alphai

L'ÉDUCATION À L'INTELLIGENCE ARTIFICIELLE

Algorithme des k plus proches voisins

(corrigé)

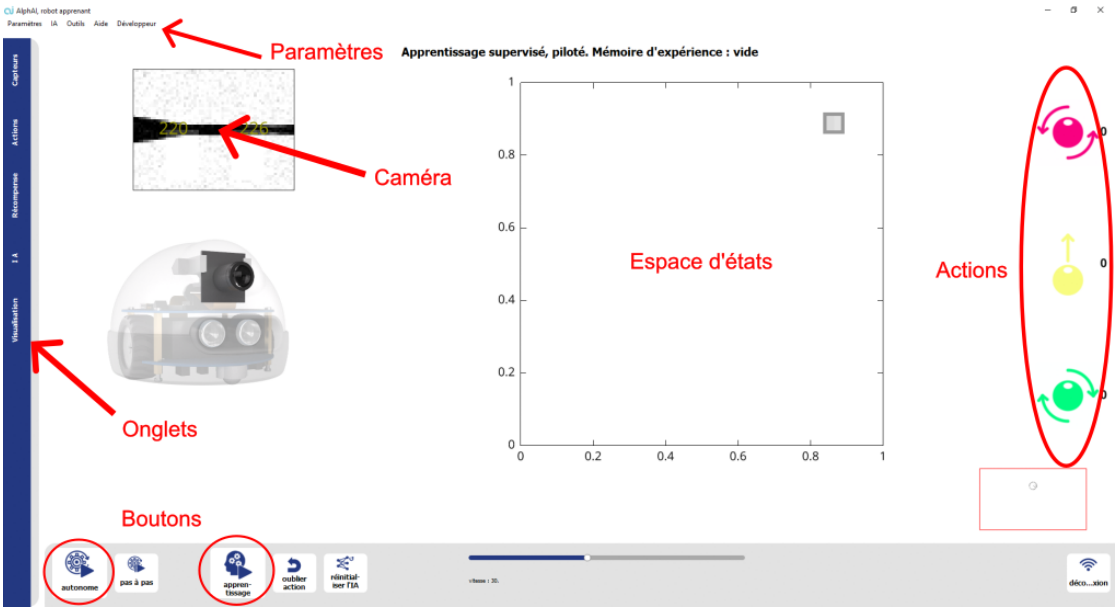


FIGURE 1 – Écran principal de **AlphAI**dans le scénario KNN

INTRODUCTION

Cette activité présente un algorithme d'apprentissage relativement simple appelé l'algorithme des k plus proches voisins. L'abbréviation de cet algorithme est KNN, qui vient de l'anglais *k nearest neighbors*.

Voici les trois objectifs de l'activité, correspondant approximativement aux trois parties qui le composent :

1. Comprendre la visualisation de l'espace d'états 2D présente dans le logiciel **AlphAI**.
2. Comprendre le principe de l'algorithme des k plus proches voisins.
3. Être capable de programmer soi-même cet algorithme en python.

1 PROGRAMMER LE ROBOT EN PYTHON

1.1 MISE EN PLACE

1. Lancez le logiciel **AlphAI**.
2. Connectez-vous à un robot ou un simulateur.
3. Dans les menus, sélectionnez **Fichier > Configurations d'exemple**, puis choisissez le premier scénario de la troisième ligne : **KNN Caméra**.
4. Dans l'onglet **Capteurs** sur le côté gauche, vous pouvez vérifier que le seul capteur actif est la caméra, et l'image apparaissant à l'écran doit être en noir et blanc.

