Directory structure:

```
└── wartim01-claude_bot/
    └── crypto_trading_bot_CLAUDE/
        ├── README.md
        ├── backtest.py
        ├── concigne.txt
        ├── download_data.py
        ├── evaluate_model.py
        ├── install.py
        ├── main.py
        ├── requirements.txt
        ├── structure globale.txt
        ├── train_model.py
        ├── .env
        ├── __pycache__/
        ├── ai/
        |   ├── __init__.py
        |   ├── market_anomaly_detector.py
        |   ├── parameter_optimizer.py
        |   ├── reasoning_engine.py
        |   ├── scoring_engine.py
        |   ├── trade_analyzer.py
        |   ├── __pycache__/
        |   └── models/
        |       ├── __init__.py
        |       ├── attention.py
        |       ├── continuous_learning.py
        |       ├── ensemble.py
        |       ├── feature_engineering.py
        |       ├── lstm_model.py
        |       ├── model_trainer.py
```

```
|   ├── model_validator.py
|   └── __pycache__/
├── config/
|   ├── __init__.py
|   ├── config.py
|   ├── model_params.py
|   ├── trading_params.py
|   └── __pycache__/
├── core/
|   ├── __init__.py
|   ├── adaptive_risk_manager.py
|   ├── api_connector.py
|   ├── data_fetcher.py
|   ├── order_manager.py
|   ├── position_tracker.py
|   ├── risk_manager.py
|   └── __pycache__/
├── dashboard/
|   ├── __init__.py
|   ├── app.py
|   ├── model_dashboard.py
|   ├── model_monitor.py
|   └── trade_dashboard.py
├── indicators/
|   ├── __init__.py
|   ├── advanced_features.py
|   ├── market_metrics.py
|   ├── momentum.py
|   ├── trend.py
|   ├── volatility.py
|   ├── volume.py
```

```
|   └── __pycache__/
├── logs/
├── strategies/
|   ├── __init__.py
|   ├── hybrid_strategy.py
|   ├── market_state.py
|   ├── strategy_base.py
|   ├── technical_bounce.py
|   └── __pycache__/
├── tests/
|   ├── __init__.py
|   ├── test_api_connection.py
|   ├── test_indicators.py
|   ├── test_risk_manager.py
|   ├── test_strategies.py
|   ├── __pycache__/
|   └── test_models/
|       ├── __init__.py
|       ├── test_backtesting.py
|       ├── test_feature_eng.py
|       └── test_lstm.py
└── utils/
    ├── __init__.py
    ├── backtest_engine.py
    ├── logger.py
    ├── model_backtester.py
    ├── model_explainer.py
    ├── model_monitor.py
    ├── notification_service.py
    ├── visualizer.py
    └── __pycache__/
```

==========

File: crypto_trading_bot_CLAUDE/README.md

==========

□ Bot de Trading Crypto avec IA Auto-Adaptative

Un bot de trading crypto sophistiqué basé sur une IA auto-adaptative, conçu pour identifier et trader les rebonds techniques après des baisses de prix.

📋 Fonctionnalités Principales

- **Stratégie de rebond technique** optimisée pour capturer les corrections haussières
- **IA auto-améliorante** qui analyse ses propres performances et ajuste ses paramètres
- **Système de gestion des risques** avec stop-loss et take-profit adaptatifs
- **Trailing stop dynamique** qui se resserre avec l'augmentation du profit
- **Module de backtest** complet pour tester la stratégie sur des données historiques
- **Visualisations** des performances et des trades individuels
- **Compatibilité avec Binance** (testnet et production)

🛠️ Installation

Prérequis

- Python 3.8 ou supérieur
- Compte Binance (standard ou testnet)
- Clés API Binance avec permissions de trading

Installation Automatique

Utilisez le script d'installation automatique qui vous guidera à travers le processus :

```bash
```

python install.py
```

Le script effectuera les actions suivantes :

1. Vérification de la version de Python
2. Installation des dépendances requises
3. Création des répertoires nécessaires
4. Configuration des paramètres Binance
5. Personnalisation des paramètres de trading (optionnel)
6. Exécution de tests de connexion

### Installation Manuelle

Si vous préférez une installation manuelle, suivez ces étapes :

1. Clonez le dépôt :
```bash
git clone https://github.com/votre-username/crypto-trading-bot.git
cd crypto-trading-bot
```

2. Installez les dépendances :
```bash
pip install -r requirements.txt
```

3. Créez un fichier `.env` à la racine du projet avec le contenu suivant :
```
BINANCE_API_KEY=votre_clé_api
BINANCE_API_SECRET=votre_clé_secrète
USE_TESTNET=True

```

4. Créez les répertoires nécessaires :

```bash
mkdir -p data/market_data data/trade_logs data/performance logs
```

🚀 Utilisation

Mode Test (Dry Run)

Pour exécuter le bot sans passer d'ordres réels (recommandé pour les tests) :

```bash
python main.py --dry-run
```

Mode Production

Pour exécuter le bot avec trading réel :

```bash
python main.py
```

Options de Configuration

Vous pouvez personnaliser différents aspects du bot en modifiant les fichiers de configuration :

- `config/config.py` : Configuration globale du bot
- `config/trading_params.py` : Paramètres spécifiques à la stratégie de trading
```

Réglage de la Stratégie

Les paramètres que vous pouvez ajuster dans `trading_params.py` incluent :

```python
# Paramètres de gestion des risques
RISK_PER_TRADE_PERCENT = 7.5  # Pourcentage du capital risqué par trade (5-10%)
STOP_LOSS_PERCENT = 4.0       # Pourcentage de stop-loss (3-5%)
TAKE_PROFIT_PERCENT = 6.0     # Pourcentage de take-profit (5-7%)
LEVERAGE = 3                  # Effet de levier (jusqu'à 5x)

# Seuils techniques
RSI_OVERSOLD = 30             # Seuil de survente du RSI
MINIMUM_SCORE_TO_TRADE = 70   # Score minimum pour entrer en position (0-100)
```

□ Backtesting

Le bot inclut un système de backtest complet pour évaluer la stratégie sur des données historiques.

Exécution d'un Backtest

```bash
python backtest.py --symbol BTCUSDT --timeframe 15m --start 2023-01-01 --end 2023-06-30 --capital 200
python download_data.py --symbol BTCUSDT --interval 15m --start 2023-01-01 --end 2024-12-30
```

Options de Backtest

- `--symbol` : Paire de trading (ex: BTCUSDT)
- `--timeframe` : Intervalle de temps (ex: 15m, 1h, 4h)
- `--start` : Date de début (YYYY-MM-DD)
- `--end` : Date de fin (YYYY-MM-DD)
- `--capital` : Capital initial en USDT
- `--strategy` : Stratégie à tester (par défaut: technical_bounce)

Visualisation des Résultats

Les résultats du backtest sont sauvegardés dans le répertoire `data/backtest_results` et incluent :
- Fichier JSON avec les statistiques complètes
- Graphique de la courbe d'équité
- Graphique de la distribution des profits/pertes

📊 Performances et Analyse

Le bot inclut des outils d'analyse pour évaluer ses performances :

Analyse des Trades

Pour générer une analyse des trades récents :

```python
from ai.trade_analyzer import TradeAnalyzer
analyzer = TradeAnalyzer(scoring_engine, position_tracker)
report = analyzer.analyze_recent_trades(days=30)
```

Visualisation des Performances

Pour générer des visualisations de performance :

```python
from utils.visualizer import TradeVisualizer
visualizer = TradeVisualizer(position_tracker)
equity_curve = visualizer.plot_equity_curve(days=30)
trade_analysis = visualizer.plot_trade_analysis(days=30)
```

🧠 Système d'IA Auto-Adaptative

L'IA du bot s'améliore automatiquement en analysant ses performances passées.

Composants de l'IA

- **Moteur de Scoring** : Évalue les opportunités de trading selon multiples critères
- **Analyseur de Trades** : Identifie les patterns de succès et d'échec
- **Optimiseur de Paramètres** : Ajuste les paramètres en fonction des résultats
- **Moteur de Raisonnement** : Génère des explications textuelles pour les décisions

Cycle d'Apprentissage

1. L'IA analyse les opportunités de trading et attribue un score
2. Le bot exécute les trades avec un score suffisant
3. L'IA analyse les résultats des trades fermés
4. Les poids des différents facteurs sont ajustés en conséquence
5. Les paramètres de la stratégie sont optimisés périodiquement

📝 Journal des Trades

Chaque trade est enregistré avec des détails complets, incluant :

- Opportunité initiale avec score et raisonnement

- Conditions de marché lors de l'entrée

- Performance réelle (PnL)

- Auto-critique de l'IA

Les journaux sont stockés au format JSON dans `data/trade_logs/` et peuvent être analysés pour comprendre les décisions du bot.

⚠ Avertissements et Risques

- **Risque de Perte** : Le trading de crypto-monnaies comporte des risques significatifs. N'investissez que ce que vous pouvez vous permettre de perdre.

- **Effet de Levier** : L'utilisation de l'effet de levier amplifie les profits mais aussi les pertes.

- **Tests** : Commencez toujours en mode test ou avec de petits montants avant d'engager des sommes importantes.

- **Maintenance** : Surveillez régulièrement le bot et son fonctionnement.

📜 License

Ce projet est distribué sous licence MIT. Voir le fichier `LICENSE` pour plus de détails.

🤝 Contribution

Les contributions sont les bienvenues! N'hésitez pas à soumettre des pull requests ou à signaler des problèmes.

📧 Contact

Pour toute question ou suggestion, veuillez me contacter à [votre-email@exemple.com].

==========

File: crypto_trading_bot_CLAUDE/backtest.py

```
================================================
# backtest.py
"""
Script de backtest pour la stratégie de trading
"""
import os
import json
import pandas as pd
import numpy as np
import argparse
import matplotlib.pyplot as plt
from typing import Dict, List, Optional, Union
from datetime import datetime, timedelta

from config.config import DATA_DIR
from config.trading_params import (
    RISK_PER_TRADE_PERCENT,
    STOP_LOSS_PERCENT,
    TAKE_PROFIT_PERCENT,
    LEVERAGE,
    MINIMUM_SCORE_TO_TRADE
)

from strategies.technical_bounce import TechnicalBounceStrategy
from ai.scoring_engine import ScoringEngine
from utils.logger import setup_logger

logger = setup_logger("backtest")

class BacktestEngine:
    """
```

```
Moteur de backtest pour les stratégies de trading
"""
def __init__(self, data_dir: str = None):
    self.data_dir = data_dir or os.path.join(DATA_DIR, "market_data")
    self.results_dir = os.path.join(DATA_DIR, "backtest_results")

    # Créer les répertoires si nécessaires
    for directory in [self.data_dir, self.results_dir]:
        if not os.path.exists(directory):
            os.makedirs(directory)

    # Initialiser les composants
    self.scoring_engine = ScoringEngine()

def load_data(self, symbol: str, timeframe: str, start_date: str, end_date: str) -> pd.DataFrame:
    """
    Charge les données historiques pour le backtest

    Args:
        symbol: Paire de trading
        timeframe: Intervalle de temps
        start_date: Date de début (YYYY-MM-DD)
        end_date: Date de fin (YYYY-MM-DD)

    Returns:
        DataFrame avec les données OHLCV
    """
    # Construire le chemin du fichier
    filename = f"{symbol}_{timeframe}_{start_date}_{end_date}.csv"
    filepath = os.path.join(self.data_dir, filename)
```

```
    # Vérifier si le fichier existe déjà
    if os.path.exists(filepath):
        logger.info(f"Chargement des données depuis {filepath}")
        df = pd.read_csv(filepath, parse_dates=['timestamp'])
        df.set_index('timestamp', inplace=True)
        return df

    # Si le fichier n'existe pas, vous pouvez implémenter la récupération des données
    # depuis une API externe (Binance, etc.)
    logger.error(f"Fichier de données non trouvé: {filepath}")
    return pd.DataFrame()

def run_backtest(self, symbol: str, timeframe: str, start_date: str, end_date: str,
                 initial_capital: float = 200, strategy_name: str = "technical_bounce") -> Dict:
    """
    Exécute un backtest sur la période spécifiée

    Args:
        symbol: Paire de trading
        timeframe: Intervalle de temps
        start_date: Date de début (YYYY-MM-DD)
        end_date: Date de fin (YYYY-MM-DD)
        initial_capital: Capital initial (USDT)
        strategy_name: Nom de la stratégie

    Returns:
        Résultats du backtest
    """
    # Charger les données
    data = self.load_data(symbol, timeframe, start_date, end_date)
```

```
    if data.empty:
        return {
            "success": False,
            "message": "Données non disponibles pour le backtest"
        }

    # Sélectionner la stratégie
    if strategy_name == "technical_bounce":
        strategy = self._create_technical_bounce_strategy()
    else:
        return {
            "success": False,
            "message": f"Stratégie non reconnue: {strategy_name}"
        }

    # Simuler le trading
    backtest_results = self._simulate_trading(data, strategy, initial_capital, symbol)

    # Sauvegarder les résultats
    self._save_backtest_results(backtest_results, symbol, strategy_name, start_date, end_date)

    return backtest_results

def _create_technical_bounce_strategy(self) -> TechnicalBounceStrategy:
    """
    Crée une instance de la stratégie de rebond technique

    Returns:
        Instance de la stratégie
    """
    # Créer un data fetcher simulé
```

```
class MockDataFetcher:
    def __init__(self, backtest_data=None):
        self.backtest_data = backtest_data

    def get_current_price(self, symbol):
        if self.backtest_data is None or self.backtest_data.empty:
            return 0
        return self.backtest_data["close"].iloc[-1]

    def get_ohlcv(self, symbol, timeframe, limit=100):
        if self.backtest_data is None or self.backtest_data.empty:
            return pd.DataFrame()
        return self.backtest_data.tail(limit)

    def get_market_data(self, symbol):
        """
        Simule la méthode get_market_data pour le backtest

        Args:
            symbol: Paire de trading

        Returns:
            Dictionnaire avec les données de marché simulées
        """
        if self.backtest_data is None or self.backtest_data.empty:
            return {
                "symbol": symbol,
                "current_price": 0,
                "primary_timeframe": {"ohlcv": pd.DataFrame()},
                "secondary_timeframes": {}
            }
```

```
# Calculer les indicateurs
from indicators.trend import calculate_ema, calculate_adx
from indicators.momentum import calculate_rsi
from indicators.volatility import calculate_bollinger_bands, calculate_atr

# Obtenir les 100 dernières lignes pour les calculs
data = self.backtest_data.tail(100).copy()

# Calculer les indicateurs
ema = calculate_ema(data)
rsi = calculate_rsi(data)
bollinger = calculate_bollinger_bands(data)
atr = calculate_atr(data)
adx = calculate_adx(data)

# Créer le dictionnaire de données de marché avec timeframes secondaires simulés
market_data = {
    "symbol": symbol,
    "current_price": data["close"].iloc[-1],
    "primary_timeframe": {
        "ohlcv": data,
        "indicators": {
            "ema": ema,
            "rsi": rsi,
            "bollinger": bollinger,
            "atr": atr,
            "adx": adx
        }
    },
    "secondary_timeframes": {
```

```
            "1h": {
                "ohlcv": data,  # Pour simplifier, on utilise les mêmes données
                "indicators": {
                    "rsi": rsi,
                    "bollinger": bollinger
                }
            }
        }
    }

    return market_data

def detect_volume_spike(self, symbol):
    return {
        "spike": False,
        "ratio": 1.0,
        "bullish": None,
        "details": {}
    }

# Créer un market analyzer simulé
class MockMarketAnalyzer:
    def analyze_market_state(self, symbol):
        return {
            "favorable": True,
            "cooldown": False,
            "details": {}
        }

# Créer les instances
mock_data_fetcher = MockDataFetcher()
```

```
        mock_market_analyzer = MockMarketAnalyzer()

        # Créer et retourner l'instance de la stratégie
        return TechnicalBounceStrategy(mock_data_fetcher, mock_market_analyzer,
self.scoring_engine)

    def _simulate_trading(self, data: pd.DataFrame, strategy, initial_capital: float, symbol: str) -> Dict:
        """
        Simule le trading sur des données historiques

        Args:
            data: DataFrame avec les données OHLCV
            strategy: Stratégie de trading
            initial_capital: Capital initial
            symbol: Paire de trading

        Returns:
            Résultats de la simulation
        """
        # Initialiser les variables de simulation
        equity = initial_capital
        position = None
        trades = []
        equity_curve = [initial_capital]
        dates = [data.index[0]]

        # Mettre à jour les données dans le data fetcher simulé
        strategy.data_fetcher.backtest_data = data.iloc[:50]  # Commencer avec les 50 premières lignes

        # Simuler chaque jour de trading
        for i in range(51, len(data)):
```

```
# Mettre à jour les données simulées (fenêtre glissante)
current_data = data.iloc[i-50:i]
strategy.data_fetcher.backtest_data = current_data

current_price = current_data["close"].iloc[-1]
current_date = current_data.index[-1]

# Gérer les positions ouvertes
if position:
    # Vérifier si le stop-loss est atteint
    if current_price <= position["stop_loss"]:
        pnl = (current_price - position["entry_price"]) / position["entry_price"] * 100 * LEVERAGE
        equity = equity * (1 + pnl/100)

        trades.append({
            "entry_date": position["entry_date"],
            "exit_date": current_date,
            "entry_price": position["entry_price"],
            "exit_price": current_price,
            "pnl_percent": pnl,
            "exit_reason": "Stop-Loss"
        })

        position = None

    # Vérifier si le take-profit est atteint
    elif current_price >= position["take_profit"]:
        pnl = (current_price - position["entry_price"]) / position["entry_price"] * 100 * LEVERAGE
        equity = equity * (1 + pnl/100)

        trades.append({
```

```
                "entry_date": position["entry_date"],
                "exit_date": current_date,
                "entry_price": position["entry_price"],
                "exit_price": current_price,
                "pnl_percent": pnl,
                "exit_reason": "Take-Profit"
            })

            position = None

        # Chercher de nouvelles opportunités si aucune position n'est ouverte
        if not position:
            opportunity = strategy.find_trading_opportunity(symbol)

            if opportunity and opportunity["score"] >= strategy.min_score:
                # Calculer la taille de position
                position_size = equity * (RISK_PER_TRADE_PERCENT/100) / (STOP_LOSS_PERCENT/100) * LEVERAGE

                # Ouvrir une position
                position = {
                    "entry_date": current_date,
                    "entry_price": current_price,
                    "stop_loss": current_price * (1 - STOP_LOSS_PERCENT/100),
                    "take_profit": current_price * (1 + TAKE_PROFIT_PERCENT/100),
                    "size": position_size,
                    "score": opportunity["score"]
                }

        # Enregistrer l'équité
        equity_curve.append(equity)
```

```
            dates.append(current_date)

        # Clôturer la position à la fin de la simulation si nécessaire
        if position:
            final_price = data["close"].iloc[-1]
            pnl = (final_price - position["entry_price"]) / position["entry_price"] * 100 * LEVERAGE
            equity = equity * (1 + pnl/100)

            trades.append({
                "entry_date": position["entry_date"],
                "exit_date": data.index[-1],
                "entry_price": position["entry_price"],
                "exit_price": final_price,
                "pnl_percent": pnl,
                "exit_reason": "Fin de simulation"
            })

            equity_curve[-1] = equity

        # Calculer les statistiques du backtest
        total_trades = len(trades)
        winning_trades = [t for t in trades if t["pnl_percent"] > 0]
        losing_trades = [t for t in trades if t["pnl_percent"] <= 0]

        win_rate = len(winning_trades) / total_trades * 100 if total_trades > 0 else 0
        avg_win = sum(t["pnl_percent"] for t in winning_trades) / len(winning_trades) if winning_trades else 0
        avg_loss = sum(t["pnl_percent"] for t in losing_trades) / len(losing_trades) if losing_trades else 0

        profit_factor = abs(sum(t["pnl_percent"] for t in winning_trades)) / abs(sum(t["pnl_percent"] for t in losing_trades)) if losing_trades and sum(t["pnl_percent"] for t in losing_trades) != 0 else float('inf')
```

```
    max_drawdown = self._calculate_max_drawdown(equity_curve)
    sharpe_ratio = self._calculate_sharpe_ratio(equity_curve)

    # Préparer les résultats
    results = {
        "success": True,
        "symbol": symbol,
        "initial_capital": initial_capital,
        "final_equity": equity,
        "total_return": (equity - initial_capital) / initial_capital * 100,
        "total_trades": total_trades,
        "winning_trades": len(winning_trades),
        "losing_trades": len(losing_trades),
        "win_rate": win_rate,
        "avg_win": avg_win,
        "avg_loss": avg_loss,
        "profit_factor": profit_factor,
        "max_drawdown": max_drawdown,
        "sharpe_ratio": sharpe_ratio,
        "trades": trades,
        "equity_curve": equity_curve,
        "dates": [str(d) for d in dates]
    }

    return results

def _calculate_max_drawdown(self, equity_curve: List[float]) -> float:
    """
    Calcule le drawdown maximum
```

```
    Args:
        equity_curve: Liste des valeurs d'équité

    Returns:
        Drawdown maximum en pourcentage
    """
    max_dd = 0
    peak = equity_curve[0]

    for equity in equity_curve:
        if equity > peak:
            peak = equity

        dd = (peak - equity) / peak * 100
        max_dd = max(max_dd, dd)

    return max_dd

def _calculate_sharpe_ratio(self, equity_curve: List[float], risk_free_rate: float = 0.01) -> float:
    """
    Calcule le ratio de Sharpe

    Args:
        equity_curve: Liste des valeurs d'équité
        risk_free_rate: Taux sans risque annuel

    Returns:
        Ratio de Sharpe
    """
    # Calculer les rendements quotidiens
    daily_returns = []
```

```
    for i in range(1, len(equity_curve)):
        daily_return = (equity_curve[i] - equity_curve[i-1]) / equity_curve[i-1]
        daily_returns.append(daily_return)

    # Calculer la moyenne et l'écart-type des rendements
    if not daily_returns:
        return 0

    avg_return = sum(daily_returns) / len(daily_returns)
    std_return = np.std(daily_returns) if len(daily_returns) > 1 else 0

    # Annualiser les rendements (252 jours de trading par an)
    annual_return = avg_return * 252
    annual_std = std_return * np.sqrt(252)

    # Calculer le ratio de Sharpe
    if annual_std == 0:
        return 0

    sharpe_ratio = (annual_return - risk_free_rate) / annual_std

    return sharpe_ratio

def _save_backtest_results(self, results: Dict, symbol: str, strategy_name: str,
                           start_date: str, end_date: str) -> None:
    """
    Sauvegarde les résultats du backtest

    Args:
        results: Résultats du backtest
```

```
        symbol: Paire de trading
        strategy_name: Nom de la stratégie
        start_date: Date de début
        end_date: Date de fin
    """
    if not results.get("success", False):
        return

    # Créer le nom du fichier
    filename = f"{symbol}_{strategy_name}_{start_date}_{end_date}.json"
    filepath = os.path.join(self.results_dir, filename)

    # Sauvegarder les résultats
    try:
        with open(filepath, 'w') as f:
            json.dump(results, f, indent=2, default=str)

        logger.info(f"Résultats du backtest sauvegardés: {filepath}")
    except Exception as e:
        logger.error(f"Erreur lors de la sauvegarde des résultats: {str(e)}")

    # Générer les graphiques
    self._generate_backtest_charts(results, symbol, strategy_name, start_date, end_date)

def _generate_backtest_charts(self, results: Dict, symbol: str, strategy_name: str,
                              start_date: str, end_date: str) -> None:
    """
    Génère des graphiques pour les résultats du backtest

    Args:
        results: Résultats du backtest
```

```
    symbol: Paire de trading
    strategy_name: Nom de la stratégie
    start_date: Date de début
    end_date: Date de fin
"""
if not results.get("success", False):
    return

# Créer le répertoire pour les graphiques
charts_dir = os.path.join(self.results_dir, "charts")
if not os.path.exists(charts_dir):
    os.makedirs(charts_dir)

# Créer le nom de base pour les fichiers
base_filename = f"{symbol}_{strategy_name}_{start_date}_{end_date}"

# 1. Graphique de la courbe d'équité
plt.figure(figsize=(12, 6))
plt.plot(results["equity_curve"])
plt.title(f"Courbe d'Équité - {symbol} ({start_date} à {end_date})")
plt.xlabel("Jours")
plt.ylabel("Équité (USDT)")
plt.grid(True)

# Ajouter des annotations
plt.annotate(f"Rendement total: {results['total_return']:.2f}%\n"
             f"Drawdown max: {results['max_drawdown']:.2f}%\n"
             f"Ratio de Sharpe: {results['sharpe_ratio']:.2f}",
             xy=(0.02, 0.95),
             xycoords='axes fraction',
             fontsize=10,
```

```
        bbox=dict(boxstyle="round,pad=0.3", fc="white", ec="gray", alpha=0.8))

# Sauvegarder le graphique
equity_chart_path = os.path.join(charts_dir, f"{base_filename}_equity.png")
plt.savefig(equity_chart_path)
plt.close()

# 2. Graphique de la distribution des profits/pertes
pnl_values = [t["pnl_percent"] for t in results["trades"]]

plt.figure(figsize=(10, 6))
plt.hist(pnl_values, bins=20, alpha=0.7, color='skyblue')
plt.title(f"Distribution des Profits/Pertes - {symbol}")
plt.xlabel("Profit/Perte (%)")
plt.ylabel("Fréquence")
plt.axvline(x=0, color='r', linestyle='--')
plt.grid(True, alpha=0.3)

# Ajouter des annotations
plt.annotate(f"Trades: {results['total_trades']}\n"
        f"Win Rate: {results['win_rate']:.1f}%\n"
        f"Gain moyen: {results['avg_win']:.2f}%\n"
        f"Perte moyenne: {results['avg_loss']:.2f}%",
        xy=(0.02, 0.95),
        xycoords='axes fraction',
        fontsize=10,
        bbox=dict(boxstyle="round,pad=0.3", fc="white", ec="gray", alpha=0.8))

# Sauvegarder le graphique
distribution_chart_path = os.path.join(charts_dir, f"{base_filename}_distribution.png")
plt.savefig(distribution_chart_path)
```

```
        plt.close()


if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Backtest de stratégies de trading")
    parser.add_argument("--symbol", type=str, default="BTCUSDT", help="Paire de trading")
    parser.add_argument("--timeframe", type=str, default="15m", help="Intervalle de temps")
    parser.add_argument("--start", type=str, required=True, help="Date de début (YYYY-MM-DD)")
    parser.add_argument("--end", type=str, required=True, help="Date de fin (YYYY-MM-DD)")
    parser.add_argument("--capital", type=float, default=200, help="Capital initial (USDT)")
    parser.add_argument("--strategy", type=str, default="technical_bounce", help="Stratégie de
trading")

    args = parser.parse_args()

    # Exécuter le backtest
    engine = BacktestEngine()
    results = engine.run_backtest(
        symbol=args.symbol,
        timeframe=args.timeframe,
        start_date=args.start,
        end_date=args.end,
        initial_capital=args.capital,
        strategy_name=args.strategy
    )

    if results.get("success", False):
        print(f"Backtest réussi pour {args.symbol} ({args.start} à {args.end})")
        print(f"Rendement total: {results['total_return']:.2f}%")
        print(f"Nombre de trades: {results['total_trades']}")
        print(f"Win rate: {results['win_rate']:.1f}%")
```

```
        print(f"Drawdown maximum: {results['max_drawdown']:.2f}%")
        print(f"Ratio de Sharpe: {results['sharpe_ratio']:.2f}")
    else:
        print(f"Échec du backtest: {results.get('message', 'Erreur inconnue')}")
```

==========

File: crypto_trading_bot_CLAUDE/concigne.txt

==========

je suis entrain de faire l'implémentation du : 1. **Prédiction multi-horizon et multi-facteur**: * Prédire simultanément les mouvements du marché à court terme (1-4h), moyen terme (1-3j) et long terme (1-2sem) * Générer des prévisions de volatilité, volume et momentum en plus des directions de prix * Créer un système d'alerte précoce pour les retournements de marché majeurs 2. : * Implémenter une architecture LSTM bidirectionnelle avec mécanisme d'attention * Intégrer des connexions résiduelles pour une meilleure propagation du gradient * Utiliser l'apprentissage par transfert entre différentes paires de trading * Inclure des mécanismes de régularisation avancés (dropout spatial, batch normalization) 3. ****Gestion des risques adaptative et sophistiquée****: * Calculer dynamiquement les tailles de position optimales basées sur la confiance du modèle * Ajuster automatiquement les niveaux de stop-loss/take-profit selon les prédictions de volatilité * Développer un système anti-fragile qui devient plus conservateur après des séquences de pertes * Créer un mécanisme de détection de "black swan events" pour réduire rapidement l'exposition 4. ****Apprentissage continu avec protection****: * Mettre en place un apprentissage incrémental avec détection de concept drift * Implémenter un système de garde-fous contre l'oubli catastrophique * Créer une mémoire à long terme des patterns de marché efficaces * Développer un framework d'auto-évaluation de la qualité des prédictions 5. ****Explicabilité et monitoring avancés****: * Créer des visualisations détaillées des facteurs influençant chaque décision * Développer un tableau de bord temps réel de la performance du modèle * Implémenter un système de logs hiérarchiques pour déboguer le comportement du modèle * Générer des rapports d'attribution de performance entre le modèle et les règles classiques Instructions pour l'implémentation Veuillez me fournir un guide complet et le code nécessaire pour intégrer un modèle LSTM avancé dans mon bot de trading. Le système doit être complexe mais extrêmement robuste, avec un backtesting rigoureux incluant des tests de résistance aux conditions de marché extrêmes. Le code doit s'inspirer des architectures les plus performantes en finance quantitative et NLP moderne (transformers, mécanismes d'attention), tout en s'intégrant harmonieusement à mon architecture existante

==========

File: crypto_trading_bot_CLAUDE/download_data.py

==========

```
# download_data.py
import os
```

```
import pandas as pd
import numpy as np
import time
from datetime import datetime, timedelta
import requests
from config.config import DATA_DIR

def download_binance_data(symbol, interval, start_date, end_date, save_path=None):
    """
    Télécharge les données historiques OHLCV depuis l'API publique de Binance

    Args:
        symbol: Paire de trading (ex: BTCUSDT)
        interval: Intervalle de temps (1m, 5m, 15m, 1h, 4h, 1d, etc.)
        start_date: Date de début au format 'YYYY-MM-DD'
        end_date: Date de fin au format 'YYYY-MM-DD'
        save_path: Chemin pour sauvegarder les données (optionnel)

    Returns:
        DataFrame pandas avec les données OHLCV
    """
    # Convertir les dates en millisecondes pour l'API Binance
    start_ts = int(datetime.strptime(start_date, '%Y-%m-%d').timestamp() * 1000)
    end_ts = int(datetime.strptime(end_date, '%Y-%m-%d').timestamp() * 1000)

    # URL de l'API Binance
    url = 'https://api.binance.com/api/v3/klines'

    # Liste pour stocker toutes les données
    all_klines = []
```

```
# Binance limite à 1000 chandeliers par requête
# Nous devons faire plusieurs requêtes pour couvrir toute la période
current_ts = start_ts

while current_ts < end_ts:
    # Paramètres de la requête
    params = {
        'symbol': symbol,
        'interval': interval,
        'startTime': current_ts,
        'endTime': end_ts,
        'limit': 1000
    }

    # Effectuer la requête
    response = requests.get(url, params=params)

    # Vérifier la réponse
    if response.status_code != 200:
        print(f"Erreur lors de la requête : {response.text}")
        return None

    # Convertir la réponse en JSON
    data = response.json()

    if not data:
        break

    # Ajouter les données à la liste
    all_klines.extend(data)
```

```
            # Mettre à jour le timestamp pour la prochaine requête
            current_ts = data[-1][0] + 1

            # Attendre un peu pour éviter de dépasser les limites de l'API
            time.sleep(0.5)

            print(f"Téléchargement en cours... {len(all_klines)} chandeliers récupérés")

    # Convertir les données en DataFrame pandas
    df = pd.DataFrame(all_klines, columns=[
        'timestamp', 'open', 'high', 'low', 'close', 'volume',
        'close_time', 'quote_asset_volume', 'number_of_trades',
        'taker_buy_base_asset_volume', 'taker_buy_quote_asset_volume', 'ignore'
    ])

    # Convertir les types de données
    df['timestamp'] = pd.to_datetime(df['timestamp'], unit='ms')

    for col in ['open', 'high', 'low', 'close', 'volume']:
        df[col] = pd.to_numeric(df[col], errors='coerce')

    # Sauvegarder les données si un chemin est fourni
    if save_path:
        df.to_csv(save_path, index=False)
        print(f"Données sauvegardées dans {save_path}")

    return df

if __name__ == "__main__":
    import argparse
```

```
    parser = argparse.ArgumentParser(description="Téléchargement de données historiques Binance")
    parser.add_argument("--symbol", type=str, default="BTCUSDT", help="Paire de trading")
    parser.add_argument("--interval", type=str, default="15m", help="Intervalle de temps")
    parser.add_argument("--start", type=str, required=True, help="Date de début (YYYY-MM-DD)")
    parser.add_argument("--end", type=str, required=True, help="Date de fin (YYYY-MM-DD)")

    args = parser.parse_args()

    # Créer le répertoire de données s'il n'existe pas
    market_data_dir = os.path.join(DATA_DIR, "market_data")
    if not os.path.exists(market_data_dir):
        os.makedirs(market_data_dir)

    # Construire le chemin de sauvegarde
    save_path = os.path.join(
        market_data_dir,
        f"{args.symbol}_{args.interval}_{args.start}_{args.end}.csv"
    )

    # Télécharger les données
    download_binance_data(
        args.symbol,
        args.interval,
        args.start,
        args.end,
        save_path
    )

================================================
File: crypto_trading_bot_CLAUDE/evaluate_model.py
================================================
```

```
#!/usr/bin/env python
# evaluate_model.py
"""
Script d'évaluation approfondie du modèle LSTM
"""
import os
import argparse
import pandas as pd
import numpy as np
from datetime import datetime, timedelta
import json
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix, classification_report, precision_recall_curve, roc_curve, auc

from ai.models.lstm_model import LSTMModel
from ai.models.feature_engineering import FeatureEngineering
from ai.models.model_validator import ModelValidator
from ai.models.continuous_learning import ContinuousLearning
from strategies.hybrid_strategy import HybridStrategy
from strategies.technical_bounce import TechnicalBounceStrategy
from core.adaptive_risk_manager import AdaptiveRiskManager
from config.config import DATA_DIR
from utils.logger import setup_logger

logger = setup_logger("evaluate_model")

def load_data(symbol: str, timeframe: str, start_date: str, end_date: str) -> pd.DataFrame:
    """
```

```
    Charge les données OHLCV depuis le disque

    Args:
        symbol: Paire de trading
        timeframe: Intervalle de temps
        start_date: Date de début (YYYY-MM-DD)
        end_date: Date de fin (YYYY-MM-DD)

    Returns:
        DataFrame avec les données OHLCV
    """
    # Construire le chemin du fichier
    data_path = os.path.join(DATA_DIR, "market_data",
f"{symbol}_{timeframe}_{start_date}_{end_date}.csv")

    # Vérifier si le fichier existe
    if not os.path.exists(data_path):
        logger.error(f"Fichier non trouvé: {data_path}")
        return pd.DataFrame()

    # Charger les données
    try:
        data = pd.read_csv(data_path)

        # Convertir la colonne timestamp en datetime
        if "timestamp" in data.columns:
            data["timestamp"] = pd.to_datetime(data["timestamp"])
            data.set_index("timestamp", inplace=True)

        logger.info(f"Données chargées: {len(data)} lignes")
        return data
```

```
    except Exception as e:
        logger.error(f"Erreur lors du chargement des données: {str(e)}")
        return pd.DataFrame()

def evaluate_direction_prediction(args):
    """
    Évalue la précision de la prédiction de direction sur des données de test

    Args:
        args: Arguments de ligne de commande
    """
    # Charger les données
    data = load_data(args.symbol, args.timeframe, args.start_date, args.end_date)

    if data.empty:
        logger.error("Données vides, impossible de continuer")
        return

    # Charger le modèle
    model_path = args.model_path or os.path.join(DATA_DIR, "models", "production", "lstm_final.h5")

    # Créer le validateur
    validator = ModelValidator()

    try:
        validator.load_model(model_path)
    except Exception as e:
        logger.error(f"Erreur lors du chargement du modèle: {str(e)}")
        return

    # Évaluer le modèle
```

```
logger.info("Évaluation du modèle...")
evaluation = validator.evaluate_on_test_set(data)

# Générer des graphiques pour chaque horizon
for horizon_key, metrics in evaluation["horizons"].items():
    horizon_name = horizon_key.replace("horizon_", "h")

    logger.info(f"\nHorizon: {horizon_name}")
    logger.info(f"Accuracy: {metrics['direction']['accuracy']:.4f}")
    logger.info(f"Precision: {metrics['direction']['precision']:.4f}")
    logger.info(f"Recall: {metrics['direction']['recall']:.4f}")
    logger.info(f"F1 Score: {metrics['direction']['f1_score']:.4f}")

    # Matrice de confusion
    cm = np.array(metrics["direction"]["confusion_matrix"])

    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
            xticklabels=["Baisse", "Hausse"],
            yticklabels=["Baisse", "Hausse"])
    plt.title(f"Matrice de confusion - {horizon_name}")
    plt.ylabel('Réalité')
    plt.xlabel('Prédiction')

    # Sauvegarder le graphique
    output_dir = os.path.join(DATA_DIR, "models", "evaluation", "figures")
    os.makedirs(output_dir, exist_ok=True)

    plt.savefig(os.path.join(output_dir, f"{args.symbol}_{horizon_name}_confusion_matrix.png"))
    plt.close()
```

```
    # Sauvegarder les résultats complets
    results_file = os.path.join(DATA_DIR, "models", "evaluation",
                                f"{args.symbol}_evaluation_{datetime.now().strftime('%Y%m%d')}.json")

    with open(results_file, 'w') as f:
        json.dump(evaluation, f, indent=2, default=str)

    logger.info(f"Résultats d'évaluation sauvegardés: {results_file}")

def backtest_trading_performance(args):
    """
    Effectue un backtest complet de la stratégie hybride vs stratégie de base

    Args:
        args: Arguments de ligne de commande
    """
    # Charger les données
    data = load_data(args.symbol, args.timeframe, args.start_date, args.end_date)

    if data.empty:
        logger.error("Données vides, impossible de continuer")
        return

    # Initialiser le validateur
    validator = ModelValidator()

    try:
        # Charger le modèle
        model_path = args.model_path or os.path.join(DATA_DIR, "models", "production",
"lstm_final.h5")
        validator.load_model(model_path)
```

```
except Exception as e:
    logger.error(f"Erreur lors du chargement du modèle: {str(e)}")
    return

# Backtest complet
logger.info(f"Backtest sur {args.symbol} du {args.start_date} au {args.end_date}...")

# Utiliser une capital initial personnalisé si spécifié
initial_capital = args.capital

# Comparer avec la stratégie de base
comparison = validator.compare_with_baseline(
    data,
    initial_capital=initial_capital
)

# Afficher les résultats
baseline = comparison["baseline"]
lstm = comparison["lstm"]
diff = comparison["comparison"]

logger.info("\n=== Résultats du backtest ===")
logger.info(f"Stratégie de base:")
logger.info(f"  Rendement: {baseline['return_pct']:.2f}%")
logger.info(f"  Drawdown max: {baseline['max_drawdown_pct']:.2f}%")
logger.info(f"  Ratio de Sharpe: {baseline['sharpe_ratio']:.2f}")
logger.info(f"  Nombre de trades: {baseline['num_trades']}")
logger.info(f"  Taux de réussite: {baseline['win_rate']:.2f}%")

logger.info(f"\nStratégie hybride LSTM:")
logger.info(f"  Rendement: {lstm['return_pct']:.2f}%")
```

```
logger.info(f"  Drawdown max: {lstm['max_drawdown_pct']:.2f}%")
logger.info(f"  Ratio de Sharpe: {lstm['sharpe_ratio']:.2f}")
logger.info(f"  Nombre de trades: {lstm['num_trades']}")
logger.info(f"  Taux de réussite: {lstm['win_rate']:.2f}%")

logger.info(f"\nDifférence (LSTM - Base):")
logger.info(f"  Rendement: {diff['return_difference']:.2f}%")
logger.info(f"  Amélioration drawdown: {diff['drawdown_improvement']:.2f}%")
logger.info(f"  Amélioration Sharpe: {diff['sharpe_improvement']:.2f}")

# Générer des graphiques

# 1. Courbes d'équité
plt.figure(figsize=(12, 6))

baseline_equity = comparison["equity_curves"]["baseline"]
lstm_equity = comparison["equity_curves"]["lstm"]

plt.plot(baseline_equity, label="Stratégie de base", color="blue", linewidth=2)
plt.plot(lstm_equity, label="Stratégie hybride LSTM", color="green", linewidth=2)

plt.title(f"Comparaison des courbes d'équité - {args.symbol}")
plt.xlabel("Jours de trading")
plt.ylabel("Équité (USDT)")
plt.legend()
plt.grid(True, alpha=0.3)

plt.savefig(os.path.join(DATA_DIR, "models", "evaluation", "figures",
                         f"{args.symbol}_equity_curves.png"))
plt.close()
```

```
# 2. Distribution des profits par trade
plt.figure(figsize=(12, 6))

baseline_profits = [t["profit_pct"] for t in comparison["trades"]["baseline"]]
lstm_profits = [t["profit_pct"] for t in comparison["trades"]["lstm"]]

plt.hist(baseline_profits, bins=20, alpha=0.5, label="Stratégie de base", color="blue")
plt.hist(lstm_profits, bins=20, alpha=0.5, label="Stratégie hybride LSTM", color="green")

plt.title(f"Distribution des profits par trade - {args.symbol}")
plt.xlabel("Profit (%)")
plt.ylabel("Nombre de trades")
plt.axvline(x=0, color='red', linestyle='--')
plt.legend()
plt.grid(True, alpha=0.3)

plt.savefig(os.path.join(DATA_DIR, "models", "evaluation", "figures",
                         f"{args.symbol}_profit_distribution.png"))
plt.close()

# Sauvegarder les résultats complets
results_file = os.path.join(DATA_DIR, "models", "evaluation",
                            f"{args.symbol}_backtest_{datetime.now().strftime('%Y%m%d')}.json")

with open(results_file, 'w') as f:
    json.dump(comparison, f, indent=2, default=str)

logger.info(f"Résultats du backtest sauvegardés: {results_file}")

# Simulation du trading sur la période complète
if args.simulate_hybrid:
```

```
        logger.info("\nSimulation du trading avec stratégie hybride...")
        simulate_hybrid_strategy(data, args.symbol, initial_capital, model_path)

def simulate_hybrid_strategy(data: pd.DataFrame, symbol: str,
                             initial_capital: float, model_path: str):
    """
    Simule le trading avec la stratégie hybride complète

    Args:
        data: Données OHLCV
        symbol: Paire de trading
        initial_capital: Capital initial
        model_path: Chemin du modèle LSTM
    """
    # Créer les composants nécessaires
    lstm_model = LSTMModel()
    lstm_model.load(model_path)

    feature_engineering = FeatureEngineering()
    adaptive_risk_manager = AdaptiveRiskManager(initial_capital=initial_capital)

    # Créer un data fetcher simulé pour le backtest
    class MockDataFetcher:
        def __init__(self, data):
            self.data = data
            self.current_idx = 0

        def get_current_price(self, symbol):
            return self.data['close'].iloc[self.current_idx]

        def get_market_data(self, symbol):
```

```
"""Simule la récupération des données de marché pour le backtest"""
# Obtenir les données récentes (jusqu'à l'indice actuel)
current_data = self.data.iloc[:self.current_idx+1].copy()

# Calculer les indicateurs sur ces données
from indicators.trend import calculate_ema, calculate_adx
from indicators.momentum import calculate_rsi
from indicators.volatility import calculate_bollinger_bands, calculate_atr

# Obtenir les 100 dernières lignes ou moins
window_start = max(0, self.current_idx - 99)
data_window = current_data.iloc[window_start:self.current_idx+1]

ema = calculate_ema(data_window)
rsi = calculate_rsi(data_window)
bollinger = calculate_bollinger_bands(data_window)
atr = calculate_atr(data_window)
adx = calculate_adx(data_window)

return {
   "symbol": symbol,
   "current_price": data_window["close"].iloc[-1],
   "primary_timeframe": {
      "ohlcv": data_window,
      "indicators": {
         "ema": ema,
         "rsi": rsi,
         "bollinger": bollinger,
         "atr": atr,
         "adx": adx
      }
```

```
        },
        "secondary_timeframes": {}
    }

# Créer un market analyzer simulé
class MockMarketAnalyzer:
    def analyze_market_state(self, symbol):
        return {
            "favorable": True,
            "cooldown": False,
            "details": {}
        }

# Créer un position tracker simulé
class MockPositionTracker:
    def __init__(self):
        self.positions = {}
        self.closed_positions = []
        self.position_id_counter = 0

    def add_position(self, position):
        position_id = position["id"]
        self.positions[position_id] = position
        return position_id

    def get_position(self, position_id):
        return self.positions.get(position_id)

    def get_open_positions(self, symbol=None):
        if symbol:
            return [p for p in self.positions.values() if p["symbol"] == symbol]
```

```
        return list(self.positions.values())

    def get_closed_positions(self, limit=100):
        return self.closed_positions[:limit]

    def close_position(self, position_id, close_data):
        if position_id in self.positions:
            position = self.positions.pop(position_id)
            position["close_time"] = datetime.now()
            position["close_data"] = close_data
            self.closed_positions.append(position)
            return True
        return False

    def generate_position_id(self):
        self.position_id_counter += 1
        return f"sim_{self.position_id_counter}"

# Initialiser les composants
mock_data_fetcher = MockDataFetcher(data)
mock_market_analyzer = MockMarketAnalyzer()
mock_position_tracker = MockPositionTracker()

# Initialiser le scoring engine
from ai.scoring_engine import ScoringEngine
scoring_engine = ScoringEngine()

# Créer la stratégie hybride
hybrid_strategy = HybridStrategy(
    mock_data_fetcher,
    mock_market_analyzer,
```

```
    scoring_engine,
    lstm_model,
    adaptive_risk_manager
)

# Simulation
equity_history = [initial_capital]
trades = []
open_positions = {}

# Parcourir les données jour par jour (à partir de l'indice 100 pour avoir assez d'historique)
window_size = lstm_model.input_length

for i in range(window_size, len(data) - 1):
    # Mettre à jour l'indice courant
    mock_data_fetcher.current_idx = i

    # Prix actuel et prochain
    current_price = data['close'].iloc[i]
    next_price = data['close'].iloc[i+1]

    # 1. Gérer les positions ouvertes
    positions_to_close = []

    for pos_id, position in open_positions.items():
        side = position["side"]
        entry_price = position["entry_price"]
        stop_loss = position["stop_loss"]
        take_profit = position["take_profit"]

        # Vérifier si le stop-loss ou take-profit est atteint au prochain pas de temps
```

```
        if side == "BUY":
            if next_price <= stop_loss:
                # Stop-loss atteint
                profit_pct = (stop_loss - entry_price) / entry_price * 100 * position["leverage"]
                positions_to_close.append((pos_id, profit_pct, "Stop-Loss"))
            elif next_price >= take_profit:
                # Take-profit atteint
                profit_pct = (take_profit - entry_price) / entry_price * 100 * position["leverage"]
                positions_to_close.append((pos_id, profit_pct, "Take-Profit"))
        else:  # SELL
            if next_price >= stop_loss:
                # Stop-loss atteint
                profit_pct = (entry_price - stop_loss) / entry_price * 100 * position["leverage"]
                positions_to_close.append((pos_id, profit_pct, "Stop-Loss"))
            elif next_price <= take_profit:
                # Take-profit atteint
                profit_pct = (entry_price - take_profit) / entry_price * 100 * position["leverage"]
                positions_to_close.append((pos_id, profit_pct, "Take-Profit"))

        # Vérification de fermeture anticipée basée sur les prédictions LSTM
        position_update = hybrid_strategy.should_close_early(symbol, position, current_price)

        if position_update["should_close"]:
            if side == "BUY":
                profit_pct = (next_price - entry_price) / entry_price * 100 * position["leverage"]
            else:
                profit_pct = (entry_price - next_price) / entry_price * 100 * position["leverage"]

            positions_to_close.append((pos_id, profit_pct, "Signal LSTM"))

    # Fermer les positions
```

```
for pos_id, profit_pct, reason in positions_to_close:
    position = open_positions.pop(pos_id)

    # Mettre à jour l'équité
    equity_change = equity_history[-1] * profit_pct / 100
    new_equity = equity_history[-1] + equity_change

    # Enregistrer le trade
    trade = {
        "day": i,
        "entry_day": position["entry_day"],
        "symbol": symbol,
        "side": position["side"],
        "entry_price": position["entry_price"],
        "exit_price": next_price,
        "profit_pct": profit_pct,
        "profit_amount": equity_change,
        "exit_reason": reason
    }
    trades.append(trade)

    # Mettre à jour le gestionnaire de risque
    adaptive_risk_manager.update_after_trade_closed({
        "pnl_absolute": equity_change,
        "pnl_percent": profit_pct
    })

# 2. Chercher de nouvelles opportunités de trading
if len(open_positions) < 3:  # Maximum 3 positions simultanées
    # Vérifier si une nouvelle position peut être ouverte
    risk_check = adaptive_risk_manager.can_open_new_position(mock_position_tracker)
```

```
        if risk_check["can_open"]:
            # Chercher une opportunité de trading
            opportunity = hybrid_strategy.find_trading_opportunity(symbol)

            if opportunity and opportunity["score"] >= hybrid_strategy.min_score:
                # Calculer la taille de position
                position_size = adaptive_risk_manager.calculate_position_size(
                    symbol,
                    opportunity,
                    opportunity.get("lstm_prediction")
                )

                # Simuler un nouveau trade
                entry_price = current_price
                stop_loss = opportunity["stop_loss"]
                take_profit = opportunity["take_profit"]
                side = opportunity["side"]

                # Levier (depuis le profil de risque)
                risk_profile =
adaptive_risk_manager.risk_levels[adaptive_risk_manager.current_risk_profile]
                leverage = risk_profile["leverage"]

                # Créer une nouvelle position
                position_id = mock_position_tracker.generate_position_id()
                position = {
                    "id": position_id,
                    "symbol": symbol,
                    "side": side,
                    "entry_price": entry_price,
```

```
                "stop_loss": stop_loss,
                "take_profit": take_profit,
                "entry_day": i,
                "score": opportunity["score"],
                "leverage": leverage
            }

            # Ajouter la position
            open_positions[position_id] = position

    # Mettre à jour l'équité si pas de changement
    if len(positions_to_close) == 0:
        equity_history.append(equity_history[-1])
    else:
        equity_history.append(new_equity)

# Calculer les statistiques finales
final_equity = equity_history[-1]
total_return = (final_equity - initial_capital) / initial_capital * 100

# Calculer le drawdown maximum
peak = initial_capital
max_drawdown = 0

for equity in equity_history:
    if equity > peak:
        peak = equity

    drawdown = (peak - equity) / peak * 100
    max_drawdown = max(max_drawdown, drawdown)
```

```
# Calculer le ratio de Sharpe
daily_returns = []

for i in range(1, len(equity_history)):
    daily_return = (equity_history[i] - equity_history[i-1]) / equity_history[i-1]
    daily_returns.append(daily_return)

if daily_returns:
    avg_return = sum(daily_returns) / len(daily_returns)
    std_return = np.std(daily_returns) if len(daily_returns) > 1 else 0

    annual_return = avg_return * 252
    annual_std = std_return * np.sqrt(252)

    sharpe_ratio = (annual_return - 0.01) / annual_std if annual_std > 0 else 0
else:
    sharpe_ratio = 0

# Calculer le taux de réussite
if trades:
    winning_trades = [t for t in trades if t["profit_pct"] > 0]
    win_rate = len(winning_trades) / len(trades) * 100
else:
    win_rate = 0

# Afficher les résultats
logger.info("\n=== Résultats de la simulation hybride ===")
logger.info(f"Capital initial: {initial_capital} USDT")
logger.info(f"Capital final: {final_equity:.2f} USDT")
logger.info(f"Rendement total: {total_return:.2f}%")
logger.info(f"Drawdown maximum: {max_drawdown:.2f}%")
```

```
logger.info(f"Ratio de Sharpe: {sharpe_ratio:.2f}")
logger.info(f"Nombre de trades: {len(trades)}")
logger.info(f"Taux de réussite: {win_rate:.2f}%")

# Générer un graphique de la courbe d'équité
plt.figure(figsize=(12, 6))
plt.plot(equity_history, linewidth=2)
plt.title(f"Courbe d'équité - Stratégie hybride LSTM - {symbol}")
plt.xlabel("Jours de trading")
plt.ylabel("Équité (USDT)")
plt.grid(True, alpha=0.3)

# Ajouter des annotations pour les trades
for trade in trades:
    day = trade["day"]
    if day < len(equity_history):
        equity = equity_history[day]

        if trade["profit_pct"] > 0:
            color = "green"
            marker = "^"
        else:
            color = "red"
            marker = "v"

        plt.plot(day, equity, marker=marker, color=color, markersize=8)

plt.savefig(os.path.join(DATA_DIR, "models", "evaluation", "figures",
                         f"{symbol}_hybrid_equity_curve.png"))
plt.close()
```

```
    # Sauvegarder les résultats
    simulation_results = {
        "symbol": symbol,
        "initial_capital": initial_capital,
        "final_equity": float(final_equity),
        "total_return": float(total_return),
        "max_drawdown": float(max_drawdown),
        "sharpe_ratio": float(sharpe_ratio),
        "trades": trades,
        "win_rate": float(win_rate),
        "equity_history": [float(eq) for eq in equity_history],
        "timestamp": datetime.now().isoformat()
    }

    results_file = os.path.join(DATA_DIR, "models", "evaluation",
                                f"{symbol}_hybrid_simulation_{datetime.now().strftime('%Y%m%d')}.json")

    with open(results_file, 'w') as f:
        json.dump(simulation_results, f, indent=2, default=str)

    logger.info(f"Résultats de la simulation hybride sauvegardés: {results_file}")

def evaluate_incremental_learning(args):
    """
    Évalue l'efficacité de l'apprentissage continu sur des données récentes

    Args:
        args: Arguments de ligne de commande
    """
    # Charger les données
    training_data = load_data(args.symbol, args.timeframe, args.train_start, args.train_end)
```

```
test_data = load_data(args.symbol, args.timeframe, args.test_start, args.test_end)

if training_data.empty or test_data.empty:
    logger.error("Données insuffisantes, impossible de continuer")
    return

# Charger ou créer un modèle LSTM
model_path = args.model_path or os.path.join(DATA_DIR, "models", "production", "lstm_final.h5")

if not os.path.exists(model_path) or args.retrain:
    logger.info("Création d'un nouveau modèle pour l'apprentissage continu...")

    # Paramètres du modèle
    model_params = {
        "input_length": args.sequence_length,
        "feature_dim": args.feature_dim,
        "lstm_units": [args.lstm_units, args.lstm_units // 2, args.lstm_units // 4],
        "dropout_rate": 0.3,
        "learning_rate": 0.001,
        "l1_reg": 0.0001,
        "l2_reg": 0.0001,
        "use_attention": True,
        "use_residual": True,
        "prediction_horizons": [12, 24, 96]
    }

    # Entraîner un modèle de base
    from ai.models.model_trainer import ModelTrainer
    trainer = ModelTrainer(model_params)

    # Préparer les données
```

```
    _, normalized_data = trainer.prepare_data(training_data)

    # Entraîner le modèle
    train_results = trainer.train_final_model(
        normalized_data,
        epochs=50,
        batch_size=32,
        test_ratio=0.15
    )

    logger.info("Modèle de base entraîné")
    model = trainer.model
else:
    logger.info(f"Chargement du modèle existant: {model_path}")
    model = LSTMModel()
    model.load(model_path)

# Initialiser le module d'apprentissage continu
continuous_learning = ContinuousLearning(
    model=model,
    feature_engineering=FeatureEngineering(),
    experience_buffer_size=5000,
    drift_threshold=0.15,
    drift_window_size=50
)

# Évaluer le modèle avant l'apprentissage continu
validator = ModelValidator(model, continuous_learning.feature_engineering)

logger.info("Évaluation du modèle avant l'apprentissage continu...")
pre_evaluation = validator.evaluate_on_test_set(test_data)
```

```
    # Diviser les données de test en mini-batches pour simuler des mises à jour progressives
    batch_size = args.batch_size
    num_batches = len(test_data) // batch_size

    logger.info(f"Simulation de l'apprentissage continu sur {num_batches} mini-batches...")
    update_history = []

    for i in range(num_batches):
        batch_start = i * batch_size
        batch_end = min((i + 1) * batch_size, len(test_data))

        batch = test_data.iloc[batch_start:batch_end]

        logger.info(f"Traitement du mini-batch {i+1}/{num_batches} ({len(batch)} échantillons)")

        # Traiter le mini-batch
        update_result = continuous_learning.process_new_data(batch,
min_samples=args.min_samples)

        # Enregistrer le résultat
        update_history.append({
            "batch": i,
            "updated": update_result.get("updated", False),
            "drift_detected": update_result.get("drift_detected", False),
            "evaluation": update_result.get("evaluation", {}),
            "timestamp": datetime.now().isoformat()
        })

    # Évaluer le modèle après l'apprentissage continu
    logger.info("Évaluation du modèle après l'apprentissage continu...")
```

```
    post_evaluation = validator.evaluate_on_test_set(test_data)

    # Calculer les améliorations
    improvements = {}

    for horizon_key, pre_metrics in pre_evaluation["horizons"].items():
        post_metrics = post_evaluation["horizons"].get(horizon_key, {})

        if "direction" in pre_metrics and "direction" in post_metrics:
            pre_acc = pre_metrics["direction"]["accuracy"]
            post_acc = post_metrics["direction"]["accuracy"]

            improvements[horizon_key] = {
                "accuracy_improvement": post_acc - pre_acc,
                "percent_improvement": (post_acc - pre_acc) / pre_acc * 100 if pre_acc > 0 else 0
            }

    # Afficher les résultats
    logger.info("\n=== Résultats de l'apprentissage continu ===")
    logger.info(f"Mini-batches traités: {num_batches}")
    logger.info(f"Mises à jour effectuées: {sum(1 for u in update_history if u['updated'])}")

    logger.info("\nPrécision de direction avant/après:")
    for horizon_key, improvement in improvements.items():
        pre_acc = pre_evaluation["horizons"][horizon_key]["direction"]["accuracy"]
        post_acc = post_evaluation["horizons"][horizon_key]["direction"]["accuracy"]

        logger.info(f"  {horizon_key}: {pre_acc:.4f} -> {post_acc:.4f}
({improvement['percent_improvement']:.2f}%)")

    # Sauvegarder les résultats
```

```
results = {
    "symbol": args.symbol,
    "timeframe": args.timeframe,
    "training_period": f"{args.train_start} to {args.train_end}",
    "testing_period": f"{args.test_start} to {args.test_end}",
    "pre_evaluation": pre_evaluation,
    "post_evaluation": post_evaluation,
    "improvements": improvements,
    "update_history": update_history,
    "timestamp": datetime.now().isoformat()
}

results_file = os.path.join(DATA_DIR, "models", "continuous_learning",
                            f"{args.symbol}_cl_results_{datetime.now().strftime('%Y%m%d')}.json")

os.makedirs(os.path.dirname(results_file), exist_ok=True)

with open(results_file, 'w') as f:
    json.dump(results, f, indent=2, default=str)

logger.info(f"Résultats de l'apprentissage continu sauvegardés: {results_file}")

# Générer des graphiques

# 1. Évolution de la précision au fil des mises à jour
plt.figure(figsize=(12, 6))

# Collecter les données pour chaque horizon
horizon_accuracies = {}

for i, update in enumerate(update_history):
```

```
            if "evaluation" in update and "horizons" in update["evaluation"]:
                for horizon_key, metrics in update["evaluation"]["horizons"].items():
                    if "direction" in metrics:
                        if horizon_key not in horizon_accuracies:
                            horizon_accuracies[horizon_key] = []

                        # Ajouter la précision
                        horizon_accuracies[horizon_key].append((i, metrics["direction"]["accuracy"]))

        # Tracer les courbes pour chaque horizon
        for horizon_key, accuracies in horizon_accuracies.items():
            if accuracies:
                x = [a[0] for a in accuracies]
                y = [a[1] for a in accuracies]

                plt.plot(x, y, label=horizon_key, linewidth=2, marker='o')

        plt.title("Évolution de la précision de direction pendant l'apprentissage continu")
        plt.xlabel("Mini-batch")
        plt.ylabel("Précision")
        plt.legend()
        plt.grid(True, alpha=0.3)

        plt.savefig(os.path.join(DATA_DIR, "models", "continuous_learning",
                                 f"{args.symbol}_cl_accuracy_evolution.png"))
        plt.close()

    def main():
        """Point d'entrée principal du script"""
        parser = argparse.ArgumentParser(description="Évaluation approfondie du modèle LSTM")
```

```
    subparsers = parser.add_subparsers(dest="command", help="Commande à exécuter")

    # Parser pour l'évaluation des prédictions de direction
    direction_parser = subparsers.add_parser("direction", help="Évaluer la précision de prédiction de direction")

    # Arguments pour les données
    direction_parser.add_argument("--symbol", type=str, default="BTCUSDT", help="Paire de trading")
    direction_parser.add_argument("--timeframe", type=str, default="15m", help="Intervalle de temps")
    direction_parser.add_argument("--start-date", type=str, required=True, help="Date de début (YYYY-MM-DD)")
    direction_parser.add_argument("--end-date", type=str, required=True, help="Date de fin (YYYY-MM-DD)")
    direction_parser.add_argument("--model-path", type=str, help="Chemin vers le modèle à évaluer")

    # Parser pour le backtest
    backtest_parser = subparsers.add_parser("backtest", help="Backtest complet de la stratégie")

    # Arguments pour les données
    backtest_parser.add_argument("--symbol", type=str, default="BTCUSDT", help="Paire de trading")
    backtest_parser.add_argument("--timeframe", type=str, default="15m", help="Intervalle de temps")
    backtest_parser.add_argument("--start-date", type=str, required=True, help="Date de début (YYYY-MM-DD)")
    backtest_parser.add_argument("--end-date", type=str, required=True, help="Date de fin (YYYY-MM-DD)")
    backtest_parser.add_argument("--model-path", type=str, help="Chemin vers le modèle à évaluer")
    backtest_parser.add_argument("--capital", type=float, default=200, help="Capital initial")
    backtest_parser.add_argument("--simulate-hybrid", action="store_true", help="Simuler la stratégie hybride complète")

    # Parser pour l'apprentissage continu
```

```
    cl_parser = subparsers.add_parser("continuous", help="Évaluer l'apprentissage continu")

    # Arguments pour les données d'entraînement
    cl_parser.add_argument("--symbol", type=str, default="BTCUSDT", help="Paire de trading")
    cl_parser.add_argument("--timeframe", type=str, default="15m", help="Intervalle de temps")
    cl_parser.add_argument("--train-start", type=str, required=True, help="Date de début d'entraînement (YYYY-MM-DD)")
    cl_parser.add_argument("--train-end", type=str, required=True, help="Date de fin d'entraînement (YYYY-MM-DD)")
    cl_parser.add_argument("--test-start", type=str, required=True, help="Date de début de test (YYYY-MM-DD)")
    cl_parser.add_argument("--test-end", type=str, required=True, help="Date de fin de test (YYYY-MM-DD)")

    # Arguments pour le modèle
    cl_parser.add_argument("--model-path", type=str, help="Chemin vers le modèle à utiliser")
    cl_parser.add_argument("--retrain", action="store_true", help="Ré-entraîner le modèle de base")
    cl_parser.add_argument("--sequence-length", type=int, default=60, help="Longueur des séquences d'entrée")
    cl_parser.add_argument("--feature-dim", type=int, default=30, help="Dimension des caractéristiques")
    cl_parser.add_argument("--lstm-units", type=int, default=128, help="Nombre d'unités LSTM")

    # Arguments pour l'apprentissage continu
    cl_parser.add_argument("--batch-size", type=int, default=100, help="Taille des mini-batches pour l'apprentissage continu")
    cl_parser.add_argument("--min-samples", type=int, default=30, help="Nombre minimum d'échantillons pour mettre à jour le modèle")

    args = parser.parse_args()

    if args.command == "direction":
        evaluate_direction_prediction(args)
    elif args.command == "backtest":
```

```
        backtest_trading_performance(args)
    elif args.command == "continuous":
        evaluate_incremental_learning(args)
    else:
        parser.print_help()

if __name__ == "__main__":
    main()
```

```
================================================
File: crypto_trading_bot_CLAUDE/install.py
================================================
```

```
# install.py
"""
Script d'installation et de configuration du bot de trading
"""
import os
import sys
import subprocess
import argparse
import json
import getpass

def check_python_version():
    """Vérifie la version de Python"""
    if sys.version_info < (3, 8):
        print("Erreur: Python 3.8 ou supérieur est requis")
        sys.exit(1)
    print(f"Python {sys.version} détecté")

def install_dependencies():
```

```
    """Installe les dépendances requises"""
    print("Installation des dépendances...")
    requirements = [
        "numpy>=1.20.0",
        "pandas>=1.3.0",
        "matplotlib>=3.4.0",
        "requests>=2.25.0",
        "python-dotenv>=0.19.0",
        "websocket-client>=1.2.0",
        "argparse>=1.4.0"
    ]

    with open("requirements.txt", "w") as f:
        f.write("\n".join(requirements))

    try:
        subprocess.check_call([sys.executable, "-m", "pip", "install", "-r", "requirements.txt"])
        print("Dépendances installées avec succès")
    except subprocess.CalledProcessError:
        print("Erreur lors de l'installation des dépendances")
        sys.exit(1)

def create_directories():
    """Crée les répertoires nécessaires"""
    directories = [
        "data",
        "data/market_data",
        "data/trade_logs",
        "data/performance",
        "logs"
    ]
```

```
    for directory in directories:
        os.makedirs(directory, exist_ok=True)

    print("Répertoires créés")

def setup_configuration():
    """Configure les paramètres du bot"""
    print("\n=== Configuration du bot de trading ===\n")

    use_testnet = input("Utiliser le réseau de test Binance? (O/n): ").lower() != "n"

    if use_testnet:
        print("\nVous allez utiliser le réseau de test Binance.")
        print("Rendez-vous sur https://testnet.binance.vision/ pour créer des clés API de test.")
    else:
        print("\nATTENTION: Vous allez utiliser le réseau de production Binance.")
        print("Le bot pourra trader avec de vrais fonds!")

    # Remplacer getpass.getpass par input standard
    print("\nNOTE: Vous allez entrer des informations sensibles. Assurez-vous que personne ne regarde votre écran.")
    api_key = input("Clé API Binance: ")
    api_secret = input("Clé secrète API Binance: ")

    # Créer le fichier .env
    with open(".env", "w") as f:
        f.write(f"BINANCE_API_KEY={api_key}\n")
        f.write(f"BINANCE_API_SECRET={api_secret}\n")
        f.write(f"USE_TESTNET={'True' if use_testnet else 'False'}\n")
```

```
    print("\nConfiguration sauvegardée dans le fichier .env")

    # Paramètres de trading personnalisés
    print("\n=== Paramètres de trading ===\n")
    print("Vous pouvez personnaliser les paramètres de trading ou utiliser les valeurs par défaut.")
    use_defaults = input("Utiliser les paramètres par défaut? (O/n): ").lower() != "n"

    if not use_defaults:
        try:
            risk_per_trade = float(input("Risque par trade (% du capital) [7.5]: ") or "7.5")
            stop_loss = float(input("Stop-loss (% du prix d'entrée) [4.0]: ") or "4.0")
            take_profit = float(input("Take-profit (% du prix d'entrée) [6.0]: ") or "6.0")
            leverage = int(input("Effet de levier [3]: ") or "3")

            # Créer un fichier de paramètres personnalisés
            params = {
                "RISK_PER_TRADE_PERCENT": risk_per_trade,
                "STOP_LOSS_PERCENT": stop_loss,
                "TAKE_PROFIT_PERCENT": take_profit,
                "LEVERAGE": leverage
            }

            with open("custom_params.json", "w") as f:
                json.dump(params, f, indent=2)

            print("\nParamètres personnalisés sauvegardés dans custom_params.json")
        except ValueError:
            print("Erreur: Valeur invalide. Utilisation des paramètres par défaut.")

def run_tests():
    """Exécute les tests unitaires"""
```

```
    print("\nExécution des tests unitaires...")

    try:
        import unittest

        if not os.path.exists("tests"):
            os.makedirs("tests")

        with open("tests/__init__.py", "w") as f:
            pass

        # Créer un test simple de connexion
        with open("tests/test_api_connection.py", "w") as f:
            f.write("""import unittest
import os
import sys

# Ajouter le répertoire parent au chemin de recherche
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

from core.api_connector import BinanceConnector

class TestAPIConnection(unittest.TestCase):
    def test_connection(self):
        connector = BinanceConnector()
        self.assertTrue(connector.test_connection())

if __name__ == "__main__":
    unittest.main()
""")
```

```
        # Exécuter le test
        print("Test de connexion à l'API Binance...")
        result = subprocess.run([sys.executable, "-m", "unittest", "tests.test_api_connection"], capture_output=True)

        if result.returncode == 0:
            print("Test de connexion réussi")
        else:
            print("Test de connexion échoué. Vérifiez vos clés API.")
            print(result.stderr.decode())

    except Exception as e:
        print(f"Erreur lors de l'exécution des tests: {str(e)}")

def main():
    """Fonction principale"""
    parser = argparse.ArgumentParser(description="Installation du bot de trading")
    parser.add_argument("--skip-deps", action="store_true", help="Ignorer l'installation des dépendances")
    parser.add_argument("--skip-config", action="store_true", help="Ignorer la configuration")
    parser.add_argument("--skip-tests", action="store_true", help="Ignorer les tests")

    args = parser.parse_args()

    print("=== Installation du Bot de Trading Crypto ===\n")

    # Vérifier la version de Python
    check_python_version()

    # Créer les répertoires
    create_directories()
```

```
    # Installer les dépendances
    if not args.skip_deps:
        install_dependencies()

    # Configurer le bot
    if not args.skip_config:
        setup_configuration()

    # Exécuter les tests
    if not args.skip_tests:
        run_tests()

    print("\nInstallation terminée!")
    print("\nPour lancer le bot en mode test sans trading réel:")
    print("  python main.py --dry-run")
    print("\nPour lancer le bot en mode production:")
    print("  python main.py")
    print("\nPour exécuter un backtest:")
    print("  python backtest.py --symbol BTCUSDT --start 2023-01-01 --end 2023-06-30")

if __name__ == "__main__":
    main()


================================================
File: crypto_trading_bot_CLAUDE/main.py
================================================
# main.py
"""
Point d'entrée principal du bot de trading crypto
"""
import logging
```

```
import time
from datetime import datetime
import signal
import sys
import requests

from config.config import LOG_LEVEL, LOG_FORMAT, LOG_FILE
from core.api_connector import BinanceConnector
from core.data_fetcher import MarketDataFetcher
from core.order_manager import OrderManager
from core.position_tracker import PositionTracker
from core.risk_manager import RiskManager
from strategies.technical_bounce import TechnicalBounceStrategy
from strategies.market_state import MarketStateAnalyzer
from ai.scoring_engine import ScoringEngine
from utils.logger import setup_logger

# Configuration du logger
logger = setup_logger("main", LOG_LEVEL, LOG_FORMAT, LOG_FILE)

class TradingBot:
    """
    Classe principale du bot de trading crypto
    """
    def __init__(self):
        logger.info("Initialisation du bot de trading...")

        # Initialisation des composants
        self.api = BinanceConnector()
        self.data_fetcher = MarketDataFetcher(self.api)
        self.risk_manager = RiskManager()
```

```
        self.position_tracker = PositionTracker()
        self.order_manager = OrderManager(self.api, self.position_tracker)

        # Initialisation des stratégies
        self.market_analyzer = MarketStateAnalyzer(self.data_fetcher)
        self.scoring_engine = ScoringEngine()
        self.strategy = TechnicalBounceStrategy(
            self.data_fetcher,
            self.market_analyzer,
            self.scoring_engine
        )

        # Variables d'état
        self.is_running = False
        self.last_trade_time = {}  # Pour suivre le temps entre les trades

        # Configuration des gestionnaires de signaux
        signal.signal(signal.SIGINT, self.handle_shutdown)
        signal.signal(signal.SIGTERM, self.handle_shutdown)

        logger.info("Bot initialisé avec succès")

    def start(self):
        """
        Démarre le bot de trading
        """
        self.is_running = True
        logger.info("Démarrage du bot de trading...")

        try:
            # Validation de la connexion à l'API
```

```
        if not self.api.test_connection():
            logger.error("Échec de la connexion à l'API Binance. Arrêt du bot.")
            return

        logger.info("Connexion à l'API Binance réussie.")
        account_info = self.api.get_account_info()
        self.risk_manager.update_account_balance(account_info)

        # Boucle principale
        while self.is_running:
            self.trading_cycle()
            time.sleep(15)  # Attente de 15 secondes entre chaque cycle

    except Exception as e:
        logger.error(f"Erreur critique lors de l'exécution: {str(e)}")
        self.shutdown()

def trading_cycle(self):
    """
    Cycle principal de trading avec gestion d'erreurs améliorée
    """
    from config.config import TRADING_PAIRS

    try:
        current_time = datetime.now()

        for pair in TRADING_PAIRS:
            try:
                # Vérification du cooldown entre trades
                if pair in self.last_trade_time:
                    from config.trading_params import MIN_TIME_BETWEEN_TRADES
```

```
                time_since_last_trade = (current_time - self.last_trade_time[pair]).total_seconds() / 60
                if time_since_last_trade < MIN_TIME_BETWEEN_TRADES:
                    logger.debug(f"Cooldown actif pour {pair}:
{time_since_last_trade:.1f}/{MIN_TIME_BETWEEN_TRADES} minutes écoulées")
                    continue

            # Analyse de l'état du marché
            market_state = self.market_analyzer.analyze_market_state(pair)
            if not market_state["favorable"]:
                logger.info(f"Marché défavorable pour {pair}: {market_state['reason']}")
                continue

            # Vérification des conditions de risque
            if not self.risk_manager.can_open_new_position(self.position_tracker):
                logger.info(f"Conditions de risque non remplies pour {pair}")
                continue

            # Recherche d'opportunités de trading
            opportunity = self.strategy.find_trading_opportunity(pair)
            if opportunity and opportunity["score"] >= self.strategy.min_score:
                logger.info(f"Opportunité trouvée pour {pair} (score: {opportunity['score']})")

                # Calculer le montant à trader
                trade_amount = self.risk_manager.calculate_position_size(pair, opportunity)

                if trade_amount > 0:
                    # Exécution de l'ordre
                    order_result = self.order_manager.place_entry_order(
                        pair,
                        opportunity["side"],
                        trade_amount,
```

```
                opportunity["entry_price"],
                opportunity["stop_loss"],
                opportunity["take_profit"]
            )

            if order_result["success"]:
                self.last_trade_time[pair] = current_time
                logger.info(f"Trade exécuté sur {pair}: {order_result}")

                # Enregistrement des données du trade pour analyse
                self.strategy.log_trade(opportunity, order_result)

                # Notification de l'ordre placé
                self._send_notification(
                    f"Ordre placé: {pair} {opportunity['side']} à {order_result['entry_price']} " +
                    f"(SL: {order_result['stop_loss_price']}, TP: {order_result['take_profit_price']})"
                )
            else:
                logger.error(f"Échec de l'ordre pour {pair}: {order_result['message']}")

        # Gestion des positions ouvertes
        self.manage_open_positions(pair)

    except requests.exceptions.RequestException as e:
        logger.error(f"Erreur réseau pour {pair}: {str(e)}")
        time.sleep(5)  # Attendre 5 secondes avant de continuer
        continue
    except Exception as e:
        logger.error(f"Erreur lors du traitement de {pair}: {str(e)}")
        continue
```

```
        except requests.exceptions.RequestException as e:
            logger.error(f"Erreur réseau lors de la communication avec Binance: {str(e)}")
            # Attendre et réessayer au prochain cycle
            time.sleep(30)
            return
        except Exception as e:
            logger.critical(f"Erreur critique dans le cycle de trading: {str(e)}")
            if self._is_critical_error(str(e)):
                self._send_emergency_notification(f"Erreur critique: {str(e)}")
                self.shutdown()

    def manage_open_positions(self, pair):
        """
        Gestion des positions ouvertes (trailing stops, etc.)
        """
        open_positions = self.position_tracker.get_open_positions(pair)

        for position in open_positions:
            # Mise à jour des données de marché
            current_price = self.data_fetcher.get_current_price(pair)

            # Mise à jour des trailing stops si nécessaire
            self.order_manager.update_trailing_stop(pair, position, current_price)

    def handle_shutdown(self, signum, frame):
        """
        Gestionnaire de signal pour arrêt propre
        """
        logger.info("Signal d'arrêt reçu. Arrêt en cours...")
        self.shutdown()
```

```
def shutdown(self):
    """
    Arrêt propre du bot
    """
    self.is_running = False

    # Fermeture propre des positions si nécessaire
    # (Peut être commenté pour conserver les positions ouvertes)
    #self.close_all_positions()

    logger.info("Bot arrêté avec succès")
    sys.exit(0)

def close_all_positions(self):
    """
    Ferme toutes les positions ouvertes
    """
    all_positions = self.position_tracker.get_all_open_positions()

    for pair, positions in all_positions.items():
        for position in positions:
            self.order_manager.close_position(pair, position["id"])

    logger.info("Toutes les positions ont été fermées")
def _is_critical_error(self, error_message: str) -> bool:
    """
    Détermine si une erreur est critique et nécessite un arrêt

    Args:
        error_message: Message d'erreur
```

```
    Returns:
        True si l'erreur est critique, False sinon
    """
    critical_keywords = [
        "Authentication failed", "API key expired", "IP has been banned",
        "Account has been frozen", "Insufficient balance", "System error",
        "Fatal error", "Database corruption"
    ]

    return any(keyword in error_message for keyword in critical_keywords)

def _send_notification(self, message: str, level: str = "info") -> None:
    """
    Envoie une notification

    Args:
        message: Message à envoyer
        level: Niveau de la notification (info, warning, critical)
    """
    logger.info(f"Notification ({level}): {message}")

    # Si les notifications sont activées dans la configuration
    if hasattr(self, 'notification_service'):
        self.notification_service.send(message, level)

def _send_emergency_notification(self, message: str) -> None:
    """
    Envoie une notification d'urgence

    Args:
        message: Message d'urgence
```

```
        """
        self._send_notification(message, "critical")

        # Tentative d'envoi par tous les canaux disponibles
        from config.config import NOTIFICATION_EMAIL, ENABLE_NOTIFICATIONS

        if ENABLE_NOTIFICATIONS and NOTIFICATION_EMAIL:
            try:
                import smtplib
                from email.mime.text import MIMEText
                from config.config import SMTP_SERVER, SMTP_PORT, SMTP_USER, SMTP_PASSWORD

                msg = MIMEText(f"URGENCE - BOT DE TRADING: {message}")
                msg['Subject'] = "ALERTE CRITIQUE - Bot de Trading"
                msg['From'] = SMTP_USER
                msg['To'] = NOTIFICATION_EMAIL

                with smtplib.SMTP(SMTP_SERVER, SMTP_PORT) as server:
                    server.starttls()
                    server.login(SMTP_USER, SMTP_PASSWORD)
                    server.send_message(msg)
            except Exception as e:
                logger.error(f"Impossible d'envoyer l'email d'urgence: {str(e)}")

if __name__ == "__main__":
    bot = TradingBot()
    bot.start()

================================================
File: crypto_trading_bot_CLAUDE/requirements.txt
================================================
```

numpy>=1.20.0

pandas>=1.3.0

matplotlib>=3.4.0

requests>=2.25.0

python-dotenv>=0.19.0

websocket-client>=1.2.0

argparse>=1.4.0

==========

File: crypto_trading_bot_CLAUDE/structure globale.txt

==========

/crypto_trading_bot/

|

├── config/ # Configuration du bot

| ├── __init__.py

| ├── config.py # Paramètres globaux de configuration

| ├── trading_params.py # Paramètres de trading (ajustables)

| └── model_params.py # Nouveaux paramètres pour les modèles LSTM

|

├── core/ # Fonctionalités principales

| ├── __init__.py

| ├── api_connector.py # Connexion aux API d'échange (Binance)

| ├── data_fetcher.py # Récupération des données de marché

| ├── order_manager.py # Gestion des ordres (entrée, sortie)

| ├── position_tracker.py # Suivi des positions ouvertes

| ├── risk_manager.py # Gestion des risques et capital

| └── adaptive_risk_manager.py # NOUVEAU: Gestion des risques adaptative par IA

|

├── strategies/ # Stratégies de trading

| ├── __init__.py

| ├── technical_bounce.py # Stratégie de rebond technique

```
|   ├── strategy_base.py        # Classe de base pour stratégies
|   ├── market_state.py         # Détection de l'état du marché
|   └── hybrid_strategy.py      # NOUVEAU: Stratégie hybride avec LSTM
|
├── indicators/               # Indicateurs techniques
|   ├── __init__.py
|   ├── trend.py              # Indicateurs de tendance (EMA, ADX)
|   ├── momentum.py           # Indicateurs de momentum (RSI)
|   ├── volatility.py         # Indicateurs de volatilité (BB, ATR)
|   ├── volume.py             # Analyse de volume
|   └── advanced_features.py    # NOUVEAU: Caractéristiques avancées pour LSTM
|
├── ai/                       # Composants d'IA
|   ├── __init__.py
|   ├── scoring_engine.py       # Moteur de scoring des opportunités
|   ├── trade_analyzer.py       # Analyse post-trade
|   ├── parameter_optimizer.py  # Optimisation des paramètres
|   ├── reasoning_engine.py     # Génération d'explications textuelles
|   └── models/               # NOUVEAU: Sous-dossier pour les modèles avancés
|       ├── __init__.py
|       ├── lstm_model.py       # Implémentation du modèle LSTM principal
|       ├── attention.py        # Mécanismes d'attention
|       ├── feature_engineering.py # Préparation des caractéristiques
|       ├── model_trainer.py    # Entraînement des modèles
|       ├── model_validator.py  # Validation et évaluation
|       ├── ensemble.py         # Ensemble de modèles pour robustesse
|       └── continuous_learning.py # Apprentissage continu
|
├── utils/                    # Utilitaires
|   ├── __init__.py
|   ├── logger.py             # Système de journalisation
```

```
|   ├── visualizer.py          # Visualisation des trades et performances
|   ├── backtest_engine.py     # Moteur de backtest
|   ├── model_backtester.py    # NOUVEAU: Backtest spécifique pour modèles
|   ├── model_monitor.py       # NOUVEAU: Monitoring des performances modèles
|   └── model_explainer.py     # NOUVEAU: Explicabilité des décisions modèles
|
├── data/                    # Stockage des données
|   ├── market_data/           # Données de marché historiques
|   ├── trade_logs/            # Journal des trades
|   ├── performance/           # Données de performance
|   ├── model_data/            # NOUVEAU: Données prétraitées pour modèles
|   |   ├── training/          # Données d'entraînement
|   |   ├── validation/        # Données de validation
|   |   └── features/          # Caractéristiques précalculées
|   └── models/                # NOUVEAU: Stockage des modèles entraînés
|       ├── checkpoints/       # Points de sauvegarde
|       ├── production/        # Modèles en production
|       └── archive/           # Versions précédentes des modèles
|
├── dashboard/               # NOUVEAU: Tableaux de bord interactifs
|   ├── __init__.py
|   ├── app.py                 # Application de tableau de bord
|   ├── model_dashboard.py     # Visualisation des performances modèles
|   └── trade_dashboard.py     # Visualisation des performances trading
|
├── tests/                   # Tests unitaires et d'intégration
|   ├── __init__.py
|   ├── test_indicators.py
|   ├── test_strategies.py
|   ├── test_risk_manager.py
|   └── test_models/           # NOUVEAU: Tests des modèles LSTM
```

```
|   ├── __init__.py
|   ├── test_lstm.py
|   ├── test_feature_eng.py
|   └── test_backtesting.py
|
├── main.py                 # Point d'entrée principal
├── backtest.py             # Script de backtest
├── train_model.py          # NOUVEAU: Script d'entraînement des modèles
├── evaluate_model.py       # NOUVEAU: Script d'évaluation des modèles
├── download_data.py        # Script de téléchargement de données historiques
├── install.py              # Script d'installation et configuration
├── requirements.txt        # Dépendances
└── README.md               # Documentation
```

