Phase 3: Reinforcement Learning & AI Integration

Advanced ML-Powered Trading Enhancement

Enhance your SMC Trading Bot with Reinforcement Learning and AI-assisted pipelines using free-tier services.

Overview

Phase 3 adds:

- **Deep Q-Network (DQN)** Reinforcement Learning agent
- **AI-assisted signal generation** pipeline
- **Continuous learning** from trade outcomes
- **Google Colab** integration for free GPU training
- **Enhanced dashboard** with ML metrics

Part 1: RL Agent Implementation

Step 1: Create RL Agent Module

Create `backend/rl_agent.py`:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential, load_model, save_model
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam
from collections import deque
import random
import os

class DQNAgent:
    """
    Deep Q-Network Reinforcement Learning Agent
    Learns optimal trading actions from experience
    """

    def __init__(self, state_size=50, action_size=3, learning_rate=0.001):
        self.state_size = state_size
        self.action_size = action_size  # 0: HOLD, 1: BUY, 2: SELL

        # Hyperparameters
        self.memory = deque(maxlen=2000)
        self.gamma = 0.95            # Discount factor
```

```
        self.epsilon = 1.0           # Exploration rate
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.995
        self.learning_rate = learning_rate
        self.batch_size = 32
        self.train_start = 100

        # Build neural networks
        self.model = self._build_model()
        self.target_model = self._build_model()
        self.update_target_model()

        self.training_step = 0

    def _build_model(self):
        """Build neural network for Q-learning"""
        model = Sequential([
            Dense(128, activation='relu', input_dim=self.state_size),
            Dropout(0.2),
            Dense(128, activation='relu'),
            Dropout(0.2),
            Dense(64, activation='relu'),
            Dense(self.action_size, activation='linear')
        ])

        model.compile(
            loss='mse',
            optimizer=Adam(learning_rate=self.learning_rate)
        )

        return model

    def update_target_model(self):
        """Copy weights from model to target_model"""
        self.target_model.set_weights(self.model.get_weights())

    def remember(self, state, action, reward, next_state, done):
        """Store experience in replay memory"""
        self.memory.append((state, action, reward, next_state, done))

    def act(self, state):
        """
        Choose action using epsilon-greedy policy
        Returns: action index (0, 1, or 2)
        """
        if np.random.random() &lt;= self.epsilon:
            return random.randrange(self.action_size)  # Explore

        q_values = self.model.predict(state, verbose=0)
        return np.argmax(q_values[^0])  # Exploit

    def replay(self):
        """Train on batch from replay memory"""
        if len(self.memory) &lt; self.train_start:
            return
```

```
        # Sample mini-batch
        batch = random.sample(self.memory, min(len(self.memory), self.batch_size))

        states = np.array([item[^0][^0] for item in batch])
        actions = np.array([item[^1] for item in batch])
        rewards = np.array([item[^2] for item in batch])
        next_states = np.array([item[^3][^0] for item in batch])
        dones = np.array([item[^4] for item in batch])

        # Predict Q-values
        targets = self.model.predict(states, verbose=0)
        next_q_values = self.target_model.predict(next_states, verbose=0)

        # Update Q-values using Bellman equation
        for i in range(len(batch)):
            if dones[i]:
                targets[i][actions[i]] = rewards[i]
            else:
                targets[i][actions[i]] = rewards[i] + self.gamma * np.max(next_q_values[i

        # Train model
        self.model.fit(states, targets, epochs=1, verbose=0, batch_size=self.batch_size)

        # Decay epsilon
        if self.epsilon &gt; self.epsilon_min:
            self.epsilon *= self.epsilon_decay

        self.training_step += 1

        # Update target model periodically
        if self.training_step % 10 == 0:
            self.update_target_model()

    def save(self, filepath):
        """Save model to disk"""
        self.model.save(filepath)
        print(f"Model saved to {filepath}")

    def load(self, filepath):
        """Load model from disk"""
        if os.path.exists(filepath):
            self.model = load_model(filepath)
            self.update_target_model()
            print(f"Model loaded from {filepath}")
            return True
        return False

    def get_metrics(self):
        """Return current agent metrics"""
        return {
            'epsilon': self.epsilon,
            'memory_size': len(self.memory),
            'training_steps': self.training_step
        }
```

Step 2: Create AI Signal Pipeline

Create `backend/ai_pipeline.py`:

