PYLAB

NeuroSnake

de zéro à l'IA

Comprendre et reconstruire une intelligence artificielle qui apprend à jouer, de la première ligne de code au projet final en Rust.

LE LIVRET COMPLET · PYTHON & RUST

Pourquoi ce livret existe

Une IA qui apprend à jouer à Snake toute seule, ça impressionne. Et puis on se dit que c'est réservé aux gens très forts, et on passe à autre chose. C'est dommage, parce que c'est faux.

Il n'y a pas de magie là-dedans. Juste des idées simples, empilées. Prises une par une, elles sont à ta portée même si tu n'as jamais codé. C'est leur empilement qui impressionne, pas chaque idée. Tout ce livret sert à démonter la pile, puis à te la faire remonter seul.

Je ne saute rien. Si tu débutes, tu es à la bonne page. Si tu codes déjà, tu sauteras les premiers chapitres pour foncer sur le réseau de neurones. Le livret est un escalier, chaque marche tient sur la précédente.

Une seule promesse. À la fin, tu ne sauras pas juste comment marche NeuroSnake. Tu sauras en faire un. Le tien.

Comment t'en servir

Code en lisant. La programmation ne s'apprend pas avec les yeux, elle s'apprend avec les doigts. Chaque bout de code ici est fait pour être tapé, lancé, cassé, réparé. Tape-le à la main, pas de copier-coller. Le copier-coller va vite et n'apprend rien.

Quand ça plante, tant mieux. Un bug, c'est le code qui te parle. Apprendre à l'écouter, c'est la moitié du métier. Je te montrerai les pièges au fur et à mesure, je les ai tous faits avant toi.

On construit le projet deux fois. D'abord en Python, où on se concentre sur les idées sans se battre avec le langage. Ensuite en Rust, pour le rendre vraiment rapide. Tu comprendras alors pourquoi le vrai NeuroSnake est en Rust, et tu auras appris deux langages au passage.

Le chemin

Partie 0. Préparer le terrain

Dix minutes de mise en place, et on n'en reparle plus.

Il te faut un éditeur de code. Pas Word. Prends Visual Studio Code, gratuit, sur Mac, Windows ou Linux. C'est là que tu vas passer tes prochaines heures.

Il te faut Python. Va sur python.org, prends la dernière version, installe. Sur Windows, une seule chose à ne pas rater : coche "Add Python to PATH" en bas de la première fenêtre. Si tu la rates, ta machine ne trouvera pas Python et tu perdras une heure à te demander pourquoi. Je te le dis parce que je l'ai vécu.

Il te faut un terminal. Cette fenêtre noire qui fait peur au début et qu'on adore après. Terminal sur Mac, PowerShell sur Windows.

Maintenant, ton premier programme. Dans VS Code, crée un fichier `bonjour.py`. L'extension `.py` annonce du Python. Dedans, tape :

```
print("Salut. Je vais coder une IA.")
```

`print` veut dire "affiche". Le texte va entre guillemets. Enregistre, puis dans le terminal, place-toi dans le dossier du fichier et tape :

```
python bonjour.py
```

Si la phrase s'affiche, tout marche. Tu viens de donner un ordre à une machine, elle a obéi. Ça paraît rien. C'est tout. Le réseau de neurones n'est qu'un empilement d'ordres comme celui-là.

Si tu vois "python n'est pas reconnu", c'est le PATH, sur Windows, réinstalle en cochant la case. Sur Mac, essaie `python3`. Ce petit accroc au départ, tout le monde y passe.

Partie 1. Les fondations

C'est la partie qu'on bâcle le plus souvent, et je ne vais pas la bâcler. On voit les quelques briques qui composent tous les programmes du monde, le tien comme celui de Google. Il y en a peu. Le dur, ce n'est pas leur nombre, c'est de les assembler. Chaque exemple ira vers notre but : un serpent sur une grille.

Valeurs et variables

Un programme manipule des informations. Un score, une position, un nom. On les appelle des valeurs. `5` est une valeur. `"pomme"` aussi.

Une valeur seule ne sert à rien si on ne peut pas la garder. D'où la variable : une boîte étiquetée où on range une valeur pour la retrouver.

```
score = 0
```

Pages 3 à 30

Découper en modules

Premier réflexe Rust. On range le projet en plusieurs fichiers, un par grand morceau, comme dans ton pack code.

```
mod snake;
mod nn;
mod evolution;
mod renderer;
```

En haut de `main.rs`, ces lignes annoncent les autres fichiers, qu'on appelle des modules. Chaque `mod` correspond à un fichier `.rs`. Plus carré que le gros fichier Python unique, et indispensable dès qu'un projet grossit.

