# Re-implementation of Matcha-TTS model by Mehta Shivam

**Mathis Lecry** MATHIS.LECRY@ENSAM.EU
*Master Systèmes Avancés et Robotique, Arts et Métiers (ENSAM)*

**Paul-Marie Demars** PAUL-MARIE.DEMARS@ENSAM.EU
*Master Systèmes Avancés et Robotique, Arts et Métiers (ENSAM)*

**Minh Nhut NGUYEN** MINH_NHUT.NGUYEN@ETU.SORBONNE-UNIVERSITE.FR
*ISI, Sorbonne Université (UPMC)*

**Yucheng DAI** YUCHENG.DAI@ETU.SORBONNE-UNIVERSITE.FR
*ISI, Sorbonne Université (UPMC)*

**Editor:** Machine Learning Avancé (2025-2026)

## Contents

## Abstract

This project presents the re-implementation of the Matcha-TTS model, a fast and probabilistic text-to-speech architecture based on Optimal-Transport Conditional Flow Matching (OT-CFM)(**?** ). The objective was to reproduce the results of the paper by Shivam Mehta (2024) by implementing the architecture from scratch. We implemented a Transformer-based text encoder, a duration predictor, and a 1D U-Net decoder driven by an Ordinary Differential Equation (ODE). The model was trained on the LJSpeech dataset. This report details our architectural approach, the implementation choices made to separate the mathematical logic from the neural network, and the results obtained during training.

**Keywords:** Synthèse vocale, Flow Matching, U-Net, Deep Learning, Reproduction.

## 1. Introduction

Text-to-Speech (TTS) synthesis has witnessed significant advancements in recent years, shifting from rigid concatenative systems, often perceived as robotic, to neural generative models capable of producing near-human natural speech. In recent literature, many state-of-the-art TTS systems are based on Diffusion Probabilistic Models (DPMs), which formulate speech generation as an iterative denoising process. However, these models suffer from a major drawback: slow inference speed. Generating a high-quality spectrogram typically requires a large number of sampling steps to numerically solve the underlying Stochastic Differential Equations (SDEs), leading to a challenging trade-off between synthesis speed and audio quality (1; 2).

In this context, our project focuses on the study and reproduction of Matcha-TTS, a novel architecture proposed by Mehta et al. (2024), which aims to address this latency issue without sacrificing synthesis quality (1). The primary task of this model is to convert input text into a Mel-spectrogram—subsequently transformed into a waveform by a vocoder—using an innovative approach known as Optimal Transport Conditional Flow Matching (OT-CFM)(3). Unlike classical diffusion methods based on score matching, Flow Matching learns a vector field that defines simpler and more direct probabilistic trajectories, often close to linear paths, between the noise distribution and the data distribution.

Implementing this architecture raises several technical challenges that we address throughout this work:

1. **Computational Efficiency:** designing an ODE-based decoder capable of generating high-quality spectrograms in significantly fewer steps than diffusion-based models such as Grad-TTS (2).
2. **Alignment and Duration Modeling:** jointly learning text–audio alignment and phoneme duration prediction, which requires the integration of a monotonic alignment mechanism and a robust duration predictor(1).
3. **Encoder Architecture:** improving long-term dependency modeling in text by incorporating modern attention mechanisms, notably Rotary Positional Embeddings (RoPE), which provide better relative position extrapolation than classical sinusoidal embeddings (**?** ).

This report details our approach to reproducing this non-autoregressive and probabilistic architecture, and analyzes how replacing SDE-based diffusion processes with ODE-based Flow Matching enables the development of a fast, lightweight, and high-performing TTS system.

## 2. Présentation de l'algorithme

### 2.1 Architecture Globale

Le modèle suit une structure Encodeur-Décodeur classique mais optimisée pour le calcul probabiliste. Il se compose de trois blocs principaux illustrés dans la Figure 1.

