L'IA et l'apprentissage automatique aux échecs

BEN EL CADI Talal

Juin 2024

Table de matière

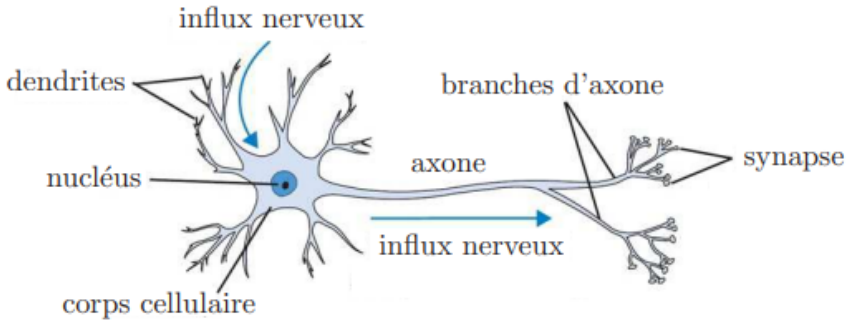
I. Introduction

Problématique

Comment automatiser, optimiser et modéliser les échecs à l'aide de l'intelligence artificielle ?

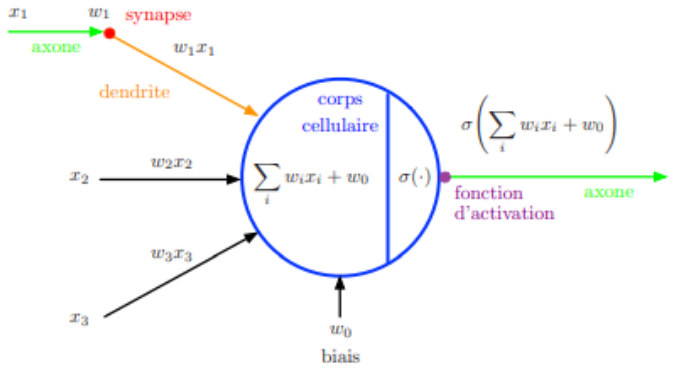
II. Analyse Théorique

Modèle



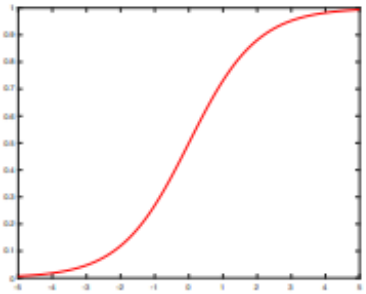
Chaque neurone artificiel calcule une sortie à partir de ses entrées.
Les opérations mises en œuvre sont :

- combinaison linéaire des entrées
- application d'une fonction d'activation



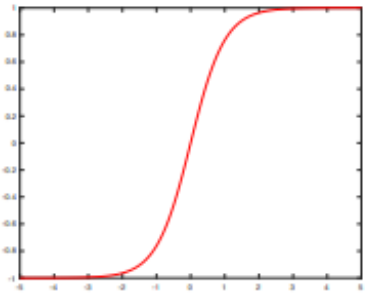
Les fonctions d'activation principalement utilisées

sigmoïde



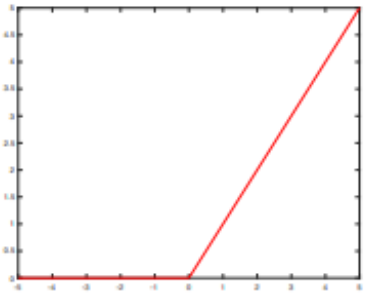
$$\sigma(a) = \frac{1}{1+\exp(-a)}$$

tangente hyperbolique



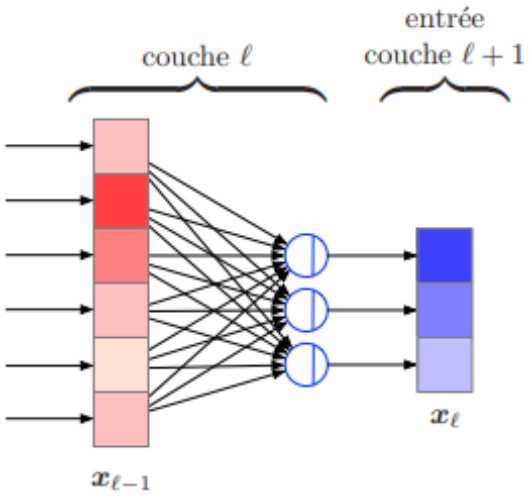
$$\sigma(a) = \frac{\exp(a)-\exp(-a)}{\exp(a)+\exp(-a)}$$

rectified linear unit (ReLU)



