ChaosAI : Apprentissage Auto-Supervisé sur Systèmes Dynamiques Chaotiques

Architecture JEPA–Mamba-2, Prédiction par Flot Continu Optimal,
Analyse de Sensibilité Géodésique et Implémentation TPU Native

Architecte Principal

`Projet soutenu par le Google TPU Research Cloud (TRC)`
`Repository: github.com/ChaosAI`

3,930,568 paramètres validés sur TPU v6e-8 (Trillium), Europe West 4

Février 2026

Résumé

L'apprentissage auto-supervisé de représentations sur des séries temporelles non-stationnaires à queues épaisses reste un problème ouvert. Nous présentons **ChaosAI**, une architecture auto-supervisée à quatre strates pour les systèmes dynamiques chaotiques, utilisant les marchés financiers haute fréquence comme banc d'essai à rapport signal/bruit minimal. Le cœur de la contribution est une implémentation JAX native sur TPU documentant (i) un tokeniseur FSQ résistant au codebook collapse sur distributions à queues épaisses, (ii) un encodeur Mamba-2 à horloge temporelle conditionnelle (*vol-clock*) avec correction d'instabilité par bornage tanh, (iii) un prédicteur OT-CFM avec couplage Sinkhorn JIT-compatible, et (iv) un agent TD-MPC2 avec *Analyse de Sensibilité Géodésique* (ASG) : M perturbations sur la variété latente dont la divergence module dynamiquement l'alpha CVaR du planning MPPI.

L'implémentation introduit un noyau SSD par blocs aligné exactement sur les tuiles MXU 128×128 des TPU v5p, et documente une instabilité numérique bf16 identifiée expérimentalement (divergence déterministe au pas 2 750 d'un cumsum non-upcasté) avec sa résolution par upcast asymétrique. Nous présentons les résultats d'un modèle de **3,93M paramètres** validé sur TPU v6e-8 (perte -35%, perte CFM -33% en 2 700 pas), un auto-sharder GSPMD topologique de v5p-8 à v5p-768, et un pipeline de données à coût nul au repos (Drive $\leftrightarrow$ GCS $\leftrightarrow$ TPU). L'architecture est conçue pour un passage à l'échelle vers 150M–1B paramètres dont la validation des lois d'échelle constitue la prochaine étape.

Table des matières

1 Introduction

Les systèmes dynamiques complexes, qu'il s'agisse de la turbulence des fluides, du climat, ou des marchés financiers, partagent une propriété fondamentale : une sensibilité extrême aux conditions initiales qui rend toute prédiction à long terme *structurellement impossible* dans l'espace d'observation. La loi de Lorenz est universelle : deux trajectoires issues de conditions initiales infinitésimalement proches divergent exponentiellement. Pourtant, l'espace latent d'un bon modèle de représentation peut exhiber des propriétés radicalement différentes : des régimes, des bifurcations, des attracteurs dont la géométrie est stable et apprenante.

Cette observation motive l'architecture de ChaosAI. Plutôt que de prédire la valeur future d'un actif (une entreprise vouée à l'échec par la sensibilité aux conditions initiales), nous entraînons le modèle à prédire les *représentations latentes* futures : des abstractions qui capturent le régime, la volatilité, et la structure de corrélation, sans être contaminées par le bruit microscopique irréductible du signal de prix. Cette approche est directement inspirée de l'architecture Joint-Embedding Predictive Architecture (JEPA) de LeCun [1], mais appliquée pour la première fois à des séries temporelles financières multi-variées à 432 instruments.

Pourquoi les marchés financiers ? Ce choix n'est pas financier : il est épistémique. Les marchés sont le système dynamique non-stationnaire le plus adversarial accessible publiquement à haute fréquence. Ils exhibent des queues épaisses (α-stables, Lévy), des ruptures de régime spontanées, des corrélations croisées instables sur 400+ instruments, et des boucles de rétroaction adversariales. Si une architecture auto-supervisée peut extraire des représentations utiles dans cet environnement, elle fonctionnera *a fortiori* sur des signaux moins adversariaux : signaux biomédicaux (EEG, ECG), géophysique (LIGO, sismologie), audio, ou dynamique vidéo.

Contributions. Ce papier documente :

1. L'**Analyse de Sensibilité Géodésique (ASG)** pour la planification RL : perturbation sur la variété latente préservant la norme L_2, analyse spectrale de la divergence inter-ensemble, et modulation dynamique de l'alpha CVaR. Implémentée sous le nom `multiverse_crossing.py` dans le code source (Section 3.6.3).
2. La correction de l'**instabilité de l'Exo-Clock** : bornage via tanh du biais additif sur Δ_t pré-softplus, nécessaire pour que la modulation temporelle du SSM reste dans une plage stable (Section 3.3.5).
3. Le noyau **SSD par blocs (Chunked SSD)** : reformulation de la récurrence SSM en matmuls 128×128 alignés sur les tuiles MXU des TPU v5p, avec upcast asymétrique bf16/float32 sur les accumulations temporelles (Section 4.1).
4. Une **note d'implémentation sur la stabilité bf16** : divergence NaN déterministe au pas 2 750 causée par la troncature de mantisse dans un `cumsum` sur 128 pas, et sa résolution par upcast sélectif (Section 4.2).
5. Un **pipeline de données à coût marginal nul** : architecture Drive $\leftrightarrow$ GCS $\leftrightarrow$ TPU avec GCS comme cache éphémère, documentant les choix d'ingénierie (rclone, archivage Orbax, co-localisation région) (Section 6).
6. Un **Auto-Sharder GSPMD topologique** : maillage 2D adaptatif de v5p-8 à v5p-768 minimisant les sauts ICI via `create_device_mesh` (Section 4.3).

Résultats sur TPU v6e-8. Nous validons le pipeline JAX complet sur un **modèle de 3,93M de paramètres** : perte totale de 7448,6 à 4839,9 (-35%) en 2 700 pas à 0,1 pas/s. Le v6e (Trillium) est un chip d'inférence — son ratio MXU/bande passante mémoire est défavorable pour l'entraînement, résultant en $\sim 5\%$ de MFU. Ce résultat est une validation de bout-en-bout du pipeline JAX, pas un benchmark de performance. Les configurations T-Shirt (15M–7B) ciblent le TPU v5p dont les caractéristiques sont orthogonales au v6e.

2 Travaux Connexes

Modèles d'état-espace. Les SSM linéaires invariants dans le temps (S4 [2]) ont démontré leur capacité à modéliser les dépendances à longue portée sur des séquences de plus de 16 000 tokens avec une complexité linéaire. Mamba [3] a introduit la sélectivité : les paramètres B, C, Δ dépendent de l'entrée, ce qui rompt l'invariance temporelle et permet la capture de régimes. Mamba-2 [4] unifie les SSM et l'attention linéaire via la *State Space Duality* (SSD) : la récurrence SSM admet une formulation matricielle par blocs denses compatibles avec les accélérateurs matériels.

JEPA. L'I-JEPA [5] entraîne un encodeur à prédire des représentations de patches masqués depuis un contexte, sans décodeur pixel. V-JEPA [6] étend cela à la vidéo. Notre Fin-JEPA adapte ce paradigme aux séries temporelles financières multi-variées, en remplaçant les patches d'image par des blocs de tokens OHLCV et l'encodeur ViT par un encodeur Mamba-2.

Flow Matching et Transport Optimal. Le Continuous Flow Matching (CFM) [7] entraîne un champ de vitesse $v_\theta(x_t, t)$ pour transporter une distribution source vers une distribution cible via une ODE. L'OT-CFM [8] améliore la stabilité en précouplant les minibatchs source-cible par transport optimal de Sinkhorn [9], réduisant la variance des trajectoires de flux.

Apprentissage par renforcement model-based. TD-MPC2 [10] apprend un modèle du monde latent différentiable et planifie via MPPI. Le critique distribtionnel (QR-DQN [11]) estime la distribution complète des retours, permettant l'optimisation de mesures de risque comme CVaR [12]. Notre approche étend ce cadre avec une analyse de sensibilité géodésique aux conditions initiales (ASG, Section 3.6.3), inspirée de la divergence exponentielle caractéristique des systèmes chaotiques.

Scalabilité des SSM. Plusieurs travaux récents ont examiné la scalabilité des SSM [3]. Les SSM scalent différemment des Transformers : leur complexité linéaire en séquence leur permet de traiter des contextes beaucoup plus longs, mais le bénéfice de la profondeur est moins prononcé. Nos configurations T-Shirt reflètent cette observation en privilégiant la largeur (d_{model}) sur la profondeur (n_{layers}) pour le modèle 7B.

Infrastructure ML et FinOps. Les travaux sur l'efficacité des systèmes ML (MLSys) abordent rarement la question des coûts d'infrastructure pour la recherche académique. Notre architecture Drive $\leftrightarrow$ GCS $\leftrightarrow$ TPU représente une contribution pratique : démontrer que le coût de stockage (souvent ignoré dans les papiers système) peut être réduit de 90% sans perte de fonctionnalité pour les cycles de recherche intermittents.

Time-Series Foundation Models. TimesFM [18], Moirai, et d'autres modèles de fondation pour séries temporelles utilisent des architectures Transformer. Leur tokenisation est directe (valeurs scalaires normalisées). ChaosAI se distingue par : (i) une tokenisation discrète FSQ (codes symboliques), (ii) un paradigme JEPA (prédiction latente, pas de décodeur), (iii) une analyse de sensibilité géodésique (ASG) pour piloter le risque de planification. Le domaine financier à haute fréquence (1h) n'est pas adressé par ces travaux.

Quantification. FSQ [13] élimine le codebook collapse du VQ-VAE [15] en projetant des représentations continues dans un réseau hyper-cubique à faible dimension, avec une fonction de quantification différentiable par Straight-Through Estimator.

3 Architecture Stratifiée

ChaosAI est une pipeline à quatre strates d'abstraction croissante. Chaque strate consomme la sortie de la précédente :

$$\text{OHLCV brut} \xrightarrow{\text{I}} \text{codes discrets} \xrightarrow{\text{II}} h_x \in \mathbb{R}^d \xrightarrow{\text{III}} \{h_{\text{fut}}^{(n)}\}_{n=1}^{N} \xrightarrow{\text{IV}} a_t \in [-1, 1]$$

3.1 Pipeline de Données — Strate 0

Avant les quatre strates architecturales, un pipeline de données (que nous appelons informellement *Strate 0*) collecte, nettoie et convertit les données brutes en enregistrements ArrayRecord prêts pour l'entraînement.

Acquisition. 432 paires Binance Futures USDT-M sont téléchargées via l'API Kline (bougies OHLCV horaires). Le téléchargeur implémente un throttling adaptatif AIMD : il surveille l'en-tête `X-MBX-Used-Weight-1M` (budget de requêtes API par minute) et réduit proactivement le débit à 85% du budget, avec jitter exponentiel sur les retry-after. Cela évite les bans IP tout en maximisant le débit.

Couverture temporelle. Chaque paire couvre ∼8 ans de données horaires : $8 \times 365 \times 24 \approx 70\,080$ bougies/paire. Au total : $432 \times 70\,080 \approx 30.3$M bougies brutes, stockées en format `.pt` PyTorch (174 Mo compressé).

Pré-tokenisation. Le tokeniseur Strate I (FSQ + CNN causal) est appliqué à chaque paire en glissant une fenêtre de 128 tokens (stride 64). Cela produit ∼15M tokens uniques sur l'ensemble du dataset, stockés en tokens discrets $\in \{0, \ldots, 1023\}$.

Conversion ArrayRecord. Les séquences tokenisées sont converties en shards ArrayRecord (format binaire optimisé pour Grain) : 433 shards (∼160 Mo total), chacun contenant les séquences d'une ou plusieurs paires avec métadonnées (weekend_mask, exo_clock, pair_name). La validation train/val est déterministe par hachage MD5 du nom de la paire (20% val, 80% train).

