Technical Report:

Reinforcement Learning Course project: “Crossy Road RL agent”

Team members:

Dmitrii Kuzmin - https://github.com/1kkiRen - dm.kuzmin@innopolis.university
Yaroslava Bryukhanova - https://github.com/YaroslavaBryukhanova - y.bryukhanova@innopolis.university
Nazgul Salikhova - https://github.com/Nazgulitos - n.salikhova@innopolis.university

Project topic:

AI agent that plays the Crossy Road game using RL and CV.

Repository:

https://github.com/1kkiRen/PMLDL-Course-Project

Artifacts:

Repository
Dataset
Script for screenshots collection
CV_Models
RL_Models

Table of Contents:

Problem statement and objective:

Crossy Road is a mobile game where a hen needs to cross the road without hitting any obstacles. In this project, we aimed to create a RL agent that can play this game.

To achieve this, we followed a research-oriented approach, experimenting with several techniques to create and refine our agent.

Initially, we attempted to train an RL model directly using raw game screenshots as input. However, this approach showed poor results, so we shifted to incorporating computer vision (CV) techniques for object detection, enabling the agent to identify key elements like obstacles and the hen (our agent).

Continuing several approaches, we further enhanced the method by transforming the detected objects into a grid representation, simplifying the game environment for the agent's understanding of the state space. This grid-based approach provided a structured and manageable input for the RL model.

Methodology:

Our project consists of two main components:

1. **Reinforcement Learning (RL) Component**:

This part focuses on the agent, which learns to interact with the environment using reinforcement learning techniques.

2. **Computer Vision (CV) Component**:

This part involves an object detection model, responsible for identifying and locating objects such as obstacles and other entities in the environment, and template matching method for restart button detection.

RL component:

Three distinct reinforcement learning (RL) approaches were explored, each optimized for the unique dynamics of the Crossy Road game. The methods were systematically tested, optimized, and compared to evaluate their performance.

Environment: The Crossy Road environment consists of a continuous scrolling road with various obstacles, such as vehicles, rivers, and gaps.

State-space: The agent's state will include the screenshot of the game, position of the agent, and position of obstacles.

Action-space: The agent can take actions such as moving left, right, backward, forward to navigate the environment and avoid obstacles.

Reward Function Design

Positive Rewards:

1. +2 for one step forward: Encourages forward progress as the main goal of the game.
2. +0.25 for one step right/left: Supports obstacle avoidance without overshadowing forward movement.
3. +0.2 for waiting: Rewards strategic patience in risky situations but keeps progress prioritized.

Negative Rewards:

1. -100 for game over: Strong penalty to prioritize survival and avoid failure.
2. -10 for stacking in an obstacle: Discourages inefficiency and promotes quick recovery.

Balancing: Rewards and penalties are scaled to ensure the agent prioritizes forward movement and survival while still adapting to obstacles strategically.

1st Iteration: Direct Screenshot Input

The initial approach involved using raw game screenshots as input for a Deep Q-Network (DQN) to determine the action the hen should take (up, down, left, or right). The main objective was to train the model to map the pixel data of the screen directly to the optimal action.

Unfortunately, this method proved inefficient. The model often failed to converge, or if it did, the convergence was extremely slow. This was primarily due to the high dimensionality of the raw pixel data, which made the learning process challenging for the DQN.

DQN Model for Crossy Road

Below is the implementation of the DQN model, where:

