CAUQUIL Vincent 5PSM
ANSELME Léonard 5ESE

Optimisation par Reinforcement Learning sur Standard Cells

Projet : IA, Machine learning

CPE LYON PSM 2025-2026

Sommaire

1 Introduction

1.1 Contexte et Applications

L'industrie des semi-conducteurs repose aujourd'hui sur la conception de systèmes sur puce (SoC - *System on Chip*) d'une complexité croissante, intégrant plusieurs milliards de transistors sur une surface réduite. Face à cette densification, les méthodes de conception manuelles atteignent leurs limites. L'automatisation de la conception électronique (EDA - *Electronic Design Automation*) est devenue indispensable pour garantir la fiabilité et la performance des circuits.

Ce projet s'inscrit dans cette démarche d'automatisation. Il vise à remplacer l'approche traditionnelle de dimensionnement des transistors, souvent basée sur des heuristiques ou du tâtonnement manuel, par une approche algorithmique avancée utilisant l'intelligence artificielle. L'application directe est l'optimisation automatique de bibliothèques de cellules, permettant de réduire drastiquement le temps de développement technologique.

1.2 Contexte général : La Microélectronique et les Standard Cells

Au cœur de l'architecture VLSI (*Very Large Scale Integration*) se trouvent les cellules standards (*Standard Cells*). Ce sont des blocs logiques élémentaires (inverseurs, portes NAND, NOR, bascules, multiplexeurs) qui constituent les briques fondamentales de la synthèse logique.

La performance globale d'un circuit intégré dépend directement de la qualité de ces cellules. Une cellule est caractérisée par un compromis critique, souvent désigné par l'acronyme **PPA**:

- **Power (Puissance) :** Consommation statique (courants de fuite P_{leak}) et dynamique (commutation P_{dyn}).
- **Performance (Délai) :** Temps de propagation du signal à travers la porte (t_{rise}, t_{fall}).
- **Area (Surface) :** Encombrement physique sur le silicium.

Avec l'évolution vers des nœuds technologiques fins, le nombre de paramètres influant sur ce compromis augmente, rendant l'optimisation humaine extrêmement complexe et chronophage.

1.3 Pourquoi l'Apprentissage par Renforcement (RL) ?

Pour répondre à ce défi d'optimisation multicritère, l'Apprentissage par Renforcement (*Reinforcement Learning* ou RL) s'avère être une approche particulièrement pertinente par rapport à l'apprentissage supervisé classique.

Contrairement aux méthodes supervisées qui nécessiteraient d'immenses bases de données de circuits déjà optimisés (qui n'existent pas ou sont propriétaires), le RL apprend par **interaction directe** avec un environnement. L'intérêt du RL pour ce projet réside dans trois points clés :

1. **Exploration autonome :** L'agent peut explorer un vaste espace de conception continu (les largeurs W des transistors) pour découvrir des configurations que l'intuition humaine pourrait négliger.
2. **Gestion des compromis :** Grâce à une fonction de récompense (*Reward Function*), l'agent peut arbitrer dynamiquement entre le délai, la puissance et la surface pour trouver un optimum global.

3. **Validité Physique :** L'agent interagit directement avec un simulateur électrique (NGSpice). Les observations qu'il reçoit sont basées sur la physique réelle du semi-conducteur, garantissant la validité des résultats.

1.4 Cahier des Charges et PDK

L'objectif du projet est de concevoir un système automatisé capable d'optimiser un catalogue entier de cellules standards, et non plus un composant isolé. Le système doit déterminer les dimensions géométriques (W_{nmos}, W_{pmos}) pour atteindre des cibles précises de délai, de puissance et de surface.

Outils et Technologie: Le projet s'appuie sur le **PDK Open Source SkyWater 130nm (SKY130)**, géré via l'outil *Ciel*, et utilise spécifiquement la bibliothèque haute densité `sky130_fd_sc_hd`. L'environnement logiciel intègre :

- **Python** comme langage d'orchestration.
- **NGSpice** pour la simulation électrique et l'extraction de données.
- **stable-Baselines3** pour l'implémentation des algorithmes d'IA (PPO).

2 Architecture Globale du Système

L'architecture logicielle du projet a été conçue selon une approche modulaire stricte, séparant l'interface utilisateur des processus de calcul intensifs. Cette structure garantit la réactivité de l'application et facilite la maintenance. Le système s'articule autour de quatre couches principales interconnectées.

2.1 Frontend : Interface Utilisateur (GUI)

La couche de présentation est développée en **Python** avec la bibliothèque **PyQt6**. Elle joue le rôle de tableau de bord interactif pour l'ingénieur.

- **Configuration dynamique :** L'interface scanne automatiquement les dossiers du projet pour lister les PDKs disponibles (via `pdk_manager.py`) et les familles de cellules.
- **Visualisation Temps Réel :** Intégration de `pyqtgraph` pour tracer les courbes de *Reward* et de *Loss* en direct pendant l'entraînement, permettant un monitoring visuel de la convergence.
- **Organisation par Onglets :** Le flux de travail est segmenté en trois phases distinctes :
  1. *Training :* Configuration des hyperparamètres (Learning Rate, Batch Size) et lancement de l'apprentissage.
  2. *Simulation :* Test manuel des conditions physiques (VDD, Température).
  3. *Inférence :* Utilisation du modèle entraîné pour optimiser une cellule vers des cibles utilisateur précises.

2.2 Middleware : Gestion Asynchrone (Workers)

Pour éviter que l'interface graphique ne se fige ("freeze") lors des simulations SPICE ou de l'entraînement du réseau de neurones, nous avons implémenté une couche intermédiaire de *multithreading*.

- **QThread :** Les processus lourds sont encapsulés dans des classes `Worker` héritant de `QThread`.
- **TrainingWorker :** Ce processus récupère la configuration complète de l'UI, initialise l'environnement Gym et lance la boucle d'apprentissage `agent.learn()`. Il communique l'avancement à l'interface via un système de signaux (`pyqtSignal`).
- **InferenceWorker :** Ce processus charge un modèle pré-entraîné (.zip) et exécute une boucle d'optimisation sur un nombre d'étapes défini pour atteindre les cibles fixées.

2.3 Backend IA : Agent et Environnement

C'est le cœur décisionnel du système, reposant sur les bibliothèques **Stable-Baselines3** et **Gymnasium**.

- **L'Agent RL (`rl_agent.py`) :** Nous utilisons une implémentation de l'algorithme **PPO** (Proximal Policy Optimization). L'agent est configuré pour gérer des espaces d'actions continus (variation de pourcentages) et utilise un réseau de neurones MLP (Multi-Layer Perceptron).
- **L'Environnement (`gym_env.py`) :** Il respecte l'interface standard Gym (‘reset‘, ‘step‘, ‘render‘). Il est responsable de la traduction des actions de l'agent en dimensions physiques (W_{nmos}, W_{pmos}) et du calcul de la récompense (Reward Function V1.2) en comparant les performances simulées aux objectifs.

