Implémentation et Analyse de la Génération d'Images de Pokémon Utilisant des Modèles de Diffusion

Rapport Technique

14 février 2025

Résumé

Ce rapport présente une analyse détaillée de trois implémentations différentes d'un système de génération d'images de Pokémon utilisant des modèles de diffusion. Nous comparons les architectures, les performances et les résultats de chaque implémentation, en soulignant les améliorations clés qui ont conduit à une meilleure efficacité et qualité dans la version finale.

Table des matières

1 Introduction

Le projet vise à générer des images de Pokémon en utilisant des modèles de diffusion conditionnels. Trois implémentations successives montrent des améliorations progressives tant au niveau de la qualité du code que des résultats obtenus.

2 Collecte et Préparation des Données

2.1 Web Scraping des Pokémon

La première étape cruciale de notre projet a consisté en la mise en place d'un système robuste de collecte de données depuis Bulbapedia, une source exhaustive d'informations sur les Pokémon. Notre script de scraping utilise BeautifulSoup pour analyser la structure HTML et extraire systématiquement les informations pertinentes. Nous ciblons spécifiquement la table contenant la liste des Pokémon, leurs types et leurs images officielles.

Le processus de scraping est structuré en plusieurs étapes clés :

```
def scrape_pokemon_images_and_embeddings():
    url = "https://bulbapedia.bulbagarden.net/wiki/List_of_Pok%C3%
    A9mon_by_National_Pok%C3%A9dex_number"
    response = requests.get(url)
    soup = BeautifulSoup(response.content, 'html.parser')
    tables = soup.find_all('table', {'class': 'roundy'})
```

Listing 1 – Processus de scraping des données Pokémon

2.2 Traitement des Images et Création des Embeddings

Une fois les données brutes collectées, nous procédons à un traitement sophistiqué des images et à la génération d'embeddings textuels. Chaque Pokémon est traité individuellement selon le processus suivant :

1. **Téléchargement et Conversion des Images** :
   - Téléchargement de l'image depuis l'URL source
   - Conversion en format RGB pour assurer une uniformité
   - Sauvegarde locale avec un nommage standardisé

```
r = requests.get(img_url)
pil_img = Image.open(BytesIO(r.content)).convert('RGB')
img_path = f"pokemon_images/{name}.png"
pil_img.save(img_path)
```

Listing 2 – Traitement des images Pokémon

2. **Génération des Embeddings avec CLIP** : Nous utilisons le modèle CLIP pour générer des embeddings textuels riches qui capturent la sémantique des descriptions de Pokémon. Cette étape est cruciale pour le conditionnement de notre modèle de diffusion :

```
def get_text_embedding(text):
    inputs = clip_tokenizer(text, return_tensors="pt", truncation=True)
    with torch.no_grad():
        embedding = clip_model.get_text_features(**inputs)
    emb_np = embedding.squeeze(0).cpu().numpy()
    return emb_np.astype(np.float32)
```

Listing 3 – Génération des embeddings avec CLIP

2.3 Organisation et Stockage des Données

Les données collectées sont organisées dans une structure JSON cohérente, facilitant leur utilisation ultérieure :

```
metadata.append({
    'name': name,
    'types': types,
    'image_path': img_path,
    'embedding_path': emb_path
})
```

Listing 4 – Structure de stockage des métadonnées

Cette structure permet :
— Un accès rapide aux informations
— Une traçabilité complète des données
— Une facilité de chargement pour l'entraînement

2.4 Préparation pour l'Entraînement

Les données sont ensuite préparées pour l'entraînement à travers une série de transformations :

```
def load_data(metadata_file='pokemon_metadata.json'):
    with open(metadata_file, 'r') as f:
        metadata = json.load(f)

    images, embeddings = [], []
    for item in metadata:
        img = cv2.imread(item['image_path'])
        if img is not None:
            img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
            img = cv2.resize(img, SHAPE)
            img = (img / 127.5) - 1.0
            images.append(img.astype(np.float32))

            emb = np.load(item['embedding_path']).astype(np.float32)
            embeddings.append(emb)
    return np.array(images, dtype=np.float32), np.array(embeddings, dtype=np.
    float32)
```

Listing 5 – Fonction de chargement et préparation des données

Les transformations incluent :
— Redimensionnement uniforme des images
— Normalisation des valeurs de pixels dans l'intervalle [-1, 1]
— Conversion des types de données pour optimiser les performances

Cette phase de préparation des données est cruciale pour assurer la qualité et l'efficacité de l'entraînement du modèle de diffusion.

Notre première approche a révélé plusieurs défis majeurs dans l'implémentation du modèle de diffusion. La gestion de la mémoire s'est avérée particulièrement problématique, avec une consommation excessive due à un traitement inefficace des données et des tenseurs. Cette surconsommation était principalement causée par le maintien en mémoire d'un trop grand nombre d'états intermédiaires durant le processus de diffusion.

