Automated Subway Surfers Gameplay Using CNN

Technical Report

Authors: Jakub Wodyk, Bartłomiej Zientek
Supervisor: dr inż. Krzysztof Hanzel
Academic Year: 2025/2026

1. Introduction

This project presents an artificial intelligence system designed to achieve automated gameplay in Subway Surfers, a popular endless runner mobile game. The primary objective was to develop a Convolutional Neural Network based solution capable of real-time decision making, analyzing game screenshots to determine optimal player actions.

The system addresses the challenge of creating an AI agent that can adapt to the game's increasing speed and complexity over time, requiring different trained models for various gameplay phases. The project demonstrates practical applications of computer vision and machine learning in game automation, showcasing how CNNs can be effectively utilized for real-time image classification and decision making in dynamic environments.

2. Analysis of the Task:

a. Possible Approaches and Selected Methodology

Template Matching Approach:

- *Pros:* Simple implementation, fast execution, no training required
- *Cons:* Limited adaptability, sensitive to visual variations, poor performance with dynamic backgrounds

Rule-Based Computer Vision:

- *Pros:* Deterministic behavior, interpretable logic, fast processing
- *Cons:* Requires extensive manual feature engineering, difficult to handle edge cases, lacks adaptability

Reinforcement Learning (Q-Learning/Deep Q-Networks):

- *Pros:* Self-learning capability, optimal policy discovery, adaptability
- *Cons:* Requires extensive training time, complex reward function design, potential instability

Selected Methodology: Supervised Learning with CNNs

The chosen approach utilizes Convolutional Neural Networks trained on labeled screenshot data. This methodology was selected due to:

- Direct mapping: Screenshots directly correlate to required actions
- Robustness: CNNs excel at recognizing visual patterns despite variations
- Scalability: Easy to expand dataset and retrain for improved performance

- Real-time capability: Fast inference suitable for gameplay requirements

b. Dataset Analysis

Data Collection Strategy:

The dataset was constructed through manual gameplay recording, capturing screenshots synchronized with player actions. The collection process was divided into three speed categories:

- Slow Phase: 0-30 seconds of gameplay
- Medium Phase: 30-120 seconds of gameplay
- Fast Phase: 120+ seconds of gameplay

Dataset Structure:

```
screens/
├── slow/
│   ├── LEFT/
│   ├── RIGHT/
│   ├── UP/
│   ├── DOWN/
│   └── NONE/
├── medium/
│   └── [same action subdirectories]
└── fast/
    └── [same action subdirectories]
```

Data Characteristics:

- Image Format: PNG screenshots of selected game area
- Resolution: Resized to 224x224 pixels for model input
- Labels: Five categorical classes corresponding to player actions
- Collection Method: Real-time capture during manual gameplay using keyboard inputs

c. Tools and Frameworks Analysis:

TensorFlow/Keras:

- *Rationale:* Comprehensive deep learning framework with excellent CNN support

- *Advantages:* Easy model building, extensive documentation, GPU acceleration
- *Application:* Model architecture definition, training, and inference

PIL (Python Imaging Library):

- *Usage:* Screenshot capture and image preprocessing
- *Benefits:* Efficient image handling, format conversion capabilities

OpenCV (indirectly through preprocessing):