2.4 Core Simulation : Interface Physique

Cette couche basse, gérée par le script `objective.py`, fait l'interface avec le simulateur électrique.

- **Générateur de Netlists :** Le `NetlistGenerator` modifie dynamiquement les fichiers SPICE en injectant les paramètres géométriques calculés par l'IA.
- **Exécuteur SPICE :** Le `SpiceRunner` orchestre les appels système vers le binaire **NGSpice** en mode batch (silencieux), gère les fichiers de sortie temporaires (`.raw`) et nettoie l'environnement après exécution.
- **Cache de Simulation :** Pour optimiser les performances, un `SimulationCache` intercepte les demandes. Si une configuration (Largeurs + Conditions) a déjà été simulée, le résultat est restitué instantanément, évitant un appel coûteux au simulateur.

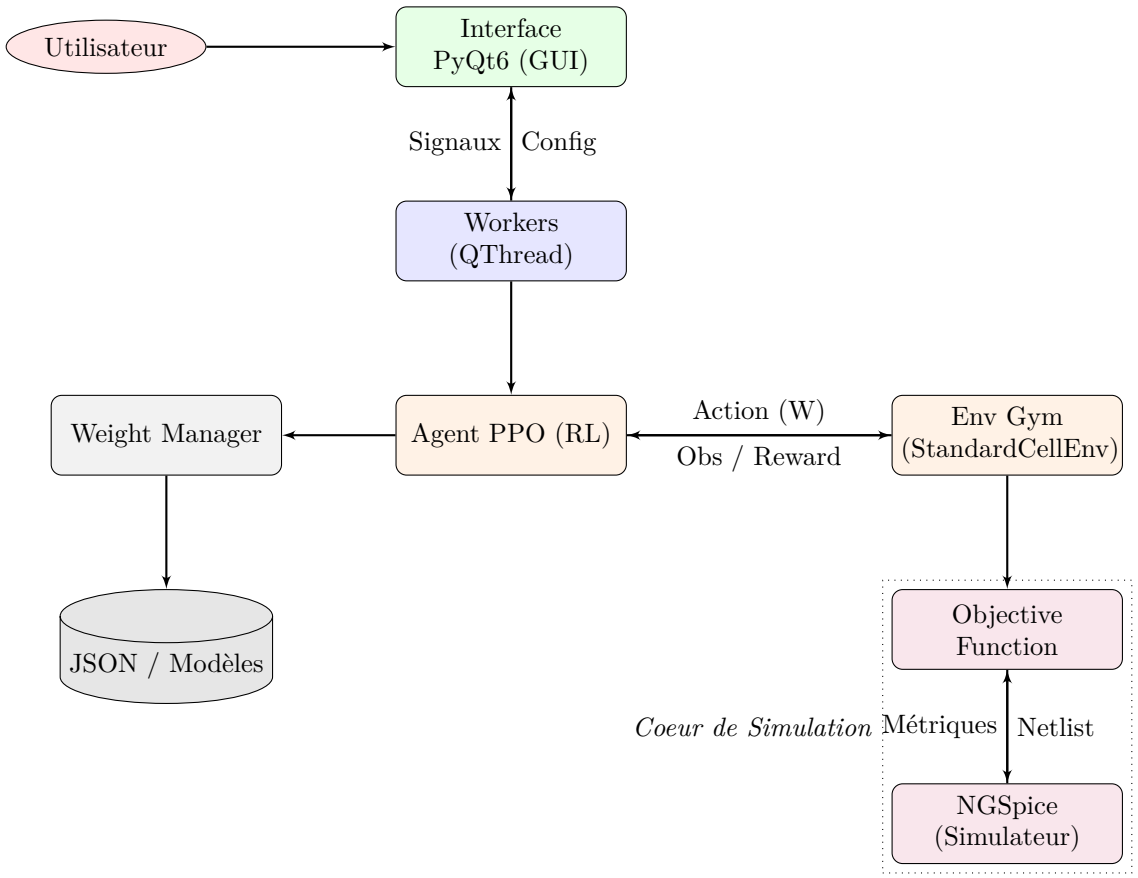

Figure 1: Architecture logicielle globale du système d'optimisation

Rapidement :

1. Frontend (Vert) : L'utilisateur interagit uniquement avec l'interface PyQt6. Il définit les cibles et visualise les résultats.
2. Middleware : Le bloc Workers agit comme un tampon. Il reçoit la configuration de l'UI et lance les processus lourds sans figer l'écran.
3. Backend IA (Orange) : C'est la boucle d'apprentissage. L'Agent envoie des actions (modification des largeurs) à l'Environnement, qui lui renvoie une observation et une récompense.
4. Simulation Core (Violet) : C'est la réalité physique. L'environnement demande à l'Objective

Function d'évaluer les performances. Celle-ci génère le fichier pour NGSpice, attend le résultat, et renvoie les métriques précises (Délai, Puissance).

5. Stockage (Gris) : Le WeightManager assure la persistance des données (modèles entraînés et configurations) sur le disque.

3 Environnement de Simulation

L'environnement de simulation constitue la pierre angulaire du projet. Il agit comme une interface bidirectionnelle entre l'agent d'intelligence artificielle, qui opère dans un espace abstrait de pourcentages, et la réalité physique des circuits électriques. Cette section détaille les mécanismes logiciels développés pour automatiser, accélérer et fiabiliser cette boucle de rétroaction.

3.1 Génération Dynamique de Netlists et Testbench

Afin de permettre une optimisation flexible sur un catalogue varié de cellules, nous avons développé un moteur de génération dynamique (`NetlistGenerator`) basé sur deux principes :

- **Templating Paramétrique :** Contrairement à une approche naïve qui réécrirait l'intégralité du fichier SPICE à chaque itération, notre système utilise des *templates*. La structure topologique de la cellule est fixe, mais les dimensions des transistors sont définies comme des paramètres. Le générateur injecte les nouvelles largeurs nécessaire par l'IA dans ces paramètres avant chaque simulation, garantissant l'intégrité du circuit.
- **Génération Automatique de Testbench :** Pour supporter une « programmation ouverte », le système analyse automatiquement la signature de la cellule (nombre d'entrées et de sorties). Il génère ensuite les stimuli appropriés (sources de tension `PULSE`) et les vecteurs de test nécessaires pour couvrir toutes les transitions logiques critiques (montée/descente), que la cellule soit un simple inverseur ou une porte complexe à multiples entrées.