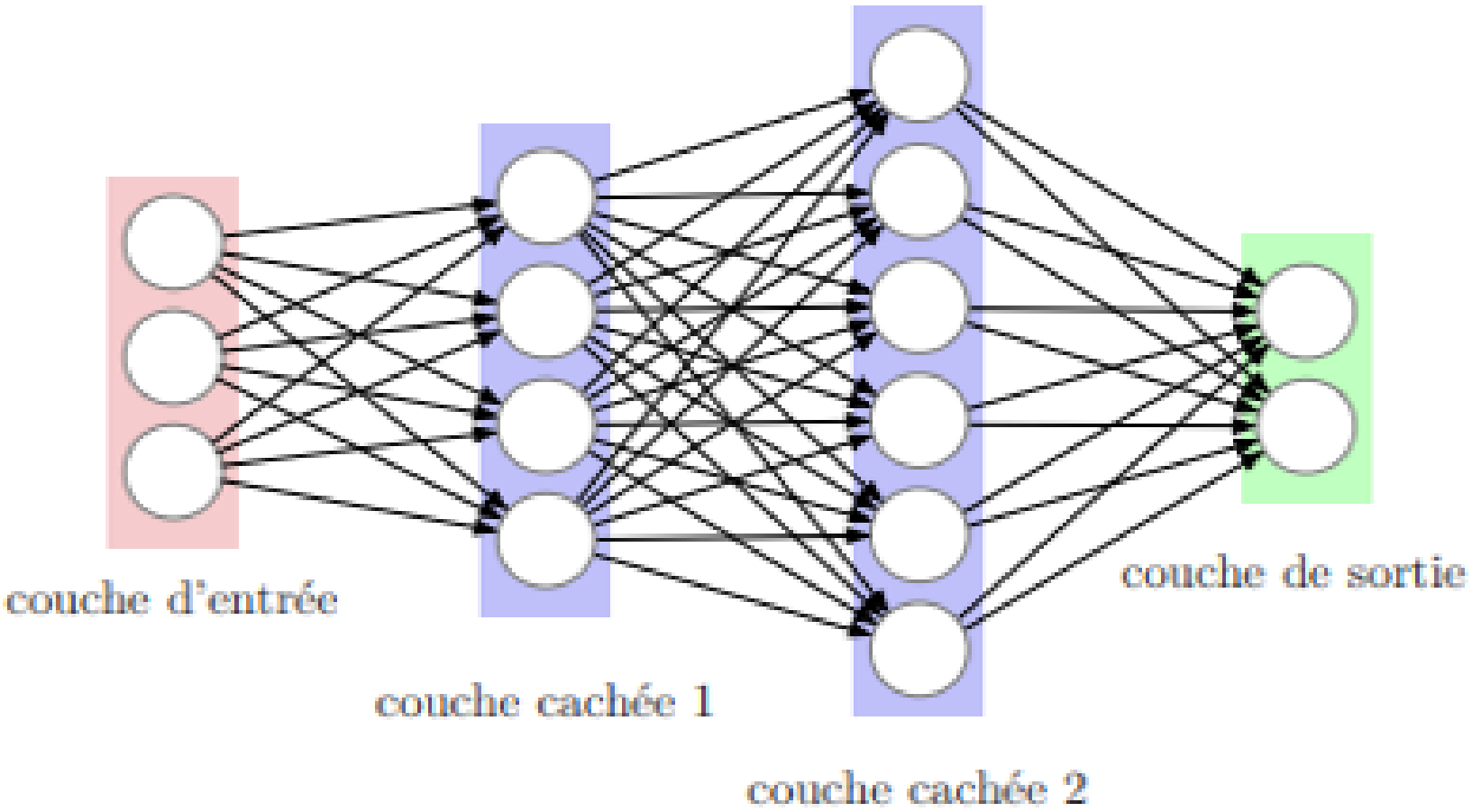
$$\sigma(a) = \max\{0, a\}$$

Architecture neuronale :



$$\boldsymbol{x}_\ell = \sigma(W_\ell \boldsymbol{x}_{\ell-1} + \boldsymbol{w}_{\ell 0})$$

Perceptron multicouche



Perceptron multicouche

La couche d'entrée est le point d'entrée des données. Elle ne dispose pas de fonction d'activation.

Chaque couche cachée ℓ est caractérisée par:

- une matrice W_ℓ de poids à estimer, de taille $\mathbf{dim}(x_\ell) \times \mathbf{dim}(x_{\ell-1})$ (notation: nbre de lignes $\times$ nbre de colonnes)
- une fonction d'activation $\sigma(\cdot)$. Notation: $\sigma(z)$ s'entend par $\sigma(z_i)$ appliqué à chaque composante z_i de z.

La couche de sortie dispose éventuellement d'une fonction d'activation $\phi(\cdot)$, différente de $\sigma(\cdot)$, qui dépend du problème à résoudre (régression, classification).

Une fonction objectif à optimiser pour déterminer les matrices W_ℓ de poids.

Formulation du perceptron multicouche à L couches

- **Couche d'entrée:** La donnée x est présentée:
  $x_0 = x$
- **Pour chaque couche cachée** $\ell = 1, \ldots, L-1$**:**
  $Z_\ell = W_\ell x_{\ell-1} + w_{\ell 0}$
- **Couche de sortie:** La sortie $\hat{y}$ est obtenue:
  $\hat{y} = \varphi(z_L)$
  $z_L = W_L x_{L-1} + w_{L0}$

Théorème d'approximation universelle

On s'intéresse aux réseaux de neurones feedforward à 1 couche cachée, c'est-à-dire, aux fonctions $f : \mathbb{R}^D \to \mathbb{R}$ de la forme :

$$f(x) = \sum_{i=1}^{N} \alpha_i \sigma(w_i^\top x + w_{i0})$$

avec $\sigma : \mathbb{R} \to \mathbb{R}$ une fonction d'activation.

Cybenko (1989)
Soit $\sigma : \mathbb{R} \to \mathbb{R}$ continue et sigmoïdale ($\lim_{x \to -\infty} \sigma(x) = 0$ et $\lim_{x \to +\infty} \sigma(x) = 1$). L'ensemble des réseaux de neurones est dense dans $C([0,1]; \mathbb{R})$.

Hornik (1991)
Soit $K \subset \mathbb{R}^D$ un compact. Soit $\sigma : \mathbb{R} \to \mathbb{R}$ continue, bornée et non-constante. L'ensemble des réseaux de neurones est dense dans $C([0,1]; \mathbb{R})$.

Fonction objectif

Soit $\hat{y} = f(x, \theta)$ la sortie du perceptron où :

- x: entrée du perceptron
- θ: ensemble de tous les paramètres $W_1, \ldots, W_L$ à estimer

Fonction objectif à minimiser:

$$\theta^* = \arg\min_\theta \sum_{n=1}^{N} J(f(x_n, \theta), y_n) \tag{1}$$

Avec :

- $A = \{(x_n, y_n)\}_{n=1}^{N}$: base d'apprentissage
- $J(\hat{y}_n, y_n)$: fonction de perte (loss function) pour la paire $(\hat{y}_n, y_n)$

Apprentissage par rétropropagation du gradient

Fonction objectif à minimiser:

$$\theta^* = \arg\min_{\theta} \frac{1}{N} \sum_{n=1}^{N} J(f(x_n, \theta), y_n)$$

Afin de résoudre ce problème, une méthode de descente de gradient stochastique doit être mise en œuvre car il n'existe pas de solution analytique.

L'algorithme de rétropropagation du gradient (backprop), de **Rumelhart (1988)**, joue un rôle central en Deep Learning.

Il permet d'estimer le gradient de la fonction objectif en tout point θ.

Descente de gradient

Soit θ^0 un point initial, et $\mu > 0$ un pas d'apprentissage : Par la formule de Taylor-Young en $\theta = \theta_o - \mu \nabla J(\theta_o)$, on a :

$$J(\theta) - J(\theta_o) = -\mu ||\nabla J(\theta_o)||^2 + o(\alpha)$$

En conséquence, si $\nabla J(\theta_o) \neq 0$ et μ suffisamment petit, alors :

$$J(\theta) < J(\theta_o)$$

Ce qui signifie que θ est une amélioration par rapport à θ_o au sens du critère $J(\theta)$ lorsque l'on souhaite le minimiser.

Algorithme de descente du gradient

Soit un problème de la forme $\theta^* = \arg\min_\theta J(\theta)$ où $J(\theta)$ est une fonction objectif différentiable. Si la condition d'optimalité $\nabla J(\theta) = 0$ n'admet pas de solution analytique, il est nécessaire de recourir à une méthode numérique.