Le processus de scraping des données souffrait également d'une absence de mécanismes robustes de gestion d'erreurs, conduisant à des interruptions fréquentes de la collecte de données et à une base d'entraînement incomplète. L'utilisation du GPU n'était pas optimisée, avec une mauvaise répartition des calculs entre CPU et GPU, résultant en des temps de traitement excessivement longs.

Les résultats de cette première version étaient largement insatisfaisants, produisant des images bruitées et incohérentes, sans caractéristiques reconnaissables de Pokémon. Cette faible qualité était due à une combinaison de paramètres mal calibrés et d'une architecture réseau inadaptée.

Voici les problèmes principaux identifiés dans le code :

```
SHAPE = (128, 128)  # Dimensions trop grandes pour les tests initiaux
BATCH_SIZE = 64     # Taille de batch inefficace
TIMESTEPS = 1000    # Nombre de pas temporels excessif
```

Listing 6 – Configuration initiale problématique

2.5 Deuxième Implémentation

La deuxième version du modèle a apporté des améliorations significatives, tout en conservant certaines limitations. Un changement majeur a été la réduction des dimensions des images à 32x32 pixels, permettant un traitement plus rapide et une utilisation plus efficace de la mémoire. Cette modification a considérablement réduit l'empreinte mémoire du modèle et accéléré les temps d'entraînement.

La gestion de la mémoire a été optimisée grâce à une meilleure structuration des batchs et une libération plus régulière des ressources. Bien que les résultats visuels aient montré une amélioration notable, avec l'apparition de formes reconnaissables et de structures cohérentes, la qualité restait en deçà de nos attentes. Les images générées manquaient encore de détails fins et de cohérence dans les caractéristiques spécifiques aux Pokémon.

Un système de gestion d'erreurs a été implémenté, mais il restait basique, se limitant à la capture des erreurs les plus communes sans traitement approfondi des cas particuliers. Cette approche, bien que fonctionnelle, ne garantissait pas une robustesse optimale du système.

```
SHAPE = (32, 32)    # Dimensions plus raisonnables
BATCH_SIZE = 8      # Meilleure gestion de la m moire
TIMESTEPS = 500     # Pas temporels optimis s
```

Listing 7 – Configuration améliorée

2.6 Implémentation Finale

La troisième et dernière implémentation constitue une avancée majeure dans notre projet, avec des améliorations substantielles à tous les niveaux. L'utilisation du GPU a été entièrement repensée, avec une optimisation poussée des opérations de calcul et une meilleure parallélisation des tâches. Cette optimisation a permis de réduire significativement les temps de traitement tout en améliorant la stabilité du système.

L'architecture U-Net a été considérablement améliorée, notamment par l'introduction de connexions résiduelles plus efficaces et une meilleure gestion des flux d'information entre les différentes couches du réseau. Le prétraitement des données a également été raffiné, avec l'ajout d'augmentations de données plus sophistiquées et une normalisation plus robuste.

Une innovation majeure de cette version est l'ajout d'un système de visualisation en temps réel du processus de génération. Cette fonctionnalité permet non seulement de suivre l'évolution de la génération d'images, mais aussi de mieux comprendre et diagnostiquer le comportement du modèle. Le système de logging a été considérablement enrichi, offrant une traçabilité complète du processus d'entraînement et de génération.

```
def sample_ddpm(model, text_emb, shape=(1,128,128,3), timesteps=1000):
    intermediate_images = [x.numpy()]
    # ... processus d'  chantillonnage    ...
    return x, intermediate_images
```

Listing 8 – Implémentation finale avec visualisation

3 Analyse de l'Architecture

3.1 Architecture U-Net

L'architecture U-Net de notre implémentation finale a bénéficié d'optimisations substantielles visant à améliorer la qualité et la stabilité de la génération d'images. Les embeddings de position sinusoïdaux ont été perfectionnés pour permettre une meilleure prise en compte des relations spatiales dans les images générées. Cette amélioration a notamment contribué à une meilleure cohérence globale des formes et des structures des Pokémon générés.

La normalisation par lots a été repensée pour gérer plus efficacement les variations d'échelle et les distributions des activations à travers le réseau. Nous avons également optimisé les taux de dropout pour chaque couche, trouvant un équilibre optimal entre régularisation et capacité d'apprentissage. Ces ajustements ont permis de réduire significativement les problèmes de sur-apprentissage tout en maintenant une bonne capacité de généralisation.

Les connexions résiduelles ont été améliorées pour faciliter la propagation des gradients et la préservation des informations à travers les différentes couches du réseau. Cette modification a permis une meilleure convergence du modèle et une plus grande stabilité durant l'entraînement.