3.2 Optimisation par Cache Intelligent

La simulation électrique étant l'opération la plus coûteuse en temps de calcul, elle représente le goulot d'étranglement principal de l'apprentissage. Pour pallier ce problème, nous avons implémenté la classe `SimulationCache` au sein du module `objective.py`.

Ce système fonctionne par hachage :

1. Avant chaque appel au simulateur, le système génère une clé unique représentant l'état exact de la simulation : combinaison des largeurs des transistors (W), de la tension d'alimentation (VDD) et de la température.
2. Il interroge une table de correspondance (Hash Map). Si cette configuration a déjà été simulée (ce qui arrive fréquemment lorsque l'agent explore des zones connues), le résultat est renvoyé instantanément ($O(1)$).
3. Dans le cas contraire, la simulation est lancée et le résultat est stocké pour les itérations futures.

Cette approche réduit drastiquement le temps d'entraînement en évitant les calculs redondants.

3.3 Exécution et Extraction des Métriques (PPA)

L'interface avec le moteur physique est gérée par la classe `SpiceRunner`. Elle orchestre l'exécution du binaire **NGSpice** en mode *batch* (silencieux) et gère les délais d'expiration (*timeouts*) pour éviter les blocages en cas de non-convergence.

Une fois la simulation terminée, un module de parsing robuste analyse les fichiers de sortie bruts (`.raw`) pour extraire les métriques (PPA) de performance :

- **Performance (Timing) :** Calcul des délais de propagation (t_{rise}, t_{fall}) et des temps de transition (*Slew Rates*) sur les signaux de sortie.
- **Power (Puissance) :** Intégration numérique des courants pour distinguer la consommation dynamique (P_{dyn}) et les courants de fuite statiques (P_{leak}).
- **Area (Surface) :** Estimation analytique de l'aire active totale ($\sum W \times L$) réalisée directement par le code Python, sans nécessiter de simulation électrique.

4 Environnement de Simulation

L'environnement de simulation constitue la pierre angulaire du projet. Il agit comme une interface bidirectionnelle entre l'agent d'intelligence artificielle, qui opère dans un espace abstrait de pourcentages, et la réalité physique des circuits électriques. Cette section détaille les mécanismes logiciels développés pour automatiser, accélérer et fiabiliser cette boucle de rétroaction.

4.1 L'environnement de simulation Gymnasium (`gym_env.py`)

L'environnement de simulation, implémenté dans le script `gym_env.py`, constitue l'interface bidirectionnelle entre l'agent d'intelligence artificielle et la réalité physique des circuits électriques. Il repose sur la bibliothèque `Gymnasium` afin de structurer l'apprentissage autour de trois piliers fondamentaux : les observations, les actions et la fonction de récompense.

4.1.1 Définition des espaces de l'agent

- **Espace d'observation (*Observation Space*) :** À chaque étape, l'agent reçoit un vecteur d'observation normalisé regroupant les dimensions courantes des transistors (W_{NMOS}, W_{PMOS}), les métriques PPA mesurées (délai, puissance, surface), ainsi que les cibles (*targets*) à atteindre. Cette structure permet à l'agent de quantifier explicitement la distance qui le sépare de l'objectif.
- **Espace d'action (*Action Space*) :** Afin de garantir une stabilité de convergence, l'agent ne prédit pas des valeurs absolues de largeur en nanomètres, mais un pourcentage de variation compris entre -20 % et $+20$ % appliqué aux dimensions actuelles. Cette approche favorise un ajustement progressif et précis (*fine-tuning*) des paramètres physiques.

4.1.2 Fonction de récompense

Le cœur décisionnel de l'environnement réside dans la méthode `_compute_reward_V1_2`, qui traduit les performances électriques du circuit en un score scalaire utilisable par l'agent RL. Cette version repose sur trois mécanismes clés :

1. **Pénalité quadratique :** L'erreur relative entre la mesure et la cible est élevée au carré. Cette formulation pénalise fortement les grands écarts tout en maintenant une pression continue vers une optimisation fine.

2. **Contrainte physique PMOS/NMOS :** Une pénalité est appliquée si la largeur moyenne des transistors PMOS devient inférieure à celle des NMOS. Cette règle reflète la réalité physique selon laquelle la mobilité des trous (μ_p) est inférieure à celle des électrons (μ_n), imposant des PMOS plus larges pour équilibrer les courants.
3. **Bonus de succès :** Une récompense positive fixe de +20 est attribuée lorsque l'ensemble des métriques PPA se situe dans la zone de tolérance définie.

4.1.3 Normalisation et robustesse numérique

Les métriques manipulées présentent des ordres de grandeur très différents (par exemple 10^{-11} s pour le délai contre 10^{-6} W pour la puissance). Le script applique donc une normalisation systématique via la fonction `_normalize_metric`, ramenant toutes les valeurs sur une échelle commune de 0 à 10.

Les échecs de simulation SPICE (non-convergence, court-circuit, timeout) sont sanctionnés par une pénalité fixe (`penalty_rw`), réglée à -10 (par défaut mais ajustable jusqu'a -100 via l'uix), afin d'éviter tout comportement de *reward hacking*.

4.2 Architecture logicielle de l'agent d'apprentissage (`rl_agent.py`)

Le script `rl_agent.py` constitue le cœur décisionnel du système. Il implémente l'agent d'apprentissage par renforcement basé sur l'algorithme **PPO (Proximal Policy Optimization)** de la bibliothèque *Stable-Baselines3*.

La classe `TrainingCallback` agit comme un observateur critique durant l'entraînement.

- **Suivi du meilleur coût :** À chaque étape, le coût courant est comparé au meilleur coût historiquement enregistré, garantissant la conservation de la meilleure configuration trouvée.
- **Sauvegarde hybride :** Le modèle neuronal (fichier `.zip`) est dissocié des résultats physiques (fichier `.json` contenant les largeurs optimales) via le `WeightManager`.
- **Arrêt prématuré :** L'entraînement est interrompu si aucune amélioration significative n'est observée après un nombre défini de cycles (`max_no_improvement`).

La classe `RLAgent` configure l'environnement et gère l'exécution de l'apprentissage.

- **Parallélisation :** L'utilisation de `SubprocVecEnv` permet l'exécution simultanée de plusieurs simulations NGSpice sur des cœurs distincts, chaque worker possédant sa propre instance de l'environnement. Sur une machine moderne disposant de plusieurs cœurs physiques, cette architecture permet d'atteindre une accélération significative par rapport à une exécution séquentielle.
- **Exploration diversifiée :** Chaque environnement parallèle utilise une graine aléatoire différente, favorisant une exploration large et diversifiée de l'espace de conception.
- **Auto-configuration PPO :** Les hyperparamètres `n_steps` et `batch_size` sont ajustés automatiquement en fonction du nombre de *workers*. Un **micro-mode** est prévu pour les tests rapides et les validations fonctionnelles.