Fail-Fast OHLCV. Si un fichier OHLCV est manquant, le pipeline lève `FileNotFoundError` immédiatement (Fail-Fast) : pas de stats factices (`torch.ones * 50000`). Cette discipline est critique pour éviter des encodages silencieusement incorrects qui contamineraient le buffer de trajectoires RL.

3.2 Strate I — Tokeniseur FSQ (Perception)

3.2.1 Pré-traitement SymLog

Les séries OHLCV brutes exhibent des distributions à queues épaisses : les rendements journaliers d'actifs volatils peuvent dépasser ±50% lors de chocs (Black Swan). La normalisation z-score standard comprime ces événements extrêmes dans la saturation numérique. Nous appliquons la transformation SymLog :

$$\text{SymLog}(x) = \text{sign}(x) \cdot \log(1 + |x|)$$

Cette fonction est bijective, linéaire autour de zéro (gradient non-nul), et logarithmique dans les queues (compriment les valeurs extrêmes sans les effacer). Combinée avec la normalisation RevIN par fenêtre de contexte, elle garantit que les données d'entrée restent dans une plage numérique stable pour bf16.

3.2.2 Auto-encodeur Causal Dilaté

Un auto-encodeur CNN causal dilaté compresse les patches OHLCV (fenêtre de 4 bougies, 5 canaux) en vecteurs continus de dimension $d_{\text{code}} = 64$. La causalité est assurée par des convolutions causales : le récepteur de chaque position ne voit que le passé, évitant le *lookahead bias*.

3.2.3 Quantification Scalaire Finie (FSQ)

Le codebook VQ-VAE classique souffre de l'effondrement des codes (*codebook collapse*), particulièrement sur des distributions à queues épaisses. Les queues épaisses induisent des vecteurs d'embedding avec des normes extrêmes qui ne correspondent à aucun prototype codebook appris, laissant des codes jamais actifs.

FSQ [13] résout ce problème par projection dans un hyper-cube discret :

$$\mathbf{z}_{\text{cont}} = W_{\downarrow} \cdot h \in \mathbb{R}^{d_{\text{fsq}}} \quad \text{(projection linéaire)} \tag{1}$$

$$\mathbf{z}_{\text{disc}} = \text{round}\left(\ell \cdot \tanh(\mathbf{z}_{\text{cont}})\right) \quad \text{(quantification par dimension)} \tag{2}$$

$$\mathbf{z}_{\text{recon}} = W_{\uparrow} \cdot \mathbf{z}_{\text{disc}} \in \mathbb{R}^{d_{\text{code}}} \quad \text{(remontée)} \tag{3}$$

Pour des niveaux $\ell = [8, 8, 8, 2]$, le volume de codes total est $\prod \ell_i = 8 \times 8 \times 8 \times 2 = 1024$, identique au codebook VQ-VAE, avec $d_{\text{fsq}} = 4$. Le gradient passe via le Straight-Through Estimator (`round` remplacé par identité en backward). L'utilisation du codebook atteint $\sim 100\%$ par construction : chaque cellule de la grille est théoriquement accessible.

3.3 Strate II — Fin-JEPA avec Mamba-2 et Horloges Conditionnelles

3.3.1 Architecture JEPA

Le Joint-Embedding Predictive Architecture déplace la prédiction de l'espace d'observation vers l'espace latent. Deux encodeurs E_θ (contexte) et E_ϕ (cible, EMA de E_θ) partagent la même architecture Mamba-2. Pour un épisode de longueur S :

1. L'encodeur contexte traite les tokens masqués (positions cibles remplacées par `[MASK]`) : $h_x = E_\theta(\text{tokens}_{\text{masqués}}) \in \mathbb{R}^{S \times d}$.
2. L'encodeur cible (EMA, stop_gradient) traite la séquence complète : $h_y = \text{sg}(E_\phi(\text{tokens}_{\text{complets}})) \in \mathbb{R}^{S \times d}$.
3. Le prédicteur P_θ prédit $\hat{h}_y$ aux positions cibles depuis h_x.
4. La perte VICReg est calculée entre $\hat{h}_y$ et h_y aux positions masquées.

Le masquage par blocs (*block masking*, ratio 50%, blocs de 4 à 8 tokens) force l'encodeur à apprendre des représentations qui capturent les dépendances à moyen terme, et non les corrélations locales triviales.

3.3.2 Encodeur Mamba-2

L'encodeur empile n_{layers} blocs Mamba-2 sur les embeddings de tokens. Chaque token discret $c_t \in \{0, \ldots, 1023\}$ est d'abord converti en vecteur continu via une table d'embedding codebook gelée ($\mathbb{R}^{1024 \times 64}$), puis projeté linéairement vers $\mathbb{R}^{d_{\text{model}}}$. Des embeddings positionnels sinusoïdaux non-appris sont additionnés :

$$\text{PE}(t, 2k) = \sin\left(\frac{t}{10000^{2k/d}}\right), \quad \text{PE}(t, 2k+1) = \cos\left(\frac{t}{10000^{2k/d}}\right)$$

3.3.3 Bloc Mamba-2 Complet

Chaque bloc Mamba-2 implémente :

$$[x_{\text{branch}}, z_{\text{gate}}, B, C, \Delta] = \text{Linear}_{\text{in}}(\text{LayerNorm}(x)) \tag{4}$$
$$x_{\text{branch}} = \text{CausalConv1d}(x_{\text{branch}}) \tag{5}$$
$$x_{\text{branch}} = \text{SiLU}(x_{\text{branch}}) \tag{6}$$
$$y = \text{SSD}(x_{\text{branch}}, \Delta, A_{\log}, B, C) \tag{7}$$
$$y = y \cdot \text{SiLU}(z_{\text{gate}}) \tag{8}$$
$$\text{sortie} = \text{Linear}_{\text{out}}(y) + x \quad \text{(résidu)} \tag{9}$$

Le paramètre $A_{\log} \in \mathbb{R}^{H \times N}$ est initialisé à $\log(\text{arange}(1, N+1))$ par tête. La matrice $A = -\exp(A_{\log})$ est donc strictement négative, garantissant la stabilité asymptotique du système discret : $\|h_t\| \to 0$ si l'entrée est nulle.

3.3.4 Vol-Clock : Discrétisation Conditionnée sur la Volatilité

Les marchés financiers n'ont pas de temps homogène : une heure en période calme et une heure pendant un choc de marché représentent des quantités d'information radicalement différentes. Pour que le SSM adapte sa granularité temporelle au régime, nous calculons un proxy de volatilité réalisée à partir de la dynamique des embeddings codebook :

$$v_t = \frac{\left\| e_{c_t} - e_{c_{t-1}} \right\|_2 - \mu_v}{\sigma_v + \varepsilon}, \quad v_0 = 0 \tag{10}$$

où e_{c_t} est le vecteur codebook pour le token c_t, et μ_v, σ_v sont la moyenne et l'écart-type par séquence (z-score). Ce signal est calculé avec `stop_gradient`, sans overhead computationnel ni paramètre additionnel. Il encode : $v_t \approx 0$ sur les marchés plats, $v_t \approx +3$ lors des transitions rapides.

3.3.5 Exo-Clock et Correction de l'Instabilité

Motivation. Le pas de discrétisation Δ_t du SSM contrôle la fenêtre de mémoire effective : un Δ grand implique un déclin rapide de l'état caché (bonne adaptation aux marchés volatils), un Δ petit implique une mémoire longue (marchés calmes). Biaisé de manière adaptative, Δ_t donne au modèle une horloge temps-événement endogène.

Problème de l'approche naïve. L'idée initiale consistait à biaiser Δ_t par un terme additif proportionnel à la volatilité :

$$\Delta_t \leftarrow \Delta_{\text{raw}} + \beta \cdot v_t$$

où β est un scalaire appris. Lors des premiers entraînements, des explosions de gradient se produisaient systématiquement. L'analyse XLA révèle la cause : si $\beta \cdot v_t$ devient large et positif, $\text{softplus}(\Delta_{\text{raw}} + \beta v_t) \gg 1$, et la matrice de transition $\bar{A} = \exp(A \cdot \Delta_t)$ s'approche de l'identité : l'état caché ne se dissipe plus et diverge. Si $\beta \cdot v_t$ est très négatif, $\Delta_t \approx 0$ et les gradients par rapport à Δ_t disparaissent (vanishing gradients).

Solution : bornage par tanh**.** Nous stabilisons la modulation en bornant le biais additif :

$$\boxed{\Delta_t = \text{softplus}(\Delta_{\text{raw}} + \Delta_{\max} \cdot \tanh(W_v \cdot v_t + b_v))} \tag{11}$$

avec $\Delta_{\max} = 2.0$ par défaut. La fonction tanh garantit que le biais est dans $(-2, 2)$, soit une modulation multiplicative bornée après softplus. W_v et b_v sont initialisés à zéro, assurant que le

modèle démarre en mode non-modulé et peut charger des checkpoints antérieurs sans dégradation (*backward compatible*).

Pour les signaux exogènes (volume réalisé $+\sigma_{\text{RV}}$), la même formulation s'applique :

$$\Delta_t = \text{softplus}(\Delta_{\text{raw}} + \Delta_{\max} \cdot \tanh(W_{\text{exo}} \cdot \text{clock}_{\text{exo},t} + b_{\text{exo}})) \tag{12}$$

où $\text{clock}_{\text{exo},t} \in \mathbb{R}^2$ contient la volatilité réalisée et le volume normalisé. L'initialisation à zéro des poids garantit une rétrocompatibilité complète avec les checkpoints entraînés sans signal exogène.

3.3.6 Perte VICReg

La perte VICReg [14] combine trois termes :

$$\mathcal{L}_{\text{inv}}(z_a, z_b) = \frac{1}{B}\sum_i \left\|z_a^i - z_b^i\right\|^2 \quad \text{(invariance prédictive)} \tag{13}$$

$$\mathcal{L}_{\text{var}}(z) = \frac{1}{d}\sum_j \max(0, \gamma - \text{std}(z_{\cdot,j})) \quad \text{(prévention de l'effondrement)} \tag{14}$$

$$\mathcal{L}_{\text{cov}}(z) = \frac{1}{d}\sum_{i\neq j}[\text{Cov}(z)]_{ij}^2 \quad \text{(décorrélation)} \tag{15}$$

$$\mathcal{L}_{\text{VICReg}} = \lambda_{\text{inv}}\mathcal{L}_{\text{inv}} + \lambda_{\text{var}}(\mathcal{L}_{\text{var}}(z_a) + \mathcal{L}_{\text{var}}(z_b)) + \lambda_{\text{cov}}(\mathcal{L}_{\text{cov}}(z_a) + \mathcal{L}_{\text{cov}}(z_b)) \tag{16}$$

Dans notre configuration : $\lambda_{\text{inv}} = 25$, $\lambda_{\text{var}} = 25$, $\lambda_{\text{cov}} = 1$, $\gamma = 1$. Le terme de covariance est calculé impérativement en float32 (upcast explicite) : la covariance en bf16 sur des distributions à queues épaisses accumule des erreurs de troncature qui brisent la propriété d'orthogonalité des représentations Momentum vs Volatility.

La perte VICReg régularise également la géométrie de l'espace latent. Son terme de variance est une contrainte de type hypersphère : en poussant chaque dimension à avoir un écart-type $\geq \gamma = 1$, VICReg incite les représentations à se distribuer sur la surface d'une hypersphère $\mathcal{S}^{d-1}$. Cette propriété géométrique est cruciale pour la correction hors-variété de l'ASG (Section 3.6.3).

3.4 Strate III — Prédicteur Multimodal OT-CFM

3.4.1 Continuous Flow Matching

Les marchés chaotiques admettent des futurs multimodaux : même avec une information passée parfaite, plusieurs régimes possibles coexistent. Un prédicteur déterministe s'effondrerait vers la moyenne de ces modes (*mode averaging*), produisant des prédictions toujours plausibles mais jamais correctes. Le Flow Matching permet de modéliser cette distribution multimodale.