Algorithme du gradient : Choisir un point initial $\theta^{(0)}$, un seuil ϵ, et un pas d'apprentissage $\mu > 0$.

Itérer les étapes suivantes à partir de $k = 0$:

1. Calculer $\nabla J(\theta^{(k)})$
2. Test d'arrêt (exemple) : si $k||\nabla J(\theta^{(k)})|| < \epsilon$, arrêt
3. Nouvel itéré : $\theta^{(k+1)} = \theta^{(k)} - \mu \nabla J(\theta^{(k)})$

Tests d'arrêt

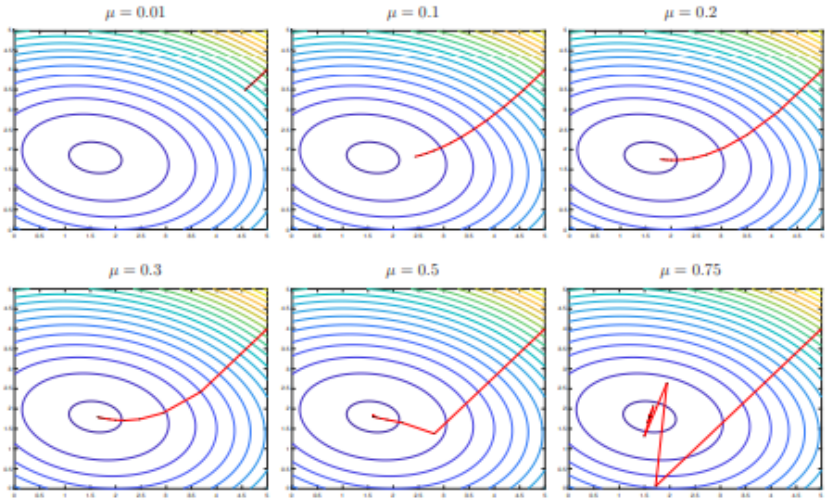
La condition nécessaire d'optimalité $||\nabla J(\theta^{(k)})|| = 0$ n'a pas d'intérêt pratique. En pratique, on préconise :

- Condition sur le gradient : $||\nabla J(\theta^{(k)})|| < \epsilon$
- Conditions sur la fonction objectif : $|J(\theta^{(k)}) - J(\theta^{(k-1)})| < \epsilon$ ou encore $\left|\frac{J(\theta^{(k)}) - J(\theta^{(k-1)})}{J(\theta^{(k-1)})}\right| < \epsilon$
- Conditions sur les itérés : $||\theta^{(k)} - \theta^{(k-1)}|| < \epsilon$ ou encore $\frac{||\theta^{(k)} - \theta^{(k-1)}||}{||\theta^{(k-1)}||} < \epsilon$

On peut remplacer les dénominateurs "$\times$" ci-dessus par $\max\{1, \times\}$.

Choix du pas d'apprentissage μ

- Choix d'un petit pas μ :
  - Avantage : convergence assurée, le long de la ligne de plus grande pente.
  - Inconvénient : convergence lente.
- Choix d'un grand pas μ :
  - Avantage : convergence rapide, en peu d'itérations.
  - Inconvénient : risque important de divergence de l'algorithme.
- En pratique :
  - Pas fixe lorsque la fonction objectif est L-Lipschitz ($\mu < \frac{1}{L}$).
  - Pas variable par recherche linéaire.



Apprentissage de réseau de neurones

Descente de gradient : suit la direction de plus grande pente en chaque point

$$\theta^{(k+1)} = \theta^{(k)} - \frac{\mu}{N} \sum_{n=1}^{N} \nabla_\theta J(f(x_n, \theta^{(k)}), y_n)$$

Descente de gradient stochastique (SGD) : estime le gradient à partir de la donnée d'apprentissage courante

$$\theta^{(k+1)} = \theta^{(k)} - \mu \nabla_\theta J(f(x_{n_k}, \theta^{(k)}), y_{n_k})$$

Descente de gradient stochastique mini-batch : estime le gradient à partir d'un sous-ensemble B_k de données d'apprentissage

$$\theta^{(k+1)} = \theta^{(k)} - \frac{\mu}{|B_k|} \sum_{n \in B_k} \nabla_\theta J(f(x_n, \theta^{(k)}), y_n)$$

Algorithme général

Algorithm 1 Mise à jour des poids d'un réseau de neurones avec SGD mini-batch

Require: Base d'apprentissage A, résultats R, initialisation aléatoire des poids θ (petites valeurs), nombre d'époques E

1: **for** $e = 1$ **to** E **do**
2: Mettre à jour le taux d'apprentissage μ_e
3: Mélanger la base d'apprentissage
4: Diviser en sous-ensembles B_k (mini-batches)
5: **for** $k = 1$ **to** K **do**
6: Mettre à jour les poids :

$$\theta^{(k+1)} = \theta^{(k)} - \frac{\mu_e}{|B_k|} \sum_{n \in B_k} \nabla_\theta J(f(x_n, \theta^{(k)}), y_n)$$

7: **end for**
8: **end for**

III. Simulation échiquéenne :

Programmation du cerveau

Programme en Python de Minimax avec élagage

```
1  import chess
2  # P = 100
3  # N = 310
4  # B = 320
5  # R = 500
6  # Q = 900
7  # Position table
8  pieceSquareTable = [
9    [ -50,-40,-30,-30,-30,-30,-40,-50 ],
10   [ -40,-20,  0,  0,  0,  0,-20,-40 ],
11   [ -30,  0, 10, 15, 15, 10,  0,-30 ],
12   [ -30,  5, 15, 20, 20, 15,  5,-30 ],
13   [ -30,  0, 15, 20, 20, 15,  0,-30 ],
14   [ -30,  5, 10, 15, 15, 10,  5,-30 ],
15   [ -40,-20,  0,  5,  5,  0,-20,-40 ],
16   [ -50,-40,-30,-30,-30,-30,-40,-50 ]
17 ]
18
```

Minimax with Alpha-Beta Pruning (Part 1)

```
1  def eval(board):
2  scoreWhite = 0
3  scoreBlack = 0
4  for i in range(0,8):
5  for j in range(0,8):
6  squareIJ = chess.square(i,j)
7  pieceIJ = board.piece_at(squareIJ)
8  if str(pieceIJ) == "P":
9  scoreWhite += (100 + pieceSquareTable[i][j])
10 if str(pieceIJ) == "N":
11     scoreWhite += (310 + pieceSquareTable[i][j])
12 if str(pieceIJ) == "B":
13     scoreWhite += (320 + pieceSquareTable[i][j])
14 if str(pieceIJ) == "R":
15     scoreWhite += (500 + pieceSquareTable[i][j])
16 if str(pieceIJ) == "Q":
17     scoreWhite += (900 + pieceSquareTable[i][j])
18 if str(pieceIJ) == "p":
19     scoreBlack += (100 + pieceSquareTable[i][j])
20 if str(pieceIJ) == "n":
21     scoreBlack += (310 + pieceSquareTable[i][j])
```