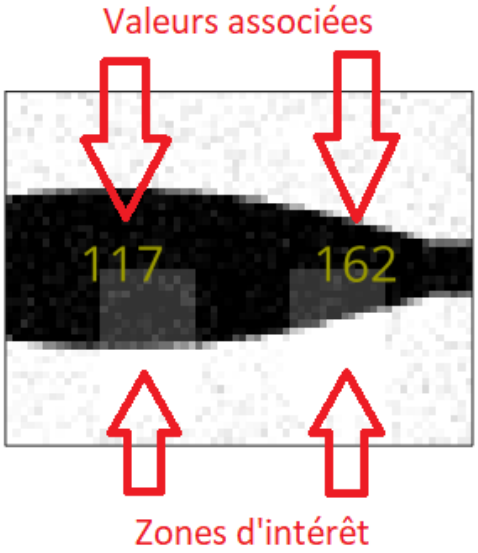


FIGURE 2 – Zones d'intérêt de l'image caméra

5. Dans l'onglet **Actions**, vous pouvez vérifier que seules les actions **avancer** et **pivoter** sont actives, ce qui donne un total de 3 actions disponibles pour le robot. Les icônes d'actions sont associées à des couleurs sur la droite de l'écran. Par la suite, vous pourrez décider de remplacer les actions **pivoter** par **pivoter un peu** si vous constatez que cela améliore le comportement du robot, mais veillez bien à toujours avoir uniquement 3 actions disponibles.

1.2 Comprendre le traitement d'image

1. Désactivez l'apprentissage en cliquant sur le bouton **apprentissage** ou en utilisant le raccourci (`L`).
2. Sur l'image de la caméra apparaissant à l'écran, vous pouvez voir deux zones plus claires et deux nombres. En plaçant le robot à différents endroits, plus ou moins près des murs de l'arène, déterminez comment ces nombres sont liés à ces deux zones de l'image. Ces nombres correspondent à la luminosité moyenne de ces deux zones. Les valeurs sont comprises entre 0 (noir) et 255 (blanc).
3. Au centre de l'écran est présent un graphique sur lequel se déplace un point. En plaçant le robot à différents endroits, déterminez comment la position du point est affectée par les deux valeurs issues de l'image. L'abscisse du point est déterminée par la valeur de luminosité dans la partie gauche, et son ordonnée est déterminée par la luminosité dans la partie droite. Les valeurs de luminosité comprises entre 0 et 255 sont divisées par 255 pour obtenir des coordonnées entre 0 et 1.
4. Indiquez comment placer le robot de manière à ce que ce point se retrouve dans chacun des 4 coins du graphique.

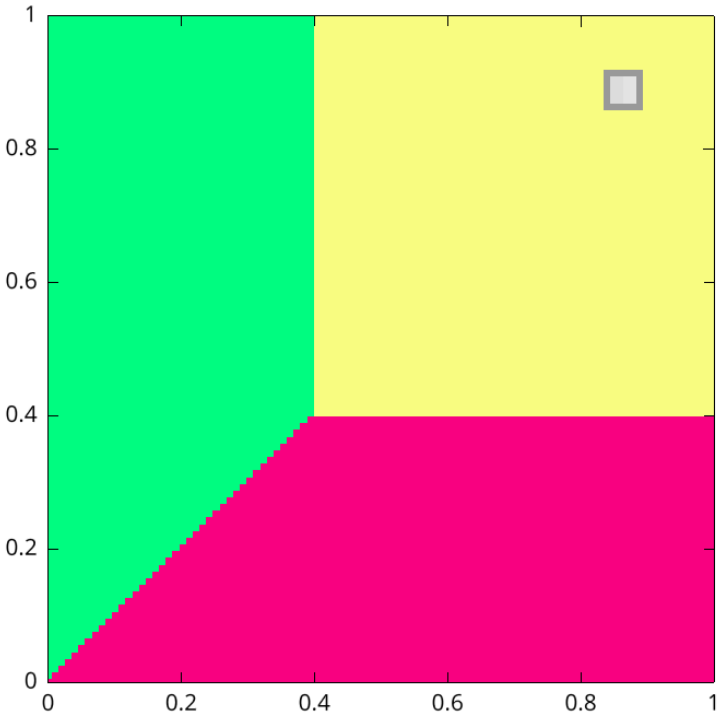
— Pour s'approcher du coin en bas à gauche (coordonnées $(0, 0)$), il faut que les deux zones de l"image soient très sombres : on peut placer le robot face à un mur.
— Pour le coin en haut à droite (coordonnées $(1, 1)$), il faut au contraire que les deux zones de l'image soient très lumineuses : on peut placer le robot face au centre de l'arène.
— Pour le coin en bas à droite (coordonnées $(1, 0)$), il faut que l'image soit lumineuse dans sa partie gauche et sombre dans sa partie droite : on peut placer le robot de sorte à toucher un mur du côté droit.
— Pour le coin en haut à gauche (coordonnées $(0, 1)$), il faut que l'image soit sombre dans sa partie gauche et lumineuse dans sa partie droite : on peut placer le robot de sorte à toucher un mur du côté gauche.