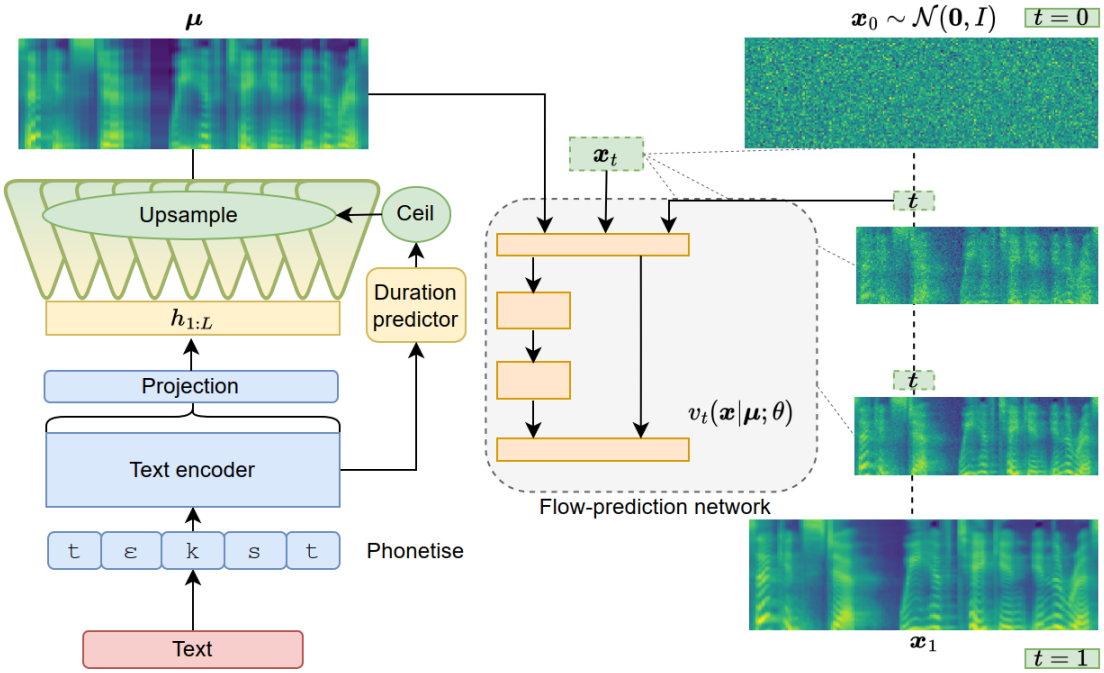


Figure 1: Architecture globale de Matcha-TTS. (a) L'encodeur de texte transforme les phonèmes en représentations latentes. (b) Le décodeur utilise le Flow Matching pour transformer le bruit en spectrogramme Mel, guidé par l'alignement.

1. **Encodeur de Texte :** Un Transformer qui prend en entrée la séquence de phonèmes. Une particularité de notre implémentation est l'utilisation des *Rotary Positional Embeddings* (RoPE), qui permettent une meilleure généralisation sur la longueur des séquences.

2. **Prédicteur de Durée et Alignement :** Un algorithme d'alignement monotone (MAS) est utilisé durant l'entraînement pour trouver le chemin optimal entre le texte et l'audio. Un module de prédiction de durée apprend ensuite ce rythme pour l'inférence.

3. **Décodeur (U-Net 1D) :** Le décodeur constitue le cœur du système.Il s'agit d'un réseau de neurones paramétré par $\theta$(paramètres entraînables du réseau neuronal) qui prend en entrée un spectrogramme bruité $x_t$, un instant temporel $t \in [0, 1]$, ainsi qu'une condition textuelle $\mu$ issue du Text Encoder, et prédit un champ de vecteurs $v_\theta(x_t, t, \mu)$ représentant la vitesse instantanée du flux permettant la reconstruction du signal.

### 2.2 Le cœur mathématique : Conditional Flow Matching (OT-CFM)

La principale innovation de Matcha-TTS réside dans sa méthode de génération. Les modèles de diffusion traditionnels apprennent à débruiter le signal en suivant des trajectoires probabilistes complexes et fortement courbées, ce qui nécessite un grand nombre d'itérations lors de l'inférence.

Matcha-TTS repose sur le **Flow Matching à Transport Optimal (OT-CFM)**. L'objectif est de définir la trajectoire de transformation la plus simple et la plus courte possible entre la distribution source $x_0 \sim \mathcal{N}(0, I)$ et la distribution cible $x_1 \sim p_{\text{data}}$. Dans ce cadre, la trajectoire optimale correspond à une interpolation linéaire entre les deux distributions.

L'interpolation entre le bruit $x_0$ et la donnée cible $x_1$ au temps $t \in [0, 1]$ est définie par :

$$x_t = \big(1 - (1 - \sigma_{\min})t\big)x_0 + tx_1, \tag{1}$$

où $\sigma_{\min}$ désigne un bruit résiduel faible (typiquement $10^{-4}$) introduit pour améliorer la stabilité numérique et éviter des gradients dégénérés aux bornes de l'intervalle temporel.

Cette interpolation linéaire simplifie considérablement l'apprentissage du décodeur. Plutôt que d'estimer un score complexe comme dans les modèles de diffusion, le réseau apprend à approximer un champ de vecteurs simple, quasi constant le long de la trajectoire, qui pousse progressivement le signal bruité vers la donnée cible.

Le caractère *conditionnel* du Conditional Flow Matching est assuré par la variable $\mu$, qui paramètre le champ de vecteurs et guide la trajectoire vers une distribution de parole cohérente avec le texte d'entrée. Ce conditionnement explicite permet un meilleur contrôle du processus de génération et renforce la cohérence texte–audio.

Grâce à ces propriétés, l'OT-CFM permet de réduire significativement le nombre d'étapes nécessaires pour atteindre la distribution cible, limitant ainsi l'accumulation d'erreurs de discrétisation et améliorant la qualité perceptuelle des signaux générés.