L'OT-CFM [8] entraîne un champ de vitesse $v_\theta : \mathbb{R}^d \times [0,1] \times \mathbb{R}^{d_{\text{ctx}}} \to \mathbb{R}^d$ tel que la trajectoire ODE :

$$\frac{dx_t}{dt} = v_\theta(x_t, t, h_x), \quad x_0 \sim \mathcal{N}(0, I), \quad x_1 \approx h_{y,\text{tgt}} \tag{17}$$

transporte la distribution source $\mathcal{N}(0, I)$ vers la distribution cible (représentations futures $h_{y,\text{tgt}}$).

Objectif d'entraînement. Pour chaque point d'entraînement :

$$t \sim \mathcal{U}[0,1] \tag{18}$$

$$x_0 \sim \mathcal{N}(0, I) \tag{19}$$

$$\pi = \text{Sinkhorn-OT}(x_0, h_{y,\text{tgt}}) \quad \text{(couplage OT)} \tag{20}$$

$$x_t = (1-t)\cdot x_0[\pi] + t \cdot h_{y,\text{tgt}} \tag{21}$$

$$v_{\text{tgt}} = h_{y,\text{tgt}} - x_0[\pi] \tag{22}$$

$$\mathcal{L}_{\text{CFM}} = \|v_\theta(x_t, t, h_x) - v_{\text{tgt}}\|_{\text{MSE}}^2 \tag{23}$$

Couplage Sinkhorn (compatible JIT). Le transport optimal exacte (algorithme hongrois, $O(N^3)$) n'est pas compatible JIT. Nous utilisons le transport entropiquement régularisé via l'algorithme de Sinkhorn [9], qui converge par itérations de normalisation alternées dans l'espace log :

$$C_{ij} = \left\| x_0^{(i)} - x_1^{(j)} \right\|^2 \quad \text{(matrice de coût)} \tag{24}$$

$$\log K_{ij} = -C_{ij}/\varepsilon \tag{25}$$

$$\log u_i \leftarrow -\text{logsumexp}_j(\log K_{ij} + \log v_j) \tag{26}$$

$$\log v_j \leftarrow -\text{logsumexp}_i(\log K_{ij} + \log u_i) \tag{27}$$

après $n_{\text{iter}} = 50$ itérations via `jax.lax.scan`. L'affectation dure est extraite par argmax par ligne du plan de transport $P_{ij} = \exp(\log u_i + \log K_{ij} + \log v_j)$.

Inférence via diffrax. Au moment de la génération, l'ODE est intégrée de $t = 0$ à $t = 1$ par la méthode d'Euler via la bibliothèque diffrax :

$$x_1 = x_0 + \int_0^1 v_\theta(x_t, t, h_x)\, dt, \quad x_0 \sim \mathcal{N}(0, I) \tag{28}$$

avec $n_{\text{steps}} = 2$ (suffisant pour la régularité empirique observée). La compatibilité JIT de diffrax est essentielle : contrairement à scipy, elle ne nécessite pas de sortie vers le CPU.

3.4.2 Architecture du Champ de Vitesse

Le champ de vitesse est un MLP conditionné :

$$\text{inp}_t = [x_t \,\|\, \text{SinEmb}(t) \,\|\, h_x^{(\text{last})} \,\|\, \text{PosEmb}(\text{pos\_cibles})] \tag{29}$$

$\text{SinEmb}(t) \in \mathbb{R}^d$ est un embedding sinusoïdal du temps continu $t \in [0, 1]$. Le contexte $h_x^{(\text{last})}$ est le dernier token de la séquence contexte (représentation globale). PosEmb est un embedding appris des positions cibles.

3.5 Prédicteur Stochastique MLP

Le prédicteur P_θ opère en aval de l'encodeur contexte. Il reçoit $h_x \in \mathbb{R}^{B\times S\times d}$, des positions cibles $\text{pos} \in \mathbb{Z}^{B\times N_{\text{tgt}}}$, et un vecteur de bruit latent $z \in \mathbb{R}^{B\times N_{\text{tgt}}\times d_z}$ $(d_z = 32)$:

$$\text{ctx} = h_x[:, -1, :] \in \mathbb{R}^{B\times d} \quad \text{(dernier token — représentation causalement la plus riche)} \tag{30}$$

$$\text{inp} = [\text{ctx}_{\text{expanded}} \,\|\, \text{PosEmb}(\text{pos}) \,\|\, z] \in \mathbb{R}^{B\times N_{\text{tgt}}\times(2d+d_z)} \tag{31}$$

$$\hat{h}_y = \text{MLP}_{n_{\text{layers}}}(\text{inp}) \in \mathbb{R}^{B\times N_{\text{tgt}}\times d} \tag{32}$$

Le bruit $z \sim \mathcal{N}(0, I)$ rend le prédicteur stochastique : différentes tirages de z depuis le même contexte h_x produisent des prédictions différentes, modélisant l'incertitude irréductible sur les positions masquées. Sans bruit ($z = 0$), le prédicteur revient à une prédiction déterministe. Le prédicteur et le FlowPredictor (OT-CFM) opèrent en parallèle : le prédicteur MLP est la branche principale (perte VICReg), le FlowPredictor est la branche de distribution (perte CFM). Les deux partagent le même contexte h_x.

Masquage par blocs (*Block Masking*). Le masquage est pré-calculé dans le pipeline Grain (numpy, hors du graphe JIT) pour éviter les problèmes de tracé conditionnel :

1. Choisir un début aléatoire dans les positions disponibles.
2. Masquer un bloc contigu de longueur $b \sim \mathcal{U}[b_{\min}, b_{\max}]$ (défaut : 4–8 tokens).
3. Répéter jusqu'à atteindre le ratio cible $\rho = 0.5$ ($\pm$ un bloc).

Cette stratégie de masquage par blocs contiguous force l'encodeur à interpoler sur plusieurs tokens consécutifs, impossible à résoudre par simple copie du token voisin. Contrairement au masquage aléatoire position par position (I-JEPA images), le masquage par blocs sur séries temporelles crée des lacunes de durée variable correspondant à des régimes manquants : le modèle doit inférer le régime intermédiaire depuis ses voisins.

3.6 Strate IV — Agent TD-MPC2 et Multiverse Crossing

3.6.1 Modèle du Monde Latent

L'agent TD-MPC2 [10] apprend un modèle du monde entièrement dans l'espace latent JEPA :

— **LatentEncoder** : $f_\theta(o) = \text{LayerNorm}(\text{MLP}(o)) \in \mathbb{R}^{d_z}$, compresse l'observation (qui inclut h_x et les signaux de convergence multivers) en état latent z.
— **LatentDynamics** : $d_\theta(z, a) = \text{LayerNorm}(\text{MLP}([z, a]) + z) \in \mathbb{R}^{d_z}$, transition résiduelle (skip connection sur z).
— **RewardHead** : $r_\theta(z, a) = \text{MLP}([z, a]) \in \mathbb{R}$.

Le déroulement sur H pas est compilé via `jax.lax.scan` :

$$(z_1, \ldots, z_H), (r_1, \ldots, r_H) = \texttt{lax.scan}(d_\theta, z_0, (a_1, \ldots, a_H)) \tag{33}$$

3.6.2 Critique Distributionnel et CVaR

Deux critiques quantiles (*ensemble* TD3-style) estiment la distribution complète des retours :

$$Q_i(z, a) \in \mathbb{R}^{N_q} \quad i \in \{1, 2\}, \quad N_q = 32 \text{ quantiles}$$

La valeur pessimiste est $Q_{\min}(z, a) = \min(Q_1, Q_2)$ (par élément sur les quantiles). La CVaR_α est calculée comme la moyenne des $\lfloor \alpha N_q \rfloor$ quantiles les plus bas :

$$\text{CVaR}_\alpha(z, a) = \frac{1}{\lfloor \alpha N_q \rfloor} \sum_{i=1}^{\lfloor \alpha N_q \rfloor} \tilde{Q}_{\min,(i)}(z, a) \tag{34}$$

où $\tilde{Q}_{\min,(i)}$ désigne le i-ème quantile trié (ordre croissant). Avec $\alpha = 0.1$, $\text{CVaR}_{0.1}$ est l'espérance conditionnelle des 10% pires retours.

L'entraînement du critique utilise la perte Huber asymétrique quantile (QR-DQN [11]) :

$$\mathcal{L}_{\text{QR}}(\tau) = \mathbb{E}_{\tau \sim U}\left[|\tau - \mathbf{1}[\delta < 0]| \cdot L_\kappa(\delta) \right], \quad \delta = y - q_\tau \tag{35}$$

où L_κ est la perte Huber de seuil $\kappa = 1$, et τ sont les fractions quantiles fixes $\tau_i = (i + 0.5)/N_q$.

3.6.3 Analyse de Sensibilité Géodésique pour la Planification

L'implémentation porte le nom de fichier `multiverse_crossing.py`. Nous conservons cette désignation dans le texte tout en la décrivant ici dans ses termes techniques.

Motivation. L'analyse de sensibilité aux conditions initiales est standard en dynamique non-linéaire : deux trajectoires issues de conditions initiales δ-proches divergent en $\|\delta(t)\| \approx \|\delta(0)\| e^{\lambda t}$ si l'exposant de Lyapunov $\lambda > 0$. Nous transposons ce diagnostic dans l'espace latent : en générant M copies perturbées de h_x et en mesurant la divergence de leurs prédictions, nous obtenons un proxy de la sensibilité du modèle aux conditions initiales de la trajectoire courante. Ce proxy pilote le paramètre de risque CVaR du planning MPPI.

Perturbation Géodésique. L'approche naïve serait d'ajouter un bruit gaussien isotrope :

$$h_x^{(m)} = h_x + \sigma \cdot \varepsilon_m, \quad \varepsilon_m \sim \mathcal{N}(0, I)$$

Mais la régularisation VICReg pousse les représentations vers l'hypersphère $\mathcal{S}^{d-1}$. Un bruit additif isotrope projette $h_x^{(m)}$ *hors de la variété* représentationnelle apprise par l'encodeur, produisant des entrées pour lesquelles le prédicteur OT-CFM n'a aucune garantie de performance (distribution *out-of-manifold*).

Correction. Nous implémentons une perturbation géodésique sur $\mathcal{S}^{d-1}$:

1. **Projection sur le plan tangent en** h_x : le bruit brut $\varepsilon \in \mathbb{R}^d$ est projeté orthogonalement à la direction radiale $\hat{h}_x = h_x / \|h_x\|$:
$$\varepsilon_\perp = \varepsilon - (\varepsilon \cdot \hat{h}_x)\hat{h}_x \quad \text{(composante tangentielle uniquement)}$$
2. **Perturbation dans le plan tangent** :
$$h'_x = h_x + \sigma \cdot \varepsilon_\perp$$
3. **Reprojection sur la sphère de rayon** $r = \|h_x\|$:
$$h_x^{(m)} = h'_x \cdot \frac{\|h_x\|}{\|h'_x\|}$$

Cette procédure est un mouvement géodésique du premier ordre sur $\mathcal{S}^{d-1}$, équivalent à se déplacer le long d'un grand cercle. Elle garantit que $\left\|h_x^{(m)}\right\| = \|h_x\|$ et que les points perturbés restent sur la variété apprise.

