

# Projet Diffusion

Charles STEPHANN - Clément MARTIN



Professeur : VAUCHER Rémi  
EPITA - Ing3

11 janvier 2026

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Contexte et Objectifs</b>	<b>3</b>
2.1	Problématique et Contraintes Techniques . . . . .	3
2.2	Objectifs du Projet . . . . .	3
<b>3</b>	<b>Méthodologie technique</b>	<b>4</b>
3.1	Préparation des Données . . . . .	4
3.2	Architecture du Modèle : Simple U-Net . . . . .	4
3.3	Protocole d'Entraînement . . . . .	5
3.4	Processus d'Echantillonnage (Reverse Process) . . . . .	5
<b>4</b>	<b>Résultats et Analyse</b>	<b>6</b>
4.1	Analyse Qualitative des Générations . . . . .	6
4.2	Évaluation de la Généralisation (Mémorisation vs Création) . . . . .	7
4.3	Conclusion des Résultats . . . . .	8

# Introduction

L'objectif de ce projet est l'implémentation d'un modèle de diffusion probabiliste (DDPM) appliqué à la génération d'images. Ce rapport détaille la chaîne de traitement mise en place, depuis le pré-traitement des données brutes jusqu'à l'échantillonnage du bruit. Il expose également les obstacles techniques rencontrés — notamment liés aux contraintes architecturales (U-Net) et matérielles (CPU) — ainsi que les stratégies d'optimisation déployées pour les surmonter.

# Contexte et Objectifs

Une nouvelle fois, l'objectif de ce projet est d'implémenter un modèle de diffusion probabiliste (DDPM) avec les contraintes matérielles qui sont les nôtres (travail sur CPU).

Nous avons choisi de travailler avec un dataset Kaggle de sprites en pixel art de taille 16x16 nommé Pixel Art. Voici un exemple d'images constituant le dataset :



FIGURE 2.1 – Exemple de sprites constituant le dataset

Ce choix de dataset n'est pas anodin. Il répond à une stratégie d'optimisation imposée par nos limitations matérielles. En effet, la dimension réduite des images ( $16 \times 16$  pixels) rend l'entraînement sur CPU envisageable dans un temps raisonnable.

## 2.1 Problématique et Contraintes Techniques

L'entraînement des modèles de diffusion est notoirement coûteux en ressources de calcul. L'architecture standard, basée sur un U-Net avec attention, nécessite généralement des accélérateurs GPU.

Dans notre configuration (exécution sur CPU), nous faisons face aux défis suivants :

- **Lenteur des calculs matriciels** : Les convolutions 2D ne bénéficient pas du parallélisme massif des cœurs Tensor/CUDA.
- **Risque de sur-apprentissage** : Avec un dataset de taille modeste, le modèle risque de mémoriser les sprites plutôt que d'apprendre leur distribution latente.

## 2.2 Objectifs du Projet

Compte tenu de ces contraintes, les objectifs de ce projet se déclinent en trois axes techniques :

1. **Implémentation d'un Pipeline Complet** : Mettre en place la chaîne de traitement de bout en bout, incluant le chargement optimisé des données (fichiers `.npy`), le processus de bruitage (Forward Process) et l'échantillonnage inverse.
2. **Optimisation Architecturale** : Adapter l'architecture U-Net classique (réduction de la profondeur et du nombre de canaux) pour obtenir un modèle capable de converger sur CPU en quelques heures.
3. **Validation de la Génération** : Démontrer la capacité du modèle à générer des sprites inédits mais cohérents, prouvant ainsi que le réseau a appris la structure statistique des données (forme, palette de couleurs) malgré la simplification du réseau.

# Méthodologie technique

## 3.1 Préparation des Données

Le jeu de données utilisé est constitué de 89 400 sprites stockés au format binaire NumPy (.npz).

Afin de gérer efficacement les ressources mémoire lors de l'accès à ce volumineux fichier binaire, nous avons implémenté une classe `SpriteDataset` utilisant le chargement différé (*lazy loading*).

```
self.images = np.load(img_path, mmap_mode='r')
```

L'option `mmap_mode='r'` permet de mapper le fichier en mémoire sans le charger intégralement dans la RAM, les segments de données n'étant lus que lorsqu'ils sont requis par le `DataLoader`.

Les images sources possèdent une résolution native de  $16 \times 16$  pixels avec 3 canaux de couleurs (RGB). Le pipeline de transformation appliqué est le suivant :

- **Conversion en Tenseur** : Permutation des dimensions de (H,W,C) vers le format PyTorch (C,H,W), soit (3,16,16).
- **Normalisation** : Mise à l'échelle des valeurs de pixels de l'intervalle  $[0,255]$  vers  $[-1,1]$  pour s'aligner avec la distribution du bruit gaussien.

$$x_{norm} = \frac{x - 127.5}{127.5} \quad (3.1)$$

Cette normalisation est implémentée via `transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))`.

## 3.2 Architecture du Modèle : Simple U-Net

Le cœur du modèle est un réseau convolutionnel de type U-Net, chargé de prédire le bruit  $\epsilon$  à l'étape  $t$ . L'architecture a été dimensionnée pour traiter des entrées de  $16 \times 16$  pixels.