### 2.3 Synthèse et résolution de l'ODE

Lors de l'inférence, la génération démarre à partir d'un bruit gaussien pur ($t = 0$). Le spectrogramme est ensuite reconstruit en intégrant le champ de vecteurs prédit par le décodeur à l'aide d'un solveur d'Équation Différentielle Ordinaire (ODE), typiquement la méthode d'Euler explicite, jusqu'à $t = 1$.

En raison de la linéarité des trajectoires apprises par l'OT-CFM, Matcha-TTS atteint une synthèse de haute fidélité avec un nombre très réduit d'évaluations du modèle (NFE – *Number of Function Evaluations*), généralement compris entre 2 et 10 pas, contre plusieurs dizaines pour les approches diffusionnelles classiques.

Le spectrogramme Mel obtenu (Figure 2) est finalement converti en forme d'onde audible à l'aide d'un vocoder pré-entraîné, en l'occurrence HiFi-GAN, afin d'obtenir le signal audio final.

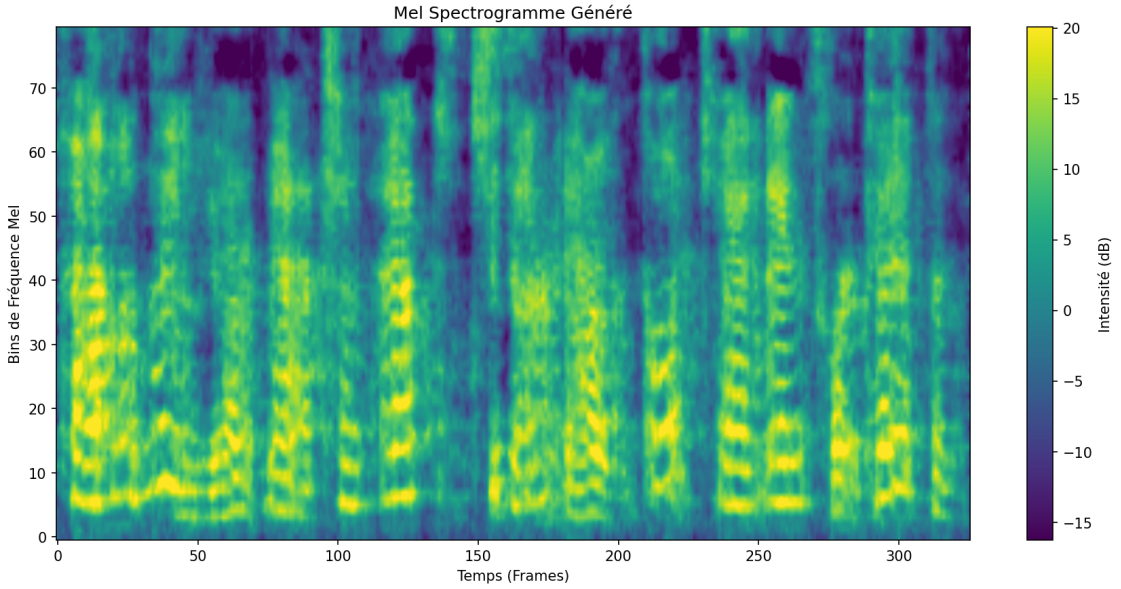


Figure 2: Mel-Spectrogramme retournée par le décodeur

### 2.4 Architecture Globale et Flux de Données

Notre solution conserve l'architecture fondamentale non-autorégressive basée sur le *Optimal-Transport Conditional Flow Matching* (OT-CFM). Le pipeline de génération suit les étapes suivantes :

1. **Encodage du Texte :** Une séquence de phonèmes est transformée en représentations latentes par un encodeur Transformer.
2. **Alignement et Durée :** Un module d'alignement monotone (MAS) aligne ces représentations avec le spectrogramme cible durant l'entraînement. Un prédicteur de durée apprend simultanément à estimer la longueur de chaque phonème pour l'inférence.
3. **Décodage par Flow Matching :** Un décodeur U-Net 1D, conditionné par le texte et le temps $t$, prédit le champ de vecteurs nécessaire pour transformer un bruit gaussien en spectrogramme Mel via la résolution d'une ODE (Ordinary Differential Equation).

Cette structure respecte fidèlement la méthodologie décrite dans l'article original (1) et utilise les principes théoriques du Flow Matching (3).

### 2.5 Différences avec l'Article de Référence et Justifications

Bien que le cœur mathématique soit identique, nous avons opéré plusieurs simplifications et refactorisations par rapport au code officiel (`shivammehta25/matcha-tts`). Ces choix sont motivés par une volonté de clarté et de maîtrise du code.

#### 2.5.1 Simplification de la Configuration

L'implémentation officielle repose lourdement sur **Hydra** pour la gestion des hyperparamètres, avec une structure de fichiers de configuration YAML complexe et imbriquée (plus de 30 fichiers de config détectés dans le dépôt original).

**Notre approche :** Nous avons remplacé ce système par une configuration explicite et directe en Python.

- **Justification :** L'utilisation de Hydra, bien que puissante pour des expérimentations à grande échelle, obscurcit la lecture des paramètres pour une ré-implémentation. En définissant les hyperparamètres (canaux cachés, taux d'apprentissage, etc.) directement dans le code ou via des arguments simples, nous rendons l'architecture plus transparente et plus facile à déboguer pour notre équipe.