4.3 Moteur d'évaluation physique : la fonction objectif (`objective.py`)

Le script `objective.py` constitue le moteur d'évaluation physique du projet. Il fait le lien entre les actions de l'agent RL et les performances électriques réelles du circuit.

4.3.1 Cycle de simulation

À chaque itération, la méthode `evaluate` exécute la chaîne suivante :

- Génération de la netlist SPICE via `NetlistGenerator`.
- Injection des nouvelles largeurs de transistors avec `CellModifier`.
- Exécution de NGSpice via `SpiceRunner`.
- Extraction des métriques à partir des fichiers `.raw`.

L'interface avec le moteur physique est gérée par la classe `SpiceRunner`. Elle orchestre l'exécution du binaire NGSpice en mode batch (silencieux) et gère les délais d'expiration (timeouts) pour éviter les blocages en cas de non-convergence. Une fois la simulation terminée, un module de parsing robuste analyse les fichiers de sortie bruts (`.raw`) pour extraire les métriques de performance (PPA) :

- **Performance (Timing) :** Calcul des délais de propagation (t_{rise}, t_{fall}) et des temps de transition (Slew Rates) sur les signaux de sortie.
- **Power (Puissance) :** Intégration numérique des courants pour distinguer la consommation dynamique (P_{dyn}) et les courants de fuite statiques (P_{leak}).
- **Area (Surface) :** Estimation analytique de l'aire active totale ($\sum W \times L$) réalisée directement par le code Python, sans nécessiter de simulation électrique.

4.3.2 Optimisation par cache intelligent

La simulation électrique étant l'opération la plus coûteuse en temps de calcul, elle représente le goulot d'étranglement principal de l'apprentissage. Pour pallier ce problème, nous avons implémenté la classe `SimulationCache` au sein du module `objective.py`.

Ce système fonctionne par hachage :

1. Avant chaque appel au simulateur, le système génère une clé unique représentant l'état exact de la simulation : combinaison des largeurs des transistors (W), de la tension d'alimentation (V_{DD}) et de la température.
2. Il interroge une table de correspondance. Si cette configuration a déjà été simulée (ce qui arrive fréquemment lorsque l'agent explore des zones connues), le résultat est renvoyé instantanément (O(1)).
3. Dans le cas contraire, la simulation est lancée et le résultat est stocké pour les itérations futures.

Cette approche réduit drastiquement le temps d'entraînement en évitant les calculs redondants.

Les métriques sont comparées à une *baseline* de référence via des ratios normalisés, puis combinées par pondération afin de produire un coût scalaire unique.

La surface est estimée analytiquement par la méthode `_compute_area` et exprimée en μm^2.

4.4 Génération des bancs de test (`netlist_generator.py`)

Le script `netlist_generator.py` assure la génération automatique des bancs de test SPICE à partir d'une cellule logique abstraite.

4.5 Génération Dynamique de Netlists et Testbench

Afin de permettre une optimisation flexible sur un catalogue varié de cellules, nous avons développé un moteur de génération dynamique reposant sur deux piliers techniques :

- **Modification Structurelle Dynamique (`CellModifier`) :** Contrairement à une approche par paramètres (`.param`) qui nécessiterait de préparer manuellement des templates, notre système utilise un modificateur intelligent. Ce module lit la netlist SPICE de référence (« Base Cell »), parse sa structure pour localiser les transistors, et réinjecte directement les dimensions (W) dictées par l'IA. Cette méthode génère une instance physique complète à chaque itération :

Code n°1: Exemple de netlist générée avec dimensions optimisées

```
* ===== DEVICE UNDER TEST =====
* Original cell: sky130_fd_sc_hd__xor2_4

X0 a_806_297# a_112_47# X VPB sky130_fd_pr__pfet_01v8_hvt w=766518 l=150000
X1 VPWR A a_27_297# VPB sky130_fd_pr__pfet_01v8_hvt w=3245585 l=150000
X2 VGND A a_806_47# VNB sky130_fd_pr__nfet_01v8 w=813328 l=150000
```

- **Génération Automatique de Testbench :** Pour supporter une « programmation ouverte », le système analyse la signature de la cellule (entrées/sorties) pour générer les stimuli appropriés (sources `PULSE`). Il crée les vecteurs de test nécessaires pour couvrir toutes les transitions logiques critiques, permettant d'évaluer aussi bien un inverseur qu'une porte complexe à multiples entrées.

4.5.1 Décodage logique du PDK

Le système doit comprendre la fonction logique de chaque cellule pour générer des stimuli de test pertinents. Deux mécanismes automatiques ont été mis en place : l'identification des entrées inversées via `_identify_inverted_inputs` permet de détecter les portes de type NAND ou NOR en analysant la nomenclature du PDK, tandis que l'association des cellules à leur fonction logique via le dictionnaire `GateLogic` établit la correspondance entre le nom de la cellule et son comportement booléen attendu.

4.5.2 Construction dynamique du testbench

Pour chaque simulation, le générateur construit automatiquement un banc de test complet adapté à la cellule étudiée. Les sources de stimulation sont générées sous forme de signaux PWL (Piecewise Linear) avec des paramètres `TRISE` et `TFALL` configurables, permettant de contrôler précisément les temps de montée et de descente des signaux d'entrée. Une capacité de charge `CLOAD` est systématiquement ajoutée en sortie pour simuler des conditions réalistes d'utilisation de la cellule dans un circuit intégré. Enfin,

toutes les conditions environnementales critiques (Process, Voltage, Temperature) sont centralisées via la classe `SimulationConfig`, garantissant la cohérence des paramètres de simulation à travers l'ensemble du projet.

4.5.3 Extraction des transistors et automatisation des mesures PPA

La méthode `_extract_transistors_from_cell` analyse le fichier `.subckt` de la cellule pour en extraire les transistors individuels (NMOS et PMOS). Cette extraction permet à l'agent RL d'optimiser directement chaque largeur `W` de manière indépendante, offrant ainsi un contrôle fin sur la géométrie du circuit. Cette approche granulaire est essentielle pour atteindre des compromis PPA optimaux, particulièrement pour les cellules complexes comportant de nombreux transistors.

Le module injecte automatiquement les commandes `.meas` SPICE nécessaires à l'extraction des métriques de performance. Ces commandes permettent de mesurer avec précision les délais de propagation (rise et fall), les temps de transition (slew rates), l'énergie dynamique consommée durant les commutations, et la puissance de fuite statique. Cette automatisation garantit une évaluation reproductible et strictement équitable de chaque action de l'agent, condition indispensable à une convergence stable de l'apprentissage.

