Especificación de API Python - Universal Predictor

Ingeniería de Software

February 18, 2026

Contents

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."""
    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
```

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)

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

    # Flags de Salud y Control
    sinkhorn_converged: Bool[Array, "1"] # Convergencia JKO
```

```
    is_stable: bool                     # is_degraded (False si viola TTL)
    mode: str                           # "Standard" | "Robust"
```

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

```python
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 $t \to t+1$)

```python
    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 geometría.
        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)
        payload = {
            "timestamp": time.time_ns(),
            "config": asdict(self.config),
            "global": state_dict["global"], # rho, G+, ema
            "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()
```

```
27
28          with open(filepath, "wb") as f:
29              f.write(data_bytes)
30              f.write(checksum.encode('utf-8')) # Append hash
31
32      def load_snapshot(self, filepath: str):
33          """
34          Carga estado. Valida SHA-256 antes de deserializar.
35          Lanza IntegrityError si falla la validación.
36          """
37          with open(filepath, "rb") as f:
38              content = f.read()
39
40          data_bytes = content[:-64] # Todo menos los últimos 64 bytes (SHA256 hex)
41          stored_checksum = content[-64:].decode('utf-8')
42
43          computed = hashlib.sha256(data_bytes).hexdigest()
44          if computed != stored_checksum:
45              raise ValueError("Snapshot corrupto: Checksum mismatch.")
46
47          payload = msgpack.unpackb(data_bytes)
48          self._state = self._deserialize_jax_state(payload)
```

6 Manejo de Errores y Excepciones

- `DomainError`: Se lanza (o se loguea crítico) si y_t excede los límites (Outlier Catastrófico).
- `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.