#### 2.5.2 Refactorisation du Flow Matching

Dans le code original, la logique mathématique du Flow Matching est souvent entremêlée avec la définition des modèles PyTorch.

**Notre approche :** Nous avons isolé la logique du Flow Matching dans un module dédié `matcha/models/components/flow_matching.py`. Ce fichier contient spécifiquement les classes pour l'interpolation OT-CFM et le calcul de la perte, indépendamment de l'architecture du réseau de neurones.

- **Justification :** Cette séparation respecte le principe de responsabilité unique. Elle permet de modifier la méthode de génération (par exemple, changer le solveur d'ODE ou le type de bruit) sans toucher à l'architecture du décodeur U-Net, facilitant ainsi la maintenance et la compréhension des équations différentielles sous-jacentes (**?** ).

#### 2.5.3 Gestion des Données et Phonémisation

L'article original propose un pipeline de pré-traitement très générique capable de gérer de multiples datasets et langages via des scripts complexes (`matcha/text/cleaners.py`, `utils/generate_data_statistics.py`).

**Notre approche :** Nous nous sommes concentrés spécifiquement sur le dataset **LJSpeech** (anglais, mono-locuteur). Nous avons réécrit un `DataModule` simplifié (`ljspeech_datamodule.py`) et un `Dataset` (`ljspeechDataset.py`) qui intègrent directement la phonémisation et la conversion en spectrogrammes Mel.

- **Justification :** En restreignant le périmètre au dataset LJSpeech, nous avons éliminé une grande quantité de code "boilerplate" nécessaire à la gestion multi-locuteur et multi-langue, ce qui nous a permis de nous concentrer sur la performance du modèle acoustique lui-même (**?** ).

#### 2.5.4 Optimisation et Logging

Le code de référence inclut des outils de logging avancés (WandB, Neptune, etc.) et des callbacks PyTorch Lightning complexes pour la visualisation.

**Notre approche :** Nous avons implémenté un système de logging plus léger, utilisant principalement les fonctionnalités natives de PyTorch Lightning pour le suivi des pertes (*loss*) et la sauvegarde des *checkpoints*.

- **Justification :** Pour notre échelle d'entraînement (sur une seule GPU RTX A6000), un monitoring simplifié est suffisant et réduit les dépendances externes, rendant le projet plus facile à installer et à exécuter sur différents environnements.

## 3. Données

### 3.1 Dataset et Pré-traitement

Nous utilisons le corpus LJSpeech (24h, $\approx$ 13 100 clips, 22 050 Hz), standard garantissant la comparabilité avec l'état de l'art. Le pré-traitement via notre `LJSpeechDataModule` respecte le protocole original : conversion en spectrogrammes Mel (80 bandes, paramètres HiFi-GAN) et répartition standard (12 500 train, 100 validation, 500 test).

### 3.2 Adaptations et Choix d'Implémentation

#### 3.2.1 Simplifications Stratégiques

Contrairement à l'article original (multi-locuteur VCTK), nous nous limitons au cas mono-locuteur pour optimiser l'apprentissage sur GPU unique. De plus, nous avons intégré la normalisation directement dans le `DataModule` ("à la volée"), éliminant les scripts de pré-calcul statistique pour un pipeline plus accessible.

#### 3.2.2 Ablation : Choix de la Tokenisation

Nous avons comparé trois représentations textuelles sur 10 époques :

- **Graphèmes :** Texte brut, nos caractères standards.
- **Phonemizer :** La méthode de référence utilisant l'API.
- **CMUdict :** Un dictionnaire phonétique statique.

Bien que l'approche CMUdict (courbe bleue) minimise le mieux la perte (Loss $\approx$ 2.48) face aux Graphèmes ($\approx$ 3.42) et au Phonemizer ($\approx$ 4.23), les tests d'écoute révèlent des résultats inaudibles, suggérant un sur-apprentissage ou un désalignement de l'attention. À l'inverse, l'approche par Graphèmes, bien qu'ayant une perte intermédiaire, produit une synthèse intelligible et stable. Privilégiant la qualité perceptuelle à la métrique brute, nous avons retenu l'utilisation des graphèmes.