1.3 Visualiser le programme du robot

1. Dans l'onglet **IA**, changez le paramètre **algorithme** et sélectionnez **code python**. Créez un nouveau fichier, enregistrez-le à l'emplacement de votre choix, puis ouvrez-le avec un éditeur de texte, par exemple Notepad++.
2. Nous allons maintenant programmer le robot depuis ce fichier python en modifiant la troisième fonction, appelée `take_decision`. Il ne faudra pas oublier d'enregistrer le fichier python après chaque modification, afin qu'elles soient bien prises en compte par le logiciel !
3. Dans un premier temps, observez les messages affichés par l'instruction `print` dans la console du logiciel **AlphAI**. Il s'agit des valeurs du paramètre `sensors`, qui correspondent aux valeurs mesurées par les capteurs du robot. Quelle opération mathématique permet d'obtenir les valeurs affichées dans la console à partir des valeurs affichés sur l'image de la caméra dans **AlphAI** ? Les valeurs sont simplement divisées par 255 comme précédemment, afin d'obtenir des valeurs entre 0 et 1 qui correspondent aux coordonnées du point sur le graphique.
4. Dans le logiciel AlphAI, activez le mode **autonome** (`barre espace`). Quelle est l'action choisie par le robot ? Le robot choisit systématiquement l'action *pivoter vers la gauche*.
5. Dans le fichier python, modifiez la valeur renvoyée par la fonction `take_decision`, enregistrez la modification, puis observez l'effet sur le comportement du robot. On observe que la valeur de retour de la fonction `take_decision` détermine l'action choisie par le robot : 0 pour pivoter vers la gauche, 1 pour aller tout droit, et 2 pour pivoter vers la droite (les actions sont numérotées de haut en bas).
6. Vous devriez maintenant avoir compris comment modifier la fonction `take_decision` afin de programmer le robot. Essayez de programmer le comportement suivant : le robot doit aller tout droit dès que possible, mais doit éviter de toucher les parois de l'arène. Voici un exemple de programme pour obtenir ce résultat (la fonction `min` renvoie le plus petit élément d'un tableau) :

```python
def take_decision(sensors: np.ndarray) -> int:
    if min(sensors) > 0.4:
        return 1
    elif sensors[0] < sensors[1]:
        return 2
    else:
        return 0
```

7. Dans le logiciel **AlphAI**, observez le déplacement et les changements de couleur du point dans l'**espace d'états** (le graphique central). Dans l'onglet **Visualisation**, activez le paramètre **arrière-plan**. À quoi correspondent les différentes zones de couleur ? Les zones de couleur sont des *zones de décision*. Elles permettent d'anticiper la décision prise par l'IA lorsque le point actif se trouve dans cette zone.
8. Quelle serait la coloration idéale pour le comportement du robot souhaitée ? Essayez d'obtenir cette coloration en améliorant votre fonction `take_decision`. Voir ci-dessous un arrière-plan *idéal* obtenu avec la fonction définie à la question 6. Prenez le temps d'analyser comment cette fonction produit ce résultat.



