ESPECIFICACIÓN TÉCNICA COMPLETA

Sistema EBM con Criticidad Geométrica Autónoma

Documento de Implementación para Google Antigravity / Project IDX

Versión	2.0 - Implementación Composicional
Fecha	2025
Tipo	Energy-Based Model para Lenguaje
Hardware Target	RTX 3070 (8GB), Ryzen 5 3400G, SSD M.2
Framework	PyTorch 2.x + CUDA 11.8+
Lenguaje	Python 3.10+

ÍNDICE DE CONTENIDOS

1. Resumen Ejecutivo y Arquitectura General

1.1 Descripción del Sistema

Este documento especifica un Energy-Based Model (EBM) para modelado de lenguaje que opera sobre una hiperesfera de 640 dimensiones. El sistema utiliza "splats" (gaussianas direccionales) como atractores en el espacio latente, dinámica de Langevin para sampleo, y un mecanismo de Self-Organized Criticality (SOC) para consolidación automática de representaciones. La arquitectura incluye extensiones composicionales explícitas para generalización sistemática.

1.2 Diagrama de Arquitectura

Componente	Función	Input/Output
Tokenizer	Texto a Tokens y viceversa	I: texto, O: token_ids
Embedding Layer	Tokens a Vectores preliminares	I: token_ids, O: vectors[seq, d]
Splat Store	Almacena gaussianas direccionales	O: {mu_k, alpha_k, sigma_k}
Energy Computer	Calcula E(x) y gradientes	I: x, splats, O: E, gradE
Langevin Sampler	Genera samples de baja energía	I: x0, O: x_sample
SOC Controller	Consolida nuevos splats	I: history, O: new_splats
Context Manager	Gestiona 3 niveles de contexto	O: context_vectors
Compositional Engine	Operaciones de composición	I: x_a, x_b, O: x_comp
Decoder (MoE)	Vectores a Tokens	I: x, O: logits

Tabla 1. Componentes principales del sistema EBM.

2. Especificación del Espacio Latente

2.1 Definición Matemática

El espacio latente es la hiperesfera unitaria S639 = {x en R640 : ||x|| = 1}. Cada punto representa un estado semántico del sistema. La restricción de normalización garantiza estabilidad numérica y permite usar geometría Riemanniana especializada.

2.2 Propiedades Geométricas

Propiedad	Valor/Ecuación	Significado
Dimensión manifold	d = 640	Dimensión del espacio ambiente
Dimensión esfera	n = 639	Grados de libertad locales
Restricción	||x||^2 = 1	Normalización unitaria
Métrica	g_x = I - x*x^T	Métrica inducida
Espacio tangente	T_x S639 = {v : v·x = 0}	Espacio de velocidades
Distancia geodésica	d(x,y) = arccos(x·y)	Distancia angular

Tabla 2. Propiedades geométricas del espacio latente.

3. Función de Energía Completa

3.1 Definición General

La función de energía combina múltiples términos para capturar diferentes aspectos de la estructura semántica. La forma completa incluye contribuciones de splats, regularización geométrica, términos de interacción composicional, y entropía:

```
E(x) = E_splats(x) + λ_geom * E_geom(x) + λ_comp * E_comp(x) - H[q(x)]
```

3.2 Energía de Splats

La energía de splats mide la compatibilidad del estado actual con los K vecinos más cercanos:

```
E_splats(x) = -log Σ_{k=1}^K exp(α_k * (x·μ_k - 1) / τ)
```

Donde: α_k es el peso del splat k, μ_k es la media direccional, τ es la temperatura de suavizado, y K es el número de vecinos considerados (típicamente 32-128).

3.3 Energía Geométrica

La energía geométrica penaliza configuraciones no deseadas y promueve estructura:

```
E_geom(x) = -λ_spread * Σ_{i&lt;j} log(1 - x_i·x_j) + λ_cover * max(0, ρ_min - ρ(x))
```

El primer término promueve dispersión (evita colapso), el segundo asegura cobertura mínima del espacio.

3.4 Energía Composicional

Términos de interacción para capturar relaciones composicionales entre pares de splats:

```
E_comp(x) = Σ_{i,j} λ_ij * g(x·μ_i, x·μ_j)
```

Donde g(u, v) = σ(W·[u, v, u·v] + b) es una función de interacción aprendida.