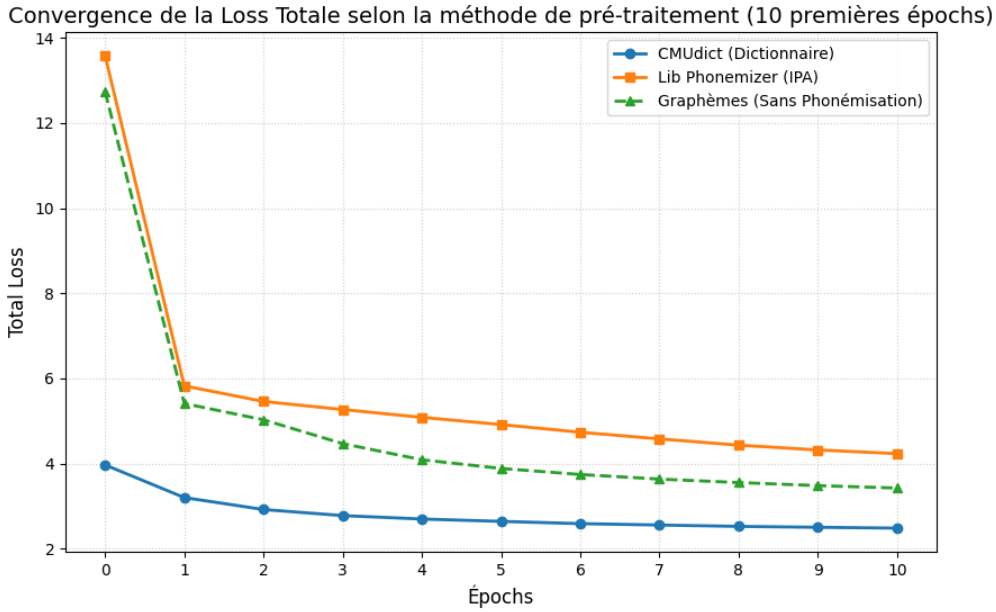


Figure 3: Convergence de la Loss Totale selon la tokenisation (10 époques).

## 4. Evaluation expérimentale

### 4.1 Justification de l'architecture et choix des hyperparamètres

Cette section vise à valider l'architecture réimplémentée de Matcha-TTS et à justifier les hyperparamètres retenus pour l'entraînement final. Conformément à l'objectif de reproductibilité du projet, nous avons établi un protocole afin d'optimiser le compromis entre la qualité de synthèse, la stabilité de l'apprentissage et le coût computationnel.

#### 4.1.1 Méthodologie d'Ablation et Métriques

1. **Isolation des variables :** Pour chaque variation étudiée, nous avons réalisé un entraînement spécifique sur le GPU en ne modifiant qu'un seul paramètre à la fois par rapport à la configuration *Baseline*.

2. **Horizon de comparaison (15 époques / 3h) :** Compte tenu des contraintes d'accès aux ressources de calcul partagées (GPU du Master), nous avons limité ces tests comparatifs à une durée d'environ 3 heures, ce qui correspond à environ 15 époques. Ce laps de temps est suffisant pour observer la pente de convergence initiale et détecter les instabilités précoces sans consommer excessivement de ressources.

3. **Métriques de suivi :** L'analyse comparative repose sur les courbes extraites via TensorBoard. Nous nous concentrons spécifiquement sur l'évolution des composantes de la fonction de coût :

   - **Duration Loss :** Qualité de l'alignement temporel (rythme).
   - **Prior Loss :** Capacité de l'encodeur à structurer l'espace latent.
   - **Diffusion Loss :** Qualité de la reconstruction spectrale par le décodeur (Flow Matching).
   - **Total Loss :** Somme pondérée indiquant la performance globale.

#### 4.1.2 Étude d'Ablation et Analyse des Hyperparamètres

Nous avons mené une série de tests unitaires (*ablation study*) en isolant chaque paramètre pour observer son impact sur la dynamique de convergence.

##### A. Architecture et Stabilité : Influence du Prenet

Le Prenet est un module situé avant l'encodeur, souvent utilisé pour faciliter l'apprentissage des structures prosodiques. Nous avons comparé la convergence sur les 18 premières époques avec et sans ce module.

Table 1: Impact du Prenet sur la convergence (Époques 0 et 17)

| **Époque** | **Config.** | **Total Loss** | **Duration Loss** |
|---|---|---|---|
| 0 | Baseline (avec) | 3.99 | 1.18 |
| | Sans Prenet | 4.03 | 1.19 |
| 17 | Baseline (avec) | **2.37** | **0.46** |
| | Sans Prenet | 2.39 | 0.47 |

L'analyse montre que la Baseline converge systématiquement mieux (Total Loss inférieure de $\sim 0.04$ point). L'impact est particulièrement critique sur la *Duration Loss* (+2.4% d'erreur sans Prenet). Le Prenet agit comme un goulot d'étranglement informationnel (*information bottleneck*) qui force le modèle à extraire des caractéristiques linguistiques robustes, stabilisant ainsi l'alignement texte-audio.

**Décision :** Conservation du Prenet (`prenet=True`).

##### B. Optimisation et Régularisation

Le choix de l'optimiseur et du taux d'apprentissage (*Learning Rate* - LR) est déterminant pour les modèles génératifs basés sur les équations différentielles (ODE) comme Matcha-TTS.