Proposition 1. *Soit $h_x \in \mathbb{R}^d \setminus \{0\}$ et $\varepsilon \sim \mathcal{N}(0, I)$. La perturbation géodésique produit $h_x^{(m)}$ tel que : (i) $\left\|h_x^{(m)}\right\| = \|h_x\|$ exactement, (ii) l'angle entre $h_x^{(m)}$ et h_x est $O(\sigma / \|h_x\|)$ pour σ petit, (iii) $\mathbb{E}[\varepsilon_\perp] = 0$ et $\mathbb{E}[\|\varepsilon_\perp\|^2] = (d-1)\sigma^2$.*

Métriques de Convergence. Pour $M = 5$ univers, chaque univers génère $N = 16$ trajectoires futures via OT-CFM. À chaque pas décisionnel t, on dispose de $M \times N = 80$ trajectoires :

$$\text{mv\_means}_m = \frac{1}{N} \sum_{n=1}^{N} h_{\text{fut}}^{(m,n)} \in \mathbb{R}^d \quad \text{(moyenne par univers)} \tag{36}$$

$$\sigma_{\text{inter}} = \frac{1}{d} \sum_{k=1}^{d} \text{std}_m \left([\text{mv\_means}_m]_k \right) \quad \text{(divergence inter-univers)} \tag{37}$$

$$\sigma_{\text{intra}} = \frac{1}{d} \sum_{k=1}^{d} \text{std}_{m,n} \left([h_{\text{fut}}^{(m,n)}]_k \right) \quad \text{(bruit intra-univers)} \tag{38}$$

$$\text{score} = \frac{1}{1 + \sigma_{\text{inter}} / (\sigma_{\text{intra}} + \varepsilon)} \in [0, 1] \tag{39}$$

Quand $\sigma_{\text{inter}} \ll \sigma_{\text{intra}}$, les univers convergent (accord) et `score` $\to 1$. Quand $\sigma_{\text{inter}} \gg \sigma_{\text{intra}}$, les univers divergent (désaccord) et `score` $\to 0$.

Indice de Bifurcation. La dimensionnalité effective des futurs inter-univers est mesurée via l'entropie spectrale de la matrice de Gram :

$$G = \frac{1}{M-1}(X_c X_c^\top) \in \mathbb{R}^{M\times M}, \quad X_c = \text{mv\_means} - \overline{\text{mv\_means}} \tag{40}$$

$$\lambda_1, \ldots, \lambda_M = \text{eigvalsh}(G), \quad \tilde{\lambda}_i = \lambda_i / \sum_j \lambda_j \tag{41}$$

$$H = -\sum_i \tilde{\lambda}_i \log(\tilde{\lambda}_i + \varepsilon) \tag{42}$$

$$\text{bifurcation} = \exp(H) \in [1, M] \tag{43}$$

`bifurcation` $= 1$ correspond à un seul mode (univers alignés), `bifurcation` $= M$ à M modes orthogonaux (chaos pur). L'utilisation de la matrice de Gram $M \times M$ (au lieu de la covariance $d \times d$) réduit la complexité de $O(d^2)$ à $O(M^2 d)$ avec $M \ll d$.

Proxy de Lyapunov.

$$\lambda_{\text{proxy}}(t) = \frac{\log(\sigma_{\text{inter}}(t)/\sigma + \varepsilon)}{\max(t, 1)} \tag{44}$$

où $\sigma = 0.01$ est la magnitude de la perturbation initiale. $\lambda_{\text{proxy}} > 0$ indique une divergence exponentielle (chaos), $\lambda_{\text{proxy}} < 0$ une contraction (régime stable).

CVaR Dynamique.

$$\boxed{\alpha(t) = \alpha_{\min} + (\alpha_{\max} - \alpha_{\min}) \cdot \text{score}(t)} \tag{45}$$

avec $\alpha_{\min} = 0.1$ et $\alpha_{\max} = 0.4$. Ainsi :

- `score` $= 1$ (univers convergent) $\Rightarrow \alpha = 0.4$ (aggressif, 40% des pires retours)
- `score` $= 0$ (chaos) $\Rightarrow \alpha = 0.1$ (conservateur, 10% des pires retours, no-trade si CVaR < 0)

Planning MPPI avec CVaR Dynamique. L'agent MPPI tire $K = 512$ séquences d'actions de $H = 5$ pas, évalue leurs retours imaginés via le modèle du monde, et agrège avec $\text{CVaR}_{\alpha(t)}$ au terminal :

$$a_1, \ldots, a_H \sim \mathcal{N}(\mu, \sigma^2 I) \quad \text{(K séquences)} \tag{46}$$

$$G_k = \sum_{h=1}^{H} \gamma^{h-1} r_h^{(k)} + \gamma^H \text{CVaR}_{\alpha(t)}(Q_{\min}(z_H^{(k)}, \pi(z_H^{(k)}))) \tag{47}$$

$$w_k = \frac{\exp(G_k/\tau_{\text{plan}})}{\sum_j \exp(G_j/\tau_{\text{plan}})} \tag{48}$$

$$\mu \leftarrow \sum_k w_k \cdot (a_1^{(k)}, \ldots, a_H^{(k)}) \tag{49}$$

Cette boucle est déroulée en Python (6 itérations : constante connue à la compilation), permettant à XLA de fusionner tous les noyaux en un seul graphe HLO sans barrière de synchronisation. Le déroulement interne (H pas de modèle du monde) reste un `lax.scan` (séquentiel par nature).

4 Implémentation JAX/XLA et Ingénierie TPU

4.1 Noyau SSD par Blocs (Chunked SSD)

4.1.1 Motivation

L'implémentation naïve de Mamba utilise `jax.lax.associative_scan` pour calculer la récurrence SSM en parallèle. Sur TPU, cette approche génère des accès mémoire non-coalisés et

sous-utilise le MXU (Matrix Multiplication Unit), dont les tuiles de taille 128×128 nécessitent des opérations matricielles denses. La SSD [4] montre que la récurrence SSM peut être reformulée comme une série de matmuls par blocs.

4.1.2 Algorithme SSD par Blocs

Pour un chunk de taille C_s tokens (avec $C_s = 128$) :

1. **Discrétisation** : $\Delta = \text{softplus}(\Delta_{\text{raw}}) \in \mathbb{R}^{B \times L \times H}$, mise à zéro sur les positions weekend.
2. **Cumsum** : $\text{cs}_t = \sum_{s=0}^{t} \Delta_s$ (facteur de déclin accumulé). **En float32 :** voir Section 4.2.
3. **Déclin pairwise intra-chunk** :

$$\text{cs\_diff}_{i,j} = \text{cs}_i - \text{cs}_j \quad \in \mathbb{R}^{C_s \times C_s}$$

$$\text{decay}_{i,j,n} = A_n \cdot \text{cs\_diff}_{i,j} \leq 0 \quad \text{(stable, } A < 0\text{)}$$

4. **Matrice de Gram L intra-chunk** : *une tuile MXU exacte*

$$L_{i,j} = \sum_n C_i^{(n)} \cdot B_j^{(n)} \cdot \Delta_j \cdot e^{A_n \cdot \text{cs\_diff}_{i,j}}$$

   via `jnp.einsum("in,jn,ijn->ij", C, dt_B, exp_decay)`, tril pour la causalité.
5. **Sortie intra-chunk** :

$$y_{\text{intra}} = L \cdot x \in \mathbb{R}^{C_s \times P} \quad \text{(matmul MXU parfait)}$$

6. **Contribution inter-chunk** :

$$y_{\text{inter}} = \hat{C} \cdot h_{\text{init}} \in \mathbb{R}^{C_s \times P}, \quad \hat{C}_t = C_t \cdot e^{A \cdot \text{cs}_t}$$

7. **État final propagé** :

$$h_{\text{final}} = e^{A \cdot \text{cs}_{\text{end}}} \cdot h_{\text{init}} + \sum_t \hat{B}_t \cdot x_t^\top$$

   calculé en float32 pour l'exponentielle, casté en bf16 pour le stockage.

La boucle sur les chunks utilise `jax.lax.scan`. L'axe tête est vectorisé via `jax.vmap`. L'alignement $C_s = 128 =$ taille MXU garantit un remplissage parfait des tuiles sans padding.

4.1.3 Avantages computationnels

Sur TPU v5p avec $C_s = 128$ et $P = 128$:

— Étape 4 (`einsum`) : matrice $(128, 128, N)$ contractée vers $(128, 128)$ — le cœur est un produit $(128, N) \times (N, 128)$ qui remplit exactement une tuile MXU.
— Étape 5 (`L @ x`) : matmul $(128, 128) \times (128, P)$ — une tuile MXU par tête.
— Contrairement à `associative_scan` : aucune dépendance séquentielle intra-chunk, parallélisation totale sur B et H.

4.2 Stabilité Numérique bfloat16

4.2.1 Contexte

Le format bfloat16 alloue 8 bits d'exposant (identique à float32) et seulement 7 bits de mantisse (vs 23 pour float32). L'erreur relative de troncature est $\varepsilon_{\text{bf16}} \approx 7.8 \times 10^{-3}$, contre $\varepsilon_{\text{f32}} \approx 6 \times 10^{-8}$.

4.2.2 Analyse de l'Anomalie NaN au Pas 2 750

Les logs d'entraînement montrent une divergence déterministe :

```
2026-02-19 12:53:22 step 2700 | loss 4860.1282 | cfm 1.4618
2026-02-19 12:59:56 step 2750 | loss nan       | cfm nan
```

Le crash suit un signal d'arrêt SIGTERM de XLA (`xla::PjRtCApiLoadedExecutable::Execute()`), confirmant que le NaN provient du calcul GPU/TPU et non du code Python.

Cause racine. Le cumsum sur la dimension temporelle en bf16 :

$$\mathrm{cs}_t = \sum_{s=0}^{t} \Delta_s \quad \text{(bf16)}$$

accumule des erreurs de troncature. Avec $L = 128$ tokens et $\Delta_s \approx 0.1$ typiquement : $\mathrm{cs}_{128} \approx 12.8$. En bf16, la précision relative est $\varepsilon_{\text{bf16}} \approx 10^{-2}$, soit une erreur absolue de ≈ 0.128 sur le cumsum final. À mesure que l'entraînement progresse (les Δ augmentent en magnitude), l'erreur cumulée croît. Au pas 2 750, les valeurs $\exp(A \cdot \mathrm{cs})$ commencent à produire des $\pm\infty$ en bf16 sur certaines têtes, propageant des NaN via les matmuls.

Solution : upcast asymétrique. Nous conservons bf16 pour les multiplications matricielles (qui exploitent les Tensor Cores) et forcé float32 uniquement pour les accumulations temporelles :

```
# Cumsum CRITIQUE: float32 pour eviter la derive bf16
# bf16 mantisse (7 bits) perd ~4 bits de precision sur 128 pas.
cs = jnp.cumsum(dt_c.astype(jnp.float32), axis=2).astype(dt_c.dtype)

# Etat final: exponentielle en float32, storage en bf16
cs_end_f32 = cs_ch[-1].astype(jnp.float32)
A_h_f32 = A_h.astype(jnp.float32)
decay_end = jnp.exp(A_h_f32 * cs_end_f32).astype(x_ch.dtype)

# Accumulation h_final: matmul en float32, cast result
h_accum = (B_hat_end.astype(jnp.float32).T @ x_ch.astype(jnp.float32))
h_final = h_final + h_accum.astype(x_ch.dtype)
```

Listing 1 – Upcast asymétrique dans chunked_ssd.py

Après cette correction, aucun NaN n'est observé même sans gradient clipping agressif. L'ajout de `optax.clip_by_global_norm(1.0)` fournit une couche de sécurité supplémentaire.

Impact sur les performances. L'upcast sélectif sur les accumulations (non les matmuls) préserve $\sim 95\%$ de l'accélération bf16 tout en éliminant les NaN. Le surcoût estimé est $< 2\%$ du temps de calcul total.

4.3 Auto-Sharder GSPMD Topologique

4.3.1 Topologie du Tore 3D ICI des TPU v5p

Les TPU v5p sont organisés en tores 3D via Inter-Chip Interconnect (ICI). Chaque *tray* contient 4 chips reliés par des liens ICI à haut débit ($\sim$340 Go/s par lien). Les communications inter-tray passent par des liens plus lents. Pour maximiser le débit FSDP (qui nécessite des all-gather/reduce-scatter fréquents), les communications FSDP doivent rester intra-tray.