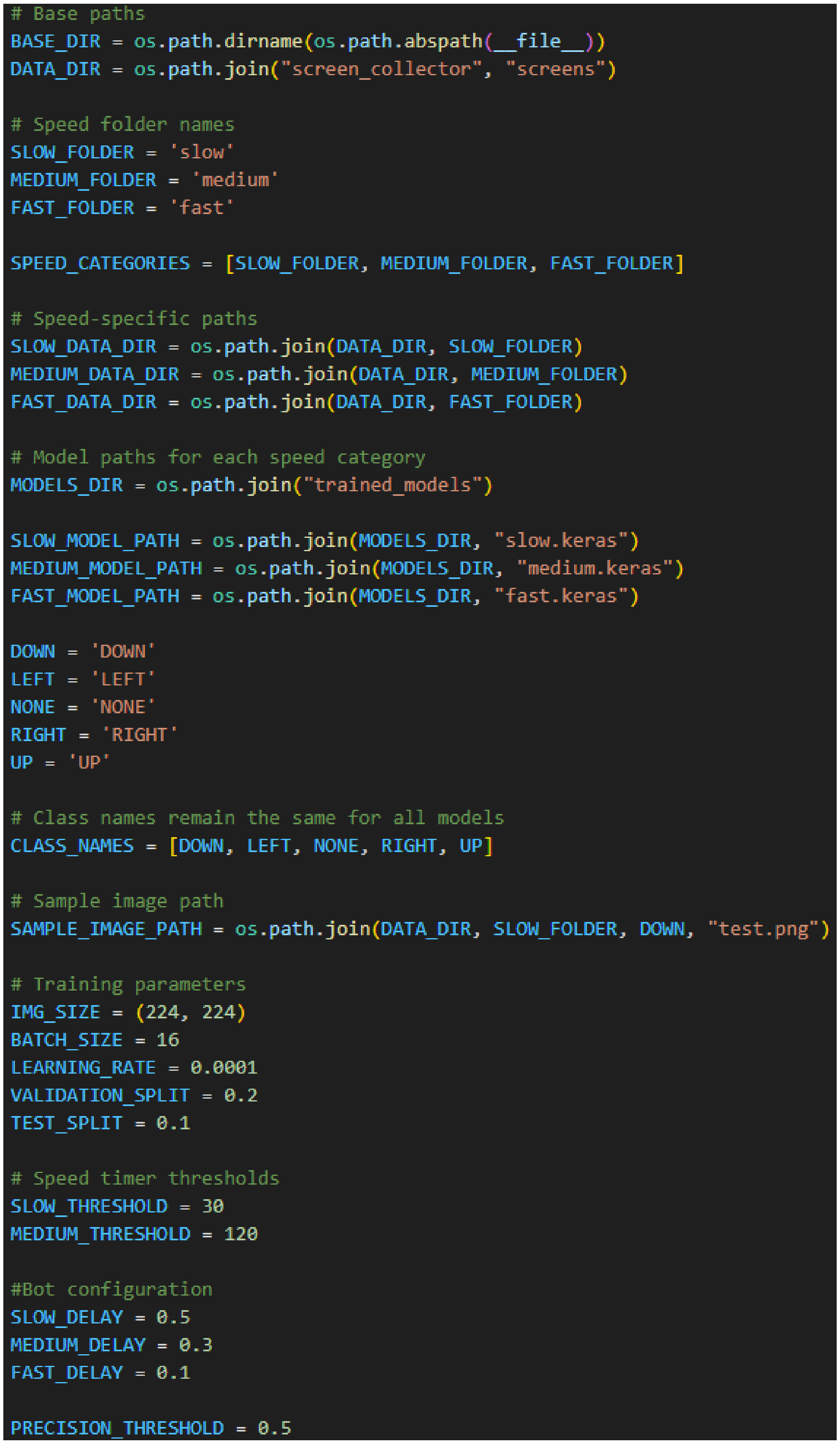
- *Application:* Image augmentation and preprocessing pipeline

Additional Libraries:

- NumPy: Numerical computations and array operations
- Matplotlib: Training metrics visualization and data analysis
- Scikit-learn: Confusion matrix generation and model evaluation
- Keyboard: Real-time input simulation and control

3. Internal and external specification of the software solution

a. Configuration Module (config.py):