3.5 Gradiente Riemanniano

El gradiente en la hiperesfera debe respetar la restricción de normalización:

```
∇_R E(x) = (I - x*x^T) * ∇E(x) = ∇E(x) - (x·∇E(x)) * x
```

Este gradiente es tangente a la esfera en el punto x y es la dirección de máximo crecimiento de energía respetando la geometría del manifold.

4. Sistema de Splats (Gaussianas Direccionales)

4.1 Definición de Splat

Cada splat es una gaussiana direccional definida sobre la hiperesfera, caracterizada por:

Parámetro	Símbolo	Dimensión	Descripción
Media direccional	μ_k	[640]	Centro del splat en S639, ||μ_k|| = 1
Peso/intensidad	α_k	escalar	Importancia del splat, α_k > 0
Concentración	κ_k	escalar	Ancho efectivo, κ = 1/σ²
ID semántico	sid_k	entero	Identificador único
Frecuencia	f_k	escalar	Veces activado (para SOC)
Edad	age_k	entero	Steps desde creación

Tabla 3. Parámetros de un splat individual.

4.2 Inicialización de Splats

Los splats se inicializan con información pre-procesada para acelerar convergencia:

OPCIÓN A - Transferencia de Embeddings Pre-entrenados: Cargar Word2Vec/FastText (300D), proyectar a 640D con matriz aprendida W, normalizar a unidad. OPCIÓN B - Inicialización Aleatoria Uniforme: Samplear de distribución uniforme en S639, asignar pesos iniciales α = 1.0, asignar concentración κ = 10.

4.3 Búsqueda de Vecinos (KNN)

Para eficiencia, usar Approximate Nearest Neighbor (ANN) con FAISS o similar:

```
# Algoritmo KNN para energía
def find_neighbors(x, splats, K):
    # x: [batch, 640], splats.mu: [N, 640]
    similarities = x @ splats.mu.T  # [batch, N]
    top_k_indices = torch.topk(similarities, K, dim=1).indices
    return top_k_indices, splats[top_k_indices]
```

5. Operaciones Riemannianas en Hiperesfera

5.1 Proyección a Espacio Tangente

Log-map: proyecta punto de la esfera al espacio tangente:

```
log_p(x) = θ / sin(θ) * (x - cos(θ) * p), donde θ = arccos(p·x)
```

5.2 Proyección desde Espacio Tangente

Exp-map: proyecta vector tangente de vuelta a la esfera:

```
exp_p(v) = cos(||v||) * p + sin(||v||) * v / ||v||
```

5.3 Transporte Paralelo

Mover vector tangente entre diferentes puntos base:

```
P_{x→y}(v) = v - [v·(y - (x·y)x)] / (1 + x·y) * (x + y)
```

5.4 Implementación Numéricamente Estable

```
# Proyección exp-map estable
def exp_map(p, v, eps=1e-8):
    norm_v = torch.norm(v, dim=-1, keepdim=True).clamp(min=eps)
    return torch.cos(norm_v) * p + torch.sin(norm_v) * v / norm_v

# Log-map estable
def log_map(p, x, eps=1e-8):
    cos_theta = (p * x).sum(dim=-1, keepdim=True).clamp(-1+eps, 1-eps)
    theta = torch.acos(cos_theta)
    sin_theta = torch.sin(theta).clamp(min=eps)
    return theta / sin_theta * (x - cos_theta * p)
```

6. Score Matching y Entrenamiento

6.1 Función de Pérdida Score Matching

El entrenamiento usa denoising score matching para aprender el gradiente de energía:

```
L = E_{x~data, ε~N(0,1)} [||s_θ(x˜) - ε/σ||²]
```

Donde $\tilde{x} = x + \sigma \cdot \varepsilon$ es la versión ruidosa del dato, $s_\theta(x) = -\nabla E(x)$ es el score (negativo del gradiente), y σ es el nivel de ruido.