```
import numpy as np
import pandas as pd
from rl_agent import DQNAgent
from datetime import datetime

class AISignalPipeline:
    """
    AI-assisted trading signal generation pipeline
    Combines SMC analysis with RL agent decisions
    """

    def __init__(self, state_size=50, model_path='models/rl_agent.h5'):
        self.state_size = state_size
        self.action_size = 3  # HOLD, BUY, SELL
        self.model_path = model_path

        # Initialize RL agent
        self.agent = DQNAgent(state_size, self.action_size)
        self.agent.load(model_path)

        # Performance tracking
        self.episode_rewards = []
        self.current_episode_reward = 0

    def extract_state(self, df, signal_data):
        """
        Extract state vector from market data and SMC signals
        Returns: numpy array of shape (1, state_size)
        """
        features = []

        # Price-based features
        if len(df) &gt;= 20:
            close_prices = df['close'].values[-20:]

            # Returns
            returns = np.diff(close_prices) / close_prices[:-1]
            features.extend([
                returns.mean(),
                returns.std(),
                returns[-1] if len(returns) &gt; 0 else 0
            ])

            # Price momentum
            features.append((close_prices[-1] - close_prices[^0]) / close_prices[^0])

            # Volatility
            features.append(close_prices.std() / close_prices.mean())
        else:
            features.extend([0, 0, 0, 0, 0])

        # SMC signal features
```

```
        order_blocks = signal_data.get('order_blocks', [])
        choch = signal_data.get('choch')
        engulfing = signal_data.get('engulfing')

        # Order block proximity
        if order_blocks:
            latest_ob = order_blocks[-1]
            current_price = df['close'].iloc[-1]

            if latest_ob['type'] == 'BULLISH':
                ob_distance = (current_price - latest_ob['low']) / current_price
                features.append(ob_distance)
                features.append(1.0)  # Bullish OB flag
            else:
                ob_distance = (latest_ob['high'] - current_price) / current_price
                features.append(ob_distance)
                features.append(0.0)  # Bearish OB flag
        else:
            features.extend([0, 0.5])

        # CHoCH signal
        if choch:
            features.append(1.0 if 'BULLISH' in choch['type'] else -1.0)
        else:
            features.append(0.0)

        # Engulfing pattern
        if engulfing:
            features.append(engulfing['strength'])
            features.append(1.0 if 'BULLISH' in engulfing['type'] else -1.0)
        else:
            features.extend([0, 0])

        # SMC confidence from original signal
        features.append(signal_data.get('confidence', 0.5))

        # Pad or truncate to state_size
        features = np.array(features)
        if len(features) &lt; self.state_size:
            features = np.pad(features, (0, self.state_size - len(features)))
        else:
            features = features[:self.state_size]

        return features.reshape(1, self.state_size)

    def get_action(self, state):
        """Get action from RL agent"""
        action_idx = self.agent.act(state)
        actions = ['HOLD', 'BUY', 'SELL']
        return actions[action_idx], action_idx

    def calculate_reward(self, trade_result):
        """
        Calculate reward for RL agent based on trade outcome
        Positive reward for profit, negative for loss
        """
```

```
        if not trade_result:
            return 0

        pnl = trade_result.get('pnl', 0)
        risk = trade_result.get('risk', 1)

        # Normalize reward by risk
        reward = pnl / risk

        # Bonus for maintaining good risk/reward
        if pnl &gt; 0:
            reward += 0.5  # Bonus for winning trade
        elif pnl &lt; 0:
            reward -= 0.3  # Penalty for losing trade

        return reward

    def learn_from_trade(self, state, action_idx, trade_result, next_state, done):
        """Update RL agent based on trade outcome"""
        reward = self.calculate_reward(trade_result)

        self.agent.remember(state, action_idx, reward, next_state, done)
        self.agent.replay()

        self.current_episode_reward += reward

        if done:
            self.episode_rewards.append(self.current_episode_reward)
            self.current_episode_reward = 0

    def get_signal(self, df, smc_signal):
        """
        Generate enhanced trading signal using AI
        Returns: dict with action, confidence, and state
        """
        state = self.extract_state(df, smc_signal)
        action, action_idx = self.get_action(state)

