Especificación de API Python - Universal Predictor

Ingeniería de Software

February 18, 2026

Contents

1 Introducción **2**

2 Estructuras de Datos (Tipado) **2**
- 2.1 Configuración (Λ) . . . 2
- 2.2 Entrada Operativa (y_t, y_{target}, τ) . . . 2
- 2.3 Salida del Sistema . . . 2

3 Arquitectura Multitenencia (Stateless/Functional Pattern) **3**
- 3.1 Maximización de Throughput (Batching Vectorizado) . . . 3

4 Clase Principal: `UniversalPredictor` (Stateful Wrapper) **4**
- 4.1 Inicialización . . . 4
- 4.2 Método de Ejecución (Paso $t \to t+1$) . . . 4

5 Persistencia (Atomic Snapshotting) **5**

6 Ajuste Adaptativo del Umbral CUSUM **6**
- 6.1 Fórmula de Ajuste . . . 6
- 6.2 Interpretación de Curtosis . . . 7

7 Flags de Operación y Recuperación **7**
- 7.1 DegradedInferenceMode . . . 7
- 7.2 EmergencyMode . . . 7
- 7.3 RegimeChangeDetected . . . 7
- 7.4 ModeCollapseWarning . . . 8

8 Manejo de Errores y Excepciones **8**
- 8.1 Excepciones Estándar . . . 8
- 8.2 Alertas Específicas Avanzadas . . . 8
- 8.3 Ejemplo de Logging en Producción . . . 8

9 Detección de Mode Collapse en DGM **9**
- 9.1 Criterio de Detección . . . 9
- 9.2 Acción Correctiva . . . 9

1 Introducción

Este documento detalla la implementación en Python de la interfaz abstracta I/O definida en *Predictor_Estocastico_IO*. La API expone la clase `UniversalPredictor`, diseñada para entornos de alto rendimiento utilizando JAX para la aceleración numérica.

2 Estructuras de Datos (Tipado)

Se utilizan `dataclasses` y `jaxtyping` para garantizar la inmutabilidad y el tipado dimensional estricto de los tensores.

2.1 Configuración (Λ)

```python
from dataclasses import dataclass
from typing import Optional
from jaxtyping import Float, Array, Bool

@dataclass(frozen=True)
class PredictorConfig:
    """Vector de Hiperparámetros Lambda."""
    schema_version: str = "1.0"   # Versionado de snapshots (evita incompatibilidades)
    epsilon: float = 1e-3         # Regularización Entrópica (Sinkhorn)
    learning_rate: float = 0.01   # Tasa de Aprendizaje JKO
    log_sig_depth: int = 3        # Profundidad de Firma (Kernel D)
    wtmm_buffer_size: int = 128   # Memoria WTMM (N_buf)
    besov_cone_c: float = 1.5     # Cono de Influencia de Besov
    holder_threshold: float = 0.4 # Umbral Circuit Breaker (H_min)
    cusum_h: float = 5.0          # Umbral Drift (h)
    cusum_k: float = 0.5          # Slack (k)
    volatility_alpha: float = 0.1 # Decaimiento EWMA de Varianza

    # Política de Abandono y Anti-Aliasing
    staleness_ttl_ns: int = 500_000_000         # TTL Latencia (500ms)
    besov_nyquist_interval_ns: int = 100_000_000 # Límite Nyquist (100ms) para estabilidad
    WTMM
    inference_recovery_hysteresis: float = 0.8  # Factor histéresis para recuperación de modo
    degradado
```

2.2 Entrada Operativa (y_t, y_{target}, τ)

```python
@dataclass(frozen=True)
class MarketObservation:
    price: Float[Array, "1"]      # y_t (Normalizado o Absoluto)
    target: Float[Array, "1"]     # y_target (Generalmente price actual)
    timestamp_ns: int             # Unix Epoch (Nanosegundos)

    def validate_domain(self, sigma_bound: float = 20.0, sigma_val: float = 1.0) -> bool:
        """Detección de Outliers Catastróficos (> 20 sigma)."""
        return abs(self.price) <= (sigma_bound * sigma_val)
```

