Revisión de código
`fiboevo.py.old`

Santiago Javier Espino Heredero

September 24, 2025

Abstract

Este capítulo documenta la estructura, las clases y funciones principales contenidas en `fiboevo.py`, compara explícitamente la versión actual con `fiboevo.py.old`, identifica regresiones y anomalías de coherencia/sintaxis y propone correcciones y mejoras concretas. El foco principal recae sobre:

- La clase de modelo central `LSTM2Head` y las utilidades adjuntas.
- Las funciones de entrenamiento/evaluación (`train_epoch`, `eval_epoch`).
- Los transformadores y generadores de secuencias (`create_sequences_from_df`) y la generación de características (`add_technical_features`).
- El orquestador evolutivo `run_evolution` y el evaluador de individuos `evaluate_individual`.
- El backtester de tipo market-maker (`backtest_market_maker`).
- Las rutinas de guardado/carga de modelo (`save_model`, `load_model`).

Contents

1 Metodología

El análisis se basa en:

1. Parsing AST para enumerar clases y funciones top-level.
2. Lectura directa de las implementaciones para entender firmas, argumentos y docstrings.
3. Diferencias textuales entre `fiboevo.py.old` y `fiboevo.py` para identificar regresiones.
4. Revisión estática para detectar problemas de indentación, puntos de posible *data-leakage*, y uso inconsistente de transformación/normalización.

2 Inventario de clases y funciones (alto nivel)

2.1 Clases principales

- **`LSTM2Head`**
  Herencia: `torch.nn.Module`.
  Constructor (argumentos): `input_size: int, hidden_size: int = 64, num_layers: int = 2, dropout: float = 0.1`.
  Descripción: LSTM recurrente seguida de dos cabezas (una para retorno predicho y otra para volatilidad predicha). La cabeza de volatilidad aplica `Softplus` en la salida para asegurar positividad.
  Método: `forward(x)` → devuelve (`pred_ret`, `pred_vol`) como tensores 1D de tamaño batch.

Listing 1: Esqueleto de `LSTM2Head`

```
class LSTM2Head(nn.Module):
   def __init__(self, input_size:int, hidden_size:int=64, num_layers:
       int=2, dropout:float=0.1):
       # LSTM + head_ret + head_vol (Softplus)
   def forward(self, x: torch.Tensor) -> Tuple[torch.Tensor, torch.
       Tensor]:
       # out, _ = self.lstm(x)
       # last = out[:, -1, :]
       # ret = self.head_ret(last).squeeze(-1)
       # vol = self.head_vol(last).squeeze(-1)
       # return ret, vol
```

- **`EvoBounds`**
  Clase contenedora de límites para parámetros usados por el algoritmo evolutivo (ej. `seq_len`, `hidden`, tasas de aprendizaje en log, etc.). No contiene lógica compleja.

2.2 Funciones top-level (más relevantes)

Listado abreviado (funciones analizadas en detalle abajo): `setup_logging, seed_everything, simple_rsi, atr_simple, load_data_from_influx, simulate_ou, add_technical_featur build_ohlc_df_from_close, create_multi_tf_merged_df_from_base, create_sequences_ train_epoch, eval_epoch, backtest_market_maker, clamp, repair_individual, evaluate_individual, run_evolution, save_model, load_model, ...`

Para cada una de las funciones clave examino firma, comportamiento esperado y observaciones.

3 Análisis funcional detallado

3.1 `create_sequences_from_df`

Firma:

```
create_sequences_from_df(df: pd.DataFrame, feature_cols: Sequence[str],
    seq_len: int = 32, horizon: int = 1) -> Tuple[np.ndarray, np.ndarray, np
    .ndarray]
```

Descripción: Construye tensores `X` de forma (N, seq_len, F), vectores objetivo `y_ret` (retorno futuro al horizonte) y `y_vol` (volatilidad realizada en el horizonte). Implementa protección por `eps` para evitar división por cero:

$$y\_ret[i] = \frac{\text{close}[end] - \text{close}[start]}{\text{close}[start] + 1e-12}$$

Observaciones:

- Correcto el uso de `seq_len` y `horizon` y la verificación de que haya suficiente data.
- **Recomendación:** Documentar y estandarizar la estrategia de retorno (simple vs log). Actualmente usa retorno simple; para estabilidad numérica en datos con picos es recomendable evaluar `log-return` o clipping robusto.
- **Posible mejora:** Añadir argumento para elegir tipo retorno: `method: Literal['simple','log']` y normalizar / winsorizar valores extremos antes de calcular volatilidad objetivo.