2 Algorithme des k plus proches voisins

2.1 Mise en place

1. Dans cette partie, nous n'utiliserons pas le mode **code python**, vous pouvez donc sauvegarder et fermer le fichier python ouvert précédemment.
2. Dans l'onglet **IA**, sélectionnez l'algorithme **k plus proches voisins**.
3. Dans l'onglet **Visualisation**, désélectionnez le paramètre **arrière-plan**.
4. Réactivez le mode apprentissage.
5. Si vous n'êtes pas sûr que votre paramétrage est correct, vous pouvez charger de nouveau les paramètres d'exemple **KNN - Caméra**, afin de revenir aux paramètres de départ.

2.2 Apprentissage supervisé

L'algorithme des k plus proches voisins est un algorithme d'apprentissage supervisé, ce qui signifie que l'IA doit être entraînée à partir de **données étiquetées** fournies par des humains. Dans le cas du robot **AlphAI**, on fournit ces données simplement en pilotant le robot.

1. Pilotez le robot dans l'arène en évitant les murs, jusqu'à avoir environ une centaine de points dans la mémoire d'expérience (voir texte en haut de l'écran). Puis activez le mode autonome. Le comportement du robot est-il satisfaisant ?
2. Au cours de l'apprentissage, des points (plus petits) sont apparus sur le graphique de l'espace d'états. Comment sont déterminés la position et la couleur de chaque point ? La position de chaque point est déterminée par les valeurs de luminosité (comme vu précédemmennt) au moment où l'utilisateur appuie sur une touche. La couleur correspond à l'action choisie par l'utilisateur.
3. Réinitialisez l'apprentissage, puis placez un point jaune (tout droit) et un point vert (pivoter à droite) dans deux coins opposés de l'espace d'états. Sans ajouter de nouveaux points (vous pouvez par exemple désactiver l'apprentissage), placez le robot dans différentes situations et observez le changement de couleur du point principal. Comment le robot prend-il sa décision ? Le robot va toujours choisir l'action correspondant à la couleur du *point mémorisé* **le plus proche** du *point actif*. Les points mémorisés sont les **données d'entraînement** du robot, et le point actif est la **valeur d'entrée** (*input*) de l'algorithme de prise de décision.
4. Activez le paramètre **arrière-plan** afin de vérifier votre hypothèse. Puis refaites un apprentissage de votre robot en prenant soin de bien choisir chaque action, afin d'obtenir des zones de couleur bien définies. Le résultat

est-il plus satisfaisant ? Le fait de visualiser en direct les zones de décision devrait vous permettre de mieux entraîner le robot.

2.3 Le paramètre k

Dans la section précédente, nous avons étudié l'algorithme des k plus proches voisins avec $k = 1$. Nous allons maintenant nous intéresser à l'impact de ce paramètre k : le nombre de voisins.