Minimax with Alpha-Beta Pruning (Part 2)

```
1  if str(pieceIJ) == "b":
2      scoreBlack += (320 + pieceSquareTable[i][j])
3  if str(pieceIJ) == "r":
4      scoreBlack += (500 + pieceSquareTable[i][j])
5  if str(pieceIJ) == "q":
6      scoreBlack += (900 + pieceSquareTable[i][j])
7  return scoreWhite - scoreBlack
8  NODECOUNT = 0
9  def alphaBeta(board, depth, alpha, beta, maximize):
10     global NODECOUNT
11     if(board.is_checkmate()):
12         if(board.turn == chess.WHITE):
13             return -100000
14         else:
15             return 1000000
16     if depth == 0:
17         return eval(board)
18     legals = board.legal_moves
19     if(maximize):
20         bestVal = -9999
21
```

Minimax with Alpha-Beta Pruning (Part 3)

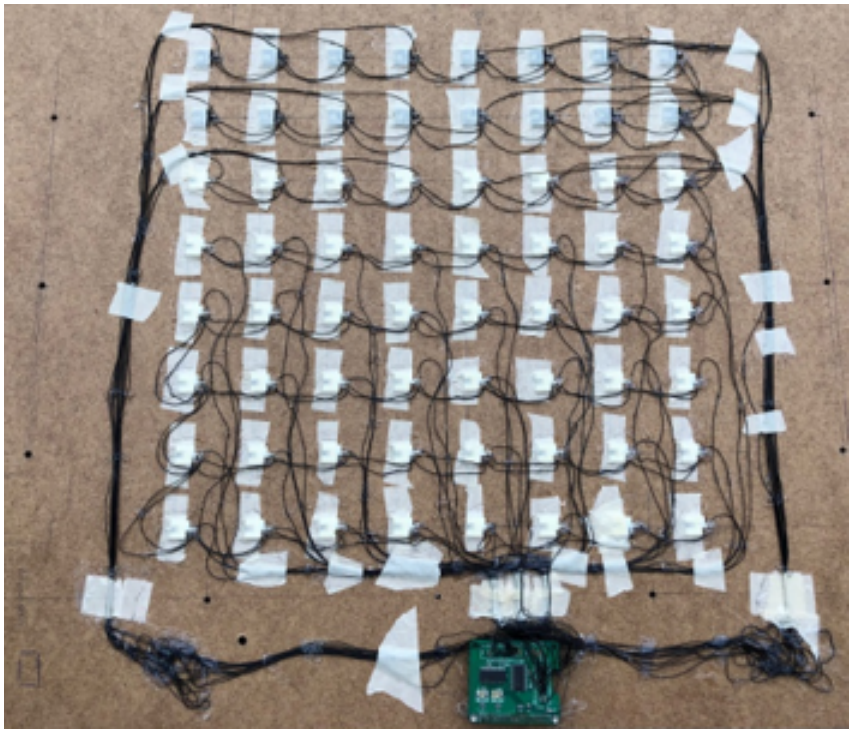
```
            for move in legals:
                board.push(move)
                NODECOUNT += 1
                bestVal = max(bestVal, alphaBeta(board, dep
                board.pop()
                alpha = max(alpha, bestVal)
                if alpha >= beta:
                    return bestVal
        return bestVal
else:
    bestVal = 9999
    for move in legals:
        board.push(move)  NODECOUNT += 1
bestVal = min(bestVal, alphaBeta(board, depth - 1, alph
        board.pop()
        beta = min(beta, bestVal)
        if beta <= alpha:  return bestVal
    return bestVal

```

Minimax with Alpha-Beta Pruning (Part 4)

```
1     def getNextMove(depth, board, maximize):
2     legals = board.legal_moves
3     bestMove = None
4     bestValue = -9999
5     if(not maximize):
6         bestValue = 9999
7     for move in legals:
8         board.push(move)
9         value = alphaBeta(board, depth-1, -10000, 10000
10        board.pop()
11        if maximize:
12            if value > bestValue:
13                bestValue = value
14                bestMove = move
15        else:
16            if value < bestValue:
17                bestValue = value
18                bestMove = move
19    return (bestMove, bestValue)
```

Minimax with Alpha-Beta Pruning (Part 5)

```
1 board = chess.Board()
2 board.push_san("e4")
3 board.push_san("e5")
4 board.push_san("Qh5")
5 board.push_san("Nc6")
6 board.push_san("Bc4")
7 board.push_san("Nf6")
8 print(getNextMove(4, board, True))
9 print(NODECOUNT)
```



Un aimant est placé dans chaque pièce et un hall effet sensor (un capteur de magnétisme) est placé sous chaque case. Il y a donc 64 capteurs possédant chacun trois broches. Comme les connexions se font entre l'ordinateur et le capteur, deux fois plus de soudage est à faire. Au total, 384 (64 capteurs 3 broches 2 connexions).

2.2.1 Mécanisme physique:

Figure: La fixation au niveau du chariot

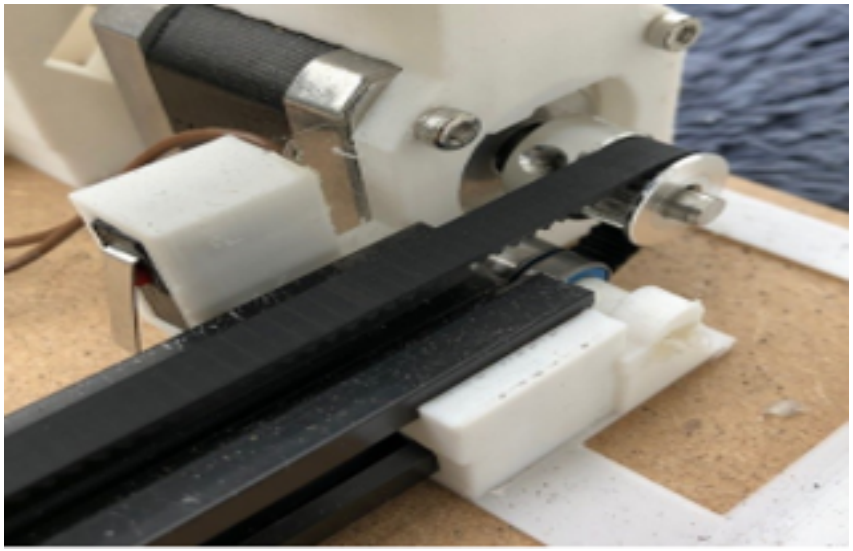


Figure: La fixation au niveau du moteur

2.2 Déplacement d'une pièce :

Prise de marque :

- Les pièces d'échecs peuvent être déplacées en activant un aimant externe, car chaque pièce est équipée d'un aimant.
- Des moteurs pas à pas sont utilisés pour le déplacement des pièces, offrant une précision estimée à 0,3 mm.
- Les électroaimants permettent de déplacer les pièces sur un plan à deux dimensions en étant placés sur deux axes.
- Le mécanisme de déplacement comprend des composants tels que des filaments à dents crantées, des chariots et des roulements à billes.