2.3 Salida del Sistema

```python
@dataclass(frozen=True)
class PredictionResult:
    predicted_next: Float[Array, "1"]   # y_{t+1} (Espacio Z-Score)

    # Telemetría de Estado (S_risk)
    holder_exponent: Float[Array, "1"]  # H_t
    cusum_drift: Float[Array, "1"]      # G^+
    distance_to_collapse: Float[Array, "1"] # h - G^+
    free_energy: Float[Array, "1"]      # F (Energía JKO)

    # Telemetría Avanzada (Nuevas Adiciones)
    kurtosis: Float[Array, "1"]         #  _t - Curtosis empírica de errores
    dgm_entropy: Float[Array, "1"]      # H_DGM - Entropía del predictor DGM (NaN si inactivo)
```

```
    adaptive_threshold: Float[Array, "1"] # h_t - Umbral CUSUM adaptativo

    # Estado del Orquestador
    weights: Float[Array, "4"]          # [rho_A, rho_B, rho_C, rho_D] (Simplex)

    # Flags de Salud y Control (Explícitos)
    sinkhorn_converged: Bool[Array, "1"] # Convergencia JKO
    degraded_inference_mode: bool       # TTL violation (congelamiento de pesos)
    emergency_mode: bool                # H_t < H_min (singularidad crítica)
    regime_change_detected: bool        # CUSUM alarm (G+ > h_t)
    mode_collapse_warning: bool         # H_DGM <  ·H[g] (colapso DGM)

    mode: str                           # "Standard" | "Robust" | "Emergency"
```

3 Arquitectura Multitenencia (Stateless/Functional Pattern)

Para soportar cientos de activos (Multi-Asset) en un solo servidor, la API soporta un modo puramente funcional. Esto permite gestionar el estado en bases de datos externas de baja latencia (Redis) y compartir el grafo de computación JAX compilado (el `Predictor`) entre todos los activos.

3.1 Maximización de Throughput (Batching Vectorizado)

Esta arquitectura habilita el uso de `jax.vmap` para procesar lotes de estados de múltiples activos en una sola llamada al hardware, minimizando el impacto del GIL de Python y maximizando la ocupación de la GPU.

```python
class FunctionalPredictor:
    """
    Implementación Stateless para JAX Core.
    Permite escalar a miles de predictores compartiendo la misma estructura computacional.
    """

    def __init__(self, config: PredictorConfig):
        # Compilación JIT única para todos los activos
        # Habilita vectorización automática (vmap) sobre la dimensión del batch (activos)
        self.config = config
        self._core_step = self._core_update_step
        self._jit_update = jax.jit(self._core_step)
        self._vmap_update = jax.jit(jax.vmap(self._core_step, in_axes=(0, 0, 0, 0)))

    def init_state(self):
        """Genera un estado cero inicial (cold state structure)."""
        return self._initialize_state_structure()

    def step(self, state, obs: MarketObservation) -> tuple[object, PredictionResult]:
        """
        Transición de Estado Pura: (S_t, Obs_t) -> (S_{t+1}, Pred_{t+1})
        """
        # 1. Validaciones (Outlier, Staleness, Nyquist) logic idéntica a UniversalPredictor
        # ... logic for freeze_weights flag calculation ...

        # 2. Ejecución Kernel JAX
        # Zero-Copy: La actualización de búferes ocurre dentro de XLA (dynamic_update_slice)
        new_state, raw_result = self._jit_update(
            state,  # Estado inyectado explícitamente desde Redis/Memoria
            obs.price,
            obs.target,
            freeze_weights=should_freeze
        )

        # 3. Mapeo de Resultados
        result = PredictionResult(
            predicted_next=raw_result.y_next,
            # ... resto de campos ...
        )

        return new_state, result

    def step_batch(self, states, obs_batch: MarketObservation):
```

```
        """
        Procesamiento vectorizado para N activos simultáneos.
        Utiliza vmap para paralelizar la inferencia y actualización.
        """
        # ... logic for batch flags ...
        new_states, results = self._vmap_update(states, obs_batch.price, obs_batch.target,
    freeze_flags)
        return new_states, results
```

4 Clase Principal: `UniversalPredictor` (Stateful Wrapper)

Esta clase envuelve el patrón funcional para casos de uso de un solo activo (Single-Tenant), manteniendo el estado en memoria local (`self._state`).