1. Désactivez la coloration de l'arrière-plan et réalisez un apprentissage avec peu de points (une dizaine environ), répartis le plus uniformément possible sur le graphique de l'espace d'états (n'ajouter un nouveau point que s'il est suffisamment éloigné des autres, voir figure 3). Dans l'onglet **IA**, augmentez progressivement la valeur du paramètre k et observez l'effet sur les prises de décision du robot. Expliquez pourquoi on choisit généralement une valeur impaire pour k. Lorsque $k > 1$, l'algorithme de décision va choisir l'action **majoritaire** parmi les k points les plus proches du point actif. Choisir une valeur impaire permet d'éviter les cas d'égalité entre 2 actions. Dans notre cas, puisque nous avons 3 actions possibles, nous pourrions aussi avoir envie d'éviter les multilples de 3, et donc de choisir par exemple les valeurs $k = 5$ ou $k = 7$.
2. Modifiez de nouveau la valeur du paramètre k, mais cette fois avec la coloration de l'arrière-plan active. Observez l'effet sur les zones de décision.
3. Réalisez maintenant un entraînement du robot avec une centaine de points, et contenant un petit nombre d'erreurs : entre 5% et 10% des points ne sont pas de la bonne couleur. Quelle peut être l'utilité d'augmenter la valeur de k ? Que se passe-t-il lorsqu'on l'augmente trop ? Quelle semble être la valeur donnant le meilleur résultat ? Augmenter la valeur de k permet d'éliminer les zones de décisions correspondant à des erreurs, c'est-à-dire des points de données d'entraînement qui ont la mauvaise couleur. En effet, ces erreurs devraient être minoritaires dans un voisinage proche. En revanche, si on augmente trop la valeur de k, on peut faire reculer ou même disparaître des zones de décisions entières, jusqu'à n'avoir qu'une seule zone de décision correspondant à l'action la plus souvent choisie. Seules les valeurs intermédiaires donnent des résultats satisfaisants au niveau du comportement du robot. Le choix précis de la valeur de k peut permettre d'optimiser le comportement du robot à partir d'un certain jeu de données d'entraînement, mais cet effet est limité. La qualité des données d'entraînement est donc primordiale pour obtenir un bon résultat.

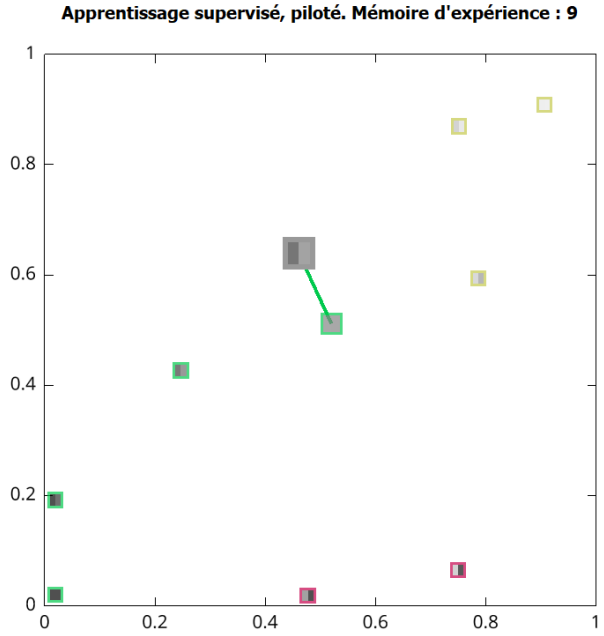


FIGURE 3 – L'espace d'états après un apprentissage avec 9 points bien séparés

3 PROGRAMMATION DE L'ALGORITHME

3.1 MISE EN PLACE

1. Chargez de nouveau les paramètres d'exemple **Apprentissage supervisé - KNN Caméra** afin de revenir dans la configuration de départ.
2. Nous allons de nouveau utiliser le mode **code python**. Créez un nouveau fichier python que vous pouvez appeler *knn.py*.

3.2 FONCTIONS AUXILIAIRES

Dans un premier temps, nous allons définir des fonctions auxiliaires qui nous serviront pour l'algorithme final. Vous pouvez définir ces fonctions dans le fichier *knn.py*, en-dessous de la fonction `take_decision`. On rappelle que la distance euclidienne d entre deux points A et B de coordonnées (x_A, y_A) et (x_B, y_B) est :

$$d = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$$

1. Définir une fonction `distance` qui prend en paramètres deux tableaux (ou listes) `a` et `b` contenant chacun deux nombres représentant les coordonnées de deux points A et B, et dont la valeur de retour est la distance entre ces deux points. Le prototype de la fonction `distance` doit être : `(List[float], List[float]) -> float`. Voir ci-dessous un exemple de

solution. Attention : pour pouvoir utiliser la fonction `sqrt`, il faut l'importer depuis le module `math` avec l'instruction `from math import sqrt` (on peut aussi l'importer depuis le module `numpy`).

```
def distance(a, b):
    return sqrt((b[0] - a[0])**2 + (b[1] - a[1])**2)
```

2. Testez votre fonction `distance` à l'aide des lignes suivantes (et vérifiez que la valeur affichée est correcte) :

```
a = [0, 0]
b = [1, 2]
print("La distance entre", a, "et", b, "est", distance(a, b))
```

Python doit afficher : `La distance entre [0, 0] et [1, 2] est 2.23606797749979`. La valeur exacte est $\sqrt{5}$.