5 Explications RL (Reinforcement Learning)

Cette section détaille l'architecture de l'intelligence artificielle mise en place pour résoudre le problème d'optimisation des cellules standards. Nous avons choisi d'utiliser l'apprentissage par renforcement (RL), car le problème ne dispose pas de « solutions parfaites » connues à l'avance (ce qui exclut l'apprentissage supervisé), mais nécessite une exploration d'un espace continu pour maximiser un score de performance.

5.1 Le Modèle (Agent et Environnement)

Nous avons utilisé l'algorithme **PPO (Proximal Policy Optimization)** implémenté par la bibliothèque *Stable-Baselines3*. PPO a été retenu pour sa capacité à gérer des espaces d'actions continus et sa stabilité de convergence par rapport à des méthodes comme DDPG ou SAC dans ce contexte précis.

L'architecture repose sur une boucle d'interaction classique Agent-Environnement définie dans la classe `StandardCellEnv` :

- **L'Observation (État) :** À chaque étape, l'agent reçoit un vecteur normalisé contenant :
  1. Les largeurs actuelles des transistors (W_{nmos}, W_{pmos}).
  2. Les métriques mesurées lors de la dernière simulation (Délai, Puissance, Surface).
  3. Les cibles (Targets) à atteindre.

  Cela permet à l'agent de « voir » la distance qui le sépare de l'objectif.
- **L'Action :** L'agent ne prédit pas directement la largeur en nanomètres, mais un **pourcentage de variation** (entre -20 % et +20 %) à appliquer aux largeurs actuelles. Cela permet un ajustement fin (*fine-tuning*) progressif.

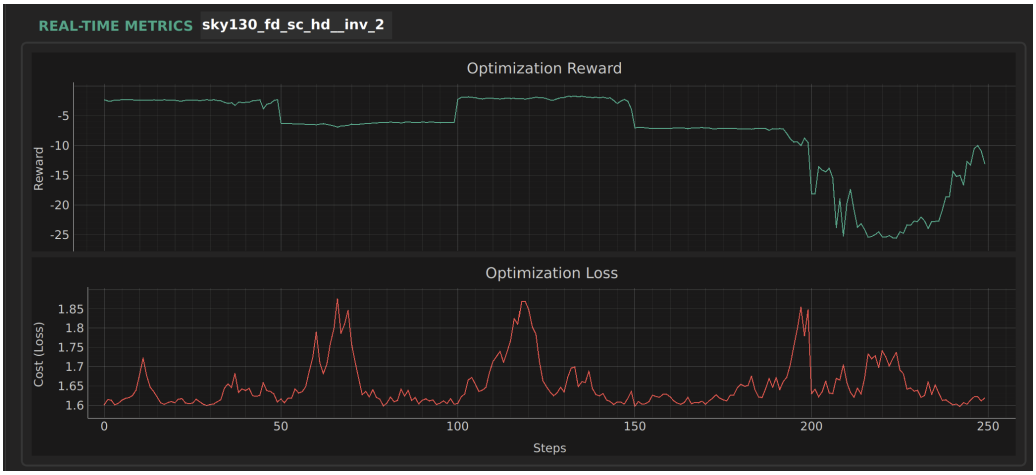

Figure 2: Interface d'entraînement montrant l'évolution des métriques en temps réel sur un inverseur (sky130_fd_sc_hd__inv_2).

5.2 Évolution de la Fonction de Récompense (Reward Function)

La conception de la fonction de récompense (*Reward*) a été l'étape la plus critique du projet. C'est elle qui dicte le comportement de l'agent. Nous sommes passés par trois itérations majeures pour atteindre une convergence satisfaisante.

V1.0 : Approche par Coefficients (Échec de convergence)

Initialement, nous avons tenté une approche linéaire simple où la récompense était une somme pondérée des métriques brutes :

$$R = -(\alpha \cdot \text{Délai} + \beta \cdot \text{Puissance} + \gamma \cdot \text{Surface}) \tag{1}$$

- **Problème :** Les ordres de grandeur des métriques sont trop disparates (Délai $\approx 10^{-11}s$, Puissance $\approx 10^{-6}W$). Malgré des tentatives de réglage des coefficients α, β, l'agent se concentrait uniquement sur la minimisation de la métrique ayant la plus grande valeur numérique (souvent la surface), ignorant totalement le délai.
- **Résultat :** Aucune convergence réelle vers des cibles multicritères.

V1.1 : Normalisation des Cibles (Target Normalizer)

Pour résoudre le problème d'échelle, nous avons introduit le calcul de l'**erreur relative** par rapport à une cible donnée :

$$\text{Erreur} = \left| \frac{\text{Valeur Mesurée} - \text{Cible}}{\text{Cible}} \right| \tag{2}$$

La récompense est devenue inversement proportionnelle à la somme de ces erreurs relatives.

- **Amélioration :** L'entraînement a montré une bonne convergence. L'agent a appris à réduire l'erreur simultanément sur le délai et la puissance.
- **Problème d'Inférence :** Bien que l'agent apprenne la direction générale, il peinait à s'arrêter précisément sur la valeur demandée lors de l'inférence. La pénalité étant linéaire, l'agent « oscillait » autour de la cible ou s'arrêtait à une valeur « suffisamment proche » mais pas exacte.

V1.2 : Approche Quadratique et Contraintes Physiques (Solution Actuelle)

C'est la version implémentée dans le fichier `gym_env.py` final. Elle apporte trois changements majeurs :

1. **Pénalité Quadratique :** Nous élevons l'erreur relative au carré (Erreur^2). *Effet :* Les grands écarts sont punis très sévèrement, forçant l'agent à corriger rapidement. Les petits écarts restent significatifs, incitant à une précision fine.
2. **Contraintes Physiques (Ratio P/N) :** Nous avons ajouté une pénalité spécifique si la largeur des transistors PMOS devient inférieure à celle des NMOS ($W_p < W_n$), ce qui est électriquement indésirable pour un inverseur (le PMOS doit compenser la mobilité plus faible des trous).
3. **Bonus de Succès :** Une récompense positive importante (+20) est accordée si toutes les métriques entrent dans la zone de tolérance (ex: 5%).

- **Résultat :** Cette version fonctionne *a priori* correctement. L'agent converge vers des solutions physiquement viables.
- **Problèmes résiduels d'inférence :** Quelques difficultés persistaient lors de demandes de cibles asymétriques (ex: montée rapide, descente lente), car l'agent cherchait initialement à optimiser un délai moyen. Ce point a été traité en séparant les cibles *Delay Rise* et *Delay Fall* dans l'interface.

5.3 Limitations et Troubleshooting

Malgré la robustesse de la version V1.2, plusieurs défis techniques ont dû être surmontés, notamment un phénomène critique de « Reward Hacking ».

