МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ

УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

МОСКОВСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ



Кафедра СМАРТ-технологии

Лабораторная работа №2

«Применение обучения с подкреплением для задач управления.»

По дисциплине: «Нейронные сети глубокого обучения в обработке изображений»

Группа 221-327

№ группы

Студент Зубарева Дарья Павловна

Подпись студента

Дата 11.04.2025

Дата сдачи

Преподаватель Байрамов Эльмин Вагифович

Подпись преподавателя

**Цель работы:** Разработать приложение для обучения нейронной сети методом обучения с подкреплением на базе генетического алгоритма

**Задачи:**

• Реализовать базовое приложение для подключения к симулятору;

• Сформировать индивидуальный вариант задания;

• Определить функцию пригодности;

• Подготовить программный модуль функции принятия решений на базе нейронной сети с одним скрытым слоем;

• Подготовить модуль загрузки-выгрузки параметров нейронной сети из текстового файла формата CSV;

• Подготовить модуль взаимодействия симулятором для реализации (проигрывания) одной поведенческой модели;

• Реализовать процесс обучения сети с помощью генетического алгоритма в рамках обучения с подкреплением.

**Ход работы.**

Создадим форму приложения. На рисунке 1 показан итоговый вид формы, где будут отображаться веса сети, где можно будет тренировать сеть, менять веса и устанавливать соединение с симулятором.

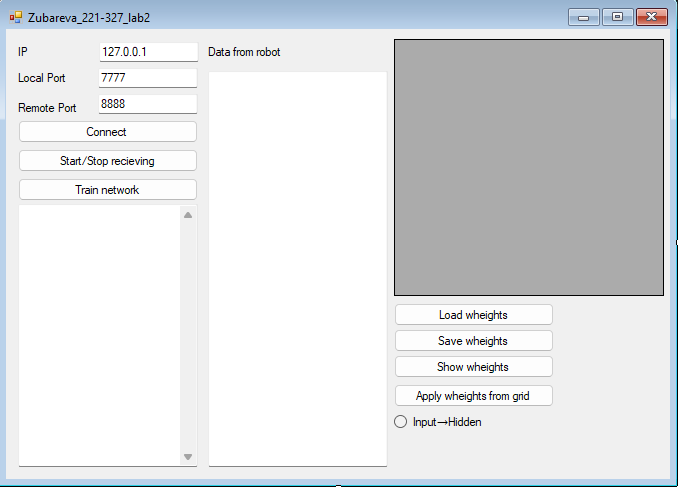


Рисунок 1 – Итоговый вид формы

Рассмотрим подробно функционал приложения. Начнем с класса нейросети. Этот класс NeuralNetwork реализует **простую полносвязную нейросеть с одним скрытым слоем**. Он используется для задач классификации, таких как управление роботом (например, "влево", "вперёд", "вправо"), и включает в себя функционал для обучения, предсказания, сохранения/загрузки весов и манипулирования ими. Вот максимально подробное описание того, что делает каждый блок:

**Структура класса**

* **Поля**:
  + inputSize, hiddenSize, outputSize — размеры слоёв нейросети: входного, скрытого и выходного.
  + weightsInputHidden, weightsHiddenOutput — матрицы весов между слоями.
  + rand — генератор случайных чисел, используется для инициализации весов.

**Конструктор NeuralNetwork(...)**

* Принимает размеры каждого слоя.
* Создаёт двумерные массивы весов.
* Запускает InitializeWeights(), чтобы заполнить веса случайными числами (от -1 до 1).

**Метод InitializeWeights()**

* Заполняет матрицы весов случайными значениями, что необходимо для начала обучения нейросети (иначе все нейроны будут одинаковыми).

**Активационные функции:**

* **ReLU()**: применяется после скрытого слоя. Обнуляет все отрицательные значения (имитируя работу биологических нейронов).
* **Softmax()**: применяется на выходном слое. Преобразует выходы в вероятности (в сумме дают 1), чтобы можно было сделать выбор наиболее вероятного действия.