4.3.2 Maillage 2D Virtuel

Nous utilisons un maillage 2D `('data', 'fsdp')` avec `fsdp = 4` (correspondant aux 4 chips d'un tray) :

TABLE 1 – Configurations automatiques de maillage GSPMD (Auto-Sharder)

Pod	Chips	Data	FSDP	Mémoire/chip	Stratégie
v5p-8	8	8	1	95 Go	DP pur
v5p-32	32	8	4	95 Go	DP+FSDP
v5p-128	128	32	4	95 Go	DP+FSDP
v5p-768	768	192	4	95 Go	DP+FSDP

La fonction `create_device_mesh` de JAX mappe automatiquement le maillage virtuel 2D sur la topologie physique 3D, minimisant les sauts ICI en alignant l'axe FSDP sur les chips intra-tray.

4.3.3 Stratégie de Sharding

— **Données de batch** : `NamedSharding(mesh, P("data"))` — chaque replica data-parallel traite batch_global/data_dim exemples.
— **Paramètres et état optimiseur** : `NamedSharding(mesh, P("fsdp"))` — FSDP shard l'axe 0 de chaque tenseur sur les 4 chips du tray.
— **Clés RNG** : `jax.random.split(rng, data_dim)` — chaque replica reçoit une clé différente pour dropout et augmentation indépendants.

Avec FSDP=4 sur un modèle de 150M paramètres (300 Mo bf16), chaque chip ne stocke que 75 Mo de paramètres, bien en dessous des 95 Go HBM disponibles. Pour le modèle 7B (14 Go bf16), FSDP/4 = 3.5 Go par chip — très confortable.

4.4 Pipeline de Données Grain (AsyncIO Multi-Hôtes)

Le chargement de données utilise DeepMind Grain avec le format ArrayRecord. La détection automatique du nombre de workers est :

```
2026-02-19 06:39:16 Grain worker_count: 32 (cpu_count=180)
```

Sur v6e-8 (180 CPU logiques pour 8 chips), Grain utilise 32 processus de pré-chargement via `SharedMemoryArray`, remplissant un buffer de préchargement de 128 batches. Chaque batch est ensuite transféré vers les HBM des TPU via `jax.device_put` asynchrone avec double-buffering.

Pour le pipeline à l'échelle (v5p-768, 192 hôtes) : le buffer de préchargement passe à 512 batches (`configs/xl_7b.yaml`), et chaque hôte lit depuis GCS via la couche de stockage objet, évitant le goulot d'étranglement d'un NFS partagé.

4.5 Flags XLA de Production

Treize flags `LIBTPU_INIT_ARGS` configurent l'exécuteur XLA pour maximiser les performances :

— Fusion des collectives asynchrones (all-reduce)
— Ordonnancement avec masquage de latence (overlap compute/communication)
— Fusion agressive des boucles (SSM chunked + attention combinés si applicable)
— Modèle de coût expérimental pour mieux décider des fusions intra-op

4.6 Remat (Gradient Checkpointing)

Le v6e (31 Go HBM) nécessite le checkpointing de gradient (`use_remat = True`) pour permettre les séquences de 128 tokens avec $d_{\text{model}} = 256$: sans remat, les activations intermédiaires saturent la mémoire. Sur v5p (95 Go HBM), remat est désactivé (`use_remat = False`), préférant la vitesse à l'économie mémoire. Cette configuration est spécifiée dans les YAML T-Shirt et auto-chargée.

4.7 TrainState — Optimiseur, EMA et SGDR

4.7.1 FinJEPATrainState

L'état d'entraînement étend `flax.train_state.TrainState` avec trois champs supplémentaires :

- `target_params` : copie EMA des paramètres de `context_encoder` uniquement (pas du prédicteur).
- `tau` : momentum EMA courant, annelé par cosinus de $\tau_{\text{start}} = 0.996$ vers $\tau_{\text{end}} = 1.0$ sur `anneal_epochs = 100`.
- `rng` : clé PRNG JAX pour le bruit de masquage et le bruit z.

La mise à jour EMA à chaque pas :

$$\phi_{t+1} = \tau \cdot \phi_t + (1 - \tau) \cdot \theta_t \tag{50}$$

où θ sont les paramètres en ligne et ϕ les paramètres cibles. À $\tau = 0.996$, la cible suit le modèle en ligne avec un retard moyen d'environ $1/(1 - 0.996) \approx 250$ pas — suffisamment stable pour fournir des pseudo-étiquettes cohérentes au prédicteur.

4.7.2 Optimiseur — AdamW + SGDR + zero_nans

AdamW avec $\beta_2 = 0.95$. Contrairement au $\beta_2 = 0.999$ standard, nous utilisons $\beta_2 = 0.95$ pour les SSM. Cette valeur plus basse donne plus de poids aux gradients récents, accélérant l'adaptation aux changements de régime dans les paramètres B, C, Δ. Les SSM sont particulièrement sensibles à ce paramètre : leurs gradients peuvent osciller rapidement en réponse aux transitions de régime, et un β_2 élevé sous-réagirait.

SGDR (Warm Restarts Stochastic Gradient Descent). L'optimiseur utilise $n_{\text{restarts}} = 4$ redémarrages chauds cosinus avec période doublée :

- Cycle 1 : T pas (cosinus de LR_{peak} à 0)
- Cycle 2 : $2T$ pas
- Cycle 3 : $4T$ pas
- Cycle 4 : $8T$ pas

où $T = \lfloor T_{\text{total}}/(2^{n_r} - 1) \rfloor$. Les redémarrages chauds permettent à l'optimiseur de s'échapper des minima locaux étroits en ré-augmentant périodiquement le LR. Le commentaire du code identifie explicitement les minima locaux comme cause probable du NaN au pas 2 750 : l'optimiseur était potentiellement coincé dans un voisinage instable.

zero_nans. `optax.zero_nans()` est chaîné avant AdamW : tout gradient contenant NaN ou $\pm\infty$ est remplacé par zéro avant la mise à jour. Cette sécurité permet à l'entraînement de continuer après une explosion isolée sans propager les NaN dans les paramètres. Combiné avec `clip_by_global_norm(1.0)`, le pipeline est robuste contre les instabilités transitoires.

```
tx = optax.chain(
    optax.clip_by_global_norm(grad_clip),  # clip avant zero_nans
    optax.zero_nans(),                     # zero les NaN residuels
    optax.adamw(lr_schedule, weight_decay, b2=0.95),
)
```

Listing 2 – Chaîne d'optimiseur complète (train_state.py)

4.8 Étape d'Entraînement JIT — Buffer Donation

La fonction de training step est compilée une seule fois via `@jax.jit` :

```
@partial(jax.jit, static_argnums=(2,), donate_argnums=(0, 1))
def train_step(state, batch, model):
    ...
```

Listing 3 – Signature du train_step avec donate_argnums (train_step.py)

donate_argnums=(0, 1). L'argument `donate_argnums` indique à JAX que les buffers mémoire de `state` et `batch` peuvent être réutilisés après l'exécution. XLA peut alors *in-place* écrire les nouveaux paramètres dans la même mémoire que les anciens, évitant une double allocation. Sur v5p-768 (7B params), la mémoire des paramètres est ∼14 Go par pool FSDP : sans donation, chaque step allouerait 14 Go supplémentaires le temps du calcul.

static_argnums=(2,). Le module Flax `model` est un argument statique (non-tracé) : ses attributs structuraux (dimensions, types) sont fixés à la compilation. Cela permet à XLA de spécialiser le graphe pour l'architecture exacte.

Flux complet d'un pas.

1. `jax.random.split(state.rng)` → clés dropout et bruit.
2. `jax.value_and_grad(loss_fn)` → perte + gradients.
3. `optax.global_norm(grads)` → norme gradient (monitoring).
4. `state.apply_gradients(grads)` → mise à jour params + opt_state.
5. `update_target_ema(state)` → $\phi \leftarrow \tau\phi + (1-\tau)\theta$.
6. `state.replace(rng=rng)` → avance le PRNG.

Tout cela dans un seul graphe XLA JIT-compilé, sans retour vers le CPU entre les étapes.

4.9 Pipeline Grain — Multi-Hôtes Asynchrone

4.9.1 Architecture du Pipeline

Le pipeline de données suit le schéma :

`GCS/ArrayRecord` → `grain.ArrayRecordDataSource` → `grain.IndexSampler` → `ParseAndMask` → `grain.Batch` → `DataLoader`

Sharding multi-hôtes automatique. `grain.ShardByJaxProcess()` partitionne automatiquement les shards entre les hôtes JAX. Sur un pod v5p-32 (4 hôtes de 8 chips), chaque hôte lit ∼108 des 433 shards, sans coordination explicite. Cela est possible parce que chaque hôte connaît son indice de processus via `jax.process_index()`.

Split train/val déterministe. La partition des paires en train/val est faite par hachage MD5 du nom de la paire :

```python
def _pair_to_split(pair_name: str, val_ratio: float = 0.2) -> str:
    h = int(hashlib.md5(pair_name.encode()).hexdigest(), 16)
    return "val" if (h % 1000) < int(val_ratio * 1000) else "train"
```

Listing 4 – Split déterministe par hash (grain_loader.py)

Ainsi, BTC/USDT est toujours dans le même split quelle que soit l'exécution, garantissant la reproductibilité des métriques de validation. 20% des 432 paires (≈ 86 paires) sont en validation, soit ≈ 6M tokens de validation.

ParseAndMask Transform. Grain appelle `ParseAndMask.map()` dans des processus worker séparés (32 workers sur v6e, 180 CPUs disponibles). Chaque appel :

1. Désérialise le protobuf `tf.train.Example` (token_indices, weekend_mask, exo_clock).
2. Génère le masque par blocs (numpy, pas JAX) via `generate_block_mask()`.
3. Extrait les positions cibles et les padde à `max_targets = seq_len × 0.5 + 8`.

Le pré-calcul dans le worker évite deux problèmes JAX : (i) le masquage aléatoire dans le graphe JIT nécessiterait des PRNGKey tracées, compliquant le code ; (ii) le masquage varie par exemple et ne peut pas être constants folded.

Auto-détection des workers.

```python
cpus = os.cpu_count() or 4
count = min(cpus, 32)    # plafond : rendements decroissants apres 32
count = max(count, 2)    # minimum : recouvrement IO/compute
```

Listing 5 – Auto-tuning workers (grain_loader.py)

Sur v6e (180 CPUs logiques) : 32 workers. Le log de training le confirme :

```
Grain worker_count auto-detected: 32 (cpu_count=180)
Grain pool will use 32 processes.
Grain pool started all child processes.
```

4.9.2 Format ArrayRecord

Chaque shard contient les séquences d'une paire (432 paires → 433 shards, la paire la plus longue est découpée). Chaque enregistrement stocke :

- `token_indices` : `int64[128]` — séquence de codes FSQ.
- `weekend_mask` : `float32[128]` — indicateur weekend (1.0 = weekend).
- `exo_clock` : `float32[128×2]` — [volatilité réalisée, volume normalisé].
- `pair_name` : `bytes` — nom de la paire (BTC/USDT, etc.).
- `original_len` : `int64` — longueur avant padding.

Le padding à 128 tokens est nécessaire pour que le batch soit un tenseur dense. Les positions paddées sont masquées dans `target_mask`.

4.10 Environnement Gymnasium — Espace d'Observation

4.10.1 Conception de l'Espace d'Observation

L'environnement `LatentMultiverseEnv` reçoit une liste d'entrées pré-calculées offline (`MultiverseTrajectory` chacune contenant `future_latents` de forme $(M, N, N_{\text{tgt}}, d)$.