- Input: A game screenshot image.
- Output: One of four possible actions for the hen.

```
class DQN(nn.Module):
    def __init__(self, input_size, output_size, hidden_size):
        super(DQN, self).__init__()
        self.input_size = input_size
        self.output_size = output_size
        self.hidden_size = hidden_size

        self.conv1 = nn.Conv2d(3, 16, kernel_size=5, stride=2)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=5, stride=2)
        self.conv3 = nn.Conv2d(32, 32, kernel_size=5, stride=2)

        def conv2d_size_out(size, kernel_size=5, stride=2):
            return (size - (kernel_size - 1) - 1) // stride + 1

        convw = conv2d_size_out(conv2d_size_out(conv2d_size_out(input_size[1])))
        convh = conv2d_size_out(conv2d_size_out(conv2d_size_out(input_size[2])))
        linear_input_size = convw * convh * 32

        self.fc1 = nn.Linear(linear_input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        x = F.relu(self.fc1(x.reshape(x.size(0), -1)))
        return self.fc2(x)

    def act(self, state, epsilon):
        if random.random() > epsilon:
```

```
            state = torch.FloatTensor(state).unsqueeze(0)
            q_value = self.forward(state)
            action = q_value.max(1)[1].data[0]
        else:
            action = random.randrange(self.output_size)
        return action
```

2nd Iteration:

To improve the model's performance, the input was shifted from raw game screenshots to detected objects. Using the computer vision (CV) pipeline, the detected objects were encoded into a more abstract and structured representation, such as positions and types of obstacles. This significantly reduced the input complexity and dimensionality.

However, the performance was still limited due to challenges in effectively encoding the detected objects into a format that the model could utilize efficiently.

DQN Model with CV for Crossy Road

Below is the implementation of the DQN model, where:

- Input: Data about detected objects in matrix format.
- Output: One of four possible actions for the hen.

```
class DQN(nn.Module):
    def __init__(self, input_size, output_size, hidden_size):
            super(DQN, self).__init__()
            self.fc1 = nn.Linear(input_size * 160, hidden_size * 16)
            self.fc2 = nn.Linear(hidden_size * 16, hidden_size)
            self.fc3 = nn.Linear(hidden_size, hidden_size // 4)
            self.fc4 = nn.Linear(hidden_size // 4, output_size)

        def forward(self, x):
            if len(x.shape) > 2:
                batch_size = x.shape[0]
                x = x.view(batch_size, -1)
            else:
                x = torch.flatten(x)

            x = F.relu(self.fc1(x))
```

```
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = self.fc4(x)

        return x


    def act(self, state, epsilon):
        if random.random() < 0.975:
            state = torch.tensor(state, dtype=torch.float32)

            q_value = self.forward(state)
            action = torch.argmax(input=q_value).item()

        else:
            action = random.choice([0, 1, 2, 3, 4])
        return action
```

3rd Iteration:

The third iteration transformed the game field into a grid representation based on the game screenshot. Detected objects were mapped onto this grid, creating a simplified and structured environment for the RL agent. [example of grid creating]

This grid-based approach proved to be the most effective, as it reduced the state space's complexity while preserving critical spatial relationships between the objects. The model could now focus on learning optimal actions in a manageable and interpretable environment, leading to significant improvements in training speed and performance.

*the model is the same as in 2nd Iteration

4th Iteration:

IIn the fourth iteration, we replaced the Deep Q-Network (DQN) model with a Recurrent Implicit Quantile Network (R-IQN) to enhance the agent's ability to handle sequential dependencies and long-term decision-making in the dynamic game environment. The R-IQN model incorporates recurrent layers, allowing the agent to better understand the

temporal relationships between states and actions, which is crucial for navigating complex, fast-changing scenarios in *Crossy Road*.