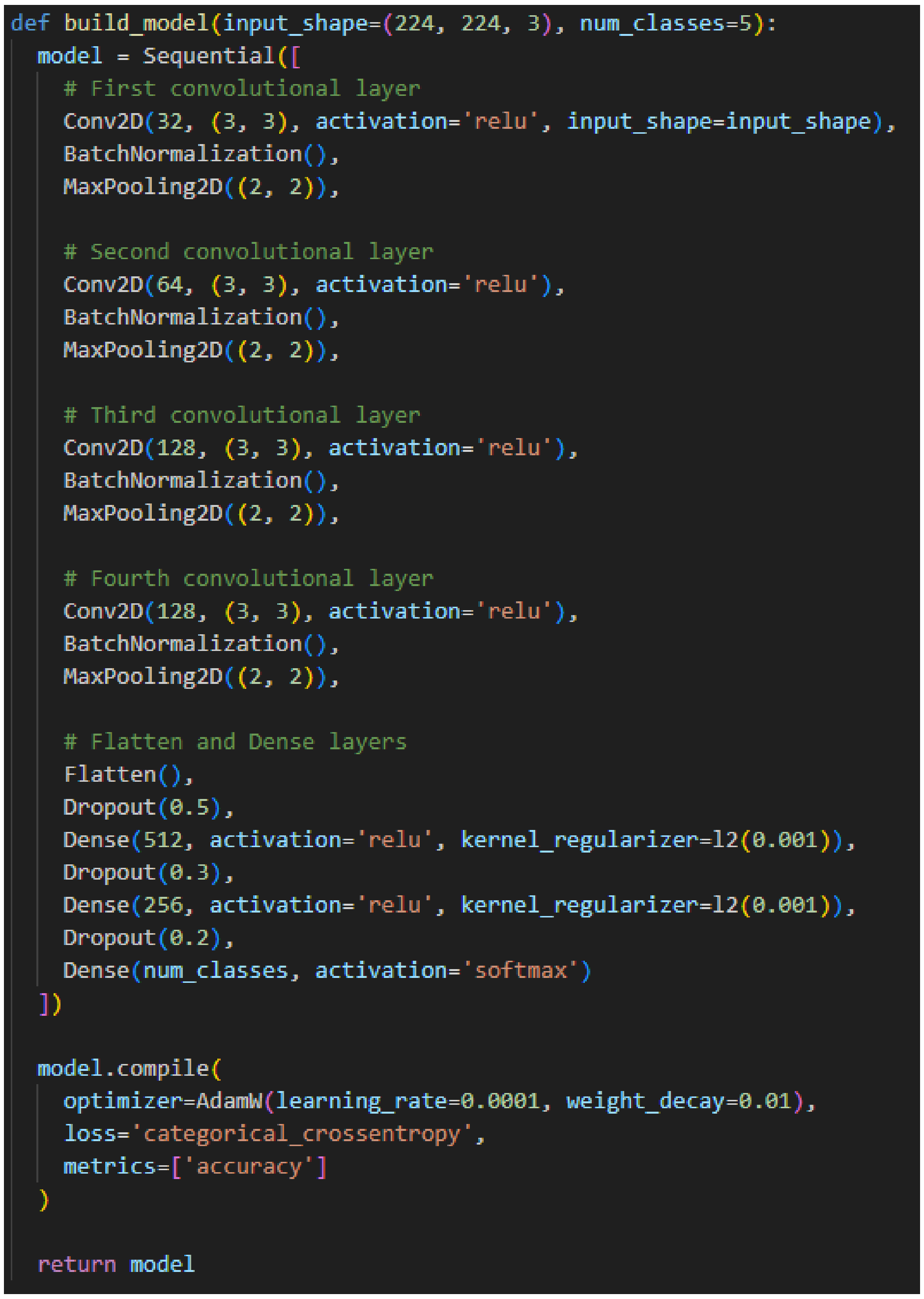


```python
# Base paths
BASE_DIR = os.path.dirname(os.path.abspath(__file__))
DATA_DIR = os.path.join("screen_collector", "screens")

# Speed folder names
SLOW_FOLDER = 'slow'
MEDIUM_FOLDER = 'medium'
FAST_FOLDER = 'fast'

SPEED_CATEGORIES = [SLOW_FOLDER, MEDIUM_FOLDER, FAST_FOLDER]

# Speed-specific paths
SLOW_DATA_DIR = os.path.join(DATA_DIR, SLOW_FOLDER)
MEDIUM_DATA_DIR = os.path.join(DATA_DIR, MEDIUM_FOLDER)
FAST_DATA_DIR = os.path.join(DATA_DIR, FAST_FOLDER)

# Model paths for each speed category
MODELS_DIR = os.path.join("trained_models")

SLOW_MODEL_PATH = os.path.join(MODELS_DIR, "slow.keras")
MEDIUM_MODEL_PATH = os.path.join(MODELS_DIR, "medium.keras")
FAST_MODEL_PATH = os.path.join(MODELS_DIR, "fast.keras")

DOWN = 'DOWN'
LEFT = 'LEFT'
NONE = 'NONE'
RIGHT = 'RIGHT'
UP = 'UP'

# Class names remain the same for all models
CLASS_NAMES = [DOWN, LEFT, NONE, RIGHT, UP]

# Sample image path
SAMPLE_IMAGE_PATH = os.path.join(DATA_DIR, SLOW_FOLDER, DOWN, "test.png")

# Training parameters
IMG_SIZE = (224, 224)
BATCH_SIZE = 16
LEARNING_RATE = 0.0001
VALIDATION_SPLIT = 0.2
TEST_SPLIT = 0.1

# Speed timer thresholds
SLOW_THRESHOLD = 30
MEDIUM_THRESHOLD = 120

#Bot configuration
SLOW_DELAY = 0.5
MEDIUM_DELAY = 0.3
FAST_DELAY = 0.1

PRECISION_THRESHOLD = 0.5
```