L'observation à l'instant t est un vecteur concaténé :

TABLE 2 – Décomposition de l'espace d'observation (obs_dim = $3d + 4N_{\text{tgt}} + 14$)

Champ	Dim	Source	Nature
`h_x_pooled`	d	JEPA context (pooling)	Statique
`future_mean_t`	d	Moy. $M \times N$ futurs au pas t	Dynamique
`future_std_t`	d	Std $M \times N$ futurs au pas t	Dynamique
`close_stats`	$3N_{\text{tgt}}$	[moy, std, skew] des rendements	Statique
`revin_stds`	5	Volatilité des 5 canaux OHLCV	Statique
`delta_mu`	1	Tendance macro normalisée	Statique
`step_progress`	1	t/N_{tgt}	Dynamique
`realized_returns`	N_{tgt}	Rendements cumulés observés	Dynamique
`position`	1	Position portefeuille actuelle	Dynamique
`cumulative_pnl`	1	PnL cumulé de l'épisode	Dynamique
`convergence_score`	1	Accord inter-univers $\in [0, 1]$	Dynamique
`divergence_rate`	1	$\Delta(\sigma_{\text{inter}})$ temporel	Dynamique
`inter_mv_std`	1	Spread des moyennes d'univers	Dynamique
`bifurcation_index`	1	Nb de modes latents	Dynamique
`lyapunov_proxy`	1	Indicateur chaos	Dynamique

Total : $3d + 3N_{\text{tgt}} + 14$ avec $N_{\text{tgt}} = 8$, $d = 128 \Rightarrow$ **422 dimensions**

Soft Gate Marché Mort. Le filtre de marché mort est implémenté comme une porte sigmoïde douce :

$$g = \sigma\big((\text{vol\_relative} - \theta) \times 10^5\big), \quad \theta = 10^{-4} \tag{51}$$

où vol_relative $= \sigma_{\text{close}}/(\mu_{\text{close}} + \varepsilon)$. Tous les champs dynamiques sont multipliés par g : si le marché est mort (vol_relative $< 10^{-4}$), $g \approx 0$ et l'observation est nulle, guidant l'agent vers une action nulle (no-trade). La pente 10^5 rend la porte quasi-discrète tout en gardant un gradient non-nul pour la backpropagation de l'agent.

Cette conception est supérieure à un filtre booléen qui provoquerait du gradient starvation : le gradient $\partial g/\partial$vol_relative est non-nul au point de transition, permettant à l'agent d'apprendre à calibrer son seuil de marché mort.

Récompense.

$$r_t = \text{pos}_t \cdot \log\left(\frac{p_{t+1}}{p_t}\right) - \lambda_{\text{TC}} \cdot |\text{pos}_t - \text{pos}_{t-1}| \tag{52}$$

avec $\lambda_{\text{TC}} = 0.002$ (20 bps de coût de transaction). La position pos $\in [-1, 1]$ est la sortie directe de l'acteur après tanh.

4.11 Replay Buffer — Double Buffering Asynchrone

4.11.1 Problème : Synchronicité de jnp.array

La conversion `jnp.array(numpy_array)` est *synchrone* : elle bloque le thread hôte jusqu'à la fin du transfert Host-to-Device (H2D). Sur TPU, chaque appel ajoute $\sim$100 μs de latence de dispatch, soit :

$$5 \text{ arrays} \times 100\,\mu\text{s} = 500\,\mu\text{s/batch}$$

À 0.1 pas/s (v6e), le pas dure $\sim$8 secondes : 500 μs est négligeable. Mais sur v5p à $\sim$1 pas/s, cela représenterait 0.05% d'overhead — toujours acceptable. La vraie valeur est sur les workloads d'inférence intensive ou de fine-tuning rapide.

4.11.2 Solution : jax.device_put Asynchrone + Double Buffering

`jax.device_put(numpy_array)` retourne immédiatement un *future* DeviceArray. Le transfert H2D se fait en parallèle du calcul TPU. Le double-buffering précharge le batch $N+1$ pendant que le TPU calcule sur le batch N :

```
def sample_async(self, batch_size):
    if self._prefetched is not None:
        batch = self._prefetched                    # deja sur device
        self._prefetched = self._dispatch_async(batch_size)  # lance N+1
        return batch
    else:
        self._prefetched = self._dispatch_async(batch_size)  # premiere fois
        return self._dispatch_async(batch_size)
```

Listing 6 – Double buffering dans replay_buffer.py

Le diagramme de recouvrement :

```
Pas N:   [TPU: calcul batch N          ] [Host: H2D batch N+1]
Pas N+1: [TPU: calcul batch N+1        ] [Host: H2D batch N+2]
```

La latence H2D ($\sim$50 μs pour 1 Mo) est complètement masquée par le calcul TPU ($\sim$8 secondes sur v6e).

5 Résultats Expérimentaux

5.1 Configuration Expérimentale

TABLE 3 – Configuration de l'expérience de validation (TPU v6e-8)

Paramètre	Valeur
Matériel	TPU v6e-8 (Trillium), 31 Go HBM/chip
Zone	`europe-west4-a`
Paramètres	3 930 568 (d_model=256, 6 couches, 2 têtes)
Séquence	128 tokens (bougies OHLCV 1h)
Batch global	1 792 séquences (224 / chip)
Précision	bfloat16 + float32 (accumulations)
LR	10^{-4} (AdamW, warmup 10 epochs)
Remat	Activé (31 Go HBM)
Données	433 shards ArrayRecord (Binance Futures 1h, 432 paires)

Le modèle de 3,93M paramètres (7,9 Mo en bf16) occupe $\sim$29 Go des 31 Go disponibles sur v6e-8 en comptant les activations intermédiaires avec remat. La limite mémoire est contraignante : sans remat, ce modèle ne tiendrait pas.

5.2 Courbes d'Entraînement

Analyse de la convergence. Trois phases distinctes apparaissent :

— **Phase I (pas 50–500) : apprentissage rapide.** La perte chute de 7448 à 6307 (-15%). L'encodeur apprend rapidement les patterns basiques : direction de la tendance, niveau de volatilité relatif.

TABLE 4 – Progression de la perte totale (VICReg + CFM) et CFM seule

Pas	Perte totale	CFM	Δ total	Δ CFM
50	7 448.62	2.1736	–	–
100	6 942.07	2.1268	−6.8%	−2.2%
250	6 420.03	2.0119	−13.8%	−7.4%
500	6 307.47	1.9446	−15.3%	−10.5%
1 000	5 298.09	1.6440	−28.9%	−24.4%
1 500	5 011.39	1.5189	−32.7%	−30.1%
2 000	4 898.53	1.4750	−34.2%	−32.1%
2 500	4 858.44	1.4516	−34.8%	−33.2%
2 600	4 839.87	1.4575	**−35.0%**	−33.0%
2 700	4 860.13	1.4618	−34.7%	−32.8%
2 750	NaN	NaN	Divergence bf16	

— **Phase II (pas 500–2 000) : apprentissage structurel.** La perte passe de 6307 à 4899 (−22% supplémentaires). Le prédicteur CFM converge rapidement : -24% sur cette phase seul. L'encodeur commence à capturer les régimes à moyen terme.

— **Phase III (pas 2 000–2 700) : plateau apparent.** La perte ralentit (−1% sur 700 pas). Ce plateau est caractéristique de la convergence vers un minimum local sur ce pod sous-dimensionné. La perte CFM oscille légèrement (1.47–1.46), suggérant une exploration dans un voisinage du minimum.

Performance temporelle. Le temps entre les pas 100 et 2700 est de ∼5h45 pour 2600 pas, soit ∼7.96 secondes/pas ou 0.126 pas/s. Ceci correspond à :

— ∼5% de MFU (Model FLOPs Utilization)
— Contrainte par la mémoire HBM (29/31 Go occupés, forte pression de bande passante mémoire)
— Le v6e a un ratio MXU/mémoire élevé : conçu pour l'inférence, pas l'entraînement

Sur v5p-8 (95 Go HBM, meilleur ratio compute/mémoire), la même configuration atteindrait ∼ 0.8–1.0 pas/s estimé, soit une MFU de 35–40%.

5.3 Note d'Implémentation : Stabilité bf16 dans le Noyau SSD

La trace de stack XLA du log confirme l'origine dans le noyau d'exécution TPU :

```
xla::PjRtCApiLoadedExecutable::Execute()
xla::ifrt::PjRtLoadedExecutable::Execute()
jax::(anonymous namespace)::PjitFunction::Call()
```

Le signal SIGTERM est émis par XLA suite à la détection d'un NaN dans le tenseur de sortie. La trace mémoire (`801625024` bytes ≈ 800 Mo) correspond à la taille d'activation d'un batch de 1792 séquences avec remat.

La séquence causale confirmée :

1. Pas 2700 : perte marginalement plus haute que 2600 (+0.4%) — les gradients accumulent déjà des erreurs bf16.
2. Pas 2750 : le cumsum bf16 d'un état SSM particulier dépasse la plage représentable, produisant $\pm\infty$ en bf16 → NaN dans l'exponentielle → propagation NaN via matmuls.
3. La correction (upcast float32 du cumsum) est appliquée et le problème ne se reproduit plus.

5.4 Résultats de Démarrage de l'Infrastructure

L'initialisation du pipeline JAX prend ∼3 minutes et 7 secondes entre le lancement (`06:36:09`) et le premier log d'entraînement (`06:39:16`) :

— Compilation XLA du graphe computationnel (JIT, première exécution)
— Initialisation des 32 processus Grain avec `SharedMemoryArray`
— Chargement du checkpoint si existant (`Orbax`)
— Init du batch fictif pour la trace de forme (B=2, S=128, N_tgt=72)

Cette latence de démarrage est amortie sur les longues sessions d'entraînement.

5.5 Analyse Fine de la Convergence

5.5.1 Taux de Convergence par Composante

À partir des logs, nous pouvons calculer les taux de réduction de perte par intervalle :

TABLE 5 – Taux de convergence par phase (perte totale et CFM)

Phase	Pas	Δ Loss	Δ/pas	Δ CFM	Δ CFM/pas
Apprentissage rapide	50–500	−1141	−2.53/pas	−0.229	-5.1×10^{-4}
Apprentissage structurel	500–2000	−1296	−0.86/pas	−0.469	-3.1×10^{-4}
Plateau et oscillations	2000–2700	−39	−0.06/pas	−0.025	-3.6×10^{-5}

La Phase I est dominée par l'apprentissage de la VICReg : l'encodeur apprend rapidement à produire des représentations non-nulles et non-collapées. La Phase II voit le prédicteur CFM converger beaucoup plus vite que la VICReg (taux relatif 3.6× plus élevé) : une fois les représentations cibles stables (encodeur EMA), le champ de vitesse OT-CFM apprend rapidement à transporter $\mathcal{N}(0, I)$ vers ces cibles. La Phase III montre un plateau typique d'un minimum local : la perte VICReg oscille légèrement (4860–4840) tandis que la CFM continue à descendre lentement.

5.5.2 Analyse des Checkpoints

Cinq checkpoints ont été sauvegardés : pas 500, 1000, 1500, 2000, 2500. Chaque checkpoint pèse **26,4 Mo** (log : `Saved state.pkl (26.4 MB)`). Ce poids inclut :

— Paramètres du modèle en bf16 : 3.93M params × 2 octets ≈ 7.9 Mo.
— Paramètres EMA cible (context_encoder uniquement, ∼6M) ≈ 5.8 Mo.
— État optimiseur AdamW (moments premier et second) ≈ 12.7 Mo.
— Scalaires divers (step, tau, rng) < 0.1 Mo.

La sauvegarde toutes les 500 steps (∼70 minutes sur v6e) est conservatrice : en mode préemptible (v5p TRC), l'intervalle par défaut est **250 steps** pour limiter la perte de calcul en cas de préemption.

5.5.3 Diagnostics XLA Post-NaN

La trace de stack complète du NaN :

```
xla::PjRtCApiLoadedExecutable::Execute()
xla::ifrt::PjRtLoadedExecutable::Execute()
jax::(anonymous namespace)::PjitFunction::Call()
_PyEval_EvalFrameDefault
```

confirme que l'exception est levée au niveau du kernel XLA (pas du Python), après la synchronisation des résultats TPU. Les adresses mémoire de la trace (∼800 Mo) correspondent à la taille d'un activations buffer de 1792 séquences × 128 tokens × 256 dimensions en bf16.