Additionally, we expanded the game environment's complexity by increasing the number of class labels from 3 to 5, encompassing a wider variety of objects: obstacles, hen, cars, timbers, water. This enhanced labeling provided the RL agent with a richer understanding of the environment.

```
class RecurrentIQN(nn.Module):
    def __init__(self, input_size, output_size, hidden_size, n_quantiles=32):
        super(RecurrentIQN, self).__init__()
        self.n_quantiles = n_quantiles
        self.input_size = input_size
        self.hidden_size = hidden_size

        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.quantile_embed = nn.Linear(hidden_size, hidden_size)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x, quantiles, hidden):
        lstm_out, hidden = self.lstm(x, hidden)

        quantiles = quantiles.unsqueeze(-1)
        pi = torch.acos(torch.zeros(1)).item() * 2
      quantile_feats = torch.cos(pi * quantiles * torch.arange(1,
self.hidden_size + 1).to(x.device))
        quantile_feats = F.relu(self.quantile_embed(quantile_feats))

        lstm_out = lstm_out[:, -1, :].unsqueeze(1)
        x = lstm_out * quantile_feats

        x = self.fc(x)
        return x, hidden

    def act(self, state, hidden, epsilon):
        if random.random() > epsilon:
            print("Model acting")
            with torch.no_grad():
            state =
torch.FloatTensor(state).unsqueeze(0).unsqueeze(0).to(next(self.parameters()).de
vice)
            quantiles = torch.rand(1, self.n_quantiles).to(state.device)
            q_values, hidden = self.forward(state, quantiles, hidden)
            q_values = q_values.mean(dim=1)
            action = q_values.argmax(dim=1).item()
        else:
            action = random.randrange(self.fc.out_features)
        return action, hidden
```

CV component:

Aim:

The goal was to detect objects on the game screen to make the agent's environment observable. The object detection included the following classes:

- **Hen**: The main character controlled by the agent.
- **Obstacle**: Static obstacles (e.g., bush, stone, stump) and moving obstacles (e.g., cars, trucks, trains).
- **Timber**: Floating timber and leaves found on the river.

Template Matching Method

The initial approach for object detection relied on OpenCV's Template Matching technique. Templates of objects were created with background removal to aid detection.

1st Iteration:

Method: Applied a common Template Matching method for all object types.

Result: A universal score threshold could not be managed effectively for all object classes, leading to poor performance.

2nd Iteration:

Method: Used Template Matching separately for each class, organizing templates into folders for specific objects:

- Hen: Front, back, right, and left views.
- Bush: Three sizes.
- Obstacle: Cars and trucks with identified movement directions (left/right).
- Timber: Three sizes, including floating leaves.

Result: The cubic nature of the game objects and their prominent corners made it difficult to set reliable score thresholds, resulting in low detection accuracy.

Alternative Computer Vision Methods

To improve detection, various alternative CV techniques were explored:

- **Sobel Gradient + Watershed Segmentation + Histogram Matching**:
  Result: Inaccurate detections.
- **SIFT (Scale-Invariant Feature Transform)**:

  Result: Weak performance due to the similarity between object corners and the surrounding game world.

Using Visual Models

Given the limitations of traditional methods, advanced visual models were employed:

Dataset Preparation:

A labeled dataset of 100 images was created and processed on the Roboflow platform.

Training:

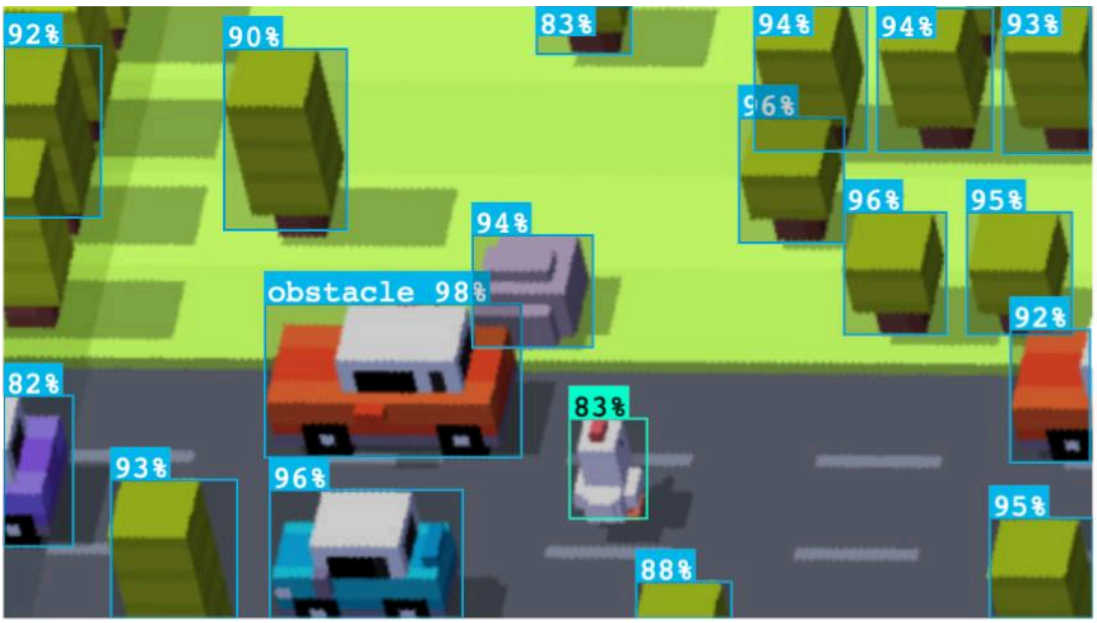
- **Roboflow 3.0 Object Detection Model**: Pretrained on the COCO dataset, fine-tuned using the custom dataset.
- **YOLOv5 Pytorch**: Chosen for testing within the dynamic environment of the game. [Model_yolov5]
- **YOLOv11l**: Chosen as a state-of-the-art (SOTA) open-source model for its exceptional object detection capabilities. Fine-tuned on the annotated dataset to achieve robust detection. [Model_yolov11l]