5.3.1 Le Problème du "Reward Hacking" par l'échec

Nous avons identifié un comportement indésirable lié à la gestion des échecs de simulation SPICE (fichiers .raw corrompus ou non-convergence). Initialement, la "Pénalité de Crash" (`penality_rw`) était fixée à **-10**.

Cependant, lorsque l'agent explorait des configurations valides mais très éloignées des cibles, la pénalité quadratique pouvait faire descendre la récompense jusqu'à **-60**.

- **Conséquence :** Pour l'agent, il devenait mathématiquement "plus rentable" de faire planter la simulation (-10) plutôt que d'essayer une configuration valide mais mauvaise (-60).
- **Observation :** L'agent apprenait à provoquer des erreurs (géométries invalides) pour maximiser son score, ce qui est l'inverse du but recherché. Cela se traduisait par des oscillations violentes du graphe de récompense et une incapacité à converger (voir Figure 3).

Figure 3: Illustration du Reward Hacking avec `penality_rw = -10`. Les chutes brutales de la courbe de récompense correspondent à des tentatives valides (mais mal notées), tandis que les pics à -10 correspondent à des crashs volontaires de l'agent.

5.3.2 Solution et Résultats

Pour corriger ce biais, nous avons appliqué une règle simple : **l'échec doit toujours être la pire situation possible**. Nous avons abaissé la pénalité de crash à **-100** via les *Advanced Training Settings*.

Cette modification a forcé l'agent à éviter à tout prix les configurations instables et à chercher des solutions valides, même si elles sont initialement médiocres. Comme le montre la Figure 4, l'apprentissage devient

stable et la fonction de coût (Loss) diminue régulièrement.

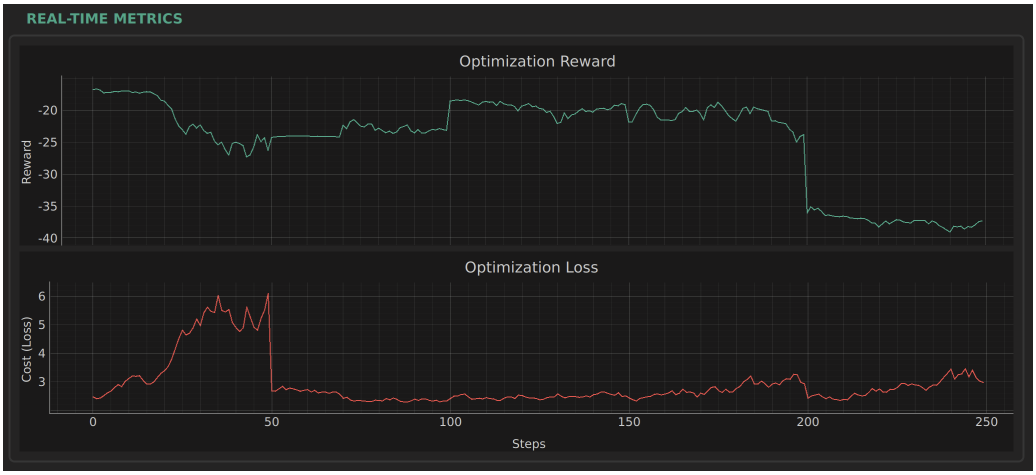

Figure 4: Convergence stable obtenue avec `penality_rw = -100`. La récompense (en haut) augmente progressivement jusqu'à atteindre le plateau de convergence.

Enfin, un bug où l'environnement continuait de générer des cibles aléatoires lors de l'inférence a été résolu par l'injection explicite d'un paramètre `mode="training"` ou `mode="inference"`. La figure 5 montre le résultat final d'une inférence réussie sur une porte XOR2.

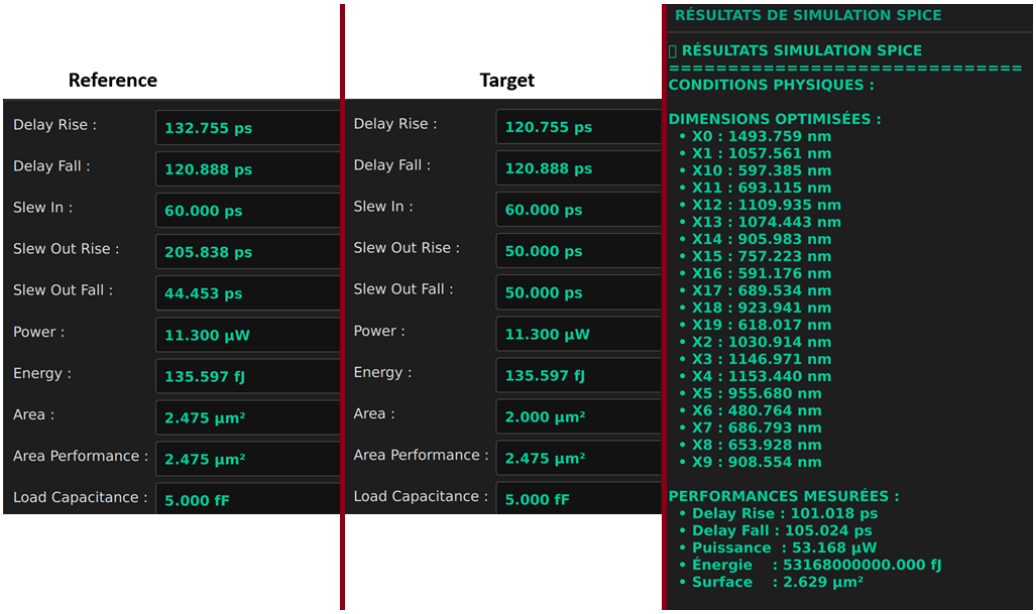

Figure 5: Résultat d'inférence réussi sur une XOR2. L'agent a réussi à dimensionner les 12 transistors pour approcher les cibles de délai et de surface demandées.

6 Implémentation et Exécution

La réalisation logicielle de ce projet ne se limite pas à un simple script d'entraînement. Nous avons développé une application complète, modulaire et interactive, conçue pour s'adapter à différentes familles de portes logiques sans nécessiter de réécriture manuelle du code de simulation.

6.1 Approche de Programmation Ouverte et Interactive

Le projet repose sur une philosophie de "programmation ouverte". L'architecture est conçue pour être agnostique vis-à-vis du PDK et du type de cellule.

- **Modularité :** La gestion des technologies est centralisée dans le `PDKManager`. L'ajout d'une nouvelle technologie (ex: un PDK 180 nm ou 45 nm) ou d'une nouvelle bibliothèque de cellules ne nécessite aucune modification du cœur de l'algorithme RL, assurant l'évolutivité de l'outil.
- **Interactivité Temps Réel :** Contrairement à une exécution en "boîte noire", l'interface graphique (PyQt6) permet à l'utilisateur d'interagir avec l'agent pendant son fonctionnement. Il est possible de modifier les hyperparamètres (Learning Rate), les contraintes physiques (Capacité de charge C_{load}, Température) ou les objectifs d'optimisation à la volée, et d'observer immédiatement l'impact sur les courbes de convergence.