        # Calculate combined confidence
        smc_confidence = smc_signal.get('confidence', 0.5)

        # Get Q-values for confidence estimation
        q_values = self.agent.model.predict(state, verbose=0)[^0]
        max_q = np.max(q_values)
        min_q = np.min(q_values)
        q_range = max_q - min_q if max_q != min_q else 1

        # Normalize Q-value to 0-1
        ai_confidence = (q_values[action_idx] - min_q) / q_range

        # Combined confidence (weighted average)
        combined_confidence = 0.6 * smc_confidence + 0.4 * ai_confidence

        return {
            'action': action,
            'action_idx': action_idx,
```

```
            'confidence': combined_confidence,
            'ai_confidence': ai_confidence,
            'smc_confidence': smc_confidence,
            'state': state,
            'q_values': q_values.tolist()
        }

    def save_model(self):
        """Save RL agent model"""
        self.agent.save(self.model_path)

    def get_performance_metrics(self):
        """Get AI pipeline performance metrics"""
        metrics = self.agent.get_metrics()

        if self.episode_rewards:
            metrics['avg_episode_reward'] = np.mean(self.episode_rewards[-100:])
            metrics['total_episodes'] = len(self.episode_rewards)

        return metrics
```

Step 3: Create Enhanced Strategy

Create `backend/enhanced_strategy.py`:

```
from smc_strategy import SMCStrategy
from ai_pipeline import AISignalPipeline
import pandas as pd

class EnhancedSMCStrategy(SMCStrategy):
    """
    Enhanced SMC Strategy with AI/RL integration
    Combines traditional SMC with reinforcement learning
    """

    def __init__(self, risk_params):
        super().__init__(risk_params)
        self.ai_pipeline = AISignalPipeline()
        self.trade_states = {}  # Store states for learning

    def generate_signal(self, df, symbol):
        """
        Generate enhanced signal using both SMC and AI
        """
        # Get base SMC signal
        smc_signal = super().generate_signal(df, symbol)

        if not smc_signal:
            return None

        # Get AI-enhanced signal
        ai_signal = self.ai_pipeline.get_signal(df, smc_signal)

        # Combine signals
        enhanced_signal = {
```

```
            **smc_signal,
            'ai_action': ai_signal['action'],
            'ai_confidence': ai_signal['ai_confidence'],
            'combined_confidence': ai_signal['confidence'],
            'q_values': ai_signal['q_values']
        }

        # Store state for later learning
        self.trade_states[symbol] = {
            'state': ai_signal['state'],
            'action_idx': ai_signal['action_idx']
        }

        # Override direction based on AI if confidence high enough
        if ai_signal['confidence'] &gt; 0.7:
            if ai_signal['action'] == 'BUY':
                enhanced_signal['direction'] = 'BUY'
            elif ai_signal['action'] == 'SELL':
                enhanced_signal['direction'] = 'SELL'
            else:  # HOLD
                enhanced_signal['direction'] = 'NEUTRAL'

        return enhanced_signal

    def record_trade_outcome(self, symbol, trade_result, next_df):
        """
        Learn from trade outcome
        Called when trade is closed
        """
        if symbol not in self.trade_states:
            return

        prev_state_data = self.trade_states[symbol]

        # Extract next state
        if next_df is not None and len(next_df) &gt; 0:
            smc_signal = super().generate_signal(next_df, symbol)
            if smc_signal:
                next_state = self.ai_pipeline.extract_state(next_df, smc_signal)
            else:
                next_state = prev_state_data['state']
        else:
            next_state = prev_state_data['state']