3. En utilisant la fonction `distance`, définir une fonction `all_distances` qui prend en paramètres un point `a` (c'est-à-dire une liste de taille 2) et une liste de points `point_list`, et qui renvoie la liste des distances entre le point A et chacun des éléments de la liste `point_list`. Le prototype de la fonction `all_distance` doit être : `(List[float], List[List[float]]) -> List[float]`. Attention à l'ordre dans lequel on place les paramètres lors de la définition et lors de l'appel d'une fonction.

```
def all_distances(a, point_list):
    distances = []
    for point in point_list:
        distances.append(distance(a, point))
    return distances
```

4. Testez votre fonction `all_distances` à l'aide des lignes suivantes (et vérifiez que la valeur affichée est correcte) :

```
a = [0.4, 0.7]
train_sensors = [[0, 0], [0, 1], [1, 0], [1, 1]]
distances_list = all_distances(a, train_sensors)
print("Liste des distances :", distances_list)
```

```
Liste des distances : [0.8062257748298549, 0.5, 0.9219544457292886,
0.6708203932499369]
```

5. Définir une fonction `find_minimum` qui prend en paramètre une liste de nombres et qui renvoie l'**indice du plus petit élément** de la liste. Son prototype doit être : `(List[float]) -> int`. Attention au cas où la liste est vide. Voici une première façon de faire en utilisant une boucle `for` :

```
def find_minimum(nb_list):
    if len(nb_list) == 0:
        return None
    i_min = 0
```

```
    for i in range(len(nb_list)):
        if nb_list[i] < nb_list[i_min]:
            i_min = i
    return i_min
```

Voici une autre façon en utilisant les fonctions `index` et `min` de python, ainsi que la syntaxe `try ... except` pour gérer les exceptions :

```
def find_minimum(nb_list):
    try:
        return nb_list.index(min(nb_list))
    except ValueError:
        return None
```

6. Testez votre fonction `find_minimum` à l'aide de la ligne suivante (et vérifiez que la valeur affichée est correcte) :

```
print("Indice du minimum :", find_minimum(distances_list))
```

```
Indice du minimum : 1
```

3.3 Fonction de décision

Maintenant que nous avons défini toutes les fonctions auxiliaires nécessaires, nous allons pouvoir créer la fonction `nearest_neighbor_decision` qui accepte 3 paramètres : `train_sensors`, une liste de points représentant les données d'entraînement du robot ; `train_decisions`, une liste d'entiers représentant les actions associées à chacun des points de `train_sensors` ; et un point `a`, représentant les valeurs des capteurs (les 2 valeurs de luminosité fournies par la caméra). On rappelle qu'un point est représenté par une liste de deux nombres décimaux qui sont ses coordonnées. Sa valeur de retour doit être l'indice de l'action choisie par l'algorithme de décision. La fonction `nearest_neighbor_decision` doit donc avoir pour prototype : `(List[List[float]], List[int], List[float]) -> int`.

1. Définir en 3 lignes la fonction `nearest_neighbor_decision`, selon les 3 étapes suivantes :
   — Calculer les distances entre le point `a` et chacun des points de la liste `train_sensors`.
   — Trouver l'indice de la plus petite de ces distances.
   — Renvoyer l'action correspondant au point d'entrainement le plus proche du point `a`.