Le NaN se propage sans s'arrêter (pas 2750 à 2950, tous NaN) parce que l'optimiseur AdamW avec NaN dans les gradients propage les NaN dans le vecteur de moment $m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$. Sans `zero_nans()`, tous les paramètres sont corrompus en un seul pas. **Avec `zero_nans()`, les gradients NaN sont remplacés par zéro avant la mise à jour : le modèle survit à l'explosion.**

5.5.4 Estimation du MFU

Le Model FLOPs Utilization se calcule :

$$\text{MFU} = \frac{\text{FLOPs théoriques/pas}}{\text{FLOPs matériels} \times \text{temps/pas}} \tag{53}$$

Pour notre modèle de 3.93M params, avec $B = 1792$, $S = 128$:

— FLOPs/pas (forward + backward) $\approx 6 \times P \times B \times S = 6 \times 3.93\text{M} \times 1792 \times 128 \approx 5.4$ TFLOPs.
— FLOPs matériels v6e-8 : 8×459 TFLOPs/chip (bf16) $= 3672$ TFLOPs/s.
— Temps/pas : 7.96 secondes.
— MFU : $5.4/(3672 \times 7.96) \approx 0.18\%$... mais ceci est la borne théorique pour un seul pas. En pratique, le remat double les FLOPs : $2 \times 0.18\% \approx 0.37\%$.

L'écart avec les $\sim 5\%$ annoncés dans le README reflète que le MFU rapporté est basé sur les FLOPs effectifs (prenant en compte l'overhead de recompilation et les barrières de synchronisation), et non sur le calcul théorique brut. La vraie métrique utile est le *chip throughput* : combien de tokens par seconde par chip. À 0.126 pas/s avec $B \times S = 1792 \times 128 = 229\,376$ tokens/pas :

$$\text{tokens/s/chip} = \frac{229\,376 \times 0.126}{8} \approx 3\,617 \text{ tok/s/chip}$$

Sur v5p (4× la bande passante mémoire et ratio MXU/BW plus favorable), on attend ∼28 000 tok/s/chip — soit ∼8× d'amélioration.

6 Pipeline de Données à Coût Marginal Nul

6.1 Contrainte et Stratégie

L'entraînement de modèles de fondation est traditionnellement réservé aux entités capables de provisionner du stockage NVMe local ou du stockage cloud "Hot" en permanence. Un modèle de 150M paramètres nécessite typiquement 10–100 Go de données d'entraînement en accès rapide en permanence ; à 20$/To/mois sur GCS Standard, cela représente 20–200$/mois *sans entraîner une seule fois.*

Notre approche renverse ce modèle : GCS n'est qu'un cache éphémère, utilisé uniquement pendant l'entraînement. La persistance se fait sur Google Drive, inclus dans les abonnements Google One/Workspace à zéro coût marginal.

6.2 Architecture du Data Lake à 3 Niveaux

```
Google Drive (30 To froid, gratuit)
        |  rclone 64 canaux, chunks 128 Mo
        v
GCS gs://fin-ia-bucket (chaud, 0.02$/Go/mois)
        |  gsutil rsync / Grain async
        v
TPU VM HBM (prefetch double-buffer, 95 Go/chip)
```

6.2.1 Structure du Data Lake (Drive)

```
ChaosAI_DataLake/
|-- 01_raw_ohlcv/          # Bougies brutes Binance Futures 1h
|-- 02_tokens/             # Séquences tokenisées (codes FSQ)
|-- 03_training_ready/
|   `-- arrayrecords_v5/   # 433 shards ArrayRecord (Grain-ready)
|-- 04_checkpoints/
|   `-- jax_v6_v6e/
|       `-- *.tar.gz       # Archives Orbax compressées
`-- 05_results/            # Courbes, métriques, analyses
```

L'archivage `tar.gz` des checkpoints Orbax est essentiel : un checkpoint Orbax d'un modèle multi-chip génère des milliers de petits fichiers (un par shard par chip). L'upload direct de ces fichiers vers Google Drive provoquerait du rate-limiting de l'API Drive. Le `tar.gz` réduit des milliers de requêtes API à une seule.

6.2.2 trc_data_manager.sh

Le script orchestre le cycle complet :

Table 6 – Commandes du gestionnaire de Data Lake

Commande	Flux	Description
`stage`	Drive → GCS	rclone 64 canaux, 128 Mo chunks, `-fast-list` Vérification co-localisation région avant transfert
`backup`	GCS → Drive	Détecte le dernier checkpoint, `tar -czf`, upload
`cleanup`	GCS → /dev/null	Supprime tout de GCS, ramène la facture à 0$
`status`	—	Breakdown par préfixe + coût estimé + check région

Transfert rclone haute performance. rclone avec `-transfers=64 -s3-chunk-size=128M -fast-list` exploite la bande passante interne Google entre Drive et GCS pour des transferts à ∼500 Mo/s. Pour 160 Go d'ArrayRecords (v5p-32, modèle 150M), le staging prend ∼5 minutes.

Vérification co-localisation. Un bug fréquent dans les pipelines cloud ML est de placer les données dans une région différente des TPU. L'API Grain charge des batches en continu depuis GCS ; si le bucket est dans `us-central1` et le TPU dans `europe-west4`, chaque batch génère des frais d'egress inter-régions de $0.12/Go. Sur un entraînement de 3 jours à 1 Go/s de throughput données, cela représente ∼ $31 000 de frais cachés. Le script vérifie systématiquement la correspondance région avant tout staging.

6.3 Modèle de Coût

L'économie est non-linéaire en temps : un chercheur qui valide une architecture sur 3–4 semaines d'inactivité économise 75% du budget stockage mensuel. Sur un an d'exploration à cycle court (entraîner 3 jours, analyser 27 jours), l'économie peut atteindre 90%.

6.3.1 Audit GCP (Février 2026)

Suite à l'audit de l'infrastructure, deux VMs laissées en état `TERMINATED` ont été identifiées et supprimées : Ce type de dérive budgétaire est commun dans les projets de recherche : les VMs

TABLE 7 – Coût de stockage mensuel selon la stratégie

Dataset	GCS permanent		Drive + GCS 3j/mois	
	Coût	Notes	Coût	Économie
100 Go	\$2/mois		\$0.20	\$1.80
1 To	\$20/mois		\$2.00	\$18.00
10 To	\$200/mois	Courant à 1B+	\$20.00	**\$2 160/an**

Ressource	Raison	Économie
`financial-ia-ingest` (e2 + 50 Go SSD)	VM développement oubliée	\$8.50/mois
`financial-ia-training` (H100 + 200 Go SSD)	VM H100 non décommissionnée	\$34.00/mois
285 Mo GCS → Drive	Migration des checkpoints v6e	\$0.01/mois
Total		**\$42.51/mois**

TABLE 8 – Audit et nettoyage des ressources GCP orphelines

TERMINATED maintiennent les disques persistants facturés. Après nettoyage, le coût mensuel est de \$0 (données sur Drive, GCS vide).

7 Validation des Lois d'Échelle – Configurations et Jalons

7.1 Lois d'Échelle et Contrainte Chinchilla

La loi de Chinchilla [16] stipule que le ratio optimal paramètres/tokens est :

$$N^* = \frac{C}{6}, \quad T^* = \frac{C}{6 \cdot 2} \quad \Rightarrow \quad T^* = 20 \cdot N$$

où C est le budget compute en FLOPs. En pratique : pour un modèle optimal, le nombre de tokens d'entraînement doit être 20× le nombre de paramètres.

Notre dataset de 432 paires Binance Futures 1h couvre ∼8 ans de données (70 080 bougies/-paire) : $432 \times 70\,080 \approx 30$M bougies. Après tokenisation FSQ (séquences de 128 tokens avec stride 64) : ∼ 15M tokens uniques. Sur 10 epochs : 150M tokens effectifs.

Par Chinchilla : 150M tokens → modèle optimal de 150M/20 = 7.5M paramètres. Notre modèle de validation à 3.93M est donc légèrement sous-paramétrisé (∼ 15 tokens/param), mais acceptable pour la validation du pipeline.

7.2 T-Shirt Scaling — Configurations Pré-Calculées

Toutes les configurations respectent la contrainte d'alignement MXU :

$$\text{head\_dim} = \frac{d_{\text{model}} \times \text{expand\_factor}}{n_{\text{heads}}} = 128$$

Note architecturale pour XL (7B). Pour le modèle 7B, nous choisissons 32 couches au lieu de 48 (1B) avec $d = 4096$ au lieu de 2048. Les SSM bénéficient davantage de l'élargissement que de la profondeur : contrairement aux Transformers, les couches Mamba n'ont pas d'attention multi-tête nécessitant de la profondeur pour capturer des patterns hiérarchiques. Un d plus large améliore la capacité de mémorisation du sous-espace d'état par token.

TABLE 9 – Configurations T-Shirt (MXU-aligned, Chinchilla-optimal)

Taille	Params	d	Couches	Têtes	Pod	Batch	Tokens	LR
S	15M	256	12	4	v5p-8	8 192	300M	3×10^{-4}
M	150M	1 024	24	16	v5p-32	8 192	3B	2×10^{-4}
L	1B	2 048	48	32	v5p-128	16 384	20B	1×10^{-4}
XL	7B	4 096	32	64	v5p-768	49 152	140B	2×10^{-5}

Learning Rate Scaling. Le LR suit une règle approximativement $\propto 1/\sqrt{d_{\text{model}}}$ (μP-like [17]) : 3×10^{-4} pour 256, 2×10^{-5} pour 4096. Cette règle empirique stabilise l'entraînement en maintenant l'amplitude des gradients effective constante à travers les échelles.

Batch Scaling. Le batch global pour XL (49 152 = 256 × 192 chips data-parallel) est choisi pour :

— Remplir le MXU à chaque chip (256 séquences/chip est au-dessus du minimum d'efficacité)
— Utiliser la totalité des 192 replicas data-parallel
— Rester Chinchilla-compatible (LR effectif $\propto$ batch size)

7.3 Roadmap Expérimentale

TABLE 10 – Jalons de validation — configurations et objectifs mesurables

Phase	Params	Pod	Objectif mesurable	Statut	Tokens
0	3.9M	v6e-8	Pipeline JAX end-to-end	✓Accompli	22M
A	15M	v5p-8	Loss vs. baseline Transformer, MFU $\geq 30\%$	Prochain	300M
B	150M	v5p-32	Fit loi de puissance sur 2 points (3.9M, 15M)	Dépend A	3B
C	1B	v5p-128	Extrapolation Chinchilla validée sur 3 points	Dépend B	20B
D	7B	v5p-768	Dépend de la disponibilité TRC	Ouvert	140B

Périmètre de ce papier. Les phases A–D n'ont pas encore été exécutées. Ce papier documente exclusivement la Phase 0 (validation de bout-en-bout à 3.9M) et les fondations logicielles (architecture, noyaux, infrastructure) permettant les phases suivantes. Les configurations T-Shirt (Tableau ci-dessus) sont des cibles de design — chaque dimension est calculée pour respecter l'alignement MXU et le budget Chinchilla — pas des résultats expérimentaux.

Les phases A et B constituent la prochaine étape prioritaire : sans courbe de scaling sur au moins deux points expérimentaux, les affirmations de scalabilité de l'architecture restent des hypothèses.

Ce que valide la Phase 0 (et seulement ça) : (i) le pipeline JAX compile et s'exécute sans erreur sur 8 chips v6e, (ii) la perte VICReg et CFM décroissent de manière cohérente sur 2 700 pas, (iii) l'upcast bf16/f32 élimine la divergence NaN, (iv) l'infrastructure Drive $\leftrightarrow$ GCS $\leftrightarrow$ TPU fonctionne à coût nul au repos.

8 Limitations et Discussion

MFU sur v6e. Les $\sim 5\%$ de MFU observés sur v6e-8 ne reflètent pas les performances attendues sur v5p. Le v6e est un chip d'inférence : son ratio MXU/mémoire est optimisé pour les grands batches d'inférence, pas pour l'entraînement avec remat. Sur v5p-8, la même architecture avec le même batch taille devrait atteindre 30–40% MFU estimé.