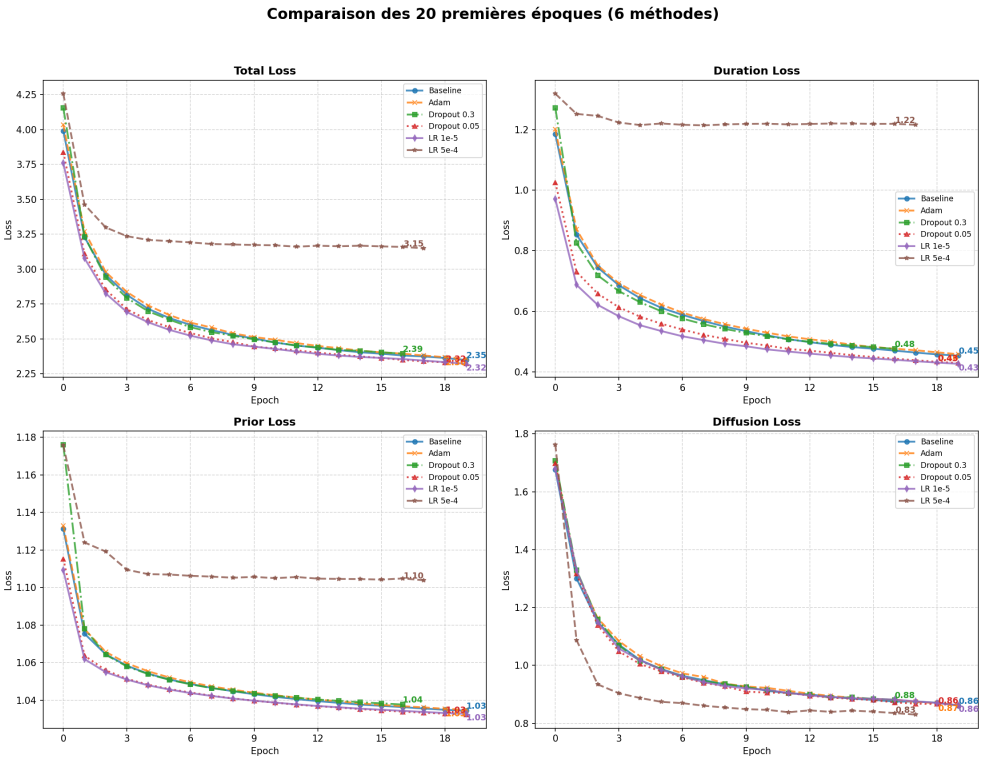


Figure 4: Impact du Learning Rate, du dropout et de l'optimiseur sur la stabilité.

1. **Learning Rate (LR) :** Nous avons observé une zone d'instabilité marquée. Avec un LR trop élevé (supérieur à la baseline), la *Duration Loss* stagne au-dessus de 1.2, indiquant un échec de l'alignement monotone. Nous retenons la valeur de la baseline (insérer valeur ici) qui représente l'optimum entre vitesse de convergence et stabilité.

2. **Optimiseur (Adam vs AdamW) :** Bien que l'optimiseur Adam standard montre une convergence initiale plus agressive (Total Loss inférieure de 0.05 point aux premières époques), nous avons privilégié **AdamW** pour l'entraînement long. En découplant la dégradation des poids (*weight decay*) du gradient, AdamW prévient l'*overfitting* et garantit une meilleure généralisation sur le long terme, ce qui est standard pour les architectures Transformer et Diffusion.

3. **Dropout :** Une valeur de 0.1 a été maintenue. Nos tests montrent qu'un dropout trop faible (0.05) accélère l'apprentissage (Loss 2.35) mais risque de faire mémoriser le bruit du dataset LJSpeech ($\sim$ 24h), tandis qu'un dropout élevé (0.3) dégrade la capture des finesses du signal.

### C. Cœur Génératif : Résolution de l'ODE et Dimensionnement

Le cœur de Matcha-TTS repose sur le *Flow Matching*, résolvant une ODE pour transformer le bruit en spectrogramme via un U-Net.