==========

File: crypto_trading_bot_CLAUDE/train_model.py

==========

```
#!/usr/bin/env python
# train_model.py
"""
Script d'entraînement du modèle LSTM pour la prédiction des mouvements de marché
"""
import os
import argparse
import pandas as pd
import numpy as np
from datetime import datetime, timedelta
import json
import matplotlib.pyplot as plt

from ai.models.lstm_model import LSTMModel
```

```
from ai.models.feature_engineering import FeatureEngineering
from ai.models.model_trainer import ModelTrainer
from ai.models.model_validator import ModelValidator
from config.config import DATA_DIR
from utils.logger import setup_logger

logger = setup_logger("train_model")

def load_data(symbol: str, timeframe: str, start_date: str, end_date: str) -> pd.DataFrame:
    """
    Charge les données OHLCV depuis le disque

    Args:
        symbol: Paire de trading
        timeframe: Intervalle de temps
        start_date: Date de début (YYYY-MM-DD)
        end_date: Date de fin (YYYY-MM-DD)

    Returns:
        DataFrame avec les données OHLCV
    """
    # Construire le chemin du fichier
    data_path = os.path.join(DATA_DIR, "market_data",
f"{symbol}_{timeframe}_{start_date}_{end_date}.csv")

    # Vérifier si le fichier existe
    if not os.path.exists(data_path):
        logger.error(f"Fichier non trouvé: {data_path}")
        return pd.DataFrame()

    # Charger les données
```

```
    try:
        data = pd.read_csv(data_path)

        # Convertir la colonne timestamp en datetime
        if "timestamp" in data.columns:
            data["timestamp"] = pd.to_datetime(data["timestamp"])
            data.set_index("timestamp", inplace=True)

        logger.info(f"Données chargées: {len(data)} lignes")
        return data
    except Exception as e:
        logger.error(f"Erreur lors du chargement des données: {str(e)}")
        return pd.DataFrame()

def download_data_if_needed(symbol: str, timeframe: str, start_date: str, end_date: str) -> bool:
    """
    Télécharge les données si elles n'existent pas sur le disque

    Args:
        symbol: Paire de trading
        timeframe: Intervalle de temps
        start_date: Date de début (YYYY-MM-DD)
        end_date: Date de fin (YYYY-MM-DD)

    Returns:
        True si les données sont disponibles, False sinon
    """
    # Construire le chemin du fichier
    data_path = os.path.join(DATA_DIR, "market_data",
f"{symbol}_{timeframe}_{start_date}_{end_date}.csv")
```

```
    # Vérifier si le fichier existe
    if os.path.exists(data_path):
        logger.info(f"Données déjà disponibles: {data_path}")
        return True

    # Télécharger les données
    logger.info(f"Téléchargement des données pour {symbol} ({timeframe}) du {start_date} au {end_date}")

    try:
        from download_data import download_binance_data

        # Télécharger les données
        df = download_binance_data(symbol, timeframe, start_date, end_date, data_path)

        if df is not None and not df.empty:
            logger.info(f"Données téléchargées avec succès: {len(df)} lignes")
            return True
        else:
            logger.error("Échec du téléchargement des données")
            return False
    except Exception as e:
        logger.error(f"Erreur lors du téléchargement des données: {str(e)}")
        return False

def train_lstm_model(args):
    """
    Entraîne le modèle LSTM avec les paramètres spécifiés

    Args:
        args: Arguments de ligne de commande
```

```
"""
# Télécharger les données si nécessaire
if not download_data_if_needed(args.symbol, args.timeframe, args.start_date, args.end_date):
    logger.error("Impossible de continuer sans données")
    return

# Charger les données
data = load_data(args.symbol, args.timeframe, args.start_date, args.end_date)

if data.empty:
    logger.error("Données vides, impossible de continuer")
    return

# Configurer les paramètres du modèle
model_params = {
    "input_length": args.sequence_length,
    "feature_dim": args.feature_dim,
    "lstm_units": [args.lstm_units, args.lstm_units // 2, args.lstm_units // 4],
    "dropout_rate": args.dropout,
    "learning_rate": args.learning_rate,
    "l1_reg": args.l1_reg,
    "l2_reg": args.l2_reg,
    "use_attention": not args.no_attention,
    "use_residual": not args.no_residual,
    "prediction_horizons": [args.short_horizon, args.mid_horizon, args.long_horizon]
}

logger.info(f"Configuration du modèle: {json.dumps(model_params, indent=2)}")

# Créer le ModelTrainer
trainer = ModelTrainer(model_params)
```

```
    # Préparer les données
    logger.info("Préparation des données...")
    featured_data, normalized_data = trainer.prepare_data(data)

    # Diviser les données en ensembles d'entraînement, validation et test
    if args.cv:
        # Entraînement avec validation croisée
        logger.info("Entraînement avec validation croisée temporelle...")
        cv_results = trainer.train_with_cv(
            normalized_data,
            n_splits=args.cv_splits,
            epochs=args.epochs,
            batch_size=args.batch_size,
            initial_train_ratio=args.train_ratio,
            patience=args.patience
        )

        logger.info(f"Entraînement terminé, perte moyenne de validation:
{cv_results['avg_val_loss']:.4f}")
    else:
        # Entraînement simple avec division train/val/test
        logger.info("Entraînement standard avec division temporelle...")
        train_data, val_data, test_data = trainer.temporal_train_test_split(
            normalized_data,
            train_ratio=args.train_ratio,
            val_ratio=args.val_ratio
        )

        # Créer et entraîner le modèle
        train_results = trainer.train_final_model(
```

```
            normalized_data,
            epochs=args.epochs,
            batch_size=args.batch_size,
            test_ratio=1.0 - args.train_ratio - args.val_ratio
        )

        logger.info(f"Entraînement terminé, perte sur le test: {train_results['test_loss']:.4f}")

        # Afficher les précisions de direction par horizon
        for i, horizon in enumerate(model_params["prediction_horizons"]):
            accuracy = train_results["direction_accuracies"][i]
            logger.info(f"Précision de direction pour horizon {horizon}: {accuracy:.2f}")

    # Valider le modèle final sur des données récentes
    if args.validate:
        logger.info("Validation du modèle sur des données récentes...")

        # Charger des données récentes pour la validation
        end_date = datetime.now().strftime("%Y-%m-%d")
        start_date = (datetime.now() - timedelta(days=30)).strftime("%Y-%m-%d")

        if download_data_if_needed(args.symbol, args.timeframe, start_date, end_date):
            validation_data = load_data(args.symbol, args.timeframe, start_date, end_date)

            if not validation_data.empty:
                # Créer le validateur
                validator = ModelValidator(trainer.model, trainer.feature_engineering)

                # Évaluer sur les données récentes
                validation_results = validator.evaluate_on_test_set(validation_data)
```

```
            logger.info(f"Validation terminée, perte: {validation_results['loss']:.4f}")

            # Afficher les métriques par horizon
            for horizon_key, metrics in validation_results["horizons"].items():
                direction_acc = metrics["direction"]["accuracy"]
                direction_f1 = metrics["direction"]["f1_score"]

                logger.info(f"{horizon_key}: Accuracy={direction_acc:.2f}, F1={direction_f1:.2f}")

    logger.info("Processus d'entraînement terminé")
    logger.info(f"Modèle final sauvegardé: {os.path.join(DATA_DIR, 'models', 'production',
'lstm_final.h5')}")


def evaluate_model(args):
    """
    Évalue un modèle LSTM existant sur de nouvelles données

    Args:
        args: Arguments de ligne de commande
    """
    # Télécharger les données de test si nécessaire
    if not download_data_if_needed(args.symbol, args.timeframe, args.start_date, args.end_date):
        logger.error("Impossible de continuer sans données")
        return

    # Charger les données
    data = load_data(args.symbol, args.timeframe, args.start_date, args.end_date)

    if data.empty:
        logger.error("Données vides, impossible de continuer")
        return
```

```
# Charger le modèle existant
model_path = args.model_path or os.path.join(DATA_DIR, "models", "production", "lstm_final.h5")

if not os.path.exists(model_path):
    logger.error(f"Modèle non trouvé: {model_path}")
    return

# Créer le validateur
validator = ModelValidator()
validator.load_model(model_path)

# Évaluer le modèle
evaluation = validator.evaluate_on_test_set(data)

logger.info(f"Évaluation terminée, perte globale: {evaluation['loss']:.4f}")

# Afficher les métriques par horizon
for horizon_key, metrics in evaluation["horizons"].items():
    direction_metrics = metrics["direction"]
    volatility_metrics = metrics["volatility"]

    logger.info(f"\nHorizon: {horizon_key}")
    logger.info(f"Direction: Accuracy={direction_metrics['accuracy']:.2f}, "
                f"Precision={direction_metrics['precision']:.2f}, "
                f"Recall={direction_metrics['recall']:.2f}, "
                f"F1={direction_metrics['f1_score']:.2f}")
    logger.info(f"Volatilité: MAE={volatility_metrics['mae']:.4f}, "
                f"RMSE={volatility_metrics['rmse']:.4f}")

# Comparaison avec la stratégie de base
```

```
    if args.compare:
        logger.info("\nComparaison avec la stratégie de base...")

        comparison = validator.compare_with_baseline(
            data,
            initial_capital=args.capital
        )

        # Afficher les résultats
        baseline = comparison["baseline"]
        lstm = comparison["lstm"]
        diff = comparison["comparison"]

        logger.info("\n=== Résultats de la comparaison ===")
        logger.info(f"Stratégie de base: {baseline['return_pct']:.2f}% (Drawdown: {baseline['max_drawdown_pct']:.2f}%, Sharpe: {baseline['sharpe_ratio']:.2f})")
        logger.info(f"Modèle LSTM: {lstm['return_pct']:.2f}% (Drawdown: {lstm['max_drawdown_pct']:.2f}%, Sharpe: {lstm['sharpe_ratio']:.2f})")
        logger.info(f"Différence: {diff['return_difference']:.2f}%, Amélioration drawdown: {diff['drawdown_improvement']:.2f}%, Amélioration Sharpe: {diff['sharpe_improvement']:.2f}")

        # Sauvegarder les résultats
        comparison_file = os.path.join(DATA_DIR, "models", "evaluation", f"comparison_{args.symbol}_{datetime.now().strftime('%Y%m%d')}.json")
        os.makedirs(os.path.dirname(comparison_file), exist_ok=True)

        with open(comparison_file, 'w') as f:
            json.dump(comparison, f, indent=2, default=str)

        logger.info(f"Résultats de comparaison sauvegardés: {comparison_file}")

def main():
```

```
    """Point d'entrée principal du script"""

    parser = argparse.ArgumentParser(description="Entraînement et évaluation du modèle LSTM")


    subparsers = parser.add_subparsers(dest="command", help="Commande à exécuter")


    # Parser pour l'entraînement

    train_parser = subparsers.add_parser("train", help="Entraîner un nouveau modèle LSTM")


    # Arguments pour les données

    train_parser.add_argument("--symbol", type=str, default="BTCUSDT", help="Paire de trading")

    train_parser.add_argument("--timeframe", type=str, default="15m", help="Intervalle de temps")

    train_parser.add_argument("--start-date", type=str, required=True, help="Date de début (YYYY-MM-DD)")

    train_parser.add_argument("--end-date", type=str, required=True, help="Date de fin (YYYY-MM-DD)")


    # Arguments pour le modèle

    train_parser.add_argument("--sequence-length", type=int, default=60, help="Longueur des séquences d'entrée")

    train_parser.add_argument("--feature-dim", type=int, default=30, help="Dimension des caractéristiques")

    train_parser.add_argument("--lstm-units", type=int, default=128, help="Nombre d'unités LSTM")

    train_parser.add_argument("--dropout", type=float, default=0.3, help="Taux de dropout")

    train_parser.add_argument("--learning-rate", type=float, default=0.001, help="Taux d'apprentissage")

    train_parser.add_argument("--l1-reg", type=float, default=0.0001, help="Régularisation L1")

    train_parser.add_argument("--l2-reg", type=float, default=0.0001, help="Régularisation L2")

    train_parser.add_argument("--no-attention", action="store_true", help="Désactiver le mécanisme d'attention")

    train_parser.add_argument("--no-residual", action="store_true", help="Désactiver les connexions résiduelles")


    # Horizons de prédiction
```

```
    train_parser.add_argument("--short-horizon", type=int, default=12, help="Horizon court terme (nb de périodes)")
    train_parser.add_argument("--mid-horizon", type=int, default=24, help="Horizon moyen terme (nb de périodes)")
    train_parser.add_argument("--long-horizon", type=int, default=96, help="Horizon long terme (nb de périodes)")

    # Arguments pour l'entraînement
    train_parser.add_argument("--epochs", type=int, default=100, help="Nombre d'époques")
    train_parser.add_argument("--batch-size", type=int, default=32, help="Taille du batch")
    train_parser.add_argument("--patience", type=int, default=20, help="Patience pour l'early stopping")
    train_parser.add_argument("--train-ratio", type=float, default=0.7, help="Ratio des données d'entraînement")
    train_parser.add_argument("--val-ratio", type=float, default=0.15, help="Ratio des données de validation")

    # Validation croisée
    train_parser.add_argument("--cv", action="store_true", help="Utiliser la validation croisée temporelle")
    train_parser.add_argument("--cv-splits", type=int, default=5, help="Nombre de plis pour la validation croisée")

    # Validation finale
    train_parser.add_argument("--validate", action="store_true", help="Valider sur des données récentes après l'entraînement")

    # Parser pour l'évaluation
    eval_parser = subparsers.add_parser("evaluate", help="Évaluer un modèle LSTM existant")

    # Arguments pour les données
    eval_parser.add_argument("--symbol", type=str, default="BTCUSDT", help="Paire de trading")
    eval_parser.add_argument("--timeframe", type=str, default="15m", help="Intervalle de temps")
    eval_parser.add_argument("--start-date", type=str, required=True, help="Date de début (YYYY-MM-DD)")
```

```
    eval_parser.add_argument("--end-date", type=str, required=True, help="Date de fin (YYYY-MM-DD)")

    # Arguments pour le modèle
    eval_parser.add_argument("--model-path", type=str, help="Chemin vers le modèle à évaluer")

    # Comparaison avec stratégie de base
    eval_parser.add_argument("--compare", action="store_true", help="Comparer avec la stratégie de base")
    eval_parser.add_argument("--capital", type=float, default=200, help="Capital initial pour la comparaison")

    args = parser.parse_args()

    if args.command == "train":
        train_lstm_model(args)
    elif args.command == "evaluate":
        evaluate_model(args)
    else:
        parser.print_help()

if __name__ == "__main__":
    main()
```

```
================================================
File: crypto_trading_bot_CLAUDE/.env
================================================
BINANCE_API_KEY=u6cP7KVlRmHLTC4RnGD0jkDZzgEkyK4nXVfIwlxQoM1j9HZZPUu8Vkrbk6ymfIlD
BINANCE_API_SECRET=P5v5e3Zw24ACZVEnM35NuX3q98ZX29b3tfVHkyzhuEjtvITfCnZUFMKExm8gV2c
USE_TESTNET=True
```

```
================================================
File: crypto_trading_bot_CLAUDE/ai/market_anomaly_detector.py
================================================
"""
Module avancé pour la détection d'anomalies et d'événements extrêmes (black swan) sur les marchés financiers
Intègre plusieurs approches statistiques et algorithmiques pour identifier les conditions de marché anormales
"""
import numpy as np
import pandas as pd
from typing import Dict, List, Union, Optional
from scipy import stats
from statsmodels.tsa.stattools import adfuller
import tensorflow as tf
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import Dense, LSTM, Dropout, Input
import warnings
import os
import pickle
from datetime import datetime, timedelta

from utils.logger import setup_logger

logger = setup_logger("market_anomaly_detector")

class MarketAnomalyDetector:
    """
    Détecteur d'anomalies et d'événements extrêmes pour les marchés financiers
    Utilise plusieurs méthodes complémentaires:
    1. Tests statistiques (détection d'outliers, fat tails, rupture de stationnarité)
```

```
2. Analyse des microstructures de marché (rupture de l'algorithme de matching, flash crashes)
3. Détection d'anomalies basée sur l'apprentissage automatique (isolation forest, autoencoder)
"""
def __init__(self,
             lookback_period: int = 100,
             confidence_level: float = 0.99,
             volatility_threshold: float = 3.0,
             volume_threshold: float = 5.0,
             price_gap_threshold: float = 3.0,
             use_ml_models: bool = True,
             model_dir: Optional[str] = None):
    """
    Initialise le détecteur d'anomalies

    Args:
        lookback_period: Période d'historique pour les calculs statistiques
        confidence_level: Niveau de confiance pour la détection des anomalies
        volatility_threshold: Seuil de multiplication d'ATR pour la volatilité extrême
        volume_threshold: Seuil de multiplication du volume moyen pour volume extrême
        price_gap_threshold: Seuil de multiplication de l'ATR pour les gaps de prix
        use_ml_models: Utiliser des modèles de ML pour la détection d'anomalies
        model_dir: Répertoire pour sauvegarder/charger les modèles
    """
    self.lookback_period = lookback_period
    self.confidence_level = confidence_level
    self.volatility_threshold = volatility_threshold
    self.volume_threshold = volume_threshold
    self.price_gap_threshold = price_gap_threshold
    self.use_ml_models = use_ml_models
    self.model_dir = model_dir
```

```
        # Historique des anomalies détectées
        self.anomaly_history = []

        # Initialisation des modèles de ML pour la détection d'anomalies
        self.isolation_forest = None
        self.autoencoder = None

        # Charger les modèles si disponibles
        if use_ml_models and model_dir:
            self._load_models()

    def detect_anomalies(self, data: pd.DataFrame, current_price: float = None,
                         return_details: bool = False) -> Union[Dict, bool]:
        """
        Détecte les anomalies dans les données de marché fournies

        Args:
            data: DataFrame avec les données OHLCV
            current_price: Prix actuel (si différent du dernier prix dans data)
            return_details: Retourner les détails de l'analyse

        Returns:
            True/False ou dictionnaire détaillé si anomalie détectée
        """
        if len(data) < self.lookback_period:
            logger.warning(f"Données insuffisantes pour la détection d'anomalies: {len(data)} <
{self.lookback_period}")
            return False if not return_details else {"detected": False, "reason": "Données insuffisantes"}

        # Utiliser les données récentes pour l'analyse
        recent_data = data.tail(self.lookback_period).copy()
```

```
# Mise à jour du prix actuel si fourni
if current_price is not None:
    current_close = current_price
else:
    current_close = recent_data['close'].iloc[-1]

# Résultats de détection pour différentes méthodes
results = {}

# 1. Vérifier la volatilité extrême
volatility_anomaly = self._detect_volatility_anomaly(recent_data, current_close)
results["volatility_anomaly"] = volatility_anomaly

# 2. Vérifier les gaps de prix significatifs
price_gap_anomaly = self._detect_price_gap(recent_data, current_close)
results["price_gap_anomaly"] = price_gap_anomaly

# 3. Vérifier le volume anormal
volume_anomaly = self._detect_volume_anomaly(recent_data)
results["volume_anomaly"] = volume_anomaly

# 4. Vérifier les fat tails dans la distribution des rendements
fat_tails = self._detect_fat_tails(recent_data)
results["fat_tails"] = fat_tails

# 5. Vérifier la rupture de stationnarité
stationarity_break = self._detect_stationarity_break(recent_data)
results["stationarity_break"] = stationarity_break

# 6. Vérifier les anomalies de microstructure
```

```
        microstructure_anomaly = self._detect_microstructure_anomaly(recent_data, current_close)
        results["microstructure_anomaly"] = microstructure_anomaly

        # 7. Utiliser les modèles de ML pour la détection d'anomalies
        ml_anomaly = self._detect_ml_anomalies(recent_data) if self.use_ml_models else False
        results["ml_anomaly"] = ml_anomaly

        # 8. Vérifier le momentum extrême
        momentum_anomaly = self._detect_momentum_anomaly(recent_data)
        results["momentum_anomaly"] = momentum_anomaly

        # Combinaison des résultats
        # Une anomalie est détectée si au moins deux méthodes différentes signalent une anomalie
        anomaly_count = sum(1 for result in results.values() if result["detected"])
        anomaly_detected = anomaly_count >= 2  # Au moins deux méthodes doivent détecter une anomalie

        # Déterminer la raison principale
        primary_reason = None
        if anomaly_detected:
            # Trouver la méthode avec le score d'anomalie le plus élevé
            max_score = 0
            for method, result in results.items():
                if result["detected"] and result.get("score", 0) > max_score:
                    max_score = result.get("score", 0)
                    primary_reason = result.get("reason", method)

        # Créer le résultat final
        result = {
            "detected": anomaly_detected,
            "reason": primary_reason if anomaly_detected else None,
```

```
            "anomaly_count": anomaly_count,
            "timestamp": datetime.now().isoformat(),
            "symbol": data.get('symbol', 'unknown'),
            "current_price": current_close
        }

        # Ajouter les détails si demandé
        if return_details:
            result["details"] = results

        # Enregistrer l'anomalie dans l'historique si détectée
        if anomaly_detected:
            self.anomaly_history.append(result)
            logger.warning(f"Anomalie de marché détectée: {primary_reason}")

        return result

    def _detect_volatility_anomaly(self, data: pd.DataFrame, current_price: float) -> Dict:
        """
        Détecte une volatilité anormalement élevée

        Args:
            data: DataFrame avec les données OHLCV
            current_price: Prix actuel

        Returns:
            Résultat de la détection
        """
        # Calculer l'ATR sur la période de lookback
        try:
            # Calculer le True Range
```

```
high_low = data['high'] - data['low']
high_close = np.abs(data['high'] - data['close'].shift(1))
low_close = np.abs(data['low'] - data['close'].shift(1))

ranges = pd.concat([high_low, high_close, low_close], axis=1)
true_range = np.max(ranges, axis=1)

# Calculer l'ATR (Average True Range)
atr = true_range.rolling(window=14).mean().iloc[-1]

# Calculer la volatilité récente (écart-type des rendements)
returns = data['close'].pct_change().dropna()
recent_volatility = returns.tail(10).std() * np.sqrt(10)  # Annualisé à 10 périodes

# Calculer la volatilité historique
historical_volatility = returns.std() * np.sqrt(self.lookback_period)

# Calculer la volatilité relative
volatility_ratio = recent_volatility / historical_volatility if historical_volatility > 0 else 1.0

# Vérifier si la volatilité actuelle dépasse le seuil
is_anomaly = volatility_ratio > self.volatility_threshold

# Calculer un score d'anomalie
score = volatility_ratio / self.volatility_threshold

return {
    "detected": is_anomaly,
    "reason": f"Volatilité extrême ({volatility_ratio:.2f}x la normale)" if is_anomaly else None,
    "atr": atr,
    "recent_volatility": recent_volatility,
```

```
            "historical_volatility": historical_volatility,
            "volatility_ratio": volatility_ratio,
            "score": score
        }

    except Exception as e:
        logger.error(f"Erreur lors de la détection d'anomalie de volatilité: {str(e)}")
        return {"detected": False, "reason": f"Erreur: {str(e)}", "score": 0}

def _detect_price_gap(self, data: pd.DataFrame, current_price: float) -> Dict:
    """
    Détecte un gap de prix significatif

    Args:
        data: DataFrame avec les données OHLCV
        current_price: Prix actuel

    Returns:
        Résultat de la détection
    """
    try:
        # Calculer l'ATR pour normaliser les gaps
        high_low = data['high'] - data['low']
        high_close = np.abs(data['high'] - data['close'].shift(1))
        low_close = np.abs(data['low'] - data['close'].shift(1))

        ranges = pd.concat([high_low, high_close, low_close], axis=1)
        true_range = np.max(ranges, axis=1)

        atr = true_range.rolling(window=14).mean().iloc[-1]
```

```
        # Calculer le gap entre le prix actuel et le précédent
        previous_close = data['close'].iloc[-2]
        gap_size = abs(current_price - previous_close)

        # Normaliser par l'ATR
        normalized_gap = gap_size / atr if atr > 0 else 0

        # Vérifier si le gap dépasse le seuil
        is_anomaly = normalized_gap > self.price_gap_threshold

        # Calculer un score d'anomalie
        score = normalized_gap / self.price_gap_threshold

        return {
            "detected": is_anomaly,
            "reason": f"Gap de prix significatif ({normalized_gap:.2f}x ATR)" if is_anomaly else None,
            "gap_size": gap_size,
            "normalized_gap": normalized_gap,
            "atr": atr,
            "score": score
        }

    except Exception as e:
        logger.error(f"Erreur lors de la détection de gap de prix: {str(e)}")
        return {"detected": False, "reason": f"Erreur: {str(e)}", "score": 0}

def _detect_volume_anomaly(self, data: pd.DataFrame) -> Dict:
    """
    Détecte un volume anormal

    Args:
```

```
        data: DataFrame avec les données OHLCV

    Returns:
        Résultat de la détection
    """
    try:
        # Calculer le volume moyen
        avg_volume = data['volume'].mean()

        # Vérifier le volume récent
        recent_volume = data['volume'].iloc[-1]

        # Calculer le ratio
        volume_ratio = recent_volume / avg_volume if avg_volume > 0 else 1.0

        # Vérifier si le volume dépasse le seuil
        is_anomaly = volume_ratio > self.volume_threshold

        # Calculer un score d'anomalie
        score = volume_ratio / self.volume_threshold

        return {
            "detected": is_anomaly,
            "reason": f"Volume anormalement élevé ({volume_ratio:.2f}x la moyenne)" if is_anomaly else None,
            "recent_volume": recent_volume,
            "avg_volume": avg_volume,
            "volume_ratio": volume_ratio,
            "score": score
        }
```

```
        except Exception as e:
            logger.error(f"Erreur lors de la détection d'anomalie de volume: {str(e)}")
            return {"detected": False, "reason": f"Erreur: {str(e)}", "score": 0}

    def _detect_fat_tails(self, data: pd.DataFrame) -> Dict:
        """
        Détecte des queues de distribution épaisses (fat tails)

        Args:
            data: DataFrame avec les données OHLCV

        Returns:
            Résultat de la détection
        """
        try:
            # Calculer les rendements
            returns = data['close'].pct_change().dropna()

            # Ignorer si pas assez de données
            if len(returns) < 30:
                return {"detected": False, "reason": "Données insuffisantes pour l'analyse des queues de distribution", "score": 0}

            # Calculer le kurtosis (mesure de l'épaisseur des queues)
            kurt = stats.kurtosis(returns)

            # Le kurtosis d'une distribution normale est de 0
            # Une valeur > 3 indique des queues épaisses
            is_anomaly = kurt > 3.0

            # Calculer un score d'anomalie
```

```
            score = kurt / 3.0 if kurt > 0 else 0

            return {
                "detected": is_anomaly,
                "reason": f"Distribution des rendements à queues épaisses (kurtosis={kurt:.2f})" if is_anomaly else None,
                "kurtosis": kurt,
                "score": score
            }

        except Exception as e:
            logger.error(f"Erreur lors de la détection des queues de distribution: {str(e)}")
            return {"detected": False, "reason": f"Erreur: {str(e)}", "score": 0}

    def _detect_stationarity_break(self, data: pd.DataFrame) -> Dict:
        """
        Détecte une rupture de stationnarité dans la série

        Args:
            data: DataFrame avec les données OHLCV

        Returns:
            Résultat de la détection
        """
        try:
            # Calculer les rendements
            returns = data['close'].pct_change().dropna()

            # Ignorer si pas assez de données
            if len(returns) < 30:
                return {"detected": False, "reason": "Données insuffisantes pour le test de stationnarité", "score": 0}
```

```
            # Test de Dickey-Fuller augmenté
            with warnings.catch_warnings():
                warnings.simplefilter("ignore")
                result = adfuller(returns)

            # Extraire les statistiques
            adf_stat = result[0]
            p_value = result[1]

            # Seuil de signification
            significance = 0.05

            # Si p-value > significance, alors la série n'est pas stationnaire
            is_anomaly = p_value > significance

            # Calculer un score d'anomalie
            score = p_value / significance if significance > 0 else 0

            return {
                "detected": is_anomaly,
                "reason": f"Rupture de stationnarité détectée (p-value={p_value:.4f})" if is_anomaly else
None,
                "adf_statistic": adf_stat,
                "p_value": p_value,
                "score": score
            }

        except Exception as e:
            logger.error(f"Erreur lors du test de stationnarité: {str(e)}")
            return {"detected": False, "reason": f"Erreur: {str(e)}", "score": 0}
```

```
def _detect_microstructure_anomaly(self, data: pd.DataFrame, current_price: float) -> Dict:
    """
    Détecte des anomalies dans la microstructure du marché

    Args:
        data: DataFrame avec les données OHLCV
        current_price: Prix actuel

    Returns:
        Résultat de la détection
    """
    try:
        # Calculer l'écart entre high/low et le prix de clôture
        recent_data = data.tail(5)  # 5 dernières périodes

        # Calculer les ratios high-close et low-close
        high_close_ratios = (recent_data['high'] - recent_data['close']) / recent_data['close']
        low_close_ratios = (recent_data['close'] - recent_data['low']) / recent_data['close']

        # Calculer les moyennes historiques
        hist_high_close_ratio = (data['high'] - data['close']) / data['close']
        hist_low_close_ratio = (data['close'] - data['low']) / data['close']

        avg_high_ratio = hist_high_close_ratio.mean()
        avg_low_ratio = hist_low_close_ratio.mean()

        # Vérifier si les ratios récents sont anormaux
        recent_high_ratio = high_close_ratios.mean()
        recent_low_ratio = low_close_ratios.mean()
```

```
# Calculer les écarts
high_deviation = recent_high_ratio / avg_high_ratio if avg_high_ratio > 0 else 1.0
low_deviation = recent_low_ratio / avg_low_ratio if avg_low_ratio > 0 else 1.0

# Vérifier si l'un des écarts dépasse le seuil
high_anomaly = high_deviation > 2.0
low_anomaly = low_deviation > 2.0

is_anomaly = high_anomaly or low_anomaly

# Calculer un score d'anomalie
score = max(high_deviation, low_deviation) / 2.0

# Construire la raison
reason = None
if is_anomaly:
    if high_anomaly and low_anomaly:
        reason = f"Mèches anormales (H:{high_deviation:.2f}x, L:{low_deviation:.2f}x)"
    elif high_anomaly:
        reason = f"Mèche supérieure anormale ({high_deviation:.2f}x la normale)"
    else:
        reason = f"Mèche inférieure anormale ({low_deviation:.2f}x la normale)"

return {
    "detected": is_anomaly,
    "reason": reason,
    "high_deviation": high_deviation,
    "low_deviation": low_deviation,
    "score": score
}
```

```
    except Exception as e:
        logger.error(f"Erreur lors de la détection d'anomalie de microstructure: {str(e)}")
        return {"detected": False, "reason": f"Erreur: {str(e)}", "score": 0}

def _detect_momentum_anomaly(self, data: pd.DataFrame) -> Dict:
    """
    Détecte un momentum extrême

    Args:
        data: DataFrame avec les données OHLCV

    Returns:
        Résultat de la détection
    """
    try:
        # Calculer les rendements sur plusieurs périodes
        returns_1d = data['close'].pct_change(1).iloc[-1]
        returns_3d = data['close'].pct_change(3).iloc[-1]
        returns_5d = data['close'].pct_change(5).iloc[-1]

        # Calculer les rendements historiques
        hist_returns_1d = data['close'].pct_change(1).std()
        hist_returns_3d = data['close'].pct_change(3).std()
        hist_returns_5d = data['close'].pct_change(5).std()

        # Calculer les z-scores
        z_score_1d = returns_1d / hist_returns_1d if hist_returns_1d > 0 else 0
        z_score_3d = returns_3d / hist_returns_3d if hist_returns_3d > 0 else 0
        z_score_5d = returns_5d / hist_returns_5d if hist_returns_5d > 0 else 0

        # Utiliser le z-score maximum
```

```
            max_z_score = max(abs(z_score_1d), abs(z_score_3d), abs(z_score_5d))

            # Seuil pour considérer un momentum comme anormal
            threshold = 2.5  # 2.5 écarts-types

            is_anomaly = max_z_score > threshold

            # Déterminer la direction du momentum
            direction = "haussier" if max(returns_1d, returns_3d, returns_5d) > 0 else "baissier"

            # Calculer un score d'anomalie
            score = max_z_score / threshold

            return {
                "detected": is_anomaly,
                "reason": f"Momentum {direction} extrême (z-score={max_z_score:.2f})" if is_anomaly else
None,
                "z_score_1d": z_score_1d,
                "z_score_3d": z_score_3d,
                "z_score_5d": z_score_5d,
                "max_z_score": max_z_score,
                "direction": direction if is_anomaly else None,
                "score": score
            }

        except Exception as e:
            logger.error(f"Erreur lors de la détection d'anomalie de momentum: {str(e)}")
            return {"detected": False, "reason": f"Erreur: {str(e)}", "score": 0}

    def _detect_ml_anomalies(self, data: pd.DataFrame) -> Dict:
        """
```

```
Utilise des modèles de ML pour détecter les anomalies

Args:
    data: DataFrame avec les données OHLCV

Returns:
    Résultat de la détection
"""
# Si les modèles de ML ne sont pas activés
if not self.use_ml_models:
    return {"detected": False, "reason": "Modèles ML non activés", "score": 0}

try:
    # Extraire les caractéristiques pertinentes
    features = self._extract_anomaly_features(data)

    # Si pas de modèles chargés, les entraîner
    if self.isolation_forest is None or self.autoencoder is None:
        self._train_anomaly_models(data)

    # Prédictions de l'Isolation Forest (si disponible)
    forest_score = 0
    if self.isolation_forest is not None:
        # -1 pour les anomalies, 1 pour les normales, convertir à un score entre 0 et 1
        forest_pred = self.isolation_forest.predict([features])
        forest_score = self.isolation_forest.score_samples([features])[0]
        forest_score = 0.5 + (forest_score * -0.5)  # Convertir à un score d'anomalie (0-1)

    # Prédictions de l'Autoencoder (si disponible)
    autoencoder_score = 0
    if self.autoencoder is not None:
```

```
            # Préparation des données pour l'autoencoder
            ae_input = np.array([features])

            # Prédiction
            ae_pred = self.autoencoder.predict(ae_input)

            # Erreur de reconstruction
            reconstruction_error = np.mean(np.square(ae_input - ae_pred))

            # Normaliser l'erreur à un score entre 0 et 1
            autoencoder_score = min(1.0, reconstruction_error / 0.1)  # Seuil d'erreur de 0.1

        # Combiner les scores (moyenne)
        combined_score = (forest_score + autoencoder_score) / 2 if (self.isolation_forest is not None and self.autoencoder is not None) else max(forest_score, autoencoder_score)

        # Seuil pour considérer comme une anomalie
        threshold = 0.7
        is_anomaly = combined_score > threshold

        return {
            "detected": is_anomaly,
            "reason": f"Anomalie détectée par ML (score={combined_score:.2f})" if is_anomaly else None,
            "forest_score": forest_score,
            "autoencoder_score": autoencoder_score,
            "combined_score": combined_score,
            "score": combined_score
        }

    except Exception as e:
        logger.error(f"Erreur lors de la détection d'anomalies par ML: {str(e)}")
```

```
            return {"detected": False, "reason": f"Erreur: {str(e)}", "score": 0}

    def _extract_anomaly_features(self, data: pd.DataFrame) -> List[float]:
        """
        Extrait les caractéristiques pour la détection d'anomalies

        Args:
            data: DataFrame avec les données OHLCV

        Returns:
            Liste des caractéristiques
        """
        # Calculer les rendements
        returns = data['close'].pct_change().dropna()

        # Caractéristiques de volatilité
        volatility_1d = returns.tail(1).std()
        volatility_5d = returns.tail(5).std() * np.sqrt(5)
        volatility_10d = returns.tail(10).std() * np.sqrt(10)

        # Caractéristiques de momentum
        returns_1d = returns.iloc[-1]
        returns_3d = (data['close'].iloc[-1] / data['close'].iloc[-4] - 1) if len(data) >= 4 else 0
        returns_5d = (data['close'].iloc[-1] / data['close'].iloc[-6] - 1) if len(data) >= 6 else 0

        # Caractéristiques de volume
        volume_ratio_1d = data['volume'].iloc[-1] / data['volume'].iloc[-2] if data['volume'].iloc[-2] > 0
else 1
        volume_ratio_5d = data['volume'].iloc[-1] / data['volume'].tail(5).mean() if
data['volume'].tail(5).mean() > 0 else 1

        # Caractéristiques des chandeliers
```

```
        body_size = abs(data['close'].iloc[-1] - data['open'].iloc[-1]) / data['open'].iloc[-1]
        upper_wick = (data['high'].iloc[-1] - max(data['open'].iloc[-1], data['close'].iloc[-1])) / data['open'].iloc[-1]
        lower_wick = (min(data['open'].iloc[-1], data['close'].iloc[-1]) - data['low'].iloc[-1]) / data['open'].iloc[-1]

        # Combiner les caractéristiques
        features = [
            volatility_1d,
            volatility_5d,
            volatility_10d,
            returns_1d,
            returns_3d,
            returns_5d,
            volume_ratio_1d,
            volume_ratio_5d,
            body_size,
            upper_wick,
            lower_wick
        ]

        # Remplacer les valeurs NaN ou infinies
        features = [0.0 if (np.isnan(f) or np.isinf(f)) else f for f in features]

        return features

    def _train_anomaly_models(self, data: pd.DataFrame) -> None:
        """
        Entraîne les modèles de détection d'anomalies sur les données historiques

        Args:
            data: DataFrame avec les données OHLCV
```

```
"""
from sklearn.ensemble import IsolationForest

try:
    # Préparer les caractéristiques pour tous les points de données
    features_list = []

    # Fenêtre glissante pour extraire les caractéristiques
    for i in range(self.lookback_period, len(data)):
        window_data = data.iloc[i-self.lookback_period:i]
        features = self._extract_anomaly_features(window_data)
        features_list.append(features)

    # S'il n'y a pas assez de données, sortir
    if len(features_list) < 50:
        logger.warning("Données insuffisantes pour entraîner les modèles d'anomalies")
        return

    # Entraîner l'Isolation Forest
    self.isolation_forest = IsolationForest(
        n_estimators=100,
        max_samples='auto',
        contamination=0.05,  # 5% d'anomalies attendues
        random_state=42
    )
    self.isolation_forest.fit(features_list)

    # Entraîner l'Autoencoder
    self._train_autoencoder(np.array(features_list))

    # Sauvegarder les modèles si un répertoire est spécifié
```

```
            if self.model_dir:
                self._save_models()

            logger.info("Modèles de détection d'anomalies entraînés avec succès")

        except Exception as e:
            logger.error(f"Erreur lors de l'entraînement des modèles d'anomalies: {str(e)}")

    def _train_autoencoder(self, features: np.ndarray) -> None:
        """
        Entraîne un autoencoder pour la détection d'anomalies

        Args:
            features: Tableau numpy avec les caractéristiques
        """
        try:
            # Nombre de caractéristiques
            input_dim = features.shape[1]

            # Définir l'architecture de l'autoencoder
            input_layer = Input(shape=(input_dim,))

            # Encodeur
            encoded = Dense(8, activation='relu')(input_layer)
            encoded = Dense(4, activation='relu')(encoded)

            # Décodeur
            decoded = Dense(8, activation='relu')(encoded)
            decoded = Dense(input_dim, activation='linear')(decoded)

            # Modèle complet
```

```
            self.autoencoder = Model(input_layer, decoded)

            # Compilation
            self.autoencoder.compile(optimizer='adam', loss='mse')

            # Entraînement
            self.autoencoder.fit(
                features,
                features,
                epochs=50,
                batch_size=32,
                shuffle=True,
                verbose=0
            )

            logger.info("Autoencoder entraîné avec succès")

        except Exception as e:
            logger.error(f"Erreur lors de l'entraînement de l'autoencoder: {str(e)}")
            self.autoencoder = None

    def _save_models(self) -> None:
        """Sauvegarde les modèles d'anomalies"""
        try:
            os.makedirs(self.model_dir, exist_ok=True)

            # Sauvegarder l'Isolation Forest
            if self.isolation_forest is not None:
                with open(os.path.join(self.model_dir, "isolation_forest.pkl"), 'wb') as f:
                    pickle.dump(self.isolation_forest, f)
```

```
        # Sauvegarder l'Autoencoder
        if self.autoencoder is not None:
            self.autoencoder.save(os.path.join(self.model_dir, "autoencoder"))

        logger.info(f"Modèles de détection d'anomalies sauvegardés dans {self.model_dir}")

    except Exception as e:
        logger.error(f"Erreur lors de la sauvegarde des modèles: {str(e)}")

def _load_models(self) -> None:
    """Charge les modèles d'anomalies"""
    try:
        # Charger l'Isolation Forest
        forest_path = os.path.join(self.model_dir, "isolation_forest.pkl")
        if os.path.exists(forest_path):
            with open(forest_path, 'rb') as f:
                self.isolation_forest = pickle.load(f)
            logger.info("Isolation Forest chargé")

        # Charger l'Autoencoder
        autoencoder_path = os.path.join(self.model_dir, "autoencoder")
        if os.path.exists(autoencoder_path):
            self.autoencoder = tf.keras.models.load_model(autoencoder_path)
            logger.info("Autoencoder chargé")

    except Exception as e:
        logger.error(f"Erreur lors du chargement des modèles: {str(e)}")

def get_anomaly_history(self, limit: int = 10) -> List[Dict]:
    """
    Récupère l'historique des anomalies détectées
```

Args:

limit: Nombre maximum d'anomalies à retourner

Returns:

Liste des anomalies détectées

"""

return self.anomaly_history[-limit:]

Exemple d'intégration dans le gestionnaire de risque adaptatif:

#

Dans adaptive_risk_manager.py, remplacer la méthode _detect_extreme_market_conditions

==========

File: crypto_trading_bot_CLAUDE/ai/parameter_optimizer.py

==========

ai/parameter_optimizer.py

"""

Optimiseur de paramètres pour la stratégie de trading

"""