6.2 Score Matching con Múltiples Escalas

Usar múltiples niveles de ruido para capturar estructura a diferentes escalas:

```
L = Σ_{i=1}^L λ(σ_i) * E[||s_θ(x˜_i) - ∇log q(x˜_i|x)||²]
```

6.3 Algoritmo de Entrenamiento por Batch

```
# Score matching training step
def train_step(model, batch, noise_levels):
    x = model.embed(batch)  # [B, 640], normalizado
    total_loss = 0
    for sigma in noise_levels:
        noise = torch.randn_like(x) * sigma
        x_noisy = x + noise
        x_noisy = F.normalize(x_noisy, dim=-1)  # Re-normalizar

        score_pred = model.compute_score(x_noisy)
        score_target = -noise / (sigma**2)  # Score verdadero

        loss = F.mse_loss(score_pred, score_target)
        total_loss += loss
    return total_loss / len(noise_levels)
```

7. Dinámica de Sampleo (Langevin Underdamped)

7.1 Ecuaciones de Movimiento

Underdamped Langevin dynamics incluye momentum para mejor exploración:

```
dx/dt = v
dv/dt = -γv - ∇_R E(x) + √(2γT) * ξ
```

Donde γ es el coeficiente de fricción, T es la temperatura, y ξ es ruido Gaussiano blanco.

7.2 Integrador Simpléctico (Störmer-Verlet)

```
# Langevin underdamped integrador
def langevin_step(x, v, energy_fn, gamma, T, dt, splats):
    # Half-step momentum
    grad_E = compute_riemannian_grad(x, energy_fn, splats)
    v = v - 0.5 * dt * (gamma * v + grad_E)

    # Full-step position (con proyección a esfera)
    x = exp_map(x, dt * v)  # Movimiento geodésico

    # Half-step momentum
    grad_E = compute_riemannian_grad(x, energy_fn, splats)
    v = v - 0.5 * dt * (gamma * v + grad_E)

    # Inyección de ruido
    noise = torch.randn_like(v) * np.sqrt(2 * gamma * T * dt)
    v = v + noise

    # Proyectar velocidad a tangente
    v = v - (v * x).sum(dim=-1, keepdim=True) * x

    return x, v
```

7.3 Parámetros de Sampleo

Parámetro	Valor Típico	Rango
Pasos de sampleo (T)	200	100-500
Step size (dt)	0.001	0.0001-0.01
Fricción (γ)	0.1	0.01-1.0
Temperatura (T)	1.0	0.1-2.0
Momentum inicial	0	N(0, 0.1)

Tabla 4. Parámetros de sampleo Langevin.

8. Mecanismo de Consolidación SOC

8.1 Principio de Self-Organized Criticality

El sistema debe detectar cuándo un nuevo splat es necesario para representar mejor las observaciones. El criterio se basa en el parámetro de orden que mide la "tensión" del sistema:

```
φ = (1/K) Σ_k α_k * ρ_k / ρ_avg
```

Donde ρ_k es la densidad local del splat k. Cuando φ > φ_threshold, el sistema está en estado crítico y debe consolidar un nuevo splat.

8.2 Algoritmo de Consolidación

```
# SOC consolidation algorithm
def maybe_consolidate(model, x_history, energy_history, threshold=0.8):
    # Calcular parámetro de orden
    phi = compute_order_parameter(model.splats, x_history)

    if phi > threshold:
        # Encontrar región de alta incertidumbre
        high_energy_idx = torch.argmax(energy_history)
        new_center = x_history[high_energy_idx]

        # Verificar que no duplica splat existente
        min_dist = compute_min_distance(new_center, model.splats.mu)
        if min_dist > model.min_splat_distance:
            # Crear nuevo splat
            new_splat = Splat(
                mu=new_center,
                alpha=model.init_alpha,
                kappa=model.init_kappa
            )
            model.splats.append(new_splat)
            return True
    return False
```

9. Sistema de Contexto Jerárquico

9.1 Niveles de Contexto

Nivel	Tokens	Propósito	Mecanismo
Local	8-16	Coherencia frase/local	Media de últimos N tokens
Medio	64-128	Coherencia párrafo	Ponderado exponencial
Global	512+	Coherencia documento	Atención simplificada

Tabla 5. Niveles de contexto jerárquico.

9.2 Combinación de Contextos

El contexto total se calcula como combinación ponderada de los tres niveles:

```
c_total = w_local * c_local + w_med * c_med + w_global * c_global
```

Los pesos pueden ser fijos o aprendidos. Típicamente: w_local=0.5, w_med=0.3, w_global=0.2.