Result: The trained YOLOv11l model became the core of the CV component, effectively detecting objects and enabling the RL agent to observe its environment accurately.

Comparison of fine-tuned models:

Model	mAP	Precision	Recall	Size of dataset	Number of epochs
Fine-tuned Roboflow 3.0 Object Detection	93.4%	96.6%	88.2%	100	100
Fine-tuned YOLOv11l	82.3%	94.7%	77.3%	100	100

Example of game screen with detected objects:



While we chose YOLOv11l as the primary model for object detection, the detection of the “end of the game” event continues to rely on a correlation-based approach. This method has been adapted to improve its efficiency by focusing on a predefined region of interest (ROI) where the restart button consistently appears, given its fixed position in the game interface.

This hybrid strategy ensures robust game state recognition while leveraging YOLOv11l’s strengths for dynamic object detection during gameplay.

Dataset:

A dataset containing game screens was not available through free online resources, necessitating the development of a custom Python script to capture game screenshots at 0.1-second intervals. Using this script, a total of 500 screenshots were collected to facilitate object detection for the computer vision (CV) component of the project. This script also served as a crucial tool for generating input data for the real-time reinforcement learning (RL) model.

Given the limitations of correlation-based methods in our object detection tasks, we shifted our focus to training advanced visual models. To achieve this, we manually annotated 100 screenshots and utilized AI-assisted labeling via the Roboflow platform to expedite the process.

In summary, the following assets were created:

- A Python script for real-time game screen capture. [Script for screenshots collection]
- A dataset of 100 annotated game screens for training object detection models. [Dataset]

Preprocessing:

At the beginning the data had almost no preprocessing, but due to changing of the training approaches we had to add some.

Step by step preprocessing algorithm:

- Take the screenshot of the game
- Rotate the image by 14 degrees
- Make a crop at the center of the image
- Make a resize of the image

Initially, the game screen is 850 by 480 pixels on the 1920 by 1080 pixels monitor.

After preprocessing, the game screen becomes 425 by 240 pixels

Justification of Preprocessing Steps:

• **Rotation**: The game is isometric, and converting it into a grid-based 2D representation requires alignment. A 14-degree rotation transforms the isometric view into a near-2D perspective, making grid creation more effective.

• **Cropping**: Due to the transformation into 2D, the full screen cannot be captured accurately. Cropping focuses on the controlled agent and nearby objects, ensuring essential information is preserved while excluding irrelevant areas.

• **Resizing**: Standardizing the image dimensions helps maintain consistency across the dataset and ensures compatibility with the model input requirements.