   On peut écrire en trois lignes :

```
def nearest_neighbor_decision(train_sensors, train_decisions, a):
    distances = all_distances(a, train_sensors)
    i_min = find_minimum(distances)
    return train_decisions[i_min]
```

Cependant, dans le cas où `train_sensors` est vide, la variable `i_min` vaudra `None`, et la dernière ligne va provoquer une erreur. Il vaut donc mieux anticiper ce cas grâce à une condition `if` :

```
def nearest_neighbor_decision(train_sensors, train_decisions, a):
    distances = all_distances(a, train_sensors)
    i_min = find_minimum(distances)
    if i_min is None:
        return 0
    else:
        return train_decisions[i_min]
```

2. Testez votre fonction `nearest_neighbor_decision` à l'aide des lignes suivantes (et vérifiez que la valeur affichée est correcte) :

```
train_decisions = [0, 2, 0, 1]
decision = nearest_neighbor_decision(train_sensors,
                                     train_decisions, a)
print("Décision :", decision)
```

```
Décision : 2
```

3.4 Utiliser les données mémorisées par le robot

1. Dans le fichier *knn.py*, définissez deux variables globales `train_sensors` et `train_decisions` dont la valeur initiale est la liste vide `[]`.
2. Recopiez la version de la fonction `learn` ci-dessous, qui permet de stocker les données `X_train` et `y_train` mémorisées par le robot dans les variables `train_sensors` et `train_decisions` afin de pouvoir les utiliser dans la fonction `take_decision`.

```
def learn(X_train, y_train):
    global train_sensors, train_decisions
    train_sensors = X_train
    train_decisions = y_train
    return 0
```

3. Modifiez la fonction `take_decision` afin que le robot prenne des décisions selon l'algorithme du plus proche voisin (utilisez votre fonction `nearest_neighbor_decison`). Dans le cas où les les variables `train_sensors` et `train_decisions` sont vides, renvoyer par défaut la valeur 0 (tourner à gauche). Si on a bien géré le cas *liste vide* dans `nearest_neighbor_decision`, on peut simplement écrire :

```
def take_decision(sensors):
    return nearest_neighbor_decision(train_sensors,
                                     train_decisions, sensors)
```

Sinon, il faut ajouter une condition `if` :

```
def take_decision(sensors):
    if len(train_sensors) == 0 or len(train_decisions) == 0:
        return 0
    else:
        return nearest_neighbor_decision(train_sensors,
                                         train_decisions, sensors)
```

4. Réalisez un apprentissage du robot et vérifiez qu'il se comporte bien. Vous pouvez utiliser la coloration de l'arrière-plan.

3.5 Utiliser des images complètes

Jusqu'à présent, nous avons utilisé l'algorithme des k plus proches voisins avec des données capteurs qui ne comportaient que deux valeurs (la luminosité à gauche et à droite). C'est ce qui nous a permis de visualiser l'espace des états en deux dimensions. Nous allons maintenant programmer ce même algorithme avec des images complètes et en couleur. Dans cette situation, il ne sera malheureusement pas possible de visualiser l'espace d'état car il comporte plus de deux dimensions, mais le principe de l'apprentissage supervisé reste valide.

1. Améliorez votre fonction `distance` afin de pouvoir calculer une distance entre deux images de mêmes dimensions. Appliquez pour cela le principe de la distance euclidienne : il faut prendre la racine carré de la somme des carrés des différences entre les coordonnées (qui sont ici les valeurs des pixels). Voici une fonction qui calcule une distance entre deux tableaux de nombres de même longueur. Attention : si les tableaux ne sont pas de même longueur, cette fonction va provoquer une `IndexError`.

   ```
   def distance(a, b):
       square_sum = 0
       for i in range(len(a)):
           square_sum += (b[i] - a[i])**2
       return sqrt(square_sum)
   ```

2. Dans **AlphAI**, dans l'onglet **Capteurs**, sélectionnez une résolution de caméra `16x12`, et désactivez le traitement d'image (*précalcul*) afin d'obtenir des images en couleurs. Vérifiez que votre programme permet à votre robot d'apprendre à partir d'images complètes.
3. L'utilisation d'images complètes permet d'apprendre au robot à faire des tours d'arènes de course. Parvenez-vous à obtenir un comportement satisfaisant ? On peut constater que plus les données d'entrée sont complexes, plus il faut de points pour entraîner convenablement le modèle.

