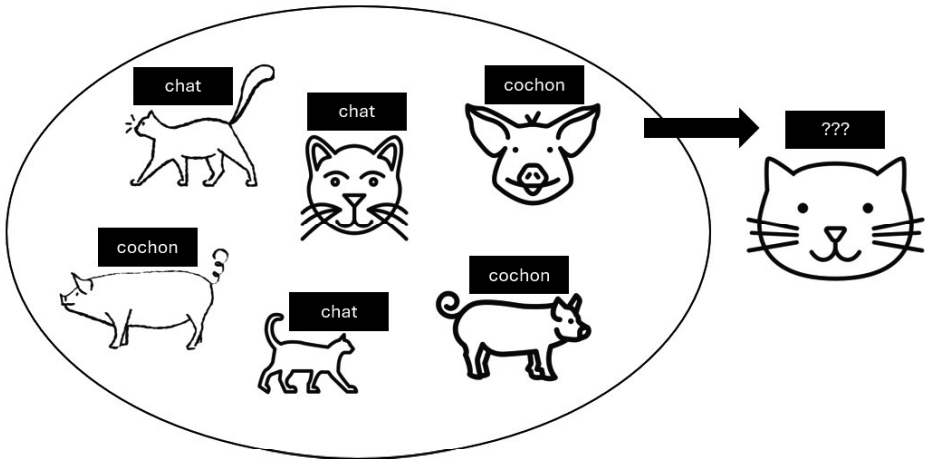



*Figure 47: apprentissage supervisé*

Remarque : on appelle phase d'inférence la phase d'utilisation de l'algorithme préalablement entraîné.

- L'apprentissage non supervisé : contrairement à l'apprentissage supervisé, ici les données ne sont pas labellisées. A la fin du processus, les groupes de données seront créées. Un humain pourra intervenir pour les labelliser. Par exemple, trier les images de fruits :

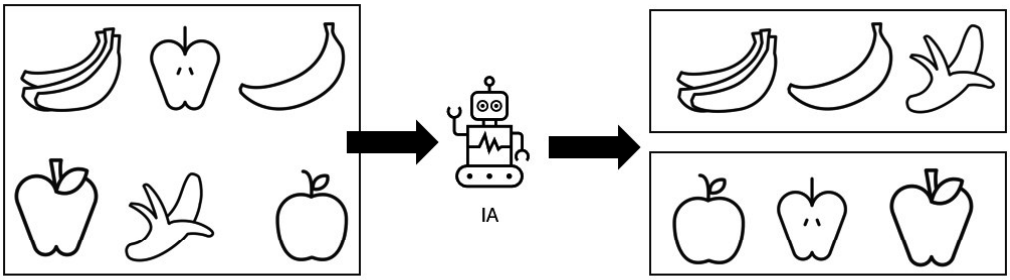



*Figure 48 : apprentissage non supervisé*

- Apprentissage par renforcement : dans ce type d'apprentissage automatique, le programme obtient des données de son environnement et essaie de trouver une solution pour résoudre un problème en effectuant des tests. Pour quantifier le degré de réussite des tests, l'algorithme fonctionne avec un système de récompense qui peut utiliser l'équation de Bellman qui affectera un coefficient atténuateur. Par exemple, pour sortir d'un labyrinthe, l'algorithme va tester des déplacements aléatoires jusqu'à trouver la sortie. Il va affecter à chaque case un coefficient atténuateur dont la valeur diminue proportionnellement à l'éloignement de la sortie. Il sera alors possible, en analysant les coefficients, de déduire le chemin le plus rapide.
- L'algorithme des k plus proches voisins : on appelle algorithme des k plus proches voisins (*kNN* pour *k nearest neighbors*) l'algorithme consistant à labelliser une nouvelle donnée en fonction des labels des k points du jeu de données d'entraînement dont elle est la plus proche :
    - Pour un problème classique, on prend le label majoritaire :

$$f(x) = \arg\left(\max\left(\sum \delta(y)\right)\right)$$

    - Pour une régression linéaire, on prend le label moyen :

$$f(x) = \frac{1}{k}\sum y$$

Par exemple, pour classer les formes :