import os

import json

import logging

import random

import numpy as np

from typing import Dict, List, Optional, Union

from datetime import datetime

from config.config import DATA_DIR

from config.trading_params import LEARNING_RATE

```
from utils.logger import setup_logger

logger = setup_logger("parameter_optimizer")

class ParameterOptimizer:
    """
    Optimise les paramètres de la stratégie en fonction des performances passées
    """
    def __init__(self, trade_analyzer):
        self.trade_analyzer = trade_analyzer
        self.optimizer_dir = os.path.join(DATA_DIR, "optimizer")
        self.parameters_file = os.path.join(self.optimizer_dir, "optimized_parameters.json")
        self.history_file = os.path.join(self.optimizer_dir, "optimization_history.json")

        # Créer le répertoire si nécessaire
        if not os.path.exists(self.optimizer_dir):
            os.makedirs(self.optimizer_dir)

        # Paramètres actuels et historique
        self.current_parameters = {}
        self.optimization_history = []

        # Charger les paramètres et l'historique
        self._load_parameters()
        self._load_history()

    def _load_parameters(self) -> None:
        """
        Charge les paramètres optimisés depuis le fichier
        """
        if os.path.exists(self.parameters_file):
```

```
        try:
            with open(self.parameters_file, 'r') as f:
                self.current_parameters = json.load(f)
            logger.info("Paramètres optimisés chargés")
        except Exception as e:
            logger.error(f"Erreur lors du chargement des paramètres: {str(e)}")
            self.current_parameters = {}

    def _load_history(self) -> None:
        """
        Charge l'historique d'optimisation depuis le fichier
        """
        if os.path.exists(self.history_file):
            try:
                with open(self.history_file, 'r') as f:
                    self.optimization_history = json.load(f)
                logger.info(f"Historique d'optimisation chargé: {len(self.optimization_history)} entrées")
            except Exception as e:
                logger.error(f"Erreur lors du chargement de l'historique: {str(e)}")
                self.optimization_history = []

    def _save_parameters(self) -> None:
        """
        Sauvegarde les paramètres optimisés dans le fichier
        """
        try:
            with open(self.parameters_file, 'w') as f:
                json.dump(self.current_parameters, f, indent=2)
            logger.debug("Paramètres optimisés sauvegardés")
        except Exception as e:
            logger.error(f"Erreur lors de la sauvegarde des paramètres: {str(e)}")
```

```
def _save_history(self) -> None:
    """
    Sauvegarde l'historique d'optimisation dans le fichier
    """
    try:
        # Limiter la taille de l'historique (garder les 100 dernières entrées)
        if len(self.optimization_history) > 100:
            self.optimization_history = self.optimization_history[-100:]

        with open(self.history_file, 'w') as f:
            json.dump(self.optimization_history, f, indent=2, default=str)
        logger.debug("Historique d'optimisation sauvegardé")
    except Exception as e:
        logger.error(f"Erreur lors de la sauvegarde de l'historique: {str(e)}")

def optimize_parameters(self) -> Dict:
    """
    Optimise les paramètres en fonction des performances récentes - Version améliorée

    Returns:
        Dictionnaire avec les paramètres optimisés
    """
    # Analyser les trades récents
    analysis = self.trade_analyzer.analyze_recent_trades(days=30)

    if not analysis.get("success", False):
        return {
            "success": False,
            "message": "Impossible d'optimiser les paramètres",
            "parameters": self.current_parameters
```

```
    }

# Générer des recommandations
recommendations = self.trade_analyzer.generate_recommendations()

# Initialiser les paramètres par défaut si nécessaire
if not self.current_parameters:
    self._initialize_default_parameters()

# Sauvegarder les paramètres actuels
previous_parameters = self.current_parameters.copy()

# NOUVELLE APPROCHE: Analyse bayésienne optimale
try:
    # Définir les bornes des paramètres à optimiser
    param_bounds = self._define_parameter_bounds()

    # Optimiser les paramètres en fonction des performances passées
    optimized_params = self._bayesian_optimization(analysis, param_bounds)

    # Mettre à jour les paramètres avec les valeurs optimisées
    if "technical_bounce" in self.current_parameters:
        for param, value in optimized_params.items():
            if param in self.current_parameters["technical_bounce"]:
                # Limiter les changements à 20% maximum par itération pour éviter les sauts extrêmes
                current_value = self.current_parameters["technical_bounce"][param]
                max_change = current_value * 0.2

                # Calculer la nouvelle valeur en limitant le changement
                if abs(value - current_value) > max_change:
                    if value > current_value:
```

```
                    new_value = current_value + max_change
                else:
                    new_value = current_value - max_change
            else:
                new_value = value

            self.current_parameters["technical_bounce"][param] = new_value
            logger.info(f"Paramètre {param} optimisé: {current_value} -> {new_value}")
except Exception as e:
    logger.error(f"Erreur dans l'optimisation bayésienne: {str(e)}")
    # Continuer avec l'approche traditionnelle en cas d'erreur

# Ajuster les paramètres en fonction des recommandations (ancienne approche en backup)
self._adjust_parameters_based_on_recommendations(recommendations)

# NOUVEAU: Analyser les corrélations entre paramètres et performance
param_performance_correlations = self._analyze_parameter_performance_correlations()

# Affiner les paramètres en fonction des corrélations
for param, correlation in param_performance_correlations.items():
    if abs(correlation) > 0.6 and param in self.current_parameters.get("technical_bounce", {}):
        current_value = self.current_parameters["technical_bounce"][param]

        # Si corrélation positive, augmenter le paramètre
        if correlation > 0:
            adjustment = current_value * 0.05
            new_value = current_value + adjustment
        # Si corrélation négative, diminuer le paramètre
        else:
            adjustment = current_value * 0.05
            new_value = current_value - adjustment
```

```
                self.current_parameters["technical_bounce"][param] = new_value
                logger.info(f"Paramètre {param} ajusté par corrélation: {current_value} -> {new_value}")

        # Appliquer une petite exploration aléatoire
        self._apply_random_exploration()

        # Valider les paramètres optimisés
        self._validate_parameters()

        # Enregistrer l'historique
        history_entry = {
            "timestamp": datetime.now().isoformat(),
            "previous_parameters": previous_parameters,
            "new_parameters": self.current_parameters,
            "analysis_summary": {
                "success_rate": analysis.get("success_rate"),
                "avg_pnl": analysis.get("avg_pnl"),
                "total_trades": analysis.get("total_trades")
            },
            "recommendations": recommendations.get("parameter_adjustments", [])
        }

        self.optimization_history.append(history_entry)

        # Sauvegarder les paramètres et l'historique
        self._save_parameters()
        self._save_history()

        return {
            "success": True,
```

```
        "message": "Paramètres optimisés avec succès",
        "parameters": self.current_parameters,
        "previous_parameters": previous_parameters,
        "changes": self._get_parameter_changes(previous_parameters)
    }


def _initialize_default_parameters(self) -> None:
    """
    Initialise les paramètres par défaut
    """
    self.current_parameters = {
        "technical_bounce": {
            # Paramètres RSI
            "rsi_period": 14,
            "rsi_oversold": 30,
            "rsi_overbought": 70,

            # Paramètres Bollinger
            "bb_period": 20,
            "bb_deviation": 2,

            # Paramètres EMA
            "ema_short": 9,
            "ema_medium": 21,
            "ema_long": 50,

            # Paramètres ATR
            "atr_period": 14,
            "atr_multiplier": 1.5,
```

```
            # Paramètres de gestion des risques
            "risk_per_trade_percent": 7.5,
            "stop_loss_percent": 4.0,
            "take_profit_percent": 6.0,
            "trailing_stop_activation": 2.0,
            "trailing_stop_step": 0.5,

            # Paramètres de scoring
            "minimum_score": 70
        }
    }

    logger.info("Paramètres par défaut initialisés")
    self._save_parameters()

def _adjust_parameters_based_on_recommendations(self, recommendations: Dict) -> None:
    """
    Ajuste les paramètres en fonction des recommandations

    Args:
        recommendations: Recommandations générées par l'analyseur
    """
    # Extraire les recommandations de paramètres
    parameter_adjustments = recommendations.get("parameter_adjustments", [])

    for adjustment in parameter_adjustments:
        parameter = adjustment.get("parameter", "")
        recommendation = adjustment.get("recommendation", "")

        # Ajuster les paramètres RSI
        if parameter == "rsi_weight" and "Augmenter" in recommendation:
```

```
        strategy_params = self.current_parameters.get("technical_bounce", {})

        # Diminuer le seuil de survente pour capturer plus de signaux RSI
        if "rsi_oversold" in strategy_params:
            current_value = strategy_params["rsi_oversold"]
            new_value = max(20, current_value - 2)  # Ne pas descendre en dessous de 20
            strategy_params["rsi_oversold"] = new_value
            logger.info(f"Seuil RSI survente ajusté: {current_value} -> {new_value}")

    # Ajuster les paramètres Bollinger
    elif parameter == "bollinger_weight" and "Augmenter" in recommendation:
        strategy_params = self.current_parameters.get("technical_bounce", {})

        # Augmenter la déviation pour des bandes plus larges
        if "bb_deviation" in strategy_params:
            current_value = strategy_params["bb_deviation"]
            new_value = min(2.5, current_value + 0.1)  # Ne pas dépasser 2.5
            strategy_params["bb_deviation"] = new_value
            logger.info(f"Déviation Bollinger ajustée: {current_value} -> {new_value}")

    # Ajuster le seuil de score minimum
    elif "score" in parameter.lower():
        strategy_params = self.current_parameters.get("technical_bounce", {})

        if "Augmenter" in recommendation and "minimum_score" in strategy_params:
            current_value = strategy_params["minimum_score"]
            new_value = min(85, current_value + 5)  # Ne pas dépasser 85
            strategy_params["minimum_score"] = new_value
            logger.info(f"Score minimum ajusté: {current_value} -> {new_value}")
        elif "Diminuer" in recommendation and "minimum_score" in strategy_params:
            current_value = strategy_params["minimum_score"]
```

```
            new_value = max(60, current_value - 5)  # Ne pas descendre en dessous de 60
            strategy_params["minimum_score"] = new_value
            logger.info(f"Score minimum ajusté: {current_value} -> {new_value}")

def _apply_random_exploration(self) -> None:
    """
    Applique une exploration aléatoire pour éviter les optima locaux
    """
    # Probabilité d'appliquer une exploration (20%)
    if random.random() > 0.2:
        return

    strategy_params = self.current_parameters.get("technical_bounce", {})

    if not strategy_params:
        return

    # Sélectionner un paramètre aléatoire à ajuster
    adjustable_params = [
        "rsi_period", "rsi_oversold", "rsi_overbought",
        "bb_period", "bb_deviation",
        "ema_short", "ema_medium",
        "atr_period", "atr_multiplier",
        "risk_per_trade_percent", "stop_loss_percent", "take_profit_percent",
        "trailing_stop_activation", "trailing_stop_step",
        "minimum_score"
    ]

    # Filtrer les paramètres présents
    adjustable_params = [p for p in adjustable_params if p in strategy_params]
```

```
if not adjustable_params:
    return

# Sélectionner un paramètre aléatoire
param = random.choice(adjustable_params)
current_value = strategy_params[param]

# Ajuster en fonction du type de paramètre
if param in ["rsi_period", "bb_period", "ema_short", "ema_medium", "atr_period"]:
    # Paramètres de période (entiers)
    adjustment = random.choice([-2, -1, 1, 2])
    new_value = max(5, current_value + adjustment)
elif param in ["rsi_oversold"]:
    # Seuil de survente
    adjustment = random.choice([-3, -2, -1, 1, 2, 3])
    new_value = max(20, min(40, current_value + adjustment))
elif param in ["rsi_overbought"]:
    # Seuil de surachat
    adjustment = random.choice([-3, -2, -1, 1, 2, 3])
    new_value = max(60, min(80, current_value + adjustment))
elif param in ["bb_deviation", "atr_multiplier"]:
    # Paramètres de multiplicateur (flottants)
    adjustment = random.choice([-0.2, -0.1, 0.1, 0.2])
    new_value = max(0.5, current_value + adjustment)
elif param in ["risk_per_trade_percent"]:
    # Pourcentage de risque
    adjustment = random.choice([-1.0, -0.5, 0.5, 1.0])
    new_value = max(5.0, min(10.0, current_value + adjustment))
elif param in ["stop_loss_percent"]:
    # Pourcentage de stop-loss
    adjustment = random.choice([-0.5, -0.25, 0.25, 0.5])
```

```
            new_value = max(3.0, min(5.0, current_value + adjustment))
        elif param in ["take_profit_percent"]:
            # Pourcentage de take-profit
            adjustment = random.choice([-0.5, -0.25, 0.25, 0.5])
            new_value = max(5.0, min(7.0, current_value + adjustment))
        elif param in ["trailing_stop_activation", "trailing_stop_step"]:
            # Paramètres de trailing stop
            adjustment = random.choice([-0.2, -0.1, 0.1, 0.2])
            new_value = max(0.5, current_value + adjustment)
        elif param in ["minimum_score"]:
            # Score minimum
            adjustment = random.choice([-5, -3, 3, 5])
            new_value = max(60, min(85, current_value + adjustment))
        else:
            return

        # Appliquer le changement
        strategy_params[param] = new_value
        logger.info(f"Exploration aléatoire: {param} ajusté de {current_value} à {new_value}")

    def _get_parameter_changes(self, previous_parameters: Dict) -> List[Dict]:
        """
        Identifie les changements entre les anciennes et nouvelles valeurs de paramètres

        Args:
            previous_parameters: Anciens paramètres

        Returns:
            Liste des changements
        """
        changes = []
```

```
    for strategy, params in self.current_parameters.items():
        if strategy in previous_parameters:
            for param, new_value in params.items():
                if param in previous_parameters[strategy]:
                    old_value = previous_parameters[strategy][param]


                    if new_value != old_value:
                        changes.append({
                            "strategy": strategy,
                            "parameter": param,
                            "old_value": old_value,
                            "new_value": new_value
                        })
        else:
            # Nouvelle stratégie ajoutée
            for param, value in params.items():
                changes.append({
                    "strategy": strategy,
                    "parameter": param,
                    "old_value": None,
                    "new_value": value,
                    "status": "new"
                })


    return changes


def get_current_parameters(self) -> Dict:
    """
    Récupère les paramètres actuels
```

```
    Returns:
        Paramètres actuels
    """
    return self.current_parameters

def apply_parameters(self, trading_bot) -> None:
    """
    Applique les paramètres optimisés au bot de trading

    Args:
        trading_bot: Bot de trading à configurer
    """
    # Cette méthode sera implémentée pour appliquer les paramètres au bot
    pass

def get_market_adaptive_parameters(self, symbol: str, data_fetcher) -> Dict:
    """Adapte dynamiquement les paramètres selon les conditions de marché actuelles"""
    market_data = data_fetcher.get_market_data(symbol)

    # Obtenir les indicateurs de volatilité
    volatility_metrics = self._calculate_volatility_metrics(market_data)
    market_trend = self._detect_market_trend(market_data)

    # Adapter les paramètres selon la volatilité
    params = self.current_parameters.get("technical_bounce", {}).copy()

    # Volatilité faible = paramètres plus agressifs
    if volatility_metrics["atr_percent"] < 1.5:
        params["risk_per_trade_percent"] = 4.0
        params["stop_loss_percent"] = 2.8
        params["take_profit_percent"] = 7.0
```

```
    # Volatilité élevée = paramètres plus conservateurs
    elif volatility_metrics["atr_percent"] > 3.0:
        params["risk_per_trade_percent"] = 2.0
        params["stop_loss_percent"] = 4.0
        params["take_profit_percent"] = 10.0

    # Adapter selon la tendance
    if market_trend["strength"] > 0.7:
        # En tendance forte, ajuster le ratio risk/reward
        if market_trend["direction"] == "up":
            params["take_profit_percent"] += 1.5  # Plus ambitieux en tendance haussière
        else:
            params["stop_loss_percent"] -= 0.5   # Plus prudent en tendance baissière

    return params

def _define_parameter_bounds(self) -> Dict:
    """
    Définit les bornes des paramètres pour l'optimisation
    """
    return {
        # Paramètres RSI
        "rsi_period": (7, 21),          # Période du RSI
        "rsi_oversold": (20, 35),       # Seuil de survente
        "rsi_overbought": (65, 80),     # Seuil de surachat

        # Paramètres Bollinger
        "bb_period": (15, 25),          # Période des bandes de Bollinger
        "bb_deviation": (1.8, 2.5),     # Déviation standard
```

```
        # Paramètres EMA
        "ema_short": (8, 12),          # EMA courte
        "ema_medium": (18, 25),        # EMA moyenne

        # Paramètres ATR
        "atr_period": (10, 20),        # Période ATR
        "atr_multiplier": (1.2, 2.0),  # Multiplicateur ATR

        # Paramètres de gestion des risques
        "risk_per_trade_percent": (2.0, 8.0),     # Risque par trade
        "stop_loss_percent": (3.0, 5.0),          # Stop loss
        "take_profit_percent": (5.0, 9.0),        # Take profit
        "trailing_stop_activation": (1.0, 3.0),   # Activation trailing stop
        "trailing_stop_step": (0.3, 0.8),         # Pas de trailing stop

        # Paramètres de scoring
        "minimum_score": (65, 80)       # Score minimum
    }

def _bayesian_optimization(self, analysis: Dict, param_bounds: Dict) -> Dict:
    """
    Exécute une optimisation bayésienne pour trouver les meilleurs paramètres
    """
    # Cette méthode utiliserait une bibliothèque d'optimisation bayésienne comme scikit-optimize
    # Pour simplifier, nous retournons une approximation basée sur l'analyse des trades

    # Récupérer les données importantes de l'analyse
    win_rate = analysis.get("success_rate", 0)
    avg_win = analysis.get("avg_win", 0)
    avg_loss = analysis.get("avg_loss", 0)
```

```
    # Paramètres optimisés
    optimized_params = {}

    # Optimiser en fonction du win rate
    if win_rate < 40:
        # Améliorer la précision des entrées
        optimized_params["minimum_score"] = 75  # Plus sélectif
        optimized_params["rsi_oversold"] = 25   # Plus conservateur
        optimized_params["bb_deviation"] = 2.2  # Plus large
    else:
        # Maintenir l'équilibre actuel
        optimized_params["minimum_score"] = 70
        optimized_params["rsi_oversold"] = 30
        optimized_params["bb_deviation"] = 2.0

    # Optimiser en fonction du ratio gain/perte
    profit_factor = abs(avg_win / avg_loss) if avg_loss != 0 else 1.0

    if profit_factor > 2.5:
        # Si ratio très bon, optimiser pour plus de trades
        optimized_params["minimum_score"] = 65  # Moins sélectif
    elif profit_factor < 1.5:
        # Si ratio faible, améliorer la qualité des sorties
        optimized_params["take_profit_percent"] = 8.0  # Plus patient
        optimized_params["stop_loss_percent"] = 3.5    # Plus serré

    return optimized_params

def _analyze_parameter_performance_correlations(self) -> Dict:
    """
    Analyse les corrélations entre les changements de paramètres et les performances
```

```
        """
        # Pour simplifier, on retourne des valeurs prédéfinies basées sur des observations courantes
        # Dans une implémentation réelle, cela serait calculé à partir de l'historique
        return {
            "minimum_score": 0.7,       # Forte corrélation positive avec la performance
            "rsi_period": -0.2,         # Faible corrélation négative
            "stop_loss_percent": -0.5,  # Corrélation négative modérée
            "take_profit_percent": 0.4,  # Corrélation positive modérée
            "bb_deviation": 0.3         # Faible corrélation positive
        }

    def _validate_parameters(self) -> None:
        """
        Valide et corrige les paramètres pour s'assurer qu'ils sont cohérents
        """
        if "technical_bounce" not in self.current_parameters:
            return

        params = self.current_parameters["technical_bounce"]

        # S'assurer que le take profit est toujours supérieur au stop loss
        if "take_profit_percent" in params and "stop_loss_percent" in params:
            if params["take_profit_percent"] <= params["stop_loss_percent"]:
                params["take_profit_percent"] = params["stop_loss_percent"] * 1.5
                logger.warning(f"Correction du take profit pour qu'il soit > stop loss: {params['take_profit_percent']}")

        # S'assurer que les paramètres restent dans des limites raisonnables
        bounds = self._define_parameter_bounds()
        for param, (min_val, max_val) in bounds.items():
            if param in params and (params[param] < min_val or params[param] > max_val):
```

```
            # Ramener le paramètre dans ses bornes
            params[param] = max(min_val, min(params[param], max_val))
            logger.warning(f"Paramètre {param} ramené dans ses bornes: {params[param]}")

================================================
File: crypto_trading_bot_CLAUDE/ai/reasoning_engine.py
================================================
# ai/reasoning_engine.py
"""
Moteur de raisonnement pour expliquer les décisions et générer du texte
"""
import logging
from typing import Dict, List, Optional, Union
from datetime import datetime

from utils.logger import setup_logger

logger = setup_logger("reasoning_engine")

class ReasoningEngine:
    """
    Génère des explications textuelles pour les décisions de trading
    """
    def __init__(self):
        self.templates = self._initialize_templates()

    def _initialize_templates(self) -> Dict:
        """
        Initialise les templates pour la génération de texte

        Returns:
```

```
        Dictionnaire de templates
    """
    return {
        "technical_bounce": {
            "opportunity": (
                "Opportunité de rebond technique détectée sur {symbol} avec un score de {score}/100. "
                "Le prix est actuellement à {price} {base_currency}, montrant des signes de retournement haussier "
                "après une période de baisse. {signals_text} "
                "Le stop-loss est placé à {stop_loss} (-{stop_loss_percent}%) et "
                "le take-profit à {take_profit} (+{take_profit_percent}%), "
                "donnant un ratio risque/récompense de {risk_reward_ratio:.2f}."
            ),
            "market_conditions": (
                "Conditions de marché: {market_conditions}. "
                "RSI actuel: {rsi:.1f}. "
                "Force de tendance (ADX): {adx:.1f}. "
                "Volatilité (ATR): {atr:.2f}."
            ),
            "entry_reasoning": (
                "Raisonnement d'entrée: "
                "Le prix est {price_position} avec {candle_pattern}. "
                "{volume_analysis} "
                "{divergence_analysis} "
                "Les signaux techniques indiquent une forte probabilité de rebond à court terme."
            )
        },
        "trade_result": {
            "success": (
                "Trade sur {symbol} clôturé avec profit: +{pnl_percent:.2f}% (+{pnl_absolute:.2f} {currency}). "
                "Durée du trade: {duration}. "
```

```
                "Entrée à {entry_price}, sortie à {exit_price}. "
                "Raison de sortie: {exit_reason}."
            ),
            "failure": (
                "Trade sur {symbol} clôturé avec perte: {pnl_percent:.2f}% ({pnl_absolute:.2f} {currency}).
"
                "Durée du trade: {duration}. "
                "Entrée à {entry_price}, sortie à {exit_price}. "
                "Raison de sortie: {exit_reason}. "
                "Leçon à retenir: {lesson}."
            )
        }
    }

    def generate_opportunity_explanation(self, opportunity: Dict) -> str:
        """
        Génère une explication détaillée pour une opportunité de trading

        Args:
            opportunity: Données de l'opportunité

        Returns:
            Explication textuelle
        """
        strategy = opportunity.get("strategy", "unknown")

        if strategy not in self.templates:
            return f"Opportunité de trading détectée avec la stratégie '{strategy}'."

        template = self.templates[strategy]
```

```
# Extraire les données de l'opportunité
symbol = opportunity.get("symbol", "UNKNOWN")
score = opportunity.get("score", 0)
entry_price = opportunity.get("entry_price", 0)
stop_loss = opportunity.get("stop_loss", 0)
take_profit = opportunity.get("take_profit", 0)
signals = opportunity.get("signals", {}).get("signals", [])

# Calculer les pourcentages
stop_loss_percent = abs((stop_loss - entry_price) / entry_price * 100)
take_profit_percent = abs((take_profit - entry_price) / entry_price * 100)
risk_reward_ratio = take_profit_percent / stop_loss_percent if stop_loss_percent > 0 else 0

# Extraire la devise de base
base_currency = "USDT"  # Par défaut
if symbol.endswith("USDT"):
    base_currency = "USDT"

# Formater les signaux
signals_text = "Signaux détectés: " + ", ".join(signals) + "." if signals else ""

# Section 1: Opportunité de base
explanation = template["opportunity"].format(
    symbol=symbol,
    score=score,
    price=entry_price,
    base_currency=base_currency,
    signals_text=signals_text,
    stop_loss=stop_loss,
    stop_loss_percent=f"{stop_loss_percent:.2f}",
    take_profit=take_profit,
```

```
    take_profit_percent=f"{take_profit_percent:.2f}",
    risk_reward_ratio=risk_reward_ratio
)

# Section 2: Conditions de marché
market_conditions = opportunity.get("market_conditions", {})
market_conditions_text = "normales"

if market_conditions.get("details"):
    details = market_conditions.get("details", {})

    if details.get("adx", {}).get("strong_trend", False):
        if details.get("adx", {}).get("bearish_trend", False):
            market_conditions_text = "tendance baissière forte"
        else:
            market_conditions_text = "tendance haussière forte"
    elif details.get("bollinger", {}).get("high_volatility", False):
        market_conditions_text = "haute volatilité"
    else:
        market_conditions_text = "favorables pour un rebond"

# Extraire les indicateurs
indicators = opportunity.get("indicators", {})
rsi = indicators.get("rsi", 50)
adx = market_conditions.get("details", {}).get("adx", {}).get("value", 25)
atr = indicators.get("atr", 0.01)

explanation += " " + template["market_conditions"].format(
    market_conditions=market_conditions_text,
    rsi=rsi,
    adx=adx,
```

```
        atr=atr
    )

    # Section 3: Raisonnement d'entrée
    price_position = "sous la bande inférieure de Bollinger" if indicators.get("bollinger", {}).get("percent_b", 0.5) < 0 else "proche d'un support technique"

    candle_pattern = "une bougie de retournement"
    if "Mèche inférieure significative" in signals:
        candle_pattern = "une mèche inférieure significative indiquant un rejet des prix bas"
    elif "Chandelier haussier après chandelier baissier" in signals:
        candle_pattern = "un chandelier haussier après une série de chandeliers baissiers"

    volume_analysis = "Le volume est normal."
    if "Pic de volume haussier" in signals:
        volume_analysis = "Un pic de volume haussier a été détecté, indiquant un fort intérêt acheteur."

    divergence_analysis = ""
    if "Divergence haussière RSI détectée" in signals:
        divergence_analysis = "Une divergence haussière a été détectée entre le prix et le RSI, un signal fort de retournement. "

    explanation += " " + template["entry_reasoning"].format(
        price_position=price_position,
        candle_pattern=candle_pattern,
        volume_analysis=volume_analysis,
        divergence_analysis=divergence_analysis
    )

    return explanation
```

```
def generate_trade_result_explanation(self, trade_result: Dict) -> str:
    """
    Génère une explication pour le résultat d'un trade

    Args:
        trade_result: Résultat du trade

    Returns:
        Explication textuelle
    """
    # Déterminer si c'est un succès ou un échec
    pnl_percent = trade_result.get("pnl_percent", 0)
    is_success = pnl_percent > 0

    template_key = "success" if is_success else "failure"
    template = self.templates["trade_result"][template_key]

    # Extraire les données du trade
    symbol = trade_result.get("symbol", "UNKNOWN")
    pnl_absolute = trade_result.get("pnl_absolute", 0)
    entry_price = trade_result.get("entry_price", 0)
    exit_price = trade_result.get("exit_price", 0)
    exit_reason = trade_result.get("exit_reason", "Take-profit/Stop-loss")
    currency = "USDT"

    # Calculer la durée du trade
    entry_time = trade_result.get("entry_time")
    close_time = trade_result.get("close_time")

    if entry_time and close_time:
        if isinstance(entry_time, str):
```

```
        entry_time = datetime.fromisoformat(entry_time)
    if isinstance(close_time, str):
        close_time = datetime.fromisoformat(close_time)

    duration_seconds = (close_time - entry_time).total_seconds()

    if duration_seconds < 60:
        duration = f"{int(duration_seconds)} secondes"
    elif duration_seconds < 3600:
        duration = f"{int(duration_seconds/60)} minutes"
    else:
        duration = f"{duration_seconds/3600:.1f} heures"
else:
    duration = "inconnue"

# Générer la leçon à retenir pour les trades en échec
lesson = ""
if not is_success:
    if pnl_percent > -2:
        lesson = "La perte est minime, la stratégie reste valide"
    elif "stop_loss" in exit_reason.lower():
        lesson = "Revoir les critères d'entrée et les niveaux de stop-loss"
    else:
        lesson = "Analyser les signaux contradictoires et la vitesse de retournement du marché"

# Formater l'explication
explanation = template.format(
    symbol=symbol,
    pnl_percent=pnl_percent,
    pnl_absolute=pnl_absolute,
    currency=currency,
```

```
            duration=duration,
            entry_price=entry_price,
            exit_price=exit_price,
            exit_reason=exit_reason,
            lesson=lesson
        )

        return explanation
```

```
================================================
File: crypto_trading_bot_CLAUDE/ai/scoring_engine.py
================================================
# ai/scoring_engine.py
"""
Moteur de scoring pour évaluer les opportunités de trading
"""
import os
import json
import logging
import numpy as np
import pandas as pd
from typing import Dict, List, Optional, Union
from datetime import datetime, timedelta

from config.config import DATA_DIR
from config.trading_params import LEARNING_RATE
from utils.logger import setup_logger

logger = setup_logger("scoring_engine")

class ScoringEngine:
```

```
"""
Moteur de scoring qui évalue les opportunités de trading et s'améliore avec le temps
"""
def __init__(self):
    self.weights = {}
    self.history = []
    self.weights_file = os.path.join(DATA_DIR, "ai_weights.json")
    self.history_file = os.path.join(DATA_DIR, "scoring_history.json")

    # Charger les poids et l'historique
    self._load_weights()
    self._load_history()

    # Initialiser les poids par défaut si nécessaire
    self._initialize_default_weights()

def _load_weights(self) -> None:
    """
    Charge les poids depuis le fichier
    """
    if os.path.exists(self.weights_file):
        try:
            with open(self.weights_file, 'r') as f:
                self.weights = json.load(f)
            logger.info("Poids chargés avec succès")
        except Exception as e:
            logger.error(f"Erreur lors du chargement des poids: {str(e)}")
            self.weights = {}

def _load_history(self) -> None:
    """
```

```
    Charge l'historique de scoring depuis le fichier
    """
    if os.path.exists(self.history_file):
        try:
            with open(self.history_file, 'r') as f:
                self.history = json.load(f)
            logger.info(f"Historique chargé: {len(self.history)} entrées")
        except Exception as e:
            logger.error(f"Erreur lors du chargement de l'historique: {str(e)}")
            self.history = []

def _save_weights(self) -> None:
    """
    Sauvegarde les poids dans le fichier
    """
    try:
        with open(self.weights_file, 'w') as f:
            json.dump(self.weights, f, indent=2)
        logger.debug("Poids sauvegardés")
    except Exception as e:
        logger.error(f"Erreur lors de la sauvegarde des poids: {str(e)}")

def _save_history(self) -> None:
    try:
        # Limiter la taille de l'historique avant de sauvegarder
        if len(self.history) > 1000:
            self.history = self.history[-1000:]

        with open(self.history_file, 'w') as f:
            json.dump(self.history, f, indent=2, default=str)
        logger.debug("Historique sauvegardé")
```

```
    except Exception as e:
        logger.error(f"Erreur lors de la sauvegarde de l'historique: {str(e)}")

def _initialize_default_weights(self) -> None:
    """
    Initialise les poids par défaut si nécessaire
    """
    # Poids pour la stratégie de rebond technique
    if "technical_bounce" not in self.weights:
        self.weights["technical_bounce"] = {
            # Poids des signaux de rebond
        "rsi_oversold": 16,
        "rsi_turning_up": 12,
        "bollinger_below_lower": 16,
        "bollinger_returning": 12,
        "significant_lower_wick": 13,
        "bullish_candle_after_bearish": 10,
        "bullish_divergence": 21,
        "volume_spike": 12,
        "adx_weak_trend": 6,
        "no_strong_bearish_trend": 11,
        "ema_alignment_not_bearish": 9,
        "no_high_volatility": 6,
        "stop_loss_percent": -12,
        "risk_reward_ratio": 16
        }

        self._save_weights()
        logger.info("Poids par défaut initialisés pour la stratégie de rebond technique")

def calculate_score(self, data: Dict, strategy: str) -> Dict:
```

```
    """
    Calcule le score d'une opportunité de trading

    Args:
        data: Données pour le calcul du score
        strategy: Nom de la stratégie

    Returns:
        Dictionnaire avec le score et les détails
    """
    if strategy not in self.weights:
        logger.error(f"Stratégie non reconnue: {strategy}")
        return {"score": 0, "details": {}, "error": "Stratégie non reconnue"}

    # Calculer le score en fonction de la stratégie
    if strategy == "technical_bounce":
        return self._calculate_technical_bounce_score(data)

    return {"score": 0, "details": {}, "error": "Méthode de calcul non implémentée"}

def _calculate_technical_bounce_score(self, data: Dict) -> Dict:
    """
    Calcule le score pour la stratégie de rebond technique

    Args:
        data: Données pour le calcul du score

    Returns:
        Dictionnaire avec le score et les détails
    """
# Validation robuste des données d'entrée
```

```
if not isinstance(data, dict):
    logger.error("Format de données invalide pour le scoring")
    return {"score": 0, "details": {}, "error": "Format de données invalide"}

weights = self.weights.get("technical_bounce", {})
if not weights:
    logger.error("Poids non initialisés pour la stratégie de rebond technique")
    return {"score": 0, "details": {}, "error": "Poids non initialisés"}

score = 0
details = {}

    # Extraire les données
bounce_signals = data.get("bounce_signals", {})
market_state = data.get("market_state", {})
ohlcv = data.get("ohlcv", pd.DataFrame())
indicators = data.get("indicators", {})

# Valider les données
if not bounce_signals or not market_state or ohlcv.empty:
    return {"score": 0, "details": {}, "error": "Données insuffisantes"}

# 1. Évaluer les signaux de rebond
signals = bounce_signals.get("signals", [])
recent_market_performance = self._get_recent_market_performance(ohlcv)

signal_weight_multiplier = 1.0
if recent_market_performance < -5:  # Marché en forte baisse
    signal_weight_multiplier = 0.8  # Réduire l'importance des signaux haussiers
elif recent_market_performance > 5:  # Marché en forte hausse
    signal_weight_multiplier = 1.2  # Augmenter l'importance des signaux haussiers
```

```
# Application des poids pour chaque signal avec ajustement dynamique
if "RSI en zone de survente" in signals:
    weight = weights["rsi_oversold"] * signal_weight_multiplier
    score += weight
    details["rsi_oversold"] = weight

if "RSI remonte depuis la zone de survente" in signals:
    score += weights["rsi_turning_up"]
    details["rsi_turning_up"] = weights["rsi_turning_up"]

if "Prix sous la bande inférieure de Bollinger" in signals:
    score += weights["bollinger_below_lower"]
    details["bollinger_below_lower"] = weights["bollinger_below_lower"]

if "Prix remonte vers la bande inférieure" in signals:
    score += weights["bollinger_returning"]
    details["bollinger_returning"] = weights["bollinger_returning"]

if "Mèche inférieure significative (rejet)" in signals:
    score += weights["significant_lower_wick"]
    details["significant_lower_wick"] = weights["significant_lower_wick"]

if "Chandelier haussier après chandelier baissier" in signals:
    score += weights["bullish_candle_after_bearish"]
    details["bullish_candle_after_bearish"] = weights["bullish_candle_after_bearish"]

if "Divergence haussière RSI détectée" in signals:
    score += weights["bullish_divergence"]
    details["bullish_divergence"] = weights["bullish_divergence"]
```

```
        if "Pic de volume haussier" in signals:
            score += weights["volume_spike"]
            details["volume_spike"] = weights["volume_spike"]

        # 2. Évaluer les conditions de marché
        market_details = market_state.get("details", {})

        # ADX faible (pas de tendance forte)
        if "adx" in market_details and market_details["adx"].get("value", 100) < 25:
            score += weights["adx_weak_trend"]
            details["adx_weak_trend"] = weights["adx_weak_trend"]

        # Pas de forte tendance baissière
        if "adx" in market_details and not (market_details["adx"].get("strong_trend", False) and
market_details["adx"].get("bearish_trend", False)):
            score += weights["no_strong_bearish_trend"]
            details["no_strong_bearish_trend"] = weights["no_strong_bearish_trend"]

        # Alignement des EMA non baissier
        if "ema_alignment" in market_details and not
market_details["ema_alignment"].get("bearish_alignment", False):
            score += weights["ema_alignment_not_bearish"]
            details["ema_alignment_not_bearish"] = weights["ema_alignment_not_bearish"]

        # Volatilité non excessive
        if "bollinger" in market_details and not market_details["bollinger"].get("high_volatility", False):
            score += weights["no_high_volatility"]
            details["no_high_volatility"] = weights["no_high_volatility"]

        contradictory_signals = self._detect_contradictory_signals(signals, indicators)
        if contradictory_signals:
            penalty = -15  # Pénalité significative
```

```
    score += penalty
    details["contradictory_signals_penalty"] = penalty

# NOUVEAU: Bonus pour confirmation sur timeframes multiples
multi_tf_confirmation = data.get("multi_timeframe_confirmation", 0)
if multi_tf_confirmation > 0:
    bonus = multi_tf_confirmation * 5  # 5 points par timeframe confirmant
    score += bonus
    details["multi_timeframe_bonus"] = bonus
# 3. Évaluer la qualité de l'opportunité
entry_price = ohlcv["close"].iloc[-1]
stop_loss_price = entry_price * 0.97  # -3% par défaut
take_profit_price = entry_price * 1.06  # +6% par défaut

# Si les prix sont fournis dans les données
if "entry_price" in data and "stop_loss" in data and "take_profit" in data:
    entry_price = data["entry_price"]
    stop_loss_price = data["stop_loss"]
    take_profit_price = data["take_profit"]

# Calculer le pourcentage de stop-loss
stop_loss_percent = abs((entry_price - stop_loss_price) / entry_price * 100)

# Pénaliser les stop-loss trop larges
if stop_loss_percent > 5:
    penalty = weights["stop_loss_percent"] * (stop_loss_percent / 5)
    score += penalty  # Négatif
    details["stop_loss_penalty"] = penalty

# Calculer le ratio risque/récompense
risk = abs(entry_price - stop_loss_price)
```

```
reward = abs(take_profit_price - entry_price)
risk_reward_ratio = reward / risk if risk > 0 else 0

# Bonus pour un bon ratio risque/récompense
if risk_reward_ratio >= 1.5:
    bonus = weights["risk_reward_ratio"] * (risk_reward_ratio / 1.5)
    score += bonus
    details["risk_reward_bonus"] = bonus

# 4. Normaliser le score (0-100)
score = max(0, min(100, score))

# 5. Enregistrer le résultat dans l'historique
history_entry = {
    "timestamp": datetime.now().isoformat(),
    "strategy": "technical_bounce",
    "score": score,
    "details": details,
    "signals": signals,
    "trade_id": None,
    "market_context": {
        "trend": market_state.get("trend", "unknown"),
        "volatility": market_state.get("volatility", "medium")
    }
}
self.history.append(history_entry)
self._save_history()
if "RSI remonte depuis la zone de survente" in signals:
# Vérifier la force du rebond RSI
    rsi_current = indicators.get("rsi", pd.Series()).iloc[-1]
    rsi_prev = indicators.get("rsi", pd.Series()).iloc[-2]
```

```
            rsi_momentum = rsi_current - rsi_prev
            if rsi_momentum > 5:  # Forte accélération du RSI
                score += weights["rsi_turning_up"] * 1.5
                details["strong_rsi_momentum"] = weights["rsi_turning_up"] * 1.5

        # Donnez plus de poids au volume lors des rebonds
        if "Pic de volume haussier" in signals:
            volume_ratio = bounce_signals.get("volume_ratio", 1.0)
            if volume_ratio > 3.0:  # Volume exceptionnellement élevé
                bonus = weights["volume_spike"] * (volume_ratio / 2)
                score += bonus
                details["high_volume_bonus"] = bonus

        # Ces instructions doivent être en dehors du bloc conditionnel
        self.history.append(history_entry)
        self._save_history()

        return {"score": int(score), "details": details}

    def _get_recent_market_performance(self, ohlcv: pd.DataFrame) -> float:
        """
        Calcule la performance récente du marché (pourcentage de changement sur les derniers jours)
        """
        if len(ohlcv) < 10:
            return 0

        # Calculer la performance sur les 5 derniers jours
        recent_close = ohlcv['close'].iloc[-1]
        past_close = ohlcv['close'].iloc[-10]
```

```
        return ((recent_close / past_close) - 1) * 100

    def _detect_contradictory_signals(self, signals: List[str], indicators: Dict) -> bool:
        """
        Détecte les signaux contradictoires qui pourraient indiquer un faux signal
        """
        # Exemple: RSI en zone de survente mais ADX fort avec tendance baissière
        if "RSI en zone de survente" in signals and indicators.get("adx", {}).get("strong_trend", False) and indicators.get("adx", {}).get("bearish_trend", False):
            return True

        # Exemple: Signal de rebond mais volume en baisse
        if "Chandelier haussier après chandelier baissier" in signals and not "Pic de volume haussier" in signals:
            return True

        return False

    def update_trade_result(self, trade_id: str, trade_result: Dict) -> None:
        """
        Met à jour l'historique avec le résultat d'un trade et ajuste les poids

        Args:
            trade_id: ID du trade
            trade_result: Résultat du trade
        """
        # Rechercher l'entrée correspondante dans l'historique
        history_entry = None
        history_index = -1

        for i, entry in enumerate(reversed(self.history)):
            if entry.get("trade_id") == trade_id:
```

```
            history_entry = entry
            history_index = len(self.history) - 1 - i
            break

    if not history_entry:
        logger.warning(f"Entrée d'historique non trouvée pour le trade {trade_id}")
        return

    # Mettre à jour l'entrée avec le résultat
    self.history[history_index]["trade_result"] = trade_result
    self.history[history_index]["pnl_percent"] = trade_result.get("pnl_percent", 0)
    self.history[history_index]["pnl_absolute"] = trade_result.get("pnl_absolute", 0)

    # Ajuster les poids en fonction du résultat
    self._adjust_weights(history_index)

    # Sauvegarder l'historique et les poids
    self._save_history()
    self._save_weights()

def _adjust_weights(self, history_index: int) -> None:
    """
    Ajuste les poids en fonction du résultat d'un trade avec mémoire adaptative

    Args:
        history_index: Index de l'entrée d'historique
    """
    if history_index < 0 or history_index >= len(self.history):
        logger.error(f"Index d'historique invalide: {history_index}")
        return
```

```
history_entry = self.history[history_index]
strategy = history_entry.get("strategy")
pnl_percent = history_entry.get("pnl_percent", 0)

if strategy not in self.weights:
    logger.error(f"Stratégie non reconnue: {strategy}")
    return

# Ne pas ajuster les poids si le PnL est nul (trade non terminé)
if pnl_percent == 0:
    return

# Déterminer si le trade est un succès ou un échec
is_success = pnl_percent > 0

# Facteur d'ajustement basé sur la performance
# Plus le gain ou la perte est importante, plus l'ajustement est grand
adjustment_factor = abs(pnl_percent) / 5 * LEARNING_RATE
adjustment_factor = min(adjustment_factor, 0.1)  # Limiter l'ajustement à 10% maximum

# Facteur d'oubli pour les ajustements passés
forget_factor = 0.85

# Initialiser le dictionnaire des ajustements récents si nécessaire
if not hasattr(self, 'recent_adjustments'):
    self.recent_adjustments = {}

# Récupérer les détails du trade
details = history_entry.get("details", {})
weights = self.weights[strategy]
```

```
# Ajuster les poids en fonction du succès ou de l'échec
for factor, value in details.items():
    if factor in weights:
        # Récupérer l'ajustement précédent pour ce facteur (avec oubli)
        previous_adj = self.recent_adjustments.get(factor, 0) * forget_factor

        # Calculer l'ajustement actuel
        current_adj = adjustment_factor * (1 if is_success else -1)

        # Combiner les ajustements précédents et actuels
        total_adj = previous_adj + current_adj

        # Limiter l'ampleur totale de l'ajustement
        if abs(total_adj) > 0.15:
            total_adj = 0.15 if total_adj > 0 else -0.15

        # Appliquer l'ajustement au poids
        weights[factor] = weights[factor] * (1 + total_adj)

        # Mémoriser cet ajustement pour les prochaines itérations
        self.recent_adjustments[factor] = total_adj

        logger.debug(f"Poids ajusté pour {factor}: {weights[factor]:.2f} (ajustement: {total_adj:.3f})")

# Normaliser les poids pour éviter l'inflation ou la déflation
total_weight = sum(abs(w) for w in weights.values())
if total_weight > 0:
    scale_factor = 100 / total_weight
    for factor in weights:
        weights[factor] = weights[factor] * scale_factor
```

```
        logger.info(f"Poids ajustés pour la stratégie {strategy} (ajustement moyen: {adjustment_factor:.3f})")

        # Sauvegarder les poids mis à jour
        self._save_weights()


==========================================
File: crypto_trading_bot_CLAUDE/ai/trade_analyzer.py
==========================================
# ai/trade_analyzer.py
"""
Analyseur post-trade pour l'amélioration continue
"""
import os
import json
import logging
import pandas as pd
import numpy as np
from typing import Dict, List, Optional, Union
from datetime import datetime, timedelta

from config.config import DATA_DIR
from utils.logger import setup_logger

logger = setup_logger("trade_analyzer")

class TradeAnalyzer:
    """
    Analyse les trades passés pour identifier les patterns de succès et d'échec
    """
```

```
def __init__(self, scoring_engine, position_tracker):
    self.scoring_engine = scoring_engine
    self.position_tracker = position_tracker
    self.analysis_dir = os.path.join(DATA_DIR, "analysis")

    # Créer le répertoire si nécessaire
    if not os.path.exists(self.analysis_dir):
        os.makedirs(self.analysis_dir)

def analyze_recent_trades(self, days: int = 7) -> Dict:
    """
    Analyse les trades récents pour identifier les facteurs de succès et d'échec

    Args:
        days: Nombre de jours à analyser

    Returns:
        Rapport d'analyse
    """
    # Récupérer les trades fermés
    closed_positions = self.position_tracker.get_closed_positions(limit=1000)

    # Filtrer sur la période demandée
    start_date = datetime.now() - timedelta(days=days)

    filtered_positions = []
    for position in closed_positions:
        # Convertir la date de fermeture si elle est sous forme de chaîne
        close_time = position.get("close_time")
        if isinstance(close_time, str):
            try:
```

```
                close_time = datetime.fromisoformat(close_time)
            except:
                continue

        if close_time and close_time > start_date:
            filtered_positions.append(position)

    if not filtered_positions:
        logger.warning(f"Aucun trade fermé dans les {days} derniers jours")
        return {
            "success": False,
            "message": f"Aucun trade fermé dans les {days} derniers jours"
        }

    # Analyser les trades
    successful_trades = [p for p in filtered_positions if p.get("pnl_percent", 0) > 0]
    failed_trades = [p for p in filtered_positions if p.get("pnl_percent", 0) <= 0]

    # Calculer les statistiques générales
    total_trades = len(filtered_positions)
    success_rate = len(successful_trades) / total_trades * 100 if total_trades > 0 else 0

    avg_pnl = sum(p.get("pnl_percent", 0) for p in filtered_positions) / total_trades if total_trades > 0 else 0
    avg_win = sum(p.get("pnl_percent", 0) for p in successful_trades) / len(successful_trades) if successful_trades else 0
    avg_loss = sum(p.get("pnl_percent", 0) for p in failed_trades) / len(failed_trades) if failed_trades else 0

    # Calculer la durée moyenne des trades
    durations = []
    for position in filtered_positions:
```

```
    entry_time = position.get("entry_time")
    close_time = position.get("close_time")

    if entry_time and close_time:
        # Convertir en datetime si nécessaire
        if isinstance(entry_time, str):
            try:
                entry_time = datetime.fromisoformat(entry_time)
            except:
                continue

        if isinstance(close_time, str):
            try:
                close_time = datetime.fromisoformat(close_time)
            except:
                continue

        duration = (close_time - entry_time).total_seconds() / 60  # en minutes
        durations.append(duration)

avg_duration = sum(durations) / len(durations) if durations else 0

# Analyser les facteurs de succès
success_factors = self._analyze_success_factors(successful_trades, failed_trades)

# Calculer les performances par paire
performance_by_pair = {}

for position in filtered_positions:
    symbol = position.get("symbol", "UNKNOWN")
    pnl = position.get("pnl_percent", 0)
```

```
    if symbol not in performance_by_pair:
        performance_by_pair[symbol] = {
            "count": 0,
            "wins": 0,
            "losses": 0,
            "total_pnl": 0,
            "avg_pnl": 0
        }

    performance_by_pair[symbol]["count"] += 1
    performance_by_pair[symbol]["total_pnl"] += pnl

    if pnl > 0:
        performance_by_pair[symbol]["wins"] += 1
    else:
        performance_by_pair[symbol]["losses"] += 1

# Calculer les moyennes par paire
for symbol in performance_by_pair:
    stats = performance_by_pair[symbol]
    stats["avg_pnl"] = stats["total_pnl"] / stats["count"] if stats["count"] > 0 else 0
    stats["win_rate"] = stats["wins"] / stats["count"] * 100 if stats["count"] > 0 else 0

# Préparer le rapport
report = {
    "success": True,
    "period": f"{days} jours",
    "total_trades": total_trades,
    "success_rate": success_rate,
    "avg_pnl": avg_pnl,
```

```
            "avg_win": avg_win,
            "avg_loss": avg_loss,
            "avg_duration_minutes": avg_duration,
            "success_factors": success_factors,
            "performance_by_pair": performance_by_pair,
            "timestamp": datetime.now().isoformat()
        }

        # Sauvegarder le rapport
        filename = os.path.join(self.analysis_dir,
f"trade_analysis_{days}d_{datetime.now().strftime('%Y%m%d')}.json")

        try:
            with open(filename, 'w') as f:
                json.dump(report, f, indent=2, default=str)

            logger.info(f"Rapport d'analyse sauvegardé: {filename}")
        except Exception as e:
            logger.error(f"Erreur lors de la sauvegarde du rapport: {str(e)}")

        return report

    def _analyze_success_factors(self, successful_trades: List[Dict], failed_trades: List[Dict]) -> Dict:
        """
        Analyse les facteurs qui contribuent au succès ou à l'échec des trades

        Args:
            successful_trades: Liste des trades réussis
            failed_trades: Liste des trades échoués

        Returns:
```

```
    Analyse des facteurs de succès
"""
# Extraire les facteurs des trades réussis et échoués
success_factors = {}

# Analyser les raisons de réussite
if successful_trades:
    # Analyser le score moyen des trades réussis
    avg_score = sum(t.get("score", 0) for t in successful_trades) / len(successful_trades)
    success_factors["avg_score"] = avg_score

    # Analyser les signaux les plus fréquents dans les trades réussis
    signal_counts = {}
    for trade in successful_trades:
        signals = trade.get("signals", {}).get("signals", [])
        for signal in signals:
            if signal not in signal_counts:
                signal_counts[signal] = 0
            signal_counts[signal] += 1

    # Trier les signaux par fréquence
    sorted_signals = sorted(signal_counts.items(), key=lambda x: x[1], reverse=True)
    success_factors["top_signals"] = [{"signal": s[0], "count": s[1]} for s in sorted_signals[:5]]

    # Analyser les conditions de marché
    market_conditions = {}
    for trade in successful_trades:
        conditions = trade.get("market_conditions", {}).get("details", {})
        for key, value in conditions.items():
            if key not in market_conditions:
                market_conditions[key] = []
```

```
                market_conditions[key].append(value)

        # Calculer les moyennes des conditions de marché
        for key, values in market_conditions.items():
            if values and isinstance(values[0], (int, float)):
                market_conditions[key] = sum(values) / len(values)

        success_factors["market_conditions"] = market_conditions

    # Analyser les raisons d'échec
    failure_factors = {}

    if failed_trades:
        # Analyser le score moyen des trades échoués
        avg_score = sum(t.get("score", 0) for t in failed_trades) / len(failed_trades)
        failure_factors["avg_score"] = avg_score

        # Analyser les signaux les plus fréquents dans les trades échoués
        signal_counts = {}
        for trade in failed_trades:
            signals = trade.get("signals", {}).get("signals", [])
            for signal in signals:
                if signal not in signal_counts:
                    signal_counts[signal] = 0
                signal_counts[signal] += 1

        # Trier les signaux par fréquence
        sorted_signals = sorted(signal_counts.items(), key=lambda x: x[1], reverse=True)
        failure_factors["top_signals"] = [{"signal": s[0], "count": s[1]} for s in sorted_signals[:5]]

    # Comparer les facteurs de réussite et d'échec
```

```
    comparison = {}

    if successful_trades and failed_trades:
        # Comparer les scores
        score_diff = success_factors.get("avg_score", 0) - failure_factors.get("avg_score", 0)
        comparison["score_difference"] = score_diff

        # Identifier les signaux qui discriminent le mieux les trades réussis des trades échoués
        success_signal_freq = {s["signal"]: s["count"] / len(successful_trades) for s in success_factors.get("top_signals", [])}
        failure_signal_freq = {s["signal"]: s["count"] / len(failed_trades) for s in failure_factors.get("top_signals", [])}

        discriminating_signals = []

        for signal, success_freq in success_signal_freq.items():
            failure_freq = failure_signal_freq.get(signal, 0)
            if success_freq > failure_freq:
                discriminating_signals.append({
                    "signal": signal,
                    "success_freq": success_freq,
                    "failure_freq": failure_freq,
                    "difference": success_freq - failure_freq
                })

        # Trier par différence de fréquence
        discriminating_signals = sorted(discriminating_signals, key=lambda x: x["difference"], reverse=True)
        comparison["discriminating_signals"] = discriminating_signals[:5]

    return {
        "success_factors": success_factors,
```

```
        "failure_factors": failure_factors,
        "comparison": comparison
    }

def generate_recommendations(self) -> Dict:
    """
    Génère des recommandations pour améliorer la stratégie

    Returns:
        Dictionnaire avec les recommandations
    """
    # Analyser les trades récents
    analysis = self.analyze_recent_trades(days=30)

    if not analysis.get("success", False):
        return {
            "success": False,
            "message": "Impossible de générer des recommandations"
        }

    recommendations = {
        "success": True,
        "timestamp": datetime.now().isoformat(),
        "general_recommendations": [],
        "parameter_adjustments": [],
        "pair_recommendations": []
    }

    # Recommandations générales
    success_rate = analysis.get("success_rate", 0)
    avg_pnl = analysis.get("avg_pnl", 0)
```

```
if success_rate < 50:
    recommendations["general_recommendations"].append({
        "importance": "high",
        "recommendation": "Augmenter le seuil de score minimum pour entrer en position",
        "reasoning": f"Taux de réussite faible ({success_rate:.1f}%)"
    })

if avg_pnl < 0:
    recommendations["general_recommendations"].append({
        "importance": "high",
        "recommendation": "Réévaluer la stratégie de gestion des risques",
        "reasoning": f"P&L moyen négatif ({avg_pnl:.2f}%)"
    })

# Recommandations par paire
for symbol, stats in analysis.get("performance_by_pair", {}).items():
    if stats["count"] >= 5:  # Au moins 5 trades pour une analyse significative
        if stats["win_rate"] < 40:
            recommendations["pair_recommendations"].append({
                "pair": symbol,
                "recommendation": "Éviter de trader cette paire temporairement",
                "reasoning": f"Faible taux de réussite ({stats['win_rate']:.1f}%)"
            })
        elif stats["win_rate"] > 70:
            recommendations["pair_recommendations"].append({
                "pair": symbol,
                "recommendation": "Augmenter l'allocation sur cette paire",
                "reasoning": f"Taux de réussite élevé ({stats['win_rate']:.1f}%)"
            })
```

```
        # Recommandations sur les paramètres
        success_factors = analysis.get("success_factors", {}).get("comparison", {}).get("discriminating_signals", [])

        for factor in success_factors:
            signal = factor.get("signal", "")
            difference = factor.get("difference", 0)

            if difference > 0.3:  # Signal significativement plus fréquent dans les trades réussis
                if "RSI" in signal:
                    recommendations["parameter_adjustments"].append({
                        "parameter": "rsi_weight",
                        "recommendation": "Augmenter le poids du RSI dans le scoring",
                        "reasoning": f"Signal '{signal}' fortement associé aux trades réussis"
                    })
                elif "Bollinger" in signal:
                    recommendations["parameter_adjustments"].append({
                        "parameter": "bollinger_weight",
                        "recommendation": "Augmenter le poids des bandes de Bollinger dans le scoring",
                        "reasoning": f"Signal '{signal}' fortement associé aux trades réussis"
                    })
                elif "volume" in signal.lower():
                    recommendations["parameter_adjustments"].append({
                        "parameter": "volume_weight",
                        "recommendation": "Augmenter le poids des signaux de volume dans le scoring",
                        "reasoning": f"Signal '{signal}' fortement associé aux trades réussis"
                    })

        # Sauvegarder les recommandations
        filename = os.path.join(self.analysis_dir, f"recommendations_{datetime.now().strftime('%Y%m%d')}.json")
```

```
        try:
            with open(filename, 'w') as f:
                json.dump(recommendations, f, indent=2, default=str)

            logger.info(f"Recommandations sauvegardées: {filename}")
        except Exception as e:
            logger.error(f"Erreur lors de la sauvegarde des recommandations: {str(e)}")

        return recommendations



================================================
File: crypto_trading_bot_CLAUDE/ai/models/attention.py
================================================
"""
Implémentation de mécanismes d'attention avancés pour le modèle LSTM
Inspiré des architectures Transformer avec attention multi-tête
"""
import tensorflow as tf
from tensorflow.keras.layers import Layer, Dense, Reshape, Permute, Concatenate, TimeDistributed, Activation
from tensorflow.keras import backend as K
import numpy as np

class SelfAttention(Layer):
    """
    Mécanisme d'attention qui permet au modèle de se concentrer sur certaines parties d'une séquence
    """
    def __init__(self, attention_units=128, return_attention=False, **kwargs):
        """
        Initialise la couche d'attention
```

```
    Args:
        attention_units: Nombre d'unités dans la couche d'attention
        return_attention: Si True, retourne également les poids d'attention
    """
    self.attention_units = attention_units
    self.return_attention = return_attention
    super(SelfAttention, self).__init__(**kwargs)

def build(self, input_shape):
    """
    Construit les couches d'attention

    Args:
        input_shape: Forme de l'entrée
    """
    # Extraction des dimensions d'entrée
    self.time_steps = input_shape[1]
    self.input_dim = input_shape[2]

    # Initialisation des poids pour l'attention
    self.W1 = self.add_weight(name='W1',
                              shape=(self.input_dim, self.attention_units),
                              initializer='glorot_uniform',
                              trainable=True)

    self.W2 = self.add_weight(name='W2',
                              shape=(self.attention_units, 1),
                              initializer='glorot_uniform',
                              trainable=True)
```

```
    super(SelfAttention, self).build(input_shape)

def call(self, inputs, mask=None):
    """
    Applique le mécanisme d'attention

    Args:
        inputs: Entrée de forme (batch_size, time_steps, input_dim)
        mask: Masque optionnel

    Returns:
        Contexte pondéré par l'attention et poids d'attention si return_attention=True
    """
    # Calcul du score d'attention pour chaque pas de temps
    # et = tanh(W1 * ht)
    et = K.tanh(K.dot(inputs, self.W1))

    # at = softmax(W2 * et)
    at = K.dot(et, self.W2)
    at = K.squeeze(at, axis=-1)

    # Application du masque si nécessaire
    if mask is not None:
        at *= K.cast(mask, K.floatx())

    # Normalisation par softmax
    at = K.softmax(at)

    # Calcul du contexte pondéré par l'attention
    # context = at * inputs
    context = K.batch_dot(at, inputs)
```

```
        if self.return_attention:
            return [context, at]
        return context

    def compute_output_shape(self, input_shape):
        """
        Calcule la forme de la sortie

        Args:
            input_shape: Forme de l'entrée

        Returns:
            Forme de la sortie
        """
        if self.return_attention:
            return [(input_shape[0], self.input_dim), (input_shape[0], self.time_steps)]
        return (input_shape[0], self.input_dim)


class MultiHeadAttention(Layer):
    """
    Attention multi-tête pour capturer différents aspects des séquences temporelles
    Inspiré des architectures Transformer
    """
    def __init__(self, num_heads=4, head_dim=32, dropout=0.1, use_bias=True,
return_attention=False, **kwargs):
        """
        Initialise la couche d'attention multi-tête

        Args:
```

```
        num_heads: Nombre de têtes d'attention
        head_dim: Dimension de chaque tête
        dropout: Taux de dropout
        use_bias: Utiliser un terme de biais
        return_attention: Si True, retourne également les poids d'attention
    """
    super(MultiHeadAttention, self).__init__(**kwargs)
    self.num_heads = num_heads
    self.head_dim = head_dim
    self.dropout = dropout
    self.use_bias = use_bias
    self.return_attention = return_attention

def build(self, input_shape):
    """
    Construit les couches de l'attention multi-tête

    Args:
        input_shape: Forme de l'entrée (batch_size, time_steps, input_dim)
    """
    if isinstance(input_shape, list):
        # Si l'entrée est [query, key, value]
        q_shape, k_shape, v_shape = input_shape
        self.query_dim = q_shape[-1]
        self.key_dim = k_shape[-1]
        self.value_dim = v_shape[-1]
    else:
        # Si une seule entrée (self-attention)
        self.query_dim = input_shape[-1]
        self.key_dim = input_shape[-1]
        self.value_dim = input_shape[-1]
```

```
self.output_dim = self.num_heads * self.head_dim

# Matrices de projection pour query, key, value
self.query_weights = self.add_weight(
    name='query_weights',
    shape=(self.query_dim, self.num_heads * self.head_dim),
    initializer='glorot_uniform',
    trainable=True
)

if self.use_bias:
    self.query_bias = self.add_weight(
        name='query_bias',
        shape=(self.num_heads * self.head_dim,),
        initializer='zeros',
        trainable=True
    )

self.key_weights = self.add_weight(
    name='key_weights',
    shape=(self.key_dim, self.num_heads * self.head_dim),
    initializer='glorot_uniform',
    trainable=True
)

if self.use_bias:
    self.key_bias = self.add_weight(
        name='key_bias',
        shape=(self.num_heads * self.head_dim,),
        initializer='zeros',
```

```
        trainable=True
    )

self.value_weights = self.add_weight(
    name='value_weights',
    shape=(self.value_dim, self.num_heads * self.head_dim),
    initializer='glorot_uniform',
    trainable=True
)

if self.use_bias:
    self.value_bias = self.add_weight(
        name='value_bias',
        shape=(self.num_heads * self.head_dim,),
        initializer='zeros',
        trainable=True
    )

# Matrice de sortie pour combiner les têtes
self.output_weights = self.add_weight(
    name='output_weights',
    shape=(self.output_dim, self.value_dim),
    initializer='glorot_uniform',
    trainable=True
)

if self.use_bias:
    self.output_bias = self.add_weight(
        name='output_bias',
        shape=(self.value_dim,),
        initializer='zeros',
```

```
            trainable=True
        )

    super(MultiHeadAttention, self).build(input_shape)

def _split_heads(self, x, batch_size):
    """
    Divise la dernière dimension en (num_heads, head_dim)

    Args:
        x: Entrée de forme (batch_size, seq_len, num_heads * head_dim)
        batch_size: Taille du batch

    Returns:
        Sortie de forme (batch_size, num_heads, seq_len, head_dim)
    """
    x = tf.reshape(x, (batch_size, -1, self.num_heads, self.head_dim))
    return tf.transpose(x, perm=[0, 2, 1, 3])  # (batch_size, num_heads, seq_len, head_dim)

def _combine_heads(self, x, batch_size):
    """
    Combine les têtes pour former (batch_size, seq_len, num_heads * head_dim)

    Args:
        x: Entrée de forme (batch_size, num_heads, seq_len, head_dim)
        batch_size: Taille du batch

    Returns:
        Sortie de forme (batch_size, seq_len, num_heads * head_dim)
    """
    x = tf.transpose(x, perm=[0, 2, 1, 3])  # (batch_size, seq_len, num_heads, head_dim)
```

```
        return tf.reshape(x, (batch_size, -1, self.num_heads * self.head_dim))

    def scaled_dot_product_attention(self, q, k, v, mask=None):
        """
        Calcule l'attention avec produit scalaire mis à l'échelle

        Args:
            q: Query (batch_size, num_heads, seq_len_q, head_dim)
            k: Key (batch_size, num_heads, seq_len_k, head_dim)
            v: Value (batch_size, num_heads, seq_len_v, head_dim)
            mask: Masque optionnel

        Returns:
            Contexte et poids d'attention
        """
        # Produit scalaire entre query et key
        matmul_qk = tf.matmul(q, k, transpose_b=True)  # (batch_size, num_heads, seq_len_q, seq_len_k)

        # Mise à l'échelle
        dk = tf.cast(self.head_dim, tf.float32)
        scaled_attention_logits = matmul_qk / tf.math.sqrt(dk)

        # Application du masque si fourni
        if mask is not None:
            scaled_attention_logits += (mask * -1e9)

        # Softmax sur la dernière dimension (seq_len_k)
        attention_weights = tf.nn.softmax(scaled_attention_logits, axis=-1)

        # Application du dropout
```

```
    attention_weights = tf.keras.layers.Dropout(self.dropout)(attention_weights)

    # Produit avec les valeurs
    output = tf.matmul(attention_weights, v)  # (batch_size, num_heads, seq_len_q, head_dim)

    return output, attention_weights

def call(self, inputs, mask=None, training=None):
    """
    Applique l'attention multi-tête

    Args:
        inputs: Entrée ou liste [query, key, value]
        mask: Masque optionnel
        training: Indique si c'est l'entraînement

    Returns:
        Sortie avec attention et poids d'attention si return_attention=True
    """
    # Gestion de différents types d'entrées
    if isinstance(inputs, list):
        query, key, value = inputs
    else:
        query = key = value = inputs

    batch_size = tf.shape(query)[0]

    # Projections linéaires et division en têtes
    if self.use_bias:
        query_proj = tf.matmul(query, self.query_weights) + self.query_bias
        key_proj = tf.matmul(key, self.key_weights) + self.key_bias
```

```
            value_proj = tf.matmul(value, self.value_weights) + self.value_bias
        else:
            query_proj = tf.matmul(query, self.query_weights)
            key_proj = tf.matmul(key, self.key_weights)
            value_proj = tf.matmul(value, self.value_weights)

        # Division en têtes
        query_heads = self._split_heads(query_proj, batch_size)  # (batch_size, num_heads, seq_len_q, head_dim)
        key_heads = self._split_heads(key_proj, batch_size)      # (batch_size, num_heads, seq_len_k, head_dim)
        value_heads = self._split_heads(value_proj, batch_size)  # (batch_size, num_heads, seq_len_v, head_dim)

        # Attention avec produit scalaire mis à l'échelle
        attention_output, attention_weights = self.scaled_dot_product_attention(
            query_heads, key_heads, value_heads, mask)

        # Combinaison des têtes
        attention_output = self._combine_heads(attention_output, batch_size)  # (batch_size, seq_len_q, output_dim)

        # Projection finale
        if self.use_bias:
            output = tf.matmul(attention_output, self.output_weights) + self.output_bias
        else:
            output = tf.matmul(attention_output, self.output_weights)

        if self.return_attention:
            return [output, attention_weights]
        return output
```

```
    def compute_output_shape(self, input_shape):
        """
        Calcule la forme de sortie

        Args:
            input_shape: Forme de l'entrée

        Returns:
            Forme de la sortie
        """
        if isinstance(input_shape, list):
            q_shape = input_shape[0]
            v_shape = input_shape[2]
            output_shape = (q_shape[0], q_shape[1], v_shape[2])
        else:
            output_shape = (input_shape[0], input_shape[1], input_shape[2])

        if self.return_attention:
            if isinstance(input_shape, list):
                q_shape = input_shape[0]
                k_shape = input_shape[1]
                attention_shape = (q_shape[0], self.num_heads, q_shape[1], k_shape[1])
            else:
                attention_shape = (input_shape[0], self.num_heads, input_shape[1], input_shape[1])
            return [output_shape, attention_shape]

        return output_shape


class TemporalAttentionBlock(Layer):
    """
```

```
Bloc d'attention temporelle pour séries financières
Combine l'attention multi-tête avec une connexion résiduelle et normalisation
"""
def __init__(self, num_heads=4, head_dim=32, ff_dim=128, dropout=0.1, **kwargs):
    """
    Initialise le bloc d'attention temporelle

    Args:
        num_heads: Nombre de têtes d'attention
        head_dim: Dimension de chaque tête
        ff_dim: Dimension du feed-forward
        dropout: Taux de dropout
    """
    super(TemporalAttentionBlock, self).__init__(**kwargs)
    self.num_heads = num_heads
    self.head_dim = head_dim
    self.ff_dim = ff_dim
    self.dropout = dropout

def build(self, input_shape):
    """
    Construit les couches du bloc d'attention

    Args:
        input_shape: Forme de l'entrée
    """
    self.attention = MultiHeadAttention(
        num_heads=self.num_heads,
        head_dim=self.head_dim,
        dropout=self.dropout
    )
```

```
        self.ff1 = Dense(self.ff_dim, activation='relu')
        self.ff2 = Dense(input_shape[-1])

        self.dropout1 = tf.keras.layers.Dropout(self.dropout)
        self.dropout2 = tf.keras.layers.Dropout(self.dropout)

        self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
        self.layernorm2 = tf.keras.layers.LayerNormalization(epsilon=1e-6)

        super(TemporalAttentionBlock, self).build(input_shape)

    def call(self, inputs, training=None, mask=None):
        """
        Applique le bloc d'attention

        Args:
            inputs: Entrée de forme (batch_size, seq_len, features)
            training: Indique si c'est l'entraînement
            mask: Masque optionnel

        Returns:
            Sortie du bloc d'attention
        """
        # Sous-couche d'attention multi-tête
        attn_output = self.attention(inputs, mask=mask)
        attn_output = self.dropout1(attn_output, training=training)
        out1 = self.layernorm1(inputs + attn_output)  # Connexion résiduelle

        # Sous-couche feed-forward
        ff_output = self.ff1(out1)
```

```
        ff_output = self.ff2(ff_output)
        ff_output = self.dropout2(ff_output, training=training)

        # Connexion résiduelle finale
        return self.layernorm2(out1 + ff_output)


class TimeSeriesAttention(Layer):
    """
    Mécanisme d'attention spécialisé pour séries temporelles financières
    """
    def __init__(self, filters=64, kernel_size=1, **kwargs):
        """
        Initialise la couche d'attention

        Args:
            filters: Nombre de filtres convolutifs
            kernel_size: Taille du noyau convolutif
        """
        super(TimeSeriesAttention, self).__init__(**kwargs)
        self.filters = filters
        self.kernel_size = kernel_size

    def build(self, input_shape):
        """
        Construit les couches de l'attention

        Args:
            input_shape: Forme de l'entrée
        """
        # Extraction des dimensions
```

```
        self.time_steps = input_shape[1]
        self.input_dim = input_shape[2]

        # Couche de réduction de dimension temporelle
        self.conv_qkv = tf.keras.layers.Conv1D(
            filters=self.filters * 3,  # Pour query, key et value
            kernel_size=self.kernel_size,
            padding='same',
            use_bias=True
        )

        # Couche de sortie
        self.conv_out = tf.keras.layers.Conv1D(
            filters=self.input_dim,
            kernel_size=self.kernel_size,
            padding='same',
            use_bias=True
        )

        super(TimeSeriesAttention, self).build(input_shape)

    def call(self, inputs, mask=None):
        """
        Applique le mécanisme d'attention

        Args:
            inputs: Entrée de forme (batch_size, time_steps, input_dim)
            mask: Masque optionnel

        Returns:
            Sortie avec attention
```

```
        """
        # Projection QKV
        qkv = self.conv_qkv(inputs)

        # Séparation en query, key, value
        batch_size = tf.shape(qkv)[0]
        q, k, v = tf.split(qkv, 3, axis=-1)

        # Calcul du score d'attention
        # Produit scalaire de q et k, puis mise à l'échelle
        score = tf.matmul(q, k, transpose_b=True)
        scale = tf.sqrt(tf.cast(self.filters, tf.float32))
        score = score / scale

        # Application du masque si nécessaire
        if mask is not None:
            score += (1.0 - mask) * -1e9

        # Appliquer softmax pour obtenir les poids d'attention
        attention_weights = tf.nn.softmax(score, axis=-1)

        # Appliquer l'attention aux valeurs
        context = tf.matmul(attention_weights, v)

        # Projection finale
        output = self.conv_out(context)

        # Connexion résiduelle
        return inputs + output

================================================
```

File: crypto_trading_bot_CLAUDE/ai/models/continuous_learning.py

==========

"""

Système d'apprentissage continu avancé avec protection contre l'oubli catastrophique

et détection de concept drift pour l'adaptation automatique aux changements de marché

"""

```
import os
import numpy as np
import pandas as pd
import tensorflow as tf
import pickle
import json
from typing import Dict, List, Tuple, Union, Optional, Any
from datetime import datetime, timedelta
from collections import deque
import random
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from scipy.stats import mannwhitneyu, ks_2samp
import shutil

from config.config import DATA_DIR
from utils.logger import setup_logger

logger = setup_logger("advanced_continuous_learning")

class ReplayMemory:
    """
    Mémoire de rejeu avec échantillonnage prioritaire
    Stocke les exemples d'entraînement passés pour éviter l'oubli catastrophique
    """
```

```
def __init__(self, max_size: int = 10000, alpha: float = 0.6, beta: float = 0.4):
    """
    Initialise la mémoire de rejeu

    Args:
        max_size: Capacité maximale de la mémoire
        alpha: Facteur d'exposition pour le calcul des priorités (0 = échantillonnage uniforme)
        beta: Facteur de correction pour le biais de l'échantillonnage prioritaire
    """
    self.max_size = max_size
    self.memory = deque(maxlen=max_size)
    self.priorities = deque(maxlen=max_size)
    self.alpha = alpha
    self.beta = beta
    self.epsilon = 1e-6  # Petite valeur pour éviter les priorités nulles

    # Métadonnées pour les statistiques et le diagnostic
    self.insertion_timestamps = deque(maxlen=max_size)
    self.memory_clusters = {}  # Pour l'organisation par concept/régime

    # Pour suivre les distributions des caractéristiques
    self.feature_stats = {
        "means": {},
        "stds": {},
        "mins": {},
        "maxs": {}
    }

def add(self, experience: Tuple, priority: float = None) -> None:
    """
    Ajoute une expérience à la mémoire
```

```
    Args:
        experience: Tuple (X, y) d'un exemple d'entraînement
        priority: Priorité de l'exemple (si None, utilise la priorité maximale actuelle)
    """
    if priority is None:
        priority = max(self.priorities) if self.priorities else 1.0

    self.memory.append(experience)
    self.priorities.append(priority)
    self.insertion_timestamps.append(datetime.now().isoformat())

    # Mettre à jour les statistiques de caractéristiques si l'expérience contient des données
    if len(experience) > 0 and isinstance(experience[0], np.ndarray) and experience[0].size > 0:
        self._update_feature_stats(experience[0])

def sample(self, batch_size: int) -> List[Tuple]:
    """
    Échantillonne un batch d'expériences selon leur priorité

    Args:
        batch_size: Taille du batch à échantillonner

    Returns:
        Liste d'expériences échantillonnées et leurs indices
    """
    if len(self.memory) == 0:
        return []

    batch_size = min(batch_size, len(self.memory))
```

```
    # Calculer les probabilités d'échantillonnage selon les priorités
    priorities = np.array(self.priorities)
    probabilities = priorities ** self.alpha
    probabilities /= np.sum(probabilities)

    # Échantillonner les indices selon les probabilités
    indices = np.random.choice(len(self.memory), batch_size, replace=False, p=probabilities)

    # Calculer les poids d'importance pour la correction du biais
    weights = (len(self.memory) * probabilities[indices]) ** (-self.beta)
    weights /= np.max(weights)  # Normaliser à 1

    # Récupérer les expériences échantillonnées
    batch = [self.memory[i] for i in indices]

    return batch, indices, weights

def update_priorities(self, indices: List[int], errors: List[float]) -> None:
    """
    Met à jour les priorités des expériences en fonction des erreurs d'entraînement

    Args:
        indices: Indices des expériences à mettre à jour
        errors: Erreurs d'entraînement correspondantes
    """
    for i, idx in enumerate(indices):
        if idx < len(self.priorities):
            # Priorité = erreur + epsilon (pour éviter les priorités nulles)
            self.priorities[idx] = errors[i] + self.epsilon

def organize_by_clusters(self, n_clusters: int = 5) -> Dict:
```

```
"""
Organise la mémoire en clusters pour identifier différents régimes de marché

Args:
    n_clusters: Nombre de clusters à former

Returns:
    Dictionnaire des clusters
"""
if len(self.memory) < n_clusters * 10:
    return {}  # Pas assez de données pour le clustering

# Extraire les caractéristiques des expériences
features = []
for exp in self.memory:
    if len(exp) > 0 and isinstance(exp[0], np.ndarray):
        # Prendre la moyenne des caractéristiques temporelles
        features.append(np.mean(exp[0], axis=0).flatten())

if not features:
    return {}

features = np.array(features)

# Déterminer le nombre optimal de clusters si non spécifié
if n_clusters is None:
    n_clusters = self._find_optimal_clusters(features, max_clusters=10)

# Appliquer KMeans
kmeans = KMeans(n_clusters=n_clusters, random_state=42)
clusters = kmeans.fit_predict(features)
```

```
    # Organiser la mémoire par cluster
    self.memory_clusters = {i: [] for i in range(n_clusters)}

    for i, cluster_id in enumerate(clusters):
        if i < len(self.memory):
            self.memory_clusters[cluster_id].append(i)

    # Calculer les statistiques des clusters
    cluster_stats = {}
    for cluster_id, indices in self.memory_clusters.items():
        if indices:
            priorities = [self.priorities[i] for i in indices]
            timestamps = [self.insertion_timestamps[i] for i in indices]

            cluster_stats[cluster_id] = {
                "size": len(indices),
                "avg_priority": np.mean(priorities),
                "newest": min(timestamps),
                "oldest": max(timestamps)
            }

    return cluster_stats

def get_balanced_batch(self, batch_size: int, recency_weight: float = 0.3) -> List[Tuple]:
    """
    Échantillonne un batch équilibré qui combine expériences récentes et anciennes

    Args:
        batch_size: Taille du batch à échantillonner
        recency_weight: Poids pour les expériences récentes vs. diverses
```

```
Returns:
    Liste d'expériences échantillonnées
"""
if len(self.memory) == 0:
    return []

batch_size = min(batch_size, len(self.memory))

# Nombre d'expériences récentes et diverses
recent_count = int(batch_size * recency_weight)
diverse_count = batch_size - recent_count

# Échantillonner les expériences récentes
recent_indices = np.argsort([i for i in range(len(self.memory))])[-recent_count:]
recent_batch = [self.memory[i] for i in recent_indices]

# Échantillonner des expériences diverses selon les priorités
diverse_batch = []
if diverse_count > 0 and len(self.memory) > recent_count:
    # Exclure les échantillons récents déjà sélectionnés
    remaining_indices = [i for i in range(len(self.memory)) if i not in recent_indices]
    remaining_priorities = [self.priorities[i] for i in remaining_indices]

    # Calculer les probabilités
    probabilities = np.array(remaining_priorities) ** self.alpha
    probabilities /= np.sum(probabilities)

    # Échantillonner
    selected_indices = np.random.choice(
        remaining_indices,
```

```
            min(diverse_count, len(remaining_indices)),
            replace=False,
            p=probabilities
        )

        diverse_batch = [self.memory[i] for i in selected_indices]

    # Combiner et mélanger
    combined_batch = recent_batch + diverse_batch
    random.shuffle(combined_batch)

    return combined_batch

def _update_feature_stats(self, X: np.ndarray) -> None:
    """
    Met à jour les statistiques des caractéristiques

    Args:
        X: Entrée du modèle
    """
    # Si X est 3D (batch, sequence, features), réduire à 2D
    if X.ndim == 3:
        X_flat = X.reshape(-1, X.shape[-1])
    else:
        X_flat = X

    # Mettre à jour les statistiques
    for i in range(X_flat.shape[1]):
        feature_values = X_flat[:, i]

        if i not in self.feature_stats["means"]:
```

```
            self.feature_stats["means"][i] = []
            self.feature_stats["stds"][i] = []
            self.feature_stats["mins"][i] = []
            self.feature_stats["maxs"][i] = []

        self.feature_stats["means"][i].append(np.mean(feature_values))
        self.feature_stats["stds"][i].append(np.std(feature_values))
        self.feature_stats["mins"][i].append(np.min(feature_values))
        self.feature_stats["maxs"][i].append(np.max(feature_values))

        # Garder seulement les 100 dernières valeurs
        for key in ["means", "stds", "mins", "maxs"]:
            self.feature_stats[key][i] = self.feature_stats[key][i][-100:]

def _find_optimal_clusters(self, features: np.ndarray, max_clusters: int = 10) -> int:
    """
    Trouve le nombre optimal de clusters avec la méthode du score de silhouette

    Args:
        features: Caractéristiques à clusteriser
        max_clusters: Nombre maximum de clusters à tester

    Returns:
        Nombre optimal de clusters
    """
    if len(features) < max_clusters * 2:
        return max(2, len(features) // 5)

    silhouette_scores = []
    cluster_range = range(2, min(max_clusters, len(features) // 10) + 1)
```

```
    for n_clusters in cluster_range:
        try:
            kmeans = KMeans(n_clusters=n_clusters, random_state=42)
            cluster_labels = kmeans.fit_predict(features)
            silhouette_avg = silhouette_score(features, cluster_labels)
            silhouette_scores.append(silhouette_avg)
        except:
            silhouette_scores.append(-1)

    if not silhouette_scores or max(silhouette_scores) < 0:
        return 3  # Valeur par défaut

    return cluster_range[np.argmax(silhouette_scores)]

def save(self, filepath: str) -> None:
    """
    Sauvegarde la mémoire de rejeu sur disque

    Args:
        filepath: Chemin du fichier de sauvegarde
    """
    os.makedirs(os.path.dirname(filepath), exist_ok=True)

    # Préparer les données à sauvegarder
    save_data = {
        "memory": list(self.memory),
        "priorities": list(self.priorities),
        "insertion_timestamps": list(self.insertion_timestamps),
        "feature_stats": self.feature_stats,
        "alpha": self.alpha,
        "beta": self.beta,
```

```
            "max_size": self.max_size
        }

        try:
            with open(filepath, 'wb') as f:
                pickle.dump(save_data, f)
            logger.info(f"Mémoire de rejeu sauvegardée: {filepath}")
        except Exception as e:
            logger.error(f"Erreur lors de la sauvegarde de la mémoire de rejeu: {str(e)}")

    def load(self, filepath: str) -> bool:
        """
        Charge la mémoire de rejeu depuis le disque

        Args:
            filepath: Chemin du fichier de sauvegarde

        Returns:
            Succès du chargement
        """
        if not os.path.exists(filepath):
            logger.warning(f"Fichier de mémoire de rejeu non trouvé: {filepath}")
            return False

        try:
            with open(filepath, 'rb') as f:
                save_data = pickle.load(f)

            self.memory = deque(save_data["memory"], maxlen=save_data["max_size"])
            self.priorities = deque(save_data["priorities"], maxlen=save_data["max_size"])
```

```
            self.insertion_timestamps = deque(save_data["insertion_timestamps"],
maxlen=save_data["max_size"])
            self.feature_stats = save_data["feature_stats"]
            self.alpha = save_data["alpha"]
            self.beta = save_data["beta"]
            self.max_size = save_data["max_size"]

            logger.info(f"Mémoire de rejeu chargée: {filepath} ({len(self.memory)} exemples)")
            return True
        except Exception as e:
            logger.error(f"Erreur lors du chargement de la mémoire de rejeu: {str(e)}")
            return False

class ConceptDriftDetector:
    """
    Détecteur avancé de concept drift pour identifier les changements dans les données
    et signaler quand le modèle doit être adapté
    """
    def __init__(self, window_size: int = 100,
                 reference_size: int = 500,
                 threshold: float = 0.05,
                 min_samples: int = 30,
                 concept_history_size: int = 10):
        """
        Initialise le détecteur de concept drift

        Args:
            window_size: Taille de la fenêtre d'observation
            reference_size: Taille de la fenêtre de référence
            threshold: Seuil de p-valeur pour détecter une dérive
            min_samples: Nombre minimum d'échantillons pour la détection
```

```
        concept_history_size: Nombre de concepts précédents à conserver
    """
    self.window_size = window_size
    self.reference_size = reference_size
    self.threshold = threshold
    self.min_samples = min_samples

    # Fenêtres de données
    self.reference_window = []
    self.current_window = []

    # Historique des drifts détectés
    self.drift_history = []

    # Conservation des concepts précédents
    self.concept_history_size = concept_history_size
    self.concept_history = []  # Liste des références précédentes

    # Compteur de stabilité pour éviter les faux positifs
    self.stability_counter = 0
    self.required_stability = 3  # Nb de détections positives avant de signaler un drift

    # Métadonnées
    self.drift_count = 0
    self.last_drift_time = None
    self.total_observations = 0

def add_observation(self, features: Dict, prediction_error: float, timestamp: str = None) -> Dict:
    """
    Ajoute une observation et vérifie s'il y a une dérive conceptuelle
```

```
Args:
    features: Dictionnaire de caractéristiques observées
    prediction_error: Erreur de prédiction associée
    timestamp: Horodatage de l'observation

Returns:
    Résultat de la détection
"""
# Créer l'observation avec métadonnées
observation = {
    "features": features,
    "error": prediction_error,
    "timestamp": timestamp or datetime.now().isoformat(),
    "id": self.total_observations
}

# Incrémenter le compteur total
self.total_observations += 1

# Ajouter à la fenêtre courante
self.current_window.append(observation)

# Si la fenêtre courante est trop grande, supprimer les plus anciennes observations
if len(self.current_window) > self.window_size:
    self.current_window.pop(0)

# Si pas assez d'observations, ou pas de référence, pas de dérive
if len(self.current_window) < self.min_samples or not self.reference_window:
    return {
        "drift_detected": False,
        "p_value": None,
```

```
            "test_statistic": None,
            "message": "Données insuffisantes pour la détection"
        }

    # Détecter la dérive
    return self._detect_drift()

def initialize_reference(self, observations: List[Dict] = None) -> None:
    """
    Initialise la fenêtre de référence

    Args:
        observations: Liste d'observations pour la référence
    """
    if observations:
        # Utiliser les observations fournies
        self.reference_window = observations[-self.reference_size:] if len(observations) >
self.reference_size else observations.copy()
    elif len(self.current_window) >= self.reference_size:
        # Utiliser la fenêtre courante comme référence
        self.reference_window = self.current_window[-self.reference_size:].copy()
    else:
        logger.warning(f"Données insuffisantes pour initialiser la référence
({len(self.current_window)}/{self.reference_size})")
        self.reference_window = self.current_window.copy()

    logger.info(f"Fenêtre de référence initialisée avec {len(self.reference_window)} observations")

    # Réinitialiser le compteur de stabilité
    self.stability_counter = 0

def update_reference(self) -> None:
```

```
"""
Met à jour la fenêtre de référence avec les données actuelles
et conserve l'ancienne référence dans l'historique des concepts
"""
# Sauvegarder la référence actuelle dans l'historique des concepts
if self.reference_window:
    # Créer un résumé du concept
    concept_summary = self._create_concept_summary(self.reference_window)

    # Ajouter à l'historique
    self.concept_history.append({
        "window": self.reference_window.copy(),
        "summary": concept_summary,
        "start_time": self.reference_window[0]["timestamp"] if self.reference_window else None,
        "end_time": datetime.now().isoformat()
    })

    # Limiter la taille de l'historique
    if len(self.concept_history) > self.concept_history_size:
        self.concept_history.pop(0)

# Mettre à jour la référence
self.initialize_reference(self.current_window)

# Enregistrer le drift
self.drift_count += 1
self.last_drift_time = datetime.now().isoformat()

# Ajouter aux métadonnées historiques
self.drift_history.append({
    "timestamp": self.last_drift_time,
```

```
            "observation_count": self.total_observations
        })

        logger.info(f"Référence mise à jour après détection de drift (#{self.drift_count})")

    def _detect_drift(self) -> Dict:
        """
        Détecte si une dérive conceptuelle s'est produite

        Returns:
            Résultat de la détection
        """
        # Extraire les erreurs des deux fenêtres
        reference_errors = [obs["error"] for obs in self.reference_window]
        current_errors = [obs["error"] for obs in self.current_window]

        # Test statistique pour comparer les distributions (test de Mann-Whitney)
        try:
            stat, p_value = mannwhitneyu(reference_errors, current_errors, alternative='two-sided')

            # Test de Kolmogorov-Smirnov en complément
            ks_stat, ks_p_value = ks_2samp(reference_errors, current_errors)

            # Considérer qu'il y a dérive si l'une des p-valeurs est inférieure au seuil
            potential_drift = p_value < self.threshold or ks_p_value < self.threshold

            if potential_drift:
                self.stability_counter += 1
            else:
                self.stability_counter = max(0, self.stability_counter - 1)  # Réduire le compteur (mais pas en-dessous de 0)
```

```
# Dérive confirmée si plusieurs détections consécutives
drift_detected = self.stability_counter >= self.required_stability

# Caractériser la dérive
drift_magnitude = None
drift_direction = None

if drift_detected:
    # Calculer la magnitude et la direction de la dérive
    ref_mean = np.mean(reference_errors)
    cur_mean = np.mean(current_errors)

    drift_magnitude = abs(cur_mean - ref_mean) / max(ref_mean, 0.001)
    drift_direction = "worse" if cur_mean > ref_mean else "better"

    # Réinitialiser le compteur de stabilité pour les prochaines détections
    self.stability_counter = 0

return {
    "drift_detected": drift_detected,
    "potential_drift": potential_drift,
    "stability_counter": self.stability_counter,
    "p_value": float(p_value),
    "test_statistic": float(stat),
    "ks_p_value": float(ks_p_value),
    "ks_statistic": float(ks_stat),
    "magnitude": float(drift_magnitude) if drift_magnitude is not None else None,
    "direction": drift_direction,
    "reference_size": len(self.reference_window),
    "current_size": len(self.current_window)
```

```
            }

        except Exception as e:
            logger.error(f"Erreur lors du test de dérive: {str(e)}")
            return {
                "drift_detected": False,
                "error": str(e),
                "message": "Erreur lors du test statistique"
            }

    def check_for_concept_return(self) -> Dict:
        """
        Vérifie si les données actuelles correspondent à un concept précédemment observé

        Returns:
            Résultat de la vérification
        """
        if not self.concept_history or len(self.current_window) < self.min_samples:
            return {
                "concept_return": False,
                "message": "Historique des concepts vide ou données insuffisantes"
            }

        # Extraire les erreurs actuelles
        current_errors = [obs["error"] for obs in self.current_window]

        # Tester contre chaque concept historique
        best_match = None
        best_p_value = 0

        for i, concept in enumerate(self.concept_history):
```

```
            concept_errors = [obs["error"] for obs in concept["window"]]

            # Test statistique
            try:
                _, p_value = mannwhitneyu(concept_errors, current_errors, alternative='two-sided')

                # Si p-value élevée, les distributions sont similaires
                if p_value > 0.1 and p_value > best_p_value:
                    best_match = i
                    best_p_value = p_value
            except:
                continue

        if best_match is not None:
            matched_concept = self.concept_history[best_match]

            return {
                "concept_return": True,
                "concept_index": best_match,
                "p_value": float(best_p_value),
                "concept_start_time": matched_concept["start_time"],
                "concept_end_time": matched_concept["end_time"],
                "message": f"Retour au concept #{best_match}"
            }

        return {
            "concept_return": False,
            "message": "Aucun concept précédent ne correspond aux données actuelles"
        }

    def _create_concept_summary(self, window: List[Dict]) -> Dict:
```

```
"""
Crée un résumé statistique d'un concept

Args:
    window: Fenêtre d'observations

Returns:
    Résumé statistique
"""
errors = [obs["error"] for obs in window]

# Calculer les statistiques de base
summary = {
    "error_mean": float(np.mean(errors)),
    "error_std": float(np.std(errors)),
    "error_min": float(np.min(errors)),
    "error_max": float(np.max(errors)),
    "error_median": float(np.median(errors)),
    "sample_count": len(window)
}

# Extraire des statistiques sur les caractéristiques
feature_stats = {}

# Obtenir toutes les clés de caractéristiques
all_keys = set()
for obs in window:
    all_keys.update(obs["features"].keys())

# Calculer les statistiques pour chaque caractéristique
for key in all_keys:
```

```
        values = [obs["features"].get(key, np.nan) for obs in window]
        values = [v for v in values if not np.isnan(v)]

        if values:
            feature_stats[key] = {
                "mean": float(np.mean(values)),
                "std": float(np.std(values)),
                "min": float(np.min(values)),
                "max": float(np.max(values))
            }

    summary["feature_stats"] = feature_stats

    return summary

def get_drift_statistics(self) -> Dict:
    """
    Récupère des statistiques sur les drifts détectés

    Returns:
        Statistiques de drift
    """
    return {
        "total_drifts": self.drift_count,
        "last_drift_time": self.last_drift_time,
        "drift_history": self.drift_history,
        "total_observations": self.total_observations,
        "concept_history_size": len(self.concept_history)
    }

def save(self, filepath: str) -> None:
```

```
"""
Sauvegarde l'état du détecteur sur disque

Args:
    filepath: Chemin du fichier de sauvegarde
"""
os.makedirs(os.path.dirname(filepath), exist_ok=True)

save_data = {
    "reference_window": self.reference_window,
    "current_window": self.current_window,
    "drift_history": self.drift_history,
    "concept_history": self.concept_history,
    "drift_count": self.drift_count,
    "last_drift_time": self.last_drift_time,
    "total_observations": self.total_observations,
    "config": {
        "window_size": self.window_size,
        "reference_size": self.reference_size,
        "threshold": self.threshold,
        "min_samples": self.min_samples,
        "concept_history_size": self.concept_history_size
    }
}

try:
    with open(filepath, 'wb') as f:
        pickle.dump(save_data, f)
    logger.info(f"État du détecteur de concept drift sauvegardé: {filepath}")
except Exception as e:
    logger.error(f"Erreur lors de la sauvegarde du détecteur: {str(e)}")
```

```
def load(self, filepath: str) -> bool:
    """
    Charge l'état du détecteur depuis le disque

    Args:
        filepath: Chemin du fichier de sauvegarde

    Returns:
        Succès du chargement
    """
    if not os.path.exists(filepath):
        logger.warning(f"Fichier de détecteur non trouvé: {filepath}")
        return False

    try:
        with open(filepath, 'rb') as f:
            save_data = pickle.load(f)

        self.reference_window = save_data["reference_window"]
        self.current_window = save_data["current_window"]
        self.drift_history = save_data["drift_history"]
        self.concept_history = save_data["concept_history"]
        self.drift_count = save_data["drift_count"]
        self.last_drift_time = save_data["last_drift_time"]
        self.total_observations = save_data["total_observations"]

        # Charger la configuration si disponible
        if "config" in save_data:
            config = save_data["config"]
            self.window_size = config.get("window_size", self.window_size)
```

```
            self.reference_size = config.get("reference_size", self.reference_size)
            self.threshold = config.get("threshold", self.threshold)
            self.min_samples = config.get("min_samples", self.min_samples)
            self.concept_history_size = config.get("concept_history_size", self.concept_history_size)

        logger.info(f"État du détecteur de concept drift chargé: {filepath}")
        return True
    except Exception as e:
        logger.error(f"Erreur lors du chargement du détecteur: {str(e)}")
        return False

class AdvancedContinuousLearning:
    """
    Système d'apprentissage continu avancé qui adapte le modèle aux nouvelles données
    tout en évitant l'oubli catastrophique et en détectant les dérives conceptuelles
    """
    def __init__(self, model,
                 feature_engineering,
                 replay_memory_size: int = 10000,
                 drift_detection_window: int = 100,
                 drift_threshold: float = 0.05,
                 learning_enabled: bool = True,
                 regularization_strength: float = 0.01,
                 elastic_weight_consolidation: bool = True,
                 model_snapshot_interval: int = 5):
        """
        Initialise le système d'apprentissage continu

        Args:
            model: Modèle à adapter (LSTM ou autre)
            feature_engineering: Module d'ingénierie des caractéristiques
```

```
    replay_memory_size: Taille de la mémoire de rejeu
    drift_detection_window: Taille de la fenêtre pour la détection de drift
    drift_threshold: Seuil de détection de drift
    learning_enabled: Active ou désactive l'apprentissage continu
    regularization_strength: Force de la régularisation pour éviter l'oubli
    elastic_weight_consolidation: Utilise la consolidation élastique des poids
    model_snapshot_interval: Intervalle de prise de snapshots du modèle
"""
# Composants principaux
self.model = model
self.feature_engineering = feature_engineering
self.learning_enabled = learning_enabled
self.regularization_strength = regularization_strength
self.elastic_weight_consolidation = elastic_weight_consolidation

# Mémoire de rejeu avec échantillonnage prioritaire
self.replay_memory = ReplayMemory(
    max_size=replay_memory_size,
    alpha=0.6,  # Priorité non uniforme (0.6 = modérée)
    beta=0.4    # Correction d'échantillonnage
)

# Détecteur de concept drift
self.drift_detector = ConceptDriftDetector(
    window_size=drift_detection_window,
    reference_size=drift_detection_window * 5,
    threshold=drift_threshold,
    min_samples=30,
    concept_history_size=10
)
```

```
# Snapshots du modèle
self.model_snapshots = []
self.model_snapshot_interval = model_snapshot_interval
self.updates_since_snapshot = 0

# Indicateurs d'état
self.total_updates = 0
self.last_update_time = None
self.update_history = []

# Poids importants du modèle (pour EWC)
self.important_weights = None
self.fisher_information = None

# Métriques de performance
self.performance_metrics = {
   "loss_history": [],
   "accuracy_history": []
}

# Répertoires pour stockage
self.data_dir = os.path.join(DATA_DIR, "continuous_learning")
self.replay_memory_path = os.path.join(self.data_dir, "replay_memory.pkl")
self.drift_detector_path = os.path.join(self.data_dir, "drift_detector.pkl")
self.snapshots_dir = os.path.join(self.data_dir, "model_snapshots")

# Créer les répertoires
os.makedirs(self.data_dir, exist_ok=True)
os.makedirs(self.snapshots_dir, exist_ok=True)

# Charger l'état précédent si disponible
```

```
    self._load_state()

def process_new_data(self, data: pd.DataFrame, prediction_errors: List[float] = None,
                     min_samples: int = 30, max_batch_size: int = 64) -> Dict:
    """
    Traite de nouvelles données et met à jour le modèle si nécessaire

    Args:
        data: DataFrame avec les nouvelles données OHLCV
        prediction_errors: Erreurs de prédiction associées (optionnel)
        min_samples: Nombre minimum d'échantillons pour la mise à jour
        max_batch_size: Taille maximum des batchs pour la mise à jour

    Returns:
        Résultats du traitement
    """
    if not self.learning_enabled:
        return {
            "success": True,
            "updated": False,
            "message": "Apprentissage continu désactivé"
        }

    # 1. Prétraiter les données
    try:
        X, y = self._prepare_data(data)
    except Exception as e:
        logger.error(f"Erreur lors de la préparation des données: {str(e)}")
        return {
            "success": False,
            "updated": False,
```

```
            "error": str(e)
        }

    if len(X) == 0:
        return {
            "success": True,
            "updated": False,
            "message": "Pas de nouvelles données à traiter"
        }

    # 2. Ajouter les données à la mémoire de rejeu
    self._add_to_replay_memory(X, y, prediction_errors)

    # 3. Vérifier le concept drift
    drift_result = self._check_concept_drift(X, prediction_errors)
    drift_detected = drift_result.get("drift_detected", False)

    # 4. Déterminer si une mise à jour est nécessaire
    update_needed = drift_detected or (self.total_updates == 0)

    # 5. Mettre à jour le modèle si nécessaire
    if update_needed and len(self.replay_memory) >= min_samples:
        # Avant la mise à jour, si utilisation d'EWC, calculer l'importance des poids actuels
        if self.elastic_weight_consolidation and self.important_weights is None and self.model is not None:
            self._compute_weight_importance()

        # Effectuer la mise à jour
        update_result = self._update_model(max_batch_size=max_batch_size)

        if update_result["success"]:
```

```
        # Après une mise à jour réussie, prendre un snapshot si l'intervalle est atteint
        self.updates_since_snapshot += 1

        if self.updates_since_snapshot >= self.model_snapshot_interval:
            self._create_model_snapshot()
            self.updates_since_snapshot = 0

        # En cas de drift, mettre à jour la référence du détecteur
        if drift_detected:
            self.drift_detector.update_reference()

        # Sauvegarder l'état
        self._save_state()

        return {
            "success": True,
            "updated": True,
            "drift_detected": drift_detected,
            "drift_info": drift_result,
            "update_result": update_result
        }
    else:
        return {
            "success": False,
            "updated": False,
            "drift_detected": drift_detected,
            "drift_info": drift_result,
            "error": update_result.get("error", "Erreur inconnue lors de la mise à jour")
        }

return {
```

```
            "success": True,
            "updated": False,
            "drift_detected": drift_detected,
            "drift_info": drift_result,
            "message": "Pas de mise à jour nécessaire ou données insuffisantes"
        }

    def _prepare_data(self, data: pd.DataFrame) -> Tuple[np.ndarray, List[np.ndarray]]:
        """
        Prétraite les données pour l'apprentissage continu

        Args:
            data: DataFrame avec les données OHLCV

        Returns:
            Tuple (X, y) des données prétraitées
        """
        # Créer les caractéristiques
        featured_data = self.feature_engineering.create_features(
            data,
            include_time_features=True,
            include_price_patterns=True
        )

        # Normaliser les caractéristiques
        normalized_data = self.feature_engineering.scale_features(
            featured_data,
            is_training=False,  # Utiliser les scalers existants
            method='standard',
            feature_group='lstm'
        )
```

```
    # Créer les séquences d'entrée et les cibles
    if hasattr(self.model, 'horizon_periods'):
        # Pour le modèle LSTM avancé
        horizons = self.model.horizon_periods
    else:
        # Pour le modèle LSTM standard
        horizons = getattr(self.model, 'prediction_horizons', [12, 24, 96])

    # Créer les données avec le bon format
    X, y = self.feature_engineering.create_multi_horizon_data(
        normalized_data,
        sequence_length=getattr(self.model, 'input_length', 60),
        horizons=horizons,
        is_training=True
    )

    return X, y

def _add_to_replay_memory(self, X: np.ndarray, y: List[np.ndarray],
                          prediction_errors: List[float] = None) -> None:
    """
    Ajoute les nouvelles données à la mémoire de rejeu

    Args:
        X: Données d'entrée
        y: Données cibles
        prediction_errors: Erreurs de prédiction associées
    """
    # Si pas d'erreurs fournies, utiliser des priorités uniformes
    if prediction_errors is None:
```

```
            prediction_errors = [1.0] * len(X)

        # S'assurer que nous avons assez d'erreurs
        if len(prediction_errors) < len(X):
            prediction_errors = prediction_errors + [1.0] * (len(X) - len(prediction_errors))

        # Ajouter chaque exemple à la mémoire avec sa priorité
        for i in range(len(X)):
            example = (X[i:i+1], [y_arr[i:i+1] for y_arr in y])
            priority = max(0.1, prediction_errors[i])  # Assurer une priorité minimum

            self.replay_memory.add(example, priority)

        logger.info(f"Ajouté {len(X)} exemples à la mémoire de rejeu (taille: {len(self.replay_memory)})")

    def _check_concept_drift(self, X: np.ndarray, prediction_errors: List[float] = None) -> Dict:
        """
        Vérifie s'il y a un concept drift dans les nouvelles données

        Args:
            X: Données d'entrée
            prediction_errors: Erreurs de prédiction associées

        Returns:
            Résultat de la détection de drift
        """
        # Si pas d'erreurs fournies, utiliser des valeurs par défaut
        if prediction_errors is None or len(prediction_errors) == 0:
            # Faire des prédictions avec le modèle actuel pour obtenir les erreurs
            if self.model is not None:
                try:
```

```
                # Les prédictions dépendent du type de modèle
                predictions = self.model.model.predict(X)

                # Calculer l'erreur moyenne pour chaque exemple
                prediction_errors = []

                for i in range(len(X)):
                    # Calculer l'erreur moyenne sur tous les horizons et facteurs
                    sample_error = 0.0
                    count = 0

                    for p_idx, pred in enumerate(predictions):
                        if i < len(pred):
                            # MSE pour les sorties numériques, BCE pour les sorties binaires
                            if p_idx % 4 == 0:  # Direction (binaire)
                                y_true = pred[i][0]
                                error = -y_true * np.log(max(pred[i][0], 1e-10)) - (1 - y_true) * np.log(max(1 - pred[i][0], 1e-10))
                            else:  # Autres facteurs (régression)
                                error = (pred[i][0] - y_true) ** 2

                            sample_error += error
                            count += 1

                    if count > 0:
                        prediction_errors.append(sample_error / count)
                    else:
                        prediction_errors.append(1.0)

            except Exception as e:
                logger.error(f"Erreur lors du calcul des erreurs de prédiction: {str(e)}")
```

```
            prediction_errors = [1.0] * len(X)
    else:
        prediction_errors = [1.0] * len(X)

# Extraire des caractéristiques représentatives pour la détection de drift
drift_features = {}

if len(X) > 0:
    # Moyennes des caractéristiques d'entrée
    feature_means = np.mean(X, axis=(0, 1))

    for i, mean in enumerate(feature_means):
        drift_features[f"feature_{i}_mean"] = float(mean)

    # Statistiques supplémentaires
    drift_features["input_std"] = float(np.std(X))
    drift_features["input_range"] = float(np.max(X) - np.min(X))

# Détection pour chaque exemple
drift_detected = False
drift_details = {
    "observations_checked": len(prediction_errors),
    "drifts_in_window": 0
}

for i, error in enumerate(prediction_errors):
    # Ajouter l'observation au détecteur
    result = self.drift_detector.add_observation(
        features=drift_features,
        prediction_error=error,
        timestamp=datetime.now().isoformat()
```

```
            )

            # Vérifier si un drift a été détecté
            if result.get("drift_detected", False):
                drift_detected = True
                drift_details = result

                # Un seul drift suffit pour cette itération
                break

        return {
            "drift_detected": drift_detected,
            "details": drift_details
        }

    def _update_model(self, max_batch_size: int = 64, epochs: int = 5) -> Dict:
        """
        Met à jour le modèle avec les données de la mémoire de rejeu

        Args:
            max_batch_size: Taille maximum des batchs d'entraînement
            epochs: Nombre d'époques d'entraînement

        Returns:
            Résultat de la mise à jour
        """
        if self.model is None:
            return {
                "success": False,
                "error": "Modèle non initialisé"
            }
```

```
# Sauvegarde des poids actuels
original_weights = self.model.model.get_weights()

# 1. Préparation des callbacks pour l'entraînement
callbacks = [
    tf.keras.callbacks.EarlyStopping(
        monitor='loss',
        patience=3,
        restore_best_weights=True
    ),
    tf.keras.callbacks.ReduceLROnPlateau(
        monitor='loss',
        factor=0.5,
        patience=2,
        min_lr=1e-6
    )
]

# 2. Obtenir un batch équilibré de la mémoire de rejeu
batch_size = min(max_batch_size, len(self.replay_memory))
batch_examples = self.replay_memory.get_balanced_batch(
    batch_size=batch_size,
    recency_weight=0.7  # 70% d'exemples récents, 30% d'exemples divers
)

if not batch_examples:
    return {
        "success": False,
        "error": "Batch vide"
    }
```

```
        # 3. Préparation des données d'entraînement
        X_train = np.vstack([example[0] for example in batch_examples])
        y_train = []

        # Pour chaque sortie, combiner les exemples
        for output_idx in range(len(batch_examples[0][1])):
            y_output = np.vstack([example[1][output_idx] for example in batch_examples])
            y_train.append(y_output)

        # 4. Intégration d'EWC si activé
        if self.elastic_weight_consolidation and self.important_weights is not None and
self.fisher_information is not None:
            # Créer une fonction de perte personnalisée avec régularisation EWC
            original_loss = self.model.model.loss

            def ewc_loss_wrapper(original_loss_fn, lambda_reg, old_params, fisher):
                """Crée une fonction de perte avec régularisation EWC"""

                def ewc_loss(y_true, y_pred):
                    # Perte originale
                    loss = original_loss_fn(y_true, y_pred)

                    # Ajout de la régularisation EWC
                    ewc_reg = 0
                    model_params = self.model.model.trainable_weights

                    for i, (p, old_p) in enumerate(zip(model_params, old_params)):
                        f = fisher[i]
                        ewc_reg += tf.reduce_sum(f * tf.square(p - old_p))
```

```
                # Perte totale = perte originale + terme de régularisation
                return loss + (lambda_reg * ewc_reg)

            return ewc_loss

        # Appliquer la régularisation EWC
        custom_losses = []

        for loss_fn in original_loss:
            custom_losses.append(
                ewc_loss_wrapper(loss_fn, self.regularization_strength, self.important_weights, self.fisher_information)
            )

        # Recompiler le modèle avec les pertes personnalisées
        self.model.model.compile(
            optimizer=self.model.model.optimizer,
            loss=custom_losses,
            metrics=self.model.model.metrics
        )

        logger.info("Modèle recompilé avec régularisation EWC")

    # 5. Entraînement du modèle
    try:
        history = self.model.model.fit(
            x=X_train,
            y=y_train,
            epochs=epochs,
            batch_size=min(32, len(X_train)),
            callbacks=callbacks,
```

```
            verbose=1
        )

        # 6. Évaluation des performances
        eval_result = self.model.model.evaluate(X_train, y_train, verbose=0)

        # Calculer les métriques agrégées
        if isinstance(eval_result, list):
            avg_loss = np.mean(eval_result[:len(eval_result)//2])  # Première moitié = pertes
            avg_metric = np.mean(eval_result[len(eval_result)//2:])  # Seconde moitié = métriques
        else:
            avg_loss = eval_result
            avg_metric = 0

        # Ajouter aux métriques historiques
        self.performance_metrics["loss_history"].append(float(avg_loss))
        self.performance_metrics["accuracy_history"].append(float(avg_metric))

        # 7. Vérifier si la mise à jour a dégradé les performances
        if avg_loss > 1.5 * self.performance_metrics["loss_history"][-2] if
len(self.performance_metrics["loss_history"]) > 1 else False:
            # Dégradation significative, revenir aux poids originaux
            self.model.model.set_weights(original_weights)

            logger.warning(f"Mise à jour annulée: dégradation des performances (perte:
{avg_loss:.4f})")

            return {
                "success": False,
                "reverted": True,
                "message": "Mise à jour annulée due à une dégradation des performances",
                "old_loss": self.performance_metrics["loss_history"][-2],
```

```
            "new_loss": avg_loss
        }

    # 8. Mise à jour des compteurs
    self.total_updates += 1
    self.last_update_time = datetime.now().isoformat()

    self.update_history.append({
        "timestamp": self.last_update_time,
        "samples": len(X_train),
        "loss": float(avg_loss),
        "accuracy": float(avg_metric),
        "epochs": len(history.history["loss"])
    })

    # 9. Mettre à jour l'importance des poids si EWC est activé
    if self.elastic_weight_consolidation:
        self._compute_weight_importance()

    return {
        "success": True,
        "loss": float(avg_loss),
        "accuracy": float(avg_metric),
        "epochs": len(history.history["loss"]),
        "samples": len(X_train),
        "timestamp": self.last_update_time
    }

except Exception as e:
    # En cas d'erreur, restaurer les poids originaux
    self.model.model.set_weights(original_weights)
```

```
            logger.error(f"Erreur lors de la mise à jour du modèle: {str(e)}")

            return {
                "success": False,
                "error": str(e)
            }

    def _compute_weight_importance(self, samples_for_fisher: int = 64) -> None:
        """
        Calcule l'importance des poids actuels pour la consolidation élastique (EWC)

        Args:
            samples_for_fisher: Nombre d'échantillons pour estimer la matrice de Fisher
        """
        if len(self.replay_memory) < samples_for_fisher:
            logger.warning(f"Pas assez d'exemples pour calculer l'importance des poids ({len(self.replay_memory)}/{samples_for_fisher})")
            return

        # Stocker les poids actuels comme importants
        self.important_weights = self.model.model.get_weights()

        # Échantillonner des exemples pour calculer la matrice de Fisher
        batch = self.replay_memory.get_balanced_batch(batch_size=samples_for_fisher)

        if not batch:
            logger.warning("Impossible d'obtenir un batch pour le calcul de la matrice de Fisher")
            return

        # Préparer les échantillons
```

```
X_samples = np.vstack([example[0] for example in batch])
y_samples = []

for output_idx in range(len(batch[0][1])):
    y_output = np.vstack([example[1][output_idx] for example in batch])
    y_samples.append(y_output)

# Calculer la matrice de Fisher
logger.info("Calcul de la matrice de Fisher pour EWC")

try:
    # Version simplifiée de l'estimation de Fisher
    # (Dans une implémentation complète, on calculerait le gradient au carré)
    self.fisher_information = []

    for param in self.important_weights:
        # Initialiser avec une petite valeur pour éviter les divisions par zéro
        fisher_diag = np.ones_like(param) * 1e-5
        self.fisher_information.append(fisher_diag)

    # Utilisez la magnitude des gradients comme proxy pour l'information de Fisher
    with tf.GradientTape() as tape:
        # Calculer les sorties du modèle
        predictions = self.model.model(X_samples)

        # Calculer une perte combinée
        total_loss = 0

        for i, y_pred in enumerate(predictions):
            # Utiliser le bon type de perte selon le type de sortie
            if i % 4 == 0:  # Direction (binaire)
```

```
                loss = tf.keras.losses.binary_crossentropy(y_samples[i], y_pred)
            else:  # Autres facteurs (régression)
                loss = tf.keras.losses.mean_squared_error(y_samples[i], y_pred)

            total_loss += tf.reduce_mean(loss)

        # Calculer les gradients
        gradients = tape.gradient(total_loss, self.model.model.trainable_weights)

        # Utiliser le carré des gradients comme approximation de Fisher
        for i, grad in enumerate(gradients):
            if i < len(self.fisher_information):
                # Prendre le carré des gradients et moyenner sur les échantillons
                square_grad = tf.square(grad).numpy()
                self.fisher_information[i] += square_grad

        logger.info("Matrice de Fisher calculée avec succès")

    except Exception as e:
        logger.error(f"Erreur lors du calcul de la matrice de Fisher: {str(e)}")

        # Fallback: utiliser une matrice identité avec une petite valeur
        self.fisher_information = []
        for param in self.important_weights:
            self.fisher_information.append(np.ones_like(param) * 0.1)

def _create_model_snapshot(self) -> None:
    """Crée un snapshot du modèle actuel"""
    if self.model is None:
        return
```

```
        # Créer un identifiant unique pour le snapshot
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        snapshot_id = f"snapshot_{self.total_updates}_{timestamp}"

        # Chemin du snapshot
        snapshot_path = os.path.join(self.snapshots_dir, f"{snapshot_id}.h5")

        try:
            # Sauvegarder le modèle
            self.model.model.save(snapshot_path)

            # Stocker les métadonnées du snapshot
            snapshot_info = {
                "id": snapshot_id,
                "path": snapshot_path,
                "timestamp": timestamp,
                "update_count": self.total_updates,
                "metrics": {
                    "loss": self.performance_metrics["loss_history"][-1] if
self.performance_metrics["loss_history"] else None,
                    "accuracy": self.performance_metrics["accuracy_history"][-1] if
self.performance_metrics["accuracy_history"] else None
                }
            }

            self.model_snapshots.append(snapshot_info)

            # Limiter le nombre de snapshots (garder les 10 plus récents)
            if len(self.model_snapshots) > 10:
                # Supprimer le snapshot le plus ancien
                old_snapshot = self.model_snapshots.pop(0)
```

```
            if os.path.exists(old_snapshot["path"]):
                os.remove(old_snapshot["path"])

        logger.info(f"Snapshot du modèle créé: {snapshot_path}")

        # Sauvegarder les métadonnées des snapshots
        self._save_snapshot_metadata()

    except Exception as e:
        logger.error(f"Erreur lors de la création du snapshot: {str(e)}")

def _save_snapshot_metadata(self) -> None:
    """Sauvegarde les métadonnées des snapshots"""
    metadata_path = os.path.join(self.snapshots_dir, "snapshots_metadata.json")

    try:
        with open(metadata_path, 'w') as f:
            json.dump(self.model_snapshots, f, indent=2)
    except Exception as e:
        logger.error(f"Erreur lors de la sauvegarde des métadonnées des snapshots: {str(e)}")

def _load_snapshot_metadata(self) -> None:
    """Charge les métadonnées des snapshots"""
    metadata_path = os.path.join(self.snapshots_dir, "snapshots_metadata.json")

    if not os.path.exists(metadata_path):
        return

    try:
        with open(metadata_path, 'r') as f:
            self.model_snapshots = json.load(f)
```

```
    except Exception as e:
        logger.error(f"Erreur lors du chargement des métadonnées des snapshots: {str(e)}")

def restore_from_snapshot(self, snapshot_id: str = None) -> bool:
    """
    Restaure le modèle à partir d'un snapshot

    Args:
        snapshot_id: ID du snapshot à restaurer (le plus récent si None)

    Returns:
        Succès de la restauration
    """
    if not self.model_snapshots:
        logger.warning("Aucun snapshot disponible")
        return False

    # Déterminer le snapshot à utiliser
    if snapshot_id is None:
        # Utiliser le snapshot le plus récent
        snapshot = self.model_snapshots[-1]
    else:
        # Rechercher le snapshot par ID
        snapshot = next((s for s in self.model_snapshots if s["id"] == snapshot_id), None)

        if snapshot is None:
            logger.warning(f"Snapshot non trouvé: {snapshot_id}")
            return False

    # Vérifier si le fichier existe
    if not os.path.exists(snapshot["path"]):
```

```
            logger.warning(f"Fichier de snapshot non trouvé: {snapshot['path']}")
            return False

        try:
            # Charger le modèle depuis le snapshot
            self.model.model = tf.keras.models.load_model(snapshot["path"])

            logger.info(f"Modèle restauré depuis le snapshot: {snapshot['id']}")

            # Réinitialiser EWC après restauration
            self.important_weights = None
            self.fisher_information = None

            return True
        except Exception as e:
            logger.error(f"Erreur lors de la restauration du snapshot: {str(e)}")
            return False

    def _save_state(self) -> None:
        """Sauvegarde l'état du système d'apprentissage continu"""
        try:
            # Sauvegarder la mémoire de rejeu
            self.replay_memory.save(self.replay_memory_path)

            # Sauvegarder le détecteur de concept drift
            self.drift_detector.save(self.drift_detector_path)

            # Sauvegarder les métadonnées d'état
            state_metadata = {
                "total_updates": self.total_updates,
                "last_update_time": self.last_update_time,
```

```
            "update_history": self.update_history,
            "performance_metrics": self.performance_metrics,
            "learning_enabled": self.learning_enabled,
            "regularization_strength": self.regularization_strength,
            "elastic_weight_consolidation": self.elastic_weight_consolidation,
            "updates_since_snapshot": self.updates_since_snapshot
        }

        metadata_path = os.path.join(self.data_dir, "continuous_learning_state.json")

        with open(metadata_path, 'w') as f:
            json.dump(state_metadata, f, indent=2, default=str)

        logger.info("État du système d'apprentissage continu sauvegardé")
    except Exception as e:
        logger.error(f"Erreur lors de la sauvegarde de l'état: {str(e)}")

def _load_state(self) -> None:
    """Charge l'état du système d'apprentissage continu"""
    try:
        # Charger la mémoire de rejeu
        if os.path.exists(self.replay_memory_path):
            self.replay_memory.load(self.replay_memory_path)

        # Charger le détecteur de concept drift
        if os.path.exists(self.drift_detector_path):
            self.drift_detector.load(self.drift_detector_path)

        # Charger les métadonnées des snapshots
        self._load_snapshot_metadata()
```

```
            # Charger les métadonnées d'état
            metadata_path = os.path.join(self.data_dir, "continuous_learning_state.json")

            if os.path.exists(metadata_path):
                with open(metadata_path, 'r') as f:
                    state_metadata = json.load(f)

                self.total_updates = state_metadata.get("total_updates", 0)
                self.last_update_time = state_metadata.get("last_update_time")
                self.update_history = state_metadata.get("update_history", [])
                self.performance_metrics = state_metadata.get("performance_metrics", {
                    "loss_history": [],
                    "accuracy_history": []
                })
                self.learning_enabled = state_metadata.get("learning_enabled", self.learning_enabled)
                self.regularization_strength = state_metadata.get("regularization_strength",
self.regularization_strength)
                self.elastic_weight_consolidation = state_metadata.get("elastic_weight_consolidation",
self.elastic_weight_consolidation)
                self.updates_since_snapshot = state_metadata.get("updates_since_snapshot", 0)

                logger.info("État du système d'apprentissage continu chargé")
        except Exception as e:
            logger.error(f"Erreur lors du chargement de l'état: {str(e)}")

    def get_status(self) -> Dict:
        """
        Récupère l'état courant du système d'apprentissage continu

        Returns:
            Dictionnaire avec l'état courant
        """
```

```
        return {
            "enabled": self.learning_enabled,
            "total_updates": self.total_updates,
            "last_update_time": self.last_update_time,
            "replay_memory_size": len(self.replay_memory),
            "drift_statistics": self.drift_detector.get_drift_statistics(),
            "snapshots": len(self.model_snapshots),
            "ewc_active": self.elastic_weight_consolidation and self.important_weights is not None,
            "regularization_strength": self.regularization_strength,
            "performance": {
                "current_loss": self.performance_metrics["loss_history"][-1] if
self.performance_metrics["loss_history"] else None,
                "loss_trend": "improving" if (len(self.performance_metrics["loss_history"]) > 1 and
                                              self.performance_metrics["loss_history"][-1] <
self.performance_metrics["loss_history"][-2]) else "steady"
            }
        }


def test_system():
    """Fonction pour tester le système d'apprentissage continu"""
    # Créer un modèle de test
    from tensorflow.keras.models import Sequential
    from tensorflow.keras.layers import Dense

    model = Sequential([
        Dense(10, activation='relu', input_shape=(5,)),
        Dense(5, activation='relu'),
        Dense(1, activation='sigmoid')
    ])

    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

