Eye Preference Evolution System

概要

視線追跡データを利用したユーザー嗜好推定と対話型進化計算を組み合わせた画像生成システム。システムは以下の特徴を持つ：

- **視線ベースの嗜好推定**: 視線追跡データから機械学習モデルがユーザーの好みの画像を推定
- **手動評価モード**: 従来型の明示的な画像選択による対話型進化計算
- **画像生成**: Stable Diffusion XL Turboを使用したテキストからの画像生成
- **進化的アルゴリズム**: ユーザーの好みに基づいて画像の潜在空間を探索

システム構成

プロジェクトは主に3つの部分から構成される：

1. **フロントエンドアプリケーション**: Electron/Reactベースのデスクトップアプリ
2. **バックエンドモデルサービス**: 画像生成と視線データ解析を行うPython/FastAPIサーバー
3. **アイトラッカー接続**: Tobiiアイトラッカーからデータを取得するWebSocketサーバー

機能

- テキストプロンプトから初期画像セットを生成
- 評価方式の選択（視線ベース/手動選択）
- 画像生成と評価の繰り返しによる進化的探索
- 進化中の画像、評価データの保存
- 視線データ収集と分析
- 複数のモデルタイプ（LSTM/Transformerベース）による視線解析

詳細な動作環境構築方法

前提条件

- Docker と Docker Compose (バージョン19.03以上推奨)
- NVIDIA GPU (CUDA 11.4以上)
- NVIDIA Container Toolkit (nvidia-docker2)
- Tobiiアイトラッカー
- Windows 10/11またはUbuntu 20.04/22.04

システム要件

- RAM: 最小16GB、推奨32GB
- GPU: VRAM 8GB以上、推奨12GB以上
- CPU: 4コア以上
- ストレージ: 20GB以上の空き容量

環境構築の詳細手順

1. 基本ソフトウェアのインストール

Docker & Docker Composeのインストール

Windows環境での準備:

1. Docker Desktop for Windows をインストール
2. WSL2の設定を有効にする
3. NVIDIAデバイスを有効にするためのDockerの設定を行う

2. Conda環境のセットアップ（アイトラッカー用）

```
# Minicondaのインストール
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh -O
miniconda.sh
bash miniconda.sh -b -p $HOME/miniconda
echo 'export PATH="$HOME/miniconda/bin:$PATH"' >> ~/.bashrc
source ~/.bashrc

# アイトラッカー用の環境を作成
cd eye_tracker
conda env create -f environment.yml
```

3. リポジトリのクローンと初期設定

```
git clone https://github.com/mmmsmm16/eye-preference-IEC.git
cd eye-preference-IEC

# 必要に応じて設定ファイルを編集
# アイトラッカーのIPアドレス設定など
nano eye_tracker/src/websocket_server.py  # ホスト設定を変更する場合
```

セットアップとビルド

1. リポジトリのクローン

```
git clone https://github.com/mmmsmm16/eye-preference-IEC.git
cd eye-preference-IEC
```

2. `.env`ファイルの設定（必要に応じて）

```
# .envファイルを作成
cat > .env << EOL
# 環境変数の設定
MODEL_API_URL=http://localhost:8000
```

```
EYE_TRACKER_URL=ws://localhost:8765
DEBUG=true
EOL
```

3. Dockerコンテナをビルドして起動

```
# ビルドオプションを指定してDockerを起動
docker-compose up --build

# バックグラウンドで実行する場合
docker-compose up -d --build

# ログを確認する場合
docker-compose logs -f
```

4. アイトラッカーを接続（視線ベース評価を使用する場合）

```
# 別のターミナルで実行
cd eye_tracker
conda activate eye_tracker
python src/websocket_server.py
```

または、Windowsの場合：

```
cd eye_tracker
start_eye_tracker.bat
```

5. 動作確認

Electronアプリケーションが自動的に起動する。

サービスの稼働確認は以下のURLでも行える：

- バックエンドAPI: http://localhost:8000/health
- アイトラッカーWebSocket: ws://localhost:8765 (WebSocketクライアントで確認) 実際にはプロジェクト固有のモデルファイルへのリンクを提供

```
wget -O models/lstm/method1/model.pth https://example.com/path/to/lstm_model.pth
```

セットアップとビルド

1. リポジトリのクローン

```
git clone https://github.com/yourusername/eye-preference-evolution.git
cd eye-preference-evolution
```