3.2 `add_technical_features`

Firma:

```
add_technical_features(close: np.ndarray, high: Optional[np.ndarray]=None,
    low: Optional[np.ndarray]=None, volume: Optional[np.ndarray]=None,
    window_long: int = 50) -> pd.DataFrame
```

Descripción: Genera SMA, EMA para ventanas múltiples, diffs, Bollinger, RSI, ATR (cuando aplica) y otras columnas; incluye un bloque para aproximación de *volume profile POC* si `volume` está presente. La función claramente documenta que *caller* debe aplicar `dropna()` para evitar look-ahead tras cálculos con ventanas. **Observaciones:**

- Buena práctica: la función evita rellenados implícitos que pueden causar look-ahead. El requisito de `dropna()` debe aplicarse consistentemente antes de construir secuencias.
- **Recomendación:** Proveer una función auxiliar `safe_dropna(df, min_valid_cols)` para controlar cuánto de la ventana es tolerable y permitir reproducibilidad.

3.3 `train_epoch` y `eval_epoch`

Firma (train_epoch):

```
train_epoch(model: nn.Module, loader: DataLoader, optimizer: optim.
    Optimizer, device: torch.device, alpha_vol_loss: float=0.5, grad_clip:
    float=1.0) -> float
```

Comportamiento general: Entrena por una época sumando pérdidas MSE entre predicción de retorno y volatilidad y la verdad; aplica grad clipping opcional. **Observación:** `train_epoch` en la versión actual está implementada correctamente (verificado) — no se detectaron errores de indentación ni de tipado.

Firma (eval_epoch):

```
eval_epoch(model: nn.Module, loader: DataLoader, device: torch.device,
    alpha_vol_loss: float=0.5) -> float
```

Problema crítico detectado (regresión): en `fiboevo.py` (versión actual) la implementación de `eval_epoch` presenta un error de indentación que provoca que la evaluación de pérdida *fuera* del bucle `for`, haciendo que:

- sólo la **última** iteración del `for` (o incluso un batch no válido si el loader está vacío) sea la usada para la computación de pérdida, en lugar de acumular la pérdida sobre todo el `DataLoader`.
- si el `DataLoader` estuviera vacío el código fallaría al acceder variables no definidas.

En la versión de respaldo (`fiboevo.py.old`) la función está correctamente indentada y calcula la pérdida en cada batch dentro del bucle.

Parche sugerido (reemplazo de `eval_epoch`)

A continuación se propone el reemplazo exacto para restaurar el comportamiento correcto. Esta es la implementación tomada de la versión `.old` (funcional y testeada):

Listing 2: Parche: implementación correcta de eval_epoch

```
def eval_epoch(
    model: nn.Module, loader: DataLoader, device: torch.device,
        alpha_vol_loss: float = 0.5
) -> float:
    """
␣␣␣␣Evaluacin␣por␣poca:␣recorre␣el␣loader␣(batches)␣y␣acumula␣la␣prdida␣MSE
␣␣␣␣entre␣retorno␣predicho␣y␣verdadero␣y␣entre␣volatilidad␣predicha␣y␣
    verdadera.
␣␣␣␣"""
    model.eval()
    total_loss = 0.0
    n = 0
    mse = nn.MSELoss()
    with torch.no_grad():
        for xb, yret, yvol in loader:
            xb = xb.to(device)
            yret = yret.to(device)
            yvol = yvol.to(device)
            # ensure target tensors are 1D to match model output
            if yret.dim() > 1:
                yret = yret.view(-1)
            if yvol.dim() > 1:
                yvol = yvol.view(-1)
            pred_ret, pred_vol = model(xb)
            loss_ret = mse(pred_ret, yret)
            loss_vol = mse(pred_vol, yvol)
            loss = loss_ret + alpha_vol_loss * loss_vol
            bs = xb.size(0)
            total_loss += loss.item() * bs
            n += bs
    return total_loss / max(1, n)
```

Impacto de la regresión: sin esta corrección, las métricas de validación quedan mal calculadas —lo que puede llevar a decisiones incorrectas en selección de modelos, early stopping y en la función de fitness usada por el evolutor. Recomiendo aplicar este parche inmediatamente y agregar un test unitario que verifique que `eval_epoch` promedia pérdidas sobre múltiples batches.