```
    # Créer une classe fictive pour simuler le feature engineering
    class MockFeatureEngineering:
        def create_features(self, data, **kwargs):
            return data

        def scale_features(self, data, **kwargs):
            return data

        def create_multi_horizon_data(self, data, **kwargs):
            # Simuler des données de séquence
            X = np.random.randn(10, 60, 5)
            y = [np.random.randint(0, 2, (10, 1)) for _ in range(4)]
            return X, y

    # Créer le système
    cl_system = AdvancedContinuousLearning(
        model=model,
        feature_engineering=MockFeatureEngineering(),
        replay_memory_size=1000,
        drift_detection_window=50
    )

    # Afficher l'état initial
    print("État initial:")
    print(cl_system.get_status())

    return cl_system

if __name__ == "__main__":
    test_system()
```

```
================================================
File: crypto_trading_bot_CLAUDE/ai/models/feature_engineering.py
================================================
# ai/models/feature_engineering.py
"""
Module d'ingénierie des caractéristiques pour le modèle LSTM
Prépare les données brutes pour l'entraînement et la prédiction
"""
import numpy as np
import pandas as pd
from typing import Dict, List, Tuple, Union, Optional
from datetime import datetime, timedelta
import talib
from sklearn.preprocessing import MinMaxScaler, StandardScaler
import pickle
import os

from indicators.trend import calculate_ema, calculate_adx, calculate_macd
from indicators.momentum import calculate_rsi, calculate_stochastic
from indicators.volatility import calculate_bollinger_bands, calculate_atr
from indicators.volume import calculate_obv, calculate_vwap
from config.config import DATA_DIR
from utils.logger import setup_logger

logger = setup_logger("feature_engineering")

class FeatureEngineering:
    """
    Classe pour la création et transformation des caractéristiques
    pour l'entraînement du modèle LSTM
    """
```

```
def __init__(self, save_scalers: bool = True):
    """
    Initialise le module d'ingénierie des caractéristiques

    Args:
        save_scalers: Indique s'il faut sauvegarder les scaler pour réutilisation
    """
    self.save_scalers = save_scalers
    self.scalers = {}
    self.scalers_path = os.path.join(DATA_DIR, "models", "scalers")

    # Créer le répertoire pour les scalers si nécessaire
    if save_scalers and not os.path.exists(self.scalers_path):
        os.makedirs(self.scalers_path, exist_ok=True)

def create_features(self, data: pd.DataFrame,
                    include_time_features: bool = True,
                    include_price_patterns: bool = True) -> pd.DataFrame:
    """
    Crée des caractéristiques avancées à partir des données OHLCV

    Args:
        data: DataFrame avec au moins les colonnes OHLCV (open, high, low, close, volume)
        include_time_features: Inclure les caractéristiques temporelles
        include_price_patterns: Inclure la détection des patterns de prix

    Returns:
        DataFrame enrichi avec les caractéristiques créées
    """
    # Vérifier que les colonnes requises sont présentes
    required_columns = ['open', 'high', 'low', 'close', 'volume']
```

```
    if not all(col in data.columns for col in required_columns):
        raise ValueError(f"Colonnes requises manquantes. Nécessite: {required_columns}")

    # Copier le DataFrame pour éviter de modifier l'original
    df = data.copy()

    # Assurer que l'index est un DatetimeIndex pour les caractéristiques temporelles
    if include_time_features and not isinstance(df.index, pd.DatetimeIndex):
        if 'timestamp' in df.columns:
            df['timestamp'] = pd.to_datetime(df['timestamp'])
            df.set_index('timestamp', inplace=True)
        else:
            logger.warning("Impossible de créer des caractéristiques temporelles sans colonne timestamp")
            include_time_features = False

    # 1. Indicateurs de tendance
    # EMA à différentes périodes
    ema_periods = [9, 21, 50, 200]
    emas = calculate_ema(df, ema_periods)
    for period, ema_series in emas.items():
        df[f'{period}'] = ema_series

    # Distances relatives aux EMAs
    for period in ema_periods:
        df[f'dist_to_ema_{period}'] = (df['close'] - df[f'ema_{period}']) / df[f'ema_{period}'] * 100

    # MACD
    macd_data = calculate_macd(df)
    df['macd'] = macd_data['macd']
    df['macd_signal'] = macd_data['signal']
```

```
df['macd_hist'] = macd_data['histogram']

# ADX (force de tendance)
adx_data = calculate_adx(df)
df['adx'] = adx_data['adx']
df['plus_di'] = adx_data['plus_di']
df['minus_di'] = adx_data['minus_di']

# 2. Indicateurs de momentum
# RSI
df['rsi'] = calculate_rsi(df)

# Stochastique
stoch_data = calculate_stochastic(df)
df['stoch_k'] = stoch_data['k']
df['stoch_d'] = stoch_data['d']

# Rate of Change (ROC) à différentes périodes
for period in [5, 10, 21]:
    df[f'roc_{period}'] = df['close'].pct_change(period) * 100

# 3. Indicateurs de volatilité
# Bandes de Bollinger
bb_data = calculate_bollinger_bands(df)
df['bb_upper'] = bb_data['upper']
df['bb_middle'] = bb_data['middle']
df['bb_lower'] = bb_data['lower']
df['bb_width'] = bb_data['bandwidth']
df['bb_percent_b'] = bb_data['percent_b']

# ATR (Average True Range)
```

```
df['atr'] = calculate_atr(df)
df['atr_percent'] = df['atr'] / df['close'] * 100  # ATR relatif au prix

# 4. Indicateurs de volume
# OBV (On-Balance Volume)
df['obv'] = calculate_obv(df)

# Volume relatif (comparé à la moyenne)
for period in [5, 10, 21]:
    df[f'rel_volume_{period}'] = df['volume'] / df['volume'].rolling(period).mean()

# VWAP (Volume-Weighted Average Price)
df['vwap'] = calculate_vwap(df)
df['vwap_dist'] = (df['close'] - df['vwap']) / df['vwap'] * 100  # Distance au VWAP

# 5. Caractéristiques de prix
# Rendements à différentes périodes
for period in [1, 3, 5, 10]:
    df[f'return_{period}'] = df['close'].pct_change(period) * 100

# Caractéristiques des chandeliers
df['body_size'] = abs(df['close'] - df['open'])
df['body_size_percent'] = df['body_size'] / df['open'] * 100
df['upper_wick'] = df['high'] - df[['open', 'close']].max(axis=1)
df['lower_wick'] = df[['open', 'close']].min(axis=1) - df['low']
df['upper_wick_percent'] = df['upper_wick'] / df['open'] * 100
df['lower_wick_percent'] = df['lower_wick'] / df['open'] * 100

# Détection des gaps
df['gap_up'] = (df['low'] > df['high'].shift(1)).astype(int)
df['gap_down'] = (df['high'] < df['low'].shift(1)).astype(int)
```

```
# 6. Caractéristiques temporelles
if include_time_features:
    # Heure de la journée (valeurs cycliques sin/cos)
    hour = df.index.hour
    df['hour_sin'] = np.sin(2 * np.pi * hour / 24)
    df['hour_cos'] = np.cos(2 * np.pi * hour / 24)

    # Jour de la semaine (valeurs cycliques sin/cos)
    day_of_week = df.index.dayofweek
    df['day_sin'] = np.sin(2 * np.pi * day_of_week / 7)
    df['day_cos'] = np.cos(2 * np.pi * day_of_week / 7)

    # Jour du mois (valeurs cycliques sin/cos)
    day = df.index.day
    df['day_of_month_sin'] = np.sin(2 * np.pi * day / 31)
    df['day_of_month_cos'] = np.cos(2 * np.pi * day / 31)

# 7. Détection des patterns de prix (via talib)
if include_price_patterns:
    try:
        # Patterns de retournement haussier
        df['hammer'] = talib.CDLHAMMER(df['open'].values, df['high'].values,
                                       df['low'].values, df['close'].values)
        df['inverted_hammer'] = talib.CDLINVERTEDHAMMER(df['open'].values, df['high'].values,
                                                        df['low'].values, df['close'].values)
        df['morning_star'] = talib.CDLMORNINGSTAR(df['open'].values, df['high'].values,
                                                  df['low'].values, df['close'].values)
        df['bullish_engulfing'] = talib.CDLENGULFING(df['open'].values, df['high'].values,
                                                     df['low'].values, df['close'].values)
```

```
            # Patterns de retournement baissier
            df['shooting_star'] = talib.CDLSHOOTINGSTAR(df['open'].values, df['high'].values,
                                                        df['low'].values, df['close'].values)
            df['evening_star'] = talib.CDLEVENINGSTAR(df['open'].values, df['high'].values,
                                                      df['low'].values, df['close'].values)
            df['bearish_engulfing'] = -talib.CDLENGULFING(df['open'].values, df['high'].values,
                                                          df['low'].values, df['close'].values)

            # Créer une caractéristique résumée des patterns
            bullish_patterns = df[['hammer', 'inverted_hammer', 'morning_star',
'bullish_engulfing']].sum(axis=1)
            bearish_patterns = df[['shooting_star', 'evening_star', 'bearish_engulfing']].sum(axis=1)

            df['bullish_patterns'] = bullish_patterns
            df['bearish_patterns'] = bearish_patterns

        except Exception as e:
            logger.warning(f"Erreur lors de la détection des patterns de prix: {str(e)}")

    # 8. Caractéristiques de support/résistance
    # Identifier les niveaux de support/résistance majeurs sur 50 périodes
    window = 50
    if len(df) >= window:
        # Détecter les sommets locaux (hauts)
        df['is_high'] = (df['high'] > df['high'].shift(1)) & (df['high'] > df['high'].shift(-1))

        # Détecter les creux locaux (bas)
        df['is_low'] = (df['low'] < df['low'].shift(1)) & (df['low'] < df['low'].shift(-1))

        # Distance par rapport au plus haut récent
        rolling_high = df['high'].rolling(window).max()
```

```
            df['dist_to_high'] = (df['close'] - rolling_high) / rolling_high * 100

            # Distance par rapport au plus bas récent
            rolling_low = df['low'].rolling(window).min()
            df['dist_to_low'] = (df['close'] - rolling_low) / rolling_low * 100

        # 9. Caractéristiques croisées
        # RSI contre les bandes de Bollinger
        df['rsi_bb'] = (df['rsi'] - 50) * df['bb_percent_b']

        # Momentum de prix et volume
        df['price_volume_trend'] = df['return_1'] * df['rel_volume_5']

        # Caractéristique d'inversion de tendance (combo ADX + RSI)
        df['reversal_signal'] = ((df['adx'] > 25) & (df['rsi'] < 30)) | ((df['adx'] > 25) & (df['rsi'] > 70))

        # 10. Nettoyer les données
        # Remplacer les valeurs infinies
        df.replace([np.inf, -np.inf], np.nan, inplace=True)

        # Supprimer les colonnes avec trop de NaN
        threshold = len(df) * 0.9  # 90% des valeurs doivent être non-NA
        df = df.dropna(axis=1, thresh=threshold)

        # Remplir les NaN restants avec des valeurs appropriées
        df.fillna(method='ffill', inplace=True)  # Forward fill
        df.fillna(0, inplace=True)  # Remplacer les NaN restants par 0

        return df

    def scale_features(self, data: pd.DataFrame, is_training: bool = True,
```

```
        method: str = 'standard', feature_group: str = 'default') -> pd.DataFrame:
"""
Normalise les caractéristiques pour l'entraînement du modèle

Args:
    data: DataFrame avec les caractéristiques
    is_training: Indique si c'est pour l'entraînement ou la prédiction
    method: Méthode de scaling ('standard' ou 'minmax')
    feature_group: Groupe de caractéristiques pour sauvegarder/charger les scalers

Returns:
    DataFrame avec les caractéristiques normalisées
"""
# Sélectionner les colonnes numériques
numeric_cols = data.select_dtypes(include=['float64', 'int64']).columns.tolist()

# Exclure les colonnes qu'on ne veut pas normaliser
cols_to_exclude = ['timestamp', 'date', 'open', 'high', 'low', 'close', 'volume']
feature_cols = [col for col in numeric_cols if col not in cols_to_exclude]

# Si aucune caractéristique à normaliser, retourner les données telles quelles
if not feature_cols:
    logger.warning("Aucune caractéristique à normaliser")
    return data

# Pour l'entraînement, créer et ajuster de nouveaux scalers
if is_training:
    if method == 'standard':
        scaler = StandardScaler()
    else:  # 'minmax'
        scaler = MinMaxScaler(feature_range=(-1, 1))
```

```
            # Ajuster le scaler sur les données d'entraînement
            scaler.fit(data[feature_cols])

            # Sauvegarder le scaler
            if self.save_scalers:
                scaler_path = os.path.join(self.scalers_path, f"{feature_group}_{method}_scaler.pkl")
                with open(scaler_path, 'wb') as f:
                    pickle.dump(scaler, f)

            # Stocker le scaler en mémoire
            self.scalers[f"{feature_group}_{method}"] = scaler

        # Pour la prédiction, utiliser un scaler existant
        else:
            scaler_key = f"{feature_group}_{method}"

            # Chercher d'abord en mémoire
            if scaler_key in self.scalers:
                scaler = self.scalers[scaler_key]

            # Sinon, charger depuis le disque
            else:
                scaler_path = os.path.join(self.scalers_path, f"{feature_group}_{method}_scaler.pkl")
                if os.path.exists(scaler_path):
                    with open(scaler_path, 'rb') as f:
                        scaler = pickle.load(f)
                    # Stocker en mémoire pour usage futur
                    self.scalers[scaler_key] = scaler
                else:
                    logger.error(f"Scaler non trouvé pour la prédiction: {scaler_key}")
```

```
            raise FileNotFoundError(f"Scaler non trouvé: {scaler_path}")

    # Transformer les données
    scaled_features = scaler.transform(data[feature_cols])

    # Créer un nouveau DataFrame avec les caractéristiques normalisées
    scaled_df = data.copy()
    scaled_df[feature_cols] = scaled_features

    return scaled_df

def prepare_lstm_data(self, data: pd.DataFrame, sequence_length: int = 60,
                      prediction_horizon: int = 12, is_training: bool = True) -> Tuple:
    """
    Prépare les données au format requis par le modèle LSTM

    Args:
        data: DataFrame avec les caractéristiques (déjà normalisées)
        sequence_length: Longueur des séquences d'entrée
        prediction_horizon: Horizon de prédiction (nombre de périodes)
        is_training: Indique si c'est pour l'entraînement ou la prédiction

    Returns:
        Tuple (X, y) pour l'entraînement ou X pour la prédiction
    """
    # Sélectionner les colonnes de caractéristiques (exclure timestamp/date/etc.)
    feature_cols = data.select_dtypes(include=['float64', 'int64']).columns.tolist()

    # Exclure des colonnes spécifiques si nécessaires
    cols_to_exclude = ['timestamp', 'date']
    feature_cols = [col for col in feature_cols if col not in cols_to_exclude]
```

```
# Créer les séquences d'entrée
X = []

for i in range(len(data) - sequence_length - (prediction_horizon if is_training else 0)):
    X.append(data[feature_cols].iloc[i:i+sequence_length].values)

X = np.array(X)

# Si c'est pour la prédiction, retourner seulement X
if not is_training:
    return X

# Sinon, créer également les labels
y_direction = []
y_volatility = []
y_volume = []
y_momentum = []

for i in range(len(data) - sequence_length - prediction_horizon):
    # Prix actuel (à la fin de la séquence d'entrée)
    current_price = data['close'].iloc[i+sequence_length-1]

    # Prix futur (après l'horizon de prédiction)
    future_price = data['close'].iloc[i+sequence_length+prediction_horizon-1]

    # Direction (1 si hausse, 0 si baisse)
    direction = 1 if future_price > current_price else 0
    y_direction.append(direction)

    # Volatilité (écart-type des rendements futurs)
```

```
            future_returns = data['close'].iloc[i+sequence_length:i+sequence_length+prediction_horizon].pct_change().dropna()
            volatility = future_returns.std() * np.sqrt(prediction_horizon)  # Annualisé
            y_volatility.append(volatility)

            # Volume relatif futur
            current_volume = data['volume'].iloc[i+sequence_length-1]
            future_volume = data['volume'].iloc[i+sequence_length:i+sequence_length+prediction_horizon].mean()
            relative_volume = future_volume / current_volume if current_volume > 0 else 1.0
            y_volume.append(relative_volume)

            # Momentum (changement de prix normalisé)
            price_change_pct = (future_price - current_price) / current_price
            momentum = np.tanh(price_change_pct * 5)  # Utiliser tanh pour normaliser entre -1 et 1
            y_momentum.append(momentum)

        # Convertir en tableaux numpy
        y_direction = np.array(y_direction)
        y_volatility = np.array(y_volatility)
        y_volume = np.array(y_volume)
        y_momentum = np.array(y_momentum)

        # Empaqueter dans un tuple
        y = (y_direction, y_volatility, y_volume, y_momentum)

        return X, y

    def create_multi_horizon_data(self, data: pd.DataFrame,
                                  sequence_length: int = 60,
                                  horizons: List[int] = [12, 24, 96],
                                  is_training: bool = True) -> Tuple:
```

```
"""
Prépare les données pour une prédiction multi-horizon

Args:
    data: DataFrame avec les caractéristiques (déjà normalisées)
    sequence_length: Longueur des séquences d'entrée
    horizons: Liste des horizons de prédiction (en périodes)
    is_training: Indique si c'est pour l'entraînement ou la prédiction

Returns:
    Tuple (X, y_list) pour l'entraînement ou X pour la prédiction
"""
# Sélectionner les colonnes de caractéristiques
feature_cols = data.select_dtypes(include=['float64', 'int64']).columns.tolist()
cols_to_exclude = ['timestamp', 'date']
feature_cols = [col for col in feature_cols if col not in cols_to_exclude]

# Créer les séquences d'entrée
X = []

for i in range(len(data) - sequence_length - (max(horizons) if is_training else 0)):
    X.append(data[feature_cols].iloc[i:i+sequence_length].values)

X = np.array(X)

# Si c'est pour la prédiction, retourner seulement X
if not is_training:
    return X

# Sinon, créer également les labels pour chaque horizon
y_list = []
```

```
    for horizon in horizons:
        y_direction = []
        y_volatility = []
        y_volume = []
        y_momentum = []

        for i in range(len(data) - sequence_length - horizon):
            # Prix actuel (à la fin de la séquence d'entrée)
            current_price = data['close'].iloc[i+sequence_length-1]

            # Prix futur (après l'horizon de prédiction)
            future_price = data['close'].iloc[i+sequence_length+horizon-1]

            # Direction (1 si hausse, 0 si baisse)
            direction = 1 if future_price > current_price else 0
            y_direction.append(direction)

            # Volatilité (écart-type des rendements futurs)
            future_returns =
data['close'].iloc[i+sequence_length:i+sequence_length+horizon].pct_change().dropna()
            volatility = future_returns.std() * np.sqrt(horizon)  # Annualisé
            y_volatility.append(volatility)

            # Volume relatif futur
            current_volume = data['volume'].iloc[i+sequence_length-1]
            future_volume = data['volume'].iloc[i+sequence_length:i+sequence_length+horizon].mean()
            relative_volume = future_volume / current_volume if current_volume > 0 else 1.0
            y_volume.append(relative_volume)

            # Momentum (changement de prix normalisé)
```

```
            price_change_pct = (future_price - current_price) / current_price
            momentum = np.tanh(price_change_pct * 5)
            y_momentum.append(momentum)

        # Convertir en tableaux numpy
        y_direction = np.array(y_direction)
        y_volatility = np.array(y_volatility)
        y_volume = np.array(y_volume)
        y_momentum = np.array(y_momentum)

        # Ajouter à la liste des sorties
        y_list.extend([y_direction, y_volatility, y_volume, y_momentum])

    return X, y_list
```

```
================================================
File: crypto_trading_bot_CLAUDE/ai/models/lstm_model.py
================================================
# ai/models/lstm_model.py
"""
Architecture LSTM avancée pour prédictions multi-horizon et multi-facteur
Intègre attention, connexions résiduelles et apprentissage par transfert
"""
import os
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow.keras.models import Model, load_model, clone_model
from tensorflow.keras.layers import (
    Input, LSTM, Dense, Dropout, BatchNormalization,
    Bidirectional, Concatenate, Add, Multiply, Reshape,
```

```
    Conv1D, MaxPooling1D, GlobalAveragePooling1D, Lambda,
    Layer, Activation
)
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint, ReduceLROnPlateau
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.regularizers import l1_l2
import tensorflow.keras.backend as K
from typing import Dict, List, Tuple, Union, Optional
from datetime import datetime
import json

from config.config import DATA_DIR
from utils.logger import setup_logger
from ai.models.attention import MultiHeadAttention, TemporalAttentionBlock, TimeSeriesAttention

logger = setup_logger("enhanced_lstm_model")

class SpatialDropout1D(Dropout):
    """
    Dropout spatial qui supprime des canaux de caractéristiques entiers plutôt que des valeurs individuelles
    """
    def __init__(self, rate, **kwargs):
        super(SpatialDropout1D, self).__init__(rate, **kwargs)
        self.input_spec = None

    def call(self, inputs, training=None):
        if training is None:
            training = K.learning_phase()

        if 0. < self.rate < 1.:
```

```
            # Créer un masque de bruit pour des canaux entiers
            noise_shape = (inputs.shape[0], 1, inputs.shape[2])
            return K.dropout(inputs, self.rate, noise_shape)
        return inputs

class ResidualBlock(Layer):
    """
    Bloc résiduel personnalisé qui combine LSTM et connexions résiduelles
    """
    def __init__(self, units, dropout_rate=0.3, use_batch_norm=True, **kwargs):
        super(ResidualBlock, self).__init__(**kwargs)
        self.units = units
        self.dropout_rate = dropout_rate
        self.use_batch_norm = use_batch_norm

        # Définir les couches
        self.lstm = Bidirectional(LSTM(units, return_sequences=True,
                                       kernel_regularizer=l1_l2(l1=1e-5, l2=1e-4)))
        self.dropout = SpatialDropout1D(dropout_rate)

        if use_batch_norm:
            self.batch_norm = BatchNormalization()

        self.projection = None

    def build(self, input_shape):
        # Projection pour faire correspondre les dimensions si nécessaire
        input_dim = input_shape[-1]
        output_dim = self.units * 2  # Bidirectionnel double la dimension

        if input_dim != output_dim:
```

```
            self.projection = Dense(output_dim)

        super(ResidualBlock, self).build(input_shape)

    def call(self, inputs, training=None):
        x = self.lstm(inputs)

        if self.use_batch_norm:
            x = self.batch_norm(x, training=training)

        x = self.dropout(x, training=training)

        # Connexion résiduelle
        if self.projection is not None:
            residual = self.projection(inputs)
        else:
            residual = inputs

        return Add()([x, residual])

    def get_config(self):
        config = super(ResidualBlock, self).get_config()
        config.update({
            'units': self.units,
            'dropout_rate': self.dropout_rate,
            'use_batch_norm': self.use_batch_norm
        })
        return config

class EnhancedLSTMModel:
    """
```

```
Modèle LSTM avancé pour prédictions multi-horizon et multi-facteur

Caractéristiques:
- Architecture LSTM bidirectionnelle avec attention multi-tête
- Prédictions multi-horizon (court, moyen, long terme)
- Prédictions multi-facteur (direction, volatilité, volume, momentum)
- Connexions résiduelles pour une meilleure propagation du gradient
- Dropout spatial pour une meilleure régularisation
- Mécanismes d'alerte précoce pour les retournements de marché
- Intégration avec l'apprentissage par transfert
"""
def __init__(self,
             input_length: int = 60,
             feature_dim: int = 30,
             lstm_units: List[int] = [128, 96, 64],
             dropout_rate: float = 0.3,
             learning_rate: float = 0.0005,
             l1_reg: float = 0.0001,
             l2_reg: float = 0.0001,
             use_attention: bool = True,
             attention_heads: int = 8,
             use_residual: bool = True,
             # Format: (périodes, nom_lisible, est_principal)
             prediction_horizons: List[Tuple[int, str, bool]] = [
                (12, "3h", True),    # Court terme (3h avec bougies de 15min)
                (48, "12h", True),   # Moyen terme (12h)
                (192, "48h", True),  # Long terme (48h)
                (384, "96h", False)  # Très long terme (96h, optionnel)
             ]):
    """
    Initialise le modèle LSTM avancé
```

```
Args:
    input_length: Nombre de pas de temps en entrée
    feature_dim: Dimension des caractéristiques d'entrée
    lstm_units: Liste des unités LSTM pour chaque couche
    dropout_rate: Taux de dropout pour la régularisation
    learning_rate: Taux d'apprentissage du modèle
    l1_reg: Régularisation L1
    l2_reg: Régularisation L2
    use_attention: Utiliser les mécanismes d'attention
    attention_heads: Nombre de têtes d'attention
    use_residual: Utiliser les connexions résiduelles
    prediction_horizons: Horizons de prédiction avec format (périodes, nom, est_principal)
"""
self.input_length = input_length
self.feature_dim = feature_dim
self.lstm_units = lstm_units
self.dropout_rate = dropout_rate
self.learning_rate = learning_rate
self.l1_reg = l1_reg
self.l2_reg = l2_reg
self.use_attention = use_attention
self.attention_heads = attention_heads
self.use_residual = use_residual
self.prediction_horizons = prediction_horizons

# Extraire juste les périodes pour la compatibilité
self.horizon_periods = [h[0] for h in prediction_horizons]

# Séparer les horizons principaux et secondaires
self.main_horizons = [h for h in prediction_horizons if h[2]]
```

```
        # Identifier les indices pour les différents horizons
        self.short_term_idx = 0  # Premier horizon (le plus court)
        self.mid_term_idx = min(1, len(prediction_horizons)-1)  # Second horizon ou le premier si un seul
        self.long_term_idx = min(2, len(prediction_horizons)-1)  # Troisième horizon ou le dernier disponible

        # Facteurs de sortie pour chaque horizon (direction, volatilité, volume, momentum)
        self.factors = ["direction", "volatility", "volume", "momentum"]
        self.num_factors = len(self.factors)

        # Variables pour mémoriser la normalisation
        self.scalers = {}

        # Créer les modèles Keras (principal et auxiliaires)
        self.model = self._build_model()
        self.reversal_detector = self._build_reversal_detector()

        # Chemins des modèles
        self.models_dir = os.path.join(DATA_DIR, "models", "production")
        os.makedirs(self.models_dir, exist_ok=True)

        self.model_path = os.path.join(self.models_dir, "enhanced_lstm_model.h5")
        self.reversal_detector_path = os.path.join(self.models_dir, "reversal_detector_model.h5")

        # Historique des performances du modèle
        self.metrics_history = []

        # Méta-informations pour l'apprentissage par transfert
        self.transfer_info = {
            "base_symbol": None,
```

```
        "trained_symbols": [],
        "transfer_history": []
    }

def _build_model(self) -> Model:
    """
    Construit le modèle LSTM multi-horizon et multi-facteur

    Returns:
        Modèle Keras compilé
    """
    # Entrée de forme (batch_size, sequence_length, num_features)
    inputs = Input(shape=(self.input_length, self.feature_dim), name="market_sequence")

    # 1. Branche d'extraction de caractéristiques à court terme (convolutive)
    short_term_features = Conv1D(filters=64, kernel_size=3, padding='same', activation='relu',
                                 kernel_regularizer=l1_l2(l1=self.l1_reg, l2=self.l2_reg),
                                 name="short_term_conv1")(inputs)
    short_term_features = BatchNormalization()(short_term_features)
    short_term_features = Conv1D(filters=32, kernel_size=3, padding='same', activation='relu',
                                 name="short_term_conv2")(short_term_features)
    short_term_features = MaxPooling1D(pool_size=2, padding='same')(short_term_features)

    # 2. Branche principale LSTM avec blocs résiduels
    x = inputs

    # Appliquer des blocs résiduels en série
    for i, units in enumerate(self.lstm_units):
        x = ResidualBlock(
            units=units,
            dropout_rate=self.dropout_rate,
```

```
        use_batch_norm=True,
        name=f"residual_block_{i+1}"
    )(x)

# 3. Appliquer l'attention si activée
if self.use_attention:
    # Attention multi-tête inspirée des transformers
    mha = MultiHeadAttention(
        num_heads=self.attention_heads,
        head_dim=32,
        dropout=self.dropout_rate,
        name="multi_head_attention"
    )(x)

    # Connexion résiduelle après l'attention
    x = Add(name="post_attention_residual")([x, mha])
    x = BatchNormalization(name="post_attention_norm")(x)

# 4. Combiner avec les caractéristiques à court terme
# Adapter les dimensions si nécessaire
if short_term_features.shape[1] != x.shape[1]:
    # Utiliser un redimensionnement adaptatif
    scale_factor = x.shape[1] / short_term_features.shape[1]

    def resize_temporal(tensor, scale):
        # Redimensionne la dimension temporelle d'un tenseur 3D
        shape = tf.shape(tensor)
        target_length = tf.cast(tf.cast(shape[1], tf.float32) * scale, tf.int32)

        # Redimensionner avec un reshape et des opérations de répétition
        resized = tf.image.resize(
```

```
            tf.expand_dims(tensor, 3),
            [target_length, shape[2]],
            method='nearest'
        )
        return tf.squeeze(resized, 3)

    short_term_features = Lambda(
        lambda t: resize_temporal(t, scale_factor),
        name="temporal_resize"
    )(short_term_features)

# Projeter les caractéristiques à court terme pour correspondre à la dimension de x
short_term_features = Dense(
    x.shape[-1],
    activation='relu',
    name="short_term_projection"
)(short_term_features)

# Combiner par addition (connexion résiduelle)
combined = Add(name="feature_combination")([x, short_term_features])

# Couche de contexte global pour la sortie
global_context = GlobalAveragePooling1D(name="global_pooling")(combined)

# 5. Couches denses partagées pour l'extraction de caractéristiques de haut niveau
shared_features = Dense(
    128,
    activation='relu',
    kernel_regularizer=l1_l2(l1=self.l1_reg, l2=self.l2_reg),
    name="shared_features"
)(global_context)
```

```
shared_features = BatchNormalization(name="shared_features_norm")(shared_features)
shared_features = Dropout(0.2, name="shared_features_dropout")(shared_features)

# 6. Créer une sortie pour chaque horizon et chaque facteur
outputs = []
output_names = []

for h_idx, (horizon, horizon_name, _) in enumerate(self.prediction_horizons):
    # Couche spécifique à l'horizon
    horizon_features = Dense(
        64,
        activation='relu',
        name=f"horizon_{horizon_name}_features"
    )(shared_features)

    # Direction (probabilité de hausse)
    direction = Dense(
        1,
        activation='sigmoid',
        name=f"direction_{horizon_name}"
    )(horizon_features)
    outputs.append(direction)
    output_names.append(f"direction_{horizon_name}")

    # Volatilité (relative)
    volatility = Dense(
        1,
        activation='relu',  # La volatilité est toujours positive
        name=f"volatility_{horizon_name}"
    )(horizon_features)
    outputs.append(volatility)
```

```
    output_names.append(f"volatility_{horizon_name}")

    # Volume relatif
    volume = Dense(
        1,
        activation='relu',  # Le volume relatif est toujours positif
        name=f"volume_{horizon_name}"
    )(horizon_features)
    outputs.append(volume)
    output_names.append(f"volume_{horizon_name}")

    # Momentum (force de la tendance)
    momentum = Dense(
        1,
        activation='tanh',  # Tanh pour avoir une valeur entre -1 et 1
        name=f"momentum_{horizon_name}"
    )(horizon_features)
    outputs.append(momentum)
    output_names.append(f"momentum_{horizon_name}")

# 7. Créer le modèle final
model = Model(inputs=inputs, outputs=outputs, name="multi_horizon_lstm")

# 8. Compiler avec des pertes et métriques appropriées
losses = []
metrics = []

for output_name in output_names:
    if output_name.startswith("direction"):
        # Classification binaire pour la direction
        losses.append('binary_crossentropy')
```

```
            metrics.append('accuracy')
        else:
            # Régression pour les autres facteurs
            losses.append('mse')  # Erreur quadratique moyenne
            metrics.append('mae')  # Erreur absolue moyenne

    # Compiler le modèle
    model.compile(
        optimizer=Adam(learning_rate=self.learning_rate),
        loss=losses,
        metrics=metrics
    )

    return model

def _build_reversal_detector(self) -> Model:
    """
    Construit un modèle spécialisé pour détecter les retournements de marché majeurs

    Returns:
        Modèle de détection des retournements
    """
    # Ce modèle utilise la même entrée que le modèle principal mais se spécialise
    # dans la détection des patterns de retournement de marché

    # Entrée de forme (batch_size, sequence_length, num_features)
    inputs = Input(shape=(self.input_length, self.feature_dim), name="market_sequence")

    # Utiliser des convolutions pour détecter des motifs locaux
    x = Conv1D(filters=64, kernel_size=3, padding='same', activation='relu')(inputs)
    x = BatchNormalization()(x)
```

```
    x = Conv1D(filters=64, kernel_size=5, padding='same', activation='relu')(x)
    x = BatchNormalization()(x)
    x = MaxPooling1D(pool_size=2)(x)

    # LSTM bidirectionnel pour capturer les dépendances temporelles
    x = Bidirectional(LSTM(64, return_sequences=True))(x)
    x = Dropout(0.3)(x)

    # Attention pour se concentrer sur les parties importantes de la séquence
    attention_layer = TimeSeriesAttention(filters=64)(x)

    # Contexte global
    global_features = GlobalAveragePooling1D()(attention_layer)

    # Couches denses
    x = Dense(32, activation='relu')(global_features)
    x = BatchNormalization()(x)
    x = Dropout(0.2)(x)

    # Sorties: probabilité et ampleur du retournement
    reversal_probability = Dense(1, activation='sigmoid', name="reversal_probability")(x)
    reversal_magnitude = Dense(1, activation='relu', name="reversal_magnitude")(x)

    # Créer le modèle
    model = Model(inputs=inputs, outputs=[reversal_probability, reversal_magnitude],
name="reversal_detector")

    # Compiler
    model.compile(
      optimizer=Adam(learning_rate=0.001),
      loss=['binary_crossentropy', 'mse'],
```

```
        metrics=[['accuracy'], ['mae']]
    )

    return model

def preprocess_data(self, data: pd.DataFrame,
                    feature_engineering, is_training: bool = True) -> Tuple:
    """
    Prétraite les données pour l'entraînement ou la prédiction

    Args:
        data: DataFrame avec les données OHLCV et indicateurs
        feature_engineering: Instance FeatureEngineering pour créer/normaliser les caractéristiques
        is_training: Indique si le prétraitement est pour l'entraînement

    Returns:
        X: Données d'entrée normalisées
        y_list: Liste des données cibles pour chaque sortie (vide si is_training=False)
    """
    # 1. Créer les caractéristiques avancées
    featured_data = feature_engineering.create_features(
        data,
        include_time_features=True,
        include_price_patterns=True
    )

    # 2. Normaliser les caractéristiques
    normalized_data = feature_engineering.scale_features(
        featured_data,
        is_training=is_training,
        method='standard',
```

```
        feature_group='lstm'
    )

    # 3. Convertir en séquences pour LSTM
    X, y_list = feature_engineering.create_multi_horizon_data(
        normalized_data,
        sequence_length=self.input_length,
        horizons=self.horizon_periods,
        is_training=is_training
    )

    # 4. Créer les labels pour le détecteur de retournement si en mode entraînement
    if is_training:
        y_reversal = self._create_reversal_labels(data)
        return X, y_list, y_reversal

    return X, y_list

def _create_reversal_labels(self, data: pd.DataFrame) -> Tuple[np.ndarray, np.ndarray]:
    """
    Crée les labels pour le détecteur de retournement

    Args:
        data: DataFrame avec les données OHLCV

    Returns:
        Tuple de (probabilité de retournement, amplitude du retournement)
    """
    # Calcul des rendements
    returns = data['close'].pct_change()
```

```
        # Identifier les retournements majeurs (mouvements brusques après une tendance)
        reversal_probability = []
        reversal_magnitude = []

        # Fenêtre pour la détection de retournements
        window = 20
        threshold = 0.03  # 3% de mouvement pour un retournement significatif

        for i in range(window, len(data) - self.input_length - max(self.horizon_periods)):
            # Tendance précédente (sur la fenêtre)
            previous_returns = returns.iloc[i-window:i]
            previous_trend = previous_returns.mean() * window  # Rendement cumulé

            # Mouvement futur (sur l'horizon le plus court)
            future_price_change = (data['close'].iloc[i+self.input_length+self.horizon_periods[0]-1] -
                                   data['close'].iloc[i+self.input_length-1]) / data['close'].iloc[i+self.input_length-1]

            # Un retournement est un mouvement dans la direction opposée à la tendance précédente
            is_reversal = (previous_trend * future_price_change < 0) and (abs(future_price_change) >
threshold)

            reversal_probability.append(1.0 if is_reversal else 0.0)
            reversal_magnitude.append(abs(future_price_change))

        return np.array(reversal_probability), np.array(reversal_magnitude)

    def train(self, train_data: pd.DataFrame, feature_engineering,
              validation_data: pd.DataFrame = None,
              epochs: int = 100, batch_size: int = 32,
              patience: int = 20, save_best: bool = True,
              symbol: str = None) -> Dict:
```

```
"""
Entraîne le modèle LSTM multi-horizon

Args:
    train_data: DataFrame avec les données d'entraînement
    feature_engineering: Instance FeatureEngineering
    validation_data: DataFrame avec les données de validation
    epochs: Nombre d'époques d'entraînement
    batch_size: Taille du batch
    patience: Patience pour l'early stopping
    save_best: Sauvegarder le meilleur modèle
    symbol: Symbole de la paire de trading

Returns:
    Historique d'entraînement
"""
# Prétraiter les données d'entraînement
X_train, y_train, y_reversal_train = self.preprocess_data(
    train_data,
    feature_engineering,
    is_training=True
)

# Prétraiter les données de validation si fournies
validation_data_main = None
validation_data_reversal = None

if validation_data is not None:
    X_val, y_val, y_reversal_val = self.preprocess_data(
        validation_data,
        feature_engineering,
```

```
        is_training=True
    )
    validation_data_main = (X_val, y_val)
    validation_data_reversal = (X_val, y_reversal_val)

# Callbacks pour l'entraînement du modèle principal
callbacks_main = [
    EarlyStopping(
        monitor='val_loss' if validation_data is not None else 'loss',
        patience=patience,
        restore_best_weights=True
    ),
    ReduceLROnPlateau(
        monitor='val_loss' if validation_data is not None else 'loss',
        factor=0.5,
        patience=patience // 2,
        min_lr=1e-6
    )
]

if save_best:
    callbacks_main.append(
        ModelCheckpoint(
            filepath=self.model_path,
            monitor='val_loss' if validation_data is not None else 'loss',
            save_best_only=True,
            save_weights_only=False
        )
    )

# 1. Entraîner le modèle principal
```

```
logger.info("Entraînement du modèle LSTM multi-horizon...")
history_main = self.model.fit(
    x=X_train,
    y=y_train,
    epochs=epochs,
    batch_size=batch_size,
    validation_data=validation_data_main,
    callbacks=callbacks_main,
    verbose=1
)

# 2. Entraîner le détecteur de retournement
logger.info("Entraînement du détecteur de retournement...")
history_reversal = self.reversal_detector.fit(
    x=X_train,
    y=y_reversal_train,
    epochs=max(30, epochs//2),  # Moins d'époques pour ce modèle plus simple
    batch_size=batch_size,
    validation_data=validation_data_reversal,
    callbacks=[
        EarlyStopping(patience=patience//2, restore_best_weights=True),
        ReduceLROnPlateau(factor=0.5, patience=patience//4, min_lr=1e-6)
    ],
    verbose=1
)

if save_best:
    self.reversal_detector.save(self.reversal_detector_path)

# Sauvegarder les métriques
self._save_metrics(history_main.history, symbol)
```

```
        # Si c'est un nouveau symbole, mettre à jour les métadonnées de transfert
        if symbol and symbol not in self.transfer_info['trained_symbols']:
            self.transfer_info['trained_symbols'].append(symbol)

            # Si c'est le premier symbole, le définir comme symbole de base
            if self.transfer_info['base_symbol'] is None:
                self.transfer_info['base_symbol'] = symbol

            # Enregistrer cette session d'entraînement
            self.transfer_info['transfer_history'].append({
                'symbol': symbol,
                'timestamp': datetime.now().isoformat(),
                'epochs': len(history_main.history['loss']),
                'final_loss': float(history_main.history['loss'][-1])
            })

            # Sauvegarder les métadonnées de transfert
            self._save_transfer_info()

        return {
            'main_model': history_main.history,
            'reversal_detector': history_reversal.history
        }

    def predict(self, data: pd.DataFrame, feature_engineering) -> Dict:
        """
        Fait des prédictions avec le modèle entraîné

        Args:
            data: DataFrame avec les données récentes
```

```
    feature_engineering: Instance FeatureEngineering

Returns:
    Dictionnaire avec les prédictions pour chaque horizon et facteur
    et l'alerte de retournement
"""
# Prétraiter les données
X, _ = self.preprocess_data(data, feature_engineering, is_training=False)

# Faire les prédictions avec le modèle principal
predictions = self.model.predict(X)

# Faire les prédictions avec le détecteur de retournement
reversal_prob, reversal_mag = self.reversal_detector.predict(X)

# Organiser les résultats
results = {}

# Dernière séquence pour la prédiction la plus récente
latest_idx = -1

# Pour chaque horizon
for h_idx, (horizon, horizon_name, _) in enumerate(self.prediction_horizons):
    # Indice de base pour les 4 facteurs de cet horizon
    base_idx = h_idx * self.num_factors

    # Prédictions pour cet horizon
    direction = float(predictions[base_idx][latest_idx][0])
    volatility = float(predictions[base_idx+1][latest_idx][0])
    volume = float(predictions[base_idx+2][latest_idx][0])
    momentum = float(predictions[base_idx+3][latest_idx][0])
```

```
            # Convertir la prédiction de direction en probabilité (0-100%)
            direction_probability = direction * 100

            # Déterminer la tendance prédite
            trend = "HAUSSIER" if direction > 0.5 else "BAISSIER"
            trend_strength = abs(direction - 0.5) * 2  # Force de 0 à 1

            # Confiance dans la prédiction
            confidence = trend_strength * (1 - volatility)  # Plus la volatilité est faible, plus la confiance est élevée

            results[horizon_name] = {
                "direction": trend,
                "direction_probability": direction_probability,
                "trend_strength": float(trend_strength),
                "predicted_volatility": float(volatility),
                "predicted_volume": float(volume),
                "predicted_momentum": float(momentum),
                "confidence": float(confidence),
                "prediction_timestamp": datetime.now().isoformat()
            }

        # Ajouter les prédictions de retournement
        results["reversal_alert"] = {
            "probability": float(reversal_prob[latest_idx][0]),
            "magnitude": float(reversal_mag[latest_idx][0]),
            "is_warning": reversal_prob[latest_idx][0] > 0.7,  # Alerte si probabilité > 70%
            "prediction_timestamp": datetime.now().isoformat()
        }
```

```
    return results

def save(self, path: Optional[str] = None) -> None:
    """
    Sauvegarde le modèle sur disque

    Args:
        path: Chemin de sauvegarde (utilise le chemin par défaut si None)
    """
    save_path = path or self.model_path

    # Créer le répertoire si nécessaire
    os.makedirs(os.path.dirname(save_path), exist_ok=True)

    # Sauvegarder le modèle principal
    self.model.save(save_path)
    logger.info(f"Modèle principal sauvegardé: {save_path}")

    # Sauvegarder le détecteur de retournement
    reversal_path = os.path.splitext(save_path)[0] + "_reversal.h5"
    self.reversal_detector.save(reversal_path)
    logger.info(f"Détecteur de retournement sauvegardé: {reversal_path}")

def load(self, path: Optional[str] = None) -> None:
    """
    Charge le modèle depuis le disque

    Args:
        path: Chemin du modèle (utilise le chemin par défaut si None)
    """
    load_path = path or self.model_path
```

```
# Vérifier si le fichier existe
if not os.path.exists(load_path):
    raise FileNotFoundError(f"Modèle non trouvé: {load_path}")

# Charger le modèle principal
self.model = load_model(
    load_path,
    custom_objects={
        'MultiHeadAttention': MultiHeadAttention,
        'TemporalAttentionBlock': TemporalAttentionBlock,
        'TimeSeriesAttention': TimeSeriesAttention,
        'SpatialDropout1D': SpatialDropout1D,
        'ResidualBlock': ResidualBlock
    }
)
logger.info(f"Modèle principal chargé: {load_path}")

# Essayer de charger le détecteur de retournement
reversal_path = os.path.splitext(load_path)[0] + "_reversal.h5"
if os.path.exists(reversal_path):
    self.reversal_detector = load_model(
        reversal_path,
        custom_objects={
            'TimeSeriesAttention': TimeSeriesAttention
        }
    )
    logger.info(f"Détecteur de retournement chargé: {reversal_path}")
else:
    logger.warning(f"Détecteur de retournement non trouvé: {reversal_path}")
```

```
def _save_metrics(self, history: Dict, symbol: Optional[str] = None) -> None:
    """
    Sauvegarde les métriques d'entraînement

    Args:
        history: Historique d'entraînement
        symbol: Symbole de la paire de trading
    """
    metrics_entry = {
        "timestamp": datetime.now().isoformat(),
        "symbol": symbol,
        "metrics": history,
        "parameters": {
            "lstm_units": self.lstm_units,
            "dropout_rate": self.dropout_rate,
            "learning_rate": self.learning_rate,
            "use_attention": self.use_attention,
            "attention_heads": self.attention_heads,
            "use_residual": self.use_residual
        }
    }

    self.metrics_history.append(metrics_entry)

    # Sauvegarder dans un fichier
    metrics_path = os.path.join(self.models_dir, "metrics_history.json")

    try:
        # Charger l'historique existant si disponible
        existing_metrics = []
        if os.path.exists(metrics_path):
```

```
            with open(metrics_path, 'r') as f:
                existing_metrics = json.load(f)

        # Ajouter la nouvelle entrée
        existing_metrics.append(metrics_entry)

        # Sauvegarder
        with open(metrics_path, 'w') as f:
            json.dump(existing_metrics, f, indent=2, default=str)

        logger.info(f"Métriques sauvegardées: {metrics_path}")
    except Exception as e:
        logger.error(f"Erreur lors de la sauvegarde des métriques: {str(e)}")

def _save_transfer_info(self) -> None:
    """Sauvegarde les métadonnées d'apprentissage par transfert"""
    transfer_path = os.path.join(self.models_dir, "transfer_learning_info.json")

    try:
        with open(transfer_path, 'w') as f:
            json.dump(self.transfer_info, f, indent=2, default=str)
        logger.info(f"Métadonnées de transfert sauvegardées: {transfer_path}")
    except Exception as e:
        logger.error(f"Erreur lors de la sauvegarde des métadonnées de transfert: {str(e)}")

def transfer_to_new_symbol(self, symbol: str) -> None:
    """
    Prépare le modèle pour l'apprentissage par transfert sur un nouveau symbole

    Args:
        symbol: Nouveau symbole pour l'apprentissage par transfert
```

```
    """
    # Sauvegarder les poids du modèle source
    source_weights = self.model.get_weights()

    # Réduire le taux d'apprentissage pour le transfert
    K.set_value(self.model.optimizer.learning_rate, self.learning_rate * 0.5)

    logger.info(f"Modèle préparé pour l'apprentissage par transfert vers {symbol}")
    logger.info(f"Taux d'apprentissage réduit à {K.get_value(self.model.optimizer.learning_rate)}")

def get_reversal_threshold(self, data: pd.DataFrame, feature_engineering,
                           percentile: float = 90) -> float:
    """
    Calcule un seuil dynamique pour les alertes de retournement

    Args:
        data: Données historiques récentes
        feature_engineering: Instance FeatureEngineering
        percentile: Percentile pour le seuil (défaut: 90e percentile)

    Returns:
        Seuil de probabilité pour les alertes de retournement
    """
    # Prétraiter les données
    X, _ = self.preprocess_data(data, feature_engineering, is_training=False)

    # Obtenir les prédictions de retournement
    reversal_probs, _ = self.reversal_detector.predict(X)

    # Calculer le seuil basé sur le percentile spécifié
    threshold = np.percentile(reversal_probs, percentile)
```

```
        return float(threshold)

def test_model():
    """Test simple du modèle pour vérifier qu'il compile correctement"""
    model = EnhancedLSTMModel(
        input_length=60,
        feature_dim=30,
        lstm_units=[64, 48, 32],
        prediction_horizons=[
            (12, "3h", True),
            (48, "12h", True),
            (192, "48h", True)
        ]
    )

    # Afficher le résumé du modèle
    model.model.summary()

    print("Le modèle a été compilé avec succès !")

    return model

if __name__ == "__main__":
    model = t

================================================
File: crypto_trading_bot_CLAUDE/ai/models/model_trainer.py
================================================
# ai/models/model_trainer.py
"""
```

```
Module d'entraînement du modèle LSTM avec validation croisée temporelle
"""
import os
import numpy as np
import pandas as pd
from typing import Dict, List, Tuple, Union, Optional
from datetime import datetime, timedelta
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint, ReduceLROnPlateau, Callback
import json

from ai.models.lstm_model import LSTMModel
from ai.models.feature_engineering import FeatureEngineering
from config.config import DATA_DIR
from utils.logger import setup_logger
from utils.visualizer import TradeVisualizer

# Configuration des GPU pour TensorFlow
gpus = tf.config.experimental.list_physical_devices('GPU')
if gpus:
    try:
        # Limiter la mémoire GPU utilisée
        for gpu in gpus:
            tf.config.experimental.set_memory_growth(gpu, True)
        logical_gpus = tf.config.experimental.list_logical_devices('GPU')
        print(f"{len(gpus)} GPU(s) physiques et {len(logical_gpus)} GPU(s) logiques détectés")
    except RuntimeError as e:
        print(f"Erreur lors de la configuration GPU: {e}")
```

```
logger = setup_logger("model_trainer")


class EarlyStoppingOnMemoryLeak(Callback):
    """
    Callback pour détecter et arrêter l'entraînement en cas de fuite mémoire
    """
    def __init__(self, memory_threshold_mb: float = 1000):
        super().__init__()
        self.memory_threshold_mb = memory_threshold_mb
        self.starting_memory = 0

    def on_train_begin(self, logs=None):
        # Mesurer l'utilisation mémoire au début
        self.starting_memory = self._get_memory_usage()
        logger.info(f"Mémoire au début de l'entraînement: {self.starting_memory:.2f} MB")

    def on_epoch_end(self, epoch, logs=None):
        # Vérifier l'utilisation mémoire à la fin de chaque époque
        current_memory = self._get_memory_usage()
        memory_increase = current_memory - self.starting_memory

        if memory_increase > self.memory_threshold_mb:
            logger.warning(f"Fuite mémoire détectée ! Augmentation: {memory_increase:.2f} MB. Arrêt de l'entraînement.")
            self.model.stop_training = True

    def _get_memory_usage(self):
        """Mesure la consommation mémoire actuelle en MB"""
        import psutil
        process = psutil.Process(os.getpid())
        memory_info = process.memory_info()
```

```
        return memory_info.rss / (1024 * 1024)  # Convertir en MB

class ConceptDriftDetector(Callback):
    """
    Callback pour détecter la dérive conceptuelle pendant l'entraînement
    """
    def __init__(self, validation_data, threshold=0.15, patience=3):
        super().__init__()
        self.validation_data = validation_data
        self.threshold = threshold
        self.patience = patience
        self.baseline_metrics = None
        self.degradation_count = 0

    def on_train_begin(self, logs=None):
        # Évaluer le modèle sur les données de validation pour établir une référence
        metrics = self.model.evaluate(
            self.validation_data[0],
            self.validation_data[1],
            verbose=0
        )
        self.baseline_metrics = metrics[0]  # Prendre la loss globale comme référence
        logger.info(f"Métrique de référence établie: {self.baseline_metrics:.4f}")

    def on_epoch_end(self, epoch, logs=None):
        # Évaluer le modèle sur les données de validation
        current_metrics = self.model.evaluate(
            self.validation_data[0],
            self.validation_data[1],
            verbose=0
        )[0]
```

```
        # Calculer la dégradation relative
        degradation = (current_metrics - self.baseline_metrics) / self.baseline_metrics

        logger.debug(f"Epoch {epoch+1}: Dégradation des métriques: {degradation:.4f}")

        # Vérifier la dérive conceptuelle
        if degradation > self.threshold:
            self.degradation_count += 1
            logger.warning(f"Dérive conceptuelle possible détectée, compteur: {self.degradation_count}/{self.patience}")

            if self.degradation_count >= self.patience:
                logger.warning("Dérive conceptuelle confirmée. Arrêt de l'entraînement.")
                self.model.stop_training = True
        else:
            # Réinitialiser le compteur si la dégradation est en dessous du seuil
            self.degradation_count = 0
            # Mettre à jour la référence si les performances s'améliorent
            if current_metrics < self.baseline_metrics:
                self.baseline_metrics = current_metrics
                logger.info(f"Nouvelle métrique de référence: {self.baseline_metrics:.4f}")

class ModelTrainer:
    """
    Classe pour l'entraînement et la validation du modèle LSTM
    """
    def __init__(self, model_params: Dict = None):
        """
        Initialise le ModelTrainer
```

```
Args:
    model_params: Paramètres du modèle LSTM
"""
# Paramètres par défaut du modèle
default_params = {
    "input_length": 60,
    "feature_dim": 30,
    "lstm_units": [128, 64, 32],
    "dropout_rate": 0.3,
    "learning_rate": 0.001,
    "l1_reg": 0.0001,
    "l2_reg": 0.0001,
    "use_attention": True,
    "use_residual": True,
    "prediction_horizons": [12, 24, 96]  # 3h, 6h, 24h avec des bougies de 15min
}

# Fusionner avec les paramètres personnalisés
self.model_params = {**default_params, **(model_params or {})}

# Instancier le modèle
self.model = LSTMModel(**self.model_params)

# Instancier l'ingénieur de caractéristiques
self.feature_engineering = FeatureEngineering(save_scalers=True)

# Répertoires pour sauvegarder les résultats
self.output_dir = os.path.join(DATA_DIR, "models")
self.train_history_dir = os.path.join(self.output_dir, "training_history")

# Créer les répertoires si nécessaire
```

```
    os.makedirs(self.output_dir, exist_ok=True)
    os.makedirs(self.train_history_dir, exist_ok=True)

    # Historique des entraînements
    self.training_history = []

def prepare_data(self, data: pd.DataFrame) -> Tuple[pd.DataFrame, pd.DataFrame]:
    """
    Prépare les données pour l'entraînement en créant des caractéristiques

    Args:
        data: DataFrame avec les données OHLCV brutes

    Returns:
        DataFrame avec les caractéristiques avancées
    """
    # Créer les caractéristiques
    featured_data = self.feature_engineering.create_features(
        data,
        include_time_features=True,
        include_price_patterns=True
    )

    # Normaliser les caractéristiques
    normalized_data = self.feature_engineering.scale_features(
        featured_data,
        is_training=True,
        method='standard',
        feature_group='lstm'
    )
```

```
    return featured_data, normalized_data

def temporal_train_test_split(self, data: pd.DataFrame,
                              train_ratio: float = 0.7,
                              val_ratio: float = 0.15) -> Tuple[pd.DataFrame, pd.DataFrame, pd.DataFrame]:
    """
    Divise les données en ensembles d'entraînement, validation et test de manière temporelle

    Args:
        data: DataFrame avec les données
        train_ratio: Proportion des données pour l'entraînement
        val_ratio: Proportion des données pour la validation

    Returns:
        Tuple avec les DataFrames (train, val, test)
    """
    # Vérifier que les ratios sont valides
    if train_ratio + val_ratio >= 1.0:
        val_ratio = (1.0 - train_ratio) / 2
        logger.warning(f"Ratio de validation ajusté à {val_ratio}")

    test_ratio = 1.0 - train_ratio - val_ratio

    # Calculer les indices de séparation
    data_size = len(data)
    train_size = int(data_size * train_ratio)
    val_size = int(data_size * val_ratio)

    # Diviser en respectant l'ordre temporel
    train_data = data.iloc[:train_size]
    val_data = data.iloc[train_size:train_size+val_size]
```

```
    test_data = data.iloc[train_size+val_size:]

    return train_data, val_data, test_data

def create_temporal_cv_folds(self, data: pd.DataFrame,
                             n_splits: int = 5,
                             initial_train_ratio: float = 0.5,
                             stride: Optional[int] = None) -> List[Tuple[pd.DataFrame, pd.DataFrame]]:
    """
    Crée des plis de validation croisée temporelle