Le réseau suit une structure Encodeur-Décodeur symétrique avec les caractéristiques suivantes :

- **Profondeur** : 4 niveaux de résolution.
- **Largeur (Canaux)** : Les filtres de convolution augmentent progressivement dans l'encodeur :  $32 \rightarrow 64 \rightarrow 128 \rightarrow 256$ , puis diminuent symétriquement dans le décodeur.
- **Connexions Résiduelles** : Utilisation de blocs résiduels additionnels et de *skip connections* entre l'encodeur et le décodeur pour préserver les détails spatiaux.

L'information du pas de temps  $t$  est injectée dans le réseau via des *Sinusoidal Position Embeddings* de dimension 32. Ces embeddings sont traités par un MLP (Perceptron Multicouche) et additionnés aux cartes de caractéristiques à chaque bloc du réseau, permettant au modèle de moduler son traitement selon le niveau de bruit. Sans ces embeddings, le modèle verrait une image bruitée sans savoir si elle est "très bruitée" (début du processus) ou "peu bruitée" (fin du processus). L'embedding temporel agit comme une "horloge" qui indique au U-Net l'intensité du débruitage à effectuer.

### 3.3 Protocole d’Entraînement

Le modèle a été entraîné sur processeur (CPU) en suivant l’algorithme standard des DDPM.

- **Processus de Diffusion (T)** : Nous avons utilisé un horizon de diffusion de  $T=1000$  étapes, avec un échéancier de bruit (*schedule*) linéaire pour les valeurs  $\beta_t$  allant de  $1e-4$  à  $0.02$ .
- **Fonction de Perte** : MSE Loss
- **Hyperparamètres** :
  - Optimiseur : AdamW
  - Taux d’apprentissage (Learning Rate) :  $1e-4$
  - Taille de lot (Batch size) : 64
  - Durée : 100 époques

La convergence de la Loss confirme la capacité du modèle à apprendre la tâche de débruitage malgré la complexité du dataset et la faible résolution.

### 3.4 Processus d’Echantillonnage (Reverse Process)

L’inversion directe de la diffusion ( $q(x_{t-1}|x_t)$ ) est impossible car elle nécessiterait de connaître la distribution de toutes les images possibles. L’astuce du DDPM est d’approximer cette inversion grâce à un réseau de neurones qui apprend la moyenne conditionnelle de la distribution, ce qui revient concrètement à prédire le bruit  $\epsilon$  ajouté à l’image.

# Résultats et Analyse

## 4.1 Analyse Qualitative des Générations

Le processus d'échantillonnage inverse (Reverse Process) sur  $T=1000$  pas a permis de générer des sprites de  $16 \times 16$  pixels. L'observation de la planche de résultats (Figure 4.1) met en évidence plusieurs points :

1. **Cohérence Sémantique** : Le modèle parvient à générer des objets distincts et reconnaissables : on identifie clairement des formes humanoïdes, des créatures (slimes, monstres), des objets (cartes, orbes) et des équipements (boucliers, casques).
2. **Structure Locale** : Les contours sont généralement respectés. Le modèle a appris la logique du "Pixel Art" où des pixels de bordure plus foncés délimitent souvent les objets.
3. **Artefacts de Bruit** : On note la présence résiduelle d'un léger "bruit de fond" ou de pixels parasites autour de certains objets. Cela est attribuable à la limite de  $T=1000$  pas d'échantillonnage ou à la capacité limitée du Simple U-Net à nettoyer parfaitement les zones de vide. De plus, une autre explication possible est que la fonction de perte (MSE) fait une moyenne. Ainsi, le modèle peut hésiter entre deux couleurs et produire du gris/flou.



FIGURE 4.1 – Grille d'échantillons générés par le modèle après 100 époques.

## 4.2 Évaluation de la Généralisation (Mémorisation vs Création)

Un défi majeur des modèles génératifs entraînés sur de petits datasets est le risque de sur-apprentissage (*overfitting*) : le modèle se contente-t-il de restituer à l'identique des images vues à l'entraînement ?

L'analyse comparative entre les images générées et le dataset d'entraînement révèle un comportement hybride intéressant :

- **Fidélité Structurale (Mémorisation)** : Nous avons retrouvé des structures géométriques très similaires à celles du dataset, notamment pour des objets complexes comme les arcs, les livres et les casques. Cela indique une part de mémorisation des formes par le réseau.
- **Variabilité Chromatique (Généralisation)** : Cependant, les palettes de couleurs appliquées à ces objets sont inédites. Le modèle génère par exemple un casque connu, mais avec une teinte ou une texture absente du jeu de données original.

Ce phénomène suggère que le modèle opère par **recombinaison** : il a appris à dissocier la forme de l'objet de son style (sa couleur), lui permettant de créer des variantes inédites d'objets existants plutôt que de simples copies conformes pixel par pixel.

## 4.3 Conclusion des Résultats

Compte tenu de la contrainte matérielle forte (CPU) qui a imposé une architecture minimale, les résultats sont positifs. Le pipeline est fonctionnel : le modèle part d'un bruit gaussien pur et construit, étape par étape, une image cohérente qui respecte la distribution des couleurs et des formes du dataset original.