10. Decoder con MoE Ligera

10.1 Arquitectura del Decoder

El decoder mapea representaciones latentes a distribuciones sobre tokens. Usa Mixture of Experts ligera para eficiencia:

```
logits = W_out * MoE([x; c_total]) + b_out
```

Donde x es el estado latente actual, c_total es el contexto combinado, y MoE es una capa de expertos con router aprendido.

10.2 Configuración MoE

Parámetro	Valor
Número de expertos	4-8
Expertos activos por token	2
Dimensión oculta experto	1024
Activación	GELU

Tabla 6. Configuración MoE del decoder.

11. Extensiones de Composicionalidad

11.1 Operador de Composición en Espacio Tangente

Para composición explícita de representaciones:

```
def compose(x_a, x_b, base_point, composition_net):
    # Proyectar ambos al espacio tangente
    v_a = log_map(base_point, x_a)
    v_b = log_map(base_point, x_b)

    # Componer en el tangente (red aprendida)
    v_ab = composition_net(torch.cat([v_a, v_b], dim=-1))

    # Proyectar de vuelta
    x_ab = exp_map(base_point, v_ab)
    return x_ab
```

11.2 Tipos de Composición Soportados

Tipo	Ejemplo	Mecanismo
Adjetivo + Sustantivo	rojo + coche	Modificación direccional
Verbo + Objeto	comer + manzana	Interacción semántica
Sustantivo + Sustantivo	casa + árbol	Composición espacial

Negación	no + bueno	Reflexión en esfera

Tabla 7. Tipos de composición soportados.

12. Estructuras de Datos

12.1 Clase Principal EBM

```
@dataclass
class EBMConfig:
    latent_dim: int = 640
    n_splats_init: int = 10000
    max_splats: int = 1000000
    knn_k: int = 64
    temperature: float = 0.1

    # Langevin
    langevin_steps: int = 200
    langevin_dt: float = 0.001
    langevin_gamma: float = 0.1
    langevin_T: float = 1.0

    # SOC
    soc_threshold: float = 0.8
    min_splat_distance: float = 0.1

    # Contexto
    context_local: int = 12
    context_medium: int = 64
    context_global: int = 512

    # Decoder
    vocab_size: int = 50000
    moe_experts: int = 4
    moe_active: int = 2
    hidden_dim: int = 1024
```

12.2 Estructura de Splats

```
class SplatStore(nn.Module):
    def __init__(self, config):
        self.mu = nn.Parameter(torch.randn(config.max_splats, config.latent_dim))
        self.alpha = nn.Parameter(torch.ones(config.max_splats))
        self.kappa = nn.Parameter(torch.ones(config.max_splats) * 10)
        self.frequency = torch.zeros(config.max_splats)
        self.age = torch.zeros(config.max_splats, dtype=torch.long)
        self.n_active = config.n_splats_init

    def normalize(self):
        with torch.no_grad():
            self.mu.data = F.normalize(self.mu.data, dim=-1)
            self.alpha.data = F.relu(self.alpha.data) + 0.01
            self.kappa.data = F.relu(self.kappa.data) + 1.0
```

12.3 Buffer de Historia para SOC

```
class HistoryBuffer:
    def __init__(self, capacity=10000, latent_dim=640):
        self.capacity = capacity
        self.states = torch.zeros(capacity, latent_dim)
        self.energies = torch.zeros(capacity)
        self.ptr = 0
        self.full = False

    def push(self, x, energy):
        self.states[self.ptr] = x.detach()
        self.energies[self.ptr] = energy.detach()
        self.ptr = (self.ptr + 1) % self.capacity
        self.full = self.full or self.ptr == 0
```

13. APIs e Interfaces

13.1 API de Entrenamiento

```
class EBMModel:
    def forward(self, token_ids: Tensor[batch, seq]) -> Tensor[batch, vocab]:
        """Forward pass completo: tokens a logits"""
        pass

    def compute_energy(self, x: Tensor[batch, 640]) -> Tensor[batch]:
        """Calcular energía para estados latentes"""
        pass

    def compute_score(self, x: Tensor[batch, 640]) -> Tensor[batch, 640]:
        """Calcular score (negativo del gradiente Riemanniano)"""
        pass

    def sample(self, n_samples: int, context: Tensor = None) -> Tensor[n, 640]:
        """Generar samples usando Langevin dynamics"""
        pass

    def decode(self, x: Tensor[batch, 640], context: Tensor) -> Tensor[batch, vocab]:
        """Decodificar estados latentes a logits"""
        pass
```