    Args:
        data: DataFrame avec les données
        n_splits: Nombre de plis à créer
        initial_train_ratio: Ratio initial des données pour l'entraînement
        stride: Pas entre les plis (en nombre de lignes), si None, calculé automatiquement

    Returns:
        Liste de tuples (train_data, val_data)
    """
    data_size = len(data)
    initial_train_size = int(data_size * initial_train_ratio)

    # Si stride n'est pas spécifié, calculer automatiquement
    if stride is None:
        remaining_size = data_size - initial_train_size
        stride = remaining_size // n_splits

    cv_folds = []

    for i in range(n_splits):
```

```
        # Calculer les indices de séparation pour ce pli
        train_end = initial_train_size + i * stride
        val_start = train_end
        val_end = min(val_start + stride, data_size)

        # Créer les ensembles d'entraînement et de validation
        train_data = data.iloc[:train_end]
        val_data = data.iloc[val_start:val_end]

        cv_folds.append((train_data, val_data))

    return cv_folds

def train_with_cv(self, data: pd.DataFrame,
                  n_splits: int = 5,
                  epochs: int = 50,
                  batch_size: int = 32,
                  initial_train_ratio: float = 0.5,
                  patience: int = 10) -> Dict:
    """
    Entraîne le modèle avec validation croisée temporelle

    Args:
        data: DataFrame avec les données OHLCV brutes
        n_splits: Nombre de plis de validation croisée
        epochs: Nombre d'époques d'entraînement
        batch_size: Taille du batch
        initial_train_ratio: Ratio initial des données pour l'entraînement
        patience: Patience pour l'early stopping

    Returns:
```

```
    Résultats de l'entraînement
"""
# Préparer les données
_, normalized_data = self.prepare_data(data)

# Créer les plis de validation croisée
cv_folds = self.create_temporal_cv_folds(
    normalized_data,
    n_splits=n_splits,
    initial_train_ratio=initial_train_ratio
)

# Résultats pour chaque pli
cv_results = []

for fold_idx, (train_data, val_data) in enumerate(cv_folds):
    logger.info(f"Entraînement sur le pli {fold_idx+1}/{n_splits}")

    # Préparer les données pour ce pli
    X_train, y_train = self.feature_engineering.create_multi_horizon_data(
        train_data,
        sequence_length=self.model_params["input_length"],
        horizons=self.model_params["prediction_horizons"],
        is_training=True
    )

    X_val, y_val = self.feature_engineering.create_multi_horizon_data(
        val_data,
        sequence_length=self.model_params["input_length"],
        horizons=self.model_params["prediction_horizons"],
        is_training=True
```

```
)

# Réinitialiser le modèle pour ce pli
self.model = LSTMModel(**self.model_params)

# Callbacks pour l'entraînement
callbacks = [
    EarlyStopping(
        monitor='val_loss',
        patience=patience,
        restore_best_weights=True
    ),
    ReduceLROnPlateau(
        monitor='val_loss',
        factor=0.5,
        patience=patience//2,
        min_lr=1e-6
    ),
    EarlyStoppingOnMemoryLeak(memory_threshold_mb=2000),
    ConceptDriftDetector(
        validation_data=(X_val, y_val),
        threshold=0.15,
        patience=3
    )
]

# Entraîner le modèle sur ce pli
history = self.model.model.fit(
    x=X_train,
    y=y_train,
    epochs=epochs,
```

```
            batch_size=batch_size,
            validation_data=(X_val, y_val),
            callbacks=callbacks,
            verbose=1
        )

        # Évaluer le modèle sur les données de validation
        evaluation = self.model.model.evaluate(X_val, y_val, verbose=1)

        # Stocker les résultats de ce pli
        fold_result = {
            "fold": fold_idx + 1,
            "train_samples": len(X_train),
            "val_samples": len(X_val),
            "history": history.history,
            "val_loss": evaluation[0],
            "metrics": {f"metric_{i}": metric for i, metric in enumerate(evaluation[1:])}
        }

        cv_results.append(fold_result)

        # Sauvegarder le modèle pour ce pli
        self.model.save(os.path.join(self.output_dir, f"lstm_fold_{fold_idx+1}.h5"))

    # Calculer les métriques moyennes sur tous les plis
    avg_val_loss = np.mean([result["val_loss"] for result in cv_results])

    # Sauvegarder les résultats
    cv_summary = {
        "n_splits": n_splits,
        "avg_val_loss": float(avg_val_loss),
```

```
        "results_per_fold": cv_results,
        "timestamp": datetime.now().isoformat()
    }

    # Sauvegarder l'historique d'entraînement
    history_filename = f"cv_history_{datetime.now().strftime('%Y%m%d_%H%M%S')}.json"
    history_path = os.path.join(self.train_history_dir, history_filename)

    with open(history_path, 'w') as f:
        json.dump(cv_summary, f, indent=2, default=str)

    logger.info(f"Résultats de la validation croisée sauvegardés: {history_path}")

    # Entraîner le modèle final sur toutes les données sauf le dernier pli (pour test)
    self.train_final_model(normalized_data, epochs, batch_size, test_ratio=0.15)

    return cv_summary

def train_final_model(self, data: pd.DataFrame,
                      epochs: int = 100,
                      batch_size: int = 32,
                      test_ratio: float = 0.15) -> Dict:
    """
    Entraîne le modèle final sur toutes les données sauf un ensemble de test

    Args:
        data: DataFrame avec les données normalisées
        epochs: Nombre d'époques d'entraînement
        batch_size: Taille du batch
        test_ratio: Ratio des données pour le test final
```

```
    Returns:
        Résultats de l'entraînement
    """
    # Diviser les données en train+val et test
    train_size = int(len(data) * (1 - test_ratio))
    train_val_data = data.iloc[:train_size]
    test_data = data.iloc[train_size:]

    # Diviser train_val en train et validation
    train_data, val_data = self.temporal_train_test_split(
        train_val_data,
        train_ratio=0.8,
        val_ratio=0.2
    )

    # Préparer les données
    X_train, y_train = self.feature_engineering.create_multi_horizon_data(
        train_data,
        sequence_length=self.model_params["input_length"],
        horizons=self.model_params["prediction_horizons"],
        is_training=True
    )

    X_val, y_val = self.feature_engineering.create_multi_horizon_data(
        val_data,
        sequence_length=self.model_params["input_length"],
        horizons=self.model_params["prediction_horizons"],
        is_training=True
    )

    X_test, y_test = self.feature_engineering.create_multi_horizon_data(
```

```
    test_data,
    sequence_length=self.model_params["input_length"],
    horizons=self.model_params["prediction_horizons"],
    is_training=True
)

# Réinitialiser le modèle
self.model = LSTMModel(**self.model_params)

# Callbacks pour l'entraînement
callbacks = [
    EarlyStopping(
        monitor='val_loss',
        patience=20,
        restore_best_weights=True
    ),
    ReduceLROnPlateau(
        monitor='val_loss',
        factor=0.5,
        patience=10,
        min_lr=1e-6
    ),
    ModelCheckpoint(
        filepath=os.path.join(self.output_dir, "production", "lstm_best.h5"),
        monitor='val_loss',
        save_best_only=True
    ),
    EarlyStoppingOnMemoryLeak(memory_threshold_mb=2000)
]

# Entraîner le modèle final
```

```
history = self.model.model.fit(
    x=X_train,
    y=y_train,
    epochs=epochs,
    batch_size=batch_size,
    validation_data=(X_val, y_val),
    callbacks=callbacks,
    verbose=1
)

# Évaluer le modèle sur les données de test
test_evaluation = self.model.model.evaluate(X_test, y_test, verbose=1)

# Calculer des métriques supplémentaires pour le test
y_pred = self.model.model.predict(X_test)

# Pour chaque horizon, calculer la précision de la direction
direction_accuracies = []

for h_idx, horizon in enumerate(self.model_params["prediction_horizons"]):
    # Indice de base pour la direction dans les sorties
    base_idx = h_idx * 4

    # Prédictions de direction (binaires)
    y_true_direction = y_test[base_idx]
    y_pred_direction = (y_pred[base_idx] > 0.5).astype(int)

    # Calculer l'accuracy
    accuracy = np.mean(y_true_direction == y_pred_direction.flatten())
    direction_accuracies.append(accuracy)
```

```
        # Sauvegarder les résultats
        final_results = {
            "train_samples": len(X_train),
            "val_samples": len(X_val),
            "test_samples": len(X_test),
            "history": history.history,
            "test_loss": test_evaluation[0],
            "direction_accuracies": direction_accuracies,
            "timestamp": datetime.now().isoformat()
        }

        # Sauvegarder l'historique d'entraînement
        history_filename =
f"final_model_history_{datetime.now().strftime('%Y%m%d_%H%M%S')}.json"
        history_path = os.path.join(self.train_history_dir, history_filename)

        with open(history_path, 'w') as f:
            json.dump(final_results, f, indent=2, default=str)

        logger.info(f"Résultats du modèle final sauvegardés: {history_path}")

        # Sauvegarder le modèle final
        self.model.save(os.path.join(self.output_dir, "production", "lstm_final.h5"))

        # Générer des visualisations
        self._generate_training_visualizations(history.history, final_results)

        return final_results

    def _generate_training_visualizations(self, history: Dict, results: Dict) -> None:
        """
```

```
Génère des visualisations de l'entraînement du modèle

Args:
    history: Historique d'entraînement
    results: Résultats du modèle
"""
# Créer le répertoire pour les visualisations
viz_dir = os.path.join(self.output_dir, "visualizations")
os.makedirs(viz_dir, exist_ok=True)

# 1. Courbe d'apprentissage (loss)
plt.figure(figsize=(12, 6))
plt.plot(history['loss'], label='Train Loss')
plt.plot(history['val_loss'], label='Validation Loss')
plt.title('Courbe d\'apprentissage du modèle LSTM')
plt.xlabel('Époque')
plt.ylabel('Loss')
plt.legend()
plt.grid(True, alpha=0.3)

# Ajouter des annotations
min_val_loss = min(history['val_loss'])
min_val_loss_epoch = history['val_loss'].index(min_val_loss)

plt.annotate(f'Min Val Loss: {min_val_loss:.4f}',
             xy=(min_val_loss_epoch, min_val_loss),
             xytext=(min_val_loss_epoch+5, min_val_loss*1.1),
             arrowprops=dict(facecolor='black', shrink=0.05, width=1.5, headwidth=8),
             fontsize=10)

# Sauvegarder la figure
```

```
plt.savefig(os.path.join(viz_dir, "learning_curve.png"))
plt.close()

# 2. Précision de la direction par horizon
plt.figure(figsize=(10, 6))
horizons = self.model_params["prediction_horizons"]
accuracies = results["direction_accuracies"]

plt.bar(range(len(horizons)), accuracies, color='skyblue')
plt.xticks(range(len(horizons)), [f"{h}" for h in horizons])
plt.title('Précision de la Direction par Horizon')
plt.xlabel('Horizon (périodes)')
plt.ylabel('Précision')
plt.ylim([0, 1])

# Ajouter les valeurs sur les barres
for i, v in enumerate(accuracies):
    plt.text(i, v + 0.02, f"{v:.2f}", ha='center')

plt.grid(True, alpha=0.3, axis='y')
plt.tight_layout()

# Sauvegarder la figure
plt.savefig(os.path.join(viz_dir, "direction_accuracy.png"))
plt.close()

# 3. Évolution du taux d'apprentissage
if 'lr' in history:
    plt.figure(figsize=(10, 4))
    plt.semilogy(history['lr'])
    plt.title('Évolution du Taux d\'Apprentissage')
```

```
        plt.xlabel('Époque')
        plt.ylabel('Learning Rate')
        plt.grid(True, alpha=0.3)

        # Sauvegarder la figure
        plt.savefig(os.path.join(viz_dir, "learning_rate.png"))
        plt.close()

================================================
File: crypto_trading_bot_CLAUDE/ai/models/model_validator.py
================================================
# ai/models/model_validator.py
"""
Module de validation et d'évaluation des performances du modèle LSTM
"""
import os
import numpy as np
import pandas as pd
from typing import Dict, List, Tuple, Union, Optional
from datetime import datetime, timedelta
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score,
confusion_matrix
import json

from ai.models.lstm_model import LSTMModel
from ai.models.feature_engineering import FeatureEngineering
from config.config import DATA_DIR
from utils.logger import setup_logger
from strategies.technical_bounce import TechnicalBounceStrategy
from strategies.market_state import MarketStateAnalyzer
```

```
from ai.scoring_engine import ScoringEngine

logger = setup_logger("model_validator")

class ModelValidator:
    """
    Classe pour valider et évaluer les performances du modèle LSTM
    et les comparer aux stratégies de base
    """
    def __init__(self, model: Optional[LSTMModel] = None,
                 feature_engineering: Optional[FeatureEngineering] = None):
        """
        Initialise le ModelValidator

        Args:
            model: Instance du modèle LSTM à valider
            feature_engineering: Instance du module d'ingénierie des caractéristiques
        """
        self.model = model
        self.feature_engineering = feature_engineering or FeatureEngineering()

        # Pour la comparaison avec la stratégie de base
        self.scoring_engine = ScoringEngine()

        # Répertoires pour les résultats
        self.output_dir = os.path.join(DATA_DIR, "models", "validation")
        os.makedirs(self.output_dir, exist_ok=True)

    def load_model(self, model_path: str) -> None:
        """
        Charge un modèle sauvegardé
```

```
    Args:
        model_path: Chemin vers le modèle sauvegardé
    """
    # Si aucun modèle n'est fourni, en créer un nouveau
    if self.model is None:
        self.model = LSTMModel()

    # Charger le modèle
    self.model.load(model_path)
    logger.info(f"Modèle chargé: {model_path}")

def evaluate_on_test_set(self, test_data: pd.DataFrame) -> Dict:
    """
    Évalue le modèle sur un ensemble de test

    Args:
        test_data: DataFrame avec les données de test

    Returns:
        Dictionnaire avec les métriques d'évaluation
    """
    if self.model is None:
        raise ValueError("Le modèle n'a pas été initialisé ou chargé.")

    # Préparer les données de test
    featured_data, normalized_data = self.prepare_data(test_data)

    # Créer des séquences pour chaque horizon de prédiction
    X_test, y_test = self.feature_engineering.create_multi_horizon_data(
        normalized_data,
```

```
    sequence_length=self.model.input_length,
    horizons=self.model.prediction_horizons,
    is_training=True
)

# Évaluer le modèle
evaluation = self.model.model.evaluate(X_test, y_test, verbose=1)

# Faire des prédictions
predictions = self.model.model.predict(X_test)

# Calculer des métriques détaillées pour chaque horizon
results = {
    "loss": evaluation[0],
    "horizons": {}
}

# Pour chaque horizon
for h_idx, horizon in enumerate(self.model.prediction_horizons):
    horizon_key = f"horizon_{horizon}"
    results["horizons"][horizon_key] = {}

    # Indice de base pour cet horizon
    base_idx = h_idx * 4

    # 1. Direction (classification binaire)
    y_true_direction = y_test[base_idx]
    y_pred_direction = (predictions[base_idx] > 0.5).astype(int).flatten()

    accuracy = accuracy_score(y_true_direction, y_pred_direction)
    precision = precision_score(y_true_direction, y_pred_direction, zero_division=0)
```

```
        recall = recall_score(y_true_direction, y_pred_direction, zero_division=0)
        f1 = f1_score(y_true_direction, y_pred_direction, zero_division=0)

        results["horizons"][horizon_key]["direction"] = {
            "accuracy": float(accuracy),
            "precision": float(precision),
            "recall": float(recall),
            "f1_score": float(f1),
            "confusion_matrix": confusion_matrix(y_true_direction, y_pred_direction).tolist()
        }

        # 2. Volatilité (régression)
        y_true_volatility = y_test[base_idx + 1]
        y_pred_volatility = predictions[base_idx + 1].flatten()

        mae = np.mean(np.abs(y_true_volatility - y_pred_volatility))
        mse = np.mean((y_true_volatility - y_pred_volatility) ** 2)

        results["horizons"][horizon_key]["volatility"] = {
            "mae": float(mae),
            "mse": float(mse),
            "rmse": float(np.sqrt(mse))
        }

        # 3. Volume (

================================================
File: crypto_trading_bot_CLAUDE/config/config.py
================================================
"""
Configuration globale du bot de trading crypto
```

```
"""
import os
from dotenv import load_dotenv
import logging

# Chargement des variables d'environnement
load_dotenv()

# Configuration de l'API Binance
BINANCE_API_KEY = os.getenv("BINANCE_API_KEY", "")
BINANCE_API_SECRET = os.getenv("BINANCE_API_SECRET", "")
USE_TESTNET = os.getenv("USE_TESTNET", "True").lower() in ("true", "1", "t")

# Paramètres généraux
INITIAL_CAPITAL = 200  # USDT
BASE_CURRENCY = "USDT"
TRADING_PAIRS = ["BTCUSDT", "ETHUSDT", "SOLUSDT", "AVAXUSDT", "BNBUSDT"]
DEFAULT_TRADING_PAIR = "BTCUSDT"

# Intervalles de temps pour l'analyse
PRIMARY_TIMEFRAME = "15m"  # Timeframe principal (15 minutes)
SECONDARY_TIMEFRAMES = ["1h", "4h"]  # Timeframes secondaires pour confirmation

# Chemins des répertoires
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
DATA_DIR = os.path.join(BASE_DIR, "data")
LOG_DIR = os.path.join(BASE_DIR, "logs")

# Création des répertoires s'ils n'existent pas
for directory in [DATA_DIR, LOG_DIR]:
    if not os.path.exists(directory):
```

```
    os.makedirs(directory)

# Configuration du logging
LOG_LEVEL = logging.INFO
LOG_FORMAT = "%(asctime)s - %(name)s - %(levelname)s - %(message)s"
LOG_FILE = os.path.join(LOG_DIR, "trading_bot.log")

# Paramètres du système
MAX_API_RETRIES = 3
API_RETRY_DELAY = 2  # secondes

# Paramètres de notification (à implémenter ultérieurement)
ENABLE_NOTIFICATIONS = False
NOTIFICATION_EMAIL = os.getenv("NOTIFICATION_EMAIL", "")



================================================
File: crypto_trading_bot_CLAUDE/config/trading_params.py
================================================
"""
Paramètres spécifiques à la stratégie de trading
Ces paramètres sont conçus pour être ajustés par l'IA auto-adaptative
"""

# Paramètres de gestion des risques
RISK_PER_TRADE_PERCENT = 3.5  # Pourcentage du capital risqué par trade (5-10%)
MAX_CONCURRENT_TRADES = 3  # Nombre maximum de trades simultanés
MAX_DAILY_TRADES = 25  # Nombre maximum de trades par jour (20-30)
LEVERAGE = 3  # Effet de levier (jusqu'à 5x)

# Paramètres des ordres
```

```
STOP_LOSS_PERCENT = 3.3  # Pourcentage de stop-loss (3-5%)
TAKE_PROFIT_PERCENT = 7.5  # Pourcentage de take-profit (5-7%)
TRAILING_STOP_ACTIVATION = 1.2  # Activation du trailing stop après x% de profit
TRAILING_STOP_STEP = 0.5 # Pas du trailing stop
MAX_DRAWDOWN_LIMIT = 30.0     # Ajout d'une limite de drawdown

# Délais et cooldown
MIN_TIME_BETWEEN_TRADES = 4  # Minutes minimum entre trades (3-5 minutes)
MARKET_COOLDOWN_PERIOD = 60  # Minutes de cooldown après détection de marché défavorable

# Paramètres des indicateurs techniques (valeurs par défaut, ajustables par l'IA)
# RSI
RSI_PERIOD = 14
RSI_OVERSOLD = 30  # Seuil de survente
RSI_OVERBOUGHT = 70  # Seuil de surachat

# Bandes de Bollinger
BB_PERIOD = 20
BB_DEVIATION = 2.1

# ATR pour mesure de volatilité
ATR_PERIOD = 14

# EMA pour détection de tendance
EMA_SHORT = 9
EMA_MEDIUM = 21
EMA_LONG = 50

# ADX pour force de tendance
ADX_PERIOD = 14
ADX_THRESHOLD = 25  # Seuil pour considérer une tendance comme forte
```

```
# Paramètres du système de scoring
MINIMUM_SCORE_TO_TRADE = 72  # Score minimum pour entrer en position (0-100)

# Facteurs d'adaptation de l'IA
LEARNING_RATE = 0.05  # Taux d'apprentissage pour l'ajustement des paramètres

================================================
File: crypto_trading_bot_CLAUDE/core/adaptive_risk_manager.py
================================================
# core/adaptive_risk_manager.py
"""
Gestionnaire de risque adaptatif avancé qui ajuste dynamiquement
les paramètres de trading en fonction des conditions de marché,
des prédictions du modèle et de l'historique des trades
"""
import os
import json
import numpy as np
import pandas as pd
from typing import Dict, List, Tuple, Optional, Union
from datetime import datetime, timedelta
import math

from config.config import DATA_DIR
from config.trading_params import (
    RISK_PER_TRADE_PERCENT,
    STOP_LOSS_PERCENT,
    TAKE_PROFIT_PERCENT,
    LEVERAGE
)
```

```
from ai.market_anomaly_detector import MarketAnomalyDetector
from utils.logger import setup_logger

logger = setup_logger("adaptive_risk_manager")

class AdaptiveRiskManager:
    """
    Gestionnaire de risque adaptatif qui ajuste dynamiquement les paramètres de trading

    Caractéristiques:
    - Ajuste la taille des positions en fonction de la confiance du modèle
    - Adapte les niveaux de stop-loss et take-profit selon la volatilité prédite
    - Devient plus conservateur après des séquences de pertes
    - Détecte les conditions de marché extrêmes et réduit l'exposition
    - Intègre un système anti-fragile qui apprend des pertes passées
    """
    def __init__(self,
                 initial_capital: float = 200,
                 max_open_positions: int = 3,
                 max_risk_per_day: float = 15.0,  # % max du capital risqué par jour
                 recovery_factor: float = 0.5,     # Facteur de récupération après une perte
                 enable_martingale: bool = False,   # Activer la stratégie de martingale (dangereux)
                 enable_anti_martingale: bool = True,  # Stratégie anti-martingale (plus sûre)
                 volatility_scaling: bool = True,   # Ajuster le risque selon la volatilité
                 use_kelly_criterion: bool = True,  # Utiliser le critère de Kelly pour le sizing
                 kelly_fraction: float = 0.5,      # Fraction du critère de Kelly (0.5 = demi-Kelly)
                 risk_control_mode: str = "balanced"  # "conservative", "balanced", "aggressive"
                 ):
        """
        Initialise le gestionnaire de risque adaptatif
```

```
Args:
    initial_capital: Capital initial en USDT
    max_open_positions: Nombre max de positions ouvertes simultanément
    max_risk_per_day: Pourcentage max du capital risqué par jour
    recovery_factor: Facteur de réduction du risque après une perte
    enable_martingale: Active la stratégie de martingale (augmente le risque après une perte)
    enable_anti_martingale: Active la stratégie anti-martingale (augmente la taille après un gain)
    volatility_scaling: Ajuster la taille en fonction de la volatilité
    use_kelly_criterion: Utiliser le critère de Kelly pour le sizing optimal
    kelly_fraction: Fraction du critère de Kelly à utiliser (1.0 = Kelly complet)
    risk_control_mode: Mode de contrôle du risque (conservateur/équilibré/agressif)
"""
# Paramètres de base
self.initial_capital = initial_capital
self.current_capital = initial_capital
self.max_open_positions = max_open_positions
self.max_risk_per_day = max_risk_per_day
self.recovery_factor = recovery_factor
self.enable_martingale = enable_martingale
self.enable_anti_martingale = enable_anti_martingale
self.volatility_scaling = volatility_scaling
self.use_kelly_criterion = use_kelly_criterion
self.kelly_fraction = kelly_fraction

# Historique de trading
self.trade_history = []
self.daily_risk_used = 0.0
self.last_risk_reset = datetime.now()
self.consecutive_losses = 0
self.consecutive_wins = 0
```

```
    # Profils de risque dynamiques
    self.risk_profiles = self._initialize_risk_profiles(risk_control_mode)
    self.current_risk_profile = "balanced"  # Commence en mode équilibré

    # Paramètres par défaut si aucun modèle LSTM n'est disponible
    self.default_params = {
        "risk_per_trade": RISK_PER_TRADE_PERCENT,
        "stop_loss": STOP_LOSS_PERCENT,
        "take_profit": TAKE_PROFIT_PERCENT,
        "leverage": LEVERAGE
    }

    # Indicateurs d'état du système
    self.market_state = "normal"  # normal, volatile, extreme
    self.risk_capacity = 1.0  # Facteur multiplicateur de risque (0.1 à 1.0)

    # Détecteur d'anomalies de marché
    self.anomaly_detector = MarketAnomalyDetector(
        model_dir=os.path.join(DATA_DIR, "models", "anomaly")
    )

    # Initialiser le journal pour la traçabilité des décisions
    self.risk_log = []

    # Charger l'historique précédent si disponible
    self._load_history()

def _initialize_risk_profiles(self, mode: str) -> Dict:
    """
    Initialise les profils de risque pour différentes conditions de marché
```

```
Args:
    mode: Mode de contrôle du risque (conservative/balanced/aggressive)

Returns:
    Dictionnaire des profils de risque
"""
# Base des profils de risque
profiles = {
    # Profil par défaut/équilibré
    "balanced": {
        "risk_per_trade_percent": RISK_PER_TRADE_PERCENT,
        "stop_loss_percent": STOP_LOSS_PERCENT,
        "take_profit_percent": TAKE_PROFIT_PERCENT,
        "leverage": LEVERAGE,
        "trailing_stop_activation": 2.0,  # % de profit pour activer le trailing stop
        "trailing_stop_step": 0.5,       # % de distance pour le trailing stop
        "risk_scaling_factor": 1.0       # Facteur d'échelle pour le risque
    },

    # Profil conservateur pour les conditions volatiles
    "conservative": {
        "risk_per_trade_percent": RISK_PER_TRADE_PERCENT * 0.5,
        "stop_loss_percent": STOP_LOSS_PERCENT * 1.25,  # Stop loss plus large
        "take_profit_percent": TAKE_PROFIT_PERCENT * 1.25, # TP plus large
        "leverage": max(1, LEVERAGE - 1),    # Levier réduit
        "trailing_stop_activation": 1.5,
        "trailing_stop_step": 0.3,
        "risk_scaling_factor": 0.5
    },

    # Profil ultra-conservateur pour les conditions extrêmes
```

```
"defensive": {
    "risk_per_trade_percent": RISK_PER_TRADE_PERCENT * 0.25,
    "stop_loss_percent": STOP_LOSS_PERCENT * 1.5,
    "take_profit_percent": TAKE_PROFIT_PERCENT * 1.5,
    "leverage": 1,  # Pas de levier en mode défensif
    "trailing_stop_activation": 1.0,
    "trailing_stop_step": 0.2,
    "risk_scaling_factor": 0.25
},

# Profil agressif pour les conditions favorables
"aggressive": {
    "risk_per_trade_percent": min(10.0, RISK_PER_TRADE_PERCENT * 1.25),
    "stop_loss_percent": STOP_LOSS_PERCENT * 0.8,
    "take_profit_percent": TAKE_PROFIT_PERCENT * 0.8,
    "leverage": min(5, LEVERAGE + 1),
    "trailing_stop_activation": 3.0,
    "trailing_stop_step": 0.8,
    "risk_scaling_factor": 1.25
},

# Profil très agressif pour les conditions très favorables
"very_aggressive": {
    "risk_per_trade_percent": min(12.5, RISK_PER_TRADE_PERCENT * 1.5),
    "stop_loss_percent": STOP_LOSS_PERCENT * 0.7,
    "take_profit_percent": TAKE_PROFIT_PERCENT * 0.7,
    "leverage": min(5, LEVERAGE + 2),
    "trailing_stop_activation": 4.0,
    "trailing_stop_step": 1.0,
    "risk_scaling_factor": 1.5
}
```

```
    }

    # Ajuster les profils en fonction du mode sélectionné
    if mode == "conservative":
        # Rendre tous les profils plus conservateurs
        for profile in profiles.values():
            profile["risk_per_trade_percent"] *= 0.8
            profile["leverage"] = max(1, profile["leverage"] - 1)
            profile["risk_scaling_factor"] *= 0.8

    elif mode == "aggressive":
        # Rendre tous les profils plus agressifs
        for profile in profiles.values():
            profile["risk_per_trade_percent"] *= 1.2
            profile["leverage"] = min(5, profile["leverage"] + 1)
            profile["risk_scaling_factor"] *= 1.2

    return profiles

def update_account_balance(self, account_info: Dict) -> None:
    """
    Met à jour le capital disponible

    Args:
        account_info: Informations sur le compte
    """
    # Mettre à jour le capital
    if "totalWalletBalance" in account_info:
        # Pour Binance Futures
        self.current_capital = float(account_info["totalWalletBalance"])
    elif "totalBalance" in account_info:
```

```
            # Pour Binance Spot
            self.current_capital = float(account_info["totalBalance"])
        else:
            logger.warning("Format d'information de compte non reconnu")

        logger.info(f"Capital mis à jour: {self.current_capital} USDT")

        # Réinitialiser le risque quotidien si nécessaire
        current_time = datetime.now()
        if (current_time - self.last_risk_reset).days >= 1:
            self.daily_risk_used = 0.0
            self.last_risk_reset = current_time
            logger.info("Limite de risque quotidien réinitialisée")

    def can_open_new_position(self, position_tracker) -> Dict:
        """
        Vérifie si une nouvelle position peut être ouverte selon les règles de gestion des risques

        Args:
            position_tracker: Objet qui suit les positions ouvertes

        Returns:
            Dictionnaire avec décision et raison
        """
        # 1. Vérifier le nombre de positions ouvertes
        open_positions = position_tracker.get_all_open_positions()
        total_open_positions = sum(len(positions) for positions in open_positions.values())

        if total_open_positions >= self.max_open_positions:
            return {
                "can_open": False,
```

```
            "reason": f"Nombre maximum de positions atteint ({self.max_open_positions})"
        }

    # 2. Vérifier le risque quotidien utilisé
    max_risk_amount = self.current_capital * (self.max_risk_per_day / 100)

    if self.daily_risk_used >= max_risk_amount:
        return {
            "can_open": False,
            "reason": f"Limite de risque quotidien atteinte ({self.max_risk_per_day}% du capital)"
        }

    # 3. Vérifier l'état du marché
    if self.market_state == "extreme":
        # En conditions extrêmes, être plus restrictif
        if self.current_risk_profile != "defensive":
            self.set_risk_profile("defensive")

        if total_open_positions > 0:
            return {
                "can_open": False,
                "reason": "Conditions de marché extrêmes - aucune nouvelle position"
            }

    # 4. Ajuster en fonction de l'historique récent
    if self.consecutive_losses >= 3:
        # Après 3 pertes consécutives, être plus conservateur
        if self.current_risk_profile not in ["conservative", "defensive"]:
            self.set_risk_profile("conservative")
            logger.info("Passage en mode conservateur après 3 pertes consécutives")
```

```
    elif self.consecutive_wins >= 3:
        # Après 3 gains consécutifs, être légèrement plus agressif
        if self.current_risk_profile == "balanced":
            self.set_risk_profile("aggressive")
            logger.info("Passage en mode agressif après 3 gains consécutifs")

    # 5. Vérifier le capital minimum
    min_capital_required = 50  # Montant minimum pour trader raisonnablement

    if self.current_capital < min_capital_required:
        return {
            "can_open": False,
            "reason": f"Capital insuffisant ({self.current_capital} < {min_capital_required} USDT)"
        }

    return {
        "can_open": True,
        "risk_profile": self.current_risk_profile,
        "available_risk": max_risk_amount - self.daily_risk_used
    }

def calculate_position_size(self, symbol: str, opportunity: Dict,
                            lstm_prediction: Optional[Dict] = None) -> float:
    """
    Calcule la taille optimale de position en fonction de multiples facteurs

    Args:
        symbol: Paire de trading
        opportunity: Opportunité de trading détectée
        lstm_prediction: Prédictions du modèle LSTM (optionnel)
```

```
Returns:
    Taille de position en USDT
"""
# Obtenir les paramètres du profil de risque actuel
profile = self.risk_profiles[self.current_risk_profile]

# Paramètres de base
base_risk_percent = profile["risk_per_trade_percent"]
risk_factor = profile["risk_scaling_factor"]
stop_loss_percent = profile["stop_loss_percent"]
leverage = profile["leverage"]

# 1. Ajuster le risque en fonction du score de l'opportunité
score = opportunity.get("score", 70)
score_factor = self._calculate_score_factor(score)

# 2. Intégrer les prédictions LSTM si disponibles
model_confidence = 0.5  # Valeur par défaut (neutre)
volatility_factor = 1.0

if lstm_prediction:
    # Trouver l'horizon pertinent (court terme pour le sizing)
    short_term = None
    for horizon_name, prediction in lstm_prediction.items():
        if horizon_name in ["3h", "4h", "short_term", "horizon_12"]:
            short_term = prediction
            break

    if short_term:
        # Confiance dans la direction prédite
        direction_prob = short_term.get("direction_probability", 50) / 100
```

```
        # Ajuster la confiance en fonction de l'écart par rapport à 0.5
        model_confidence = abs(direction_prob - 0.5) * 2

        # Si la direction prédite est opposée à celle de l'opportunité, réduire la taille
        predicted_direction = direction_prob > 0.5
        opportunity_direction = opportunity.get("side", "BUY") == "BUY"

        if predicted_direction != opportunity_direction:
            model_confidence = 0.2  # Faible confiance en cas de contradiction

        # Ajuster en fonction de la volatilité prédite
        predicted_volatility = short_term.get("predicted_volatility", 0.03)
        volatility_factor = self._volatility_adjustment(predicted_volatility)

# 3. Calcul de la taille de base (basée sur le risque)
risk_amount = self.current_capital * (base_risk_percent / 100) * risk_factor

# 4. Appliquer le facteur de Kelly si activé
if self.use_kelly_criterion and len(self.trade_history) >= 10:
    kelly_size = self._calculate_kelly_criterion()
    # Appliquer la fraction de Kelly
    kelly_adjusted_risk = risk_amount * kelly_size * self.kelly_fraction
    risk_amount = min(risk_amount, kelly_adjusted_risk)

# 5. Ajuster avec les facteurs de confiance
confidence_factor = (score_factor + model_confidence) / 2
adjusted_risk = risk_amount * confidence_factor

# 6. Appliquer le facteur de volatilité si activé
if self.volatility_scaling:
```

```
    adjusted_risk *= volatility_factor

# 7. Appliquer les stratégies de martingale/anti-martingale si activées
if self.consecutive_losses > 0 and self.enable_martingale:
    # Augmenter légèrement la taille après une perte (risqué!)
    martingale_factor = 1.0 + (0.1 * min(self.consecutive_losses, 3))
    adjusted_risk *= martingale_factor

elif self.consecutive_wins > 0 and self.enable_anti_martingale:
    # Augmenter la taille après un gain
    anti_martingale_factor = 1.0 + (0.15 * min(self.consecutive_wins, 3))
    adjusted_risk *= anti_martingale_factor

# 8. Ajuster en fonction de la capacity de risque globale
adjusted_risk *= self.risk_capacity

# 9. Limites de sécurité pour éviter les overrides manuels
# Plafond à 15% du capital quelle que soit la configuration
max_risk_allowed = self.current_capital * 0.15
adjusted_risk = min(adjusted_risk, max_risk_allowed)

# Calculer la taille finale
# Position size = (Capital * Risk%) / (StopLoss% * Leverage)
position_size = adjusted_risk / (stop_loss_percent / 100) * leverage

# Limiter la taille aux fonds disponibles
position_size = min(position_size, self.current_capital * 0.95)

# Mettre à jour le risque quotidien utilisé
self.daily_risk_used += adjusted_risk
```

```
    # Journaliser la décision
    self._log_position_sizing(
        symbol=symbol,
        base_risk=risk_amount,
        final_size=position_size,
        factors={
            "score_factor": score_factor,
            "model_confidence": model_confidence,
            "volatility_factor": volatility_factor,
            "risk_capacity": self.risk_capacity,
            "kelly_criterion": self.use_kelly_criterion,
            "profile": self.current_risk_profile
        }
    )

    return position_size

def update_stop_loss(self, symbol: str, original_stop: float,
                     current_price: float, lstm_prediction: Dict) -> Dict:
    """
    Calcule un niveau de stop-loss adaptatif basé sur la volatilité prédite

    Args:
        symbol: Paire de trading
        original_stop: Niveau de stop-loss original
        current_price: Prix actuel
        lstm_prediction: Prédictions du modèle LSTM

    Returns:
        Nouveau niveau de stop-loss et raisonnement
    """
```

```
# Si pas de prédiction LSTM disponible, conserver le stop original
if not lstm_prediction:
    return {
        "stop_level": original_stop,
        "updated": False,
        "reason": "Aucune prédiction LSTM disponible"
    }

# Trouver l'horizon pertinent (court terme)
short_term = None
for horizon_name, prediction in lstm_prediction.items():
    if horizon_name in ["3h", "4h", "short_term", "horizon_12"]:
        short_term = prediction
        break

if not short_term:
    return {
        "stop_level": original_stop,
        "updated": False,
        "reason": "Aucune prédiction court terme disponible"
    }

# Récupérer la volatilité prédite
predicted_volatility = short_term.get("predicted_volatility", 0.03)

# Calculer la différence en pourcentage entre le prix actuel et le stop original
original_stop_percent = abs((current_price - original_stop) / current_price * 100)

# Ajuster en fonction de la volatilité prédite
# Plus la volatilité est élevée, plus le stop doit être large
volatility_percent = predicted_volatility * 100
```

```
# Facteur d'ajustement: si la volatilité prédite est élevée, augmenter le stop
# si elle est faible, réduire le stop
adjustment_factor = volatility_percent / 3.0  # 3% est considéré comme une volatilité standard

# Calculer le nouveau stop en pourcentage
new_stop_percent = original_stop_percent * adjustment_factor

# Limites de sécurité
min_stop_percent = 1.0  # Minimum 1%
max_stop_percent = 10.0  # Maximum 10%

new_stop_percent = max(min_stop_percent, min(new_stop_percent, max_stop_percent))

# Calculer le nouveau niveau de stop-loss
side = "BUY" if current_price > original_stop else "SELL"

if side == "BUY":
    # Pour les positions longues, le stop est en dessous du prix
    new_stop_level = current_price * (1 - new_stop_percent / 100)
else:
    # Pour les positions courtes, le stop est au-dessus du prix
    new_stop_level = current_price * (1 + new_stop_percent / 100)

# Ne pas déplacer le stop dans la mauvaise direction
if side == "BUY" and new_stop_level < original_stop:
    new_stop_level = original_stop
elif side == "SELL" and new_stop_level > original_stop:
    new_stop_level = original_stop

return {
```

```
        "stop_level": new_stop_level,
        "updated": new_stop_level != original_stop,
        "reason": f"Ajustement basé sur volatilité prédite de {volatility_percent:.2f}%",
        "volatility_percent": volatility_percent,
        "adjustment_factor": adjustment_factor
    }

def update_take_profit(self, symbol: str, original_tp: float,
                       current_price: float, lstm_prediction: Dict) -> Dict:
    """
    Calcule un niveau de take-profit adaptatif basé sur le momentum et la volatilité prédits

    Args:
        symbol: Paire de trading
        original_tp: Niveau de take-profit original
        current_price: Prix actuel
        lstm_prediction: Prédictions du modèle LSTM

    Returns:
        Nouveau niveau de take-profit et raisonnement
    """
    # Si pas de prédiction LSTM disponible, conserver le TP original
    if not lstm_prediction:
        return {
            "tp_level": original_tp,
            "updated": False,
            "reason": "Aucune prédiction LSTM disponible"
        }

    # Récupérer les prédictions à court et moyen terme
    short_term = None
```

```
medium_term = None

for horizon_name, prediction in lstm_prediction.items():
    if horizon_name in ["3h", "4h", "short_term", "horizon_12"]:
        short_term = prediction
    elif horizon_name in ["12h", "24h", "medium_term", "horizon_48"]:
        medium_term = prediction

if not short_term or not medium_term:
    return {
        "tp_level": original_tp,
        "updated": False,
        "reason": "Prédictions insuffisantes"
    }

# Récupérer les indicateurs pertinents
short_momentum = short_term.get("predicted_momentum", 0.0)
medium_momentum = medium_term.get("predicted_momentum", 0.0)

# Moyenne pondérée du momentum (plus de poids au court terme)
momentum_score = (short_momentum * 0.7 + medium_momentum * 0.3)

# Volatilité prédite
volatility = short_term.get("predicted_volatility", 0.03) * 100  # en pourcentage

# Calculer la différence en pourcentage entre le prix actuel et le TP original
original_tp_percent = abs((original_tp - current_price) / current_price * 100)

# Ajuster en fonction du momentum et de la volatilité
# Plus le momentum est fort, plus le TP peut être agressif
momentum_factor = 1.0 + (abs(momentum_score) * 0.5)
```

```
# Ajuster également en fonction de la volatilité
volatility_factor = max(0.8, min(1.5, volatility / 3.0))

# Calculer le nouveau TP en pourcentage
new_tp_percent = original_tp_percent * momentum_factor * volatility_factor

# Limites de sécurité
min_tp_percent = 2.0  # Minimum 2%
max_tp_percent = 15.0  # Maximum 15%

new_tp_percent = max(min_tp_percent, min(new_tp_percent, max_tp_percent))

# Calculer le nouveau niveau de take-profit
side = "BUY" if original_tp > current_price else "SELL"

if side == "BUY":
    # Pour les positions longues, le TP est au-dessus du prix
    new_tp_level = current_price * (1 + new_tp_percent / 100)
else:
    # Pour les positions courtes, le TP est en dessous du prix
    new_tp_level = current_price * (1 - new_tp_percent / 100)

# Ne pas déplacer le TP dans la mauvaise direction
if side == "BUY" and new_tp_level < original_tp:
    new_tp_level = original_tp
elif side == "SELL" and new_tp_level > original_tp:
    new_tp_level = original_tp

return {
    "tp_level": new_tp_level,
```

```
            "updated": new_tp_level != original_tp,
            "reason": f"Ajustement basé sur momentum ({momentum_score:.2f}) et volatilité ({volatility:.2f}%)",
            "momentum_score": momentum_score,
            "volatility_percent": volatility,
            "momentum_factor": momentum_factor,
            "volatility_factor": volatility_factor
        }

    def detect_extreme_market_conditions(self, data_fetcher, symbol: str) -> Dict:
        """
        Détecte des conditions de marché extrêmes qui nécessitent une réduction de l'exposition

        Args:
            data_fetcher: Instance du gestionnaire de données de marché
            symbol: Paire de trading

        Returns:
            Résultat de la détection
        """
        try:
            # Récupérer les données de marché
            market_data = data_fetcher.get_market_data(symbol)

            # Utiliser le détecteur d'anomalies
            anomaly_result = self.anomaly_detector.detect_anomalies(
                market_data["primary_timeframe"]["ohlcv"],
                current_price=market_data["current_price"],
                return_details=True
            )
```

```
        if anomaly_result["detected"]:
            # Condition de marché extrême détectée
            self.market_state = "extreme"

            # Ajuster le profil de risque
            self.set_risk_profile("defensive")

            # Réduire la capacité de risque
            self.risk_capacity = 0.2  # Réduction drastique

            return {
                "extreme_condition": True,
                "reason": anomaly_result["reason"],
                "action_taken": "Passage en mode défensif",
                "risk_capacity": self.risk_capacity,
                "anomaly_details": anomaly_result["details"]
            }
        else:
            # Vérifier si c'est juste volatile mais pas extrême
            volatility_detected = False

            # Vérifier les indicateurs de volatilité
            atr_percent = market_data["primary_timeframe"]["indicators"]["atr"][-1] /
market_data["current_price"] * 100

            if atr_percent > 5.0:  # 5% est considéré comme très volatile
                volatility_detected = True
                self.market_state = "volatile"
                self.set_risk_profile("conservative")
                self.risk_capacity = 0.5
            else:
```

```
            # Conditions normales
            self.market_state = "normal"

            # Si on était en mode défensif, revenir en mode équilibré
            if self.current_risk_profile == "defensive":
                self.set_risk_profile("conservative")

            # Restaurer progressivement la capacité de risque
            self.risk_capacity = min(1.0, self.risk_capacity + 0.1)

        return {
            "extreme_condition": False,
            "volatile_condition": volatility_detected,
            "market_state": self.market_state,
            "risk_capacity": self.risk_capacity
        }

    except Exception as e:
        logger.error(f"Erreur lors de la détection des conditions extrêmes: {str(e)}")
        return {
            "extreme_condition": False,
            "error": str(e)
        }

def should_close_early(self, symbol: str, position: Dict, current_price: float,
                       lstm_prediction: Optional[Dict] = None) -> Dict:
    """
    Détermine si une position doit être fermée de manière anticipée
    en fonction des prédictions du modèle ou des conditions de marché

    Args:
```

```
        symbol: Paire de trading
        position: Données de la position ouverte
        current_price: Prix actuel
        lstm_prediction: Prédictions du modèle LSTM

    Returns:
        Décision de fermeture anticipée avec raison
    """
    # Récupérer les détails de la position
    side = position.get("side", "BUY")
    entry_price = position.get("entry_price", current_price)
    position_age = datetime.now() - datetime.fromisoformat(position.get("entry_time",
datetime.now().isoformat()))

    # Calculer le profit actuel
    if side == "BUY":
        profit_percent = (current_price - entry_price) / entry_price * 100 * position.get("leverage", 1)
    else:
        profit_percent = (entry_price - current_price) / entry_price * 100 * position.get("leverage", 1)

    # 1. Vérifier les conditions de marché extrêmes
    if self.market_state == "extreme":
        # Fermer avec profit ou petite perte
        if profit_percent > 0 or profit_percent > -1.5:
            return {
                "should_close": True,
                "reason": f"Conditions de marché extrêmes ({self.market_state})"
            }

    # 2. Utiliser les prédictions LSTM si disponibles
    if lstm_prediction:
```

```
        reversal_alert = lstm_prediction.get("reversal_alert", {})
        reversal_probability = reversal_alert.get("probability", 0.0)

        # Fermeture en cas d'alerte de retournement forte
        if reversal_probability > 0.8 and profit_percent > 0:
            return {
                "should_close": True,
                "reason": f"Alerte de retournement imminente (probabilité: {reversal_probability:.2f})"
            }

        # Vérifier si la prédiction est maintenant contre la position
        for horizon_name, prediction in lstm_prediction.items():
            if horizon_name in ["3h", "4h", "short_term", "horizon_12"]:
                direction_prob = prediction.get("direction_probability", 50) / 100
                direction_contradicts = (side == "BUY" and direction_prob < 0.3) or (side == "SELL" and direction_prob > 0.7)

                if direction_contradicts and profit_percent > 1.0:
                    return {
                        "should_close": True,
                        "reason": f"Prédiction de retournement à court terme (direction: {direction_prob:.2f})"
                    }

    # 3. Lock in profits pour les positions très profitables
    if profit_percent > 10.0:
        return {
            "should_close": True,
            "reason": f"Sécurisation du profit exceptionnel ({profit_percent:.2f}%)"
        }

    # 4. Fermeture des positions en stagnation prolongée
```

```
    stagnation_hours = 24  # Considérer la fermeture après 24h sans progrès
    if position_age.total_seconds() / 3600 > stagnation_hours and -1.0 < profit_percent < 2.0:
        return {
            "should_close": True,
            "reason": f"Position en stagnation après {position_age.total_seconds()/3600:.1f}h"
        }

    # Aucune raison de fermer maintenant
    return {
        "should_close": False,
        "current_profit": profit_percent,
        "position_age_hours": position_age.total_seconds() / 3600
    }

def update_after_trade_closed(self, trade_result: Dict) -> None:
    """
    Met à jour l'état interne après qu'un trade a été fermé

    Args:
        trade_result: Résultat du trade fermé
    """
    # Extraire les résultats du trade
    pnl_percent = trade_result.get("pnl_percent", 0.0)
    pnl_absolute = trade_result.get("pnl_absolute", 0.0)

    # Mettre à jour le capital
    self.current_capital += pnl_absolute

    # Mettre à jour l'historique des trades
    self.trade_history.append({
        "timestamp": datetime.now().isoformat(),
```

```
            "pnl_percent": pnl_percent,
            "pnl_absolute": pnl_absolute,
            "risk_profile": self.current_risk_profile
        })

        # Limiter la taille de l'historique
        if len(self.trade_history) > 100:
            self.trade_history = self.trade_history[-100:]

        # Mettre à jour les compteurs de victoires/défaites consécutives
        if pnl_percent > 0:
            self.consecutive_wins += 1
            self.consecutive_losses = 0
        else:
            self.consecutive_losses += 1
            self.consecutive_wins = 0

        # Ajuster le profil de risque si nécessaire
        self._adjust_risk_profile_based_on_performance()

        # Sauvegarder l'historique
        self._save_history()

    def set_risk_profile(self, profile_name: str) -> bool:
        """
        Change explicitement le profil de risque

        Args:
            profile_name: Nom du profil de risque

        Returns:
```

```
        Succès du changement
    """
    if profile_name in self.risk_profiles:
        self.current_risk_profile = profile_name
        logger.info(f"Profil de risque mis à jour: {profile_name}")
        return True
    else:
        logger.error(f"Profil de risque inconnu: {profile_name}")
        return False

def _calculate_score_factor(self, score: float) -> float:
    """
    Calcule un facteur de taille basé sur le score de l'opportunité

    Args:
        score: Score de l'opportunité (0-100)

    Returns:
        Facteur de taille (0.5-1.2)
    """
    # Convertir le score (0-100) en facteur (0.5-1.2)
    # 70 = 1.0 (neutre)
    # < 70 = réduit
    # > 70 = augmenté

    if score < 70:
        # Réduction linéaire pour les scores < 70
        # 50 -> 0.5, 60 -> 0.75, 70 -> 1.0
        return 0.5 + (score - 50) / 40
    else:
        # Augmentation pour les scores > 70
```

```
        # 70 -> 1.0, 85 -> 1.1, 100 -> 1.2
        return 1.0 + (score - 70) / 150

    def _volatility_adjustment(self, volatility: float) -> float:
        """
        Calcule un facteur d'ajustement basé sur la volatilité

        Args:
            volatility: Volatilité prédite (0-1)

        Returns:
            Facteur d'ajustement (0.5-1.5)
        """
        # Convertir la volatilité en facteur
        # Volatilité standard (0.03) = facteur 1.0
        standard_volatility = 0.03

        if volatility <= standard_volatility:
            # Volatilité faible -> positions plus grandes
            # 0.01 -> 1.5, 0.02 -> 1.25, 0.03 -> 1.0
            return 1.0 + ((standard_volatility - volatility) / standard_volatility) * 0.5
        else:
            # Volatilité élevée -> positions plus petites
            # 0.03 -> 1.0, 0.06 -> 0.5
            return max(0.5, 1.0 - ((volatility - standard_volatility) / standard_volatility) * 0.5)

    def _calculate_kelly_criterion(self) -> float:
        """
        Calcule la portion optimale du capital à risquer selon le critère de Kelly

        Returns:
```

```
    Fraction optimale du capital à risquer (0-1)
"""
# Calculer la win rate sur l'historique récent
if len(self.trade_history) < 10:
    return 0.5  # Valeur par défaut si pas assez de données

# Récupérer les 20 derniers trades (ou moins si pas assez)
recent_trades = self.trade_history[-20:]

# Calculer le win rate
winners = [t for t in recent_trades if t["pnl_percent"] > 0]
win_rate = len(winners) / len(recent_trades)

# Calculer le ratio gain/perte moyen
if winners and len(recent_trades) > len(winners):
    avg_win = sum(t["pnl_percent"] for t in winners) / len(winners)
    losers = [t for t in recent_trades if t["pnl_percent"] <= 0]
    avg_loss = abs(sum(t["pnl_percent"] for t in losers) / len(losers))

    # Formule de Kelly: f* = (p*b - q) / b
    # où p = probabilité de gagner, q = probabilité de perdre, b = ratio gain/perte
    win_loss_ratio = avg_win / avg_loss if avg_loss > 0 else 1

    kelly_fraction = (win_rate * win_loss_ratio - (1 - win_rate)) / win_loss_ratio

    # Limiter la fraction entre 0.1 et 0.5 pour plus de sécurité
    kelly_fraction = max(0.1, min(0.5, kelly_fraction))

    return kelly_fraction

return 0.25  # Valeur conservatrice par défaut
```

```
def _adjust_risk_profile_based_on_performance(self) -> None:
    """
    Ajuste automatiquement le profil de risque en fonction des performances récentes
    """
    # Calculer la performance sur les derniers trades
    if len(self.trade_history) < 5:
        return  # Pas assez de données

    # Récupérer les 10 derniers trades (ou moins si pas assez)
    recent_trades = self.trade_history[-10:]

    # Calculer le PnL cumulé
    cumulative_pnl = sum(t["pnl_percent"] for t in recent_trades)

    # Calculer le win rate
    winners = len([t for t in recent_trades if t["pnl_percent"] > 0])
    win_rate = winners / len(recent_trades) * 100

    # Ajuster le profil en fonction de la performance
    if cumulative_pnl > 15 and win_rate > 60:
        # Performance très bonne -> profil agressif
        if self.current_risk_profile != "very_aggressive":
            self.set_risk_profile("very_aggressive")
            logger.info("Passage en mode très agressif basé sur la performance exceptionnelle")

    elif cumulative_pnl > 8 and win_rate > 55:
        # Bonne performance -> profil légèrement agressif
        if self.current_risk_profile != "aggressive" and self.current_risk_profile != "very_aggressive":
            self.set_risk_profile("aggressive")
            logger.info("Passage en mode agressif basé sur la bonne performance")
```

```
    elif cumulative_pnl < -10 or win_rate < 40:
        # Mauvaise performance -> profil conservateur
        if self.current_risk_profile != "conservative" and self.current_risk_profile != "defensive":
            self.set_risk_profile("conservative")
            logger.info("Passage en mode conservateur basé sur la performance médiocre")

    elif cumulative_pnl < -15 or win_rate < 30:
        # Très mauvaise performance -> profil défensif
        if self.current_risk_profile != "defensive":
            self.set_risk_profile("defensive")
            logger.info("Passage en mode défensif basé sur la performance très faible")

    else:
        # Performance moyenne -> profil équilibré
        if self.current_risk_profile not in ["balanced", "aggressive", "very_aggressive"]:
            self.set_risk_profile("balanced")
            logger.info("Retour au mode équilibré basé sur la performance stable")

def _log_position_sizing(self, symbol: str, base_risk: float,
                         final_size: float, factors: Dict) -> None:
    """
    Enregistre les détails de la décision de sizing pour l'analyse future

    Args:
        symbol: Paire de trading
        base_risk: Risque de base calculé
        final_size: Taille finale de la position
        factors: Facteurs qui ont influencé la décision
    """
    log_entry = {
```

```
            "timestamp": datetime.now().isoformat(),
            "symbol": symbol,
            "capital": self.current_capital,
            "base_risk_amount": base_risk,
            "final_position_size": final_size,
            "current_profile": self.current_risk_profile,
            "risk_capacity": self.risk_capacity,
            "consecutive_wins": self.consecutive_wins,
            "consecutive_losses": self.consecutive_losses,
            "factors": factors
        }

        self.risk_log.append(log_entry)

        # Limiter la taille du journal
        if len(self.risk_log) > 100:
            self.risk_log = self.risk_log[-100:]

        # Sauvegarder périodiquement
        if len(self.risk_log) % 10 == 0:
            self._save_risk_log()

    def _save_history(self) -> None:
        """Sauvegarde l'historique des trades et l'état actuel"""
        history_path = os.path.join(DATA_DIR, "risk_management", "trade_history.json")

        # Créer le répertoire si nécessaire
        os.makedirs(os.path.dirname(history_path), exist_ok=True)

        # Préparer les données à sauvegarder
        state_data = {
```

```
            "current_capital": self.current_capital,
            "risk_profile": self.current_risk_profile,
            "consecutive_wins": self.consecutive_wins,
            "consecutive_losses": self.consecutive_losses,
            "market_state": self.market_state,
            "risk_capacity": self.risk_capacity,
            "daily_risk_used": self.daily_risk_used,
            "last_risk_reset": self.last_risk_reset.isoformat(),
            "trade_history": self.trade_history,
            "updated_at": datetime.now().isoformat()
        }

        try:
            with open(history_path, 'w') as f:
                json.dump(state_data, f, indent=2, default=str)
        except Exception as e:
            logger.error(f"Erreur lors de la sauvegarde de l'historique des trades: {str(e)}")

    def _load_history(self) -> None:
        """Charge l'historique des trades et l'état précédent"""
        history_path = os.path.join(DATA_DIR, "risk_management", "trade_history.json")

        if not os.path.exists(history_path):
            return

        try:
            with open(history_path, 'r') as f:
                state_data = json.load(f)

            # Restaurer l'état
            self.current_capital = state_data.get("current_capital", self.initial_capital)
```

```
            self.current_risk_profile = state_data.get("risk_profile", "balanced")
            self.consecutive_wins = state_data.get("consecutive_wins", 0)
            self.consecutive_losses = state_data.get("consecutive_losses", 0)
            self.market_state = state_data.get("market_state", "normal")
            self.risk_capacity = state_data.get("risk_capacity", 1.0)
            self.daily_risk_used = state_data.get("daily_risk_used", 0.0)

            # Restaurer le timestamp du dernier reset
            try:
                self.last_risk_reset = datetime.fromisoformat(state_data.get("last_risk_reset",
datetime.now().isoformat()))
            except ValueError:
                self.last_risk_reset = datetime.now()

            # Restaurer l'historique des trades
            self.trade_history = state_data.get("trade_history", [])

            logger.info(f"Historique des trades et état chargés: capital={self.current_capital},
profil={self.current_risk_profile}")
        except Exception as e:
            logger.error(f"Erreur lors du chargement de l'historique des trades: {str(e)}")

    def _save_risk_log(self) -> None:
        """Sauvegarde le journal des décisions de risque"""
        log_path = os.path.join(DATA_DIR, "risk_management", "sizing_decisions.json")

        # Créer le répertoire si nécessaire
        os.makedirs(os.path.dirname(log_path), exist_ok=True)

        try:
            with open(log_path, 'w') as f:
                json.dump(self.risk_log, f, indent=2, default=str)
```

```
    except Exception as e:
        logger.error(f"Erreur lors de la sauvegarde du journal des décisions: {str(e)}")

def get_risk_profile_params(self) -> Dict:
    """
    Récupère les paramètres du profil de risque actuel

    Returns:
        Paramètres du profil actuel
    """
    return self.risk_profiles.get(self.current_risk_profile, self.risk_profiles["balanced"])

def get_risk_parameters(self, symbol: str, lstm_prediction: Optional[Dict] = None) -> Dict:
    """
    Récupère tous les paramètres de risque actuels, possiblement ajustés par les prédictions LSTM

    Args:
        symbol: Paire de trading
        lstm_prediction: Prédictions du modèle LSTM (optionnel)

    Returns:
        Paramètres de risque complets
    """
    # Obtenir les paramètres de base du profil actuel
    profile = self.get_risk_profile_params()

    # Paramètres de base
    params = {
        "risk_per_trade_percent": profile["risk_per_trade_percent"],
        "stop_loss_percent": profile["stop_loss_percent"],
        "take_profit_percent": profile["take_profit_percent"],
```

```
    "leverage": profile["leverage"],
    "trailing_stop_activation": profile["trailing_stop_activation"],
    "trailing_stop_step": profile["trailing_stop_step"],
    "risk_profile": self.current_risk_profile,
    "risk_capacity": self.risk_capacity,
    "market_state": self.market_state
}

# Ajuster en fonction des prédictions LSTM si disponibles
if lstm_prediction:
    # Chercher les prédictions de volatilité
    volatility_predicted = False

    for horizon_name, prediction in lstm_prediction.items():
        if horizon_name in ["3h", "4h", "short_term", "horizon_12"]:
            volatility = prediction.get("predicted_volatility", None)

            if volatility is not None:
                volatility_predicted = True

                # Ajuster le stop-loss et take-profit en fonction de la volatilité
                volatility_percent = volatility * 100
                std_volatility = 3.0  # 3% est considéré comme standard

                # Si volatilité élevée, élargir les stops; si faible, les resserrer
                volatility_factor = volatility_percent / std_volatility

                params["stop_loss_percent"] = profile["stop_loss_percent"] * volatility_factor
                params["take_profit_percent"] = profile["take_profit_percent"] * volatility_factor

                # S'assurer que les valeurs restent dans des limites raisonnables
```

```
                params["stop_loss_percent"] = max(2.0, min(10.0, params["stop_loss_percent"]))
                params["take_profit_percent"] = max(3.0, min(15.0, params["take_profit_percent"]))

                # Ajouter l'info de l'ajustement
                params["volatility_adjustment_applied"] = True
                params["predicted_volatility"] = volatility_percent

                break

        # Si pas de prédiction de volatilité, utiliser les paramètres standard
        if not volatility_predicted:
            params["volatility_adjustment_applied"] = False

    return params

def test_risk_manager():
    """Fonction de test simple pour vérifier l'initialisation du gestionnaire de risque"""
    risk_manager = AdaptiveRiskManager(
        initial_capital=200,
        risk_control_mode="balanced"
    )

    # Afficher les profils de risque
    print("Profils de risque configurés:")
    for profile_name, profile in risk_manager.risk_profiles.items():
        print(f"  {profile_name}:")
        for param, value in profile.items():
            print(f"    {param}: {value}")

    print(f"\nProfil actuel: {risk_manager.current_risk_profile}")
    print(f"Capacité de risque: {risk_manager.risk_capacity}")
```

```
    return risk_manager

if __name__ == "__main__":
    risk_manager = test_risk_manager()

================================================
File: crypto_trading_bot_CLAUDE/core/api_connector.py
================================================
# core/api_connector.py
"""
Connecteur pour l'API Binance
Gère les connexions et les appels à l'API de l'échange
"""
import time
import logging
import hmac
import hashlib
import requests
import urllib.parse
from typing import Dict, List, Any, Optional, Union

from config.config import (
    BINANCE_API_KEY,
    BINANCE_API_SECRET,
    USE_TESTNET,
    MAX_API_RETRIES,
    API_RETRY_DELAY
)
from utils.logger import setup_logger
```

```
logger = setup_logger("api_connector")


class BinanceConnector:
    """
    Gère les connexions et les appels à l'API Binance
    """
    def __init__(self):
        self.api_key = BINANCE_API_KEY
        self.api_secret = BINANCE_API_SECRET

        # Configuration des URLs en fonction du mode (testnet ou production)
        if USE_TESTNET:
            self.base_url = "https://testnet.binance.vision/api"
            logger.info("Mode TestNet activé")
        else:
            self.base_url = "https://api.binance.com/api"
            logger.info("Mode Production activé")

    def _get_signature(self, params: Dict) -> str:
        """
        Génère la signature HMAC SHA256 requise pour les requêtes authentifiées

        Args:
            params: Paramètres de la requête

        Returns:
            Signature encodée en hexadécimal
        """
        query_string = urllib.parse.urlencode(params)
        signature = hmac.new(
            self.api_secret.encode('utf-8'),
```

```
        query_string.encode('utf-8'),
        hashlib.sha256
    ).hexdigest()
    return signature

def _make_request(self, method: str, endpoint: str, params: Optional[Dict] = None,
                  signed: bool = False) -> Dict:
    """
    Effectue une requête à l'API Binance avec gestion des erreurs et des tentatives

    Args:
        method: Méthode HTTP (GET, POST, DELETE)
        endpoint: Point de terminaison de l'API
        params: Paramètres de la requête
        signed: Indique si la requête nécessite une signature

    Returns:
        Réponse de l'API sous forme de dictionnaire
    """
    url = f"{self.base_url}{endpoint}"
    headers = {"X-MBX-APIKEY": self.api_key}

    # Paramètres par défaut
    if params is None:
        params = {}

    # Ajout du timestamp et de la signature pour les requêtes authentifiées
    if signed:
        params['timestamp'] = int(time.time() * 1000)
        params['signature'] = self._get_signature(params)
```

```
    # Tentatives avec backoff exponentiel
    for attempt in range(1, MAX_API_RETRIES + 1):
        try:
            if method == "GET":
                response = requests.get(url, params=params, headers=headers)
            elif method == "POST":
                response = requests.post(url, params=params, headers=headers)
            elif method == "DELETE":
                response = requests.delete(url, params=params, headers=headers)
            else:
                raise ValueError(f"Méthode HTTP non supportée: {method}")

            # Vérification du code de statut
            response.raise_for_status()

            return response.json()

        except requests.exceptions.RequestException as e:
            logger.error(f"Tentative {attempt}/{MAX_API_RETRIES} échouée: {str(e)}")

            if attempt < MAX_API_RETRIES:
                # Backoff exponentiel
                wait_time = API_RETRY_DELAY * (2 ** (attempt - 1))
                logger.info(f"Nouvelle tentative dans {wait_time} secondes...")
                time.sleep(wait_time)
            else:
                logger.error("Nombre maximum de tentatives atteint")
                raise

def test_connection(self) -> bool:
    """
```

```
    Teste la connexion à l'API

    Returns:
        True si la connexion est établie, False sinon
    """
    try:
        response = self._make_request("GET", "/v3/ping")
        return True
    except Exception as e:
        logger.error(f"Échec du test de connexion: {str(e)}")
        return False

def get_exchange_info(self) -> Dict:
    """
    Récupère les informations sur l'échange

    Returns:
        Informations sur l'échange
    """
    return self._make_request("GET", "/v3/exchangeInfo")

def get_account_info(self) -> Dict:
    """
    Récupère les informations du compte (nécessite une authentification)

    Returns:
        Informations du compte
    """
    return self._make_request("GET", "/v3/account", signed=True)

def get_order_book(self, symbol: str, limit: int = 100) -> Dict:
```

```
    """
    Récupère le carnet d'ordres pour un symbole donné

    Args:
        symbol: Paire de trading (ex: BTCUSDT)
        limit: Nombre d'ordres à récupérer (max 5000)

    Returns:
        Carnet d'ordres
    """
    params = {
        "symbol": symbol,
        "limit": limit
    }
    return self._make_request("GET", "/v3/depth", params=params)

def get_recent_trades(self, symbol: str, limit: int = 500) -> List[Dict]:
    """
    Récupère les trades récents pour un symbole donné

    Args:
        symbol: Paire de trading
        limit: Nombre de trades à récupérer (max 1000)

    Returns:
        Liste des trades récents
    """
    params = {
        "symbol": symbol,
        "limit": limit
    }
```

```
    return self._make_request("GET", "/v3/trades", params=params)

def get_klines(self, symbol: str, interval: str, limit: int = 500,
               start_time: Optional[int] = None, end_time: Optional[int] = None) -> List:
    """
    Récupère les données de chandelier (klines/OHLCV)

    Args:
        symbol: Paire de trading
        interval: Intervalle de temps (1m, 3m, 5m, 15m, 30m, 1h, 2h, 4h, 6h, 8h, 12h, 1d, 3d, 1w, 1M)
        limit: Nombre de chandeliers à récupérer (max 1000)
        start_time: Timestamp de début (millisecondes)
        end_time: Timestamp de fin (millisecondes)

    Returns:
        Liste des données OHLCV
    """
    params = {
        "symbol": symbol,
        "interval": interval,
        "limit": limit
    }

    if start_time:
        params["startTime"] = start_time
    if end_time:
        params["endTime"] = end_time

    return self._make_request("GET", "/v3/klines", params=params)

def create_order(self, symbol: str, side: str, order_type: str,
```

```
            quantity: Optional[float] = None, price: Optional[float] = None,
            time_in_force: str = "GTC", **kwargs) -> Dict:
    """
    Crée un nouvel ordre

    Args:
        symbol: Paire de trading
        side: Côté (BUY ou SELL)
        order_type: Type d'ordre (LIMIT, MARKET, STOP_LOSS, STOP_LOSS_LIMIT, etc.)
        quantity: Quantité à acheter ou vendre
        price: Prix pour les ordres à cours limité
        time_in_force: Durée de validité de l'ordre (GTC, IOC, FOK)
        **kwargs: Paramètres supplémentaires

    Returns:
        Détails de l'ordre créé
    """
    params = {
        "symbol": symbol,
        "side": side,
        "type": order_type,
    }

    if quantity:
        params["quantity"] = quantity

    if price and order_type != "MARKET":
        params["price"] = price

    if order_type == "LIMIT":
        params["timeInForce"] = time_in_force
```

```
    # Ajout des paramètres supplémentaires
    for key, value in kwargs.items():
        params[key] = value

    return self._make_request("POST", "/v3/order", params=params, signed=True)

def get_order(self, symbol: str, order_id: Optional[int] = None,
              orig_client_order_id: Optional[str] = None) -> Dict:
    """
    Récupère les détails d'un ordre

    Args:
        symbol: Paire de trading
        order_id: ID de l'ordre
        orig_client_order_id: ID client de l'ordre

    Returns:
        Détails de l'ordre
    """
    params = {"symbol": symbol}

    if order_id:
        params["orderId"] = order_id
    elif orig_client_order_id:
        params["origClientOrderId"] = orig_client_order_id
    else:
        raise ValueError("Vous devez spécifier order_id ou orig_client_order_id")

    return self._make_request("GET", "/v3/order", params=params, signed=True)
```

```
def cancel_order(self, symbol: str, order_id: Optional[int] = None,
                 orig_client_order_id: Optional[str] = None) -> Dict:
    """
    Annule un ordre

    Args:
        symbol: Paire de trading
        order_id: ID de l'ordre
        orig_client_order_id: ID client de l'ordre

    Returns:
        Détails de l'ordre annulé
    """
    params = {"symbol": symbol}

    if order_id:
        params["orderId"] = order_id
    elif orig_client_order_id:
        params["origClientOrderId"] = orig_client_order_id
    else:
        raise ValueError("Vous devez spécifier order_id ou orig_client_order_id")

    return self._make_request("DELETE", "/v3/order", params=params, signed=True)

def get_open_orders(self, symbol: Optional[str] = None) -> List[Dict]:
    """
    Récupère tous les ordres ouverts

    Args:
        symbol: Paire de trading (optionnel)
```

```
    Returns:
        Liste des ordres ouverts
    """
    params = {}
    if symbol:
        params["symbol"] = symbol

    return self._make_request("GET", "/v3/openOrders", params=params, signed=True)

def get_leverage_brackets(self, symbol: Optional[str] = None) -> List[Dict]:
    """
    Récupère les paliers d'effet de levier disponibles

    Args:
        symbol: Paire de trading (optionnel)

    Returns:
        Informations sur les paliers d'effet de levier
    """
    params = {}
    if symbol:
        params["symbol"] = symbol

    return self._make_request("GET", "/fapi/v1/leverageBracket", params=params, signed=True)

def set_leverage(self, symbol: str, leverage: int) -> Dict:
    """
    Définit l'effet de levier pour un symbole donné

    Args:
        symbol: Paire de trading
```

```
        leverage: Effet de levier (1-125)

    Returns:
        Résultat de l'opération
    """
    params = {
        "symbol": symbol,
        "leverage": leverage
    }

    # Pour l'API spot avec margin, utiliser l'endpoint correct
    return self._make_request("POST", "/sapi/v1/margin/leverage", params=params, signed=True)


================================================
File: crypto_trading_bot_CLAUDE/core/data_fetcher.py
================================================
# core/data_fetcher.py
"""
Récupération et traitement des données de marché
"""
import pandas as pd
import numpy as np
import time
from typing import Dict, List, Optional, Tuple, Union
from datetime import datetime, timedelta

from config.config import PRIMARY_TIMEFRAME, SECONDARY_TIMEFRAMES
from core.api_connector import BinanceConnector
from utils.logger import setup_logger
```

```
logger = setup_logger("data_fetcher")


class MarketDataFetcher:
    """
    Récupère et traite les données de marché depuis l'API Binance
    """
    def __init__(self, api_connector: BinanceConnector):
        self.api = api_connector
        self.data_cache = {}  # Cache pour les données OHLCV
        self.last_update = {}  # Dernière mise à jour des données
        self.cache_duration = 60  # Durée de validité du cache en secondes

    def get_current_price(self, symbol: str) -> float:
        """
        Récupère le prix actuel d'un symbole

        Args:
            symbol: Paire de trading

        Returns:
            Prix actuel
        """
        try:
            # Utilisation des trades récents pour obtenir le dernier prix
            recent_trades = self.api.get_recent_trades(symbol, limit=1)
            return float(recent_trades[0]['price'])
        except Exception as e:
            logger.error(f"Erreur lors de la récupération du prix actuel pour {symbol}: {str(e)}")
            # Fallback: utiliser le dernier prix des données OHLCV
            ohlcv = self.get_ohlcv(symbol, PRIMARY_TIMEFRAME, limit=1)
            if not ohlcv.empty:
```

```
            return ohlcv['close'].iloc[-1]
        raise

def get_ohlcv(self, symbol: str, timeframe: str, limit: int = 100,
        start_time: Optional[int] = None, end_time: Optional[int] = None) -> pd.DataFrame:
    """
    Récupère les données OHLCV (Open, High, Low, Close, Volume) avec cache adaptatif

    Args:
        symbol: Paire de trading
        timeframe: Intervalle de temps
        limit: Nombre de chandeliers à récupérer
        start_time: Timestamp de début (millisecondes)
        end_time: Timestamp de fin (millisecondes)

    Returns:
        DataFrame pandas avec les données OHLCV
    """

    cache_key = f"{symbol}_{timeframe}_{limit}_{start_time}_{end_time}"
    current_time = time.time()

    if timeframe in ["1m", "5m"]:
        cache_duration = 20 if self._is_market_volatile(symbol) else 60
    elif timeframe in ["15m", "30m"]:
        cache_duration = 60 if self._is_market_volatile(symbol) else 180
    else:  # Timeframes plus longs
        cache_duration = 120 if self._is_market_volatile(symbol) else 300
    # Déterminer si le marché est volatil pour adapter la durée du cache
    is_volatile = self._is_market_volatile(symbol)
    cache_duration = 30 if is_volatile else 90  # 30s si volatil, 90s sinon
```

```
# Vérification du cache
if (cache_key in self.data_cache and cache_key in self.last_update and
    current_time - self.last_update[cache_key] < cache_duration):
    return self.data_cache[cache_key]

try:
    # Récupération des données depuis l'API
    klines = self.api.get_klines(symbol, timeframe, limit, start_time, end_time)

    # Conversion en DataFrame pandas
    df = pd.DataFrame(klines, columns=[
        'timestamp', 'open', 'high', 'low', 'close', 'volume',
        'close_time', 'quote_asset_volume', 'number_of_trades',
        'taker_buy_base_asset_volume', 'taker_buy_quote_asset_volume', 'ignore'
    ])

    # Conversion des types de données
    df['timestamp'] = pd.to_datetime(df['timestamp'], unit='ms')
    df['close_time'] = pd.to_datetime(df['close_time'], unit='ms')

    for col in ['open', 'high', 'low', 'close', 'volume', 'quote_asset_volume',
            'taker_buy_base_asset_volume', 'taker_buy_quote_asset_volume']:
        df[col] = pd.to_numeric(df[col], errors='coerce')

    # Définition de l'index
    df.set_index('timestamp', inplace=True)

    # Mise à jour du cache
    self.data_cache[cache_key] = df
    self.last_update[cache_key] = current_time
```

```
            return df

        except Exception as e:
            logger.error(f"Erreur lors de la récupération des données OHLCV pour {symbol} ({timeframe}): {str(e)}")

            # Si les données sont dans le cache, renvoyer les données du cache même si elles sont périmées
            if cache_key in self.data_cache:
                logger.info(f"Utilisation des données en cache périmées pour {symbol} ({timeframe})")
                return self.data_cache[cache_key]

            # Sinon, retourner un DataFrame vide
            columns = ['open', 'high', 'low', 'close', 'volume']
            return pd.DataFrame(columns=columns)


    def get_market_data(self, symbol: str, indicators: bool = True) -> Dict:
        """
        Récupère toutes les données de marché pertinentes pour un symbole

        Args:
            symbol: Paire de trading
            indicators: Indique si les indicateurs techniques doivent être calculés

        Returns:
            Dictionnaire contenant toutes les données de marché
        """
        data = {
            "symbol": symbol,
            "current_price": self.get_current_price(symbol),
```