**MatrixMultiply(...)**

* Реализует перемножение вектора на матрицу (входные данные на веса). Это основа прямого распространения (feedforward).

**Predict(...)**

* Выполняет прямое распространение:
  + Сначала вход → скрытый слой с ReLU
  + Потом скрытый → выходной слой с Softmax
* Возвращает индекс выходного нейрона с максимальной вероятностью (например, 0 = влево, 1 = вперёд, 2 = вправо).

**PredictWithConfidence(...)**

* Почти как Predict, но **возвращает не только выбранное действие**, а ещё **все вероятности** по каждому выходу. Это нужно, если важна степень уверенности.

**SendCommand(...)**

* Преобразует число действия (например, 1) в строку команды для робота: "FORWARD", "LEFT", "RIGHT".

**Train(...)**

Обучает нейросеть на одном примере (один вход и правильное действие):

1. Выполняет **прямое распространение**.
2. Считает **ошибку выхода** (разницу между желаемым и фактическим).
3. Обновляет веса от **скрытого к выходному** слою.
4. Расчёт ошибки для скрытого слоя (обратное распространение).
5. Обновляет веса от **входного к скрытому** слою.

Используется **метод градиентного спуска**.

**TrainOnDataset(...)**

* Обучает нейросеть сразу на всем наборе данных (списке пар "вход → правильное действие") в течение заданного количества эпох (по умолчанию 1000).

**SaveWeights(...) и LoadWeights(...)**

* Сохраняют и загружают веса из/в файл.
* Структура сохранения: сначала веса input→hidden, потом маркер "----", затем веса hidden→output.
* Это позволяет сохранить прогресс обучения между запусками.

**GetWeights() и SetWeights(...)**

* Получают/устанавливают **все веса** как один одномерный массив.
* Удобно использовать для копирования или эволюционных алгоритмов (например, мутации).

**Общий смысл класса**

* Это **базовая модель нейросети**, способная:
  + Принимать входные данные (например, расстояния до препятствий)
  + Выдавать решение (куда двигаться)
  + Обучаться на основе ошибок
  + Сохранять/загружать своё состояние
  + Быть встроенной в более сложную систему (например, симулятор, контроллер робота, игровую среду)

Далее рассмотрим генетический алгоритм нейросети. Этот код реализует **основу эволюционного алгоритма** для обучения нейросетей. Он имитирует естественный отбор: создаётся популяция нейросетей (каждая — это особь, или «геном»), затем они оцениваются по своей пригодности (fitness), лучшие из них отбираются, комбинируются между собой (кроссовер), немного мутируют, и создаётся новое поколение. Цикл повторяется, пока нейросети не начнут показывать нужное поведение.

Давай разберём максимально подробно, что происходит в каждой части:

**Класс Genome**

Это обёртка над нейросетью. Он содержит два поля:

* Network — сама нейросеть (экземпляр NeuralNetwork), которая управляет поведением.
* Fitness — оценка качества этой нейросети. Чем выше, тем "лучше" работает сеть.

**SelectTop(...)**

Функция выбора лучших геномов (нейросетей) из популяции:

* Сортирует популяцию по убыванию значения Fitness.
* Берёт указанное количество самых лучших.
* Возвращает их список.

Это имитация **естественного отбора** — выживают и воспроизводятся только самые "приспособленные".

**CreateInitialPopulation(...)**

Создаёт первую популяцию нейросетей:

* Инициализирует нужное количество нейросетей случайными весами.
* Оборачивает каждую в Genome и добавляет в список.
* Это первый "генерационный пул" до обучения — все особи случайны.

**EvaluateFitness(...)**

Оценивает каждую нейросеть в популяции:

* Применяет к каждой Genome метод TestNetwork.
* Присваивает полученное значение Fitness.
* Это критически важно — без оценки не будет отбора и прогресса.