Le cerveau, pour de vrai

Reprenons le réseau de la Partie 4, et écrivons-le entièrement en Rust. D'abord les tailles et un peu de hasard.

```
use rand::Rng;

const ENTREES: usize = 24;
const CACHES:  usize = 16;
const SORTIES: usize = 4;

fn poids_hasard() -> f32 {
    rand::thread_rng().gen_range(-1.0..1.0)
}

fn relu(x: f32) -> f32 {
    if x > 0.0 { x } else { 0.0 }
}
```

Les trois constantes sont les mêmes qu'en Python, mais typées en `usize` car elles serviront d'indices et de tailles. `poids_hasard` tire un décimal entre moins un et un, en utilisant la bibliothèque rand. La notation `-1.0..1.0` est l'intervalle, l'équivalent du `range` Python. Et `relu` est notre petit pli, identique, juste avec des accolades et des `0.0` décimaux au lieu de `0`, parce qu'on est en `f32`.

Maintenant la structure du réseau et sa naissance au hasard.

```
struct Network {
    w1: Vec<f32>,   // entrees -> couche cachee 1
    w2: Vec<f32>,   // couche cachee 1 -> couche cachee 2
    w3: Vec<f32>,   // couche cachee 2 -> sorties
    b1: Vec<f32>,
    b2: Vec<f32>,
    b3: Vec<f32>,
}

impl Network {
    fn nouveau_hasard() -> Network {
        Network {
            w1: (0..CACHES * ENTREES).map(|_| poids_hasard()).collect(),
            w2: (0..CACHES * CACHES ).map(|_| poids_hasard()).collect(),
            w3: (0..SORTIES * CACHES).map(|_| poids_hasard()).collect(),
            b1: (0..CACHES ).map(|_| poids_hasard() * 0.5).collect(),
            b2: (0..CACHES ).map(|_| poids_hasard() * 0.5).collect(),
            b3: (0..SORTIES).map(|_| poids_hasard() * 0.5).collect(),
```

```
                Color::from_rgba(255, 170, 170, 255));   // reflet
}
```

On place le cercle au centre de la case. Le `battement` utilise un sinus du temps pour faire osciller très légèrement la taille, un détail qui rend la pomme moins figée. Le deuxième cercle, plus petit, plus clair, décalé vers le haut-gauche, simule un reflet de lumière et donne du volume. Ces petites touches ne coûtent rien et changent tout dans la sensation de soin.

Couleurs et finitions

Un mot sur l'esthétique, parce que c'est ce qui sépare un projet d'élève d'un projet qu'on a envie de regarder. Choisis une palette restreinte et tiens-t'y. Le vrai NeuroSnake tourne autour d'un vert sombre de fond, d'un vert vif pour le serpent, d'un rouge pour la pomme, d'un bleu pour les accents de l'interface. Quatre couleurs, pas quinze. Une palette serrée fait toujours plus pro qu'un arc-en-ciel.

```
const FOND:    Color = Color { r: 0.13, g: 0.22, b: 0.08, a: 1.0 };
const SERPENT: Color = Color { r: 0.37, g: 0.69, b: 0.29, a: 1.0 };
const ACCENT:  Color = Color { r: 0.38, g: 0.50, b: 0.89, a: 1.0 };
```

Définir ses couleurs comme des constantes en haut du fichier, c'est se donner une charte, et pouvoir changer toute l'ambiance du jeu en modifiant trois lignes. Tu retrouves exactement ces constantes au début de ton `main.rs`. Note qu'ici les composantes vont de zéro à un plutôt que de zéro à 255, c'est l'autre façon d'écrire une couleur en macroquad, les deux coexistent.

La boucle de rendu complète

On rassemble. Voilà à quoi ressemble la boucle qui dessine une partie, image après image.

```
#[macroquad::main(window_conf)]
async fn main() {
    let mut jeu = Snake::nouveau();
    let mut dernier_saut = get_time();

    loop {
        // logique : avancer d'un cran tous les TICK
        if get_time() - dernier_saut >= TICK {
            jeu.avancer();
            dernier_saut = get_time();
        }

        // affichage : à chaque image
        let avancement = ((get_time() - dernier_saut) / TICK).min(1.0) as f32;
        clear_background(FOND);
        dessiner_plateau();
        dessiner_pomme(jeu.pomme, get_time() as f32);
        dessiner_serpent_fluide(&jeu, avancement);
        next_frame().await;
    }
}
```

Regarde la séparation, elle est belle et elle est importante. La logique du jeu n'avance que quand un TICK s'est écoulé, par sauts nets. L'affichage, lui, se refait à chaque image, soixante fois par seconde, et utilise l'avancement pour dessiner le serpent en glissement


Pages 42 à 55