```
    "primary_timeframe": {},
    "secondary_timeframes": {}
}

# Données du timeframe principal
primary_data = self.get_ohlcv(symbol, PRIMARY_TIMEFRAME, limit=100)
data["primary_timeframe"]["ohlcv"] = primary_data

# Données des timeframes secondaires
for tf in SECONDARY_TIMEFRAMES:
    secondary_data = self.get_ohlcv(symbol, tf, limit=100)
    data["secondary_timeframes"][tf] = {"ohlcv": secondary_data}

# Calcul des indicateurs techniques si demandé
if indicators:
    from indicators.trend import calculate_ema, calculate_adx
    from indicators.momentum import calculate_rsi
    from indicators.volatility import calculate_bollinger_bands, calculate_atr

    # Indicateurs pour le timeframe principal
    data["primary_timeframe"]["indicators"] = {
        "ema": calculate_ema(primary_data),
        "rsi": calculate_rsi(primary_data),
        "bollinger": calculate_bollinger_bands(primary_data),
        "atr": calculate_atr(primary_data),
        "adx": calculate_adx(primary_data)
    }

    # Indicateurs pour les timeframes secondaires
    for tf, tf_data in data["secondary_timeframes"].items():
        data["secondary_timeframes"][tf]["indicators"] = {
```

```
            "ema": calculate_ema(tf_data["ohlcv"]),
            "rsi": calculate_rsi(tf_data["ohlcv"]),
            "adx": calculate_adx(tf_data["ohlcv"])
        }

    return data

def get_order_book_analysis(self, symbol: str, depth: int = 20) -> Dict:
    """
    Analyse le carnet d'ordres pour déterminer la pression d'achat/vente

    Args:
        symbol: Paire de trading
        depth: Profondeur du carnet à analyser

    Returns:
        Analyse du carnet d'ordres
    """
    try:
        order_book = self.api.get_order_book(symbol, limit=depth)

        # Extraction des offres (asks) et des demandes (bids)
        bids = np.array([[float(price), float(qty)] for price, qty in order_book["bids"]])
        asks = np.array([[float(price), float(qty)] for price, qty in order_book["asks"]])

        # Calcul de la pression d'achat/vente
        bid_volume = np.sum(bids[:, 1])
        ask_volume = np.sum(asks[:, 1])

        # Calcul des murs d'achat/vente (concentrations importantes d'ordres)
        bid_walls = []
```

```
    ask_walls = []

    # Seuil pour considérer un niveau comme un mur (% du volume total)
    wall_threshold = 0.15

    for price, qty in bids:
        if qty / bid_volume > wall_threshold:
            bid_walls.append({"price": price, "quantity": qty, "percentage": qty / bid_volume * 100})

    for price, qty in asks:
        if qty / ask_volume > wall_threshold:
            ask_walls.append({"price": price, "quantity": qty, "percentage": qty / ask_volume * 100})

    # Calcul du déséquilibre achat/vente
    if ask_volume > 0:
        buy_sell_ratio = bid_volume / ask_volume
    else:
        buy_sell_ratio = float('inf')

    return {
        "bid_volume": bid_volume,
        "ask_volume": ask_volume,
        "buy_sell_ratio": buy_sell_ratio,
        "bid_walls": bid_walls,
        "ask_walls": ask_walls,
        "buy_pressure": buy_sell_ratio > 1.2,  # Forte pression d'achat
        "sell_pressure": buy_sell_ratio < 0.8,  # Forte pression de vente
        "timestamp": datetime.now().timestamp()
    }

except Exception as e:
```

```
        logger.error(f"Erreur lors de l'analyse du carnet d'ordres pour {symbol}: {str(e)}")
        return {
            "error": str(e),
            "timestamp": datetime.now().timestamp()
        }

    def get_volume_profile(self, symbol: str, timeframe: str, periods: int = 24) -> Dict:
        """
        Calcule le profil de volume pour identifier les niveaux de prix significatifs

        Args:
            symbol: Paire de trading
            timeframe: Intervalle de temps
            periods: Nombre de périodes à analyser

        Returns:
            Profil de volume
        """
        try:
            # Récupération des données OHLCV
            ohlcv = self.get_ohlcv(symbol, timeframe, limit=periods)

            if ohlcv.empty:
                return {"error": "Données OHLCV vides", "timestamp": datetime.now().timestamp()}

            # Trouver le prix min et max sur la période
            price_min = ohlcv['low'].min()
            price_max = ohlcv['high'].max()

            # Création de tranches de prix (10 tranches)
            price_range = price_max - price_min
```

```
slice_height = price_range / 10

volume_profile = []

for i in range(10):
    price_level_min = price_min + i * slice_height
    price_level_max = price_min + (i + 1) * slice_height

    # Sélection des chandeliers qui traversent cette tranche de prix
    mask = (ohlcv['high'] >= price_level_min) & (ohlcv['low'] <= price_level_max)
    volume_in_range = ohlcv.loc[mask, 'volume'].sum()

    volume_profile.append({
        "price_level_min": price_level_min,
        "price_level_max": price_level_max,
        "volume": volume_in_range
    })

# Tri par volume décroissant
volume_profile.sort(key=lambda x: x["volume"], reverse=True)

# Identification des niveaux de prix à fort volume (Value Area)
total_volume = ohlcv['volume'].sum()
cumulative_volume = 0
value_area = []

for level in volume_profile:
    cumulative_volume += level["volume"]
    level["percentage"] = level["volume"] / total_volume * 100

    if cumulative_volume <= total_volume * 0.7:  # 70% du volume total
```

```
                value_area.append({
                    "price_min": level["price_level_min"],
                    "price_max": level["price_level_max"],
                    "volume": level["volume"],
                    "percentage": level["percentage"]
                })

            return {
                "volume_profile": volume_profile,
                "value_area": value_area,
                "point_of_control": volume_profile[0],  # Niveau de prix avec le plus de volume
                "timestamp": datetime.now().timestamp()
            }

        except Exception as e:
            logger.error(f"Erreur lors du calcul du profil de volume pour {symbol}: {str(e)}")
            return {
                "error": str(e),
                "timestamp": datetime.now().timestamp()
            }
    def _is_market_volatile(self, symbol: str) -> bool:
        """
        Détermine si le marché est actuellement volatil

        Args:
            symbol: Paire de trading

        Returns:
            True si le marché est volatil, False sinon
        """
        try:
```

```
            # Récupérer les dernières données
            ohlcv = self.get_ohlcv(symbol, "1m", limit=10)

            if ohlcv.empty:
                return False

            # Calculer la volatilité (ATR sur les 10 dernières minutes)
            high_low = ohlcv['high'] - ohlcv['low']
            high_close = abs(ohlcv['high'] - ohlcv['close'].shift())
            low_close = abs(ohlcv['low'] - ohlcv['close'].shift())

            tr = pd.concat([high_low, high_close, low_close], axis=1).max(axis=1)
            atr = tr.mean()

            # Calculer la volatilité relative au prix
            current_price = ohlcv['close'].iloc[-1]
            volatility_percent = (atr / current_price) * 100

            # Considérer comme volatil si la volatilité est supérieure à 0.5% sur 10 minutes
            return volatility_percent > 0.5

        except Exception as e:
            logger.error(f"Erreur lors de l'évaluation de la volatilité pour {symbol}: {str(e)}")
            return False  # Par défaut, considérer comme non volatil

================================================
File: crypto_trading_bot_CLAUDE/core/order_manager.py
================================================
"""
Gestionnaire d'ordres pour le bot de trading
Gère la création, la modification et l'annulation des ordres
```

```
"""
import time
import uuid
import logging
from typing import Dict, List, Optional, Union
from datetime import datetime

from core.api_connector import BinanceConnector
from core.position_tracker import PositionTracker
from config.trading_params import (
    STOP_LOSS_PERCENT,
    TAKE_PROFIT_PERCENT,
    TRAILING_STOP_ACTIVATION,
    TRAILING_STOP_STEP,
    LEVERAGE
)
from utils.logger import setup_logger

logger = setup_logger("order_manager")

class OrderManager:
    """
    Gère les ordres de trading (entrée, sortie, stop-loss, take-profit)
    """
    def __init__(self, api_connector: BinanceConnector, position_tracker: PositionTracker):
        self.api = api_connector
        self.position_tracker = position_tracker
        self.leverage_set = set()  # Paires pour lesquelles le levier a déjà été défini

    def set_leverage_if_needed(self, symbol: str) -> bool:
        """
```

```
        Définit l'effet de levier pour un symbole si ce n'est pas déjà fait

        Args:
            symbol: Paire de trading

        Returns:
            True si l'opération a réussi, False sinon
        """
        if symbol in self.leverage_set:
            return True

        try:
            # Définir l'effet de levier
            response = self.api.set_leverage(symbol, LEVERAGE)

            # Vérifier si l'opération a réussi
            if "leverage" in response:
                self.leverage_set.add(symbol)
                logger.info(f"Levier défini à {LEVERAGE}x pour {symbol}")
                return True
            else:
                logger.error(f"Échec de la définition du levier pour {symbol}: {response}")
                return False

        except Exception as e:
            logger.error(f"Erreur lors de la définition du levier pour {symbol}: {str(e)}")
            return False

    def place_entry_order(self, symbol: str, side: str, quantity: float, price: Optional[float] = None,
                          stop_loss_price: Optional[float] = None, take_profit_price: Optional[float] = None) ->
Dict:
```

```
"""
Place un ordre d'entrée avec stop-loss et take-profit

Args:
    symbol: Paire de trading
    side: Direction (BUY/SELL)
    quantity: Quantité à acheter/vendre
    price: Prix d'entrée (None pour un ordre au marché)
    stop_loss_price: Prix du stop-loss (calculé automatiquement si None)
    take_profit_price: Prix du take-profit (calculé automatiquement si None)

Returns:
    Résultat de l'opération
"""
# Vérifier et définir l'effet de levier si nécessaire
if not self.set_leverage_if_needed(symbol):
    return {"success": False, "message": "Échec de la définition du levier"}

# Générer un identifiant unique pour l'ordre
client_order_id = f"bot_{int(time.time()*1000)}_{uuid.uuid4().hex[:8]}"

try:
    # Déterminer le type d'ordre (MARKET ou LIMIT)
    order_type = "MARKET" if price is None else "LIMIT"

    # Paramètres de l'ordre
    order_params = {
        "newClientOrderId": client_order_id
    }

    # Placer l'ordre d'entrée
```

```
if order_type == "MARKET":
    entry_order = self.api.create_order(
        symbol=symbol,
        side=side,
        order_type=order_type,
        quantity=quantity,
        **order_params
    )
else:
    entry_order = self.api.create_order(
        symbol=symbol,
        side=side,
        order_type=order_type,
        quantity=quantity,
        price=price,
        time_in_force="GTC",
        **order_params
    )

logger.info(f"Ordre d'entrée placé pour {symbol}: {entry_order}")

# Traiter la réponse de l'ordre
if "orderId" in entry_order:
    # Calculer les prix de stop-loss et take-profit si non fournis
    entry_price = float(entry_order.get("price") or entry_order.get("fills", [{}])[0].get("price", 0))

    if stop_loss_price is None:
        if side == "BUY":
            stop_loss_price = entry_price * (1 - STOP_LOSS_PERCENT/100)
        else:
            stop_loss_price = entry_price * (1 + STOP_LOSS_PERCENT/100)
```

```
if take_profit_price is None:
    if side == "BUY":
        take_profit_price = entry_price * (1 + TAKE_PROFIT_PERCENT/100)
    else:
        take_profit_price = entry_price * (1 - TAKE_PROFIT_PERCENT/100)

# Placer les ordres de stop-loss et take-profit
stop_loss_side = "SELL" if side == "BUY" else "BUY"

# Ordre stop-loss
stop_loss_order = self.api.create_order(
    symbol=symbol,
    side=stop_loss_side,
    order_type="STOP_LOSS_LIMIT" if price else "STOP_MARKET",
    quantity=quantity,
    price=stop_loss_price if price else None,
    stop_price=stop_loss_price,
    time_in_force="GTC",
    newClientOrderId=f"sl_{client_order_id}"
)

logger.info(f"Ordre stop-loss placé pour {symbol}: {stop_loss_order}")

# Ordre take-profit
take_profit_order = self.api.create_order(
    symbol=symbol,
    side=stop_loss_side,
    order_type="LIMIT",
    quantity=quantity,
    price=take_profit_price,
```

```
    time_in_force="GTC",
    newClientOrderId=f"tp_{client_order_id}"
)

logger.info(f"Ordre take-profit placé pour {symbol}: {take_profit_order}")

# Enregistrer la position dans le tracker
position = {
    "id": client_order_id,
    "symbol": symbol,
    "side": side,
    "entry_price": entry_price,
    "quantity": quantity,
    "stop_loss_price": stop_loss_price,
    "take_profit_price": take_profit_price,
    "entry_order_id": entry_order["orderId"],
    "stop_loss_order_id": stop_loss_order.get("orderId"),
    "take_profit_order_id": take_profit_order.get("orderId"),
    "entry_time": datetime.now(),
    "trailing_stop_activated": False,
    "highest_price": entry_price if side == "BUY" else float('inf'),
    "lowest_price": entry_price if side == "SELL" else float('-inf')
}

self.position_tracker.add_position(position)

return {
    "success": True,
    "position_id": client_order_id,
    "entry_price": entry_price,
    "stop_loss_price": stop_loss_price,
```

```
                "take_profit_price": take_profit_price
            }
        else:
            logger.error(f"Échec de l'ordre d'entrée pour {symbol}: {entry_order}")
            return {"success": False, "message": "Échec de l'ordre d'entrée", "response": entry_order}

    except Exception as e:
        logger.error(f"Erreur lors du placement de l'ordre pour {symbol}: {str(e)}")
        return {"success": False, "message": str(e)}

def update_trailing_stop(self, symbol: str, position: Dict, current_price: float) -> Dict:
    """
    Met à jour le trailing stop d'une position si nécessaire

    Args:
        symbol: Paire de trading
        position: Position à mettre à jour
        current_price: Prix actuel

    Returns:
        Résultat de l'opération
    """
    side = position["side"]
    entry_price = position["entry_price"]
    stop_loss_price = position["stop_loss_price"]
    position_id = position["id"]

    # Vérifier si le trailing stop doit être activé ou mis à jour
    if side == "BUY":
        # Pour les positions longues
        profit_percent = (current_price - entry_price) / entry_price * 100
```

```
# Mettre à jour le prix le plus haut observé
if current_price > position["highest_price"]:
    position["highest_price"] = current_price
    self.position_tracker.update_position(position_id, position)

# Activation du trailing stop
if (not position["trailing_stop_activated"] and
    profit_percent >= TRAILING_STOP_ACTIVATION):

    position["trailing_stop_activated"] = True
    self.position_tracker.update_position(position_id, position)
    logger.info(f"Trailing stop activé pour {symbol} (ID: {position_id})")

# Mise à jour du trailing stop
if position["trailing_stop_activated"]:
    new_stop_loss = position["highest_price"] * (1 - TRAILING_STOP_STEP/100)

    if new_stop_loss > stop_loss_price:
        try:
            # Annuler l'ancien ordre stop-loss
            self.api.cancel_order(
                symbol=symbol,
                order_id=position["stop_loss_order_id"]
            )

            # Créer un nouvel ordre stop-loss
            new_stop_order = self.api.create_order(
                symbol=symbol,
                side="SELL",
                order_type="STOP_MARKET",
```

```
                    quantity=position["quantity"],
                    stop_price=new_stop_loss,
                    newClientOrderId=f"sl_trail_{position_id}"
                )

                # Mettre à jour la position
                position["stop_loss_price"] = new_stop_loss
                position["stop_loss_order_id"] = new_stop_order["orderId"]
                self.position_tracker.update_position(position_id, position)

                logger.info(f"Trailing stop mis à jour pour {symbol} (ID: {position_id}) à {new_stop_loss}")
                return {"success": True, "new_stop_loss": new_stop_loss}

            except Exception as e:
                logger.error(f"Erreur lors de la mise à jour du trailing stop pour {symbol} (ID: {position_id}): {str(e)}")
                return {"success": False, "message": str(e)}

    elif side == "SELL":
        # Pour les positions courtes
        profit_percent = (entry_price - current_price) / entry_price * 100

        # Mettre à jour le prix le plus bas observé
        if current_price < position["lowest_price"]:
            position["lowest_price"] = current_price
            self.position_tracker.update_position(position_id, position)

        # Activation du trailing stop
        if (not position["trailing_stop_activated"] and
            profit_percent >= TRAILING_STOP_ACTIVATION):
```

```
    position["trailing_stop_activated"] = True
    self.position_tracker.update_position(position_id, position)
    logger.info(f"Trailing stop activé pour {symbol} (ID: {position_id})")

# Mise à jour du trailing stop
if position["trailing_stop_activated"]:
    new_stop_loss = position["lowest_price"] * (1 + TRAILING_STOP_STEP/100)

    if new_stop_loss < stop_loss_price:
        try:
            # Annuler l'ancien ordre stop-loss
            self.api.cancel_order(
                symbol=symbol,
                order_id=position["stop_loss_order_id"]
            )

            # Créer un nouvel ordre stop-loss
            new_stop_order = self.api.create_order(
                symbol=symbol,
                side="BUY",
                order_type="STOP_MARKET",
                quantity=position["quantity"],
                stop_price=new_stop_loss,
                newClientOrderId=f"sl_trail_{position_id}"
            )

            # Mettre à jour la position
            position["stop_loss_price"] = new_stop_loss
            position["stop_loss_order_id"] = new_stop_order["orderId"]
            self.position_tracker.update_position(position_id, position)
```

```
                logger.info(f"Trailing stop mis à jour pour {symbol} (ID: {position_id}) à {new_stop_loss}")
                return {"success": True, "new_stop_loss": new_stop_loss}

            except Exception as e:
                logger.error(f"Erreur lors de la mise à jour du trailing stop pour {symbol} (ID: {position_id}): {str(e)}")
                return {"success": False, "message": str(e)}

        return {"success": True, "message": "Aucune mise à jour nécessaire"}

    def close_position(self, symbol: str, position_id: str) -> Dict:
        """
        Ferme une position manuellement

        Args:
            symbol: Paire de trading
            position_id: ID de la position à fermer

        Returns:
            Résultat de l'opération
        """
        position = self.position_tracker.get_position(position_id)

        if not position:
            return {"success": False, "message": f"Position {position_id} non trouvée"}

        try:
            # Annuler les ordres de stop-loss et take-profit
            orders_to_cancel = [
                position.get("stop_loss_order_id"),
                position.get("take_profit_order_id")
```

```
            ]

            for order_id in orders_to_cancel:
                if order_id:
                    try:
                        self.api.cancel_order(symbol=symbol, order_id=order_id)
                    except Exception as e:
                        logger.warning(f"Erreur lors de l'annulation de l'ordre {order_id}: {str(e)}")

            # Créer un ordre de fermeture au marché
            close_side = "SELL" if position["side"] == "BUY" else "BUY"

            close_order = self.api.create_order(
                symbol=symbol,
                side=close_side,
                order_type="MARKET",
                quantity=position["quantity"],
                newClientOrderId=f"close_{position_id}"
            )

            logger.info(f"Position {position_id} fermée: {close_order}")

            # Marquer la position comme fermée dans le tracker
            self.position_tracker.close_position(position_id, close_order)

            return {"success": True, "close_order": close_order}

        except Exception as e:
            logger.error(f"Erreur lors de la fermeture de la position {position_id}: {str(e)}")
            return {"success": False, "message": str(e)}
```

```
================================================
File: crypto_trading_bot_CLAUDE/core/position_tracker.py
================================================
# core/position_tracker.py
"""
Suivi des positions ouvertes et fermées
"""
import json
import os
from typing import Dict, List, Optional, Union
from datetime import datetime

from config.config import DATA_DIR
from utils.logger import setup_logger

logger = setup_logger("position_tracker")

class PositionTracker:
    """
    Gère le suivi des positions ouvertes et fermées
    """
    def __init__(self):
        self.open_positions = {}  # {position_id: position_data}
        self.closed_positions = []  # Liste des positions fermées

        self.positions_file = os.path.join(DATA_DIR, "positions.json")
        self.load_positions()

    def load_positions(self) -> None:
        """
        Charge les positions depuis le fichier de sauvegarde
```

```
        """
        if os.path.exists(self.positions_file):
            try:
                with open(self.positions_file, 'r') as f:
                    data = json.load(f)
                    self.open_positions = data.get("open_positions", {})
                    self.closed_positions = data.get("closed_positions", [])
                logger.info(f"Positions chargées: {len(self.open_positions)} ouvertes, {len(self.closed_positions)} fermées")
            except Exception as e:
                logger.error(f"Erreur lors du chargement des positions: {str(e)}")

    def save_positions(self) -> None:
        """
        Sauvegarde les positions dans un fichier
        """
        try:
            data = {
                "open_positions": self.open_positions,
                "closed_positions": self.closed_positions
            }

            with open(self.positions_file, 'w') as f:
                json.dump(data, f, indent=2, default=str)

            logger.debug("Positions sauvegardées")
        except Exception as e:
            logger.error(f"Erreur lors de la sauvegarde des positions: {str(e)}")

    def add_position(self, position: Dict) -> None:
        """
```

```
    Ajoute une nouvelle position

    Args:
        position: Données de la position
    """
    position_id = position["id"]
    self.open_positions[position_id] = position
    logger.info(f"Position ajoutée: {position_id} ({position['symbol']})")
    self.save_positions()

def update_position(self, position_id: str, position_data: Dict) -> bool:
    """
    Met à jour une position existante

    Args:
        position_id: ID de la position
        position_data: Nouvelles données de la position

    Returns:
        True si la mise à jour a réussi, False sinon
    """
    if position_id in self.open_positions:
        self.open_positions[position_id] = position_data
        logger.debug(f"Position mise à jour: {position_id}")
        self.save_positions()
        return True
    else:
        logger.warning(f"Tentative de mise à jour d'une position inexistante: {position_id}")
        return False

def close_position(self, position_id: str, close_data: Dict) -> bool:
```

```
"""
Marque une position comme fermée

Args:
    position_id: ID de la position
    close_data: Données de fermeture

Returns:
    True si la fermeture a réussi, False sinon
"""
if position_id in self.open_positions:
    position = self.open_positions.pop(position_id)

    # Ajouter les informations de fermeture
    position["close_time"] = datetime.now()
    position["close_data"] = close_data

    # Calculer le P&L
    if "fills" in close_data:
        close_price = float(close_data["fills"][0]["price"])
        entry_price = float(position["entry_price"])
        quantity = float(position["quantity"])

        if position["side"] == "BUY":
            pnl_percent = (close_price - entry_price) / entry_price * 100
            pnl_absolute = (close_price - entry_price) * quantity
        else:
            pnl_percent = (entry_price - close_price) / entry_price * 100
            pnl_absolute = (entry_price - close_price) * quantity

        position["pnl_percent"] = pnl_percent
```

```
            position["pnl_absolute"] = pnl_absolute

            logger.info(f"Position {position_id} fermée: {pnl_percent:.2f}% ({pnl_absolute:.2f} USDT)")
        else:
            logger.info(f"Position {position_id} fermée (P&L non calculable)")

        # Ajouter à la liste des positions fermées
        self.closed_positions.append(position)
        self.save_positions()
        return True
    else:
        logger.warning(f"Tentative de fermeture d'une position inexistante: {position_id}")
        return False

def get_position(self, position_id: str) -> Optional[Dict]:
    """
    Récupère les données d'une position

    Args:
        position_id: ID de la position

    Returns:
        Données de la position, ou None si non trouvée
    """
    return self.open_positions.get(position_id)

def get_open_positions(self, symbol: Optional[str] = None) -> List[Dict]:
    """
    Récupère les positions ouvertes pour un symbole donné

    Args:
```

```
        symbol: Paire de trading (optionnel)

    Returns:
        Liste des positions ouvertes
    """
    if symbol:
        return [p for p in self.open_positions.values() if p["symbol"] == symbol]
    else:
        return list(self.open_positions.values())

def get_all_open_positions(self) -> Dict[str, List[Dict]]:
    """
    Récupère toutes les positions ouvertes, groupées par symbole

    Returns:
        Dictionnaire {symbole: [positions]}
    """
    positions_by_symbol = {}

    for position in self.open_positions.values():
        symbol = position["symbol"]
        if symbol not in positions_by_symbol:
            positions_by_symbol[symbol] = []
        positions_by_symbol[symbol].append(position)

    return positions_by_symbol

def get_closed_positions(self, symbol: Optional[str] = None, limit: int = 100) -> List[Dict]:
    """
    Récupère les positions fermées
```

```
    Args:
        symbol: Paire de trading (optionnel)
        limit: Nombre maximum de positions à récupérer

    Returns:
        Liste des positions fermées
    """
    if symbol:
        filtered = [p for p in self.closed_positions if p["symbol"] == symbol]
    else:
        filtered = self.closed_positions

    # Tri par date de fermeture (du plus récent au plus ancien)
    sorted_positions = sorted(
        filtered,
        key=lambda p: p.get("close_time", datetime.min),
        reverse=True
    )

    return sorted_positions[:limit]

def get_position_count(self, symbol: Optional[str] = None) -> int:
    """
    Compte le nombre de positions ouvertes

    Args:
        symbol: Paire de trading (optionnel)

    Returns:
        Nombre de positions ouvertes
    """
```

```
    if symbol:
        return len([p for p in self.open_positions.values() if p["symbol"] == symbol])
    else:
        return len(self.open_positions)

def get_daily_trades_count(self, symbol: Optional[str] = None) -> int:
    """
    Compte le nombre de trades effectués aujourd'hui

    Args:
        symbol: Paire de trading (optionnel)

    Returns:
        Nombre de trades aujourd'hui
    """
    today = datetime.now().date()

    # Compter les positions fermées aujourd'hui
    closed_today = [
        p for p in self.closed_positions
        if p.get("close_time") and p.get("close_time").date() == today
        and (symbol is None or p["symbol"] == symbol)
    ]

    # Compter les positions ouvertes aujourd'hui
    opened_today = [
        p for p in self.open_positions.values()
        if p.get("entry_time") and p.get("entry_time").date() == today
        and (symbol is None or p["symbol"] == symbol)
    ]
```

```
        return len(closed_today) + len(opened_today)


================================================
File: crypto_trading_bot_CLAUDE/core/risk_manager.py
================================================
# core/risk_manager.py
"""
Gestionnaire de risques pour le bot de trading
"""
from typing import Dict, Optional
import logging
from datetime import datetime, timedelta

from config.config import INITIAL_CAPITAL
from config.trading_params import (
    RISK_PER_TRADE_PERCENT,
    MAX_CONCURRENT_TRADES,
    MAX_DAILY_TRADES,
    LEVERAGE,
    MAX_DRAWDOWN_LIMIT
)
from utils.logger import setup_logger

logger = setup_logger("risk_manager")

class RiskManager:
    """
    Gère les risques et le capital du bot de trading
    """
    def __init__(self):
        self.initial_capital = INITIAL_CAPITAL
```

```
        self.available_balance = INITIAL_CAPITAL
        self.equity = INITIAL_CAPITAL
        self.daily_losses = 0
        self.daily_profits = 0
        self.positions_count = 0
        self.daily_trade_count = 0
        self.last_reset_date = datetime.now().date()
        self.consecutive_losses = 0
        self.win_streak = 0
        self.loss_streak = 0
        self.max_loss_streak = 0
        self.max_win_streak = 0
        self.current_drawdown = 0
        self.peak_equity = INITIAL_CAPITAL

        # NOUVEAU: Historique des performances
        self.performance_history = []
        self.volatility_history = []

    def update_account_balance(self, account_info: Dict) -> None:
        """
        Met à jour les informations de solde du compte

        Args:
            account_info: Informations du compte depuis l'API
        """
        # Extraire le solde USDT
        for asset in account_info.get("balances", []):
            if asset["asset"] == "USDT":
                self.available_balance = float(asset["free"])
                self.equity = float(asset["free"]) + float(asset["locked"])
```

```
            logger.info(f"Solde mis à jour: {self.available_balance} USDT disponible, {self.equity} USDT total")
            break

    # Réinitialiser les compteurs journaliers si nécessaire
    current_date = datetime.now().date()
    if current_date > self.last_reset_date:
        self.reset_daily_stats()
        self.last_reset_date = current_date

def reset_daily_stats(self) -> None:
    """
    Réinitialise les statistiques journalières
    """
    self.daily_losses = 0
    self.daily_profits = 0
    self.daily_trade_count = 0
    logger.info("Statistiques journalières réinitialisées")

def update_position_stats(self, position_tracker) -> None:
    """
    Met à jour les statistiques de positions

    Args:
        position_tracker: Tracker de positions
    """
    self.positions_count = len(position_tracker.get_open_positions())
    self.daily_trade_count = position_tracker.get_daily_trades_count()

    # Calculer les profits/pertes journaliers
    today = datetime.now().date()
```

```
        daily_closed_positions = [
            p for p in position_tracker.get_closed_positions(limit=1000)
            if p.get("close_time") and p.get("close_time").date() == today
        ]

        self.daily_profits = sum([
            p.get("pnl_absolute", 0) for p in daily_closed_positions
            if p.get("pnl_absolute", 0) > 0
        ])

        self.daily_losses = sum([
            p.get("pnl_absolute", 0) for p in daily_closed_positions
            if p.get("pnl_absolute", 0) < 0
        ])

        logger.debug(f"Stats mises à jour: {self.positions_count} positions, {self.daily_trade_count} trades aujourd'hui")

    def can_open_new_position(self, position_tracker) -> bool:
        """
        Vérifie si une nouvelle position peut être ouverte

        Args:
            position_tracker: Tracker de positions

        Returns:
            True si une nouvelle position peut être ouverte, False sinon
        """
        # Mettre à jour les statistiques
        self.update_position_stats(position_tracker)
```

```
        # Vérifier le nombre maximum de positions simultanées
        if self.positions_count >= MAX_CONCURRENT_TRADES:
            logger.info(f"Nombre maximum de positions simultanées atteint
({MAX_CONCURRENT_TRADES})")
            return False

        # Vérifier le nombre maximum de trades par jour
        if self.daily_trade_count >= MAX_DAILY_TRADES:
            logger.info(f"Nombre maximum de trades journaliers atteint ({MAX_DAILY_TRADES})")
            return False

        # Vérifier si le solde disponible est suffisant
        if self.available_balance <= 0:
            logger.info("Solde insuffisant pour ouvrir une nouvelle position")
            return False

        return True

    def calculate_position_size(self, symbol: str, opportunity: Dict) -> float:
        """
        Calcule la taille de position optimale en fonction du risque - Version adaptative

        Args:
            symbol: Paire de trading
            opportunity: Opportunité de trading avec entrée et stop-loss

        Returns:
            Quantité à trader
        """
        entry_price = opportunity.get("entry_price", 0)
```

```
stop_loss_price = opportunity.get("stop_loss", 0)
score = opportunity.get("score", 50)

if entry_price <= 0 or stop_loss_price <= 0:
    logger.error("Prix d'entrée ou de stop-loss invalide")
    return 0

# Calculer le risque en pourcentage
if opportunity.get("side") == "BUY":
    risk_percent = (entry_price - stop_loss_price) / entry_price * 100
else:
    risk_percent = (stop_loss_price - entry_price) / entry_price * 100

if risk_percent <= 0:
    logger.error(f"Risque en pourcentage invalide: {risk_percent}%")
    return 0

# NOUVEAU: Risque adaptatif basé sur:
# 1. Score de l'opportunité
# 2. Historique récent de pertes/gains
# 3. Drawdown actuel
base_risk = RISK_PER_TRADE_PERCENT

# Ajuster le risque en fonction du score
score_factor = self._calculate_score_factor(score)

# Ajuster le risque en fonction des pertes consécutives
streak_factor = self._calculate_streak_factor()

# Ajuster le risque en fonction du drawdown actuel
drawdown_factor = self._calculate_drawdown_factor()
```

```
        # Calculer le facteur de risque combiné
        risk_factor = score_factor * streak_factor * drawdown_factor

        # Limite pour éviter des risques trop extrêmes
        risk_factor = max(0.3, min(1.5, risk_factor))

        # Risque final ajusté
        adjusted_risk = base_risk * risk_factor
        logger.info(f"Risque ajusté: {adjusted_risk:.2f}% (base: {base_risk}%, facteur: {risk_factor:.2f})")

        # Calculer le montant à risquer
        risk_amount = self.equity * (adjusted_risk / 100)

        # Calculer la taille de position en fonction du risque
        position_size = risk_amount / (risk_percent / 100 * entry_price)

        # Prendre en compte l'effet de levier
        position_size = position_size * LEVERAGE

        # Limiter la taille de position au solde disponible
        max_position_size = self.available_balance * LEVERAGE / entry_price
        position_size = min(position_size, max_position_size)

    # NOUVEAU: Limiter la taille maximale de position à 25% du capital total, quelle que soit la situation
        max_allowed_size = self.equity * 0.25 * LEVERAGE / entry_price
        position_size = min(position_size, max_allowed_size)

        # Arrondir la taille de position à la précision requise
        position_size = round(position_size, 5)
```

```
        logger.info(f"Taille de position calculée pour {symbol}: {position_size} (risque: {risk_amount:.2f} USDT)")

        return position_size


    def update_after_trade_closed(self, trade_result: Dict) -> None:
        """
        Met à jour les statistiques après la fermeture d'un trade

        Args:
            trade_result: Résultat du trade
        """
        pnl = trade_result.get("pnl_absolute", 0)

        if pnl > 0:
            self.daily_profits += pnl
            logger.info(f"Profit ajouté: {pnl} USDT (total journalier: {self.daily_profits} USDT)")
        else:
            self.daily_losses += pnl
            logger.info(f"Perte ajoutée: {pnl} USDT (total journalier: {self.daily_losses} USDT)")


    def get_risk_metrics(self) -> Dict:
        """
        Récupère les métriques de risque actuelles

        Returns:
            Métriques de risque
        """
        return {
            "initial_capital": self.initial_capital,
            "current_equity": self.equity,
```

```
        "available_balance": self.available_balance,
        "positions_count": self.positions_count,
        "daily_trade_count": self.daily_trade_count,
        "daily_profits": self.daily_profits,
        "daily_losses": self.daily_losses,
        "net_daily_pnl": self.daily_profits + self.daily_losses,
        "daily_roi_percent": (self.daily_profits + self.daily_losses) / self.initial_capital * 100
    }

def calculate_adaptive_risk(self, symbol: str, opportunity: Dict, market_volatility: float) -> float:
    """
    Calcule le risque par trade de manière adaptative en fonction de la volatilité du marché

    Args:
        symbol: Paire de trading
        opportunity: Opportunité de trading
        market_volatility: Niveau de volatilité du marché (0-1)

    Returns:
        Pourcentage du capital à risquer
    """
    base_risk = RISK_PER_TRADE_PERCENT

    # Réduire le risque quand la volatilité est élevée
    if market_volatility > 0.7:  # Volatilité élevée
        return base_risk * 0.7
    elif market_volatility > 0.4:  # Volatilité moyenne
        return base_risk * 0.85
    else:  # Volatilité faible
        return base_risk * 1.1  # Plus agressif quand le marché est calme
```

```
def _calculate_score_factor(self, score: int) -> float:
    """
    Calcule un facteur de risque basé sur le score de l'opportunité
    """
    # Plus le score est élevé, plus nous sommes confiants, donc nous prenons plus de risque
    if score >= 90:
        return 1.3  # Très confiant - augmenter le risque de 30%
    elif score >= 80:
        return 1.2
    elif score >= 70:
        return 1.1
    elif score >= 60:
        return 1.0  # Score normal - risque standard
    elif score >= 50:
        return 0.8
    else:
        return 0.6  # Score faible - réduire le risque de 40%

def _calculate_streak_factor(self) -> float:
    """
    Calcule un facteur de risque basé sur les séquences de pertes/gains
    """
    # Après des pertes consécutives, réduire progressivement le risque
    if self.consecutive_losses >= 4:
        return 0.5  # Réduire le risque de 50% après 4 pertes consécutives
    elif self.consecutive_losses >= 3:
        return 0.6
    elif self.consecutive_losses >= 2:
        return 0.7
    elif self.consecutive_losses >= 1:
        return 0.8
```

```
    # Après des gains consécutifs, augmenter légèrement le risque
    if self.win_streak >= 3:
        return 1.2  # Augmenter le risque de 20% après 3 gains consécutifs
    elif self.win_streak >= 2:
        return 1.1

    return 1.0  # Pas d'ajustement

def _calculate_drawdown_factor(self) -> float:
    """
    Calcule un facteur de risque basé sur le drawdown actuel
    """
    # Réduire le risque si nous sommes en drawdown significatif
    if self.current_drawdown > 35:
        return 0.5  # Réduire le risque de 50% si drawdown > 35%
    elif self.current_drawdown > 25:
        return 0.6
    elif self.current_drawdown > 15:
        return 0.8

    return 1.0  # Pas d'ajustement

def update_after_trade_closed(self, trade_result: Dict) -> None:
    """
    Met à jour les statistiques après la fermeture d'un trade - Version améliorée

    Args:
        trade_result: Résultat du trade
    """
    pnl = trade_result.get("pnl_absolute", 0)
```

```
pnl_percent = trade_result.get("pnl_percent", 0)

# Mettre à jour les métriques de base
if pnl > 0:
    self.daily_profits += pnl
    self.consecutive_losses = 0
    self.win_streak += 1
    self.loss_streak = 0
    logger.info(f"Profit ajouté: {pnl} USDT (total journalier: {self.daily_profits} USDT)")
else:
    self.daily_losses += pnl
    self.consecutive_losses += 1
    self.win_streak = 0
    self.loss_streak += 1
    logger.info(f"Perte ajoutée: {pnl} USDT (total journalier: {self.daily_losses} USDT)")

# Mettre à jour les records
self.max_win_streak = max(self.max_win_streak, self.win_streak)
self.max_loss_streak = max(self.max_loss_streak, self.loss_streak)

# Mettre à jour l'équité
self.equity += pnl

# Mettre à jour le peak equity et le drawdown
if self.equity > self.peak_equity:
    self.peak_equity = self.equity
    self.current_drawdown = 0
else:
    self.current_drawdown = (self.peak_equity - self.equity) / self.peak_equity * 100

# Enregistrer les données de performance
```

```
        self.performance_history.append({
            "timestamp": datetime.now().isoformat(),
            "trade_id": trade_result.get("trade_id"),
            "pnl": pnl,
            "pnl_percent": pnl_percent,
            "equity": self.equity,
            "drawdown": self.current_drawdown
        })

        # Si le drawdown dépasse 40%, déclencher une alerte
        if self.current_drawdown > 40:
            logger.warning(f"ALERTE: Drawdown élevé détecté ({self.current_drawdown:.2f}%)!")

            # Vous pourriez ajouter ici une logique pour arrêter temporairement le trading
            # ou réduire davantage la taille des positions

    def can_open_new_position(self, position_tracker) -> Dict:
        """
        Vérifie si une nouvelle position peut être ouverte - Version améliorée

        Args:
            position_tracker: Tracker de positions

        Returns:
            Dict avec résultat et raison
        """
        # Mettre à jour les statistiques
        self.update_position_stats(position_tracker)

        # Vérifier le nombre maximum de positions simultanées
        if self.positions_count >= MAX_CONCURRENT_TRADES:
```

```
            return {
                "can_open": False,
                "reason": f"Nombre maximum de positions simultanées atteint ({MAX_CONCURRENT_TRADES})"
            }

        # Vérifier le nombre maximum de trades par jour
        if self.daily_trade_count >= MAX_DAILY_TRADES:
            return {
                "can_open": False,
                "reason": f"Nombre maximum de trades journaliers atteint ({MAX_DAILY_TRADES})"
            }

        # Vérifier si le solde disponible est suffisant
        if self.available_balance <= 0:
            return {
                "can_open": False,
                "reason": "Solde insuffisant pour ouvrir une nouvelle position"
            }

        # NOUVEAU: Vérifier si nous sommes en drawdown critique
        if self.current_drawdown > MAX_DRAWDOWN_LIMIT:
            return {
                "can_open": False,
                "reason": f"Drawdown maximum dépassé ({self.current_drawdown:.2f}% > {MAX_DRAWDOWN_LIMIT}%)"
            }

        # NOUVEAU: Vérifier si nous avons trop de pertes consécutives
        if self.consecutive_losses >= 5:
            return {
                "can_open": False,
```

```
            "reason": f"Trop de pertes consécutives ({self.consecutive_losses})"
        }

    return {
        "can_open": True,
        "reason": "Conditions de risque respectées"
    }
```

```
================================================
File: crypto_trading_bot_CLAUDE/dashboard/model_monitor.py
================================================
"""
Système de monitoring avancé pour visualiser et analyser les performances du modèle
et les décisions de trading en temps réel
"""
import os
import json
import pandas as pd
import numpy as np
from typing import Dict, List, Tuple, Union, Optional, Any
from datetime import datetime, timedelta
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
from matplotlib.dates import DateFormatter
import seaborn as sns
from collections import defaultdict
import plotly.graph_objects as go
from plotly.subplots import make_subplots
import base64
from io import BytesIO
```

```
from config.config import DATA_DIR
from utils.logger import setup_logger

logger = setup_logger("model_monitor")

class ModelMonitor:
    """
    Système de monitoring pour visualiser et analyser les performances du modèle LSTM
    et les décisions de trading en temps réel
    """
    def __init__(self, model=None, data_dir: str = None):
        """
        Initialise le système de monitoring

        Args:
            model: Modèle à monitorer (LSTM ou autre)
            data_dir: Répertoire de données
        """
        self.model = model
        self.data_dir = data_dir or os.path.join(DATA_DIR, "monitoring")

        # Créer le répertoire si nécessaire
        os.makedirs(self.data_dir, exist_ok=True)

        # Historique des prédictions
        self.prediction_history = []

        # Historique des trades exécutés
        self.trade_history = []

        # Performances du modèle
```

```
        self.model_performance = {
            "accuracy": [],
            "precision": [],
            "recall": [],
            "f1_score": [],
            "timestamps": []
        }

        # Logs hiérarchiques
        self.logs = {
            "error": [],
            "warning": [],
            "info": [],
            "debug": []
        }

        # Attributions de performance
        self.performance_attribution = {
            "model_contribution": [],
            "technical_contribution": [],
            "market_contribution": [],
            "timestamps": []
        }

        # Charger les données existantes
        self._load_data()

    def record_prediction(self, symbol: str, prediction: Dict, actual_data: Optional[Dict] = None,
                          timestamp: str = None) -> None:
        """
        Enregistre une prédiction du modèle
```

```
    Args:
        symbol: Paire de trading
        prediction: Prédiction du modèle
        actual_data: Données réelles (si disponibles)
        timestamp: Horodatage de la prédiction
    """
    if timestamp is None:
        timestamp = datetime.now().isoformat()

    # Ajouter à l'historique des prédictions
    prediction_record = {
        "symbol": symbol,
        "timestamp": timestamp,
        "prediction": prediction,
        "actual": actual_data
    }

    self.prediction_history.append(prediction_record)

    # Limiter la taille de l'historique
    if len(self.prediction_history) > 10000:
        self.prediction_history = self.prediction_history[-10000:]

    # Sauvegarder périodiquement
    if len(self.prediction_history) % 100 == 0:
        self._save_data()

def record_trade(self, trade_data: Dict) -> None:
    """
    Enregistre un trade exécuté
```

```
    Args:
        trade_data: Données du trade
    """
    # Ajouter à l'historique des trades
    self.trade_history.append(trade_data)

    # Limiter la taille de l'historique
    if len(self.trade_history) > 1000:
        self.trade_history = self.trade_history[-1000:]

    # Sauvegarder
    self._save_data()

def update_model_performance(self, metrics: Dict, timestamp: str = None) -> None:
    """
    Met à jour les métriques de performance du modèle

    Args:
        metrics: Métriques de performance
        timestamp: Horodatage de l'évaluation
    """
    if timestamp is None:
        timestamp = datetime.now().isoformat()

    # Ajouter les métriques
    self.model_performance["accuracy"].append(metrics.get("accuracy", 0))
    self.model_performance["precision"].append(metrics.get("precision", 0))
    self.model_performance["recall"].append(metrics.get("recall", 0))
    self.model_performance["f1_score"].append(metrics.get("f1_score", 0))
    self.model_performance["timestamps"].append(timestamp)
```

```
    # Limiter la taille des historiques
    max_history = 1000
    for key in self.model_performance:
        if len(self.model_performance[key]) > max_history:
            self.model_performance[key] = self.model_performance[key][-max_history:]

    # Sauvegarder
    self._save_data()

def add_log(self, level: str, message: str, context: Optional[Dict] = None,
            timestamp: str = None) -> None:
    """
    Ajoute une entrée au journal hiérarchique

    Args:
        level: Niveau de log (error, warning, info, debug)
        message: Message de log
        context: Contexte additionnel
        timestamp: Horodatage du log
    """
    if timestamp is None:
        timestamp = datetime.now().isoformat()

    if level not in self.logs:
        level = "info"

    # Créer l'entrée de log
    log_entry = {
        "timestamp": timestamp,
        "message": message,
```

```
        "context": context or {}
    }

    self.logs[level].append(log_entry)

    # Limiter la taille des logs
    max_logs = 1000
    self.logs[level] = self.logs[level][-max_logs:]

    # Sauvegarder périodiquement
    if sum(len(logs) for logs in self.logs.values()) % 100 == 0:
        self._save_logs()

def update_performance_attribution(self, model_contrib: float, technical_contrib: float,
                                   market_contrib: float, timestamp: str = None) -> None:
    """
    Met à jour l'attribution de performance entre le modèle et les règles classiques

    Args:
        model_contrib: Contribution du modèle (0-1)
        technical_contrib: Contribution des indicateurs techniques (0-1)
        market_contrib: Contribution des conditions de marché (0-1)
        timestamp: Horodatage de l'évaluation
    """
    if timestamp is None:
        timestamp = datetime.now().isoformat()

    # Ajouter les attributions
    self.performance_attribution["model_contribution"].append(model_contrib)
    self.performance_attribution["technical_contribution"].append(technical_contrib)
    self.performance_attribution["market_contribution"].append(market_contrib)
```

```
        self.performance_attribution["timestamps"].append(timestamp)

        # Limiter la taille des historiques
        max_history = 1000
        for key in self.performance_attribution:
            if len(self.performance_attribution[key]) > max_history:
                self.performance_attribution[key] = self.performance_attribution[key][-max_history:]

        # Sauvegarder
        self._save_data()

    def _save_data(self) -> None:
        """Sauvegarde les données de monitoring"""
        try:
            # Sauvegarder l'historique des prédictions
            predictions_path = os.path.join(self.data_dir, "prediction_history.json")
            with open(predictions_path, 'w') as f:
                # Convertir en format sérialisable
                serializable_predictions = []

                for pred in self.prediction_history[-1000:]:  # Limiter à 1000 entrées pour la sauvegarde
                    serializable_pred = {
                        "symbol": pred["symbol"],
                        "timestamp": pred["timestamp"]
                    }

                    # Inclure les prédictions principales en format sérialisable
                    if "prediction" in pred:
                        prediction = pred["prediction"]
                        serializable_pred["prediction"] = {}
```

```
            # Parcourir les horizons et facteurs
            for horizon_key, horizon_data in prediction.items():
                serializable_pred["prediction"][horizon_key] = {}

                for factor_key, factor_value in horizon_data.items():
                    # Convertir les valeurs numpy et autres en types natifs
                    if isinstance(factor_value, (np.integer, np.floating)):
                        factor_value = float(factor_value)

                    serializable_pred["prediction"][horizon_key][factor_key] = factor_value

        # Inclure les données réelles si disponibles
        if "actual" in pred and pred["actual"] is not None:
            serializable_pred["actual"] = {}

            for key, value in pred["actual"].items():
                # Convertir les valeurs numpy et autres en types natifs
                if isinstance(value, (np.integer, np.floating)):
                    value = float(value)

                serializable_pred["actual"][key] = value

        serializable_predictions.append(serializable_pred)

    json.dump(serializable_predictions, f, indent=2)

# Sauvegarder l'historique des trades
trades_path = os.path.join(self.data_dir, "trade_history.json")
with open(trades_path, 'w') as f:
    json.dump(self.trade_history, f, indent=2, default=str)
```

```
            # Sauvegarder les performances du modèle
            performance_path = os.path.join(self.data_dir, "model_performance.json")
            with open(performance_path, 'w') as f:
                json.dump(self.model_performance, f, indent=2)

            # Sauvegarder l'attribution de performance
            attribution_path = os.path.join(self.data_dir, "performance_attribution.json")
            with open(attribution_path, 'w') as f:
                json.dump(self.performance_attribution, f, indent=2)

            logger.debug("Données de monitoring sauvegardées")
        except Exception as e:
            logger.error(f"Erreur lors de la sauvegarde des données de monitoring: {str(e)}")

    def _save_logs(self) -> None:
        """Sauvegarde les logs hiérarchiques"""
        try:
            logs_path = os.path.join(self.data_dir, "monitoring_logs.json")
            with open(logs_path, 'w') as f:
                json.dump(self.logs, f, indent=2)

            logger.debug("Logs de monitoring sauvegardés")
        except Exception as e:
            logger.error(f"Erreur lors de la sauvegarde des logs de monitoring: {str(e)}")

    def _load_data(self) -> None:
        """Charge les données de monitoring existantes"""
        try:
            # Charger l'historique des prédictions
            predictions_path = os.path.join(self.data_dir, "prediction_history.json")
            if os.path.exists(predictions_path):
```

```
                with open(predictions_path, 'r') as f:
                    self.prediction_history = json.load(f)

            # Charger l'historique des trades
            trades_path = os.path.join(self.data_dir, "trade_history.json")
            if os.path.exists(trades_path):
                with open(trades_path, 'r') as f:
                    self.trade_history = json.load(f)

            # Charger les performances du modèle
            performance_path = os.path.join(self.data_dir, "model_performance.json")
            if os.path.exists(performance_path):
                with open(performance_path, 'r') as f:
                    self.model_performance = json.load(f)

            # Charger l'attribution de performance
            attribution_path = os.path.join(self.data_dir, "performance_attribution.json")
            if os.path.exists(attribution_path):
                with open(attribution_path, 'r') as f:
                    self.performance_attribution = json.load(f)

            # Charger les logs
            logs_path = os.path.join(self.data_dir, "monitoring_logs.json")
            if os.path.exists(logs_path):
                with open(logs_path, 'r') as f:
                    self.logs = json.load(f)

            logger.info("Données de monitoring chargées")
        except Exception as e:
            logger.error(f"Erreur lors du chargement des données de monitoring: {str(e)}")
```

```
def get_recent_predictions(self, symbol: str = None, horizon: str = None,
                           limit: int = 100) -> List[Dict]:
    """
    Récupère les prédictions récentes

    Args:
        symbol: Filtrer par paire de trading
        horizon: Filtrer par horizon de prédiction
        limit: Nombre maximum de prédictions à retourner

    Returns:
        Liste des prédictions récentes
    """
    # Filtrer par symbole si spécifié
    if symbol:
        filtered_predictions = [p for p in self.prediction_history if p["symbol"] == symbol]
    else:
        filtered_predictions = self.prediction_history

    # Filtrer par horizon si spécifié
    if horizon and filtered_predictions:
        result = []

        for pred in filtered_predictions:
            if "prediction" in pred and horizon in pred["prediction"]:
                # Copier la prédiction et conserver uniquement l'horizon demandé
                filtered_pred = pred.copy()
                filtered_pred["prediction"] = {horizon: pred["prediction"][horizon]}
                result.append(filtered_pred)

        return result[-limit:]
```

```
    # Retourner les prédictions filtrées
    return filtered_predictions[-limit:]

def get_prediction_accuracy(self, symbol: str = None, horizon: str = None,
                            days: int = 30) -> Dict:
    """
    Calcule la précision des prédictions sur une période récente

    Args:
        symbol: Filtrer par paire de trading
        horizon: Filtrer par horizon de prédiction
        days: Nombre de jours à analyser

    Returns:
        Métriques de précision
    """
    # Calculer la date limite
    cutoff_date = (datetime.now() - timedelta(days=days)).isoformat()

    # Filtrer les prédictions récentes avec données réelles
    recent_predictions = [
        p for p in self.prediction_history
        if p.get("timestamp", "") >= cutoff_date and p.get("actual") is not None
    ]

    # Filtrer par symbole si spécifié
    if symbol:
        recent_predictions = [p for p in recent_predictions if p["symbol"] == symbol]

    if not recent_predictions:
```

```
    return {
        "accuracy": 0,
        "total_predictions": 0,
        "message": "Données insuffisantes"
    }

# Compteurs
correct = 0
total = 0

# Analyser chaque prédiction
for pred in recent_predictions:
    if "prediction" not in pred or "actual" not in pred:
        continue

    # Filtrer par horizon si spécifié
    horizons_to_check = [horizon] if horizon else pred["prediction"].keys()

    for h in horizons_to_check:
        if h in pred["prediction"]:
            # Récupérer la direction prédite
            predicted_direction = pred["prediction"][h].get("direction", "")

            # Récupérer la direction réelle
            actual_direction = "HAUSSIER" if pred["actual"].get("price_change", 0) > 0 else "BAISSIER"

            # Comparer
            if predicted_direction and predicted_direction == actual_direction:
                correct += 1

            total += 1
```

```
    # Calculer la précision
    accuracy = correct / total if total > 0 else 0

    return {
        "accuracy": accuracy,
        "correct_predictions": correct,
        "total_predictions": total,
        "period_days": days
    }

def get_trade_performance(self, symbol: str = None, days: int = 30) -> Dict:
    """
    Calcule les performances des trades sur une période récente

    Args:
        symbol: Filtrer par paire de trading
        days: Nombre de jours à analyser

    Returns:
        Métriques de performance des trades
    """
    # Calculer la date limite
    cutoff_date = (datetime.now() - timedelta(days=days)).isoformat()

    # Filtrer les trades récents
    recent_trades = [
        t for t in self.trade_history
        if t.get("timestamp", "") >= cutoff_date or t.get("entry_time", "") >= cutoff_date
    ]
```

```
# Filtrer par symbole si spécifié
if symbol:
    recent_trades = [t for t in recent_trades if t.get("symbol", "") == symbol]

if not recent_trades:
    return {
        "total_trades": 0,
        "win_rate": 0,
        "profit_factor": 0,
        "message": "Données insuffisantes"
    }

# Calculer les métriques
total_trades = len(recent_trades)
winning_trades = [t for t in recent_trades if t.get("pnl_percent", 0) > 0]
losing_trades = [t for t in recent_trades if t.get("pnl_percent", 0) <= 0]

win_rate = len(winning_trades) / total_trades if total_trades > 0 else 0

# Calculer le profit factor
total_profit = sum(t.get("pnl_percent", 0) for t in winning_trades)
total_loss = abs(sum(t.get("pnl_percent", 0) for t in losing_trades))

profit_factor = total_profit / total_loss if total_loss > 0 else float('inf')

# Calculer les autres métriques
avg_profit = total_profit / len(winning_trades) if winning_trades else 0
avg_loss = total_loss / len(losing_trades) if losing_trades else 0

# Calculer le drawdown maximum
equity_curve = self._calculate_equity_curve(recent_trades)
```

```
    max_drawdown = self._calculate_max_drawdown(equity_curve)

    return {
        "total_trades": total_trades,
        "winning_trades": len(winning_trades),
        "losing_trades": len(losing_trades),
        "win_rate": win_rate,
        "profit_factor": profit_factor,
        "avg_profit": avg_profit,
        "avg_loss": avg_loss,
        "max_drawdown": max_drawdown,
        "period_days": days
    }

def _calculate_equity_curve(self, trades: List[Dict]) -> List[float]:
    """
    Calcule la courbe d'équité à partir des trades

    Args:
        trades: Liste des trades

    Returns:
        Courbe d'équité
    """
    # Trier les trades par date
    sorted_trades = sorted(trades, key=lambda t: t.get("entry_time", "") or t.get("timestamp", ""))

    # Initialiser la courbe d'équité avec un capital initial de 100
    equity = 100.0
    equity_curve = [equity]
```

```
    # Calculer l'équité après chaque trade
    for trade in sorted_trades:
        pnl_percent = trade.get("pnl_percent", 0)
        equity *= (1 + pnl_percent / 100)
        equity_curve.append(equity)

    return equity_curve

def _calculate_max_drawdown(self, equity_curve: List[float]) -> float:
    """
    Calcule le drawdown maximum à partir de la courbe d'équité

    Args:
        equity_curve: Courbe d'équité

    Returns:
        Drawdown maximum en pourcentage
    """
    if not equity_curve or len(equity_curve) < 2:
        return 0.0

    max_dd = 0.0
    peak = equity_curve[0]

    for equity in equity_curve:
        if equity > peak:
            peak = equity

        dd = (peak - equity) / peak * 100 if peak > 0 else 0
        max_dd = max(max_dd, dd)
```

```
        return max_dd

    def generate_model_insights(self, symbol: str = None, days: int = 30) -> Dict:
        """
        Génère des insights sur les performances du modèle

        Args:
            symbol: Filtrer par paire de trading
            days: Nombre de jours à analyser

        Returns:
            Insights sur les performances du modèle
        """
        # Récupérer les prédictions récentes
        prediction_accuracy = self.get_prediction_accuracy(symbol, days=days)

        # Récupérer les performances des trades
        trade_performance = self.get_trade_performance(symbol, days=days)

        # Récupérer les métriques de performance du modèle
        model_metrics = {
            "accuracy": self.model_performance["accuracy"][-1] if self.model_performance["accuracy"]
else 0,
            "precision": self.model_performance["precision"][-1] if self.model_performance["precision"]
else 0,
            "recall": self.model_performance["recall"][-1] if self.model_performance["recall"] else 0,
            "f1_score": self.model_performance["f1_score"][-1] if self.model_performance["f1_score"]
else 0
        }

        # Récupérer les attributions de performance
        attributions = {
```

```
            "model": self.performance_attribution["model_contribution"][-1] if
self.performance_attribution["model_contribution"] else 0,
            "technical": self.performance_attribution["technical_contribution"][-1] if
self.performance_attribution["technical_contribution"] else 0,
            "market": self.performance_attribution["market_contribution"][-1] if
self.performance_attribution["market_contribution"] else 0
        }

        # Générer les insights
        insights = []

        # Insight sur la précision du modèle
        if prediction_accuracy["accuracy"] > 0.65:
            insights.append({
                "type": "strength",
                "message": f"Le modèle montre une forte précision de prédiction
({prediction_accuracy['accuracy']:.1%})"
            })
        elif prediction_accuracy["accuracy"] < 0.5:
            insights.append({
                "type": "weakness",
                "message": f"La précision de prédiction est faible ({prediction_accuracy['accuracy']:.1%})"
            })

        # Insight sur le win rate
        if trade_performance["win_rate"] > 0.6:
            insights.append({
                "type": "strength",
                "message": f"Excellent win rate sur les trades ({trade_performance['win_rate']:.1%})"
            })
        elif trade_performance["win_rate"] < 0.4:
            insights.append({
```

```
            "type": "weakness",
            "message": f"Win rate insuffisant ({trade_performance['win_rate']:.1%}), réévaluer la stratégie"
        })

    # Insight sur le profit factor
    if trade_performance["profit_factor"] > 2.0:
        insights.append({
            "type": "strength",
            "message": f"Profit factor excellent ({trade_performance['profit_factor']:.2f})"
        })
    elif trade_performance["profit_factor"] < 1.0:
        insights.append({
            "type": "weakness",
            "message": f"Profit factor inférieur à 1 ({trade_performance['profit_factor']:.2f}), les pertes dépassent les gains"
        })

    # Insight sur l'attribution de performance
    if attributions["model"] > 0.6:
        insights.append({
            "type": "strength",
            "message": f"Le modèle LSTM contribue fortement à la performance ({attributions['model']:.1%})"
        })
    elif attributions["technical"] > attributions["model"]:
        insights.append({
            "type": "info",
            "message": f"Les indicateurs techniques sont plus déterminants ({attributions['technical']:.1%}) que le modèle ({attributions['model']:.1%})"
        })
```

```
        # Calculer les tendances
        trends = self._calculate_performance_trends()

        return {
            "prediction_accuracy": prediction_accuracy,
            "trade_performance": trade_performance,
            "model_metrics": model_metrics,
            "performance_attribution": attributions,
            "insights": insights,
            "trends": trends
        }

    def _calculate_performance_trends(self) -> Dict:
        """
        Calcule les tendances de performance

        Returns:
            Tendances de performance
        """
        trends = {}

        # Calculer la tendance de précision
        if len(self.model_performance["accuracy"]) > 5:
            accuracy_trend = self.model_performance["accuracy"][-1] -
self.model_performance["accuracy"][-5]
            trends["accuracy"] = {
                "direction": "improving" if accuracy_trend > 0 else "declining",
                "change": accuracy_trend
            }

        # Calculer la tendance de f1_score
```

```
    if len(self.model_performance["f1_score"]) > 5:
        f1_trend = self.model_performance["f1_score"][-1] - self.model_performance["f1_score"][-5]
        trends["f1_score"] = {
            "direction": "improving" if f1_trend > 0 else "declining",
            "change": f1_trend
        }

    # Calculer la tendance de contribution du modèle
    if len(self.performance_attribution["model_contribution"]) > 5:
        model_contrib_trend = (
            self.performance_attribution["model_contribution"][-1] -
            self.performance_attribution["model_contribution"][-5]
        )
        trends["model_contribution"] = {
            "direction": "improving" if model_contrib_trend > 0 else "declining",
            "change": model_contrib_trend
        }

    return trends

def create_performance_dashboard(self, days: int = 30, symbol: str = None) -> BytesIO:
    """
    Crée un tableau de bord complet des performances

    Args:
        days: Nombre de jours à analyser
        symbol: Filtrer par paire de trading

    Returns:
        Tableau de bord sous forme d'image
    """
```

```
# Créer une figure avec plusieurs sous-graphiques
plt.style.use('fivethirtyeight')
fig = plt.figure(figsize=(14, 16))
gs = gridspec.GridSpec(5, 2, figure=fig)

# Récupérer les données
insights = self.generate_model_insights(symbol, days)
prediction_accuracy = insights["prediction_accuracy"]
trade_performance = insights["trade_performance"]
model_metrics = insights["model_metrics"]
attributions = insights["performance_attribution"]

# 1. Courbe d'équité
equity_ax = fig.add_subplot(gs[0, :])
self._plot_equity_curve(equity_ax, days, symbol)

# 2. Précision des prédictions par horizon
pred_ax = fig.add_subplot(gs[1, 0])
self._plot_prediction_accuracy(pred_ax, days, symbol)

# 3. Distribution des profits/pertes
pnl_ax = fig.add_subplot(gs[1, 1])
self._plot_pnl_distribution(pnl_ax, days, symbol)

# 4. Performance du modèle
model_ax = fig.add_subplot(gs[2, 0])
self._plot_model_performance(model_ax)

# 5. Attribution de performance
attr_ax = fig.add_subplot(gs[2, 1])
self._plot_performance_attribution(attr_ax)
```

```
    # 6. Répartition des trades par résultat
    trade_ax = fig.add_subplot(gs[3, 0])
    self._plot_trade_breakdown(trade_ax, days, symbol)

    # 7. Analyse des horizons
    horizon_ax = fig.add_subplot(gs[3, 1])
    self._plot_horizon_analysis(horizon_ax, days, symbol)

    # 8. Tableau des métriques clés
    metrics_ax = fig.add_subplot(gs[4, :])
    self._plot_key_metrics_table(metrics_ax, insights)

    # Ajuster la mise en page
    plt.tight_layout()

    # Sauvegarder la figure dans un buffer
    buf = BytesIO()
    plt.savefig(buf, format='png', dpi=100, bbox_inches='tight')
    buf.seek(0)

    # Fermer la figure pour libérer la mémoire
    plt.close(fig)

    return buf

def _plot_equity_curve(self, ax, days: int, symbol: str = None) -> None:
    """
    Trace la courbe d'équité

    Args:
```

```
    ax: Axes matplotlib
    days: Nombre de jours à analyser
    symbol: Filtrer par paire de trading
"""
# Calculer la date limite
cutoff_date = (datetime.now() - timedelta(days=days)).isoformat()

# Filtrer les trades récents
recent_trades = [
    t for t in self.trade_history
    if (t.get("timestamp", "") >= cutoff_date or t.get("entry_time", "") >= cutoff_date)
]

# Filtrer par symbole si spécifié
if symbol:
    recent_trades = [t for t in recent_trades if t.get("symbol", "") == symbol]

if not recent_trades:
    ax.text(0.5, 0.5, "Données insuffisantes", ha='center', va='center')
    ax.set_title("Courbe d'équité")
    return

# Trier les trades par date
def get_timestamp(trade):
    return trade.get("exit_time", "") or trade.get("entry_time", "") or trade.get("timestamp", "")

sorted_trades = sorted(recent_trades, key=get_timestamp)

# Initialiser la courbe d'équité avec un capital initial de 100
equity = 100.0
equity_curve = [equity]
```

```
    dates = [datetime.fromisoformat(get_timestamp(sorted_trades[0]))] if sorted_trades else [datetime.now()]

    # Calculer l'équité après chaque trade
    for trade in sorted_trades:
        pnl_percent = trade.get("pnl_percent", 0)
        equity *= (1 + pnl_percent / 100)
        equity_curve.append(equity)

        # Ajouter la date
        try:
            date = datetime.fromisoformat(get_timestamp(trade))
        except:
            date = dates[-1] + timedelta(hours=1)  # Fallback

        dates.append(date)

    # Tracer la courbe d'équité
    ax.plot(dates, equity_curve, 'b-', linewidth=2)

    # Ajouter les points de trade
    for i, trade in enumerate(sorted_trades):
        pnl = trade.get("pnl_percent", 0)
        color = 'green' if pnl > 0 else 'red'
        marker = '^' if pnl > 0 else 'v'
        try:
            date = datetime.fromisoformat(get_timestamp(trade))
            idx = dates.index(date)
            ax.plot(date, equity_curve[idx], marker=marker, color=color, markersize=8)
        except:
            continue
```

```
        # Calculer le drawdown
        max_dd = self._calculate_max_drawdown(equity_curve)
        total_return = (equity_curve[-1] / equity_curve[0] - 1) * 100

        # Ajouter les informations sur le graphique
        ax.set_title(f"Courbe d'équité {symbol + ' ' if symbol else ''}(Rendement: {total_return:.1f}%, DD Max: {max_dd:.1f}%)")
        ax.set_ylabel("Équité (%)")
        ax.xaxis.set_major_formatter(DateFormatter("%d/%m"))
        ax.grid(True, alpha=0.3)

    def _plot_prediction_accuracy(self, ax, days: int, symbol: str = None) -> None:
        """
        Trace la précision des prédictions par horizon

        Args:
            ax: Axes matplotlib
            days: Nombre de jours à analyser
            symbol: Filtrer par paire de trading
        """
        # Horizons à analyser
        horizons = ["3h", "12h", "48h", "96h", "short_term", "medium_term", "long_term"]

        # Calculer la précision pour chaque horizon
        accuracies = []
        labels = []

        for horizon in horizons:
            accuracy = self.get_prediction_accuracy(symbol, horizon, days)
```

```
            # Si suffisamment de prédictions
            if accuracy["total_predictions"] > 10:
                accuracies.append(accuracy["accuracy"] * 100)  # En pourcentage
                labels.append(horizon)

        if not accuracies:
            ax.text(0.5, 0.5, "Données insuffisantes", ha='center', va='center')
            ax.set_title("Précision des prédictions par horizon")
            return

        # Tracer le graphique à barres
        colors = ['#3498db', '#2980b9', '#1f618d', '#154360', '#512E5F', '#4A235A', '#0B5345']
        bars = ax.bar(labels, accuracies, color=colors[:len(labels)])

        # Ajouter les valeurs au-dessus des barres
        for bar in bars:
            height = bar.get_height()
            ax.text(bar.get_x() + bar.get_width()/2., height + 1,
                    f'{height:.1f}%', ha='center', va='bottom')

        # Ajouter une ligne horizontale à 50% (hasard)
        ax.axhline(y=50, color='r', linestyle='--', alpha=0.7)

        # Configurer le graphique
        ax.set_title("Précision des prédictions par horizon")
        ax.set_ylabel("Précision (%)")
        ax.set_ylim(0, 100)
        ax.grid(True, alpha=0.3, axis='y')

    def _plot_pnl_distribution(self, ax, days: int, symbol: str = None) -> None:
        """
```

```
Trace la distribution des profits/pertes

Args:
    ax: Axes matplotlib
    days: Nombre de jours à analyser
    symbol: Filtrer par paire de trading
"""
# Calculer la date limite
cutoff_date = (datetime.now() - timedelta(days=days)).isoformat()

# Filtrer les trades récents
recent_trades = [
    t for t in self.trade_history
    if (t.get("timestamp", "") >= cutoff_date or t.get("entry_time", "") >= cutoff_date)
]

# Filtrer par symbole si spécifié
if symbol:
    recent_trades = [t for t in recent_trades if t.get("symbol", "") == symbol]

if not recent_trades:
    ax.text(0.5, 0.5, "Données insuffisantes", ha='center', va='center')
    ax.set_title("Distribution des profits/pertes")
    return

# Récupérer les pourcentages de PnL
pnl_values = [t.get("pnl_percent", 0) for t in recent_trades]

# Tracer l'histogramme
bins = np.linspace(min(pnl_values), max(pnl_values), 20)
ax.hist(pnl_values, bins=bins, alpha=0.7, color='skyblue', edgecolor='black')
```

```
        # Ajouter une ligne verticale à 0
        ax.axvline(x=0, color='r', linestyle='--', alpha=0.7)

        # Configurer le graphique
        ax.set_title("Distribution des profits/pertes")
        ax.set_xlabel("P&L (%)")
        ax.set_ylabel("Fréquence")
        ax.grid(True, alpha=0.3)

    def _plot_model_performance(self, ax) -> None:
        """
        Trace les métriques de performance du modèle

        Args:
            ax: Axes matplotlib
        """
        # Vérifier s'il y a suffisamment de données
        if not self.model_performance["timestamps"]:
            ax.text(0.5, 0.5, "Données insuffisantes", ha='center', va='center')
            ax.set_title("Performance du modèle")
            return

        # Convertir les timestamps en datetime
        try:
            dates = [datetime.fromisoformat(ts) for ts in self.model_performance["timestamps"]]
        except:
            # Fallback: créer des dates séquentielles
            dates = [datetime.now() - timedelta(days=i) for i in
range(len(self.model_performance["timestamps"]), 0, -1)]
```

```
    # Tracer les métriques
    ax.plot(dates, self.model_performance["accuracy"], 'b-', label="Accuracy")
    ax.plot(dates, self.model_performance["f1_score"], 'g--', label="F1-Score")
    ax.plot(dates, self.model_performance["precision"], 'r-.', label="Precision")
    ax.plot(dates, self.model_performance["recall"], 'c:', label="Recall")

    # Configurer le graphique
    ax.set_title("Performance du modèle")
    ax.set_ylabel("Métrique")
    ax.xaxis.set_major_formatter(DateFormatter("%d/%m"))
    ax.grid(True, alpha=0.3)
    ax.legend()

def _plot_performance_attribution(self, ax) -> None:
    """
    Trace l'attribution de performance

    Args:
        ax: Axes matplotlib
    """
    # Vérifier s'il y a suffisamment de données
    if not self.performance_attribution["timestamps"]:
        ax.text(0.5, 0.5, "Données insuffisantes", ha='center', va='center')
        ax.set_title("Attribution de performance")
        return

    # Convertir les timestamps en datetime
    try:
        dates = [datetime.fromisoformat(ts) for ts in self.performance_attribution["timestamps"]]
    except:
        # Fallback: créer des dates séquentielles
```

```
        dates = [datetime.now() - timedelta(days=i) for i in
range(len(self.performance_attribution["timestamps"]), 0, -1)]

        # Tracer les attributions
        ax.stackplot(
            dates,
            self.performance_attribution["model_contribution"],
            self.performance_attribution["technical_contribution"],
            self.performance_attribution["market_contribution"],
            labels=["Modèle LSTM", "Indicateurs techniques", "Conditions de marché"],
            colors=['#3498db', '#2ecc71', '#f39c12'],
            alpha=0.7
        )

        # Configurer le graphique
        ax.set_title("Attribution de performance")
        ax.set_ylabel("Contribution")
        ax.xaxis.set_major_formatter(DateFormatter("%d/%m"))
        ax.grid(True, alpha=0.3)
        ax.legend(loc='upper left', fontsize='small')

    def _plot_trade_breakdown(self, ax, days: int, symbol: str = None) -> None:
        """
        Trace la répartition des trades par résultat

        Args:
            ax: Axes matplotlib
            days: Nombre de jours à analyser
            symbol: Filtrer par paire de trading
        """
        # Calculer la date limite
```

```
        cutoff_date = (datetime.now() - timedelta(days=days)).isoformat()

        # Filtrer les trades récents
        recent_trades = [
            t for t in self.trade_history
            if (t.get("timestamp", "") >= cutoff_date or t.get("entry_time", "") >= cutoff_date)
        ]

        # Filtrer par symbole si spécifié
        if symbol:
            recent_trades = [t for t in recent_trades if t.get("symbol", "") == symbol]

        if not recent_trades:
            ax.text(0.5, 0.5, "Données insuffisantes", ha='center', va='center')
            ax.set_title("Répartition des trades")
            return

        # Catégoriser les trades
        categories = {
            "profit_major": len([t for t in recent_trades if t.get("pnl_percent", 0) > 5]),
            "profit_minor": len([t for t in recent_trades if 0 < t.get("pnl_percent", 0) <= 5]),
            "loss_minor": len([t for t in recent_trades if -5 <= t.get("pnl_percent", 0) < 0]),
            "loss_major": len([t for t in recent_trades if t.get("pnl_percent", 0) < -5])
        }

        # Tracer le graphique à camembert
        labels = ["Profit majeur (>5%)", "Profit mineur (0-5%)", "Perte mineure (0-5%)", "Perte majeure (>5%)"]
        sizes = [categories["profit_major"], categories["profit_minor"], categories["loss_minor"], categories["loss_major"]]
        colors = ['#27ae60', '#2ecc71', '#e74c3c', '#c0392b']
        explode = (0.1, 0, 0, 0.1)  # Faire ressortir les catégories extrêmes
```

```
    ax.pie(sizes, explode=explode, labels=labels, colors=colors, autopct='%1.1f%%',
           shadow=True, startangle=90)
    ax.axis('equal')  # Pour avoir un cercle parfait

    # Configurer le graphique
    ax.set_title("Répartition des trades par résultat")

def _plot_horizon_analysis(self, ax, days: int, symbol: str = None) -> None:
    """
    Trace l'analyse des horizons

    Args:
        ax: Axes matplotlib
        days: Nombre de jours à analyser
        symbol: Filtrer par paire de trading
    """
    # Horizons à analyser
    horizons = ["3h", "12h", "48h", "96h", "short_term", "medium_term", "long_term"]

    # Calculer la précision pour chaque horizon
    accuracies = {}

    for horizon in horizons:
        accuracy = self.get_prediction_accuracy(symbol, horizon, days)

        # Si suffisamment de prédictions
        if accuracy["total_predictions"] > 10:
            accuracies[horizon] = {
                "accuracy": accuracy["accuracy"],
                "total": accuracy["total_predictions"]
```

```
        }

if not accuracies:
    ax.text(0.5, 0.5, "Données insuffisantes", ha='center', va='center')
    ax.set_title("Analyse des horizons")
    return

# Créer un dataframe pour l'analyse
df = pd.DataFrame({
    "Horizon": list(accuracies.keys()),
    "Précision": [acc["accuracy"] * 100 for acc in accuracies.values()],
    "Nombre de prédictions": [acc["total"] for acc in accuracies.values()]
})

# Trier par précision
df = df.sort_values("Précision", ascending=False)

# Tracer le graphique à barres
sns.barplot(x="Horizon", y="Précision", data=df, ax=ax, palette="viridis")

# Ajouter une ligne horizontale à 50% (hasard)
ax.axhline(y=50, color='r', linestyle='--', alpha=0.7)

# Ajouter les nombres de prédictions au-dessus des barres
for i, v in enumerate(df["Nombre de prédictions"]):
    ax.text(i, df["Précision"].iloc[i] + 1, str(v), ha='center')

# Configurer le graphique
ax.set_title("Analyse des horizons (précision)")
ax.set_ylim(0, 100)
ax.grid(True, alpha=0.3, axis='y')
```

```
    def _plot_key_metrics_table(self, ax, insights: Dict) -> None:
        """
        Trace un tableau des métriques clés