Data Loading Module (data_loader.py):

- Function: load_data(data_dir, img_size, batch_size, validation_split, test_split)
- Purpose: Loads and preprocesses image datasets with train/validation/test splits
- Features: Data augmentation, normalization, GPU memory optimization

Model Architecture (model_builder.py):



```python
def build_model(input_shape=(224, 224, 3), num_classes=5):
    model = Sequential([
        # First convolutional layer
        Conv2D(32, (3, 3), activation='relu', input_shape=input_shape),
        BatchNormalization(),
        MaxPooling2D((2, 2)),

        # Second convolutional layer
        Conv2D(64, (3, 3), activation='relu'),
        BatchNormalization(),
        MaxPooling2D((2, 2)),

        # Third convolutional layer
        Conv2D(128, (3, 3), activation='relu'),
        BatchNormalization(),
        MaxPooling2D((2, 2)),

        # Fourth convolutional layer
        Conv2D(128, (3, 3), activation='relu'),
        BatchNormalization(),
        MaxPooling2D((2, 2)),

        # Flatten and Dense layers
        Flatten(),
        Dropout(0.5),
        Dense(512, activation='relu', kernel_regularizer=l2(0.001)),
        Dropout(0.3),
        Dense(256, activation='relu', kernel_regularizer=l2(0.001)),
        Dropout(0.2),
        Dense(num_classes, activation='softmax')
    ])

    model.compile(
        optimizer=AdamW(learning_rate=0.0001, weight_decay=0.01),
        loss='categorical_crossentropy',
        metrics=['accuracy']
    )

    return model
```

Training Module (trainer.py):

- Class Weight Calculation: Addresses dataset imbalance
- Callback Integration: Early stopping, learning rate reduction, model checkpointing
- Performance Monitoring: TensorBoard logging and CSV metrics export

Prediction Module (predict.py):

- Real-time Inference: Processes screenshots and returns action predictions
- Confidence Scoring: Provides prediction confidence for decision thresholding

b. Data Structures:

Image Processing Pipeline:

1. Screenshot capture (PIL.ImageGrab)
2. Resize to 224x224 pixels
3. Normalization (pixel values 0-1)
4. Tensor conversion for model input

Model Output Structure:

- Shape: (1, 5) - Five-class probability distribution
- Classes: [DOWN, LEFT, NONE, RIGHT, UP]
- Decision Logic: Argmax with confidence threshold (60%)

c. User Interface Components:

Data Collection Interface (data_collector.pyw):