**TestNetwork(...)**

Оценивает, насколько хорошо конкретная нейросеть справляется с задачей:

* Прогоняет её по всему датасету.
* Если предсказанное действие совпадает с правильным, добавляет балл.
* Чем больше совпадений, тем выше итоговая пригодность.

Это **объективная метрика качества** поведения сети.

**Crossover(...)**

Создаёт потомка двух нейросетей:

* Берёт веса от обеих "родительских" сетей.
* Смешивает их по принципу: с вероятностью 50% берётся значение из одной или другой сети.
* Устанавливает новые веса в новую сеть.
* Это имитация **генетического обмена**, как в биологии.

**Mutate(...)**

Изменяет веса нейросети случайным образом:

* С вероятностью (например, 5%) каждый вес может измениться.
* Изменение небольшое, в пределах указанной амплитуды.
* Это предотвращает застой и помогает алгоритму исследовать новые варианты.

Мутации важны, чтобы алгоритм не «застрял» в локальном максимуме.

**NextGeneration(...)**

Генерирует **новое поколение нейросетей** на основе текущего:

1. **Отбирает лучших (топ-3)** — они переходят в следующее поколение без изменений. Это сохранение элит.
2. **Выбирает лучших (топ-5)** — они участвуют в размножении:
   * Случайные пары из них создают потомков через кроссовер.
   * Потомки мутируют.
   * Добавляются в новое поколение.
3. **Добавляет случайные нейросети (2 шт)** — это нужно для поддержания разнообразия.
4. Возвращает новое поколение.

Это основа **генетического алгоритма**: отбор + кроссовер + мутация + элитизм + случайность.

**Что делает весь этот код в целом?**

Он позволяет:

1. **Создать популяцию нейросетей**.
2. **Оценить, насколько хорошо каждая справляется с задачей**.
3. **Выбрать лучших и создать из них следующее поколение**, улучшая поведение через отбор, кроссовер и мутацию.
4. **Повторять процесс**, пока нейросети не научатся успешно решать задачу.

Далее рассмотрим управление роботом в целом. Этот код — это **управление автономным роботом с помощью нейросети, получающей данные по UDP**. Он включает обработку сенсорных данных, принятие решений, онлайн-обучение и анализ производительности в реальном времени. Всё это происходит в среде, где робот движется к цели и обучается «на лету» на основе своей успешности.

Разберёмся **по частям и максимально подробно**:

**Distance(...)**

Это базовая функция, вычисляющая **евклидово расстояние** между двумя точками (x1, y1) и (x2, y2). Используется для оценки прогресса робота: насколько он приблизился к цели.

**dataset**

Это **обучающая выборка** для нейросети, содержащая пары: входные данные (8 чисел — вероятно, данные с датчиков расстояния) и правильное действие (correctAction). Значения 0, 1, 2 — вероятно, команды **LEFT**, **FORWARD**, **RIGHT**. Эта выборка используется для тестирования и обучения сети.

**Глобальные переменные и компоненты**

* udpClient, udpCommands — клиенты для приёма и отправки данных по UDP.
* iPEndPoint, thread — конфигурация и поток для работы с UDP.
* Много переменных (n, s, d0-d7, X, Y, T и др.) — значения от датчиков или координаты робота.
* commands — словарь, содержащий команды для робота.
* jsonString, decodeText и прочее — данные в JSON-формате, для отправки/приёма и логирования.

**Кнопка BtnTrainNetwork\_Click**

Вызывается при нажатии на кнопку "Train Network":

1. Загружает карту лабиринта из файла.
2. Вызывает TestBestNetwork(dataset) — вероятно, это проверка нейросети на датасете (метод не показан, но, видимо, сравнивает предсказания сети с correctAction и оценивает её точность).

**Метод DecodingData(...)**

1. Получает **UDP-сообщение** и декодирует его из JSON-строки в словарь (decodeText).
2. Выводит данные в интерфейс (в textBoxData).
3. Передаёт словарь в AnalyzeData(...).