        Args:
            ax: Axes matplotlib
            insights: Insights sur les performances du modèle
        """
        # Désactiver les axes
        ax.axis('off')

        # Extraire les données clés
        prediction_accuracy = insights["prediction_accuracy"]
        trade_performance = insights["trade_performance"]
        model_metrics = insights["model_metrics"]
        attributions = insights["performance_attribution"]
        trends = insights.get("trends", {})

        # Créer les données du tableau
        data = [
            ["Métrique", "Valeur", "Tendance"],
            ["Précision de prédiction", f"{prediction_accuracy['accuracy']:.1%}",
self._get_trend_arrow(trends.get("accuracy", {}))],
            ["Win rate", f"{trade_performance['win_rate']:.1%}", ""],
            ["Profit factor", f"{trade_performance['profit_factor']:.2f}", ""],
            ["Drawdown maximum", f"{trade_performance['max_drawdown']:.1f}%", ""],
            ["F1-Score du modèle", f"{model_metrics['f1_score']:.2f}",
self._get_trend_arrow(trends.get("f1_score", {}))],
            ["Contribution du modèle", f"{attributions['model']:.1%}",
self._get_trend_arrow(trends.get("model_contribution", {}))]
        ]
```

```
    # Créer le tableau
    table = ax.table(
        cellText=data,
        cellLoc='center',
        loc='center',
        colWidths=[0.4, 0.3, 0.3]
    )

    # Styliser le tableau
    table.auto_set_font_size(False)
    table.set_fontsize(12)
    table.scale(1, 1.5)

    # Styliser l'en-tête
    for i in range(3):
        table[(0, i)].set_facecolor('#3498db')
        table[(0, i)].set_text_props(color='white', fontweight='bold')

    # Styliser les lignes alternées
    for i in range(1, len(data)):
        if i % 2 == 0:
            for j in range(3):
                table[(i, j)].set_facecolor('#f5f5f5')

def _get_trend_arrow(self, trend: Dict) -> str:
    """
    Retourne une flèche indiquant la tendance

    Args:
        trend: Dictionnaire de tendance
```

```
    Returns:
        Flèche de tendance
    """
    direction = trend.get("direction", "")

    if direction == "improving":
        return "↑"
    elif direction == "declining":
        return "↓"
    else:
        return "→"

def create_plotly_dashboard(self, days: int = 30, symbol: str = None) -> Dict:
    """
    Crée un tableau de bord interactif avec Plotly

    Args:
        days: Nombre de jours à analyser
        symbol: Filtrer par paire de trading

    Returns:
        Dictionnaire avec les figures Plotly
    """
    # Récupérer les données
    insights = self.generate_model_insights(symbol, days)

    # 1. Courbe d'équité
    equity_fig = self._create_plotly_equity_curve(days, symbol)

    # 2. Précision des prédictions par horizon
```

```
    prediction_fig = self._create_plotly_prediction_accuracy(days, symbol)

    # 3. Distribution des profits/pertes
    pnl_fig = self._create_plotly_pnl_distribution(days, symbol)

    # 4. Performance du modèle
    model_fig = self._create_plotly_model_performance()

    # 5. Attribution de performance
    attribution_fig = self._create_plotly_performance_attribution()

    # 6. Tableau des métriques clés
    metrics_fig = self._create_plotly_metrics_table(insights)

    return {
        "equity_curve": equity_fig,
        "prediction_accuracy": prediction_fig,
        "pnl_distribution": pnl_fig,
        "model_performance": model_fig,
        "performance_attribution": attribution_fig,
        "metrics_table": metrics_fig,
        "insights": insights
    }

def _create_plotly_equity_curve(self, days: int, symbol: str = None) -> go.Figure:
    """
    Crée une courbe d'équité interactive avec Plotly

    Args:
        days: Nombre de jours à analyser
        symbol: Filtrer par paire de trading
```

```
Returns:
    Figure Plotly
"""
# Calculer la date limite
cutoff_date = (datetime.now() - timedelta(days=days)).isoformat()

# Filtrer les trades récents
recent_trades = [
    t for t in self.trade_history
    if (t.get("timestamp", "") >= cutoff_date or t.get("entry_time", "") >= cutoff_date)
]

# Filtrer par symbole si spécifié
if symbol:
    recent_trades = [t for t in recent_trades if t.get("symbol", "") == symbol]

if not recent_trades:
    fig = go.Figure()
    fig.add_annotation(
        text="Données insuffisantes",
        xref="paper", yref="paper",
        x=0.5, y=0.5,
        showarrow=False,
        font=dict(size=20)
    )
    fig.update_layout(title="Courbe d'équité")
    return fig

# Trier les trades par date
def get_timestamp(trade):
```

```
            return trade.get("exit_time", "") or trade.get("entry_time", "") or trade.get("timestamp", "")

        sorted_trades = sorted(recent_trades, key=get_timestamp)

        # Initialiser la courbe d'équité avec un capital initial de 100
        equity = 100.0
        equity_curve = [equity]
        dates = [datetime.fromisoformat(get_timestamp(sorted_trades[0]))] if sorted_trades else [datetime.now()]
        trade_pnl = []

        # Calculer l'équité après chaque trade
        for trade in sorted_trades:
            pnl_percent = trade.get("pnl_percent", 0)
            equity *= (1 + pnl_percent / 100)
            equity_curve.append(equity)
            trade_pnl.append(pnl_percent)

            # Ajouter la date
            try:
                date = datetime.fromisoformat(get_timestamp(trade))
            except:
                date = dates[-1] + timedelta(hours=1)  # Fallback

            dates.append(date)

        # Créer la figure
        fig = go.Figure()

        # Ajouter la courbe d'équité
        fig.add_trace(go.Scatter(
```

```
    x=dates,
    y=equity_curve,
    mode='lines',
    name='Équité',
    line=dict(color='blue', width=2)
))

# Ajouter les points de trade
winning_trades_x = []
winning_trades_y = []
losing_trades_x = []
losing_trades_y = []

for i, trade in enumerate(sorted_trades):
    pnl = trade.get("pnl_percent", 0)

    try:
        date = datetime.fromisoformat(get_timestamp(trade))
        idx = dates.index(date)

        if pnl > 0:
            winning_trades_x.append(date)
            winning_trades_y.append(equity_curve[idx])
        else:
            losing_trades_x.append(date)
            losing_trades_y.append(equity_curve[idx])
    except:
        continue

# Ajouter les trades gagnants
fig.add_trace(go.Scatter(
```

```
    x=winning_trades_x,
    y=winning_trades_y,
    mode='markers',
    name='Trades gagnants',
    marker=dict(
        color='green',
        size=10,
        symbol='triangle-up'
    )
))

# Ajouter les trades perdants
fig.add_trace(go.Scatter(
    x=losing_trades_x,
    y=losing_trades_y,
    mode='markers',
    name='Trades perdants',
    marker=dict(
        color='red',
        size=10,
        symbol='triangle-down'
    )
))

# Calculer le drawdown
max_dd = self._calculate_max_drawdown(equity_curve)
total_return = (equity_curve[-1] / equity_curve[0] - 1) * 100

# Configurer la mise en page
title = f"Courbe d'équité {symbol + ' ' if symbol else ''}"
title += f"(Rendement: {total_return:.1f}%, DD Max: {max_dd:.1f}%)"
```

```
    fig.update_layout(
        title=title,
        xaxis_title="Date",
        yaxis_title="Équité (%)",
        legend=dict(
            x=0.01,
            y=0.99,
            bordercolor="Black",
            borderwidth=1
        ),
        hovermode="x unified",
        plot_bgcolor='white',
        margin=dict(l=20, r=20, t=50, b=20)
    )

    # Ajouter une grille
    fig.update_xaxes(
        showgrid=True,
        gridcolor='lightgray'
    )
    fig.update_yaxes(
        showgrid=True,
        gridcolor='lightgray'
    )

    return fig

def _create_plotly_prediction_accuracy(self, days: int, symbol: str = None) -> go.Figure:
    """
    Crée un graphique de précision des prédictions interactif avec Plotly
```

```
Args:
    days: Nombre de jours à analyser
    symbol: Filtrer par paire de trading

Returns:
    Figure Plotly
"""
# Horizons à analyser
horizons = ["3h", "12h", "48h", "96h", "short_term", "medium_term", "long_term"]

# Calculer la précision pour chaque horizon
accuracies = []
labels = []
counts = []

for horizon in horizons:
    accuracy = self.get_prediction_accuracy(symbol, horizon, days)

    # Si suffisamment de prédictions
    if accuracy["total_predictions"] > 10:
        accuracies.append(accuracy["accuracy"] * 100)  # En pourcentage
        labels.append(horizon)
        counts.append(accuracy["total_predictions"])

if not accuracies:
    fig = go.Figure()
    fig.add_annotation(
        text="Données insuffisantes",
        xref="paper", yref="paper",
        x=0.5, y=0.5,
```

```
            showarrow=False,
            font=dict(size=20)
        )
        fig.update_layout(title="Précision des prédictions par horizon")
        return fig

    # Créer la figure
    fig = go.Figure()

    # Ajouter le graphique à barres
    fig.add_trace(go.Bar(
        x=labels,
        y=accuracies,
        text=[f"{acc:.1f}%<br>({count})" for acc, count in zip(accuracies, counts)],
        textposition='outside',
        marker_color=['#3498db', '#2980b9', '#1f618d', '#154360', '#512E5F', '#4A235A', '#0B5345'],
        hovertemplate="Horizon: %{x}<br>Précision: %{y:.1f}%<br>Échantillons:
%{text}<extra></extra>"
    ))

    # Ajouter une ligne horizontale à 50% (hasard)
    fig.add_shape(
        type="line",
        x0=-0.5,
        y0=50,
        x1=len(labels) - 0.5,
        y1=50,
        line=dict(
            color="red",
            width=2,
            dash="dash",
```

```
        )
    )

    # Configurer la mise en page
    fig.update_layout(
        title="Précision des prédictions par horizon",
        xaxis_title="Horizon",
        yaxis_title="Précision (%)",
        yaxis_range=[0, 100],
        plot_bgcolor='white',
        margin=dict(l=20, r=20, t=50, b=20)
    )

    # Ajouter une grille
    fig.update_xaxes(
        showgrid=False
    )
    fig.update_yaxes(
        showgrid=True,
        gridcolor='lightgray'
    )

    return fig

def _create_plotly_pnl_distribution(self, days: int, symbol: str = None) -> go.Figure:
    """
    Crée un histogramme interactif de distribution des P&L avec Plotly

    Args:
        days: Nombre de jours à analyser
        symbol: Filtrer par paire de trading
```

```
Returns:
    Figure Plotly
"""
# Calculer la date limite
cutoff_date = (datetime.now() - timedelta(days=days)).isoformat()

# Filtrer les trades récents
recent_trades = [
    t for t in self.trade_history
    if (t.get("timestamp", "") >= cutoff_date or t.get("entry_time", "") >= cutoff_date)
]

# Filtrer par symbole si spécifié
if symbol:
    recent_trades = [t for t in recent_trades if t.get("symbol", "") == symbol]

if not recent_trades:
    fig = go.Figure()
    fig.add_annotation(
        text="Données insuffisantes",
        xref="paper", yref="paper",
        x=0.5, y=0.5,
        showarrow=False,
        font=dict(size=20)
    )
    fig.update_layout(title="Distribution des profits/pertes")
    return fig

# Récupérer les pourcentages de PnL
pnl_values = [t.get("pnl_percent", 0) for t in recent_trades]
```

```
# Créer l'histogramme
fig = go.Figure()

fig.add_trace(go.Histogram(
    x=pnl_values,
    histnorm='',
    marker=dict(
        color='skyblue',
        line=dict(
            color='darkblue',
            width=1
        )
    ),
    hovertemplate="P&L: %{x:.2f}%<br>Fréquence: %{y}<extra></extra>"
))

# Ajouter une ligne verticale à 0
fig.add_shape(
    type="line",
    x0=0,
    y0=0,
    x1=0,
    y1=1,
    yref="paper",
    line=dict(
        color="red",
        width=2,
        dash="dash",
    )
)
```

```
    # Configurer la mise en page
    fig.update_layout(
        title="Distribution des profits/pertes",
        xaxis_title="P&L (%)",
        yaxis_title="Fréquence",
        bargap=0.05,
        plot_bgcolor='white',
        margin=dict(l=20, r=20, t=50, b=20)
    )

    # Ajouter une grille
    fig.update_xaxes(
        showgrid=True,
        gridcolor='lightgray'
    )
    fig.update_yaxes(
        showgrid=True,
        gridcolor='lightgray'
    )

    return fig

def _create_plotly_model_performance(self) -> go.Figure:
    """
    Crée un graphique interactif des métriques de performance du modèle avec Plotly

    Returns:
        Figure Plotly
    """
    # Vérifier s'il y a suffisamment de données
```

```
        if not self.model_performance["timestamps"]:
            fig = go.Figure()
            fig.add_annotation(
                text="Données insuffisantes",
                xref="paper", yref="paper",
                x=0.5, y=0.5,
                showarrow=False,
                font=dict(size=20)
            )
            fig.update_layout(title="Performance du modèle")
            return fig

        # Convertir les timestamps en datetime
        try:
            dates = [datetime.fromisoformat(ts) for ts in self.model_performance["timestamps"]]
        except:
            # Fallback: créer des dates séquentielles
            dates = [datetime.now() - timedelta(days=i) for i in
range(len(self.model_performance["timestamps"]), 0, -1)]

        # Créer la figure
        fig = go.Figure()

        # Ajouter les métriques
        fig.add_trace(go.Scatter(
            x=dates,
            y=self.model_performance["accuracy"],
            mode='lines+markers',
            name='Accuracy',
            line=dict(color='blue', width=2)
        ))
```

```
fig.add_trace(go.Scatter(
    x=dates,
    y=self.model_performance["f1_score"],
    mode='lines+markers',
    name='F1-Score',
    line=dict(color='green', width=2, dash='dash')
))

fig.add_trace(go.Scatter(
    x=dates,
    y=self.model_performance["precision"],
    mode='lines+markers',
    name='Precision',
    line=dict(color='red', width=2, dash='dot')
))

fig.add_trace(go.Scatter(
    x=dates,
    y=self.model_performance["recall"],
    mode='lines+markers',
    name='Recall',
    line=dict(color='purple', width=2, dash='dashdot')
))

# Configurer la mise en page
fig.update_layout(
    title="Performance du modèle",
    xaxis_title="Date",
    yaxis_title="Métrique",
    legend=dict(
```

```
            x=0.01,
            y=0.99,
            bordercolor="Black",
            borderwidth=1
        ),
        hovermode="x unified",
        plot_bgcolor='white',
        margin=dict(l=20, r=20, t=50, b=20)
    )

    # Ajouter une grille
    fig.update_xaxes(
        showgrid=True,
        gridcolor='lightgray'
    )
    fig.update_yaxes(
        showgrid=True,
        gridcolor='lightgray'
    )

    return fig

def _create_plotly_performance_attribution(self) -> go.Figure:
    """
    Crée un graphique interactif d'attribution de performance avec Plotly

    Returns:
        Figure Plotly
    """
    # Vérifier s'il y a suffisamment de données
    if not self.performance_attribution["timestamps"]:
```

```
            fig = go.Figure()
            fig.add_annotation(
                text="Données insuffisantes",
                xref="paper", yref="paper",
                x=0.5, y=0.5,
                showarrow=False,
                font=dict(size=20)
            )
            fig.update_layout(title="Attribution de performance")
            return fig

        # Convertir les timestamps en datetime
        try:
            dates = [datetime.fromisoformat(ts) for ts in self.performance_attribution["timestamps"]]
        except:
            # Fallback: créer des dates séquentielles
            dates = [datetime.now() - timedelta(days=i) for i in
range(len(self.performance_attribution["timestamps"]), 0, -1)]

        # Créer la figure
        fig = go.Figure()

        # Ajouter les traces d'aire empilée
        fig.add_trace(go.Scatter(
            x=dates,
            y=self.performance_attribution["model_contribution"],
            mode='lines',
            name='Modèle LSTM',
            stackgroup='one',
            fillcolor='#3498db',
            line=dict(width=0)
```

```
))

fig.add_trace(go.Scatter(
    x=dates,
    y=self.performance_attribution["technical_contribution"],
    mode='lines',
    name='Indicateurs techniques',
    stackgroup='one',
    fillcolor='#2ecc71',
    line=dict(width=0)
))

fig.add_trace(go.Scatter(
    x=dates,
    y=self.performance_attribution["market_contribution"],
    mode='lines',
    name='Conditions de marché',
    stackgroup='one',
    fillcolor='#f39c12',
    line=dict(width=0)
))

# Configurer la mise en page
fig.update_layout(
    title="Attribution de performance",
    xaxis_title="Date",
    yaxis_title="Contribution",
    legend=dict(
        x=0.01,
        y=0.99,
        bordercolor="Black",
```

```
            borderwidth=1
        ),
        hovermode="x unified",
        plot_bgcolor='white',
        margin=dict(l=20, r=20, t=50, b=20)
    )

    # Ajouter une grille
    fig.update_xaxes(
        showgrid=True,
        gridcolor='lightgray'
    )
    fig.update_yaxes(
        showgrid=True,
        gridcolor='lightgray'
    )

    return fig

def _create_plotly_metrics_table(self, insights: Dict) -> go.Figure:
    """
    Crée un tableau des métriques clés avec Plotly

    Args:
        insights: Insights sur les performances du modèle

    Returns:
        Figure Plotly
    """
    # Extraire les données clés
    prediction_accuracy = insights["prediction_accuracy"]
```

```
        trade_performance = insights["trade_performance"]
        model_metrics = insights["model_metrics"]
        attributions = insights["performance_attribution"]
        trends = insights.get("trends", {})

        # Créer les données du tableau
        headers = ["Métrique", "Valeur", "Tendance"]
        cells = [
            ["Précision de prédiction", f"{prediction_accuracy['accuracy']:.1%}",
self._get_trend_arrow(trends.get("accuracy", {}))],
            ["Win rate", f"{trade_performance['win_rate']:.1%}", ""],
            ["Profit factor", f"{trade_performance['profit_factor']:.2f}", ""],
            ["Drawdown maximum", f"{trade_performance['max_drawdown']:.1f}%", ""],
            ["F1-Score du modèle", f"{model_metrics['f1_score']:.2f}",
self._get_trend_arrow(trends.get("f1_score", {}))],
            ["Contribution du modèle", f"{attributions['model']:.1%}",
self._get_trend_arrow(trends.get("model_contribution", {}))]
        ]

        # Transposer pour le format Plotly
        cell_values = [headers]
        for row in cells:
            cell_values.append(row)

        # Couleurs pour les cellules
        colors = [[None, None, None]]  # En-tête
        for i, row in enumerate(cells):
            if i % 2 == 0:
                colors.append(['white', 'white', 'white'])
            else:
                colors.append(['#f5f5f5', '#f5f5f5', '#f5f5f5'])
```

```
        # Créer la figure
        fig = go.Figure(data=[go.Table(
            header=dict(
                values=headers,
                align=['left', 'center', 'center'],
                fill_color='#3498db',
                font=dict(color='white', size=14)
            ),
            cells=dict(
                values=[
                    [cell[0] for cell in cells],
                    [cell[1] for cell in cells],
                    [cell[2] for cell in cells]
                ],
                align=['left', 'center', 'center'],
                fill_color=[color[0] for color in colors[1:]],
                font=dict(size=12)
            )
        )])

        # Configurer la mise en page
        fig.update_layout(
            title="Métriques clés",
            margin=dict(l=20, r=20, t=50, b=20)
        )

        return fig

def test_monitor():
    """
    Fonction de test pour le monitor
```

```
"""
monitor = ModelMonitor()

# Ajouter des prédictions fictives
for i in range(100):
    date = datetime.now() - timedelta(days=i)

    # Créer une prédiction fictive
    prediction = {
        "3h": {
            "direction": "HAUSSIER" if np.random.random() > 0.5 else "BAISSIER",
            "direction_probability": np.random.uniform(50, 100),
            "predicted_volatility": np.random.uniform(0.01, 0.05),
            "predicted_volume": np.random.uniform(0.8, 1.2),
            "predicted_momentum": np.random.uniform(-0.5, 0.5),
            "confidence": np.random.uniform(0.3, 0.9)
        },
        "12h": {
            "direction": "HAUSSIER" if np.random.random() > 0.5 else "BAISSIER",
            "direction_probability": np.random.uniform(50, 100),
            "predicted_volatility": np.random.uniform(0.01, 0.05),
            "predicted_volume": np.random.uniform(0.8, 1.2),
            "predicted_momentum": np.random.uniform(-0.5, 0.5),
            "confidence": np.random.uniform(0.3, 0.9)
        }
    }

    # Créer des données réelles fictives
    actual_data = {
        "price_change": np.random.uniform(-3, 3)
    }
```

```
    # Enregistrer la prédiction
    monitor.record_prediction("BTCUSDT", prediction, actual_data, date.isoformat())

# Ajouter des trades fictifs
for i in range(50):
    date = datetime.now() - timedelta(days=i)

    # Créer un trade fictif
    trade = {
        "symbol": "BTCUSDT",
        "entry_time": (date - timedelta(hours=6)).isoformat(),
        "exit_time": date.isoformat(),
        "entry_price": 20000 + np.random.uniform(-1000, 1000),
        "exit_price": 20000 + np.random.uniform(-1000, 1000),
        "pnl_percent": np.random.uniform(-10, 15),
        "pnl_absolute": np.random.uniform(-20, 30),
        "side": "BUY" if np.random.random() > 0.5 else "SELL",
        "leverage": 1
    }

    # Enregistrer le trade
    monitor.record_trade(trade)

# Ajouter des métriques de performance fictives
for i in range(10):
    date = datetime.now() - timedelta(days=i*3)

    metrics = {
        "accuracy": np.random.uniform(0.55, 0.75),
        "precision": np.random.uniform(0.5, 0.8),
```

```
        "recall": np.random.uniform(0.5, 0.8),
        "f1_score": np.random.uniform(0.55, 0.75)
    }

    monitor.update_model_performance(metrics, date.isoformat())

# Ajouter des attributions de performance fictives
for i in range(10):
    date = datetime.now() - timedelta(days=i*3)

    # Générer des contributions aléatoires qui somment à 1
    model_contrib = np.random.uniform(0.3, 0.6)
    technical_contrib = np.random.uniform(0.2, 0.4)
    market_contrib = 1 - model_contrib - technical_contrib

    monitor.update_performance_attribution(
        model_contrib,
        technical_contrib,
        market_contrib,
        date.isoformat()
    )

# Générer et sauvegarder un tableau de bord
dashboard_image = monitor.create_performance_dashboard(days=30)

with open("dashboard.png", "wb") as f:
    f.write(dashboard_image.getvalue())

print("Tableau de bord sauvegardé dans dashboard.png")

return monitor
```

```
if __name__ == "__main__":
    test_monitor()
```

```
================================================
File: crypto_trading_bot_CLAUDE/indicators/market_metrics.py
================================================
import pandas as pd
from typing import Dict

def calculate_market_regime(df: pd.DataFrame, lookback: int = 50) -> Dict:
    """
    Détecte le régime de marché actuel (tendance, range, volatil)
    """
    # Calculer la volatilité historique
    volatility = df['close'].rolling(window=lookback).std()

    # Détecter les structures de prix et identifier les niveaux
    regime = {
        'volatility': volatility.iloc[-1],
        'trend': 'undefined'
    }

    return regime
```

```
================================================
File: crypto_trading_bot_CLAUDE/indicators/momentum.py
================================================
# indicators/momentum.py
"""
Indicateurs de momentum
```

```
"""
import pandas as pd
import numpy as np
from typing import Dict, List, Optional, Union

def calculate_rsi(df: pd.DataFrame, period: int = 14) -> pd.Series:
    """
    Calcule le Relative Strength Index (RSI)

    Args:
        df: DataFrame avec les données OHLCV
        period: Période pour le RSI

    Returns:
        Série pandas avec les valeurs du RSI
    """
    if len(df) < period + 1:
        return pd.Series(np.nan, index=df.index)

    # Calculer les variations de prix
    delta = df['close'].diff()

    # Séparer les gains et les pertes
    gain = delta.copy()
    loss = delta.copy()

    gain[gain < 0] = 0
    loss[loss > 0] = 0
    loss = abs(loss)

    # Calculer la moyenne des gains et des pertes
```

```
    avg_gain = gain.rolling(window=period).mean()
    avg_loss = loss.rolling(window=period).mean()

    # Calculer le RS (Relative Strength)
    rs = avg_gain / avg_loss

    # Calculer le RSI
    rsi = 100 - (100 / (1 + rs))

    return rsi

def calculate_stochastic(df: pd.DataFrame, k_period: int = 14, d_period: int = 3) -> Dict[str,
pd.Series]:
    """
    Calcule l'oscillateur stochastique

    Args:
        df: DataFrame avec les données OHLCV
        k_period: Période pour %K
        d_period: Période pour %D

    Returns:
        Dictionnaire avec %K et %D
    """
    if len(df) < k_period:
        empty_series = pd.Series(np.nan, index=df.index)
        return {
            'k': empty_series,
            'd': empty_series
        }
```

```
    # Calculer le plus haut et le plus bas sur la période
    low_min = df['low'].rolling(window=k_period).min()
    high_max = df['high'].rolling(window=k_period).max()

    # Calculer %K
    k = 100 * ((df['close'] - low_min) / (high_max - low_min))

    # Calculer %D (moyenne mobile simple de %K)
    d = k.rolling(window=d_period).mean()

    return {
        'k': k,
        'd': d
    }

def detect_divergence(price_df: pd.DataFrame, indicator: pd.Series,
                      lookback: int = 15, threshold: float = 0.02,
                      min_peak_distance: int = 3) -> Dict:
    """
    Détecte les divergences entre le prix et un indicateur

    Args:
        price_df: DataFrame avec les données de prix
        indicator: Série de l'indicateur (RSI, MACD, etc.)
        lookback: Nombre de périodes à analyser
        threshold: Seuil pour déterminer les sommets/creux significatifs

    Returns:
        Dictionnaire avec les divergences détectées
    """
    if len(price_df) < lookback or indicator.isna().all():
```

```
    return {
        'bullish': False,
        'bearish': False,
        'details': {
            'message': 'Données insuffisantes'
        }
    }

# Récupérer les données récentes
recent_price = price_df['close'].iloc[-lookback:].values
recent_indicator = indicator.iloc[-lookback:].values

# Fonction pour trouver les sommets et creux
def find_peaks(data):
    peaks = []
    valleys = []

    for i in range(1, len(data) - 1):
        if data[i] > data[i-1] and data[i] > data[i+1]:
            if i > 0 and i < len(data) - 1:
                peaks.append(i)
        elif data[i] < data[i-1] and data[i] < data[i+1]:
            if i > 0 and i < len(data) - 1:
                valleys.append(i)

    return peaks, valleys

# Trouver les sommets et creux
price_peaks, price_valleys = find_peaks(recent_price)
indicator_peaks, indicator_valleys = find_peaks(recent_indicator)
```

```
# Filtrer les sommets/creux non significatifs
def is_significant(data, peaks, threshold):
    significant = []
    for p in peaks:
        max_nearby = max(data[max(0, p-2):min(len(data), p+3)])
        if data[p] > max_nearby * (1 - threshold):
            significant.append(p)
    return significant

price_peaks = is_significant(recent_price, price_peaks, threshold)
price_valleys = is_significant(recent_price, price_valleys, threshold)
indicator_peaks = is_significant(recent_indicator, indicator_peaks, threshold)
indicator_valleys = is_significant(recent_indicator, indicator_valleys, threshold)

# Vérifier les divergences
bullish_divergence = False
bearish_divergence = False
details = {}

# Divergence haussière: prix fait un creux plus bas, indicateur fait un creux plus haut
if len(price_valleys) >= 2 and len(indicator_valleys) >= 2:
    if recent_price[price_valleys[-1]] < recent_price[price_valleys[-2]] and \
       recent_indicator[indicator_valleys[-1]] > recent_indicator[indicator_valleys[-2]]:
        bullish_divergence = True
        details['bullish'] = {
            'price_valley1': price_valleys[-2],
            'price_valley2': price_valleys[-1],
            'indicator_valley1': indicator_valleys[-2],
            'indicator_valley2': indicator_valleys[-1]
        }
```

```
    # Divergence baissière: prix fait un sommet plus haut, indicateur fait un sommet plus bas
    if len(price_peaks) >= 2 and len(indicator_peaks) >= 2:
        if recent_price[price_peaks[-1]] > recent_price[price_peaks[-2]] and \
           recent_indicator[indicator_peaks[-1]] < recent_indicator[indicator_peaks[-2]]:
            bearish_divergence = True
            details['bearish'] = {
                'price_peak1': price_peaks[-2],
                'price_peak2': price_peaks[-1],
                'indicator_peak1': indicator_peaks[-2],
                'indicator_peak2': indicator_peaks[-1]
            }

    return {
        'bullish': bullish_divergence,
        'bearish': bearish_divergence,
        'details': details
    }



================================================
File: crypto_trading_bot_CLAUDE/indicators/trend.py
================================================
# indicators/trend.py
"""
Indicateurs de tendance
"""
import pandas as pd
import numpy as np
from typing import Dict, List, Optional, Union

def calculate_ema(df: pd.DataFrame, periods: List[int] = [9, 21, 50, 200]) -> Dict[str, pd.Series]:
```

```
    """
    Calcule les moyennes mobiles exponentielles (EMA)

    Args:
        df: DataFrame avec les données OHLCV
        periods: Périodes pour les EMA

    Returns:
        Dictionnaire des EMA calculées
    """
    result = {}

    for period in periods:
        if len(df) >= period:
            result[f'ema_{period}'] = df['close'].ewm(span=period, adjust=False).mean()
        else:
            # Si les données sont insuffisantes, créer une série avec des NaN
            result[f'ema_{period}'] = pd.Series(np.nan, index=df.index)

    return result

def calculate_adx(df: pd.DataFrame, period: int = 14) -> Dict[str, pd.Series]:
    """
    Calcule l'Average Directional Index (ADX) - Version optimisée

    Args:
        df: DataFrame avec les données OHLCV
        period: Période pour le calcul de l'ADX

    Returns:
        Dictionnaire avec ADX, +DI et -DI
```

```
"""
if len(df) < period + 1:
    # Données insuffisantes
    empty_series = pd.Series(np.nan, index=df.index)
    return {
        'adx': empty_series,
        'plus_di': empty_series,
        'minus_di': empty_series
    }

# Créer une copie pour éviter de modifier l'original
df = df.copy()

# Calcul des True Range (TR)
df['high_low'] = df['high'] - df['low']
df['high_close'] = abs(df['high'] - df['close'].shift(1))
df['low_close'] = abs(df['low'] - df['close'].shift(1))
df['tr'] = df[['high_low', 'high_close', 'low_close']].max(axis=1)

# Utiliser EWM au lieu de rolling pour plus d'efficacité
df['atr'] = df['tr'].ewm(alpha=1/period, min_periods=period).mean()

# Calcul des mouvements directionnels
df['up_move'] = df['high'].diff()
df['down_move'] = -df['low'].diff()

# Calculer +DM et -DM
df['plus_dm'] = np.where(
    (df['up_move'] > df['down_move']) & (df['up_move'] > 0),
    df['up_move'],
    0
```

```
    )

    df['minus_dm'] = np.where(
        (df['down_move'] > df['up_move']) & (df['down_move'] > 0),
        df['down_move'],
        0
    )

    # Utiliser EWM pour les smoothed DM
    df['plus_dm_smoothed'] = df['plus_dm'].ewm(alpha=1/period, min_periods=period).mean()
    df['minus_dm_smoothed'] = df['minus_dm'].ewm(alpha=1/period, min_periods=period).mean()

    # Calculer +DI et -DI
    df['plus_di'] = 100 * df['plus_dm_smoothed'] / df['atr']
    df['minus_di'] = 100 * df['minus_dm_smoothed'] / df['atr']

    # Calculer DX
    df['dx'] = 100 * abs(df['plus_di'] - df['minus_di']) / (df['plus_di'] + df['minus_di']).replace(0, np.nan)

    # Calculer ADX
    df['adx'] = df['dx'].ewm(alpha=1/period, min_periods=period).mean()

    return {
        'adx': df['adx'],
        'plus_di': df['plus_di'],
        'minus_di': df['minus_di']
    }
def calculate_macd(df: pd.DataFrame, fast_period: int = 12, slow_period: int = 26,
                   signal_period: int = 9) -> Dict[str, pd.Series]:
    """
    Calcule le MACD (Moving Average Convergence Divergence)
```

```
Args:
    df: DataFrame avec les données OHLCV
    fast_period: Période pour l'EMA rapide
    slow_period: Période pour l'EMA lente
    signal_period: Période pour la ligne de signal

Returns:
    Dictionnaire avec MACD, signal et histogramme
"""
if len(df) < slow_period:
    # Données insuffisantes
    empty_series = pd.Series(np.nan, index=df.index)
    return {
        'macd': empty_series,
        'signal': empty_series,
        'histogram': empty_series
    }

# Calcul des EMA
fast_ema = df['close'].ewm(span=fast_period, adjust=False).mean()
slow_ema = df['close'].ewm(span=slow_period, adjust=False).mean()

# Calcul du MACD
macd = fast_ema - slow_ema

# Calcul de la ligne de signal
signal = macd.ewm(span=signal_period, adjust=False).mean()

# Calcul de l'histogramme
histogram = macd - signal
```

```
    return {
        'macd': macd,
        'signal': signal,
        'histogram': histogram
    }

def detect_trend(df: pd.DataFrame, ema_periods: List[int] = [9, 21, 50]) -> Dict:
    """
    Détecte la tendance à partir des EMA

    Args:
        df: DataFrame avec les données OHLCV
        ema_periods: Périodes pour les EMA

    Returns:
        Dictionnaire avec la tendance détectée
    """
    # Calculer les EMA
    emas = calculate_ema(df, ema_periods)

    # S'assurer que toutes les EMA sont disponibles
    if any(emas[f'ema_{p}'].isna().all() for p in ema_periods):
        return {
            'trend': 'unknown',
            'strength': 0,
            'details': {
                'message': 'EMA non disponibles'
            }
        }
```

```
# Récupérer les dernières valeurs
current_price = df['close'].iloc[-1]
ema_values = {p: emas[f'ema_{p}'].iloc[-1] for p in ema_periods}

# Détection de la tendance
ema_short = ema_values[ema_periods[0]]
ema_medium = ema_values[ema_periods[1]]
ema_long = ema_values[ema_periods[2]]

# Calcul de la force de la tendance en fonction de l'alignement des EMA
trend_strength = 0
trend = 'neutral'
details = {}

# Vérifier l'alignement haussier (EMA courte > EMA moyenne > EMA longue)
if ema_short > ema_medium > ema_long:
    trend = 'bullish'

    # Évaluer la force de la tendance
    price_vs_ema_short = (current_price / ema_short - 1) * 100
    ema_short_vs_medium = (ema_short / ema_medium - 1) * 100
    ema_medium_vs_long = (ema_medium / ema_long - 1) * 100

    trend_strength = (price_vs_ema_short + ema_short_vs_medium + ema_medium_vs_long) / 3
    if trend_strength > 2:
        trend_strength = 1.0  # Fort
    elif trend_strength > 1:
        trend_strength = 0.7  # Modéré
    else:
        trend_strength = 0.3  # Faible
```

```
    details = {
        'price_vs_ema_short': price_vs_ema_short,
        'ema_short_vs_medium': ema_short_vs_medium,
        'ema_medium_vs_long': ema_medium_vs_long
    }

# Vérifier l'alignement baissier (EMA courte < EMA moyenne < EMA longue)
elif ema_short < ema_medium < ema_long:
    trend = 'bearish'

    # Évaluer la force de la tendance
    price_vs_ema_short = (ema_short / current_price - 1) * 100
    ema_short_vs_medium = (ema_medium / ema_short - 1) * 100
    ema_medium_vs_long = (ema_long / ema_medium - 1) * 100

    trend_strength = (price_vs_ema_short + ema_short_vs_medium + ema_medium_vs_long) / 3
    if trend_strength > 2:
        trend_strength = 1.0  # Fort
    elif trend_strength > 1:
        trend_strength = 0.7  # Modéré
    else:
        trend_strength = 0.3  # Faible

    details = {
        'price_vs_ema_short': price_vs_ema_short,
        'ema_short_vs_medium': ema_short_vs_medium,
        'ema_medium_vs_long': ema_medium_vs_long
    }

# Tendance neutre ou en transition
else:
```

```
        # Vérifier le croisement des EMA
        if ema_short > ema_medium and ema_medium < ema_long:
            trend = 'potentially_bullish'  # Possible*
            trend = 'potentially_bullish'  # Possible renversement haussier
            trend_strength = 0.2
        elif ema_short < ema_medium and ema_medium > ema_long:
            trend = 'potentially_bearish'  # Possible renversement baissier
            trend_strength = 0.2
        else:
            if current_price > ema_long:
                trend = 'weak_bullish'
                trend_strength = 0.1
            elif current_price < ema_long:
                trend = 'weak_bearish'
                trend_strength = 0.1
            else:
                trend = 'neutral'
                trend_strength = 0

    return {
        'trend': trend,
        'strength': trend_strength,
        'details': details
    }



================================================
File: crypto_trading_bot_CLAUDE/indicators/volatility.py
================================================
# indicators/volatility.py
"""
```

```
Indicateurs de volatilité
"""
import pandas as pd
import numpy as np
from typing import Dict, List, Optional, Union

def calculate_bollinger_bands(df: pd.DataFrame, period: int = 20,
                              std_dev: float = 2.0) -> Dict[str, pd.Series]:
    """
    Calcule les bandes de Bollinger

    Args:
        df: DataFrame avec les données OHLCV
        period: Période pour la moyenne mobile
        std_dev: Nombre d'écarts-types pour les bandes

    Returns:
        Dictionnaire avec les bandes supérieure, moyenne et inférieure
    """
    if len(df) < period:
        empty_series = pd.Series(np.nan, index=df.index)
        return {
            'upper': empty_series,
            'middle': empty_series,
            'lower': empty_series,
            'bandwidth': empty_series,
            'percent_b': empty_series
        }

    # Calculer la moyenne mobile
    middle = df['close'].rolling(window=period).mean()
```

```
    # Calculer l'écart-type
    rolling_std = df['close'].rolling(window=period).std()

    # Calculer les bandes supérieure et inférieure
    upper = middle + (rolling_std * std_dev)
    lower = middle - (rolling_std * std_dev)

    # Calculer la largeur des bandes (bandwidth)
    bandwidth = (upper - lower) / middle

    # Calculer %B (position du prix dans les bandes)
    percent_b = (df['close'] - lower) / (upper - lower)

    return {
        'upper': upper,
        'middle': middle,
        'lower': lower,
        'bandwidth': bandwidth,
        'percent_b': percent_b
    }

def calculate_atr(df: pd.DataFrame, period: int = 14) -> pd.Series:
    """
    Calcule l'Average True Range (ATR)

    Args:
        df: DataFrame avec les données OHLCV
        period: Période pour l'ATR

    Returns:
```

```
    Série pandas avec les valeurs de l'ATR
    """
    if len(df) < period + 1:
        return pd.Series(np.nan, index=df.index)

    # Calculer le True Range
    high_low = df['high'] - df['low']
    high_close = np.abs(df['high'] - df['close'].shift())
    low_close = np.abs(df['low'] - df['close'].shift())

    tr = pd.concat([high_low, high_close, low_close], axis=1).max(axis=1)

    # Calculer l'ATR (moyenne mobile exponentielle du TR)
    atr = tr.ewm(alpha=1/period, adjust=False).mean()

    return atr

def detect_volatility_squeeze(df: pd.DataFrame, bb_period: int = 20,
                              kc_period: int = 20, kc_mult: float = 1.5) -> Dict:
    """
    Détecte le 'squeeze' (compression de la volatilité) à l'aide des bandes de Bollinger et du canal de Keltner

    Args:
        df: DataFrame avec les données OHLCV
        bb_period: Période pour les bandes de Bollinger
        kc_period: Période pour le canal de Keltner
        kc_mult: Multiplicateur pour le canal de Keltner

    Returns:
        Dictionnaire avec la détection du squeeze
```

```
"""
if len(df) < max(bb_period, kc_period) + 1:
    return {
        'squeeze': False,
        'strength': 0,
        'details': {
            'message': 'Données insuffisantes'
        }
    }

# Calculer les bandes de Bollinger
bb = calculate_bollinger_bands(df, period=bb_period)

# Calculer l'ATR pour le canal de Keltner
atr = calculate_atr(df, period=kc_period)

# Calculer le canal de Keltner
kc_middle = df['close'].rolling(window=kc_period).mean()
kc_upper = kc_middle + (atr * kc_mult)
kc_lower = kc_middle - (atr * kc_mult)

# Détecter le squeeze (les bandes de Bollinger à l'intérieur du canal de Keltner)
squeeze = (bb['lower'] > kc_lower) & (bb['upper'] < kc_upper)

# Calculer la force du squeeze (compression)
# Plus la valeur est faible, plus la compression est forte
compression_ratio = (bb['upper'] - bb['lower']) / (kc_upper - kc_lower)

# Convertir le ratio en force (0-1, où 1 est le squeeze le plus fort)
strength = 1 - compression_ratio.fillna(1)
strength = strength.clip(0, 1)
```

```
    # Récupérer les valeurs récentes
    current_squeeze = squeeze.iloc[-1]
    current_strength = strength.iloc[-1]

    # Vérifier combien de temps le squeeze dure
    if current_squeeze:
        squeeze_duration = squeeze.iloc[-20:].sum()
    else:
        squeeze_duration = 0

    return {
        'squeeze': bool(current_squeeze),
        'strength': float(current_strength),
        'duration': int(squeeze_duration),
        'details': {
            'bb_width': float(bb['upper'].iloc[-1] - bb['lower'].iloc[-1]),
            'kc_width': float(kc_upper.iloc[-1] - kc_lower.iloc[-1]),
            'compression_ratio': float(compression_ratio.iloc[-1]),
            'historical_squeezes': squeeze.iloc[-50:].sum()
        }
    }

================================================
File: crypto_trading_bot_CLAUDE/indicators/volume.py
================================================
# indicators/volume.py
"""
Indicateurs de volume
"""
import pandas as pd
```

```
import numpy as np
from typing import Dict, List, Optional, Union

def calculate_obv(df: pd.DataFrame) -> pd.Series:
    """
    Calcule l'On-Balance Volume (OBV)

    Args:
        df: DataFrame avec les données OHLCV

    Returns:
        Série pandas avec les valeurs de l'OBV
    """
    if df.empty:
        return pd.Series(dtype=float)

    obv = pd.Series(index=df.index, dtype=float)
    obv.iloc[0] = 0

    for i in range(1, len(df)):
        if df['close'].iloc[i] > df['close'].iloc[i-1]:
            obv.iloc[i] = obv.iloc[i-1] + df['volume'].iloc[i]
        elif df['close'].iloc[i] < df['close'].iloc[i-1]:
            obv.iloc[i] = obv.iloc[i-1] - df['volume'].iloc[i]
        else:
            obv.iloc[i] = obv.iloc[i-1]

    return obv

def calculate_vwap(df: pd.DataFrame) -> pd.Series:
    """
```

```
    Calcule le Volume-Weighted Average Price (VWAP)

    Args:
        df: DataFrame avec les données OHLCV

    Returns:
        Série pandas avec les valeurs du VWAP
    """
    if df.empty:
        return pd.Series(dtype=float)

    # Calculer le prix typique
    df = df.copy()
    df['typical_price'] = (df['high'] + df['low'] + df['close']) / 3

    # Calculer le produit du prix typique et du volume
    df['tp_volume'] = df['typical_price'] * df['volume']

    # Calculer les sommes cumulatives
    df['cum_tp_volume'] = df['tp_volume'].cumsum()
    df['cum_volume'] = df['volume'].cumsum()

    # Calculer le VWAP
    vwap = df['cum_tp_volume'] / df['cum_volume']

    return vwap

def detect_volume_spike(df: pd.DataFrame, periods: int = 14,
                        threshold: float = 2.0) -> Dict:
    """
    Détecte les pics de volume
```

```
Args:
    df: DataFrame avec les données OHLCV
    periods: Nombre de périodes pour la moyenne
    threshold: Seuil pour détecter un pic (multiplicateur de la moyenne)

Returns:
    Dictionnaire avec la détection de pic de volume
"""
if len(df) < periods + 1:
    return {
        'spike': False,
        'ratio': 0,
        'details': {
            'message': 'Données insuffisantes'
        }
    }

# Calculer la moyenne mobile du volume
volume_ma = df['volume'].rolling(window=periods).mean()

# Calculer le ratio du volume actuel par rapport à la moyenne
current_volume = df['volume'].iloc[-1]
current_volume_ma = volume_ma.iloc[-1]

volume_ratio = current_volume / current_volume_ma if current_volume_ma > 0 else 0

# Détecter un pic de volume
is_spike = volume_ratio > threshold

# Vérifier si le pic est associé à une hausse ou à une baisse
```

```
    price_change = df['close'].iloc[-1] - df['open'].iloc[-1]
    is_bullish = price_change > 0

    return {
        'spike': bool(is_spike),
        'ratio': float(volume_ratio),
        'threshold': float(threshold),
        'bullish': bool(is_bullish) if is_spike else None,
        'details': {
            'current_volume': float(current_volume),
            'average_volume': float(current_volume_ma),
            'price_change': float(price_change),
            'price_change_percent': float(price_change / df['open'].iloc[-1] * 100) if df['open'].iloc[-1] > 0
else 0
        }
    }

def detect_volume_climax(df: pd.DataFrame, periods: int = 14,
                         threshold: float = 3.0) -> Dict:
    """
    Détecte un climax de volume (volume très élevé associé à une forte variation de prix)

    Args:
        df: DataFrame avec les données OHLCV
        periods: Nombre de périodes pour la moyenne
        threshold: Seuil pour détecter un climax (multiplicateur de la moyenne)

    Returns:
        Dictionnaire avec la détection de climax de volume
    """
    if len(df) < periods + 1:
```

```
    return {
        'climax': False,
        'type': None,
        'details': {
            'message': 'Données insuffisantes'
        }
    }

# Vérifier si c'est un pic de volume
spike_result = detect_volume_spike(df, periods, threshold)

if not spike_result['spike']:
    return {
        'climax': False,
        'type': None,
        'details': {
            'message': 'Pas de pic de volume',
            'spike_ratio': spike_result['ratio']
        }
    }

# Calculer la taille de la bougie
body_size = abs(df['close'].iloc[-1] - df['open'].iloc[-1])
body_size_percent = body_size / df['open'].iloc[-1] * 100 if df['open'].iloc[-1] > 0 else 0

# Calculer la moyenne des tailles de bougie
body_sizes = abs(df['close'] - df['open'])
avg_body_size_percent = (body_sizes / df['open']) * 100
avg_body_size_percent = avg_body_size_percent.replace([np.inf, -np.inf], np.nan).dropna()

avg_body_size_percent = avg_body_size_percent.rolling(window=periods).mean()
```

```
    current_avg_body_size_percent = avg_body_size_percent.iloc[-1] if not
avg_body_size_percent.empty else 0

    # Vérifier si la taille de la bougie est significative
    body_ratio = body_size_percent / current_avg_body_size_percent if
current_avg_body_size_percent > 0 else 0
    significant_body = body_ratio > 1.5

    # Déterminer le type de climax
    is_bullish = df['close'].iloc[-1] > df['open'].iloc[-1]

    if significant_body and is_bullish:
        climax_type = 'buying_climax'
    elif significant_body and not is_bullish:
        climax_type = 'selling_climax'
    else:
        climax_type = 'volume_climax'

    return {
        'climax': bool(significant_body),
        'type': climax_type,
        'details': {
            'spike_ratio': float(spike_result['ratio']),
            'body_size_percent': float(body_size_percent),
            'avg_body_size_percent': float(current_avg_body_size_percent),
            'body_ratio': float(body_ratio),
            'is_bullish': bool(is_bullish)
        }
    }

def detect_volume_divergence(df: pd.DataFrame, periods: int = 14) -> Dict:
    """
```

```
Détecte les divergences entre le prix et le volume

Args:
    df: DataFrame avec les données OHLCV
    periods: Nombre de périodes à analyser

Returns:
    Dictionnaire avec la détection de divergence
"""
if len(df) < periods + 1:
    return {
        'divergence': False,
        'type': None,
        'details': {
            'message': 'Données insuffisantes'
        }
    }

# Récupérer les données récentes
recent_df = df.iloc[-periods:].copy()

# Calculer les variations en pourcentage
recent_df['price_change'] = recent_df['close'].pct_change()
recent_df['volume_change'] = recent_df['volume'].pct_change()

# Ignorer la première ligne (NaN)
recent_df = recent_df.iloc[1:]

# Compter les cas où les variations de prix et de volume vont dans des directions opposées
opposite_directions = ((recent_df['price_change'] > 0) & (recent_df['volume_change'] < 0)) | \
                      ((recent_df['price_change'] < 0) & (recent_df['volume_change'] > 0))
```

```
divergence_count = opposite_directions.sum()
divergence_percent = divergence_count / len(recent_df) * 100

# Vérifier les 5 dernières périodes
recent_opposite = opposite_directions.iloc[-5:].sum()
recent_percent = recent_opposite / 5 * 100

# Détecter une divergence significative
significant_divergence = recent_percent > 60

# Déterminer le type de divergence
recent_price_trend = recent_df['close'].iloc[-1] > recent_df['close'].iloc[0]
recent_volume_trend = recent_df['volume'].iloc[-1] > recent_df['volume'].iloc[0]

if recent_price_trend and not recent_volume_trend:
    divergence_type = 'price_up_volume_down'
elif not recent_price_trend and recent_volume_trend:
    divergence_type = 'price_down_volume_up'
else:
    divergence_type = None

return {
    'divergence': bool(significant_divergence),
    'type': divergence_type,
    'details': {
        'divergence_count': int(divergence_count),
        'divergence_percent': float(divergence_percent),
        'recent_divergence_percent': float(recent_percent),
        'price_trend': 'up' if recent_price_trend else 'down',
        'volume_trend': 'up' if recent_volume_trend else 'down'
```

```
    }
  }
```

```
================================================
File: crypto_trading_bot_CLAUDE/strategies/hybrid_strategy.py
================================================
# strategies/hybrid_strategy.py
"""
Stratégie hybride combinant l'analyse technique classique avec des prédictions LSTM
"""
import os
import pandas as pd
import numpy as np
from typing import Dict, List, Optional, Union
from datetime import datetime, timedelta

from strategies.strategy_base import StrategyBase
from strategies.technical_bounce import TechnicalBounceStrategy
from ai.models.lstm_model import LSTMModel
from ai.models.feature_engineering import FeatureEngineering
from core.adaptive_risk_manager import AdaptiveRiskManager
from config.config import DATA_DIR
from utils.logger import setup_logger

logger = setup_logger("hybrid_strategy")

class HybridStrategy(StrategyBase):
  """
  Stratégie hybride qui combine:
  1. Détection classique de rebond technique
  2. Prédictions du modèle LSTM pour la direction, volatilité et momentum
```

```
    3. Gestion adaptative du risque
    """
    def __init__(self, data_fetcher, market_analyzer, scoring_engine,
                 lstm_model: Optional[LSTMModel] = None,
                 adaptive_risk_manager: Optional[AdaptiveRiskManager] = None):
        """
        Initialise la stratégie hybride

        Args:
            data_fetcher: Module de récupération des données
            market_analyzer: Analyseur d'état du marché
            scoring_engine: Moteur de scoring
            lstm_model: Modèle LSTM (chargé automatiquement si None)
            adaptive_risk_manager: Gestionnaire de risque adaptatif
        """
        super().__init__(data_fetcher, market_analyzer, scoring_engine)

        # Composants spécifiques à la stratégie hybride
        self.technical_strategy = TechnicalBounceStrategy(data_fetcher, market_analyzer,
scoring_engine)
        self.feature_engineering = FeatureEngineering()
        self.adaptive_risk_manager = adaptive_risk_manager or AdaptiveRiskManager()

        # Chargement du modèle LSTM
        self.lstm_model = lstm_model
        if self.lstm_model is None:
            self._load_lstm_model()

        # Paramètres de combinaison des signaux
        self.lstm_weight = 0.6  # Poids des prédictions LSTM
        self.technical_weight = 0.4  # Poids des signaux techniques
```

```
        # Seuil minimum de score combiné pour trader
        self.min_score = 75

        # Cache des prédictions LSTM (pour éviter de recalculer)
        self.lstm_predictions_cache = {}
        self.cache_duration = 300  # 5 minutes de durée de cache

    def _load_lstm_model(self) -> None:
        """
        Charge le modèle LSTM depuis le disque
        """
        try:
            model_path = os.path.join(DATA_DIR, "models", "production", "lstm_final.h5")
            if not os.path.exists(model_path):
                logger.warning(f"Modèle LSTM non trouvé: {model_path}")
                return

            self.lstm_model = LSTMModel()
            self.lstm_model.load(model_path)
            logger.info(f"Modèle LSTM chargé: {model_path}")
        except Exception as e:
            logger.error(f"Erreur lors du chargement du modèle LSTM: {str(e)}")

    def find_trading_opportunity(self, symbol: str) -> Optional[Dict]:
        """
        Cherche une opportunité de trading en combinant les signaux techniques et les prédictions LSTM

        Args:
            symbol: Paire de trading
```

```
Returns:
    Opportunité de trading ou None si aucune opportunité
"""
# 1. Rechercher une opportunité selon la stratégie technique classique
technical_opportunity = self.technical_strategy.find_trading_opportunity(symbol)

# Si aucune opportunité technique, pas besoin d'aller plus loin
if not technical_opportunity:
    return None

# 2. Récupérer les données de marché
market_data = self.data_fetcher.get_market_data(symbol)

# 3. Obtenir les prédictions du modèle LSTM
lstm_prediction = self._get_lstm_prediction(symbol, market_data)

# Si aucune prédiction LSTM disponible, utiliser uniquement la stratégie technique
if not lstm_prediction:
    logger.warning(f"Aucune prédiction LSTM disponible pour {symbol}")

    # Si le score technique est très élevé, on peut quand même trader
    if technical_opportunity["score"] >= 85:
        return technical_opportunity

    return None

# 4. Combiner les signaux pour une décision finale
combined_opportunity = self._combine_signals(
    symbol,
    technical_opportunity,
```

```
            lstm_prediction,
            market_data
        )

        return combined_opportunity

    def _get_lstm_prediction(self, symbol: str, market_data: Dict) -> Optional[Dict]:
        """
        Obtient les prédictions du modèle LSTM pour le symbole donné

        Args:
            symbol: Paire de trading
            market_data: Données de marché

        Returns:
            Prédictions LSTM ou None si indisponibles
        """
        # Vérifier si le modèle LSTM est disponible
        if self.lstm_model is None:
            return None

        # Vérifier si des prédictions récentes sont en cache
        cache_key = f"{symbol}_{datetime.now().strftime('%Y%m%d_%H%M')}"
        if cache_key in self.lstm_predictions_cache:
            cached_prediction = self.lstm_predictions_cache[cache_key]
            if (datetime.now() - cached_prediction["timestamp"]).total_seconds() < self.cache_duration:
                return cached_prediction["data"]

        try:
            # Récupérer les données OHLCV
            ohlcv_data = market_data["primary_timeframe"]["ohlcv"]
```

```
        # Créer les caractéristiques avancées
        featured_data = self.feature_engineering.create_features(
            ohlcv_data,
            include_time_features=True,
            include_price_patterns=True
        )

        # Normaliser les caractéristiques
        normalized_data = self.feature_engineering.scale_features(
            featured_data,
            is_training=False,
            method='standard',
            feature_group='lstm'
        )

        # Créer une séquence pour la prédiction
        sequence_length = self.lstm_model.input_length

        # Vérifier si nous avons assez de données
        if len(normalized_data) < sequence_length:
            logger.warning(f"Données insuffisantes pour la prédiction LSTM ({len(normalized_data)} < {sequence_length})")
            return None

        # Obtenir la dernière séquence
        X = self.feature_engineering.prepare_lstm_data(
            normalized_data,
            sequence_length=sequence_length,
            is_training=False
        )
```

```
        # Faire la prédiction
        prediction = self.lstm_model.predict(normalized_data)

        # Stocker en cache
        self.lstm_predictions_cache[cache_key] = {
            "data": prediction,
            "timestamp": datetime.now()
        }

        # Nettoyer le cache (garder seulement les 10 prédictions les plus récentes)
        if len(self.lstm_predictions_cache) > 10:
            oldest_key = min(self.lstm_predictions_cache.keys(),
                             key=lambda k: self.lstm_predictions_cache[k]["timestamp"])
            del self.lstm_predictions_cache[oldest_key]

        return prediction

    except Exception as e:
        logger.error(f"Erreur lors de la prédiction LSTM: {str(e)}")
        return None

def _combine_signals(self, symbol: str, technical_opportunity: Dict,
                     lstm_prediction: Dict, market_data: Dict) -> Optional[Dict]:
    """
    Combine les signaux techniques et les prédictions LSTM

    Args:
        symbol: Paire de trading
        technical_opportunity: Opportunité de la stratégie technique
        lstm_prediction: Prédictions du modèle LSTM
```

```
    market_data: Données de marché

Returns:
    Opportunité combinée ou None si pas d'opportunité
"""
# Extraire les informations pertinentes
technical_score = technical_opportunity["score"]
technical_side = technical_opportunity["side"]

# Vérifier si les prédictions LSTM sont cohérentes avec la stratégie technique
lstm_confidence = self._calculate_lstm_confidence(lstm_prediction, technical_side)

# Calculer le score combiné
combined_score = (technical_score * self.technical_weight +
                  lstm_confidence["score"] * self.lstm_weight)

# Si le score combiné est trop faible, pas d'opportunité
if combined_score < self.min_score:
    return None

# Fusionner les informations en une seule opportunité
combined_opportunity = technical_opportunity.copy()
combined_opportunity["score"] = combined_score
combined_opportunity["lstm_prediction"] = lstm_prediction
combined_opportunity["lstm_confidence"] = lstm_confidence

# Ajuster les niveaux de stop-loss et take-profit en fonction des prédictions
entry_price = technical_opportunity["entry_price"]

# Utiliser le gestionnaire de risque adaptatif pour calculer les niveaux optimaux
exit_levels = self.adaptive_risk_manager.calculate_optimal_exit_levels(
```

```
            entry_price,
            technical_side,
            technical_opportunity,
            lstm_prediction
        )

        combined_opportunity["stop_loss"] = exit_levels["stop_loss_price"]
        combined_opportunity["take_profit"] = exit_levels["take_profit_price"]
        combined_opportunity["stop_loss_percent"] = exit_levels["stop_loss_percent"]
        combined_opportunity["take_profit_percent"] = exit_levels["take_profit_percent"]

        # Ajouter des informations explicatives
        combined_opportunity["reasoning"] = self._generate_reasoning(
            technical_opportunity,
            lstm_prediction,
            lstm_confidence,
            combined_score
        )

        return combined_opportunity

    def _calculate_lstm_confidence(self, lstm_prediction: Dict, technical_side: str) -> Dict:
        """
        Calcule la confiance dans les prédictions LSTM et leur alignement avec la stratégie technique

        Args:
            lstm_prediction: Prédictions du modèle LSTM
            technical_side: Direction de la stratégie technique ('BUY'/'SELL')

        Returns:
            Dictionnaire avec le score de confiance et les détails
```

```
"""
# Initialiser le score de confiance
confidence_score = 50  # Score neutre par défaut

# Vérifier l'alignement de la direction
direction_alignment = 0
direction_confidence = 0

# Pour chaque horizon, vérifier la direction prédite
for horizon, prediction in lstm_prediction.items():
    # Pondération selon l'horizon (plus de poids au court terme)
    if "horizon_12" in horizon:  # Court terme
        weight = 0.6
    elif "horizon_24" in horizon:  # Moyen terme
        weight = 0.3
    else:  # Long terme
        weight = 0.1

    direction_prob = prediction.get("direction_probability", 0.5)

    # Convertir la probabilité en score (0-100)
    # Pour BUY: direction_prob > 0.5 est favorable
    # Pour SELL: direction_prob < 0.5 est favorable
    if technical_side == "BUY":
        horizon_score = (direction_prob - 0.5) * 200 * weight  # -100 à +100, pondéré
    else:
        horizon_score = (0.5 - direction_prob) * 200 * weight  # -100 à +100, pondéré

    direction_alignment += horizon_score

    # Calculer la confiance dans la direction (indépendamment de l'alignement)
```

```
            confidence = abs(direction_prob - 0.5) * 2 * 100 * weight  # 0 à 100, pondéré
            direction_confidence += confidence

    # Ajuster le score en fonction de l'alignement de direction
    confidence_score += direction_alignment

    # Vérifier le momentum
    momentum_alignment = 0

    for horizon, prediction in lstm_prediction.items():
        # Pondération selon l'horizon
        if "horizon_12" in horizon:  # Court terme
            weight = 0.6
        elif "horizon_24" in horizon:  # Moyen terme
            weight = 0.3
        else:  # Long terme
            weight = 0.1

        momentum = prediction.get("predicted_momentum", 0)

        # Convertir le momentum en score (0-100)
        # Pour BUY: momentum > 0 est favorable
        # Pour SELL: momentum < 0 est favorable
        if technical_side == "BUY":
            horizon_score = momentum * 100 * weight  # -100 à +100, pondéré
        else:
            horizon_score = -momentum * 100 * weight  # -100 à +100, pondéré

        momentum_alignment += horizon_score

    # Ajuster le score en fonction de l'alignement de momentum
```

```
confidence_score += momentum_alignment

# Vérifier la volatilité
volatility_factor = 0

for horizon, prediction in lstm_prediction.items():
    # Pondération selon l'horizon
    if "horizon_12" in horizon:  # Court terme
        weight = 0.5
    elif "horizon_24" in horizon:  # Moyen terme
        weight = 0.3
    else:  # Long terme
        weight = 0.2

    volatility = prediction.get("predicted_volatility", 1.0)

    # Volatilité faible est généralement favorable pour les positions longues
    # Volatilité élevée peut être favorable pour les positions courtes
    if technical_side == "BUY":
        if volatility < 0.8:
            volatility_score = 10 * weight
        elif volatility > 1.5:
            volatility_score = -20 * weight
        else:
            volatility_score = 0
    else:
        if volatility > 1.5:
            volatility_score = 10 * weight
        elif volatility < 0.8:
            volatility_score = -10 * weight
        else:
```

```
                volatility_score = 0

            volatility_factor += volatility_score

        # Ajuster le score en fonction de la volatilité
        confidence_score += volatility_factor

        # Limiter le score final entre 0 et 100
        confidence_score = max(0, min(100, confidence_score))

        return {
            "score": confidence_score,
            "direction_alignment": direction_alignment,
            "direction_confidence": direction_confidence,
            "momentum_alignment": momentum_alignment,
            "volatility_factor": volatility_factor
        }

    def _generate_reasoning(self, technical_opportunity: Dict,
                            lstm_prediction: Dict,
                            lstm_confidence: Dict,
                            combined_score: float) -> str:
        """
        Génère une explication détaillée pour l'opportunité combinée

        Args:
            technical_opportunity: Opportunité de la stratégie technique
            lstm_prediction: Prédictions du modèle LSTM
            lstm_confidence: Confiance dans les prédictions LSTM
            combined_score: Score combiné
```

```
    Returns:
        Explication textuelle
    """
    # Extraire les signaux techniques
    technical_signals = technical_opportunity.get("signals", {}).get("signals", [])
    technical_score = technical_opportunity["score"]
    technical_side = technical_opportunity["side"]

    # Prendre les horizons court et moyen terme
    short_term = None
    mid_term = None

    for horizon, prediction in lstm_prediction.items():
        if "horizon_12" in horizon:
            short_term = prediction
        elif "horizon_24" in horizon:
            mid_term = prediction

    # Construire l'explication
    reasoning = f"Opportunité de trading {technical_side} détectée avec un score combiné de {combined_score:.1f}/100. "

    # Explication technique
    reasoning += f"Analyse technique ({technical_score:.1f} pts): "
    if technical_signals:
        reasoning += ", ".join(technical_signals[:3])
        if len(technical_signals) > 3:
            reasoning += f" et {len(technical_signals)-3} autres signaux"
    else:
        reasoning += "Signaux de rebond technique détectés"
```

```
# Explication LSTM
reasoning += f". Prédictions IA: "

if short_term:
    direction_prob = short_term.get("direction_probability", 0.5) * 100
    momentum = short_term.get("predicted_momentum", 0)
    volatility = short_term.get("predicted_volatility", 1.0)

    # Direction
    if technical_side == "BUY":
        direction_text = f"{direction_prob:.1f}% de chance de hausse à court terme"
    else:
        direction_text = f"{(100-direction_prob):.1f}% de chance de baisse à court terme"

    # Momentum
    if abs(momentum) < 0.2:
        momentum_text = "momentum faible"
    elif abs(momentum) < 0.5:
        momentum_text = f"momentum {'positif' if momentum > 0 else 'négatif'} modéré"
    else:
        momentum_text = f"momentum {'positif' if momentum > 0 else 'négatif'} fort"

    # Volatilité
    if volatility < 0.8:
        volatility_text = "volatilité faible"
    elif volatility < 1.2:
        volatility_text = "volatilité normale"
    else:
        volatility_text = "volatilité élevée"

    reasoning += f"{direction_text}, {momentum_text}, {volatility_text}"
```

```
# Ajouter des informations sur le mid-term si disponible
if mid_term:
    direction_prob_mid = mid_term.get("direction_probability", 0.5) * 100

    if technical_side == "BUY":
        trend_coherence = "en cohérence" if direction_prob_mid > 50 else "en divergence"
    else:
        trend_coherence = "en cohérence" if direction_prob_mid < 50 else "en divergence"

    reasoning += f". Tendance à moyen terme {trend_coherence} ({direction_prob_mid:.1f}%)"

# Ajouter des informations sur le risk/reward
stop_loss = technical_opportunity.get("stop_loss", 0)
take_profit = technical_opportunity.get("take_profit", 0)
entry_price = technical_opportunity.get("entry_price", 0)

if entry_price > 0 and stop_loss > 0 and take_profit > 0:
    if technical_side == "BUY":
        risk = (entry_price - stop_loss) / entry_price * 100
        reward = (take_profit - entry_price) / entry_price * 100
    else:
        risk = (stop_loss - entry_price) / entry_price * 100
        reward = (entry_price - take_profit) / entry_price * 100

    risk_reward_ratio = reward / risk if risk > 0 else 0

    reasoning += f". Ratio risque/récompense: {risk_reward_ratio:.2f} ({risk:.2f}% / {reward:.2f}%)"

return reasoning
```

```
def update_position_stops(self, symbol: str, position: Dict, current_price: float) -> Dict:
    """
    Met à jour les niveaux de stop-loss d'une position en utilisant les prédictions

    Args:
        symbol: Paire de trading
        position: Données de la position
        current_price: Prix actuel

    Returns:
        Nouvelles données de stop-loss
    """
    position_id = position.get("id", "unknown")

    # 1. Récupérer les données de marché
    market_data = self.data_fetcher.get_market_data(symbol)

    # 2. Obtenir les prédictions LSTM
    lstm_prediction = self._get_lstm_prediction(symbol, market_data)

    # 3. Mettre à jour les stops en fonction des prédictions
    stops_update = self.adaptive_risk_manager.calculate_position_dynamic_stops(
        position_id,
        current_price,
        position,
        lstm_prediction
    )

    return stops_update

def should_close_early(self, symbol: str, position: Dict, current_price: float) -> Dict:
```

```
"""
Détermine si une position doit être fermée prématurément

Args:
    symbol: Paire de trading
    position: Données de la position
    current_price: Prix actuel

Returns:
    Décision de fermeture anticipée
"""
# Si le modèle LSTM n'est pas disponible, pas de fermeture anticipée
if self.lstm_model is None:
    return {"should_close": False}

position_id = position.get("id", "unknown")
side = position.get("side", "BUY")
entry_price = position.get("entry_price", current_price)

# Calculer le profit actuel en pourcentage
if side == "BUY":
    current_profit_pct = (current_price - entry_price) / entry_price * 100
else:
    current_profit_pct = (entry_price - current_price) / entry_price * 100

# 1. Récupérer les données de marché
market_data = self.data_fetcher.get_market_data(symbol)

# 2. Obtenir les prédictions LSTM
lstm_prediction = self._get_lstm_prediction(symbol, market_data)
```

```
# Si aucune prédiction disponible, ne pas fermer
if not lstm_prediction:
    return {"should_close": False}

# 3. Évaluer si la position doit être fermée
should_close = False
reason = ""

# Extraire les prédictions de court terme
short_term = None
for horizon, prediction in lstm_prediction.items():
    if "horizon_12" in horizon:
        short_term = prediction
        break

if short_term:
    direction_prob = short_term.get("direction_probability", 0.5)
    momentum = short_term.get("predicted_momentum", 0)

    # Pour les positions longues
    if side == "BUY":
        # Si forte probabilité de baisse et position en profit
        if direction_prob < 0.3 and momentum < -0.3 and current_profit_pct > 1:
            should_close = True
            reason = f"Forte probabilité de renversement baissier ({(1-direction_prob)*100:.1f}%)"

    # Pour les positions courtes
    else:
        # Si forte probabilité de hausse et position en profit
        if direction_prob > 0.7 and momentum > 0.3 and current_profit_pct > 1:
            should_close = True
```

```
                reason = f"Forte probabilité de renversement haussier ({direction_prob*100:.1f}%)"

        # 4. Vérifier les conditions de marché extrêmes
        extreme_conditions = self.adaptive_risk_manager._detect_extreme_market_conditions(market_data)
        if extreme_conditions["detected"] and current_profit_pct > 0:
            should_close = True
            reason = f"Conditions de marché extrêmes: {extreme_conditions['reason']}"

        return {
            "should_close": should_close,
            "reason": reason,
            "current_profit_pct": current_profit_pct
        }

    def get_market_prediction(self, symbol: str) -> Dict:
        """
        Fournit une prédiction de marché complète pour le tableau de bord

        Args:
            symbol: Paire de trading

        Returns:
            Prédiction complète du marché
        """
        market_data = self.data_fetcher.get_market_data(symbol)

        # Si le modèle LSTM n'est pas disponible, retourner une analyse technique standard
        if self.lstm_model is None:
            technical_analysis = self._get_technical_analysis(market_data)
            return {
```

```
            "symbol": symbol,
            "timestamp": datetime.now().isoformat(),
            "technical_analysis": technical_analysis,
            "lstm_available": False,
            "message": "Modèle LSTM non disponible, analyse technique uniquement"
        }

    # Obtenir les prédictions LSTM
    lstm_prediction = self._get_lstm_prediction(symbol, market_data)

    # Obtenir l'analyse technique
    technical_analysis = self._get_technical_analysis(market_data)

    # Combiner les analyses
    combined_analysis = self._combine_analysis(technical_analysis, lstm_prediction)

    return {
        "symbol": symbol,
        "timestamp": datetime.now().isoformat(),
        "technical_analysis": technical_analysis,
        "lstm_prediction": lstm_prediction,
        "combined_analysis": combined_analysis,
        "lstm_available": True
    }

def _get_technical_analysis(self, market_data: Dict) -> Dict:
    """
    Effectue une analyse technique standard

    Args:
        market_data: Données de marché
```

```
    Returns:
        Résultat de l'analyse technique
    """
    # Extraire les indicateurs
    indicators = market_data.get("primary_timeframe", {}).get("indicators", {})

    # Analyse RSI
    rsi_analysis = "neutre"
    rsi_value = 50

    if "rsi" in indicators:
        rsi_value = float(indicators["rsi"].iloc[-1])
        if rsi_value < 30:
            rsi_analysis = "survente"
        elif rsi_value < 40:
            rsi_analysis = "baissier modéré"
        elif rsi_value > 70:
            rsi_analysis = "surachat"
        elif rsi_value > 60:
            rsi_analysis = "haussier modéré"

    # Analyse des bandes de Bollinger
    bb_analysis = "neutre"
    bb_position = 0.5

    if "bollinger" in indicators and "percent_b" in indicators["bollinger"]:
        bb_position = float(indicators["bollinger"]["percent_b"].iloc[-1])
        if bb_position < 0:
            bb_analysis = "sous-bande inférieure"
        elif bb_position < 0.2:
```

```
            bb_analysis = "proche de la bande inférieure"
        elif bb_position > 1:
            bb_analysis = "au-dessus de la bande supérieure"
        elif bb_position > 0.8:
            bb_analysis = "proche de la bande supérieure"

    # Analyse de la tendance (EMA)
    trend_analysis = "neutre"

    if "ema" in indicators:
        ema_short = indicators["ema"].get("ema_9", pd.Series()).iloc[-1] if "ema_9" in indicators["ema"] else None
        ema_medium = indicators["ema"].get("ema_21", pd.Series()).iloc[-1] if "ema_21" in indicators["ema"] else None
        ema_long = indicators["ema"].get("ema_50", pd.Series()).iloc[-1] if "ema_50" in indicators["ema"] else None

        if ema_short is not None and ema_medium is not None and ema_long is not None:
            if ema_short > ema_medium > ema_long:
                trend_analysis = "haussier fort"
            elif ema_short > ema_medium:
                trend_analysis = "haussier"
            elif ema_short < ema_medium < ema_long:
                trend_analysis = "baissier fort"
            elif ema_short < ema_medium:
                trend_analysis = "baissier"

    # Analyse ADX (force de tendance)
    adx_analysis = "tendance faible"
    adx_value = 0

    if "adx" in indicators:
```

```
            adx_data = indicators["adx"]
            adx_value = float(adx_data["adx"].iloc[-1])
            plus_di = float(adx_data["plus_di"].iloc[-1])
            minus_di = float(adx_data["minus_di"].iloc[-1])

            if adx_value > 25:
                if plus_di > minus_di:
                    adx_analysis = "tendance haussière forte"
                else:
                    adx_analysis = "tendance baissière forte"
            else:
                adx_analysis = "tendance faible"

        return {
            "trend": trend_analysis,
            "momentum": {
                "rsi": rsi_analysis,
                "rsi_value": rsi_value
            },
            "volatility": {
                "bollinger": bb_analysis,
                "bollinger_position": bb_position
            },
            "strength": {
                "adx": adx_analysis,
                "adx_value": adx_value
            },
            "summary": self._generate_technical_summary(trend_analysis, rsi_analysis, bb_analysis,
adx_analysis)
        }
```

```
def _generate_technical_summary(self, trend: str, rsi: str, bollinger: str, adx: str) -> Dict:
    """
    Génère un résumé de l'analyse technique

    Args:
        trend: Analyse de tendance
        rsi: Analyse RSI
        bollinger: Analyse Bollinger
        adx: Analyse ADX

    Returns:
        Résumé de l'analyse technique
    """
    # Calculer un score haussier/baissier
    bullish_score = 0
    bearish_score = 0

    # Évaluer la tendance
    if "haussier fort" in trend:
        bullish_score += 3
    elif "haussier" in trend:
        bullish_score += 2
    elif "baissier fort" in trend:
        bearish_score += 3
    elif "baissier" in trend:
        bearish_score += 2

    # Évaluer le RSI
    if "survente" in rsi:
        bullish_score += 2  # Potentiel rebond
    elif "surachat" in rsi:
```

```
    bearish_score += 2  # Potentiel repli
elif "baissier" in rsi:
    bearish_score += 1
elif "haussier" in rsi:
    bullish_score += 1

# Évaluer les bandes de Bollinger
if "sous-bande" in bollinger:
    bullish_score += 2  # Potentiel rebond
elif "au-dessus" in bollinger:
    bearish_score += 2  # Potentiel repli
elif "proche de la bande inférieure" in bollinger:
    bullish_score += 1
elif "proche de la bande supérieure" in bollinger:
    bearish_score += 1

# Évaluer l'ADX
adx_multiplier = 1
if "forte" in adx:
    adx_multiplier = 1.5

# Conclusion
total_bullish = bullish_score * adx_multiplier
total_bearish = bearish_score * adx_multiplier

bias = "neutre"
if total_bullish > total_bearish * 1.5:
    bias = "fortement haussier"
elif total_bullish > total_bearish:
    bias = "modérément haussier"
elif total_bearish > total_bullish * 1.5:
```

```
            bias = "fortement baissier"
        elif total_bearish > total_bullish:
            bias = "modérément baissier"

        return {
            "bias": bias,
            "bullish_score": total_bullish,
            "bearish_score": total_bearish
        }

    def _combine_analysis(self, technical: Dict, lstm: Optional[Dict]) -> Dict:
        """
        Combine l'analyse technique et les prédictions LSTM