Result: The 500 images in total were collected, labeled, and preprocessed for further trainings.

Results:

Iteration	Description	Advantages	Disadvantages	Outcome	Best agent score (max steps number)
1st: Raw Screenshots	Directly input the raw game screenshot into the DQN model.	-Simple to implement. -No preprocessing required.	-High dimensionality makes training difficult. -Slow,no convergence. - Inefficient use of resources.	Ineffective: The model failed to converge after 4hrs.	6 points
2nd: Detected Objects	Used detected objects from the CV pipeline as the model's input for DQN model.	- Reduced input dimensionality. -Abstract representation of the state.	- Challenges in encoding detected objects effectively. - Limited improvement in performance.	Partially effective: Some improvement, but still not sufficient	13 points
3rd: Grid Representation	Transformed the game field into a grid representation, with detected objects mapped onto the grid that is used as input	- Simplified state representation. -Preserved spatial relationships. - Faster and more stable training.	- Additional preprocessing step required to create the grid. - Requires robust object detection.	Effective: Achieved improvements in training speed and model performance	21 points

	for DQN model				
4th: R-IQN with Enhanced Environment	Replaced DQN with R-IQN model and increased the number of class labels in the environment to 5 (obstacles, hen, cars, timbers, water).	- Improved sequential decision-making due to recurrent layers. - Enhanced state representation with diverse class labels.	- Increased complexity in model implementation and training. - Higher computational costs.	Highly effective: Significant improvement in adaptability and performance.	29 points

In conclusion, the fourth iteration, which utilized the R-IQN model and an enhanced game environment with five class labels, proved to be the most effective method. This iteration achieved the highest performance in the Crossy Road game, with the agent scoring 29 points. By replacing the DQN model with the R-IQN model, the AI gained the ability to handle uncertainty more effectively, enabling more robust decision-making.

Project timeline with team members' responsibilities:

Week №	Task	Assigned to
Week 1	Research & Planning the project	Nazgul - CV methods; Yaroslava, Dmitrii - RL methods
Week 2	Data collection and labeling manually and using visual model	Nazgul
Week 3	1) Determine the main model architecture 2) Setup training environment	1) Dmitrii 2) Yaroslava
Week 4	Initial approach - training model without object	Dmitrii

	detection	
Week 5	1) Fine-tuning YOLOv11 for object detection and classification 2) Integrate new methodology into RL agent and compare results 3) Train RL agent	1) Yaroslava 2) Nazgul 3) Dmitrii
Week 6	1) Developing grid approach 2) Integrate new methodology into RL agent, train the model	1) Yaroslava 2) Nazgul
Week 7	1) Develop R-IQN and test 2) Finalize the project, compare all the results	Nazgul, Dmitrii, Yaroslava
Week 8	Prepare the report and project demo	Nazgul, Dmitrii, Yaroslava

Limitations:

Despite the progress made in developing the RL agent for playing Crossy Road, our project faced several limitations.

- First, despite the high accuracy of CV models, there might be some mistakes, which potentially can lead to inaccuracies in object detection, which could affect the agent's decision-making.
- Second, the grid representation, while simplifying the environment, might have oversimplified certain game dynamics, leading to suboptimal learning in complex scenarios.
- Additionally, the RL training process was computationally intensive and required significant time. One significant constraint was the lack of computational resources, which impacted the training speed.

References:

To deepen your understanding of the task, we recommend exploring the following related projects that served as inspiration for our project:

1. **Deep Learning Crossy Road**
   Written by Yilong Song; Project by Abdul Rauf Abdul Karim, Juliet Mankin, and Yilong Song.
   Repository: GitHub
2. **CS221 Poster: Reinforcement Learning for Crossy Road**
   Sajana Weerawardena, Alwyn Tan, and Nick Rubin.
   Poster: Stanford University
3. **Deep Q-Learning Crossy Road**
   Marlon Facey, Abraham Hussain, Jeffrey Pak, Kevin Tat, and Michelle Zhao.
   Repository: GitHub