**Метод AnalyzeData(...)**

1. Проверяет, есть ли ключ "n" — если есть, значит данные валидны.
2. Присваивает значения из словаря переменным, например d0–d7 (датчики расстояний), X, Y (позиция робота), b (индикатор столкновения) и др.
3. Если данных нет — предупреждает пользователя.

**Метод Receive(...)**

1. Бесконечно слушает входящие UDP-сообщения.
2. При получении сохраняет данные в переменную data.
3. Используется в отдельном потоке.

**Метод Send(...)**

**Самое важное место, где происходит принятие решения ИИ и отправка команд роботу**:

1. Отправляет команды в JSON-формате по UDP (commands, jsonString, и другие).
2. Нормализует сенсорные данные (d0–d7) для нейросети.
3. Получает **предсказание действия от нейросети** и уровень уверенности.
4. Логирует выбранную команду и уверенность (например: "FORWARD: 0.89").
5. Вызывает Robot.Moving(command) — посылает команду физическому роботу.
6. Вычисляет расстояние до цели, время и **оценку (fitness)** на основе прогресса.
7. Проверяет, **был ли удар** (по значению b), останавливает робота при необходимости.
8. Вычисляет **награду (reward)** за улучшение результата.
9. Обновляет score и проверяет, не нужно ли остановить обучение (если баллы сильно упали).
10. Выполняет **онлайн-обучение** нейросети: на текущем примере предсказания/действия.

**Метод DetermineCorrectAction(...)**

Это простая эвристика, определяющая "правильное" действие на основе входных данных — используется как **эталон для онлайн-обучения**:

* Если передний датчик свободен — идём вперёд.
* Если правые датчики свободны — поворачиваем направо.
* Иначе — налево.

**Метод LoadMapToArray(...)**

Загружает карту из файла в виде текстового лабиринта. Преобразует его в двумерный массив:

* # — стена (0),
* всё остальное — проходимое пространство (1).

**Кнопка btnConnect\_Click**

Подключение к роботу:

1. Берёт IP и порт из формы.
2. Создаёт UdpClient.
3. Запускает поток Receive().
4. Включает таймер timer1, который отвечает за **периодическую обработку данных** и отправку команд.

**Метод timer1\_Tick(...)**

Этот метод вызывается **каждые полсекунды** (или с заданным интервалом):

1. Проверяет, получены ли данные.
2. Если да — декодирует их и запускает Send() для принятия решения и обучения.

**Метод CalculateReward(...)**

Вычисляет **награду за прогресс**:

* Вычисляет скорость приближения к цели (изменение расстояния за время).
* Даёт награду за движение в нужную сторону.
* Штрафует за движение назад.
* Добавляет небольшой штраф за время.

Используется для оценки качества действий робота.

**Кнопка btnStartStop\_Click**

Просто включает или выключает таймер, тем самым **запуская или останавливая цикл обработки данных и отправки команд**.

**Что делает система в целом?**

1. Подключается к роботу по UDP.
2. Принимает от него данные с сенсоров (расстояния, положение, столкновения и т.п.).
3. Анализирует данные.
4. Нейросеть принимает решение: куда двигаться (вперёд, влево, вправо).
5. Команда отправляется роботу.
6. Система следит за прогрессом: насколько робот приблизился к цели, не врезался ли.
7. Проводит **самообучение** нейросети на основе логики или текущих данных.
8. Логирует все данные и обеспечивает обратную связь пользователю.

Рассмотрим также функции, отвечающие за загрузку, сохранение и изменение весов нейросети. Этот код — это **интерфейс для визуализации, редактирования, сохранения и загрузки весов нейросети** через таблицу (DataGridView) в Windows Forms. Он позволяет вручную просматривать и изменять веса между слоями нейронной сети, что полезно для отладки, анализа или экспериментов.