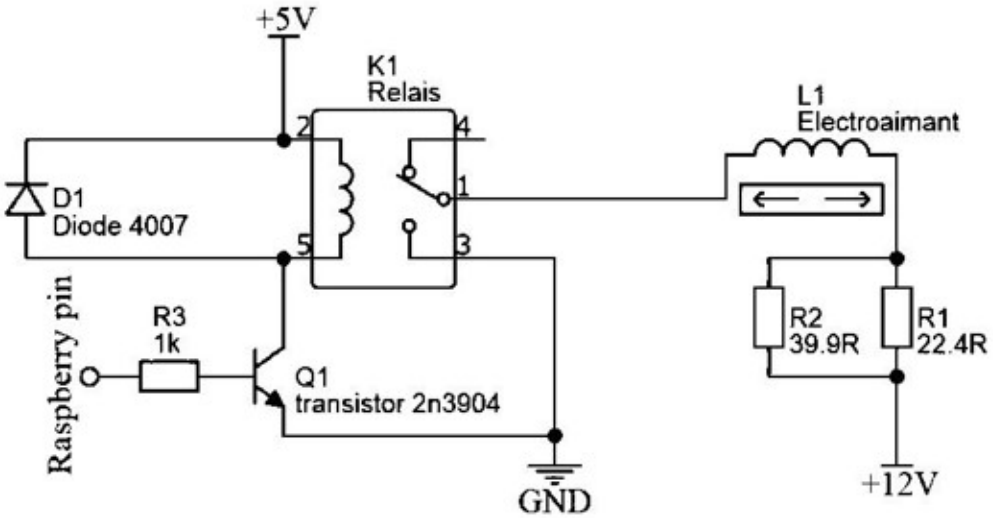
2.2.2 Fonctionnement théorique du logiciel

- Choix du micro-contrôleur : La Raspberry Pi 3 est choisie pour sa puissance de calcul et sa connectivité réseau. Trois cartes d'extension permettent d'ajouter suffisamment de pins I/O pour contrôler 96 pins, dont 64 dédiées aux capteurs.
- Système d'exploitation : Raspbian lite est sélectionné pour sa compatibilité avec Raspberry Pi, sa gratuité et sa faible consommation de ressources, adaptée à l'absence d'interface graphique.

2.2.2 Fonctionnement théorique du logiciel

- Utilisation d'un serveur externe pour le calcul : Étant donné la puissance limitée de la Raspberry Pi 3, un serveur externe plus puissant sera chargé de calculer les meilleurs coups à jouer pour éviter un niveau de jeu faible de l'IA.
- Algorithme A pour le déplacement des pièces : L'algorithme A* est choisi pour résoudre le problème de recherche de chemin dans un graphe, en évaluant la distance des nœuds au but tout en tenant compte de la distance déjà parcourue.
- Détermination des coordonnées physiques de chaque case : Des interrupteurs placés à l'origine des deux axes permettent aux moteurs de s'initialiser à chaque démarrage de la machine. Les coordonnées de chaque case sont déterminées empiriquement en mesurant la distance par rapport à ces points d'origine.

2.3.1 Le circuit dédié à l'électroaimant



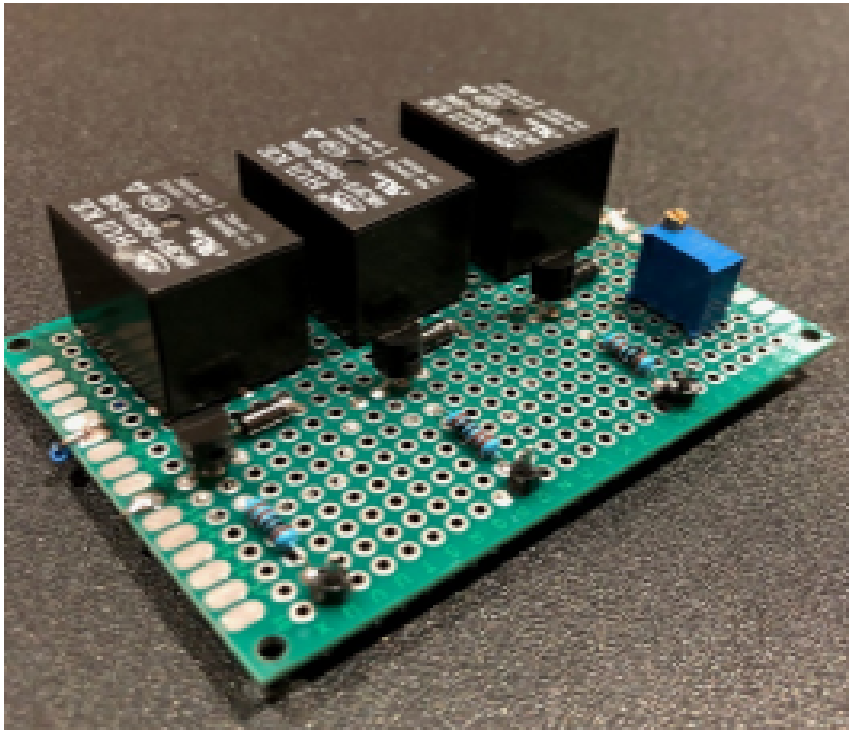
La tension alimentant l'électroaimant est réduite à 6,9 volts, diminuant ainsi son champ magnétique :