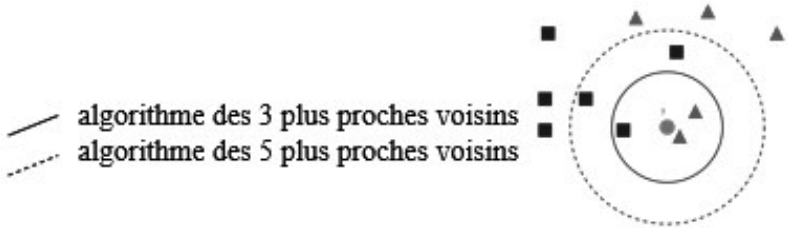



*Figure 49 : algorithme des k plus proches voisins*

Dans Python, il existe une bibliothèque dans laquelle on trouve cette fonction toute faite : from sklearn.neighbors import KNeighborsClassifier.

## 8.1.2. Le deep learning

Le *deep learning* (ou apprentissage profond) est un sous-domaine qui s'appuie sur les réseaux de neurones et l'apprentissage expérimental. Le fonctionnement est donc inspiré du fonctionnement du cerveau humain qui est lui composé de milliers de neurones qui reçoivent l'information, la traite et selon l'importance, avertissent les neurones voisins au moyen d'un potentiel d'action.

Ainsi, un réseau de neurones algorithmique est composé d'une succession de couches dont chacune prend ses entrées sur les sorties de la précédente. Chaque couche (i) est composée de $N_i$ neurones, prenant leurs entrées sur les $N_{i-1}$ neurones de la couche précédente. À chaque synapse est associé un poids synaptique, de sorte que les $N_{i-1}$ sont multipliés par ce poids, puis additionnés par les neurones de niveau i, ce qui est équivalent à multiplier le vecteur d'entrée par une matrice de transformation. On ajoute ensuite au résultat obtenu un biais pour corriger la valeur finale. Cette valeur finale passe par une fonction d'activation qui dépend du modèle souhaité (les plus utilisées sont la fonction sigmoïde et la fonction de Heaviside).

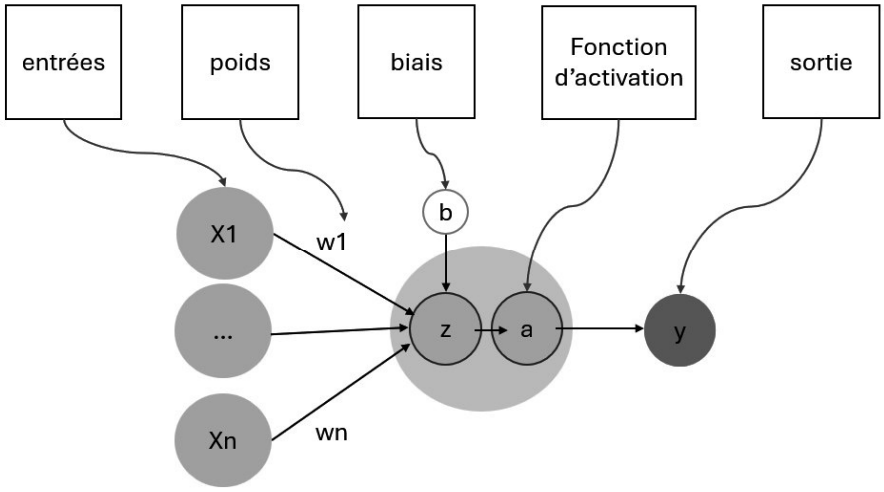



*Figure 50 : neurone de deep learning*

Mettre l'une derrière l'autre les différentes couches d'un réseau de neurones reviendrait à mettre en cascade plusieurs matrices de transformation et pourrait se ramener à une seule matrice, produit des autres, s'il n'y avait pas à chaque couche, la fonction de sortie qui introduit une non-linéarité à chaque étape.

Une fois le réseau de neurone créé, il faut l'entraîner avec des données labélisées afin de faire évoluer les valeurs de poids synaptique et des biais. Cet entraînement pourra se faire à l'aide de la méthode de descente de gradient, de l'algorithme de CART ou encore de la marge maximum. Ces méthodes d'entraînement ne seront pas développées ici. On doit noter néanmoins que la qualité de l'entraînement aura une influence directe sur la fiabilité de la sortie.

## 8.2. Exemple d'utilisation de l'intelligence artificielle

L'utilisation de l'outil intelligence artificielle peut se faire avec différents supports. Pour un besoin particulièrement spécifique ou très simple, le plus

adapté est le codage d'un algorithme d'intelligence artificielle. Pour un besoin plus classique, il est plus pertinent d'utiliser les outils déjà créés et destinés aux besoins.