2. .envファイルの設定（必要に応じて）

```
# .envファイルを作成
cat > .env << EOL
# 環境変数の設定
MODEL_API_URL=http://localhost:8000
EYE_TRACKER_URL=ws://localhost:8765
DEBUG=true
EOL
```

3. Dockerコンテナをビルドして起動

```
# ビルドオプションを指定してDockerを起動
docker-compose up --build

# バックグラウンドで実行する場合
docker-compose up -d --build

# ログを確認する場合
docker-compose logs -f
```

4. アイトラッカーを接続（視線ベース評価を使用する場合）

```
# 別のターミナルで実行
cd eye_tracker
conda activate eye_tracker
python src/websocket_server.py
```

または、Windowsの場合：

```
cd eye_tracker
start_eye_tracker.bat
```

5. 動作確認

Electronアプリケーションが自動的に起動します。

サービスの稼働確認は以下のURLでも行えます：

- バックエンドAPI: http://localhost:8000/health

- アイトラッカーWebSocket: ws://localhost:8765 (WebSocketクライアントで確認)

詳細な利用方法

アプリケーションの起動と実行

1. **システム起動**:
    - すべてのサービスが起動していることを確認
    - バックエンドサーバーの準備完了確認: `http://localhost:8000/health` にアクセスして `{"status":"healthy"}`が返ることを確認
    - アイトラッカーの接続確認: アプリケーション起動後、「Eye Tracker: Connected」と表示されることを確認
2. **スタート画面での設定**:
    - **評価方法**: 「Manual Evaluation」(手動評価)または「Gaze-based Evaluation」(視線ベース評価)を選択
    - **配置設定**:
        - 「Fixed placement」: 選択された画像が常に左上に配置されます
        - 「Random placement」: 選択された画像がランダムな位置に配置されます
    - **生成プロンプト**: 画像生成のための指示文（例: "a beautiful mountain landscape with snow and trees"）
    - **ネガティブプロンプト**: 生成から除外したい要素（例: "blurry, low quality, bad anatomy"）
    - **モデル選択** (視線ベース評価のみ):
        - LSTM Method 1: 生データLSTMベースの予測モデル
        - LSTM Method 2: 注視点データLSTMモデル
        - Transformer Method 1: 生データTransformerベースモデル
        - Transformer Method 2: 注視点データTransformerモデル
3. **評価セッションの操作**:
    - **視線ベース評価モード**:
        - 画像を自然に見て、好みの画像により注視してください
        - 視線データが十分収集されたら、**スペースキー**（左クリック）を押して次の世代に進む
        - 満足のいく画像が得られたら、**Enterキー**（右クリック）を押して終了
        - デバッグモードでは視線位置が表示されます（Ctrl+Dで切り替え）
    - **手動評価モード**:
        - **スペースキー**を押すと選択画面に移行
        - 好みの画像をクリックして選択
        - 「Continue Evolution」ボタンで次の世代に進む
        - 「Select and Finish」ボタンで終了
4. **結果の確認**:
    - 最終選択画像が表示されます
    - プロンプト情報が表示されます
    - 視線ベース評価の場合は予測の信頼度（%）も表示

- 「Exit to Start Screen」ボタンで新しいセッションを開始

キーボードショートカット

キー	機能
スペースキー	現在の画像を評価し、次の世代へ進む
Enter	現在の画像を最終選択として終了（視線ベース評価時）
Ctrl+D	デバッグモードの切り替え（視線位置表示）

パラメータのカスタマイズ

フロントエンド設定の変更

フロントエンドの主要なパラメータは `app/src/components/App.js` で定義されています：

```
// アプリケーションのフェーズを定義
const PHASES = {
  SELECTION: 'selection',
  CHOOSING: 'choosing',
  PROCESSING: 'processing',
  CONFIRMING: 'confirming',
  FINAL_RESULT: 'final_result'
};

// 利用可能なモデルの定義
const AVAILABLE_MODELS = [
  { id: 'lstm_method1', name: 'LSTM Method 1', type: 'lstm' },
  { id: 'lstm_method2', name: 'LSTM Method 2', type: 'lstm' },
  { id: 'transformer_method1', name: 'Transformer Method 1', type: 'transformer'
},
  { id: 'transformer_method2', name: 'Transformer Method 2', type: 'transformer' }
];
```