$$B = \frac{\nu_0 \cdot n \cdot I}{l}$$

2.3 Circuit électrique de l'échiquier

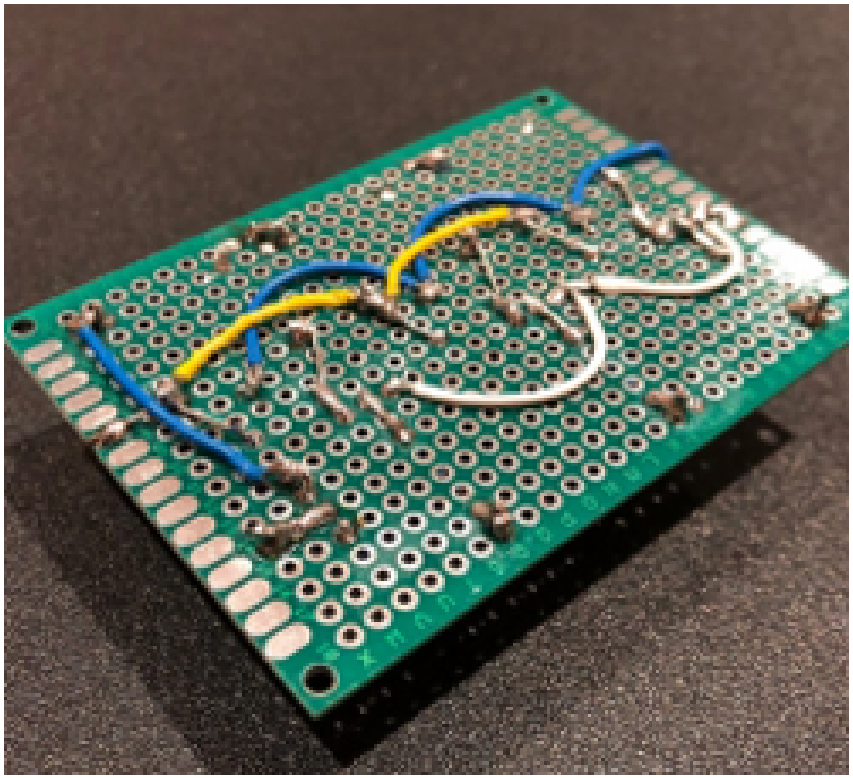
1 : - Pour ajuster la tension :

$V_s = \frac{R_a}{R_a+R_b} \times V_e \Rightarrow R_b = \frac{V_e R_a}{V_s} - R_b \Rightarrow R_b = \left(\frac{12.4 \times 18}{6.9}\right) - 18 = 14.3\,\Omega$



2: - la résistance variable :

$\Rightarrow R_2 = \frac{R_b \times R_1}{R_1 - R_b} = 39.9\,\Omega$



Le circuit permettant d'activer l'électroaimant et la lumière de certains boutons.

2.3.2 Le circuit pour faire fonctionner les moteurs pas à pas :

- Utilisation du driver A4988 (une impulsion de 5 en 12 volts sur 4 spins)
- Séparation de l'alimentation des moteurs de de l'électroaimant (à cause des interférences).
- Un potentiomètre régule la tension entre le 12 volts et la masse pour prévenir les surtensions.
- Le 5 volts des drivers est connecté au Raspberry Pi pour assurer une alimentation stable.
- Les drivers ajustent l'intensité des moteurs via un potentiomètre, en utilisant la formule $I_{\text{max}} = \frac{V_{\text{ref}}}{8R_s}$

Vref: est le voltage aux bornes du potentiomètre.
Rs: est la résistance du driver utilisé.

3. Explication du code de fonctions importantes liant le corps et le cerveau

```
1  def get_move():
2      piece_removed = False
3      capturing_piece = False
4      before = get_board_map()
5      nbr_pieces_before = get_nbr_pieces(before)
6      case_piecer = boardr = after = case_after = 0
7      while True:
8          bmap = get_board_map()
9          nbr_pieces = get_nbr_pieces(bmap)
10         if bmap != before and not piece_removed:
11             piece_removed = True
12             case_piecer = bmap ^ before
13             boardr = bmap
14         elif (piece_removed and nbr_pieces == nbr_piec
15                 nbr_pieces == (nbr_pieces_before - 1))
16             after = bmap
17             case_after = boardr ^ after
18             break
19
```

3. Explication du code de fonctions importantes liant le corps et le cerveau

```
1         elif piece_removed and nbr_pieces == (nbr_piec
2             case_piecer = bmap ^ boardr
3             boardr = bmap
4             capturing_piece = True
5             time.sleep(2)  # so there's less probabili
6         time.sleep(0.5)
7     return case_index_to_coo(math.log2(case_piecer)) +
8
```

3.1 La fonction move nb magnet

```
1  def move_nb_magnet(self, tick, dir):
2      delta = 0
3      busMot.write_pin(dirnb, dir)
4      if dir == home_dir:
5          while self.nb != 0 and delta != tick:
6              busMot.write_pin(stepnb, 1)
7              time.sleep(500/1000000)
8              busMot.write_pin(stepnb, 0)
9              time.sleep(500/1000000)
10             delta += 1
11             self.nb -= 1
12             print("[+] Magnet set to", self.nb, "nb.")
13     elif dir == ext_dir:
14         while self.nb < MAXNB and delta < tick:
15             busMot.write_pin(stepnb, 1)
16             time.sleep(500/1000000)
17             busMot.write_pin(stepnb, 0)
18             time.sleep(500/1000000)
19             delta += 1
20             self.nb += 1
21             print("[+] Magnet set to", self.nb, "nb.")
```

3.2 La fonction Astar

```
1  def Astar(graph, start, end):
2      open_list = [] # list of nodes visited but not exp
3      closed_list = [] # list of nodes visited and expan
4      open_list.append(Node(start[0], start[1], None))
5      while len(open_list) > 0: # while not empty
6          open_list.sort(key=lambda z: z.h) # get lowest
7          current_n = open_list[0]
8          del open_list[0]
9          closed_list.append(current_n)
10         if current_n.x == end[0] and current_n.y == en
11             return current_n
```

3.2 La fonction Astar

```
        # generate children
        for shift in [(0, 1), (1, 0), (0, -1), (-1, 0)]
            child = evaluate_child(graph, shift, curre
            if not child:
                continue
            if child in closed_list:
                continue
            elif child in open_list and open_list[open
                continue
            else:
                child.c += SAFETY if graph[child.y][ch
                child.h = child.c + (end[0] - child.x)
                open_list.append(child)
    return None

```

Dans toute la fonction Astar un graphe est utilisé. Celui-ci est généré grâce à une fonction qui prend comme point de départ le graphe suivant :

```
graph =[[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
        [0,2,0,2,0,2,0,2,0,2,0,2,0,2,0],
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
        [0,2,0,2,0,2,0,2,0,2,0,2,0,2,0],
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
        [0,2,0,2,0,2,0,2,0,2,0,2,0,2,0],
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
        [0,2,0,2,0,2,0,2,0,2,0,2,0,2,0],
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
        [0,2,0,2,0,2,0,2,0,2,0,2,0,2,0],
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
        [0,2,0,2,0,2,0,2,0,2,0,2,0,2,0],
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
        [0,2,0,2,0,2,0,2,0,2,0,2,0,2,0],
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
        [2,2,2,2,2,2,2,2,2,2,2,2,2,2,2]]
```

Les 0 symbolisent des noeuds vierges. Les 1 symbolisent des murs et les 2 sont les noeuds qu'il est préférable de ne pas traverser (pour les noeuds entre deux cases). Ensuite la fonction qui génère le graphe n'a plus qu'à ajouter des 1 sur les noeuds où il se trouve une pièce.