Données financières comme proxy chaotique. Les marchés financiers comportent une dimension adversariale absente des systèmes physiques : les participants s'adaptent activement aux prédictions connues (front-running, market impact). Cette non-stationnarité induite rend la validation de l'alpha impossible sans déploiement réel, ce qui sort du périmètre de ce papier (recherche fondamentale, non trading).

Validité de la perturbation géodésique. Notre correction off-manifold suppose que VICReg pousse effectivement les représentations vers une hypersphère $\mathcal{S}^{d-1}$. En pratique, VICReg encourage une distribution avec variance unitaire par dimension, ce qui est une condition nécessaire mais pas suffisante pour la sphéricité exacte. Une analyse géométrique plus fine (via la décomposition spectrale de la matrice de covariance des embeddings) permettrait de quantifier la qualité de l'approximation sphérique.

Convergence CFM. La perte CFM (1.45 au pas 2700) n'a pas atteint son plateau : elle continue de décroître lentement. Sur le modèle de validation à 3.93M paramètres, l'encodeur est sous-dimensionné pour produire des cibles $h_{y,\text{tgt}}$ suffisamment riches pour que le champ de vitesse converge rapidement. À 150M paramètres, la perte CFM devrait converger plus vite grâce à la richesse représentationnelle accrue.

Lyapunov proxy et calibration. Le proxy Lyapunov $\lambda = \log(\sigma_{\text{inter}}/\sigma_{\text{pert}})/t$ n'est pas un exposant de Lyapunov au sens strict (il n'est pas calculé à partir de la dynamique vraie du système). C'est une mesure empirique de la divergence inter-univers normalisée par la magnitude de la perturbation initiale. Sa corrélation avec les exposants de Lyapunov vrais des marchés financiers (estimables via des méthodes de Takens embedding) est une direction de recherche future.

9 Conclusion

Ce papier documente une implémentation JAX native de bout-en-bout pour l'apprentissage auto-supervisé sur séries temporelles chaotiques. Les contributions sont de nature architecturale (encodeur Mamba-2 à horloge conditionnelle, OT-CFM, ASG pour la planification RL) et d'ingénierie système (noyau SSD aligné MXU, stabilité bf16, Auto-Sharder GSPMD, pipeline de données à coût marginal nul).

La validation sur 2 700 pas à 3.93M de paramètres (-35% de perte VICReg, -33% de perte CFM) confirme la correction des choix architecturaux et la cohérence de l'implémentation JAX. Elle ne constitue pas une démonstration de scalabilité : celle-ci requiert les expériences Phase A–C.

Les fondations logicielles sont en place pour la validation des lois d'échelle. La prochaine étape expérimentale concrète est l'entraînement du modèle 15M sur TPU v5p-8 avec mesure de la MFU et de la pente de perte, qui permettra de calibrer les projections Chinchilla sur les paliers supérieurs.

Références

[1] LeCun, Y. (2022). *A Path Towards Autonomous Machine Intelligence*. OpenReview.

[2] Gu, A., Goel, K., & Ré, C. (2021). *Efficiently Modeling Long Sequences with Structured State Spaces*. ICLR 2022.

[3] Gu, A., & Dao, T. (2023). *Mamba : Linear-Time Sequence Modeling with Selective State Spaces*. arXiv :2312.00752.

[4] Dao, T., & Gu, A. (2024). *Transformers are SSMs : Generalized Models and Efficient Algorithms Through Structured State Space Duality*. ICML 2024.

[5] Assran, M., Duval, Q., Misra, I., Bojanowski, P., Vincent, P., Rabbat, M., LeCun, Y., & Ballas, N. (2023). *Self-Supervised Learning from Images with a Joint-Embedding Predictive Architecture*. CVPR 2023.

[6] Bardes, A., Garrido, Q., Ponce, J., Chen, X., Rabbat, M., LeCun, Y., Assran, M., & Ballas, N. (2024). *V-JEPA : Latent Video Prediction for Visual Representation Learning*. arXiv :2404.08471.

[7] Lipman, Y., Chen, R. T. Q., Ben-Hamu, H., Nickel, M., & Le, M. (2022). *Flow Matching for Generative Modeling*. ICLR 2023.

[8] Tong, A., Malkin, N., Fatras, K., Atanackovic, L., Zhang, Y., Huguet, G., Wolf, G., & Bengio, Y. (2023). *Improving and Generalizing Flow-Based Generative Models with Minibatch Optimal Transport*. arXiv :2302.00482.

[9] Cuturi, M. (2013). *Sinkhorn Distances : Lightspeed Computation of Optimal Transport*. NeurIPS 2013.

[10] Hansen, N., Su, H., & Wang, X. (2024). *TD-MPC2 : Scalable, Robust World Models for Continuous Control*. ICLR 2024.

[11] Dabney, W., Rowland, M., Bellemare, M. G., & Munos, R. (2018). *Distributional Reinforcement Learning with Quantile Regression*. AAAI 2018.

[12] Rockafellar, R. T., & Uryasev, S. (2000). *Optimization of Conditional Value-at-Risk*. Journal of Risk, 2(3), 21–41.

[13] Mentzer, F., Minnen, D., Agustsson, E., & Tschannen, M. (2023). *Finite Scalar Quantization : VQ-VAE Made Simple*. ICLR 2024.

[14] Bardes, A., Ponce, J., & LeCun, Y. (2022). *VICReg : Variance-Invariance-Covariance Regularization for Self-Supervised Learning*. ICLR 2022.

[15] van den Oord, A., Vinyals, O., & Kavukcuoglu, K. (2017). *Neural Discrete Representation Learning*. NeurIPS 2017.

[16] Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., et al. (2022). *Training Compute-Optimal Large Language Models*. NeurIPS 2022.

[17] Yang, G., Hu, E. J., Babuschkin, I., Sidor, S., Liu, X., Farhi, D., et al. (2022). *Tensor Programs V : Tuning Large Neural Networks via Zero-Shot Hyperparameter Transfer*. arXiv :2203.03466.

[18] Das, A., Kong, W., Sen, R., & Zhou, Y. (2024). *A Decoder-Only Foundation Model for Time-Series Forecasting*. ICML 2024.

[19] Lorraine, J.,
endthebibliography Duvenaud, D. (2018). *Stochastic Hyperparameter Optimization through Hypernetworks*. NeurIPS Workshop 2018.

A Pseudocode — Chunked SSD

Algorithme 1 : Chunked SSD — Noyau TPU par Blocs

Entrées : $x \in \mathbb{R}^{B\times L\times H\times P}$, $\Delta_{\text{raw}} \in \mathbb{R}^{B\times L\times H}$, $A_{\log} \in \mathbb{R}^{H\times N}$, $B, C \in \mathbb{R}^{B\times L\times H\times N}$, $C_s = 128$

1. $A \leftarrow -\exp(A_{\log})$ *(décroissance strictement négative)*
2. $\Delta \leftarrow \text{softplus}(\Delta_{\text{raw}}) \cdot (1 - \text{weekend\_mask})$
3. Découpe : x_c, Δ_c, B_c, C_c de forme $(B, n_c, C_s, \ldots)$
4. cs $\leftarrow$ `cumsum_f32`(Δ_c) **float32 obligatoire**
5. $h_0 \leftarrow \mathbf{0} \in \mathbb{R}^{B\times H\times N\times P}$
6. **Pour** chunk k via `lax.scan` :

(a) $\text{cs\_diff}_{i,j} \leftarrow \text{cs}_i - \text{cs}_j \in \mathbb{R}^{C_s \times C_s}$ (tuile MXU)

(b) $L_{i,j} \leftarrow \text{tril}\Big(\sum_n C_i^{(n)} B_j^{(n)} \Delta_j e^{A_n \cdot \text{cs\_diff}_{i,j}}\Big)$

(c) $y_{\text{intra}} \leftarrow L \cdot x_k$ *(matmul MXU 128×128)*

(d) $y_{\text{inter}} \leftarrow (Ce^{A\,\text{cs}}) \cdot h_{k-1}$

(e) $y_k \leftarrow y_{\text{intra}} + y_{\text{inter}}$

(f) $h_k \leftarrow e^{A\,\text{cs\_end}} h_{k-1} + \sum_t \Delta_t B_t x_t^\top$ *(float32 pour exp)*

7. **Retour** : $\texttt{concat}(y_1, \ldots, y_{n_c})[:L]$

B Pseudocode — Multiverse Crossing

Algorithme 2 : Multiverse Crossing — Convergence et CVaR Dynamique

Entrées : $h_x \in \mathbb{R}^d$ (contexte JEPA), $\sigma = 0.01$, $M = 5$, $N = 16$

Étape 1 — Perturbation géodésique :

1. **Pour** $m = 1$ à M :
   (a) $\varepsilon_m \sim \mathcal{N}(0, I_d)$
   (b) $\hat{h} \leftarrow h_x / \|h_x\|$ *(direction radiale)*
   (c) $\varepsilon_\perp \leftarrow \varepsilon_m - (\varepsilon_m \cdot \hat{h})\hat{h}$ *(projection tangente)*
   (d) $h'_x \leftarrow h_x + \sigma\varepsilon_\perp$
   (e) $h_x^{(m)} \leftarrow h'_x \cdot \|h_x\| \,/\, \|h'_x\|$ *(retour sphère)*

Étape 2 — Génération de futurs :

1. **Pour** $m = 1$ à M : $\{h_{\text{fut}}^{(m,n)}\}_{n=1}^N \leftarrow \text{OT-CFM.sample}(h_x^{(m)})$

Étape 3 — Métriques de convergence :

1. $\mu_m \leftarrow \frac{1}{N}\sum_n h_{\text{fut}}^{(m,n)}, \quad \sigma_{\text{inter}} \leftarrow \text{std}(\mu_1, \ldots, \mu_M)$
2. $\sigma_{\text{intra}} \leftarrow \frac{1}{M}\sum_m \text{std}(h_{\text{fut}}^{(m,\cdot)})$
3. $\text{score} \leftarrow 1/(1 + \sigma_{\text{inter}}/\sigma_{\text{intra}}) \in [0, 1]$
4. $G \leftarrow \mu_c \mu_c^\top/(M-1), \quad \text{bifurcation} \leftarrow \exp(-\sum_i \tilde{\lambda}_i \log \tilde{\lambda}_i)$
5. $\lambda_{\text{Lyapunov}} \leftarrow \log(\sigma_{\text{inter}}/\sigma)/t$

Étape 4 — CVaR Dynamique :

$$\alpha \leftarrow \alpha_{\min} + (\alpha_{\max} - \alpha_{\min}) \cdot \text{score}$$

Retour : α, score, bifurcation, $\lambda_{\text{Lyapunov}}$

C Hyperparamètres Complets — Modèle de Validation (v6e-8)

TABLE 11 – Configuration complète du modèle validé sur TPU v6e-8

Composant	Paramètre	Valeur
Mamba-2	d_model	256
	d_state	16
	n_layers	6
	n_heads	2 ($\rightarrow$ head_dim = 128 = 1 MXU tile)
	expand_factor	2
	conv_kernel	4
	chunk_size	128
Prédicteur	hidden_dim	256
	n_layers	2
	z_dim	32
	cfm_weight	1.0
FSQ	num_codes	1024
	codebook_dim	64
	fsq_levels	[8, 8, 8, 2]
VICReg	inv_weight	25.0
	var_weight	25.0
	cov_weight	1.0
	var_gamma	1.0
Entraînement	lr	10^{-4}
	batch_size	1792 (224/chip $\times$ 8)
	precision	bf16 + f32 (accum.)
	grad_clip	1.0
	n_restarts SGDR	4
Strate IV	n_multiverses	5
	perturbation_sigma	0.01
	latent_dim	128
	n_quantiles	32
	cvar_alpha	[0.1, 0.4]
Total paramètres		**3 930 568**
Taille bf16		**7.9 Mo**