3.6 Modifier k, le nombre de voisins

Jusqu'à présent, nous avons programmé l'algorithme des k plus proches voisins avec $k = 1$. Pour pouvoir modifier la valeur de k, il va falloir améliorer plusieurs des fonctions que nous avons programmées.

1. Définir une fonction `find_k_minima`, qui prend en paramètres une liste de nombres `numbers_list` et un entier `k` et qui renvoie une liste contenant les indices des `k` plus petits éléments de la liste. Dans le cas où la liste contient moins de `k` éléments, on pourra renvoyer les indices de tous ces éléments (et ainsi renvoyer une liste de moins de `k` éléments au total). **Conseil** : créer une nouvelle liste contenant des couples `(valeur, indice)`, puis trier cette liste. Voici une version qui utilise des boucles `for`, ainsi que les méthodes `append` et `sort` :

```
def find_k_minima(numbers_list, k):
    nb_idx_list = []
    for i in range(len(numbers_list)):
        nb_idx_list.append((numbers_list[i], i))
    nb_idx_list.sort()  # tri de la liste
    minima = []
    for i in range(min(k, len(numbers_list))):
        minima.append(nb_idx_list[i][1])
    return minima
```

   On peut écrire une version plus concise en utilisant les définitions de listes en compréhension, la fonction `enumerate`, et les tranches (*slices*) :

```
def find_k_minima(numbers_list, k):
    nb_idx_list = [(val, idx)
                   for idx, val in enumerate(numbers_list)]
    nb_idx_list.sort()  # tri de la liste
    return [idx for val, idx in nb_idx_list[:k]]
```

2. Définir une fonction `knn` sur le modèle de `nearest_neighbor_decision`, en utilisant cette fois la fonction `find_k_minima`. La variable k pourra être définie et par la suite modifiée dans le code de cette fonction `knn`. Modifiez également la fonction `take_decision` afin d'utiliser cette nouvelle fonction `knn`. On peut commencer par une fonction auxiliaire qui renvoie la valeur majoritaire dans une liste :

```
def majority(nb_list):
    count = [0]*(max(nb_list) + 1)
    for n in nb_list:
        count[n] += 1
    return count.index(max(count))
```

   Grâce à cette fonction `majority`, la fonction `knn` reste assez concise :

```
def knn(train_sensors, train_decisions, sensors):
```

```
    k = 5  # valeur de k à modifier si nécessaire
    distances = all_distances(sensors, train_sensors)
    idx_min = find_k_minima(distances, k)
    decisions = [train_decisions[idx] for idx in idx_min]
    return majority(decisions)
```

Pour la fonction `take_decision`, on peut simplement écrire :

```
def take_decision(sensors):
    return knn(train_sensors, train_decisions, sensors)
```

3. Vérifiez que votre programme fonctionne. Pour cela, commencez par utiliser les paramètres de la caméra permettant d'avoir la visualisation en 2D (essayez différentes résolutions et différents modes de traitement d'image, parvenez-vous à retrouver les paramètres utilisés dans le scénario KNN ?) Vérifiez que le changement du paramètre k affecte la coloration de l'arrière-plan.
4. Enfin, vous pouvez paramétrer la caméra afin d'utiliser des images entières et en couleurs afin d'entraîner votre robot à faire des tours de l'arène de course. Modifiez dans votre programme la valeur de k. Quelle valeur de k semble donner le meilleur comportement ? Comment évaluer la performance de l'algorithme en l'absence de visualisation ? Pour évaluer l'algorithme, on peut : compter le nombre de fois où le robot est bloqué ; et en l'absence de blocage, chronométrer le temps nécessaire pour faire un tour de piste.