        Args:
            technical: Analyse technique
            lstm: Prédictions LSTM

        Returns:
            Analyse combinée
        """
        # Si pas de prédictions LSTM, retourner l'analyse technique
        if not lstm:
            return {
                "overall_bias": technical["summary"]["bias"],
                "confidence": "moyenne",
                "timeframes": {
                    "short_term": technical["summary"]["bias"],
                    "mid_term": "indéterminé",
                    "long_term": "indéterminé"
                },
```

```
        "explanation": "Basé uniquement sur l'analyse technique, LSTM non disponible"
    }

# Extraire les prédictions par horizon
short_term = None
mid_term = None
long_term = None

for horizon, prediction in lstm.items():
    if "horizon_12" in horizon:
        short_term = prediction
    elif "horizon_24" in horizon:
        mid_term = prediction
    else:
        long_term = prediction

# Déterminer le biais pour chaque horizon
short_term_bias = "neutre"
short_term_confidence = "faible"
mid_term_bias = "neutre"
long_term_bias = "neutre"

if short_term:
    direction_prob = short_term.get("direction_probability", 0.5)
    momentum = short_term.get("predicted_momentum", 0)

    if direction_prob > 0.7:
        short_term_bias = "fortement haussier"
        short_term_confidence = "élevée"
    elif direction_prob > 0.6:
        short_term_bias = "modérément haussier"
```

```
            short_term_confidence = "moyenne"
        elif direction_prob < 0.3:
            short_term_bias = "fortement baissier"
            short_term_confidence = "élevée"
        elif direction_prob < 0.4:
            short_term_bias = "modérément baissier"
            short_term_confidence = "moyenne"

    if mid_term:
        direction_prob = mid_term.get("direction_probability", 0.5)

        if direction_prob > 0.65:
            mid_term_bias = "haussier"
        elif direction_prob < 0.35:
            mid_term_bias = "baissier"

    if long_term:
        direction_prob = long_term.get("direction_probability", 0.5)

        if direction_prob > 0.6:
            long_term_bias = "haussier"
        elif direction_prob < 0.4:
            long_term_bias = "baissier"

    # Combiner les analyses
    technical_bias = technical["summary"]["bias"]

    # Déterminer la cohérence entre technique et LSTM
    is_coherent = (
        ("haussier" in technical_bias and "haussier" in short_term_bias) or
        ("baissier" in technical_bias and "baissier" in short_term_bias)
```

```
        )

        overall_bias = "neutre"
        confidence = "moyenne"

        if is_coherent:
            # Si cohérent, renforcer le signal
            if "fortement" in technical_bias or "fortement" in short_term_bias:
                overall_bias = "fortement " + ("haussier" if "haussier" in short_term_bias else "baissier")
                confidence = "élevée"
            else:
                overall_bias = "modérément " + ("haussier" if "haussier" in short_term_bias else "baissier")
                confidence = "moyenne"
        else:
            # Si incohérent, favoriser légèrement les prédictions LSTM
            if short_term_confidence == "élevée":
                overall_bias = short_term_bias
                confidence = "moyenne"  # Réduite en raison de l'incohérence
            else:
                # Compromis
                overall_bias = "neutre avec tendance " + (
                    "haussière" if "haussier" in technical_bias or "haussier" in short_term_bias else
"baissière"
                )
                confidence = "faible"

        # Générer une explication
        explanation = self._generate_combined_explanation(
            technical_bias, short_term_bias, mid_term_bias, is_coherent
        )
```

```
        return {
            "overall_bias": overall_bias,
            "confidence": confidence,
            "is_coherent": is_coherent,
            "timeframes": {
                "short_term": short_term_bias,
                "mid_term": mid_term_bias,
                "long_term": long_term_bias
            },
            "explanation": explanation
        }

    def _generate_combined_explanation(self, technical_bias: str, short_term_bias: str,
                                       mid_term_bias: str, is_coherent: bool) -> str:
        """
        Génère une explication pour l'analyse combinée

        Args:
            technical_bias: Biais de l'analyse technique
            short_term_bias: Biais LSTM court terme
            mid_term_bias: Biais LSTM moyen terme
            is_coherent: Indique si les analyses sont cohérentes

        Returns:
            Explication textuelle
        """
        if is_coherent:
            explanation = f"L'analyse technique ({technical_bias}) est en accord avec les prédictions IA ({short_term_bias}), "

            if "haussier" in technical_bias:
```

```
            explanation += "suggérant un potentiel de hausse. "
        else:
            explanation += "indiquant une pression vendeuse. "

        if mid_term_bias != "neutre":
            explanation += f"Le moyen terme est également {mid_term_bias}. "

            if ("haussier" in short_term_bias and "haussier" in mid_term_bias) or \
               ("baissier" in short_term_bias and "baissier" in mid_term_bias):
                explanation += "La cohérence entre horizons renforce la fiabilité du signal."
            else:
                explanation += "Attention à la divergence entre court et moyen terme."
    else:
        explanation = f"L'analyse technique ({technical_bias}) diverge des prédictions IA ({short_term_bias}). "

        explanation += "Cette divergence suggère une période d'incertitude. "

        if "haussier" in technical_bias:
            explanation += "Les indicateurs techniques montrent des signes haussiers, "
        else:
            explanation += "Les indicateurs techniques montrent des signes baissiers, "

        if "haussier" in short_term_bias:
            explanation += "tandis que l'IA prédit une tendance haussière à court terme. "
        else:
            explanation += "tandis que l'IA prédit une tendance baissière à court terme. "

        explanation += "Considérez une exposition réduite dans ce contexte contradictoire."

    return explanation
```

```
================================================
File: crypto_trading_bot_CLAUDE/strategies/market_state.py
================================================
# strategies/market_state.py
"""
Analyseur de l'état du marché
"""
import logging
import pandas as pd
import numpy as np
from typing import Dict, List, Optional, Union
from datetime import datetime, timedelta

def detect_divergence(ohlcv: pd.DataFrame, rsi: pd.Series, lookback: int = 10) -> Dict[str, bool]:
    """
    Détecte les divergences haussières entre le prix et le RSI

    Args:
        ohlcv: DataFrame avec les données OHLCV
        rsi: Series contenant les valeurs RSI
        lookback: Nombre de périodes à analyser

    Returns:
        Dict avec le résultat de l'analyse des divergences
    """
    # Prendre les n dernières périodes
    price_lows = ohlcv['low'].tail(lookback)
    rsi_values = rsi.tail(lookback)

    # Trouver les plus bas
```

```
    price_min = np.min(price_lows)
    price_min_idx = price_lows.idxmin()
    rsi_min = np.min(rsi_values)
    rsi_min_idx = rsi_values.idxmin()

    # Détecter divergence haussière (prix fait un plus bas mais pas le RSI)
    bullish = (price_min_idx > rsi_min_idx and
            price_lows.iloc[-1] <= price_min * 1.02 and
            rsi_values.iloc[-1] > rsi_min * 1.02)

    return {"bullish": bullish}

from config.config import PRIMARY_TIMEFRAME, SECONDARY_TIMEFRAMES
from config.trading_params import ADX_THRESHOLD, MARKET_COOLDOWN_PERIOD
from utils.logger import setup_logger

logger = setup_logger("market_state")

class MarketStateAnalyzer:
    """
    Analyse l'état du marché pour déterminer si les conditions sont favorables au trading
    """
    def __init__(self, data_fetcher):
        self.data_fetcher = data_fetcher
        self.unfavorable_since = {}  # {symbol: timestamp}

    def analyze_market_state(self, symbol: str) -> Dict:
        """
        Analyse l'état du marché pour un symbole donné

        Args:
```

```
        symbol: Paire de trading

    Returns:
        Dictionnaire avec l'analyse de l'état du marché
    """
    # Récupérer les données de marché
    market_data = self.data_fetcher.get_market_data(symbol)

    # Vérifier si des données sont disponibles
    if market_data["primary_timeframe"].get("ohlcv") is None or
market_data["primary_timeframe"].get("ohlcv").empty:
        logger.warning(f"Données de marché non disponibles pour {symbol}")
        return {
            "favorable": False,
            "reason": "Données non disponibles",
            "cooldown": False,
            "details": {}
        }

    # Extraire les données et indicateurs
    ohlcv = market_data["primary_timeframe"]["ohlcv"]
    indicators = market_data["primary_timeframe"].get("indicators", {})

    # Analyse de l'état du marché
    market_state = {
        "favorable": True,  # Par défaut, considérer le marché comme favorable
        "reason": "Conditions normales",
        "cooldown": False,
        "details": {}
    }
```

```
    # 1. Vérifier la force de la tendance avec ADX
    adx_data = indicators.get("adx", {})
    if adx_data and "adx" in adx_data:
        adx_value = adx_data["adx"].iloc[-1]
        plus_di = adx_data["plus_di"].iloc[-1]
        minus_di = adx_data["minus_di"].iloc[-1]

        strong_trend = adx_value > ADX_THRESHOLD
        bearish_trend = minus_di > plus_di

        market_state["details"]["adx"] = {
            "value": float(adx_value),
            "plus_di": float(plus_di),
            "minus_di": float(minus_di),
            "strong_trend": bool(strong_trend),
            "bearish_trend": bool(bearish_trend)
        }

        # Si forte tendance baissière, marché défavorable
        if strong_trend and bearish_trend:
            market_state["favorable"] = False
            market_state["reason"] = "Forte tendance baissière"

    # 2. Vérifier l'alignement des EMA
    ema_data = indicators.get("ema", {})
    if ema_data:
        ema_short = ema_data.get("ema_9", pd.Series()).iloc[-1] if "ema_9" in ema_data else None
        ema_medium = ema_data.get("ema_21", pd.Series()).iloc[-1] if "ema_21" in ema_data else
None
        ema_long = ema_data.get("ema_50", pd.Series()).iloc[-1] if "ema_50" in ema_data else None
        ema_baseline = ema_data.get("ema_200", pd.Series()).iloc[-1] if "ema_200" in ema_data else
None
```

```
            current_price = ohlcv["close"].iloc[-1]

            # Vérifier l'alignement baissier
            if (ema_short is not None and ema_medium is not None and
                ema_long is not None and ema_baseline is not None):

                bearish_alignment = (ema_short < ema_medium < ema_long < ema_baseline)
                price_below_baseline = current_price < ema_baseline

                market_state["details"]["ema_alignment"] = {
                    "bearish_alignment": bool(bearish_alignment),
                    "price_below_baseline": bool(price_below_baseline)
                }

                # Si prix sous les EMA importantes, marché défavorable
                if bearish_alignment and price_below_baseline:
                    # Vérifier si les conditions étaient déjà défavorables
                    if market_state["favorable"]:
                        market_state["favorable"] = False
                        market_state["reason"] = "Alignement baissier des EMA"

        # 3. Vérifier les bandes de Bollinger
        bb_data = indicators.get("bollinger", {})
        if bb_data:
            bb_middle = bb_data.get("middle", pd.Series()).iloc[-1] if "middle" in bb_data else None
            bb_bandwidth = bb_data.get("bandwidth", pd.Series()).iloc[-1] if "bandwidth" in bb_data else
None

            # Vérifier si la volatilité est excessive
            if bb_bandwidth is not None:
```

```
            high_volatility = bb_bandwidth > 0.1  # Seuil arbitraire, à ajuster

            market_state["details"]["bollinger"] = {
                "bandwidth": float(bb_bandwidth),
                "high_volatility": bool(high_volatility)
            }

        # 4. Vérifier le RSI pour les conditions de survente/surachat
        rsi_data = indicators.get("rsi", None)
        if rsi_data is not None:
            rsi_value = rsi_data.iloc[-1]

            market_state["details"]["rsi"] = {
                "value": float(rsi_value)
            }

        # 5. Vérifier s'il faut mettre en place un cooldown
        if not market_state["favorable"]:
            current_time = datetime.now()

            # Si le marché vient de devenir défavorable, enregistrer le timestamp
            if symbol not in self.unfavorable_since:
                self.unfavorable_since[symbol] = current_time
                logger.info(f"Marché défavorable pour {symbol}, début du cooldown")

            # Vérifier si le cooldown est toujours actif
            cooldown_end = self.unfavorable_since[symbol] +
timedelta(minutes=MARKET_COOLDOWN_PERIOD)
            if current_time < cooldown_end:
                market_state["cooldown"] = True
                minutes_remaining = int((cooldown_end - current_time).total_seconds() / 60)
```

```
                market_state["cooldown_remaining"] = minutes_remaining
                market_state["reason"] += f" (Cooldown: {minutes_remaining} min restantes)"
        else:
            # Si le marché est favorable, réinitialiser le cooldown
            if symbol in self.unfavorable_since:
                del self.unfavorable_since[symbol]

        # Ajouter un filtre de tendance
        ema_data = indicators.get("ema", {})
        if ema_data:
            price = ohlcv["close"].iloc[-1]
            ema_200 = ema_data.get("ema_200", pd.Series()).iloc[-1] if "ema_200" in ema_data else
None

            # Ne trader à l'achat que si le prix est au-dessus de la EMA200
            if ema_200 is not None and price < ema_200:
                market_state["favorable"] = False
                market_state["reason"] = "Prix sous la EMA200, éviter les longs"

        # Ajouter un filtre de volatilité
        if "bollinger" in indicators:
            bandwidth = indicators["bollinger"].get("bandwidth", pd.Series()).iloc[-1]
            if bandwidth > 0.06:  # Seuil de volatilité élevée
                market_state["favorable"] = False
                market_state["reason"] = "Volatilité trop élevée"

        return market_state

    def check_for_reversal(self, symbol: str) -> Dict:
        """
        Recherche des signaux de retournement pour sortir d'un cooldown
```

```
        Args:
            symbol: Paire de trading

        Returns:
            Dictionnaire avec les signaux de retournement
        """
        # Récupérer les données de marché
        market_data = self.data_fetcher.get_market_data(symbol)

        # Vérifier si des données sont disponibles
        if market_data["primary_timeframe"].get("ohlcv") is None or
market_data["primary_timeframe"].get("ohlcv").empty:
            return {
                "reversal_detected": False,
                "confidence": 0,
                "reason": "Données non disponibles"
            }

        # Extraire les données et indicateurs
        ohlcv = market_data["primary_timeframe"]["ohlcv"]
        indicators = market_data["primary_timeframe"].get("indicators", {})

        # Initialiser les signaux de retournement
        reversal_signals = []
        confidence = 0

        # 1. Vérifier le RSI pour les conditions de survente
        rsi_data = indicators.get("rsi", None)
        if rsi_data is not None:
            rsi_value = rsi_data.iloc[-1]
```

```
    rsi_prev = rsi_data.iloc[-2] if len(rsi_data) > 1 else None

    # RSI en zone de survente
    if rsi_value < 30:
        reversal_signals.append("RSI en zone de survente")
        confidence += 20

    # RSI qui remonte depuis la zone de survente
    if rsi_prev is not None and rsi_prev < 30 and rsi_value > rsi_prev:
        reversal_signals.append("RSI remonte depuis la zone de survente")
        confidence += 15

# 2. Vérifier les bougies de retournement
if len(ohlcv) >= 3:
    current_candle = {
        "open": ohlcv["open"].iloc[-1],
        "high": ohlcv["high"].iloc[-1],
        "low": ohlcv["low"].iloc[-1],
        "close": ohlcv["close"].iloc[-1]
    }

    prev_candle = {
        "open": ohlcv["open"].iloc[-2],
        "high": ohlcv["high"].iloc[-2],
        "low": ohlcv["low"].iloc[-2],
        "close": ohlcv["close"].iloc[-2]
    }

    prev_prev_candle = {
        "open": ohlcv["open"].iloc[-3],
        "high": ohlcv["high"].iloc[-3],
```

```
            "low": ohlcv["low"].iloc[-3],
            "close": ohlcv["close"].iloc[-3]
        }

        # Vérifier le marteau ou étoile du matin
        if (prev_candle["close"] < prev_candle["open"] and  # Bougie baissière
            current_candle["close"] > current_candle["open"] and  # Bougie haussière
            current_candle["close"] > prev_candle["open"]):  # Clôture au-dessus de l'ouverture précédente

            reversal_signals.append("Motif de retournement haussier")
            confidence += 25

        # Vérifier le double bottom
        if (prev_prev_candle["low"] < prev_prev_candle["open"] and
            prev_candle["low"] <= prev_prev_candle["low"] * 1.01 and  # Deuxième creux similaire
            current_candle["close"] > prev_candle["high"]):  # Cassure haussière

            reversal_signals.append("Double bottom potentiel")
            confidence += 30

    # 3. Vérifier la divergence haussière sur le RSI
    if rsi_data is not None and len(ohlcv) >= 10:
        divergence = detect_divergence(ohlcv, rsi_data)

        if divergence["bullish"]:
            reversal_signals.append("Divergence haussière RSI")
            confidence += 35

    # 4. Vérifier le croisement des EMA courtes
    ema_data = indicators.get("ema", {})
```

```
        if "ema_9" in ema_data and "ema_21" in ema_data:
            ema_short = ema_data["ema_9"]
            ema_medium = ema_data["ema_21"]

            if len(ema_short) >= 2 and len(ema_medium) >= 2:
                current_cross = ema_short.iloc[-1] > ema_medium.iloc[-1]
                prev_cross = ema_short.iloc[-2] <= ema_medium.iloc[-2]

                if current_cross and prev_cross:
                    reversal_signals.append("Croisement EMA 9/21 haussier")
                    confidence += 20

        # Résultat final
        reversal_detected = confidence >= 50  # Seuil de confiance

        return {
            "reversal_detected": reversal_detected,
            "confidence": confidence,
            "signals": reversal_signals,
            "details": {
                "rsi": float(rsi_data.iloc[-1]) if rsi_data is not None else None
            }
        }
```

```
================================================
File: crypto_trading_bot_CLAUDE/strategies/strategy_base.py
================================================
# strategies/strategy_base.py
"""
Classe de base pour les stratégies de trading
"""
```

```
import os
import json
import logging
from abc import ABC, abstractmethod
from typing import Dict, List, Optional, Union
from datetime import datetime

from config.config import DATA_DIR
from config.trading_params import MINIMUM_SCORE_TO_TRADE
from utils.logger import setup_logger

logger = setup_logger("strategy_base")

class StrategyBase(ABC):
    """
    Classe de base abstraite pour les stratégies de trading
    """
    def __init__(self, data_fetcher, market_analyzer, scoring_engine):
        self.data_fetcher = data_fetcher
        self.market_analyzer = market_analyzer
        self.scoring_engine = scoring_engine
        self.min_score = MINIMUM_SCORE_TO_TRADE

        # Répertoire pour les journaux de trades
        self.trades_dir = os.path.join(DATA_DIR, "trade_logs")
        if not os.path.exists(self.trades_dir):
            os.makedirs(self.trades_dir)

    @abstractmethod
    def find_trading_opportunity(self, symbol: str) -> Optional[Dict]:
        """
```

```
    Cherche une opportunité de trading pour le symbole donné

    Args:
        symbol: Paire de trading

    Returns:
        Opportunité de trading ou None si aucune opportunité n'est trouvée
    """
    pass

def log_trade(self, opportunity: Dict, order_result: Dict) -> None:
    """
    Enregistre les détails d'un trade dans un fichier JSON

    Args:
        opportunity: Opportunité de trading
        order_result: Résultat de l'ordre
    """
    # Créer un identifiant unique pour le trade
    trade_id = order_result.get("position_id", f"trade_{int(datetime.now().timestamp())}")

    # Préparer les données du trade
    trade_data = {
        "trade_id": trade_id,
        "symbol": opportunity.get("symbol"),
        "timestamp": datetime.now().isoformat(),
        "score": opportunity.get("score"),
        "reasoning": opportunity.get("reasoning"),
        "entry_price": order_result.get("entry_price"),
        "stop_loss": order_result.get("stop_loss_price"),
        "take_profit": order_result.get("take_profit_price"),
```

```
            "indicators": opportunity.get("indicators", {}),
            "market_conditions": opportunity.get("market_conditions", {})
        }

        # Enregistrer dans un fichier JSON
        filename = os.path.join(self.trades_dir, f"{trade_id}.json")

        try:
            with open(filename, 'w') as f:
                json.dump(trade_data, f, indent=2, default=str)

            logger.info(f"Trade enregistré: {filename}")
        except Exception as e:
            logger.error(f"Erreur lors de l'enregistrement du trade: {str(e)}")

    def update_trade_result(self, trade_id: str, result: Dict) -> None:
        """
        Met à jour le fichier de journal d'un trade avec les résultats

        Args:
            trade_id: ID du trade
            result: Résultat du trade
        """
        filename = os.path.join(self.trades_dir, f"{trade_id}.json")

        if not os.path.exists(filename):
            logger.warning(f"Fichier de trade non trouvé: {filename}")
            return

        try:
            # Charger les données existantes
```

```
            with open(filename, 'r') as f:
                trade_data = json.load(f)

            # Ajouter les résultats
            trade_data["close_timestamp"] = datetime.now().isoformat()
            trade_data["exit_price"] = result.get("exit_price")
            trade_data["pnl_percent"] = result.get("pnl_percent")
            trade_data["pnl_absolute"] = result.get("pnl_absolute")
            trade_data["trade_duration"] = result.get("trade_duration")
            trade_data["exit_reason"] = result.get("exit_reason")

            # Enregistrer les données mises à jour
            with open(filename, 'w') as f:
                json.dump(trade_data, f, indent=2, default=str)

            logger.info(f"Résultat du trade mis à jour: {filename}")
        except Exception as e:
            logger.error(f"Erreur lors de la mise à jour du résultat du trade: {str(e)}")


================================================
File: crypto_trading_bot_CLAUDE/strategies/technical_bounce.py
================================================

# strategies/technical_bounce.py
"""
Stratégie de rebond technique
"""
import pandas as pd
import numpy as np
from typing import Dict, List, Optional, Union
```

```
from datetime import datetime

from strategies.strategy_base import StrategyBase
from config.trading_params import (
    RSI_OVERSOLD,
    STOP_LOSS_PERCENT,
    TAKE_PROFIT_PERCENT
)
from utils.logger import setup_logger

# Importer la fonction de détection de divergence
from indicators.momentum import detect_divergence

logger = setup_logger("technical_bounce")

class TechnicalBounceStrategy(StrategyBase):
    """
    Stratégie qui cherche à capturer les rebonds techniques après des baisses de prix
    """
    def __init__(self, data_fetcher, market_analyzer, scoring_engine):
        super().__init__(data_fetcher, market_analyzer, scoring_engine)

    def find_trading_opportunity(self, symbol: str) -> Optional[Dict]:
        """
        Cherche une opportunité de rebond technique pour le symbole donné

        Args:
            symbol: Paire de trading

        Returns:
            Opportunité de trading ou None si aucune opportunité n'est trouvée
```

```
"""
# Récupérer les données de marché
market_data = self.data_fetcher.get_market_data(symbol)

# Vérifier si des données sont disponibles
if (market_data["primary_timeframe"].get("ohlcv") is None or
    market_data["primary_timeframe"].get("ohlcv").empty):
    logger.warning(f"Données de marché non disponibles pour {symbol}")
    return None

# Extraire les données et indicateurs
ohlcv = market_data["primary_timeframe"]["ohlcv"]
indicators = market_data["primary_timeframe"].get("indicators", {})

# Vérifier l'état du marché
market_state = self.market_analyzer.analyze_market_state(symbol)
if not market_state["favorable"] or market_state["cooldown"]:
    return None

# Rechercher des signaux de rebond technique
bounce_signals = self._detect_bounce_signals(symbol, ohlcv, indicators)

# Si aucun signal de rebond n'est trouvé, retourner None
if not bounce_signals["bounce_detected"]:
    return None

# Calculer les niveaux d'entrée, de stop-loss et de take-profit
current_price = ohlcv["close"].iloc[-1]

# Pour un ordre long (achat)
entry_price = current_price
```

```
        stop_loss = entry_price * (1 - STOP_LOSS_PERCENT/100)
        take_profit = entry_price * (1 + TAKE_PROFIT_PERCENT/100)

        # Calculer le score de l'opportunité
        opportunity_score = self._calculate_opportunity_score(bounce_signals, market_state, ohlcv, indicators)

        # Générer une explication textuelle
        reasoning = self._generate_reasoning(bounce_signals, market_state, opportunity_score)

        # Créer l'opportunité de trading
        opportunity = {
            "symbol": symbol,
            "strategy": "technical_bounce",
            "side": "BUY",  # Cette stratégie ne prend que des positions longues
            "entry_price": entry_price,
            "stop_loss": stop_loss,
            "take_profit": take_profit,
            "score": opportunity_score,
            "reasoning": reasoning,
            "signals": bounce_signals,
            "market_conditions": market_state,
            "timestamp": datetime.now(),
            "indicators": {
                "rsi": float(indicators["rsi"].iloc[-1]) if "rsi" in indicators else None,
                "bollinger": {
                    "lower": float(indicators["bollinger"]["lower"].iloc[-1]) if "bollinger" in indicators else None,
                    "percent_b": float(indicators["bollinger"]["percent_b"].iloc[-1]) if "bollinger" in indicators else None
                }
            }
```

```
        }

        logger.info(f"Opportunité de rebond technique trouvée pour {symbol} (score: {opportunity_score})")
        return opportunity

    def _detect_bounce_signals(self, symbol: str, ohlcv: pd.DataFrame, indicators: Dict) -> Dict:
        """
        Détecte les signaux de rebond technique avec critères améliorés

        Args:
            symbol: Paire de trading
            ohlcv: DataFrame avec les données OHLCV
            indicators: Dictionnaire des indicateurs techniques

        Returns:
            Dictionnaire avec les signaux de rebond détectés
        """
        bounce_signals = {
            "bounce_detected": False,
            "signals": [],
            "strength": 0,
            "volume_ratio": 1.0,  # Par défaut
            "multi_timeframe_confirmation": 0  # Nouveau: nombre de timeframes confirmant
        }

        # Vérifier la présence des indicateurs nécessaires
        if "rsi" not in indicators or "bollinger" not in indicators:
            return bounce_signals
         # Détection de la tendance de marché
        trend_direction = self._detect_market_trend(ohlcv, indicators)
```

```
# Seuil de force pour détecter un rebond - RÉDUIT pour être moins strict
strength_threshold = 35  # Était probablement plus élevé, réduit à 35
# Extraire les indicateurs
rsi = indicators["rsi"]
bollinger = indicators["bollinger"]

# NOUVEAU: Analyser le contexte de marché et filtrer les conditions défavorables
trend_direction = self._detect_market_trend(ohlcv, indicators)
if trend_direction == "strong_bearish":
    # Si tendance fortement baissière, exiger des signaux plus forts
    bounce_signals["trend_context"] = "strong_bearish"
    strength_threshold = 70  # Exiger des signaux plus forts dans un marché baissier
else:
    strength_threshold = 40  # Seuil normal

# NOUVEAU: Analyse de la structure de prix
price_structure = self._analyze_price_structure(ohlcv)
if price_structure.get("double_bottom", False):
    bounce_signals["signals"].append("Structure de double fond détectée")
    bounce_signals["strength"] += 25

# NOUVEAU: Vérification des niveaux de support
support_test = self._check_support_test(ohlcv, indicators)
if support_test["support_tested"]:
    bounce_signals["signals"].append(f"Test de support à {support_test['support_level']:.2f}")
    bounce_signals["strength"] += 20

# 1. Vérifier le RSI en zone de survente
if len(rsi) >= 2:
    rsi_current = rsi.iloc[-1]
```

```
            rsi_prev = rsi.iloc[-2]

            # ASSOUPLI: Seuil de RSI légèrement augmenté pour capturer plus de signaux
            oversold_condition = rsi_current < 32  # Légèrement plus permissif (standard est 30)
            rsi_turning_up = rsi_current > rsi_prev and rsi_prev < 35  # Moins strict sur le seuil

            if oversold_condition:
                bounce_signals["signals"].append("RSI en zone de survente")
                bounce_signals["strength"] += 20  # Augmenté pour donner plus d'importance

            if rsi_turning_up:
                bounce_signals["signals"].append("RSI remonte depuis zone basse")
                bounce_signals["strength"] += 15

        # 2. Vérifier les bandes de Bollinger
        if "percent_b" in bollinger and len(bollinger["percent_b"]) >= 2:
            percent_b_current = bollinger["percent_b"].iloc[-1]
            percent_b_prev = bollinger["percent_b"].iloc[-2]

            # ASSOUPLI: Seuils légèrement ajustés
            price_below_lower_band = percent_b_current < 0.05  # Était 0
            price_returning_to_band = percent_b_current > percent_b_prev and percent_b_prev < 0.1  # Était probablement plus strict

            if price_below_lower_band:
                bounce_signals["signals"].append("Prix sous la bande inférieure de Bollinger")
                bounce_signals["strength"] += 20

            if price_returning_to_band:
                bounce_signals["signals"].append("Prix remonte vers la bande inférieure")
                bounce_signals["strength"] += 15
```

```
    # 3. Vérifier les mèches (wicks) des chandeliers
    if len(ohlcv) >= 2:
        current_candle = ohlcv.iloc[-1]
        prev_candle = ohlcv.iloc[-2]

        current_body = abs(current_candle["close"] - current_candle["open"])
        current_total_range = current_candle["high"] - current_candle["low"]
        current_lower_wick = min(current_candle["open"], current_candle["close"]) - current_candle["low"]

        # ASSOUPLI: Seuil légèrement réduit pour les mèches inférieures
        if current_total_range > 0 and current_lower_wick / current_total_range > 0.4:  # Était 0.5
            bounce_signals["signals"].append("Mèche inférieure significative (rejet)")
            bounce_signals["strength"] += 18

        # Vérifier si le chandelier actuel est haussier après un chandelier baissier
        current_bullish = current_candle["close"] > current_candle["open"]
        prev_bearish = prev_candle["close"] < prev_candle["open"]

        if current_bullish and prev_bearish:
            bounce_signals["signals"].append("Chandelier haussier après chandelier baissier")
            bounce_signals["strength"] += 15

    # 4. Vérifier les divergences haussières - conservé tel quel
    if "rsi" in indicators and len(ohlcv) >= 10:
        from indicators.momentum import detect_divergence
        divergence = detect_divergence(ohlcv, rsi)

        if divergence["bullish"]:
            bounce_signals["signals"].append("Divergence haussière RSI détectée")
```

```
        bounce_signals["strength"] += 25  # Augmenté car signal fiable

# 5. Vérifier les pics de volume
if len(ohlcv) >= 5:
    # Calculer la moyenne des volumes récents (sauf le dernier)
    avg_volume = ohlcv['volume'].iloc[-5:-1].mean()
    current_volume = ohlcv['volume'].iloc[-1]
    volume_ratio = current_volume / avg_volume if avg_volume > 0 else 1.0

    # ASSOUPLI: Seuil réduit pour les volumes
    if volume_ratio > 1.7:  # Était probablement 2.0
        # Vérifier si c'est un volume de capitulation avec clôture haussière
        if ohlcv['close'].iloc[-1] > ohlcv['open'].iloc[-1]:
            bounce_signals["signals"].append("Volume élevé avec clôture haussière")
            bounce_signals["strength"] += 18
            bounce_signals["volume_ratio"] = volume_ratio

# Déterminer si un rebond est détecté
# ASSOUPLI: Exigences réduites pour la détection de rebond
bounce_signals["bounce_detected"] = (
    len(bounce_signals["signals"]) >= 2 and  # Maintenu à 2 minimum
    bounce_signals["strength"] >= strength_threshold  # Seuil réduit
)

# MODIFICATION IMPORTANTE: Moins strict sur les conditions de marché en tendance baissière
if trend_direction == "bearish" or trend_direction == "strong_bearish":
    # Ne pas rejeter automatiquement en tendance baissière
    # mais exiger plus de signaux de confirmation
    if bounce_signals["bounce_detected"] and len(bounce_signals["signals"]) < 3:
        bounce_signals["bounce_detected"] = False
# 6. Vérifier les patterns de reversal
```

```
        if len(ohlcv) >= 3:
            # Vérifier le pattern de hammer (marteau)
            last_candle = ohlcv.iloc[-1]
            body_size = abs(last_candle['close'] - last_candle['open'])
            total_range = last_candle['high'] - last_candle['low']
            lower_wick = min(last_candle['open'], last_candle['close']) - last_candle['low']

            # Un marteau a une petite tête et une longue mèche basse
            if total_range > 0 and body_size / total_range < 0.3 and lower_wick / total_range > 0.6:
                bounce_signals["signals"].append("Pattern de marteau détecté")
                bounce_signals["strength"] += 20

        # Déterminer si un rebond est détecté
        bounce_signals["bounce_detected"] = len(bounce_signals["signals"]) >= 2 and
bounce_signals["strength"] >= 40

        # Vérifier la convergence de plusieurs indicateurs
        signal_count = len(bounce_signals["signals"])

        # Renforcer les critères pour détecter un rebond
        if signal_count >= 3:  # Exiger au moins 3 signaux au lieu de 2
            bounce_signals["bounce_detected"] = True
        else:
            bounce_signals["bounce_detected"] = False

        # Vérifier la confirmation sur plusieurs timeframes
        if bounce_signals["bounce_detected"]:
            # Obtenir des données du timeframe supérieur
            higher_tf_data = self.data_fetcher.get_market_data(symbol)["secondary_timeframes"]

            if bounce_signals["bounce_detected"]:
```

```
        # Obtenir des données du timeframe supérieur
        market_data = self.data_fetcher.get_market_data(symbol)
        higher_tf_data = market_data.get("secondary_timeframes", {})

        # Vérifier si le timeframe supérieur confirme aussi un rebond
        if not self._check_higher_timeframe_confirmation(higher_tf_data):
            bounce_signals["bounce_detected"] = False
            bounce_signals["signals"].append("Non confirmé sur timeframe supérieur")
    # NOUVEAU: Vérifier la confirmation sur plusieurs timeframes
    tf_confirmations = self._check_higher_timeframe_confirmation(symbol)
    bounce_signals["multi_timeframe_confirmation"] = tf_confirmations

    # AMÉLIORÉ: Analyse de volume plus sophistiquée
    volume_analysis = self._analyze_volume_pattern(ohlcv)
    if volume_analysis["volume_spike"]:
        bounce_signals["signals"].append("Pic de volume haussier")
        bounce_signals["strength"] += 15
        bounce_signals["volume_ratio"] = volume_analysis["volume_ratio"]

        # Volume climax après une forte baisse (capitulation)
        if volume_analysis["capitulation"]:
            bounce_signals["signals"].append("Volume de capitulation détecté")
            bounce_signals["strength"] += 15

    # NOUVEAU: Calculer le score de confiance
    confidence_score = bounce_signals["strength"]

    # Ajuster en fonction des confirmations multi-timeframe
    confidence_score += tf_confirmations * 5

    # Pénaliser en cas de tendance fortement baissière sans volume de capitulation
```

```
    if trend_direction == "strong_bearish" and not volume_analysis.get("capitulation", False):
        confidence_score *= 0.7

    # Déterminer si un rebond est détecté avec des critères plus stricts
    bounce_signals["bounce_detected"] = (
        len(bounce_signals["signals"]) >= 3 and confidence_score >= strength_threshold
    )
    return bounce_signals

def _check_higher_timeframe_confirmation(self, higher_tf_data: Dict) -> bool:
    """
    Vérifie si les timeframes supérieurs confirment également un signal de rebond

    Args:
        higher_tf_data: Données des timeframes supérieurs

    Returns:
        True si confirmé, False sinon
    """
    # Par défaut, considérer comme confirmé si aucune donnée n'est disponible
    if not higher_tf_data:
        return True

    confirmation_count = 0
    timeframes_checked = 0

    # Parcourir les timeframes supérieurs (1h, 4h)
    for tf, tf_data in higher_tf_data.items():
        if "ohlcv" not in tf_data or tf_data["ohlcv"].empty:
            continue
```

```
timeframes_checked += 1
ohlcv = tf_data["ohlcv"]
indicators = tf_data.get("indicators", {})

# Vérifier le RSI
if "rsi" in indicators:
    rsi = indicators["rsi"]
    if len(rsi) >= 2:
        rsi_current = rsi.iloc[-1]
        rsi_prev = rsi.iloc[-2]

        if rsi_current > rsi_prev and rsi_current < 50:
            # RSI en hausse mais encore sous 50 = bon signe pour un rebond
            confirmation_count += 1

# Vérifier les bandes de Bollinger
if "bollinger" in indicators and "percent_b" in indicators["bollinger"]:
    percent_b = indicators["bollinger"]["percent_b"]
    if len(percent_b) >= 2:
        percent_b_current = percent_b.iloc[-1]
        percent_b_prev = percent_b.iloc[-2]

        if percent_b_current > percent_b_prev and percent_b_current < 0.5:
            # %B en hausse mais encore sous 0.5 = bon signe pour un rebond
            confirmation_count += 1

# Vérifier le pattern de chandelier
if len(ohlcv) >= 3:
    last_candle = ohlcv.iloc[-1]
    prev_candle = ohlcv.iloc[-2]
```

```
            if last_candle["close"] > last_candle["open"] and prev_candle["close"] <
prev_candle["open"]:
                # Bougie haussière après bougie baissière = bon signe pour un rebond
                confirmation_count += 1

        # Considérer comme confirmé si au moins la moitié des vérifications sont positives
        # et qu'au moins un timeframe a été vérifié
        return timeframes_checked > 0 and confirmation_count >= timeframes_checked / 2

    def _calculate_opportunity_score(self, bounce_signals: Dict, market_state: Dict,
                                     ohlcv: pd.DataFrame, indicators: Dict) -> int:
        """
        Calcule le score de l'opportunité de trading

        Args:
            bounce_signals: Signaux de rebond détectés
            market_state: État du marché
            ohlcv: DataFrame avec les données OHLCV
            indicators: Dictionnaire des indicateurs techniques

        Returns:
            Score de l'opportunité (0-100)
        """
        # Utiliser le moteur de scoring pour calculer le score
        score_data = {
            "bounce_signals": bounce_signals,
            "market_state": market_state,
            "ohlcv": ohlcv,
            "indicators": indicators
        }
```

```
    # Appeler le scoring engine et vérifier le résultat
    score_result = self.scoring_engine.calculate_score(score_data, "technical_bounce")

    # Vérifier si score_result est None ou ne contient pas la clé 'score'
    if score_result is None:
        # Logging pour diagnostic
        logger.error("Le scoring engine a retourné None au lieu d'un résultat valide")
        return 0  # Score par défaut si erreur

    # Vérifier si la clé 'score' existe dans score_result
    if "score" not in score_result:
        logger.error(f"Résultat du scoring incomplet: {score_result}")
        return 0  # Score par défaut si erreur

    return score_result["score"]



def _generate_reasoning(self, bounce_signals: Dict, market_state: Dict, score: int) -> str:
    """
    Génère une explication textuelle pour l'opportunité de trading

    Args:
        bounce_signals: Signaux de rebond détectés
        market_state: État du marché
        score: Score de l'opportunité

    Returns:
        Explication textuelle
    """
    signals_text = ", ".join(bounce_signals["signals"])
```

```
    reasoning = f"Opportunité de rebond technique détectée (score: {score}/100). "
    reasoning += f"Signaux: {signals_text}. "

    # Ajouter des détails sur l'état du marché
    if "details" in market_state:
        market_details = market_state["details"]

        if "rsi" in market_details:
            reasoning += f"RSI actuel: {market_details['rsi'].get('value', 'N/A'):.1f}. "

        if "bollinger" in market_details:
            reasoning += f"Volatilité: {market_details['bollinger'].get('bandwidth', 'N/A'):.3f}. "

        if "adx" in market_details:
            reasoning += f"Force de tendance (ADX): {market_details['adx'].get('value', 'N/A'):.1f}. "

    return reasoning
def _get_rsi_oversold_duration(self, rsi: pd.Series) -> int:
    """
    Calcule la durée pendant laquelle le RSI est resté en zone de survente

    Returns:
        Nombre de périodes consécutives en zone de survente
    """
    duration = 0
    for i in range(len(rsi)-1, -1, -1):
        if rsi.iloc[i] < RSI_OVERSOLD:
            duration += 1
        else:
            break
    return duration
```

```
def _analyze_volume_pattern(self, ohlcv: pd.DataFrame) -> Dict:
    """
    Analyse avancée des patterns de volume
    """
    if len(ohlcv) < 10:
        return {"volume_spike": False, "volume_ratio": 1.0}

    # Calculer la moyenne des volumes récents (sauf le dernier)
    avg_volume = ohlcv['volume'].iloc[-10:-1].mean()
    current_volume = ohlcv['volume'].iloc[-1]
    volume_ratio = current_volume / avg_volume if avg_volume > 0 else 1.0

    # Vérifier si le volume actuel est significativement plus élevé
    is_spike = volume_ratio > 2.0

    # Vérifier si c'est un volume de capitulation
    is_capitulation = False
    if is_spike and volume_ratio > 3.0:
        # Vérifier si le prix a chuté significativement avant ce volume
        price_drop = (ohlcv['close'].iloc[-3] - ohlcv['low'].iloc[-1]) / ohlcv['close'].iloc[-3]
        if price_drop > 0.05:  # Chute d'au moins 5%
            # Et si la clôture est plus haute que l'ouverture (rebond)
            if ohlcv['close'].iloc[-1] > ohlcv['open'].iloc[-1]:
                is_capitulation = True

    return {
        "volume_spike": is_spike,
        "volume_ratio": volume_ratio,
        "capitulation": is_capitulation
    }
```

```
def _analyze_price_structure(self, ohlcv: pd.DataFrame) -> Dict:
    """
    Analyse la structure de prix pour détecter les patterns de retournement
    """
    if len(ohlcv) < 20:
        return {"double_bottom": False}

    # Détecter double bottom (W-pattern)
    first_low = None
    second_low = None

    # Rechercher les deux derniers creux significatifs
    for i in range(len(ohlcv)-15, len(ohlcv)-1):
        if (ohlcv['low'].iloc[i] < ohlcv['low'].iloc[i-1] and
            ohlcv['low'].iloc[i] < ohlcv['low'].iloc[i+1]):
            if first_low is None:
                first_low = (i, ohlcv['low'].iloc[i])
            else:
                second_low = (i, ohlcv['low'].iloc[i])
                break

    if first_low and second_low:
        # Vérifier si les deux creux sont à des niveaux similaires (tolérance de 1%)
        price_diff = abs(first_low[1] - second_low[1]) / first_low[1]
        time_diff = second_low[0] - first_low[0]

        is_double_bottom = (price_diff < 0.01 and time_diff >= 5 and time_diff <= 20)

        return {"double_bottom": is_double_bottom}
```

```
    return {"double_bottom": False}

def _check_support_test(self, ohlcv: pd.DataFrame, indicators: Dict) -> Dict:
    """
    Vérifie si le prix teste un niveau de support important
    """
    if len(ohlcv) < 50:
        return {"support_tested": False}

    # Identifier les niveaux de support potentiels
    support_levels = []

    # 1. EMA 200 comme support dynamique
    if "ema" in indicators and "ema_200" in indicators["ema"]:
        ema200 = indicators["ema"]["ema_200"].iloc[-1]
        current_price = ohlcv['close'].iloc[-1]

        # Vérifier si le prix est proche de l'EMA 200
        if abs(current_price - ema200) / ema200 < 0.01:
            return {
                "support_tested": True,
                "support_level": ema200,
                "support_type": "EMA 200"
            }

    # 2. Détecter les niveaux de support statiques basés sur les creux précédents
    lows = []
    for i in range(20, len(ohlcv)-1):
        if (ohlcv['low'].iloc[i] < ohlcv['low'].iloc[i-1] and
            ohlcv['low'].iloc[i] < ohlcv['low'].iloc[i+1]):
            lows.append(ohlcv['low'].iloc[i])
```

```
    # Regrouper les niveaux proches
    if lows:
        current_price = ohlcv['close'].iloc[-1]
        for low in lows:
            if abs(current_price - low) / low < 0.02:
                return {
                    "support_tested": True,
                    "support_level": low,
                    "support_type": "Support historique"
                }

    return {"support_tested": False}
def _detect_market_trend(self, ohlcv: pd.DataFrame, indicators: Dict) -> str:
    """
    Détecte la direction et la force de la tendance actuelle du marché

    Args:
        ohlcv: DataFrame avec les données OHLCV
        indicators: Dictionnaire des indicateurs techniques

    Returns:
        Direction de la tendance ('strong_bullish', 'bullish', 'neutral', 'bearish', 'strong_bearish')
    """
    trend_scores = []  # Initialize the trend_scores list

    if len(ohlcv) < 20:
            return "neutral"  # Données insuffisantes

        # Méthode 1: Utiliser les EMA pour déterminer la tendance
    ema_data = indicators.get("ema", {})
```

```
if "ema_9" in ema_data and "ema_21" in ema_data and "ema_50" in ema_data:
        ema_short = ema_data["ema_9"].iloc[-1]
        ema_medium = ema_data["ema_21"].iloc[-1]
        ema_long = ema_data["ema_50"].iloc[-1]

        # Vérifier l'alignement des EMA - version moins stricte
        if ema_short > ema_medium > ema_long:
            # Tendance haussière
            return "bullish"  # Moins strict: "bullish" au lieu de "strong_bullish"
        elif ema_short < ema_medium < ema_long:
            # Tendance baissière
            current_price = ohlcv['close'].iloc[-1]
            # Moins strict: vérifier uniquement si le prix est sous l'EMA courte
            if current_price < ema_short:
                return "bearish"  # Moins strict: "bearish" au lieu de "strong_bearish"

    # Méthode 2: Utiliser l'ADX pour déterminer la force de la tendance - version moins stricte
adx_data = indicators.get("adx", {})
if "adx" in adx_data and "plus_di" in adx_data and "minus_di" in adx_data:
        adx = adx_data["adx"].iloc[-1]
        plus_di = adx_data["plus_di"].iloc[-1]
        minus_di = adx_data["minus_di"].iloc[-1]

        # Seuil ADX réduit pour être moins strict
        if adx > 20:  # Était 25, réduit à 20
            if plus_di > minus_di:
                return "bullish"  # Moins strict: "bullish" au lieu de "strong_bullish"
            else:
                return "bearish"  # Moins strict: "bearish" au lieu de "strong_bearish"

    # Méthode 3: Analyser la pente des prix récents - version moins stricte
```

```
recent_closes = ohlcv['close'].tail(10).values
if len(recent_closes) >= 10:
        # Calculer la pente linéaire
        x = np.arange(len(recent_closes))
        slope, _, _, _, _ = np.polyfit(x, recent_closes, 1, full=True)

        # Normaliser la pente par rapport au prix moyen
        avg_price = np.mean(recent_closes)
        norm_slope = slope[0] / avg_price * 100

        # Seuils réduits pour être moins stricts
        if norm_slope > 0.3:  # Était 0.5, réduit à 0.3
            return "bullish"
        elif norm_slope < -0.3:  # Était -0.5, réduit à -0.3
            return "bearish"


    # 4. NOUVEAU: Analyse des structures de prix
price_structure_trend = self._get_price_structure_trend(ohlcv)
trend_scores.append(price_structure_trend)

# 5. NOUVEAU: Analyse des volumes
volume_trend = self._get_volume_trend(ohlcv)
trend_scores.append(volume_trend)

# Calcul du score global pondéré
# Strong bearish: -2, Bearish: -1, Neutral: 0, Bullish: 1, Strong bullish: 2
weights = {
    "strong_bearish": -2,
    "bearish": -1,
    "neutral": 0,
```

```
    "bullish": 1,
    "strong_bullish": 2
}

# Pondération des méthodes (certaines sont plus fiables que d'autres)
method_weights = {
    "ema": 0.3,       # EMA a un poids important
    "adx": 0.25,      # ADX également
    "price_slope": 0.15,
    "price_structure": 0.15,
    "volume": 0.15
}

# Calcul du score final pondéré
weighted_score = 0
for i, trend in enumerate(trend_scores):
    method_name = ["ema", "adx", "price_slope", "price_structure", "volume"][i]
    weighted_score += weights.get(trend, 0) * method_weights.get(method_name, 0.1)

# Détermination finale de la tendance basée sur le score
if weighted_score >= 1.2:
    return "strong_bullish"
elif weighted_score >= 0.5:
    return "bullish"
elif weighted_score <= -1.2:
    return "strong_bearish"
elif weighted_score <= -0.5:
    return "bearish"
else:
    return "neutral"
```

```
def _check_higher_timeframe_confirmation(self, symbol: str) -> int:
    """
    Vérifie le nombre de timeframes supérieurs qui confirment le signal de rebond

    Args:
        symbol: Paire de trading

    Returns:
        Nombre de timeframes confirmant le signal (0-2)
    """
    # Dans un contexte de backtest, simuler une confirmation
    return 1  # Par défaut, considérer qu'un timeframe confirme

def _analyze_price_structure(self, ohlcv: pd.DataFrame) -> Dict:
    """
    Analyse la structure de prix pour détecter les patterns de retournement

    Args:
        ohlcv: DataFrame avec les données OHLCV

    Returns:
        Dictionnaire avec les patterns détectés
    """
    if len(ohlcv) < 20:
        return {"double_bottom": False}

    # Recherche simplifiée de double bottom
    is_double_bottom = False

    # Vérifier si les 5 dernières bougies montrent une reprise après un creux
```

```
        if ohlcv['low'].iloc[-5:-3].min() < ohlcv['low'].iloc[-10:-5].min() and ohlcv['close'].iloc[-1] > ohlcv['close'].iloc[-5]:
            is_double_bottom = True

        return {"double_bottom": is_double_bottom}

    def _check_support_test(self, ohlcv: pd.DataFrame, indicators: Dict) -> Dict:
        """
        Vérifie si le prix teste un niveau de support important

        Args:
            ohlcv: DataFrame avec les données OHLCV
            indicators: Dictionnaire des indicateurs techniques

        Returns:
            Dictionnaire avec les informations de test de support
        """
        if len(ohlcv) < 20:
            return {"support_tested": False}

        # Version simplifiée: vérifier si le prix est proche d'un creux récent
        recent_low = ohlcv['low'].iloc[-20:].min()
        current_close = ohlcv['close'].iloc[-1]

        # Si le prix actuel est à moins de 2% du creux récent
        if abs(current_close - recent_low) / recent_low < 0.02:
            return {
                "support_tested": True,
                "support_level": recent_low,
                "support_type": "Support récent"
            }
```

```
        return {"support_tested": False}

    def _analyze_volume_pattern(self, ohlcv: pd.DataFrame) -> Dict:
        """
        Analyse les patterns de volume

        Args:
            ohlcv: DataFrame avec les données OHLCV

        Returns:
            Dictionnaire avec les analyses de volume
        """
        if len(ohlcv) < 10:
            return {"volume_spike": False, "volume_ratio": 1.0, "capitulation": False}

        # Calculer la moyenne des volumes récents
        avg_volume = ohlcv['volume'].iloc[-10:-1].mean()
        current_volume = ohlcv['volume'].iloc[-1]

        # Calculer le ratio de volume
        volume_ratio = current_volume / avg_volume if avg_volume > 0 else 1.0

        # Vérifier s'il y a un pic de volume
        is_spike = volume_ratio > 2.0

        # Vérifier s'il y a une capitulation (forte baisse suivie d'un fort volume et d'un rebond)
        is_capitulation = False
        if is_spike and volume_ratio > 3.0:
            price_drop = (ohlcv['high'].iloc[-3] - ohlcv['low'].iloc[-1]) / ohlcv['high'].iloc[-3]
            if price_drop > 0.03 and ohlcv['close'].iloc[-1] > ohlcv['open'].iloc[-1]:
```

```
            is_capitulation = True

    return {
        "volume_spike": is_spike,
        "volume_ratio": volume_ratio,
        "capitulation": is_capitulation
    }

def _get_rsi_oversold_duration(self, rsi: pd.Series) -> int:
    """
    Calcule la durée pendant laquelle le RSI est resté en zone de survente

    Args:
        rsi: Série pandas contenant les valeurs du RSI

    Returns:
        Nombre de périodes en zone de survente
    """
    if rsi.empty:
        return 0

    # Compter le nombre de périodes où le RSI était en zone de survente
    oversold_count = 0

    # Parcourir les valeurs du RSI en partant de la dernière
    for i in range(len(rsi)-1, max(0, len(rsi)-10), -1):
        if rsi.iloc[i] < 30:  # RSI_OVERSOLD
            oversold_count += 1
        else:
            break  # Sortir dès qu'on trouve une valeur non survente
```

```
        return oversold_count

    def _get_ema_trend(self, indicators: Dict) -> str:
        """
        Analyse le positionnement des EMA pour déterminer la tendance
        """
        ema_data = indicators.get("ema", {})
        if "ema_9" in ema_data and "ema_21" in ema_data and "ema_50" in ema_data:
            ema_short = ema_data["ema_9"].iloc[-1]
            ema_medium = ema_data["ema_21"].iloc[-1]
            ema_long = ema_data["ema_50"].iloc[-1]

            # Alignement parfaitement haussier
            if ema_short > ema_medium > ema_long:
                return "strong_bullish"

            # Alignement partiellement haussier
            elif ema_short > ema_medium and ema_short > ema_long:
                return "bullish"

            # Alignement parfaitement baissier
            elif ema_short < ema_medium < ema_long:
                return "strong_bearish"

            # Alignement partiellement baissier
            elif ema_short < ema_medium and ema_short < ema_long:
                return "bearish"

        return "neutral"

    def _get_adx_trend(self, indicators: Dict) -> str:
```

```
    """
    Utilise l'ADX pour déterminer la force et la direction de la tendance
    """
    adx_data = indicators.get("adx", {})
    if "adx" in adx_data and "plus_di" in adx_data and "minus_di" in adx_data:
        adx = adx_data["adx"].iloc[-1]
        plus_di = adx_data["plus_di"].iloc[-1]
        minus_di = adx_data["minus_di"].iloc[-1]

        # Tendance forte (ADX > 25)
        if adx > 25:
            # Tendance haussière forte
            if plus_di > minus_di and plus_di > 25:
                return "strong_bullish"
            # Tendance baissière forte
            elif minus_di > plus_di and minus_di > 25:
                return "strong_bearish"

        # Tendance modérée (ADX entre 15 et 25)
        elif adx > 15:
            if plus_di > minus_di:
                return "bullish"
            elif minus_di > plus_di:
                return "bearish"

    return "neutral"

def _get_price_slope_trend(self, ohlcv: pd.DataFrame) -> str:
    """
    Analyse la pente des prix récents pour déterminer la tendance
    """
```

```
    if len(ohlcv) < 10:
        return "neutral"

    # Pente sur 10 périodes
    recent_closes = ohlcv['close'].tail(10).values
    x = np.arange(len(recent_closes))
    slope, _, _, _, _ = np.polyfit(x, recent_closes, 1, full=True)

    # Normaliser la pente par rapport au prix moyen
    avg_price = np.mean(recent_closes)
    norm_slope = slope[0] / avg_price * 100

    if norm_slope > 1.0:
        return "strong_bullish"
    elif norm_slope > 0.3:
        return "bullish"
    elif norm_slope < -1.0:
        return "strong_bearish"
    elif norm_slope < -0.3:
        return "bearish"

    return "neutral"

def _get_price_structure_trend(self, ohlcv: pd.DataFrame) -> str:
    """
    Analyse des structures de prix pour déterminer la tendance
    """
    if len(ohlcv) < 20:
        return "neutral"

    # Identifier les niveaux importants (hauts/bas)
```

```
        recent_highs = ohlcv['high'].rolling(5).max()
        recent_lows = ohlcv['low'].rolling(5).min()

        # Vérifier si les hauts/bas sont ascendants ou descendants
        higher_highs = recent_highs.iloc[-1] > recent_highs.iloc[-10]
        higher_lows = recent_lows.iloc[-1] > recent_lows.iloc[-10]
        lower_highs = recent_highs.iloc[-1] < recent_highs.iloc[-10]
        lower_lows = recent_lows.iloc[-1] < recent_lows.iloc[-10]

        # Structure haussière: hauts plus hauts ET bas plus hauts
        if higher_highs and higher_lows:
            return "strong_bullish"

        # Structure partiellement haussière
        elif higher_lows:
            return "bullish"

        # Structure baissière: hauts plus bas ET bas plus bas
        elif lower_highs and lower_lows:
            return "strong_bearish"

        # Structure partiellement baissière
        elif lower_highs:
            return "bearish"

        return "neutral"

    def _get_volume_trend(self, ohlcv: pd.DataFrame) -> str:
        """
        Analyse le volume pour confirmer la tendance des prix
        """
```

```
    if len(ohlcv) < 10 or 'volume' not in ohlcv.columns:
        return "neutral"

    # Calculer le volume moyen sur 10 périodes
    avg_volume = ohlcv['volume'].rolling(10).mean()

    # Comparer les 3 derniers volumes à la moyenne
    recent_vols = ohlcv['volume'].iloc[-3:].values
    recent_prices = ohlcv['close'].iloc[-3:].values

    # Prix en hausse + volume en hausse = confirmation haussière
    if recent_prices[-1] > recent_prices[0]:
        # Volume croissant avec les prix
        if recent_vols[-1] > avg_volume.iloc[-1] * 1.2:
            return "strong_bullish"
        elif recent_vols[-1] > avg_volume.iloc[-1]:
            return "bullish"

    # Prix en baisse + volume en hausse = confirmation baissière
    elif recent_prices[-1] < recent_prices[0]:
        # Volume croissant avec la baisse des prix
        if recent_vols[-1] > avg_volume.iloc[-1] * 1.2:
            return "strong_bearish"
        elif recent_vols[-1] > avg_volume.iloc[-1]:
            return "bearish"

    return "neutral"

================================================
File: crypto_trading_bot_CLAUDE/tests/test_api_connection.py
================================================
```

```
import unittest
import os
import sys

# Ajouter le répertoire parent au chemin de recherche
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

from core.api_connector import BinanceConnector

class TestAPIConnection(unittest.TestCase):
    def test_connection(self):
        connector = BinanceConnector()
        self.assertTrue(connector.test_connection())

if __name__ == "__main__":
    unittest.main()



================================================
File: crypto_trading_bot_CLAUDE/utils/backtest_engine.py
================================================
import pandas as pd
import numpy as np
from typing import Dict

def _simulate_trading(self, data: pd.DataFrame, strategy, initial_capital: float, symbol: str) -> Dict:
    # Utiliser des structures de données plus efficaces
    equity_history = np.zeros(len(data))
    equity_history[0] = initial_capital

    # Préallouer les tableaux pour les statistiques
```

```
    position_active = np.zeros(len(data), dtype=bool)

    # Apply strategy signals
    signals = strategy.generate_signals(data, symbol)
    position_active[signals] = True

    # Calculate returns using self and return results
    return {
        "equity": equity_history,
        "positions": position_active
    }

================================================
File: crypto_trading_bot_CLAUDE/utils/logger.py
================================================
# utils/logger.py
"""
Configuration du système de journalisation
"""
import os
import logging
from logging.handlers import RotatingFileHandler
from typing import Optional

from config.config import LOG_LEVEL, LOG_FORMAT, LOG_FILE, LOG_DIR

# Créer le répertoire de logs s'il n'existe pas
if not os.path.exists(LOG_DIR):
    os.makedirs(LOG_DIR)

def setup_logger(name: str, level: Optional[int] = None,
```

```
        log_format: Optional[str] = None, log_file: Optional[str] = None) -> logging.Logger:
"""
Configure un logger avec rotation des fichiers

Args:
    name: Nom du logger
    level: Niveau de journalisation
    log_format: Format des messages de log
    log_file: Chemin du fichier de log

Returns:
    Logger configuré
"""
# Utiliser les valeurs par défaut si non spécifiées
if level is None:
    level = LOG_LEVEL
if log_format is None:
    log_format = LOG_FORMAT
if log_file is None:
    log_file = LOG_FILE.replace('.log', f'_{name}.log')

# Créer et configurer le logger
logger = logging.getLogger(name)

# Éviter d'ajouter des handlers multiples
if not logger.handlers:
    logger.setLevel(level)

    # Formatter pour les messages de log
    formatter = logging.Formatter(log_format)
```

```
        # Handler pour la console
        console_handler = logging.StreamHandler()
        console_handler.setFormatter(formatter)
        logger.addHandler(console_handler)

        # Handler pour le fichier avec rotation
        file_handler = RotatingFileHandler(
            log_file,
            maxBytes=5*1024*1024,  # 5 MB
            backupCount=10
        )
        file_handler.setFormatter(formatter)
        logger.addHandler(file_handler)

    return logger


================================================
File: crypto_trading_bot_CLAUDE/utils/notification_service.py
================================================
# utils/notification_service.py
"""
Service de notification pour le bot de trading
"""
import logging
import smtplib
import requests
from email.mime.text import MIMEText
from typing import Dict, Optional, Union
from datetime import datetime
```

```
from config.config import (
    ENABLE_NOTIFICATIONS,
    NOTIFICATION_EMAIL,
    TELEGRAM_BOT_TOKEN,
    TELEGRAM_CHAT_ID,
    SMTP_SERVER,
    SMTP_PORT,
    SMTP_USER,
    SMTP_PASSWORD
)
from utils.logger import setup_logger

logger = setup_logger("notification_service")

class NotificationService:
    """
    Gère l'envoi de notifications par différents canaux
    """
    def __init__(self):
        self.enabled = ENABLE_NOTIFICATIONS
        self.last_notification_time = {}  # {channel: timestamp}
        self.notification_cooldown = 60  # secondes entre les notifications

    def send(self, message: str, level: str = "info") -> bool:
        """
        Envoie une notification par tous les canaux disponibles

        Args:
            message: Message à envoyer
            level: Niveau de la notification (info, warning, critical)
```

```
        Returns:
            True si au moins une notification a été envoyée, False sinon
        """
        if not self.enabled:
            logger.debug(f"Notifications désactivées, message ignoré: {message}")
            return False

        # Formater le message
        formatted_message = self._format_message(message, level)

        # Flag pour suivre si au moins une notification a été envoyée
        notification_sent = False

        # Email
        if NOTIFICATION_EMAIL and (level == "critical" or self._can_send_notification("email")):
            if self._send_email(NOTIFICATION_EMAIL, formatted_message, level):
                notification_sent = True
                self.last_notification_time["email"] = datetime.now()

        # Telegram
        if TELEGRAM_BOT_TOKEN and TELEGRAM_CHAT_ID and (level == "critical" or
self._can_send_notification("telegram")):
            if self._send_telegram(formatted_message):
                notification_sent = True
                self.last_notification_time["telegram"] = datetime.now()

        return notification_sent

    def _format_message(self, message: str, level: str) -> str:
        """
        Formate un message de notification
```

```
    Args:
        message: Message à formater
        level: Niveau de la notification

    Returns:
        Message formaté
    """
    prefix = {
        "info": "INFO",
        "warning": "⚠ AVERTISSEMENT",
        "critical": "🚨 ALERTE CRITIQUE"
    }.get(level, "INFO")

    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")

    return f"[{prefix}] [{timestamp}] {message}"

def _can_send_notification(self, channel: str) -> bool:
    """
    Vérifie si une notification peut être envoyée sur un canal

    Args:
        channel: Canal de notification

    Returns:
        True si une notification peut être envoyée, False sinon
    """
    if channel not in self.last_notification_time:
        return True
```

```
    time_since_last = (datetime.now() - self.last_notification_time[channel]).total_seconds()
    return time_since_last >= self.notification_cooldown

def _send_email(self, recipient: str, message: str, level: str) -> bool:
    """
    Envoie une notification par email

    Args:
        recipient: Adresse email du destinataire
        message: Message à envoyer
        level: Niveau de la notification

    Returns:
        True si l'email a été envoyé, False sinon
    """
    if not (SMTP_SERVER and SMTP_PORT and SMTP_USER and SMTP_PASSWORD):
        logger.error("Configuration SMTP incomplète, impossible d'envoyer l'email")
        return False

    try:
        subject = {
            "info": "Bot de Trading - Information",
            "warning": "Bot de Trading - Avertissement",
            "critical": "BOT DE TRADING - ALERTE CRITIQUE"
        }.get(level, "Bot de Trading - Notification")

        msg = MIMEText(message)
        msg['Subject'] = subject
        msg['From'] = SMTP_USER
        msg['To'] = recipient
```

```
            with smtplib.SMTP(SMTP_SERVER, SMTP_PORT) as server:
                server.starttls()
                server.login(SMTP_USER, SMTP_PASSWORD)
                server.send_message(msg)

            logger.info(f"Email envoyé à {recipient}")
            return True

        except Exception as e:
            logger.error(f"Erreur lors de l'envoi de l'email: {str(e)}")
            return False

    def _send_telegram(self, message: str) -> bool:
        """
        Envoie une notification par Telegram

        Args:
            message: Message à envoyer

        Returns:
            True si le message a été envoyé, False sinon
        """
        if not (TELEGRAM_BOT_TOKEN and TELEGRAM_CHAT_ID):
            logger.error("Configuration Telegram incomplète")
            return False

        try:
            url = f"https://api.telegram.org/bot{TELEGRAM_BOT_TOKEN}/sendMessage"
            data = {
                "chat_id": TELEGRAM_CHAT_ID,
                "text": message,
```

```
            "parse_mode": "HTML"
        }

        response = requests.post(url, data=data, timeout=10)

        if response.status_code == 200:
            logger.info("Message Telegram envoyé")
            return True
        else:
            logger.error(f"Erreur lors de l'envoi du message Telegram: {response.text}")
            return False

    except Exception as e:
        logger.error(f"Erreur lors de l'envoi du message Telegram: {str(e)}")
        return False
```

```
================================================
File: crypto_trading_bot_CLAUDE/utils/visualizer.py
================================================
# utils/visualizer.py
"""
Visualisation des performances et des trades
"""
import os
import json
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
from typing import Dict, List, Optional, Union
from datetime import datetime, timedelta
```

```
from config.config import DATA_DIR
from utils.logger import setup_logger

logger = setup_logger("visualizer")

class TradeVisualizer:
    """
    Crée des visualisations pour les trades et les performances
    """
    def __init__(self, position_tracker):
        self.position_tracker = position_tracker
        self.output_dir = os.path.join(DATA_DIR, "visualizations")

        # Créer le répertoire de sortie s'il n'existe pas
        if not os.path.exists(self.output_dir):
            os.makedirs(self.output_dir)

    def plot_equity_curve(self, days: int = 30) -> str:
        """
        Génère une courbe d'équité sur la période spécifiée

        Args:
            days: Nombre de jours à inclure

        Returns:
            Chemin du fichier image généré
        """
        # Récupérer les positions fermées
        closed_positions = self.position_tracker.get_closed_positions(limit=1000)

        # Filtrer sur la période demandée
```

```
start_date = datetime.now() - timedelta(days=days)
filtered_positions = [
    p for p in closed_positions
    if p.get("close_time") and p.get("close_time") > start_date
]

if not filtered_positions:
    logger.warning(f"Aucune position fermée dans les {days} derniers jours")
    return ""

# Trier par date de fermeture
sorted_positions = sorted(
    filtered_positions,
    key=lambda p: p.get("close_time", datetime.min)
)

# Créer des listes pour le graphique
dates = [p.get("close_time") for p in sorted_positions]
pnls = [p.get("pnl_absolute", 0) for p in sorted_positions]

# Calculer l'équité cumulée
from config.config import INITIAL_CAPITAL
equity = [INITIAL_CAPITAL]
for pnl in pnls:
    equity.append(equity[-1] + pnl)

equity = equity[1:]  # Supprimer le capital initial

# Créer le graphique
plt.figure(figsize=(12, 6))
plt.plot(dates, equity, 'b-', linewidth=2)
```

```
    plt.title(f'Courbe d\'Équité sur les {days} derniers jours')
    plt.xlabel('Date')
    plt.ylabel('Équité (USDT)')
    plt.grid(True)

    # Formater l'axe des dates
    plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m-%d'))
    plt.gcf().autofmt_xdate()

    # Ajouter des annotations
    initial_equity = INITIAL_CAPITAL
    final_equity = equity[-1]
    roi = (final_equity - initial_equity) / initial_equity * 100

    plt.annotate(f'ROI: {roi:.2f}%',
                 xy=(0.02, 0.95),
                 xycoords='axes fraction',
                 fontsize=12,
                 bbox=dict(boxstyle="round,pad=0.3", fc="white", ec="gray", alpha=0.8))

    # Sauvegarder le graphique
    filename = os.path.join(self.output_dir, f'equity_curve_{days}d.png')
    plt.savefig(filename)
    plt.close()

    logger.info(f"Courbe d'équité générée: {filename}")
    return filename

def plot_trade_analysis(self, days: int = 30) -> str:
    """
    Génère une analyse visuelle des trades sur la période spécifiée
```

```
Args:
    days: Nombre de jours à inclure

Returns:
    Chemin du fichier image généré
"""
# Récupérer les positions fermées
closed_positions = self.position_tracker.get_closed_positions(limit=1000)

# Filtrer sur la période demandée
start_date = datetime.now() - timedelta(days=days)
filtered_positions = [
    p for p in closed_positions
    if p.get("close_time") and p.get("close_time") > start_date
]

if not filtered_positions:
    logger.warning(f"Aucune position fermée dans les {days} derniers jours")
    return ""

# Créer une figure avec plusieurs sous-graphiques
fig, axs = plt.subplots(2, 2, figsize=(14, 10))
fig.suptitle(f'Analyse des Trades sur les {days} derniers jours', fontsize=16)

# 1. Distribution des profits/pertes
pnls = [p.get("pnl_percent", 0) for p in filtered_positions]
axs[0, 0].hist(pnls, bins=20, color='skyblue', edgecolor='black')
axs[0, 0].set_title('Distribution des Profits/Pertes (%)')
axs[0, 0].set_xlabel('Profit/Perte (%)')
axs[0, 0].set_ylabel('Nombre de Trades')
```

```
axs[0, 0].grid(True, alpha=0.3)

# 2. Performance par paire de trading
pairs = {}
for p in filtered_positions:
    symbol = p.get("symbol", "UNKNOWN")
    pnl = p.get("pnl_absolute", 0)

    if symbol not in pairs:
        pairs[symbol] = {'count': 0, 'pnl': 0}

    pairs[symbol]['count'] += 1
    pairs[symbol]['pnl'] += pnl

symbols = list(pairs.keys())
pnls = [pairs[s]['pnl'] for s in symbols]

# Tri par P&L
sorted_indices = sorted(range(len(pnls)), key=lambda i: pnls[i])
symbols = [symbols[i] for i in sorted_indices]
pnls = [pnls[i] for i in sorted_indices]

axs[0, 1].barh(symbols, pnls, color=['red' if p < 0 else 'green' for p in pnls])
axs[0, 1].set_title('P&L par Paire de Trading (USDT)')
axs[0, 1].set_xlabel('P&L (USDT)')
axs[0, 1].grid(True, alpha=0.3)

# 3. Ratio de réussite par jour de la semaine
day_performance = {i: {'wins': 0, 'losses': 0} for i in range(7)}

for p in filtered_positions:
```

```
    if p.get("close_time"):
        day = p.get("close_time").weekday()
        pnl = p.get("pnl_absolute", 0)

        if pnl >= 0:
            day_performance[day]['wins'] += 1
        else:
            day_performance[day]['losses'] += 1

days = ['Lundi', 'Mardi', 'Mercredi', 'Jeudi', 'Vendredi', 'Samedi', 'Dimanche']
win_rates = []

for i in range(7):
    wins = day_performance[i]['wins']
    losses = day_performance[i]['losses']
    total = wins + losses

    if total > 0:
        win_rates.append(wins / total * 100)
    else:
        win_rates.append(0)

axs[1, 0].bar(days, win_rates, color='orange')
axs[1, 0].set_title('Ratio de Réussite par Jour de la Semaine')
axs[1, 0].set_xlabel('Jour')
axs[1, 0].set_ylabel('Ratio de Réussite (%)')
axs[1, 0].set_ylim([0, 100])
axs[1, 0].grid(True, alpha=0.3)

# 4. Performance au fil du temps
dates = [p.get("close_time") for p in filtered_positions]
```

```
    pnls = [p.get("pnl_absolute", 0) for p in filtered_positions]

    # Trier par date
    sorted_indices = sorted(range(len(dates)), key=lambda i: dates[i])
    dates = [dates[i] for i in sorted_indices]
    pnls = [pnls[i] for i in sorted_indices]

    # Calculer le cumul
    cumulative_pnl = [pnls[0]]
    for pnl in pnls[1:]:
        cumulative_pnl.append(cumulative_pnl[-1] + pnl)

    axs[1, 1].plot(dates, cumulative_pnl, 'b-')
    axs[1, 1].set_title('P&L Cumulatif au Fil du Temps')
    axs[1, 1].set_xlabel('Date')
    axs[1, 1].set_ylabel('P&L Cumulatif (USDT)')
    axs[1, 1].grid(True, alpha=0.3)
    axs[1, 1].xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m-%d'))

    # Ajuster la mise en page
    plt.tight_layout(rect=[0, 0, 1, 0.95])

    # Sauvegarder le graphique
    filename = os.path.join(self.output_dir, f'trade_analysis_{days}d.png')
    plt.savefig(filename)
    plt.close()

    logger.info(f"Analyse des trades générée: {filename}")
    return filename

def plot_trade_history(self, symbol: str, data_fetcher, position_id: str) -> str:
```

```
"""
Génère un graphique montrant un trade spécifique avec entrée, sortie, et évolution du prix

Args:
    symbol: Paire de trading
    data_fetcher: Récupérateur de données
    position_id: ID de la position

Returns:
    Chemin du fichier image généré
"""
# Récupérer les données de la position
position = None

# Rechercher d'abord dans les positions fermées
for p in self.position_tracker.get_closed_positions():
    if p.get("id") == position_id:
        position = p
        break

# Si non trouvée, rechercher dans les positions ouvertes
if not position:
    position = self.position_tracker.get_position(position_id)

if not position:
    logger.error(f"Position {position_id} non trouvée")
    return ""

# Récupérer les données OHLCV
from config.config import PRIMARY_TIMEFRAME
```

```
# Déterminer la période à visualiser
entry_time = position.get("entry_time")
close_time = position.get("close_time")

if not entry_time:
    logger.error(f"Heure d'entrée non disponible pour la position {position_id}")
    return ""

# Si la position est toujours ouverte, utiliser l'heure actuelle
if not close_time:
    close_time = datetime.now()

# Ajouter une marge avant et après
start_time = entry_time - timedelta(hours=2)
end_time = close_time + timedelta(hours=2)

# Convertir en millisecondes pour l'API
start_ms = int(start_time.timestamp() * 1000)
end_ms = int(end_time.timestamp() * 1000)

# Récupérer les données
ohlcv = data_fetcher.get_ohlcv(
    symbol, PRIMARY_TIMEFRAME,
    start_time=start_ms, end_time=end_ms
)

if ohlcv.empty:
    logger.error(f"Données OHLCV non disponibles pour {symbol}")
    return ""

# Créer le graphique
```

```
plt.figure(figsize=(12, 6))

# Graphique des prix
plt.plot(ohlcv.index, ohlcv['close'], 'b-', linewidth=1.5)

# Marquer l'entrée
entry_price = position.get("entry_price")
plt.axhline(y=entry_price, color='g', linestyle='--', alpha=0.5)
plt.plot(entry_time, entry_price, 'go', markersize=8)

# Marquer la sortie si la position est fermée
if close_time and close_time != datetime.now():
    close_price = position.get("close_data", {}).get("fills", [{}])[0].get("price")
    if close_price:
        close_price = float(close_price)
        plt.plot(close_time, close_price, 'ro', markersize=8)

# Marquer le stop-loss et le take-profit
stop_loss = position.get("stop_loss_price")
take_profit = position.get("take_profit_price")

plt.axhline(y=stop_loss, color='r', linestyle='--', alpha=0.5)
plt.axhline(y=take_profit, color='g', linestyle='--', alpha=0.5)

# Ajouter des annotations
side = position.get("side")
pnl_percent = position.get("pnl_percent", 0)
pnl_absolute = position.get("pnl_absolute", 0)

title = f'Trade {position_id} - {symbol} ({side})'
if pnl_percent != 0:
```

```
    title += f' - P&L: {pnl_percent:.2f}% ({pnl_absolute:.2f} USDT)'

plt.title(title)
plt.xlabel('Date')
plt.ylabel('Prix')
plt.grid(True)

# Formater l'axe des dates
plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m-%d %H:%M'))
plt.gcf().autofmt_xdate()

# Ajouter une légende
plt.legend(['Prix', 'Entrée', 'Sortie', 'Stop-Loss', 'Take-Profit'],
        loc='best')

# Sauvegarder le graphique
filename = os.path.join(self.output_dir, f'trade_{position_id}.png')
plt.savefig(filename)
plt.close()

logger.info(f"Graphique du trade généré: {filename}")
return filename
```