Разберём **максимально подробно** каждую часть:

**Метод ShowWeightsInGrid(bool showInputToHidden)**

Этот метод **отображает веса нейросети в таблице**, которая называется dataGridWeights. В зависимости от параметра showInputToHidden он показывает веса:

* **между входным и скрытым слоями** (weightsInputHidden) — если true
* **между скрытым и выходным слоями** (weightsHiddenOutput) — если false

**Как работает:**

1. **Очищает** все столбцы и строки dataGridWeights.
2. Если showInputToHidden == true:
   * Добавляет в таблицу hiddenSize столбцов — по одному на каждый нейрон скрытого слоя (H0, H1, H2, ...).
   * Затем по строкам добавляет веса от каждого входного нейрона ко всем скрытым нейронам.
   * То есть в каждой строке: веса из **одного входного нейрона** во **все скрытые**.
3. Если showInputToHidden == false:
   * Добавляет в таблицу outputSize столбцов (O0, O1, ...).
   * Каждая строка — веса от одного **скрытого нейрона** ко всем **выходным**.

**Метод ApplyWeightsFromGrid(bool applyToInputHidden)**

Этот метод **читает значения из таблицы и применяет их обратно в нейросеть**, заменяя веса на те, которые пользователь мог вручную изменить.

**Как работает:**

1. Если applyToInputHidden == true:
   * Проходит по строкам и столбцам таблицы.
   * Каждую ячейку интерпретирует как вес между соответствующим входным и скрытым нейронами.
   * Присваивает значения в nn.weightsInputHidden[i, j].
2. Если false:
   * То же самое, но для nn.weightsHiddenOutput[i, j] — веса между скрытым и выходным слоями.

**Пример:**

Если в таблице пользователь поменял вес с 0.5 на 0.8 — этот метод «заберёт» новое значение и применит в структуру нейросети.

**Метод btnShowWeights\_Click(...)**

Этот обработчик **вызвается при нажатии кнопки "Показать веса"**:

1. Читает значение из переключателя (radioInputHidden) — то есть какую часть весов пользователь хочет видеть: вход->скрытый или скрытый->выход.
2. Вызывает метод ShowWeightsInGrid(...) с этим значением.

**Метод btnApplyWeightsFromGrid\_Click(...)**

Срабатывает при нажатии кнопки **"Применить веса из таблицы"**:

1. Определяет, к какому слою применять (вход→скрытый или скрытый→выход).
2. Вызывает ApplyWeightsFromGrid(...) — пользовательские значения переносятся в нейросеть.

**Метод btnSaveWeightsToFile\_Click(...)**

Позволяет **сохранить веса нейросети в текстовый файл**:

1. Открывает диалог сохранения файла.
2. Если пользователь выбрал файл — вызывает nn.SaveWeights(...), передавая путь.

**Метод btnLoadWeightsFromFile\_Click(...)**

Позволяет **загрузить веса из файла**:

1. Показывает диалог открытия файла.
2. Если выбран файл — вызывает nn.LoadWeights(...), который читает веса и применяет их в сеть.

**Что это всё вместе делает?**

Этот код создаёт **графический интерфейс для взаимодействия с весами нейросети**. Пользователь может:

* Просматривать текущие веса между слоями нейросети.
* Редактировать их вручную (например, для тестов).
* Сохранять и загружать веса для разных конфигураций.
* Изменять веса «на лету», не перекомпилируя программу.

Последний класс – это класс самого робота. Этот код представляет класс Robot с единственным методом Moving, который управляет движением робота на основе переданной команды.

Вот **подробное описание логики метода**:

**Общая цель**

Метод Moving получает строковую команду (например, "FORWARD", "LEFT", "RIGHT" и т.д.) и, в зависимости от неё, **устанавливает значения параметров движения**, которые хранятся в словаре commands. Эти параметры, скорее всего, позже используются для управления физическим роботом или его виртуальным аналогом.