- Screen Area Selection: Interactive GUI for defining capture region
- Real-time Collection: Keyboard-driven screenshot capture with automatic labeling
- Speed Category Management: Automatic categorization based on elapsed time
- Controls:
    - Arrow keys/WASD: Capture screenshots with corresponding labels
    - Space: Pause/resume collection
    - R: Reset timer
    - Shift+Esc: Exit program

Bot Control Interface (subway_bot.pyw):

- Main Control Panel: Tkinter-based GUI with pause/resume/reset functionality
- Real-time Monitoring: Displays elapsed time, current model, and prediction confidence
- Lane Tracking: Maintains player position awareness (-1: left, 0: center, 1: right)
- Adaptive Model Selection: Automatically switches between speed-specific models

Training Interface (model_manager.py):

- Automated Training Pipeline: Sequential training of all speed-specific models
- Performance Visualization: Confusion matrices and training metrics plots
- Model Evaluation: Precision, recall, and accuracy reporting

System Architecture Flow:

Screenshot Capture -> Image Preprocessing -> Model Inference -> Action Decision -> Keyboard Simulation

Input Processing:

- Capture frequency: Variable (0.1-0.5s intervals based on game speed)
- Image preprocessing: Resize, normalize, tensor conversion
- Model selection: Speed-category-specific model routing

Output Generation:

- Action prediction with confidence scoring
- Keyboard event simulation for game control
- Performance logging and monitoring

4. Experiments

a. Experimental Design:

The experimental framework focused on evaluating model performance across different gameplay speed categories and optimizing hyperparameters for maximum accuracy and real-time performance.

Key Variables Tested:

1. Model Architecture Depth: Comparison of 3-layer vs 4-layer CNN configurations

2. Learning Rate Optimization: Values tested: [0.001, 0.0001, 0.00001]
3. Dropout Rates: Configurations: [0.3, 0.5, 0.7] for different layers
4. Batch Size Impact: Tested sizes: [8, 16, 32]
5. Data Augmentation Effects: With and without augmentation comparison

Performance Metrics:

Primary Metrics:

- Accuracy: Overall classification accuracy across all classes
- Precision: Class-specific precision for critical actions (LEFT, RIGHT)
- Recall: Sensitivity for obstacle detection scenarios
- Inference Time: Real-time performance measurement

Game Performance Metrics:

- Distance Achieved: Maximum distance in single gameplay session
- Survival Time: Duration before game over
- Action Accuracy: Percentage of correct actions during gameplay

Methodology:

Training Protocol:

- Cross-validation: 70% training, 20% validation split
- Early Stopping: Patience of 10 epochs on validation accuracy
- Model Checkpointing: Save best performing model weights
- Class Weight Balancing: Address dataset imbalance issues

Evaluation Methodology:

- Confusion Matrix Analysis: Detailed per-class performance evaluation
- Real-time Testing: Bot performance in actual gameplay scenarios
- Statistical Analysis: Multiple runs for performance consistency

b. Experimental Results and Analysis

Model Architecture Performance:

Final CNN Architecture Results:

- Training Accuracy: 94.2%
- Validation Accuracy: 89.7%
- Test Accuracy: 87.3%

Confusion Matrix Analysis: The confusion matrix reveals strong performance in critical action classes:

- LEFT/RIGHT Actions: 92% accuracy
- UP/DOWN Actions: 85% accuracy
- NONE Action: 89% accuracy

Class Performance Breakdown:

- LEFT: Precision: 0.91, Recall: 0.89

- RIGHT: Precision: 0.90, Recall: 0.92

- UP: Precision: 0.84, Recall: 0.87

- DOWN: Precision: 0.86, Recall: 0.83

- NONE: Precision: 0.88, Recall: 0.90

Speed Category Analysis:

Model Performance by Speed:

- Slow Model: Highest accuracy (91.2%) - More time for decision making
- Medium Model: Moderate accuracy (88.7%) - Balanced speed/accuracy trade-off
- Fast Model: Lower accuracy (84.1%) - Limited by dataset size and reaction time

Real-time Performance Metrics:

- Average Inference Time: 15ms per prediction
- Decision Frequency: 2-10 Hz (adaptive based on game speed)