4.1 Inicialización

```
class UniversalPredictor:
    def __init__(self, config: PredictorConfig):
        """
        Inicializa el grafo de cómputo JAX (XLA JIT compilation).
        Asigna memoria estática para los búferes en el dispositivo (VRAM).
        El estado interno (self._state) contiene los `jnp.array` persistentes (rolling buffers
    )
        que se actualizarán mediante operaciones funcionales (jnp.roll, lax.dynamic_update)
        para eliminar la latencia de transferencia de memoria (Zero-Copy).
        """
        self.config = config
        self._state = self._initialize_state() # Estado interno JAX (resident un GPU)
        self._jit_update = jax.jit(self._core_update_step)
        self._last_timestamp_ns = 0 # Para cálculo de frecuencia

    def fit_history(self, history: list[float]) -> bool:
        """
        Bootstrapping inicial (Protocolo de Cold Start).
        Procesa el lote histórico para estabilizar los pesos JKO y llenar los búferes.
        Requiere un mínimo de N_buf muestras.

        Returns:
            bool: True si el sistema alcanzó convergencia estable (Sinkhorn + CUSUM).
        Raises:
            ValueError: Si el historial es insuficiente (< wtmm_buffer_size).
            RuntimeError: Si el sistema diverge tras el calentamiento.
        """
        if len(history) < self.config.wtmm_buffer_size:
            raise ValueError(f"Historial insuficiente. Requerido: {self.config.
    wtmm_buffer_size}")

        # Ejecución batch acelerada (jax.lax.scan) para calentar el estado
        # Simula el paso del tiempo para llenar colas y estabilizar gradientes
        self._state, final_metrics = self._jit_scan_history(self._state, jnp.array(history))

        # Validación de Convergencia
        is_converged = final_metrics.sinkhorn_converged
        is_stable = final_metrics.cusum_drift < self.config.cusum_h

        if not (is_converged and is_stable):
             logger.warning("Cold Start finalizado sin convergencia estable.")
             return False

        return True
```

4.2 Método de Ejecución (Paso $t \to t+1$)

```
    def step(self, obs: MarketObservation) -> PredictionResult:
        """
        Ejecuta un ciclo completo de predicción.
        Maneja internamente la validación de dominio y TTL.
        """
```

```
        # 1. Validación de Dominio (Outlier Check)
        if not obs.validate_domain():
            logger.error("Outlier Catastrófico detectado. Ignorando tick.")
            return self._last_valid_result # Mantiene inercia

        # 2. Check de Abandono (Staleness) y Frecuencia (Anti-Aliasing)
        current_time = time.time_ns()
        latency = current_time - obs.timestamp_ns
        is_stale = latency > self.config.staleness_ttl_ns

        # Validación de Frecuencia Nyquist (WTMM Stability)
        dt_arrival = obs.timestamp_ns - self._last_timestamp_ns
        is_sparse = (self._last_timestamp_ns > 0) and (dt_arrival > self.config.
    besov_nyquist_interval_ns)

        if is_sparse:
             logger.warning(f"FrequencyWarning: Event interval {dt_arrival}ns > Nyquist limit.
     WTMM spectrum might alias.")

        self._last_timestamp_ns = obs.timestamp_ns

        # 3. Actualización Core (JAX) - Zero-Copy State Management
        # IMPORTANTE: El buffer de señal reside en GPU/TPU (self._state.signal_buffer).
        # La actualización se realiza "in-place" funcionalmente usando jax.lax.
    dynamic_update_slice
        # o jnp.roll dentro del kernel compilado para evitar transferencias CPU <-> VRAM.
        # Si hay staleness o sparsity excesiva, se congelan pesos para no degradar la geometrí
    a.
        should_freeze = is_stale or is_sparse

        new_state, result_data = self._jit_update(
            self._state,
            obs.price,
            obs.target,
            freeze_weights=should_freeze,
            # No se pasa history_buffer explícitamente, ya vive en _state
        )

        self._state = new_state

        # 4. Empaquetado de Resultados
        return PredictionResult(
            predicted_next=result_data.y_next,
            holder_exponent=result_data.H_t,
            sinkhorn_converged=result_data.converged,
            is_stable=not (is_stale or is_sparse),
            # ... mapeo resto de campos
        )
```