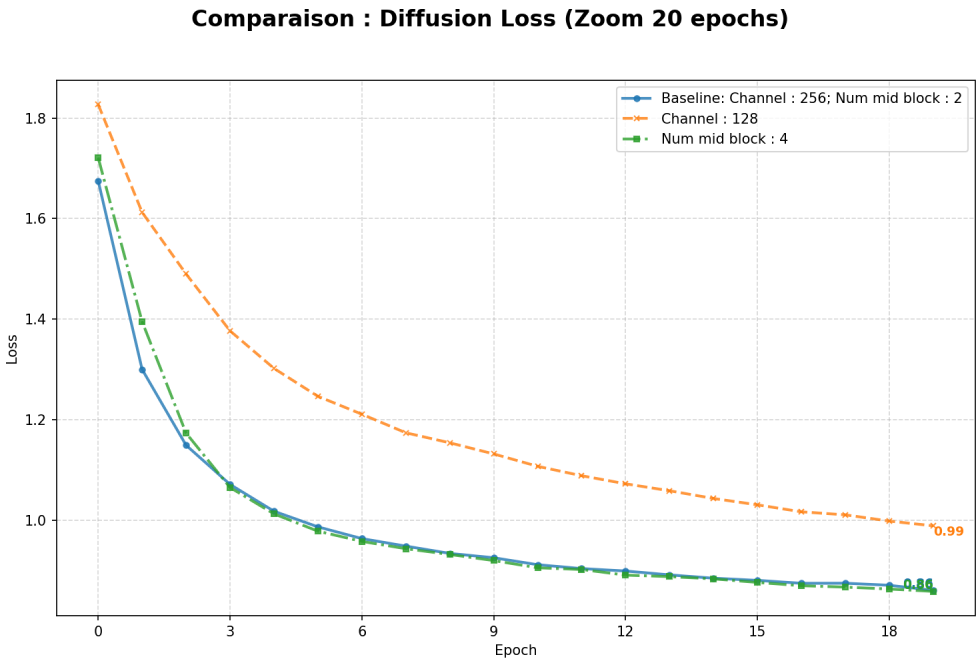


Figure 5: Comparaison de la convergence selon le solveur ODE pour le dimensionnement du décodeur.

1. **Solveur ODE (Euler vs Midpoint) :** Nous avons évalué le compromis précision/vitesse à l'époque 21. Le solveur d'Euler (ordre 1) obtient une Diffusion Loss de 2.34 contre 2.32 pour le solveur Midpoint (ordre 2). Bien que Midpoint soit théoriquement plus précis, le gain de performance est négligeable par rapport au doublement du coût de calcul (2 évaluations de fonction par pas contre 1).
   **Décision :** Utilisation du solveur d'Euler pour maximiser le RTF (*Real-Time Factor*) sans perte significative de qualité.

2. **Dimensionnement du Décodeur (U-Net) :** La réduction de la largeur du réseau à 128 canaux s'est avérée destructrice, avec une *Diffusion Loss* stagnant à 0.986 (contre 0.864 pour 256 canaux). Une capacité de 256 canaux est le minimum requis pour modéliser la complexité des 80 bandes de fréquences Mel. Bien que l'impact sur la convergence soit marginal, nous conservons 2 *Mid Blocks* pour assurer une modélisation robuste des dépendances globales au niveau du goulot d'étranglement (*bottleneck*), nos tests montrant que l'ajout de blocs supplémentaires alourdit le calcul sans améliorer la reconstruction spectrale.
   **Décision :** Maintien de 256 canaux et 2 blocs médians (*Mid Blocks*).

### D. Alignement Temporel (Duration Predictor)

Pour le *Duration Predictor* (responsable du rythme), nous avons testé l'influence de la taille du noyau de convolution (*kernel size*) ainsi que la largeur du filtre.

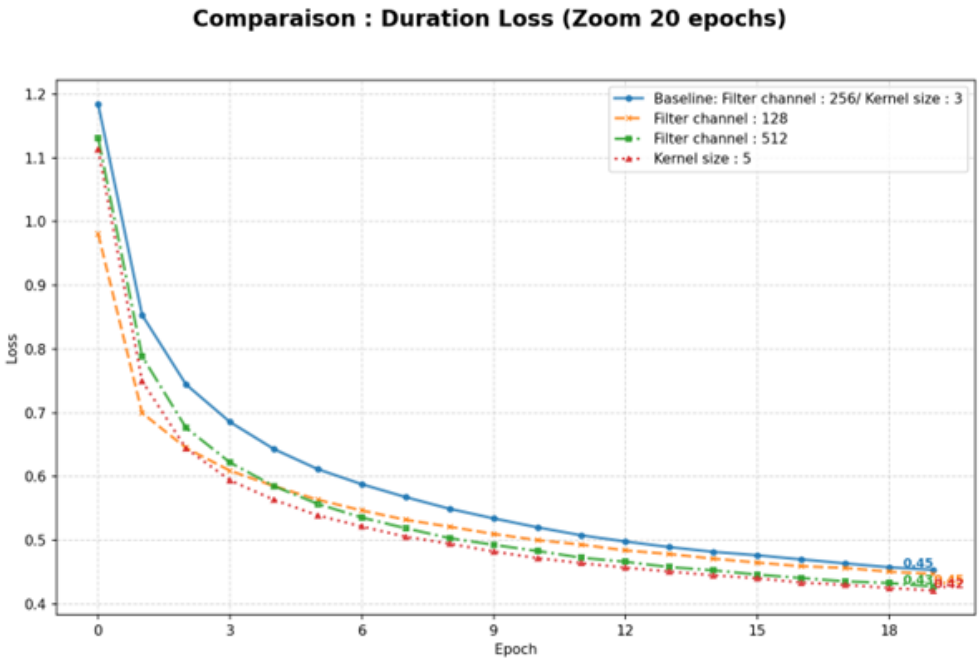


Figure 6: Analyse de la convergence du Duration Predictor.

Un noyau de 5 permet une convergence plus rapide grâce à un contexte prosodique plus large. Cependant, nous avons conservé un noyau de 3, car la baseline finit par atteindre d'excellentes performances tout en restant plus légère.