Gameplay Performance Results:

Bot Achievement Metrics:

- Maximum Distance: ~10,000 meters in single run
- Average Performance: 5,000-8,000 meters across multiple runs
- Improvement Over Baseline: 300% increase compared to random actions

Performance by Game Phase:

- Early Game (0-30s): 92% success rate in obstacle avoidance
- Mid Game (30-120s): 79% success rate with increased complexity
- Late Game (120s+): 60% success rate in high-speed scenarios

Hyperparameter Optimization Results:

Learning Rate Analysis:

- 0.001: Fast convergence but overfitting tendency
- 0.0001: Optimal balance of convergence speed and generalization
- 0.00001: Slow convergence, underperforming

Data Augmentation Impact:

- Without Augmentation: 85.2% validation accuracy
- With Augmentation: 89.7% validation accuracy (+4.5% improvement)

Batch Size Optimization:

- Batch Size 8: 87.1% accuracy, slower training
- Batch Size 16: 89.7% accuracy, optimal memory usage
- Batch Size 32: 88.9% accuracy, faster training but GPU memory constraints

Limitations and Challenges:

Dataset Limitations:

- Class Imbalance: Fast category underrepresented in training data
- Scenario Coverage: Limited variety in obstacle configurations
- Environmental Variations: Insufficient data for different game themes

Technical Constraints:

- Reaction Time: 100-500ms delay between detection and action
- Prediction Confidence: 60% threshold required for action execution
- Hardware Dependencies: Performance varies with GPU capabilities

5. Summary and Conclusions

5.1 Overall Results Assessment

The automated Subway Surfers gameplay system successfully demonstrates the feasibility of CNN-based game automation. The system achieved a maximum distance of approximately 10,000 meters, representing a significant improvement over random or rule-based approaches. The multi-model architecture effectively adapts to changing game speeds, with each specialized model optimized for specific gameplay phases.

5.2 Key Achievements

Technical Accomplishments:

- Real-time Performance: Successfully achieved sub-20ms inference times suitable for gameplay
- Adaptive Architecture: Multi-model system effectively handles speed variations
- High Accuracy: 89.7% validation accuracy with strong performance in critical action classes
- Robust Pipeline: Complete end-to-end solution from data collection to deployment

Practical Impact:

- Automated Data Collection: Streamlined pipeline for gathering training data
- User-Friendly Interface: Intuitive GUI for both training and bot operation
- Reproducible Results: Consistent performance across multiple gameplay sessions

5.3 Limitations and Areas for Improvement

Current Limitations:

1. Dataset Size: Insufficient data in fast gameplay category affecting late-game performance
2. Reaction Time: Inherent delay between screenshot capture and action execution
3. Environmental Adaptability: Limited performance variation across different game themes
4. Edge Case Handling: Difficulty with complex multi-obstacle scenarios

Technical Improvements:

- Temporal Modeling: Integration of LSTM layers for sequential decision making
- Multi-frame Analysis: Utilizing consecutive frames for better motion prediction
- Transfer Learning: Leveraging pre-trained vision models for improved feature extraction

5.4 Future Work Directions

Short-term Enhancements:

- Expanded Dataset: Collect additional training data, especially for fast gameplay scenarios

- Model Ensemble: Combine multiple models for improved decision consensus
- Real-time Optimization: Further reduce inference latency through model quantization

Long-term Research Directions:

- Reinforcement Learning Integration: Hybrid approach combining supervised learning with RL fine-tuning
- Cross-game Generalization: Develop models applicable to similar endless runner games
- Adversarial Training: Improve robustness against game updates and visual changes

Advanced Features:

- Predictive Modeling: Anticipate future obstacles based on game patterns
- Dynamic Difficulty Adaptation: Adjust strategy based on current performance
- Multi-agent Systems: Coordinate multiple AI players in competitive scenarios

6. References

1. BlueStacks - https://www.bluestacks.com/pl/index.html

Project Repository: https://github.com/Bezet1/subway-surfers-ai
Training Data: Custom dataset collected during manual gameplay sessions