### 8.2.1. Codage d'un algorithme d'IA

Le but de cet exemple n'est pas de vous apprendre à coder dans un langage de programmation. Si telle est votre ambition, référez-vous aux multiples cours que vous trouverez sur le web. Le but est ici simplement de vous montrer que si vous maîtriser un langage de programmation, il peut être rapide de réaliser un programme qui répondra à un besoin simple. En revanche si vous n'avez pas ces compétences ou si votre besoin est plus complexe, il est préférable de sous-traiter ce travail à un expert.

Pour cet exemple nous utiliserons le logiciel Python mais le travail et le même avec d'autres logiciels.

La problématique de l'exemple est la suivante : sur des voitures de rallyes, on suppose que les grosses écuries, pour ne pas prendre de risque, choisissent de remplacer les pneus avant à chaque spéciale pour éviter de prendre le départ avec une crevaison lente. Pour éviter un changement de pneus inutile, on souhaiterait connaître l'état des pneus en temps réel. On choisit alors d'équiper nos roues de capteurs de pression. Seulement lors de la course la température des pneus varie en fonction des virages, freinages, dérapages… On a alors choisi d'équiper nos roues de capteurs de température pour pouvoir coupler ces 2 données et on a fait des tests pour voir comment influaient ces paramètres. On a obtenu les résultats suivants :

| Etat | Crevé | Bon | Bon | Crevé | Crevé | Bon | Crevé | Bon | ? |
|---|---|---|---|---|---|---|---|---|---|
| Pression (en bar) | 2,55 | 2,54 | 2,58 | 2,00 | 2,30 | 2,60 | 2,41 | 2,45 | 2,3 |
| Température (en degré) | 50 | 30 | 35 | 30 | 40 | 52 | 50 | 40 | 49 |

On souhaite maintenant obtenir un programme qui pourrait nous donner l'état d'un pneu en fonction des 2 grandeurs mesurées et du tableau de données relevé précédemment.

Pour cela, nous utiliserons un réseau de neurones (*deep learning*) avec le fonctionnement présenté dans la partie précédente.

Voici le programme utilisé :

```
import numpy as np

X = np.array(([2.55,50], [2.54,30], [2.58,35], [2,30], [2.3,40], [2.6,52], [2.41,50], [2.45,50], [2.3,49]), dtype=float) # données d'entrée

y = np.array(([1],[0],[0],[1],[1],[0],[1],[0]), dtype=float) # données de sortie /  1 = crevé /  0 = bon

# Changement de l'échelle de nos valeurs pour être entre 0 et 1

X = X/np.amax(X, axis=0) # On divise chaque entrée par la valeur max des entrées
```

```
xPrediction = np.array([2.3,49]) # Valeur qu'on veut trouver


#Notre classe de réseau neuronal
class Neural_Network(object):
  def __init__(self):


  #Nos paramètres
    self.inputSize = 2 # Nombre de neurones d'entrée
    self.outputSize = 1 # Nombre de neurones de sortie
    self.hiddenSize = 3 # Nombre de neurones cachés


  #Nos poids
    self.W1 = np.random.randn(self.inputSize, self.hiddenSize) # (2x3) Matrice de poids entre les neurones d'entrée et cachés
    self.W2 = np.random.randn(self.hiddenSize, self.outputSize) # (3x1) Matrice de poids entre les neurones cachés et sortis




  #Fonction de propagation
  def forward(self, X):
```

```
    self.z = np.dot(X, self.W1) # Multiplication matricielle entre les valeurs d'entrée et les poids W1

    self.z2 = self.sigmoid(self.z) # Application de la fonction d'activation (Sigmoïde)

    self.z3 = np.dot(self.z2, self.W2) # Multiplication matricielle entre les valeurs cachées et les poids W2

    o = self.sigmoid(self.z3) # Application de la fonction d'activation, et obtention de notre valeur de sortie finale

    return o


  # Fonction d'activation

  def sigmoid(self, s):

    return 1/(1+np.exp(-s))


  # Dérivée de la fonction d'activation (méthode du gradient)

  def sigmoidPrime(self, s):

    return s * (1 - s)


  #Fonction de rétropropagation

  def backward(self, X, y, o):


    self.o_error = y - o # Calcul de l'erreur
```

```
    self.o_delta = self.o_error*self.sigmoidPrime(o) # Application de la dérivée de la sigmoïde à cette erreur



    self.z2_error = self.o_delta.dot(self.W2.T) # Calcul de l'erreur de nos neurones cachés

    self.z2_delta = self.z2_error*self.sigmoidPrime(self.z2) # Application de la dérivée de la sigmoïde à cette erreur



    self.W1 += X.T.dot(self.z2_delta) # On ajuste nos poids W1

    self.W2 += self.z2.T.dot(self.o_delta) # On ajuste nos poids W2



  #Fonction d'entraînement

  def train(self, X, y):



    o = self.forward(X)

    self.backward(X, y, o)



  #Fonction de prédiction

  def predict(self):



    print("Données prédites après entraînement: ")

    print("Entrée : \n" + str(xPrediction))
```

```
    print("Sortie : \n" + str(self.forward(xPrediction)))


    if(self.forward(xPrediction) < 0.5):
        print("Le pneu est crevé ! \n")
    else:
        print("Le pneu est bon ! \n")



NN = Neural_Network()


for i in range(1000): #Choisissez un nombre d'itération
    print("# " + str(i) + "\n")
    print("Valeurs d'entrées: \n" + str(X))
    print("Sortie actuelle: \n" + str(y))
    print("Sortie prédite: \n" + str(np.matrix.round(NN.forward(X),2)))
    print("\n")
    NN.train(X,y)


NN.predict()
```