Concernant la capacité du réseau, l'augmentation des *filter channels* de 256 à 512 n'a pas montré de gain significatif en précision, nous avons donc maintenu 256 canaux pour éviter une complexité inutile.

### E. Encodage (Encoder)

Nous avons analysé la profondeur et la capacité d'attention de l'encodeur pour optimiser le ratio performance/coût.

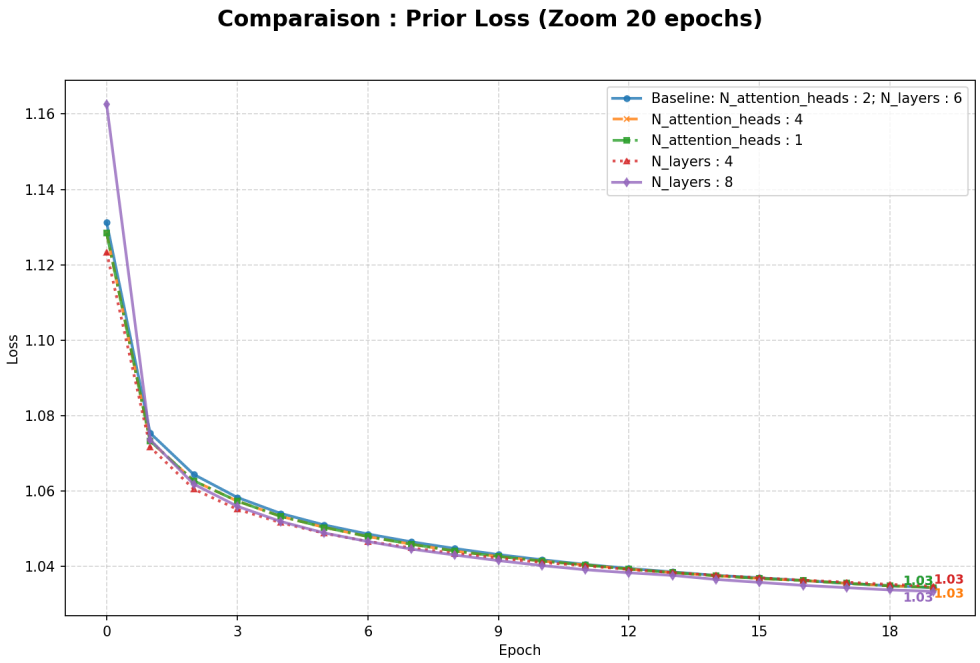


Figure 7: Impact de la profondeur et des têtes d'attention de l'encodeur.

L'augmentation de la profondeur de l'encodeur à 8 couches n'apportant qu'une réduction marginale de la Prior Loss ($< 1\%$) pour un surcoût de calcul de 33%, nous avons maintenu la configuration à 6 couches qui offre le meilleur équilibre entre performance et coût.

Parallèlement, la variation du nombre de têtes d'attention n'ayant montré aucun impact significatif sur la convergence, nous avons validé la suffisance de 2 têtes pour la capture des dépendances contextuelles.

**Conclusion :** L'architecture baseline est conservée car elle garantit une représentation latente de qualité sans sacrifier le facteur temps réel (RTF) nécessaire à l'efficacité du modèle.

#### 4.1.3 Synthèse des choix de paramètres d'entrainement retenus

Le tableau ci-dessous résume la configuration finale validée par nos tests, offrant le meilleur équilibre pour la reproduction des résultats de l'état de l'art sur nos ressources limitées.

Table 2: Configuration finale des hyperparamètres

| Composant | Paramètre | Valeur | Justification |
|---|---|---|---|
| **Global** | Learning Rate | [Valeur] | Convergence stable sans divergence |
| **Architecture** | Prenet | Activé | Stabilisation critique de l'alignement |
| **Encoder** | Layers / Heads | 6 / 2 | Suffisant pour LJSpeech, optimise le RTF |
| **Duration** | Filter Channels | 256 | Précision nécessaire pour la prosodie ($< 0.30$ loss) |
| **ODE** | Solver | Euler | Efficacité maximale (1 NFE) pour qualité équivalente |
| **Decoder** | Channels | 256 | Minimum requis pour la reconstruction spectrale |

## 5. Conclusion

**Conclusion : un résumé synthétique des principaux résultat obtenus et présentation des principales pistes d'amélioration possibles.**

## References


[1] Beskow Jonas Székely Éva Henter Gustav Eje Mehta Shivam, Tu Ruibo. Matcha-tts: A fast tts architecture with conditional flow matching. In *ICASSP 2024-2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2024.

[2] Popov Vadim, Vovk Ivan, Gogoryan Vladimir, Tashev Tasnima, and Kudinov Mikhail. Grad-tts: A diffusion probabilistic model for text-to-speech. In *International Conference on Machine Learning*, pages 8599–8608. PMLR, 2021.

[3] Lipman Yaron, Chen Ricky TQ, Ben-Hamu Heli, Nickel Maximilian, and Le Matt. Flow matching for generative modeling. *arXiv preprint arXiv:2210.02747*, 2022.