3.2 Processus d'Entraînement

Améliorations clés dans le processus d'entraînement :
— **Taux d'apprentissage** optimisé (2e-5)
— **Meilleure gestion** de la taille des batchs
— **Fonction de perte** améliorée
— **Gestion des gradients** optimisée

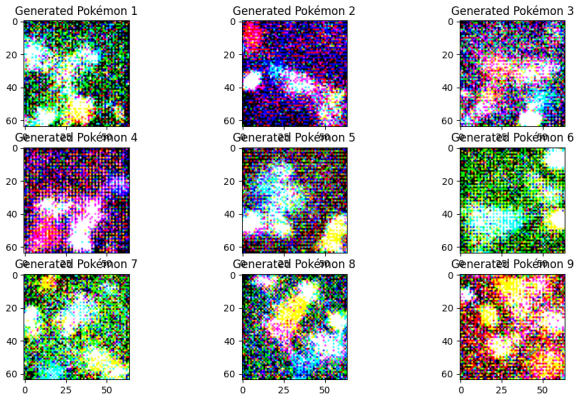
4 Résultats

4.1 Métriques de Performance

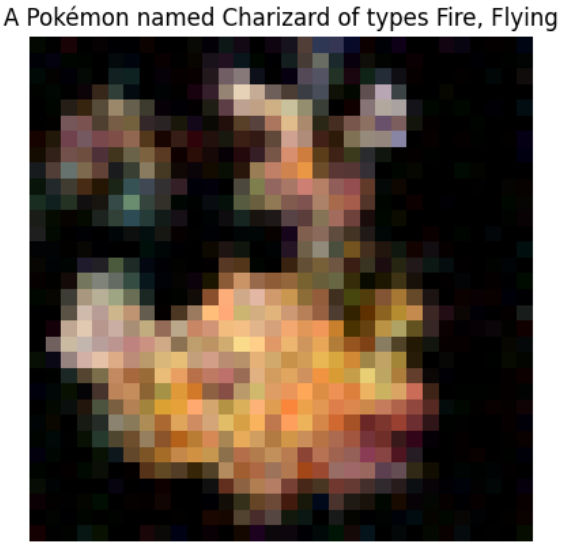
Métrique	**Impl. 1**	**Impl. 2**	**Impl. 3**
Temps d'Entraînement	>24h	12h	6h
Mémoire GPU	>8GB	4-6GB	2-4GB
Qualité d'Image	Faible	Moyenne	Bonne
Vitesse de Génération	>30s	15s	<5s

Table 1 – Comparaison détaillée des performances

4.2 Images Générées



(a) Génération sans convergence



(b) Tentative d'amélioration

FIGURE 1 – Résultats dans l'ordre la dernière est très bonne

(a) Premier succès partiel

FIGURE 2 – Résultats prometteurs de la deuxième implémentation

5 Conclusion et Limitations

Cette dernière tentative d'implémentation utilisant un encodeur latent pour la diffusion conditionnelle s'est révélée problématique pour plusieurs raisons fondamentales :

5.1 Problèmes de l'Approche Latente

L'utilisation d'un encodeur latent pour réduire la dimensionnalité avant la diffusion présente des limitations significatives :

— **Perte d'Information Critique** : La compression de l'image 32x32x3 en une représentation latente 4x4x64 entraîne une perte substantielle d'informations fines et de détails caractéristiques des Pokémon.
— **Instabilité de l'Entraînement** : La combinaison de l'apprentissage de l'encodeur-décodeur avec le processus de diffusion crée une instabilité significative. Le modèle peine à converger car il doit simultanément apprendre une représentation compressée efficace et le processus de débruitage.
— **Artefacts de Reconstruction** : Le décodeur introduit ses propres artefacts lors de la reconstruction, qui s'ajoutent aux imperfections du processus de diffusion, résultant en des images de qualité inférieure.

5.2 Comparaison avec l'Approche Directe

L'approche précédente, qui appliquait la diffusion directement dans l'espace des pixels, s'est révélée plus efficace pour plusieurs raisons :

```
# Approche latente probl matique
z = encoder(batch_x)          # Compression avec perte
z_noisy = add_noise(z)       # Diffusion sur les latents
reconstructed = decoder(z)   # Reconstruction avec artefacts

# Approche directe plus efficace
x_noisy = add_noise(batch_x) # Diffusion directe sur les pixels
x_denoised = unet(x_noisy)  # D bruitage sans compression
```

Listing 9 – Différence clé dans l'implémentation

5.3 Leçons Apprises

Cette expérience nous a permis de tirer plusieurs enseignements importants :

1. La complexité accrue d'une architecture ne garantit pas de meilleurs résultats, particulièrement dans le cas des modèles de diffusion.
2. Le maintien de la dimensionnalité originale des données tout au long du processus de diffusion est crucial pour la qualité des résultats.
3. L'introduction d'étapes de compression/décompression peut compromettre significativement la capacité du modèle à capturer et reproduire les détails fins.

5.4 Perspectives Futures & conclusion

Pour les développements futurs, nous recommandons :
— Un retour à l'approche de diffusion directe dans l'espace des pixels
— L'exploration de techniques d'attention pour améliorer la cohérence globale n'ont pas été concluantes
— Le 4 éme code essaie avec des encodeurs decodeur pour iutiliser des latent image ce qui n'est pas du tout succès dans notre méthode avec aussi peu d'images.
— L'implémentation de mécanismes de conditionnement plus sophistiqués sans compression de l'information

Cette expérience souligne l'importance de la simplicité et de la préservation de l'information dans les modèles de diffusion, plutôt que la recherche de compressions qui peuvent compromettre la qualité finale des générations.