Dans cet algorithme, on retrouve les étapes suivantes :

- Définition de nos données d'entrée (données d'entraînement)
- Définition du nombre de neurones d'entrée et de sortie et du nombre de couches (2 entrées : pression et température, 1 sortie : crevé ou non, 1 couche intermédiaire avec 3 neurones).
- On définit ensuite des poids aléatoires pour nos synapses. Les valeurs de ces poids seront ensuite ajustées avec l'entraînement de l'algorithme.
- On choisit la fonction sigmoïde comme fonction d'activation car elle a l'avantage de renvoyer des valeurs dans l'intervalle [0 ;1].
- Pour calculer la marge d'erreur, on utilise la méthode de descente du gradient. Sans rentrer dans les détails mathématiques, c'est la méthode la plus adaptée pour les modèles linéaires comme le nôtre.
- On effectue alors l'entraînement de l'algorithme en modifiant le poids de nos synapses. L'objectif est ici d'essayer d'approcher la marge d'erreur de 0.
- On choisit ici de faire 1000 itérations pour entraîner notre algorithme. Il est à noter que plus le nombre d'itérations est grand, plus le résultat doit être fiable, mais également que plus le temps d'entraînement sera long. A partir d'un certain nombre d'itérations, les paramètres ne varient presque plus et il est inutile de continuer l'entraînement. L'algorithme peut donc être programmé en ce sens.

Notre programme est donc prêt à renvoyer l'état d'un pneu en fonction de sa pression et de sa température. On notera cependant que pour plus de fiabilité, il aurait été préférable d'ajouter plus de données d'entraînement.

Cet algorithme peut alors vous servir de base pour programmer le vôtre selon vos besoins. Il peut en pratique être copié et légèrement adapté à votre cas.

## 8.2.2. Utilisation d'un logiciel basé sur l'IA (conception générative)

De nombreux logiciels intègrent désormais l'intelligence artificielle, et celle-ci est employée dans divers secteurs industriels, à différentes étapes des projets, que ce soit pour la gestion de la production, le tri des déchets ou, plus

évidemment, dans le domaine de la robotique. Nous allons ici présenter l'utilisation de l'IA dans la conception, qui est le sujet de ce livre, avec en particulier la conception générative, une méthode de création qui tend à s'imposer de plus en plus dans les projets grâce à son efficacité.

La conception générative (ou *generativ design* en anglais) est une méthode de création assistée par ordinateur qui utilise l'intelligence artificielle pour explorer automatiquement une large gamme de solutions de design en fonction de critères et contraintes définis par l'utilisateur. Contrairement aux méthodes de conception traditionnelles, où le concepteur produit manuellement des options, la conception générative permet de générer un grand nombre de propositions en simulant différents scénarios et configurations. Ce processus repose sur des algorithmes d'IA qui testent et ajustent les designs en fonction de paramètres tels que la cinématique, la résistance, le poids, l'efficacité énergétique, les coûts, ou encore les impacts environnementaux. La conception générative se distingue ainsi par sa capacité à optimiser le processus créatif, offrant des solutions innovantes et souvent inattendues, tout en intégrant des critères de performance, de durabilité et d'efficacité au cœur même de la conception.