これらを変更することで、使用可能なモデルや表示名を変更

バックエンド設定の変更

バックエンドのパラメータは `model/src/api/server.py` で定義：

```
# 設定のデフォルト値
MODEL_CONFIG = {
    "model_dir": {
        "lstm": {
            "method1": Path("/app/src/models/lstm/method1"),
            "method2": Path("/app/src/models/lstm/method2")
        },
        "transformer": {
```

```
            "method1": Path("/app/src/models/transformer/method1"),
            "method2": Path("/app/src/models/transformer/method2")
        }
    },
    "device": torch.device("cuda" if torch.cuda.is_available() else "cpu"),
    "batch_size": 1,
    "default_method": "method1",
}
```

また、画像生成のパラメータは `model/src/services/stable_diffusion_service.py` で調整：

```
def __init__(self):
    self.model_id = "stabilityai/sdxl-turbo"
    # 他のパラメータ
    self.current_sigma = 1.0  # 突然変異の強度
```

画像生成の変異強度は以下のメソッドで調整される：

```
# 突然変異強度を更新
self.current_sigma *= 0.9  # 世代ごとに減衰率を調整可能
if self.current_sigma <= 0.5:
    self.current_sigma = 0.5  # 最小変異強度
```

アイトラッカー設定の変更

アイトラッカーの設定は `eye_tracker/src/websocket_server.py` で変更：

```
def __init__(self, host="localhost", port=8765):
    self.host = host
    self.port = port
```

実行時に引数として指定することも可能：

```
python src/websocket_server.py --host 0.0.0.0 --port 8765
```

プロジェクト構造と出力ファイルの説明

```
.
├── app/                        # フロントエンドアプリケーション
│   ├── electron/               # Electron関連ファイル
│   │   ├── main.js             # Electronのメインプロセス
│   │   └── preload.js          # プリロードスクリプト
```

```
│   ├── src/                    # Reactアプリケーションソース
│   │   ├── components/         # UIコンポーネント
│   │   │   ├── App.js          # メインアプリケーションコンポーネント
│   │   │   ├── ImageGallery.js# 画像ギャラリー表示
│   │   │   ├── StartScreen.js # 開始画面
│   │   │   └── ...             # その他のコンポーネント
│   │   ├── hooks/              # Reactカスタムフック
│   │   │   ├── useEyeTracker.js # アイトラッカー連携フック
│   │   │   └── ...             # その他のフック
│   │   ├── services/           # APIサービス
│   │   │   ├── imageGenerationService.js # 画像生成サービス
│   │   │   └── predictionService.js      # 予測サービス
│   │   └── utils/              # ユーティリティ関数
│   │       ├── dataManager.js # データ管理
│   │       └── sessionManager.js # セッション管理
│   ├── build/                  # ビルド出力ディレクトリ
│   ├── index.html              # メインHTMLファイル
│   ├── package.json            # npm設定
│   └── webpack.config.js       # Webpack設定
│
├── model/                      # バックエンドモデルサービス
│   ├── src/                    # ソースコード
│   │   ├── api/                # FastAPI定義
│   │   │   ├── server.py       # APIサーバーメイン
│   │   │   └── routes/         # APIルート
│   │   │       ├── data.py     # データ保存API
│   │   │       └── generation.py # 画像生成API
│   │   ├── models/             # 機械学習モデル
│   │   │   ├── architectures.py # モデルアーキテクチャ
│   │   │   └── model_loader.py # モデル読み込み
│   │   ├── data/               # データ処理
│   │   │   └── preprocess.py  # 前処理関数
│   │   └── services/           # サービス実装
│   │       └── stable_diffusion_service.py # Stable Diffusion連携
│   ├── tests/                  # テストコード
│   └── requirements.txt        # Pythonパッケージ要件
│
├── eye_tracker/                # アイトラッカー連携
│   ├── src/                    # アイトラッカー接続コード
│   │   ├── eye_tracker.py      # 基本実装
│   │   └── websocket_server.py # WebSocketサーバー
│   ├── environment.yml         # Conda環境定義
│   └── start_eye_tracker.bat   # Windowsスタートスクリプト
│
├── data/                       # データ保存ディレクトリ
│   └── experiment_sessions/    # 実験セッションデータ
│       ├── gaze_evaluation/    # 視線ベース評価データ
│       │   └── [session_id]/   # 各セッションディレクトリ
│       │       ├── session_info.json    # セッションメタデータ
│       │       ├── final_selected_image.png # 最終選択画像
│       │       └── step_[n]/   # 各ステップデータ
│       │           ├── step_data.json   # ステップメタデータ
│       │           ├── gaze_data.csv    # 視線データCSV
│       │           └── image_[i].png    # 生成画像ファイル
```

```
│       └── manual_evaluation/ # 手動評価データ（同様の構造）
│
├── models/                    # 学習済みモデル保存ディレクトリ
│   ├── lstm/                  # LSTMモデル
│   │   ├── method1/           # Method1モデル
│   │   └── method2/           # Method2モデル
│   └── transformer/           # Transformerモデル
│       ├── method1/           # Method1モデル
│       └── method2/           # Method2モデル
│
├── docker/                    # Dockerファイル
│   ├── app.Dockerfile         # フロントエンドDockerfile
│   └── model.Dockerfile       # バックエンドDockerfile
├── docker-compose.yml         # Docker Compose設定
└── README.md                  # このファイル
```