3.4 `run_evolution` y `evaluate_individual`

run_evolution: Orquesta DEAP para evolución de hiperparámetros. Usa `evaluate_individual` para evaluar fitness de cada individuo. Implementación estándar: generación de población, evaluación, cross-over, mutación y selección.

evaluate_individual:

- Repara el individuo (`repair_individual`) y descifra sus parámetros (e.g., `hidden`, `log_lr`, `seq_len`, `tp_sigma`, `sl_sigma`, etc.).
- Construye secuencias con `create_sequences_from_df`, separa en train/val con split del 70%, entrena un modelo temporalmente (epochs especificadas por individuo) y ejecuta `backtest_market_maker` sobre el conjunto de validación para computar una métrica (p.ej. PnL).

Observaciones:

- **Costo computacional:** evaluar muchos individuos implica entrenamientos repetidos —se recomienda limitar generaciones/población o usar evaluaciones proxy (retrain ligero, warm-start, o evaluación sobre subsample).
- **Reproducibilidad:** el uso de semillas está presente —mantener consistencia en `np.random`, `random` y `torch` como ya hace el código.

3.5 `backtest_market_maker`

Función de backtesting simulando ordenes market-making / scalping con múltiples controles (intraday capital, gating por volatilidad, uso de indicadores SMA/EMA, POC/volume profile simplificado, reglas TP/SL en función de volatilidad predicha).

Observaciones importantes:

- En `fiboevo.py.old` existían funciones utilitarias importantes para simular fills con ticks (`quantize_price`, `quantize_amount`, `simulate_fill`). Estas funciones **no están presentes** en la versión actual del archivo. Si el backtester o código externo las requiere, su ausencia puede provocar errores o cambios silenciosos en la simulación (por ejemplo, cambios en el rounding que afectan resultados).
- **Recomendación:** si la eliminación fue intencional, mover esas utilidades a un módulo común (p.ej. `utils/backtesting.py`) y asegurar que el backtester importe la versión canónica. Si no fue intencional, restaurar las funciones desde la versión `.old`.

3.6 `save_model` y `load_model`

save_model: guarda `model.state_dict()` en disco y opcionalmente guarda un `meta` JSON en ruta separada.

load_model: trata de cargar un `state_dict` y reconstruir una `LSTM2Head` intentando inferir dimensiones (input/hidden) examinando shapes en el `state_dict`.

Observaciones y recomendaciones:

- **Robustez:** el intento de inferir dimensiones es útil, pero frágil. Recomiendo guardar un único archivo `.pt` que contenga tanto el `state_dict` como la `meta` (ej. `torch.save('state': model.state_dict(), 'meta': meta, path)`). Al cargar, se puede reconstruir con seguridad a partir de `meta`.

- **Compatibilidad DataParallel:** cuando el modelo fue guardado con `torch.nn.DataParallel`, las keys en el state dict pueden llevar el prefijo `'module.'`. El `load_model` debe contemplar esto removiendo `'module.'` si existe.
- **Key-mismatch handling:** Actualmente el código intenta `load_state_dict(...)` y en excepción vuelve a intentar con `strict=False`. Recomendable: informar al usuario qué keys faltan/extra con `missing, unexpected = model.load_state_dict(state, strict=False)` y loggear la diferencia.

4 Diferencias encontradas entre `fiboevo.py.old` y `fiboevo.py`

- **Regresión crítica:** `eval_epoch` en la versión actual tiene indentación errónea que provoca cómputo de pérdida fuera del bucle (ver parche en sección previa). Esta es la corrección más urgente.
- **Funciones de simulación eliminadas:** `quantize_price`, `quantize_amount` y `simulate_fill` están presentes en `.old` y ausentes en la versión actual. Si no han sido movidas a otro módulo, esto representa pérdida de funcionalidad del backtester.
- **Cambios menores de estilo y reordenación:** varias utilidades y bloques se reestructuraron; comprobar importaciones/exports si existe código que importe funciones desde la ruta original.