Côté client :

```
1  import socket
2  CPPPORT = ***
3  class SocketConnexion(object):
4  def init(self):
5  self.sock =
6  socket.socket(socket.AFINET,
7  socket.SOCKSTREAM)
8  self.sock.connect(("***.***.***.**"
9  , CPPPORT))
10 def senddata(self, strdata) :
11 self.sock.sendall(strdata.encode('
12 ascii'))
13 def recvdata(self) :
14 return
15 self.sock.recv(1024).decode('ascii'
16 )
```

3.3 La connexion réseau liant le corps et le cerveau

- Le programme installé sur le Raspberry Pi utilise le module socket en Python pour établir une connexion. Il utilise le protocole TCP avec socket.SOCK_STREAM et socket.AF_INET pour IPv4 (ligne 9).
- Les données sont transmises par blocs de 1024 bits, bien que cette limite ne soit généralement pas atteinte si le programme fonctionne correctement. Les données sont encodées en ASCII avant d'être transmises, car seuls l'alphabet et les chiffres sont utilisés lors de la communication.

Côté serveur :

Le code qui ouvre le port du serveur et communique avec le Raspberry Pi est écrit en C++ pour correspondre à l'environnement et à la version finale du cerveau, programmés en C++. Le code n'est pas inclus dans ce document en raison de sa longueur (plus de 180 lignes). La bibliothèque WinSock2 de Microsoft est utilisée pour la communication réseau. Le code est basé sur le tutoriel de Microsoft, ce qui facilite sa compréhension car les fonctions de la bibliothèque WinSock2 sont expliquées en détail. Il est à noter que ce programme ne fonctionnera que sous Windows, car WinSock2 est conçu pour ce système d'exploitation.

Illustrations du projet :

Vu externe du plateau :

Si une pièce est capturée, les moteurs doivent d'abord déplacer la pièce capturée pour déplacer ensuite sa propre pièce. Il y a donc deux actions. Les actions sont séparées par "" donc dans l'exemple si la pièce déplacée précédemment par le joueur est capturée, le coup ressemblera à "a3CAb4a3".

IV. Conclusion

Merci de votre attention

Annexe

Réseau neuronal profond dans AlphaZero :

- Le réseau neuronal utilisé dans AlphaZero est un réseau neuronal convolutif (CNN) avec des blocs résiduels.
- Il prend les états de jeu en entrée et produit deux sorties : les probabilités de mouvement p_i et le taux de victoire prédit à cet état de jeu v_i.
- Le réseau peut être représenté par l'équation $(p, v) = f_\theta(s)$, où f_θ désigne le réseau neuronal paramétré par θ, qui approxime la fonction de mapping entre l'état d'entrée s et la sortie (p, v).
- Pour entraîner le réseau, des données d'entraînement sont générées en combinant les états de jeu s, les probabilités de recherche π et le vainqueur du jeu z.

Réseau neuronal profond dans AlphaZero :

- La fonction de perte est définie comme suit :

$$l = (z - v)^2 - \pi_T \log p + c\|\theta\|^2$$

- L'objectif est de maximiser la similarité entre le vecteur de politique p et les probabilités de recherche π en utilisant une perte de cross-entropy, tout en minimisant la différence quadratique moyenne entre le vainqueur prédit v et le vainqueur réel du jeu z.
- Un terme supplémentaire $c\|\theta\|^2$ est ajouté en tant que régularisation de la norme L2 pour éviter le surajustement du réseau.

Recherche d'arborescence de Monte Carlo dans AlphaZero

L'algorithme peut être divisé en trois phases: **la phase de sélection, la phase d'évaluation et d'expansion, et la phase de rétropropagation**.

Dans **la phase de sélection**, l'algorithme traverse de manière itérative depuis l'état racine jusqu'à ce qu'il rencontre un état jamais visité auparavant (nœud feuille) ou un état terminal. Le fait qu'un état soit terminal est décidé par la règle du jeu, qui peut être automatiquement vérifiée par une fonction. **argmaxP UCT** est la fonction principale dans **MCTS**, comme le montre l'équation :

$$a = \text{argmax}_a \left(Q(s, a) + U(s, a) \right)$$

argmaxP UCT décide quelle action **MCTS** doit choisir au nœud actuel pour rechercher plus en profondeur dans l'arbre. Il est affecté par deux termes : **Q(s, a)** indique la valeur moyenne de l'état pour le nœud enfant qui sera atteint si nous effectuons a à l'état s, et **U(s, a) contrôle l'exploration**, comme le montre l'équation :

$$U(s,a) = c_{puct} \frac{P(s,a)}{1+N(s,a)} \sqrt{\frac{N(s)}{1+N(s,b)}}$$

- Dans la phase d'évaluation et d'expansion, l'algorithme vérifie si un nœud est un nœud terminal ou un nœud feuille. Pour un nœud terminal, sa valeur d'état est déterminée par les règles du jeu. Pour un nœud feuille non terminal, MCTS utilise son réseau neuronal actuel pour inférer p et v, puis définir v comme la valeur d'état pour ce nœud et sauvegarder p pour une expansion ultérieure de l'arbre. Pour étendre un nœud feuille non terminal, l'algorithme insère tous les nœuds enfants de ce nœud dans l'arbre de recherche, puis initialise leurs statistiques avec p, comme le montre l'équation :

$$N(s_{\text{Leaf}}, a) = 0, \qquad W(s_{\text{Leaf}}, a) = 0,$$
$$Q(s_{\text{Leaf}}, a) = 0, \qquad P(s_{\text{Leaf}}, a) = p_a$$

- Dans la phase de rétro propagation, après que la recherche a atteint une profondeur maximale prédéfinie, la valeur d'état est rétro propagée jusqu'à la racine, tout en mettant à jour les valeurs d'état moyennes de tous les nœuds le long du chemin de recherche. La mise à jour est effectuée selon les équations :

$$N(s_t, a_t) = N(s_t, a_t) + 1,$$
$$W(s_t, a_t) = W(s_t, a_t) + v,$$
$$Q(s_t, a_t) = \frac{W(s_t, a_t)}{N(s_t, a_t)}$$