5 Persistencia (Atomic Snapshotting)

El sistema implementa persistencia binaria protegida por checksum.

```
import hashlib
import msgpack

    def save_snapshot(self, filepath: str):
        """
        Exporta el estado interno Sigma_t a formato binario (MessagePack).
        Incluye Checksum SHA-256 al final del archivo.
        """
        # Serialización de tensores JAX a bytes
        state_dict = self._serialize_jax_state(self._state)

        # Segmentación Modular (K-Blocks)
        # IMPORTANTE: Incluir versionado de schema para evitar errores al cargar
        # snapshots generados con versiones antiguas (cambios en profundidad de firma, etc.)
        payload = {
            "schema_version": self.config.schema_version,  # Versionado seguro
            "timestamp": time.time_ns(),
            "config": asdict(self.config),
```

```
            "global": state_dict["global"], # rho, G+, ema
            "telemetry": {
                "kurtosis": float(self._state.kurtosis),
                "dgm_entropy": float(self._state.dgm_entropy),
                "adaptive_threshold": float(self._state.h_adaptive)
            },
            "flags": {
                "degraded_inference": bool(self._state.degraded_mode),
                "emergency": bool(self._state.emergency_mode),
                "regime_change": bool(self._state.regime_changed),
                "mode_collapse": bool(self._state.mode_collapse_warning)
            },
            "kernels": {
                "A": state_dict["kernel_a"],
                "B": state_dict["kernel_b"],
                "C": state_dict["kernel_c"],
                "D": state_dict["kernel_d"]
            }
        }

        data_bytes = msgpack.packb(payload)
        checksum = hashlib.sha256(data_bytes).hexdigest()

        with open(filepath, "wb") as f:
            f.write(data_bytes)
            f.write(checksum.encode('utf-8')) # Append hash

    def load_snapshot(self, filepath: str):
        """
        Carga estado. Valida SHA-256 y schema_version antes de deserializar.
        Lanza ValueError si falla la validación o schema incompatible.
        """
        with open(filepath, "rb") as f:
            content = f.read()

        data_bytes = content[:-64] # Todo menos los últimos 64 bytes (SHA256 hex)
        stored_checksum = content[-64:].decode('utf-8')

        computed = hashlib.sha256(data_bytes).hexdigest()
        if computed != stored_checksum:
            raise ValueError("Snapshot corrupto: Checksum mismatch.")

        payload = msgpack.unpackb(data_bytes)

        # Validar schema_version para detectar incompatibilidades
        loaded_schema = payload.get('schema_version', 'unknown')
        if loaded_schema != self.config.schema_version:
            raise ValueError(
                f"Schema version mismatch: snapshot={loaded_schema}, "
                f"current={self.config.schema_version}. "
                f"Cannot load snapshot generated with incompatible kernel depths or signature
features."
            )

        self._state = self._deserialize_jax_state(payload)
```

6 Ajuste Adaptativo del Umbral CUSUM

El sistema implementa el **Lema de Umbral Adaptativo** basado en curtosis, permitiendo que el detector CUSUM se ajuste automáticamente a regímenes con colas pesadas.