13.2 API de Inferencia

```
class EBMGenerator:
    def generate(
        self,
        prompt: str,
        max_tokens: int = 100,
        temperature: float = 1.0,
        top_k: int = 50,
        top_p: float = 0.9
    ) -> str:
        """Generar texto desde prompt"""
        pass

    def embed(self, text: str) -> Tensor[640]:
        """Obtener representación latente de texto"""
        pass

    def compose(self, text_a: str, text_b: str) -> Tensor[640]:
        """Componer representaciones de dos textos"""
        pass
```

14. Hiperparámetros y Configuración

14.1 Configuración por Fase

Parámetro	Prototipo	Pequeña	Media	Producción
Tokens entrenamiento	10M	100M	1B	10B
Splats iniciales	10K	50K	200K	1M
Batch size	32	64	128	256
Learning rate	1e-3	5e-4	1e-4	5e-5
Epochs	10	5	3	1
Warmup steps	100	500	1000	5000

Tabla 8. Configuración por fase de desarrollo.

14.2 Configuración de Hardware

Parámetro	Valor RTX 3070	Justificación
Mixed precision	FP16/BF16	Ahorro VRAM, aceleración
Gradient checkpointing	Activado	Reduce VRAM ~50%
KNN en CPU	Para splats >100K	Libera VRAM
Accumulación gradientes	4-8 steps	Batch efectivo mayor
Prefetch datos	2 workers	Oculta latencia I/O

Tabla 9. Optimizaciones para hardware limitado.

15. Pipeline de Entrenamiento

15.1 Loop Principal de Entrenamiento

```
# Training loop completo
def train_epoch(model, dataloader, optimizer, config):
    model.train()
    total_loss = 0

    for batch_idx, batch in enumerate(dataloader):
        # 1. Forward pass
        x = model.embed(batch.token_ids)  # [B, seq, 640]
        x = x.mean(dim=1)  # Pooling temporal [B, 640]
        x = F.normalize(x, dim=-1)

        # 2. Score matching loss
        loss = score_matching_loss(model, x, config.noise_levels)

        # 3. Regularización
        loss += config.reg_weight * spread_regularization(model.splats)

        # 4. Backward
        optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), config.grad_clip)
        optimizer.step()

        # 5. Normalización de splats
        model.splats.normalize()

        # 6. SOC check
        if batch_idx % config.soc_check_interval == 0:
            maybe_consolidate(model, x, loss)

        total_loss += loss.item()

    return total_loss / len(dataloader)
```

15.2 Función de Pérdida Score Matching

```
def score_matching_loss(model, x, noise_levels):
    total_loss = 0
    batch_size = x.shape[0]

    for sigma in noise_levels:
        # Añadir ruido
        noise = torch.randn_like(x) * sigma
        x_noisy = x + noise
        x_noisy = F.normalize(x_noisy, dim=-1)

        # Predecir score
        score_pred = model.compute_score(x_noisy)

        # Score objetivo (condicional)
        score_target = -noise / (sigma ** 2)

        # Proyectar a tangente
        score_pred = score_pred - (score_pred * x_noisy).sum(-1, keepdim=True) * x_noisy
        score_target = score_target - (score_target * x_noisy).sum(-1, keepdim=True) * x_noisy

        loss = F.mse_loss(score_pred, score_target)
        total_loss += loss

    return total_loss / len(noise_levels)
```

16. Pipeline de Inferencia

16.1 Generación de Texto

```
def generate(model, tokenizer, prompt, max_tokens=100, temperature=1.0):
    model.eval()

    # Tokenizar prompt
    tokens = tokenizer.encode(prompt)

    for _ in range(max_tokens):
        # Obtener representación actual
```

```
        x = model.embed(torch.tensor([tokens]))
        x = x.mean(dim=1)  # [1, 640]
        x = F.normalize(x, dim=-1)

        # Samplear nuevo estado
        x_new = model.sample(n_samples=1, context=x)[0]

        # Decodificar
        logits = model.decode(x_new, context=x)
        logits = logits / temperature

        # Samplear token
        probs = F.softmax(logits, dim=-1)
        next_token = torch.multinomial(probs, 1)

        tokens.append(next_token.item())

        if next_token == tokenizer.eos_id:
            break

    return tokenizer.decode(tokens)
```