        # Learn from experience
        self.ai_pipeline.learn_from_trade(
            state=prev_state_data['state'],
            action_idx=prev_state_data['action_idx'],
            trade_result=trade_result,
            next_state=next_state,
            done=True
        )

        # Clean up
        del self.trade_states[symbol]
```

```
    def save_models(self):
        """Save AI models"""
        self.ai_pipeline.save_model()

    def get_ai_metrics(self):
        """Get AI performance metrics"""
        return self.ai_pipeline.get_performance_metrics()
```

Part 2: Continuous Learning Pipeline

Create Training Pipeline

Create `backend/training_pipeline.py`:

```
import schedule
import time
from datetime import datetime, timedelta
import pandas as pd
import numpy as np

class ContinuousLearningPipeline:
    """
    Continuous model retraining pipeline
    Automatically improves models based on recent trading data
    """

    def __init__(self, strategy, delta_client):
        self.strategy = strategy
        self.delta_client = delta_client
        self.training_interval_hours = 24
        self.last_training = datetime.now()
        self.min_trades_for_training = 20

    def collect_recent_trades(self, hours=24):
        """Collect recent closed trades for training"""
        trades = self.strategy.closed_trades

        if not trades:
            return []

        # Filter trades from last N hours
        cutoff_time = datetime.now() - timedelta(hours=hours)

        recent_trades = [
            t for t in trades
            if datetime.fromisoformat(t.get('closed_at', '')) &gt; cutoff_time
        ]

        return recent_trades

    def prepare_training_data(self, trades):
        """
        Prepare training data from trade history
```

```
        Returns: features (X) and labels (y)
        """
        X = []
        y = []

        for trade in trades:
            if 'features' not in trade:
                continue

            features = trade['features']
            # Label: 1 for profitable, 0 for loss
            label = 1 if trade.get('pnl', 0) &gt; 0 else 0

            X.append(features)
            y.append(label)

        return np.array(X), np.array(y)

    def retrain_models(self):
        """
        Retrain RL agent based on recent performance
        """
        print(f"\n{'='*50}")
        print(f"🔄 Starting continuous learning session")
        print(f"{'='*50}\n")

        # Collect recent trades
        recent_trades = self.collect_recent_trades(hours=self.training_interval_hours)

        print(f"📊 Collected {len(recent_trades)} recent trades")

        if len(recent_trades) &lt; self.min_trades_for_training:
            print(f"⚠️  Insufficient trades ({len(recent_trades)} &lt; {self.min_trades_f
            print("   Skipping retraining")
            return

        # Calculate performance metrics
        profitable_trades = sum(1 for t in recent_trades if t.get('pnl', 0) &gt; 0)
        win_rate = profitable_trades / len(recent_trades)
        avg_pnl = np.mean([t.get('pnl', 0) for t in recent_trades])

        print(f"📈 Performance Summary:")
        print(f"   Win Rate: {win_rate*100:.1f}%")
        print(f"   Avg PnL: ${avg_pnl:.2f}")

        # Retrain RL agent
        print("\n🧠 Retraining RL agent...")

        for trade in recent_trades:
            if 'state' in trade and 'action_idx' in trade:
                reward = trade.get('pnl', 0) / max(trade.get('risk', 1), 0.01)

                self.strategy.ai_pipeline.agent.remember(
                    state=trade['state'],
                    action=trade['action_idx'],
                    reward=reward,
```

```
                    next_state=trade.get('next_state', trade['state']),
                    done=True
                )

        # Perform multiple replay sessions
        for _ in range(10):
            self.strategy.ai_pipeline.agent.replay()

        # Save updated models
        self.strategy.save_models()

        self.last_training = datetime.now()

        print(f"\n✓ Retraining complete at {self.last_training}")
        print(f"   Next training in {self.training_interval_hours} hours")
        print(f"{'='*50}\n")

    def schedule_training(self):
        """Schedule periodic retraining"""
        schedule.every(self.training_interval_hours).hours.do(self.retrain_models)

        print(f"🔄 Scheduled continuous learning every {self.training_interval_hours} hour

        while True:
            schedule.run_pending()
            time.sleep(3600)  # Check every hour

    def manual_retrain(self):
        """Trigger manual retraining"""
        print("🔄 Manual retraining triggered")
        self.retrain_models()
```

Part 3: Google Colab Training

Create Colab Notebook

Create `training/google_colab_training.ipynb`:

```
# SMC Trading Bot - Model Training on Google Colab
# Run this notebook to train models using free GPU

# Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')

# Install dependencies
!pip install tensorflow pandas numpy scikit-learn

# Download historical data
import requests
import pandas as pd
import numpy as np
```

```
def fetch_training_data(symbol='BTCUSD', days=365):
    """Fetch historical data for training"""
    # Use public API to get historical data
    url = f"https://api.delta.exchange/v2/history/candles"

    end_time = int(time.time())
    start_time = end_time - (days * 24 * 60 * 60)

    params = {
        'symbol': symbol,
        'resolution': '15',
        'start': start_time,
        'end': end_time
    }

    response = requests.get(url, params=params)
    data = response.json()

    df = pd.DataFrame(
        data['result'],
        columns=['time', 'open', 'high', 'low', 'close', 'volume']
    )

    return df

# Fetch data for BTC and ETH
print("Fetching training data...")
btc_data = fetch_training_data('BTCUSD', days=365)
eth_data = fetch_training_data('ETHUSD', days=365)

print(f"BTC data: {len(btc_data)} candles")
print(f"ETH data: {len(eth_data)} candles")

# Train RL agent
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam

def build_model(state_size=50, action_size=3):
    model = Sequential([
        Dense(128, activation='relu', input_dim=state_size),
        Dropout(0.2),
        Dense(128, activation='relu'),
        Dropout(0.2),
        Dense(64, activation='relu'),
        Dense(action_size, activation='linear')
    ])

    model.compile(
        loss='mse',
        optimizer=Adam(learning_rate=0.001)
    )

    return model

model = build_model()
```

```
# Generate training samples
# (Add your feature extraction logic here)

# Train model
print("Training model...")
history = model.fit(
    X_train, y_train,
    epochs=50,
    batch_size=32,
    validation_split=0.2,
    verbose=1
)

# Save model
model.save('/content/drive/MyDrive/smc_models/rl_agent.h5')
print("Model saved to Google Drive!")

# Plot training history
import matplotlib.pyplot as plt

plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.legend()
plt.title('Model Training History')
plt.show()
```

Part 4: Integration with Main Bot

Update `backend/app.py`

Add RL integration:

```
# Import enhanced strategy
from enhanced_strategy import EnhancedSMCStrategy
from training_pipeline import ContinuousLearningPipeline

# Replace SMC strategy with enhanced version
smc_strategy = EnhancedSMCStrategy(RISK_PARAMS)

# Initialize training pipeline
training_pipeline = ContinuousLearningPipeline(smc_strategy, delta_client)

# Add new API endpoints
@app.route('/api/ai/metrics', methods=['GET'])
def get_ai_metrics():
    """Get AI/RL performance metrics"""
    metrics = smc_strategy.get_ai_metrics()
    return jsonify(metrics)

@app.route('/api/ai/retrain', methods=['POST'])
def trigger_retrain():
    """Manually trigger model retraining"""
```

```
    training_pipeline.manual_retrain()
    return jsonify({'status': 'Retraining initiated'})

# Start training pipeline in background
import threading
training_thread = threading.Thread(
    target=training_pipeline.schedule_training,
    daemon=True
)
training_thread.start()
```

Part 5: Enhanced Dashboard

Update Dashboard with AI Metrics

Add to `frontend/static/js/app.js`:

```
// Fetch AI metrics
async function fetchAIMetrics() {
    const response = await fetch(`${API_BASE}/ai/metrics`);
    return await response.json();
}

// Update AI metrics display
async function updateAIMetrics() {
    const metrics = await fetchAIMetrics();

    document.getElementById('ai-epsilon').textContent =
        metrics.epsilon.toFixed(3);
    document.getElementById('ai-memory-size').textContent =
        metrics.memory_size;
    document.getElementById('ai-training-steps').textContent =
        metrics.training_steps;

    if (metrics.avg_episode_reward) {
        document.getElementById('ai-avg-reward').textContent =
            metrics.avg_episode_reward.toFixed(2);
    }
}