Algorithm 2 MCTS algorithm in AlphaZero (Part 1)

```
1: for i = 1 to I do
2:     curr_node = root
3:     while curr_node.visits > 0 and not curr_node.is_term do
4:         curr_node = argmaxP UCT(curr_node.children)
5:     end while
6:     if curr_node.visits = 0 then
7:         (curr_node.is_term, z) = Ter_Eval(curr_node)
8:         if curr_node.is_term then
9:             curr_node.value = z
10:        else
11:            (p, v) = f_θ(s)
12:            curr_node.value = v
13:            curr_node.children = expand(curr_node, p)
14:        end if
15:    end if
16: end for
```

Algorithm 3 MCTS algorithm in AlphaZero (Part 2)

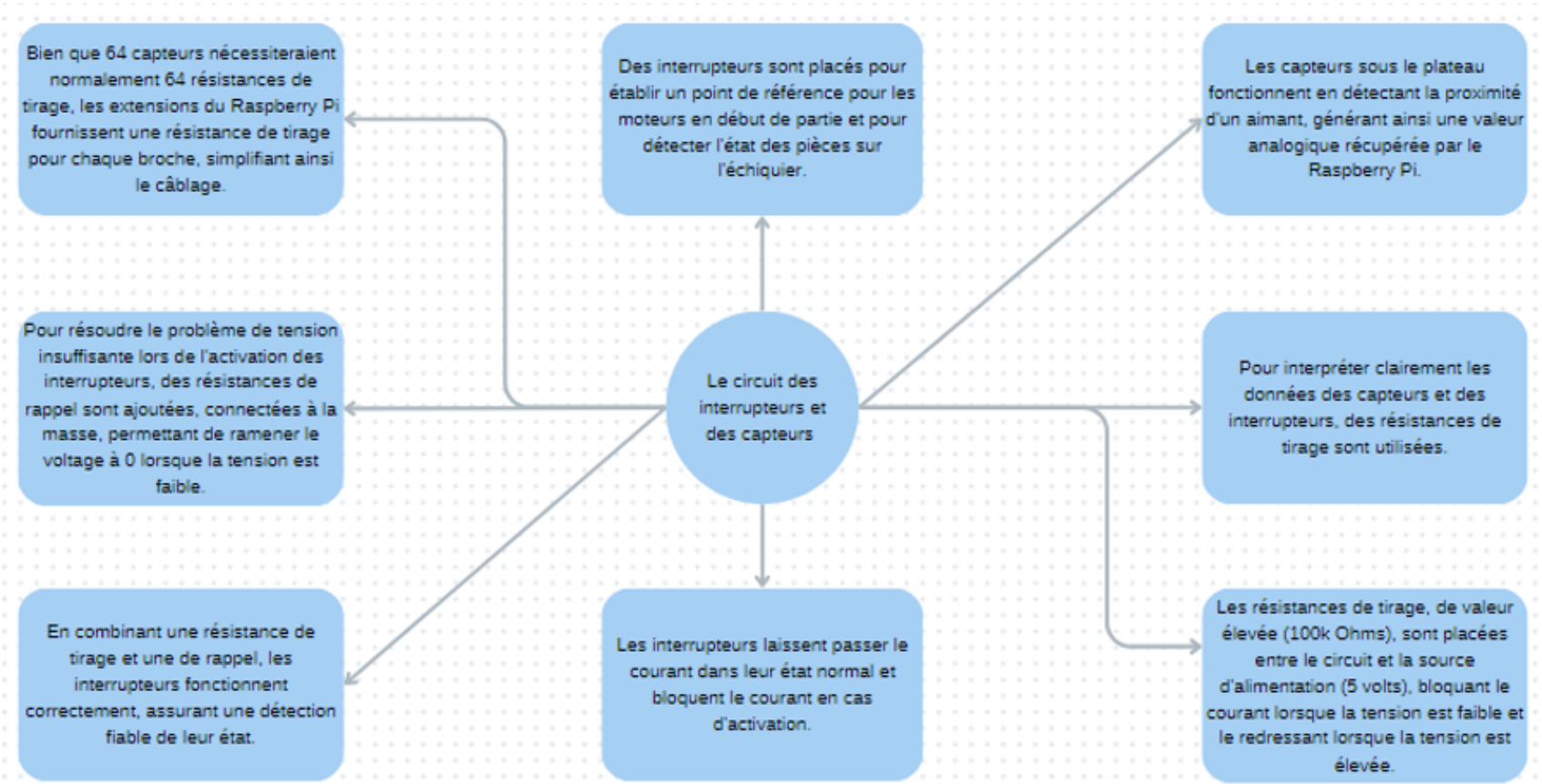
```
for i = 1 to I do
    curr_node = root
    while not curr_node.is_root do
        curr_node = curr_node.parent
        curr_node.visits++
        curr_node.value = curr_node.value + value
    end while
end for
```

Après un nombre suffisant d'étapes de recherche MCTS, AlphaZero effectue une action selon les statistiques du nœud racine (nœud actuel), passe à l'état de jeu suivant et démarre une nouvelle recherche MCTS sur le nouvel état de jeu. L'action réelle est choisie selon la distribution donnée par l'équation :

$$\pi(a|s) = \frac{N(s,a)}{\frac{1}{\tau}\sum_b N(s,b)}$$

où τ est un paramètre de température qui contrôle le degré d'exploration. L'algorithme aura une plus grande probabilité d'exploration (c'est-à-dire de ne pas choisir l'action avec le plus grand $N(s,a)$) si τ est élevé.

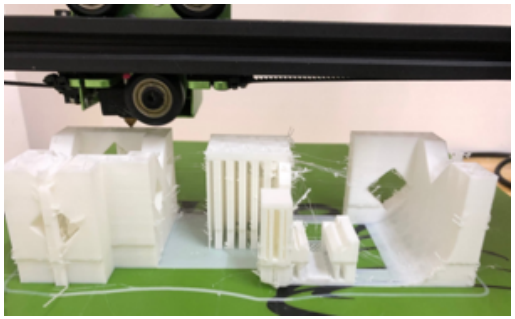
2.3.3 Le circuit des interrupteurs et des capteurs :



2.4 Matériel de construction de l'échiquier :

- L'utilisation d'une imprimante 3D pour créer des pièces sur mesure s'est avérée nécessaire pour construire un slider bidimensionnel. Le choix du PLA comme matériau d'impression offre une bonne précision et solidité si les pièces sont bien conçues.
- Fusion 360 est envisagé pour la conception des pièces.
- Utilisation d'un outil multi-usage de Dremel :
  - Trou à l'arrière du plateau pour faire passer l'alimentation.
  - Trous d'aération du circuit, disposés de manière à évoquer un échiquier avec un carré de huit par huit trous.
  - Trous sur le plateau pour permettre le passage des vis.

2.4 Matériel de construction de l'échiquier :



L'impression de la pièce qui soutient plusieurs profils et un moteur.

Les étapes de la peinture de l'échiquier