17. Dependencias y Requisitos

17.1 Dependencias Python

```
# requirements.txt
torch>=2.0.0
numpy>=1.24.0
faiss-cpu>=1.7.4  # o faiss-gpu si disponible
transformers>=4.30.0  # tokenizer
tokenizers>=0.13.0
datasets>=2.12.0
wandb>=0.15.0  # logging
tqdm>=4.65.0
hydra-core>=1.3.0  # configuración
accelerate>=0.20.0  # mixed precision
```

17.2 Requisitos de Sistema

Recurso	Mínimo	Recomendado
Python	3.9	3.10+
CUDA	11.7	11.8+
VRAM	6 GB	8+ GB
RAM	16 GB	32+ GB
Disco	50 GB	200+ GB

Tabla 10. Requisitos de sistema.

18. Plan de Implementación por Fases

18.1 Fase 1: Prototipo (1-2 días)

Objetivo: Validar arquitectura básica con datos sintéticos. Implementar espacio latente hiperesférico, splats básicos, función de energía simple, test con datos aleatorios, verificar gradientes Riemannianos.

18.2 Fase 2: Core (3-5 días)

Objetivo: Sistema funcional con datos reales pequeños. Implementar score matching, Langevin sampler, decoder básico, cargar embeddings pre-entrenados, entrenar en 1M tokens, métricas básicas de calidad.

18.3 Fase 3: Escalado (1-2 semanas)

Objetivo: Sistema escalable con dataset medio. Implementar SOC, contexto jerárquico, optimizar KNN con FAISS, mixed precision training, entrenar en 100M tokens, evaluación completa.

18.4 Fase 4: Producción (2-4 semanas)

Objetivo: Sistema completo para uso real. Implementar composicionalidad, MoE decoder, optimizar inferencia, documentación completa, tests exhaustivos, benchmark vs baselines.

19. Métricas de Evaluación

19.1 Métricas de Entrenamiento

Métrica	Descripción	Target
Loss (score matching)	Error cuadrático del score	< 0.1
Energía media	Energía promedio de samples	Estable/decreciente
Cobertura de splats	Fracción de splats activos	> 80%
Tasa de consolidación	Nuevos splats por epoch	Decreciente

Tabla 11. Métricas de entrenamiento.

19.2 Métricas de Generación

- Perplexidad en held-out set - BLEU/ROUGE para tareas de generación - Diversidad léxica (unique n-grams ratio) - Repetición (repetition rate) - Coherencia (evaluada por humanos o modelo) - Generalización composicional (tests sintéticos)

20. Apéndice: Fórmulas Matemáticas Completas

20.1 Resumen de Ecuaciones Principales

Nombre	Ecuación
Energía total	$E(x) = -\log \Sigma \exp(\alpha_k(x \cdot \mu_{k-1})/\tau) + \lambda_g E_g + \lambda_c E_c$
Gradiente Riemanniano	$\square_R E = (I - x x^T)\square E = \square E - (x \cdot \square E)x$
Score matching loss	$L = E[\|\|s_\theta(\tilde{x}) - \varepsilon/\sigma\|\|^2]$
Exp-map	$\exp_p(v) = \cos(\|\|v\|\|)p + \sin(\|\|v\|\|)v/\|\|v\|\|$
Log-map	$\log_p(x) = \theta/\sin(\theta)\cdot(x - \cos(\theta)p)$
Langevin	$dx/dt = v,\ dv/dt = -\gamma v - \square_R E + \sqrt{(2\gamma T)}\xi$
Parámetro SOC	$\varphi = (1/K)\Sigma\ \alpha_k \cdot \rho_k/\rho_{avg}$
Distancia geodésica	$d(x,y) = \arccos(x \cdot y)$

Tabla 12. Resumen de fórmulas matemáticas principales.

Notas de Implementación

Este documento proporciona especificaciones completas para implementar el sistema EBM con criticidad geométrica. Se recomienda implementar incrementalmente, validando cada componente antes de avanzar al siguiente. El enfoque composicional con información pre-procesada reduce el tiempo de entrenamiento de aproximadamente 75 días a aproximadamente 12 días en hardware consumidor RTX 3070.

Para cualquier duda sobre implementación, referirse a los documentos complementarios de análisis de composicionalidad y costo-beneficio generados previamente.