// Trigger manual retraining
async function triggerRetraining() {
    const response = await fetch(`${API_BASE}/ai/retrain`, {
        method: 'POST'
    });
    const result = await response.json();
    alert(result.status);
}

// Update every 10 seconds
setInterval(updateAIMetrics, 10000);
```

Add HTML for AI metrics:

```
<div>
    <h3>🤖 AI Performance</h3>

    <div>
        <div>
            <p>Exploration Rate</p>
            <p>0.500</p>
        </div>

        <div>
            <p>Memory Size</p>
            <p>0</p>
        </div>

        <div>
            <p>Training Steps</p>
            <p>0</p>
        </div>

        <div>
            <p>Avg Reward</p>
            <p>0.00</p>
        </div>
    </div>

    &lt;button onclick="triggerRetraining()"
            class="mt-4 w-full bg-purple-600 hover:bg-purple-700
                   text-white px-4 py-2 rounded"&gt;
        🔄 Retrain Now
    &lt;/button&gt;
</div>
```

Part 6: Deployment

Update Requirements

Add to `backend/requirements.txt`:

```
tensorflow==2.15.0
```

Rebuild and Deploy

```
# Local
docker-compose down
docker-compose build --no-cache
docker-compose up -d
```

```
# VPS (via git push - GitHub Actions will deploy)
git add .
git commit -m "Add Phase 3: RL/AI integration"
git push origin main
```

Part 7: Monitoring AI Performance

Track RL Agent Metrics

Monitor:

- **Epsilon:** Exploration vs exploitation balance
- **Memory Size:** Experiences stored
- **Training Steps:** Number of training iterations
- **Average Reward:** Performance indicator

Expected Behavior

- Epsilon starts at 1.0 (full exploration)
- Gradually decreases to 0.01 (mostly exploitation)
- Memory fills up to 2000 experiences
- Average reward should increase over time

Summary

✅ **Phase 3 Complete** - AI/RL integrated
✅ DQN agent learning from trades
✅ Continuous learning pipeline active
✅ Google Colab ready for GPU training
✅ Enhanced dashboard with AI metrics

Free Services Used:

- TensorFlow (open-source)
- Google Colab (free GPU)
- All running on Oracle Cloud free tier

Next: Monitor AI performance and fine-tune hyperparameters!

[71][72][^75]

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16]

⁂

1. https://help.ovhcloud.com/csm/en-vps-deploy-website-github-actions?id=kb_article_view&sysparm_article=KB0066184
2. https://deltaexchangeindia.freshdesk.com/support/solutions/articles/80001181136-getting-started-with-delta-exchange-api-copilot
3. https://docs.docker.com/reference/compose-file/services/
4. https://www.hostinger.com/support/deploy-to-hostinger-vps-using-github-actions/
5. https://www.youtube.com/watch?v=TSzz-Fllq14
6. https://learn.microsoft.com/en-us/dotnet/architecture/microservices/multi-container-microservice-net-applications/multi-container-applications-docker-compose
7. https://blog.ando.ai/posts/github-actions-vps-deployment/
8. https://www.youtube.com/watch?v=O5i5DJUAap4
9. https://docs.docker.com/compose/how-tos/multiple-compose-files/
10. https://docs.github.com/actions/deployment/about-deployments/deploying-with-github-actions
11. https://www.delta.exchange/support/solutions/articles/80001153884-your-complete-api-faq-guide-everything-you-need-to-know
12. https://www.profitaddaweb.com/2025/04/delta-exchange-api-in-python.html
13. https://stackoverflow.com/questions/49191304/how-to-deal-with-multiple-services-inside-a-docker-compose-yml
14. https://gist.github.com/danielwetan/4f4db933531db5dd1af2e69ec8d54d8a
15. https://www.youtube.com/watch?v=RoVeASkeMpc
16. https://matthiasnoback.nl/2018/03/defining-multiple-similar-services-with-docker-compose/