La conception générative, en explorant un espace de solutions considérablement élargi, offre des opportunités uniques pour optimiser les performances environnementales des produits. En intégrant dès le départ des contraintes environnementales dans le processus de conception, elle permet la minimisation de la quantité de matière, de comparer et de choisir les matériaux répondant à des critères fixés tels que la recyclabilité, ou la réduction des déchets de fabrication. Elle permet l'optimisation des formes face à des critères fonctionnels, de structure ou encore d'efficacité énergétique. Finalement, la conception générative ouvre de nouvelles perspectives pour créer des produits plus durables et plus respectueux de l'environnement. Elle sert également pour réduire l'empreinte carbone, prolonger la durée de vie des produits et stimuler l'innovation dans tous les secteurs.

Nous allons alors mettre en œuvre cette méthode dans l'exemple de la reconception d'un bras d'un robot sériel indiqué par la flèche sur l'image suivante :

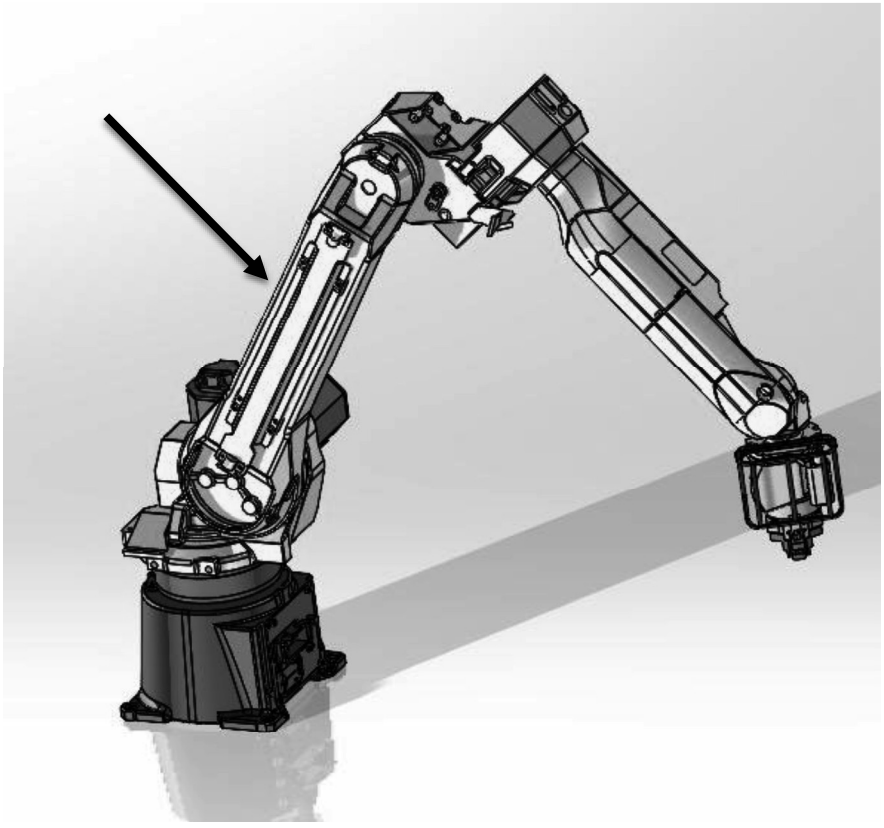


*Figure 51 : bras du robot à reconcevoir*

Le but de cette reconception sera d'obtenir un bras plus léger pour réduire son inertie et ainsi réduire la consommation des moteurs et la précision du déplacement tout en garantissant sa résistance avec un coefficient de sécurité de 2. Toujours dans cette optique de précision, nous souhaiterons que le bras soit le plus rigide possible. Par ailleurs, le système étant susceptible de travailler en compression, nous souhaiterons qu'il garantisse les conditions de non flambage.

Pour la reconception de ce bras, nous utiliserons le logiciel Fusion 360 d'Autodesk avec son module *Generativ design*.