5 Pruebas unitarias sugeridas

Para prevenir regresiones similares a la detectada recomiendo el siguiente conjunto mínimo de tests:

- **Unit test de eval_epoch/train_epoch:** construir un `DataLoader` artificial con 2-3 batches y verificar que la pérdida devuelta es promedio ponderado por batch size (no sólo último batch).
- **Test de create_sequences_from_df:** pequeña serie conocida con valores exactos; comparar `y_ret` y `y_vol` con valores precomputados.
- **Persistencia de modelos:** guardar y cargar con `save_model`/`load_model`; verificar que la carga produce modelo con mismo número de parámetros y que una inferencia con tensores aleatorios no produce excepciones.
- **Backtester determinista:** controlar semilla y comprobar que, para dataset sintético, la secuencia de órdenes es constante entre ejecuciones.

6 Recomendaciones operacionales y de diseño

1. **Corrección inmediata:** aplicar el parche a `eval_epoch` (sección correspondiente).
2. **Restaurar o centralizar helpers de simulación:** si `simulate_fill`/`quantize_*` fueron eliminadas accidentalmente, restaurarlas desde `.old` o mover a `utils/backtesting.py` e importar desde ahí.
3. **Guardar metadatos del modelo en el mismo archivo:** usar la convención `torch.save({'state':..., 'meta':...}, path)` y adaptar `load_model` a buscar `'state'` y `'meta'`.

4. **Normalización única y reproducible:** proveer una clase/objeto `FeaturePipeline` que contenga `scaler` + lista de columnas y que se serialize (pickle) con el modelo. Asegurar que `create_sequences_from_df` reciba features ya escala-das o reciba pipeline para aplicar la misma transformación.

5. **Pruebas automáticas y CI:** agregar pipeline CI (tox/pytest) que corra los tests unitarios y verifique funciones críticas (train/eval/backtest/load/save).

6. **Instrumentación y logging:** mejorar mensajes en `load_model` cuando falla carga estricta, enumerando keys faltantes/extra para facilitar debugging.

7 Checklist de correcciones y tareas (priorizadas)

Críticas (aplicar inmediatamente)

- Parche `eval_epoch` (ver sección anterior).
- Restaurar/centralizar `simulate_fill` y las funciones `quantize_*` o actualizar la implementación del backtester para usar su reemplazo definido expresamente.
- Añadir tests que cubran training/evaluation para detectar regresiones futuras.

Importantes

- Refactor `load_model` para soportar `torch.save('state':..., 'meta':...)` y el prefijo `'module.'` de `DataParallel`.
- Implementar pipeline de features serializable y asegurar uso consistente en entrenamiento y backtesting.
- Añadir logging detallado en procesos de evaluación y en el evolutor (para diagnosticar por qué un individuo falla).

Menores

- Mejorar docstrings y añadir ejemplos de uso en la cabecera del archivo.
- Eliminar archivos duplicados o respaldos con nombres problemáticos (ej. nombres con espacios).
- Añadir parámetros para tipo de retorno en `create_sequences_from_df` (simple vs log).

8 Anexo: Patch sugerido (diff conceptual)

Parche clave: restaurar **`eval_epoch`**

Reemplazar la implementación actual de `eval_epoch` por la versión listada en la sección `Parche sugerido`. Adicionalmente, agregar un test unitario que crea un `DataLoader` con 2 batches y verifica que el promedio de pérdida coincide con la expectativa.

9 Conclusión

He revisado `fiboevo.py` y su respaldo `fiboevo.py.old`. La versión actual contiene una regresión crítica en `eval_epoch` (indentación) que afecta cálculo de métricas de validación; además faltan utilidades de simulación presentes en la versión `.old`. Propuse un parche listo para aplicar, recomendé mejoras de persistencia, normalización y pruebas automáticas. Si lo deseas, puedo:

- Aplicar el parche automáticamente sobre `fiboevo.py` y entregarte el archivo parcheado.
- Generar tests de `pytest` mínimos para `train_epoch`/`eval_epoch`/`create_sequences_from_df`.
- Extraer y proponer la reubicación de `simulate_fill` y funciones relacionadas en un módulo `utils/backtesting.py` con pruebas unitarias.
- Entregar un `diff` unificado (patch .diff) listo para aplicar con `git apply`.