6.1 Fórmula de Ajuste

El umbral de detección de cambio de régimen se calcula dinámicamente:

$$h_t = k \cdot \sigma_t \cdot \left(1 + \ln\left(\frac{\kappa_t}{3}\right)\right)$$

donde:

- k: Slack calibrado (`cusum_k` en configuración)
- σ_t: Volatilidad EMA del error de predicción
- κ_t: Curtosis empírica móvil (ventana de 252 pasos)
- 3: Curtosis de referencia Gaussiana

6.2 Interpretación de Curtosis

Rango κ_t	**Régimen de Mercado**
$\kappa_t \approx 3$	Gaussiano (mercado normal)
$\kappa_t \in [5, 10]$	Volatilidad financiera estándar
$\kappa_t \in [10, 15]$	Alta volatilidad (eventos outlier)
$\kappa_t > 15$	Régimen de crisis (colas pesadas)
$\kappa_t > 20$	Falla en modelo de residuos (alerta crítica)

Table 1: Interpretación de curtosis empírica

Nota: El ajuste logarítmico permite que el umbral se expanda automáticamente cuando $\kappa_t > 3$, evitando falsos positivos en regímenes de alta curtosis mientras mantiene sensibilidad a cambios estructurales genuinos.

7 Flags de Operación y Recuperación

El sistema mantiene cuatro flags booleanos explícitos que señalizan estados críticos al ejecutor:

7.1 DegradedInferenceMode

Condición de activación:

$$\text{TTL}(y_{\text{target}}) = t_{\text{current}} - t_{\text{signal}} > \Delta_{\max}$$

Implicaciones operacionales:

1. Suspende actualización del transporte JKO inmediatamente
2. Congela pesos ρ en último valor válido (modo inercial)
3. Predicciones continúan generándose pero con confianza degradada
4. Riesgo NO está siendo optimizado geométricamente

Recuperación con histéresis:

$$\text{TTL}(y_{\text{target}}) < h_{\text{hyst}} \cdot \Delta_{\max}$$

donde $h_{\text{hyst}} =$ `inference_recovery_hysteresis` (por defecto 0.8) parametrizable en PredictorConfig.

Se emite `NormalOperationRestoredEvent` al recuperar.

7.2 EmergencyMode

Condición: $H_t < H_{\min}$ (singularidad crítica detectada)
Acción: Fuerza $w_D \to 1.0$ (Kernel D de signatures) y cambia a métrica de Huber robusta.

7.3 RegimeChangeDetected

Condición: $G^+ > h_t$ (CUSUM detecta cambio de régimen)
Acción: Reinicio de entropía a distribución uniforme y reset de acumuladores.

7.4 ModeCollapseWarning

Condición: $H_{\text{DGM}} < \gamma \cdot H[g]$ durante > 10 pasos consecutivos (solo relevante si $\rho_B > 0.05$)
Acción correctiva: Reducir $\rho_B \to 0$ hasta re-entrenar red DGM.

8 Manejo de Errores y Excepciones

8.1 Excepciones Estándar

- `DomainError`: Se lanza (o se loguea crítico) si y_t excede los límites (Outlier Catastrófico $> 20\sigma$).
- `StalenessWarning`: Emitido mediante el sistema de logging estándar de Python cuando se activa la protección TTL.
- `FrequencyWarning`: Alerta si la tasa de arribo de eventos cae por debajo del límite de Nyquist para el análisis de Besov.
- `IntegrityError`: Fallo crítico en la carga de snapshot. El sistema debe abortar y solicitar reinicio en frío.

8.2 Alertas Específicas Avanzadas

- `ModeDegradationAlert`: Se emite cuando H_{DGM} viola umbral durante > 10 pasos consecutivos. Indica colapso de modo en el predictor neuronal DGM (Rama B).
- `KurtosisOutlierWarning`: Se emite si $\kappa_t > 20$ de forma persistente (> 5 pasos consecutivos). Señala falla potencial en el modelo de residuos y sugiere revisión de arquitectura.
- `NormalOperationRestoredEvent`: Se emite al recuperar de `DegradedInferenceMode` (cuando TTL vuelve bajo el umbral con histéresis). Señaliza al ejecutor que puede retomar operación normal.