出力ファイルの説明

システムは実験セッション中に様々なデータを保存する。主な出力ファイルは以下の通り：

セッションメタデータ (`session_info.json`)

各セッションのメタデータを保存：

```
{
  "evaluationMethod": "gaze",       // 評価方法（"gaze"または"manual"）
  "modelId": "lstm_method1",        // 使用したモデルID
  "prompt": "mountain landscape",   // 生成プロンプト
  "negativePrompt": "blurry",       // ネガティブプロンプト
  "fixedPlacement": true,           // 配置設定
  "eyeTrackerConnected": true,      // アイトラッカー接続状態
  "timestamp": "2024-01-07T06:30:45.000Z", // ISO形式タイムスタンプ
  "startTime": "2024-01-07 15:30:45",      // 日本時間開始時刻
  "endTime": "2024-01-07 15:35:20",        // 日本時間終了時刻
  "sessionDurationSeconds": 275.0,         // セッション時間（秒）
  "finalSelectedImage": {           // 最終選択画像情報
    "src": "/session-data/123/step_5/image_2.png",
    "index": 2,
    "prediction": 2,
    "confidence": 0.85
  }
}
```

ステップデータ (`step_data.json`)

各進化ステップの情報を保存：

```
{
  "sessionId": "123",             // セッションID
```

```
  "stepId": 2,                    // ステップID
  "timestamp": "2024-01-07T06:32:15.000Z",  // タイムスタンプ
  "gazeData": {                   // 視線データ概要（詳細はCSVに保存）
    "sampleCount": 120,           // サンプル数
    "duration": 2000000           // 持続時間（マイクロ秒）
  },
  "images": [                     // 画像情報
    {
      "url": "/session-data/123/step_2/image_0.png",
      "latent_vector": "base64encoded...",  // 潜在ベクトル（Base64）
      "prompt": "mountain landscape",
      "generation": 2
    },
    // ... 他の画像
  ],
  "prompt": "mountain landscape", // 使用プロンプト
  "generation": 2,                // 世代番号
  "selection": {                  // 選択情報
    "selectedImageIndex": 1,      // 選択画像インデックス
    "predictedIndex": 1,          // 予測インデックス
    "predictionConfidence": 0.78  // 予測信頼度
  },
  "evaluationMethod": "gaze"      // 評価方法
}
```

視線データ (`gaze_data.csv`)

視線追跡データをCSV形式で保存：

```
timestamp,left_x,left_y,right_x,right_y
1641536445000000,0.231,0.452,0.235,0.448
1641536445016667,0.232,0.453,0.236,0.449
...
```

- `timestamp`: マイクロ秒単位のタイムスタンプ
- `left_x`, `left_y`: 左目の正規化された座標（0〜1）
- `right_x`, `right_y`: 右目の正規化された座標（0〜1）

画像ファイル (`image_[i].png`)

各ステップで生成された画像ファイル。最終選択画像は別途 `final_selected_image.png` としても保存される。

新しい予測モデルの追加

1. モデルの学習後、`.pth`ファイルを適切なディレクトリに配置:

   ```
   models/[model_type]/[method_name]/model.pth
   ```

2. `model/src/models/architectures.py` に新しいモデルアーキテクチャを追加（必要な場合）
3. `model/src/models/model_loader.py` で新しいモデルタイプに対応するローダーを追加
4. `app/src/components/App.js` のモデル定義を更新:

```
const AVAILABLE_MODELS = [
  // 既存のモデル
  { id: 'new_model_id', name: 'New Model Name', type: 'new_type' },
];
```