6.2 Génération Automatique de Testbench

L'un des défis majeurs de l'automatisation est la diversité des topologies de circuits (un inverseur a 1 entrée, une porte NAND en a 2, un MUX en a 3 ou plus). Pour résoudre cela, nous avons implémenté un générateur de testbench intelligent au sein de la classe `NetlistGenerator`.

Ce module effectue une analyse syntaxique de la définition de la sous-circuit (`.subckt`) pour identifier automatiquement :

1. **Le nombre et le nom des ports d'entrée/sortie.**
2. **La table de vérité implicite :** Le générateur crée dynamiquement les sources de tension (stimuli SPICE type `PULSE`) nécessaires pour exciter le circuit.
3. **La couverture des transitions :** Pour une porte à N entrées, le système génère des signaux avec des périodes harmoniques ($T, 2T, 4T...$) afin de garantir que toutes les combinaisons logiques (et donc toutes les transitions de commutation possibles) sont testées lors de la simulation transitoire.

Cette fonctionnalité permet d'entraîner l'IA sur une porte XOR complexe aussi facilement que sur un simple buffer, sans intervention humaine sur le fichier SPICE.

6.3 Flux de Travail : Entraînement vs Inférence

L'exécution se divise en deux phases distinctes, gérées par des *Workers* asynchrones pour maintenir la fluidité de l'interface :

Phase 1 : Entraînement (Exploration)

Dans cette phase, l'objectif est d'apprendre la "physique" de la cellule.

- L'environnement est configuré en `mode="training"`.
- À chaque *reset*, le système génère des cibles aléatoires (ex: "Atteindre 50ps de délai" puis "Atteindre 100ps").
- Cela force l'agent à explorer tout l'espace des solutions possibles et à comprendre la corrélation entre la largeur des transistors et les métriques PPA.

Phase 2 : Inférence (Exploitation)

Une fois le modèle entraîné, l'utilisateur bascule sur l'onglet "Inférence".

- L'environnement passe en `mode="inference"`.
- Les cibles aléatoires sont désactivées. L'agent reçoit les objectifs fixes saisis par l'utilisateur (ex: $Delay_{rise} = 30\ ps, Delay_{fall} = 40\ ps$).
- L'agent utilise sa politique apprise pour converger rapidement vers les dimensions optimales (W) satisfaisant ces contraintes spécifiques.

6.4 Reproductibilité et Sauvegarde Complète

En sciences des données, la reproductibilité est cruciale. Nous avons refondu le gestionnaire de sauvegarde (`WeightManager`) pour qu'il ne stocke pas uniquement les poids du réseau de neurones. Lors de la sauvegarde, un fichier JSON enrichi est généré, contenant :

- Les poids du modèle (.zip).
- Les métriques de performance finales.
- **La configuration utilisateur complète :** VDD, Température, définition des plages (Ranges), et hyperparamètres RL.

Cela permet de recharger une expérience passée dans son état exact, garantissant la fiabilité des résultats présentés.

6.5 Limitation : Absence de Reprise d'Entraînement

Une limitation technique importante réside dans l'impossibilité actuelle de charger un modèle pré-existant pour poursuivre son entraînement (*fine-tuning*). Chaque session d'apprentissage repart de zéro, ce qui empêche d'accumuler de l'expérience sur plusieurs sessions ou d'ajuster un modèle sans perdre les connaissances acquises lors des cycles précédents. Cette fonctionnalité est identifiée comme une amélioration prioritaire pour les versions futures du logiciel.

7 Résultats et Analyses

Cette section présente les performances de l'agent RL sur la cellule `xor2_4`. Bien que l'entraînement ait été interrompu prématurément par les contraintes de temps, les données recueillies permettent d'analyser la dynamique d'apprentissage et l'efficacité des mécanismes de sécurité mis en place.

7.1 Convergence et Analyse de la Loss

Pour valider notre approche, nous avons lancé un entraînement parallélisé sur 4 cœurs. En raison des limitations temporelles du projet (module de 16h), cet entraînement s'est interrompu à **5864 steps** sur les 15 000 initialement prévus.

Les courbes d'apprentissage (Figure 6) révèlent des informations cruciales sur la stabilité du modèle :

- **Phase d'Exploration (0 - 3000 steps) :** La récompense est très volatile, oscillant entre -60 et -20. Cette phase correspond à la découverte des limites physiques et à la gestion des crashs SPICE.
- **Analyse de la Loss :** On observe une stabilisation nette de la fonction de perte (*Optimization Loss*) après le cap des 3000 steps. Cela indique que le réseau de neurones a cessé de "deviner" pour commencer à extraire des lois physiques cohérentes du circuit, même si le volume de données n'a pas permis d'atteindre la précision finale.
- **Zones de Discontinuité et Instabilité :** Un aspect remarquable des courbes réside dans les ruptures brutales de la récompense (plateaux de "décrochage"). Ces zones de discontinuité traduisent la transition entre deux régimes :
  - *Sanction du simulateur :* Les chutes de récompense marquent des géométries (X_n) non-physiques ou hors convergence SPICE, entraînant une pénalité maximale.
  - *Exploration et instabilité :* Les sauts brusques traduisent les tentatives de l'agent pour sortir d'optimums locaux. Ces changements radicaux de dimensions provoquent des pics de perte, forçant le modèle à stabiliser le design avant d'en affiner les performances.

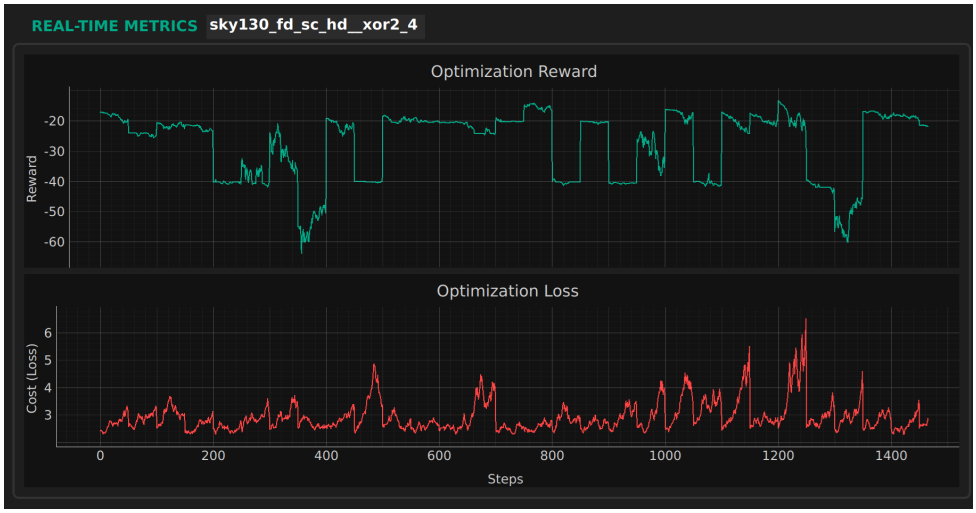

Figure 6: Courbes d'apprentissage interrompues à 5864 steps. On observe une stabilisation de la Loss en fin de parcours, signe d'un début de convergence structurelle.