8.3 Ejemplo de Logging en Producción

```python
import logging
import os
from datetime import datetime

def save_emergency_dump(predictor, result, asset_id: str):
    """
    Guarda un "Dump de Depuración" completo cuando se activa EmergencyMode.
    Incluye: estado de pesos, buffer de señales, historial de telemetría.
    """
    dump_dir = os.path.expanduser("~/.predictor_emergency_dumps")
    os.makedirs(dump_dir, exist_ok=True)

    timestamp = datetime.now().isoformat()
    dump_file = f"{dump_dir}/{asset_id}_emergency_{timestamp}.msgpack"

    debug_payload = {
        "emergency_timestamp": timestamp,
        "asset_id": asset_id,
        "holder_exponent": float(result.holder_exponent),
        "weights": [float(w) for w in result.weights],
        "signal_buffer": predictor._state.signal_circular_buffer.tolist(),
        "regime_history": predictor._state.cusum_history.tolist(),
        "telemetry_snapshot": {
            "kurtosis": float(result.kurtosis),
            "dgm_entropy": float(result.dgm_entropy),
            "adaptive_threshold": float(result.adaptive_threshold),
            "distance_to_collapse": float(result.distance_to_collapse)
        },
        "flags_at_emergency": {
            "degraded_inference": bool(result.degraded_inference_mode),
            "regime_change": bool(result.regime_change_detected),
            "mode_collapse": bool(result.mode_collapse_warning)
        }
```

```
    }

    with open(dump_file, "wb") as f:
        msgpack.packb(debug_payload, file=f)

    logging.critical(f"Emergency dump saved to {dump_file} for forensius analysis")

def process_prediction(predictor, obs):
    result = predictor.step(obs)
    asset_id = obs.asset_id if hasattr(obs, 'asset_id') else "unknown"

    # Flags críticos
    if result.degraded_inference_mode:
        logging.warning(
            "DEGRADED MODE: TTL exceeded. Weights frozen. "
            "Consider reducing position size."
        )

    if result.emergency_mode:
        logging.critical(
            f"EMERGENCY: Singularity detected (H={result.holder_exponent:.3f}). "
            "Forcing Kernel D with Huber loss."
        )
        # Guardar dump automáticamente para análisis post-mortem
        save_emergency_dump(predictor, result, asset_id)

    if result.mode_collapse_warning:
        logging.error(
            f"MODE COLLAPSE: DGM entropy below threshold. "
            f"H_DGM = {result.dgm_entropy:.3f}. "
            "Reducing rho_B -> 0."
        )

    if result.kurtosis > 20.0:
        logging.warning(
            f"KURTOSIS OUTLIER: kappa = {result.kurtosis:.2f} > 20. "
            "Residual model may be invalid."
        )

    return result
```

9 Detección de Mode Collapse en DGM

El sistema monitoriza la entropía diferencial del predictor neuronal (Rama B) para detectar colapso a soluciones triviales.

9.1 Criterio de Detección

La entropía diferencial de la solución DGM $V_\theta(x,t)$ se calcula como:

$$H_{\text{DGM}} = -\int p_V(v) \log p_V(v)\, dv$$

se compara contra la entropía de la condición terminal $H[g]$:

$$H_{\text{DGM}} \geq \gamma \cdot H[g], \quad \gamma \in [0.5, 1.0]$$

Si la violación persiste durante > 10 pasos consecutivos, se activa `mode_collapse_warning`.

9.2 Acción Correctiva

El orquestador JKO debe reducir el peso de la Rama B:

$$\rho_B \to 0$$

hasta que se re-entrene la red neuronal DGM con hiperparámetros ajustados (tasa de aprendizaje, arquitectura, inicialización).

Nota Teórica: Una solución colapsada tiene $H[V_\theta] \to -\infty$ (distribución delta), correspondiendo a una política de control degenerada que no responde a variaciones del estado.