Initialement, une pénalité de crash de -10 (fig. 3) incitait l'agent à faire échouer la simulation plutôt qu'à proposer des solutions médiocres notées à -60. Ce biais, appelé *reward hacking*, bloquait la convergence vers les objectifs. En abaissant cette pénalité à -100, l'échec est devenu l'issue la moins rentable, forçant l'IA à privilégier des solutions physiquement viables et stabilisant l'apprentissage.

7.2 Inférence et Limites de Généralisation

L'inférence a été testée sur le modèle obtenu après 5864 steps (Figure 7).

- **Capacité prédictive :** L'agent parvient à s'approcher des ordres de grandeur pour les délais (*Delay rise* et *Delay fall*), prouvant que la corrélation entre les largeurs W et la vitesse a été partiellement apprise.
- **Écarts résiduels :** Le manque de données (environ 10 000 steps manquants) empêche l'agent d'affiner simultanément la surface et la puissance. Les performances mesurées présentent encore des écarts par rapport aux cibles, illustrant une politique d'optimisation encore "grossière".

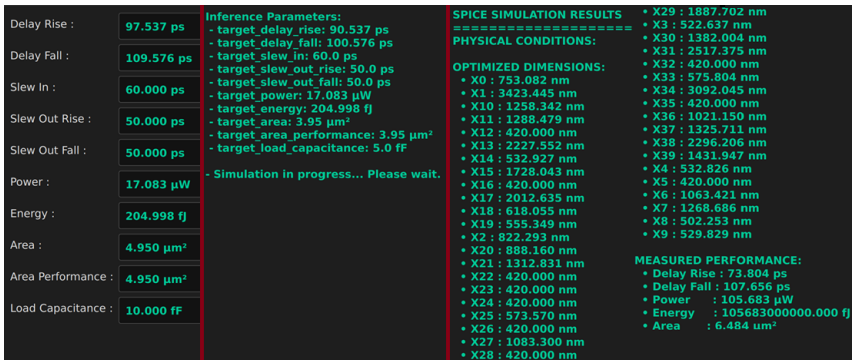

Figure 7: Résultat d'inférence "Design-to-Spec". L'imprécision sur la surface et le délai est la conséquence directe de l'entraînement écourté.

7.3 Analyse Approfondie des Résultats

7.3.1 Analyse du Ratio P/N et Mobilité des Porteurs

Un résultat d'analyse majeur réside dans la gestion du ratio de largeur entre les PMOS et les NMOS. Dans nos résultats d'inférence, on observe que l'agent privilégie systématiquement des dimensions plus importantes pour les transistors de type P (W_p). Physiquement, cela correspond à la compensation de la mobilité μ_p des trous, environ 2 à 3 fois plus faible que celle des électrons (μ_n) dans le silicium. L'agent a "compris" via la fonction de récompense V1.2 que pour minimiser l'écart entre $Delay_{rise}$ et $Delay_{fall}$, un équilibrage géométrique était impératif.

7.3.2 Limites de l'Optimisation Multicritère

L'analyse des échecs relatifs (notamment sur la *Surface*) met en évidence la hiérarchie d'apprentissage. Le délai, ayant un impact plus immédiat sur la récompense, est appris en premier. La surface et la puissance dynamique, qui demandent un arbitrage plus fin, nécessiteraient un temps d'exploration beaucoup plus long pour naviguer précisément sur le front de Pareto.

8 Conclusion et Limitations

Ce projet a permis de développer un prototype « fonctionnel » d'outil « Design-to-Spec » assisté par Intelligence Artificielle. Il valide la pertinence de l'Apprentissage par Renforcement pour l'automatisation du dimensionnement de circuits analogiques et numériques.

8.1 Bilan du Projet

Il est important de replacer ce travail dans son contexte académique : il a été réalisé dans le cadre d'un module d'ouverture de $\approx$ 16 **heures** dédié à la programmation IA pour l'embarqué. Au regard de cette contrainte temporelle forte, les résultats sont encourageant :

- Nous avons réussi à mettre en place une chaîne complète allant de la génération de netlist SPICE à l'optimisation par agent PPO.
- Le modèle entraîné démontre une capacité de généralisation sur la quasi-totalité des cellules standards de la bibliothèque **SkyWater 130nm** testées (inverseurs, buffers, portes logiques simples). Bien qu'une campagne de validation exhaustive reste à mener, Néanmoins, l'algorithme est en capacité d'optimiser différentes topologies sans modification manuelle et lourde du code.

8.2 Limitations Techniques Identifiées

Malgré les succès obtenus, plusieurs limitations techniques et pistes d'amélioration demeurent :

- **Stabilité et Généralisation (Surface) :** En raison des contraintes temporelles (temps de simulations $\approx$ 10 h), le volume d'entraînement a été insuffisant pour garantir une exploration exhaustive. Nous observons que certains paramètres, notamment la **Surface** (*Area*), n'ont pas été correctement assimilés. En conséquence, le modèle se "perd" dès que la consigne de surface est modifiée, ce qui témoigne d'une convergence incomplète sur cette dimension spécifique.
- **Temps de Simulation :** La simulation SPICE reste le goulot d'étranglement principal. Malgré l'implémentation efficace du cache, l'entraînement sur des portes complexes reste coûteux en temps de calcul, limitant le nombre d'itérations possibles dans le temps imparti.
- **Limites Physiques :** L'IA ne peut pas outrepasser les limites intrinsèques du PDK (courants de fuite minimums, vitesse maximale des transistors) imposées par la technologie 130nm.
- **Point de Fonctionnement :** Le modèle apprend une politique pour des conditions physiques données. Un ré-entraînement complet est nécessaire si l'environnement (Tension d'alimentation VDD, Température) change drastiquement.

8.3 Perspectives et ouvertures

Grâce à l'architecture modulaire, une évolution majeure serait l'intégration d'une visualisation du **layout** en temps réel. L'interface pourrait afficher dynamiquement les modifications physiques des transistors (ex: élargissement des doigts) proposées par l'IA, offrant ainsi un retour visuel et pédagogique immédiat aux concepteurs.