**Что делает метод по шагам:**

1. **Формирует текст сообщения**, в котором указывается текущая команда. Это сообщение сохраняется в глобальной переменной message.
2. **Создаёт локальные переменные B и F**:
   * F (Forward) отвечает за движение вперёд.
   * B вращение/смещение.
3. **Реагирует на тип команды**:
   * Если команда — "FORWARD":
     + F устанавливается в 100 (вперёд на полную скорость),
     + B — в 0 (без поворота).
   * Если команда — "LEFT":
     + F устанавливается в 10 (медленное движение),
     + B — в -100 (поворот влево).
   * Если команда — "RIGHT":
     + F — 10,
     + B — 100 (поворот вправо).
   * Любая другая команда:
     + F и B обнуляются — робот не двигается.
4. **Сохраняет рассчитанные значения** в глобальный словарь commands:
   * Ключ "B" получает значение B.
   * Ключ "F" — значение F.
   * Ключ "N" увеличивается на 1. Это, по сути, счётчик количества команд (или шагов), которые были выданы роботу.

**Назначение**

Метод позволяет **интерпретировать текстовую команду в числовые значения управления**, понятные низкоуровневому интерфейсу робота. Эти числовые значения потом, скорее всего, сериализуются и передаются на устройство через, например, UDP, как в твоём другом коде.

**Вывод.**

**Общий обзор проекта**

Разработана **интеллектуальная система управления роботом**, основанная на **нейросетевом подходе с возможностью обучения и эволюции**. Система может:

* Получать данные с датчиков в реальном времени по UDP.
* Обрабатывать эти данные и передавать команды роботу.
* Использовать нейросеть для выбора действий на основе входных параметров.
* Оценивать эффективность действий и обучаться на лету.
* Отображать и редактировать веса нейросети вручную через интерфейс.
* Сохранять и загружать веса из файлов.
* Проводить эволюцию популяции нейросетей — отбор, кроссовер, мутация.

**Нейросеть и обучение**

1. **Нейросеть** обучается на основе датасета, содержащего пары «вход — правильное действие».
2. Веса нейросети можно:
   * визуализировать,
   * редактировать вручную,
   * сохранить в файл или загрузить обратно.
3. Реализовано **обучение в реальном времени**: после каждого шага робот может быть обучен на корректное действие.
4. Вводятся метрики качества, такие как:
   * расстояние до цели,
   * время, потраченное на достижение цели,
   * фитнес-функция, оценивающая эффективность.

**Управление роботом**

1. Команды для движения робота формируются на основе вывода нейросети.
2. Команды представляют собой числовые значения (например, мощность двигателей), которые отправляются через UDP.
3. Реализована логика безопасного управления:
   * реакция на столкновение,
   * остановка при слишком низком прогрессе (наградный механизм),
   * перезапуск при необходимости.
4. Есть механизм логирования действий, вероятностей выбора и наград.

**Генетический алгоритм**

1. Система поддерживает **эволюционный подход** к обучению нейросетей:
   * создаётся начальная популяция;
   * оценивается фитнес;
   * отбираются лучшие;
   * проводится кроссовер и мутация;
   * создаётся новое поколение.
2. Этот подход позволяет найти хорошие архитектуры и параметры даже без ручного обучения.

**Сеть и взаимодействие**

1. Система активно обменивается данными по **UDP-соединению**:
   * получает данные с датчиков (в формате JSON);
   * отправляет команды роботу;
   * поддерживает интерфейс пользователя с возможностью подключения.
2. Есть реализация асинхронного приёма и отправки данных.
3. Информация отображается в интерфейсе: входящие данные, действия нейросети, награды и ошибки.

**Пользовательский интерфейс**

1. Пользователь может:
   * начать или остановить работу системы;
   * вручную изменить веса нейросети;
   * загрузить или сохранить конфигурации;
   * видеть действия нейросети и оценку решений в виде текстовых логов.
2. Также реализована загрузка карты лабиринта, возможно, для симуляции или планирования маршрута.

**Оценка и логика обучения**

1. Метод DetermineCorrectAction — это зачаток **рефлексивного поведения**, где система может сравнивать свои действия с "разумными".
2. Используется **фитнес-функция**, которая penalизирует медленное или неэффективное движение.
3. Система накапливает очки (score), оценивая себя на каждом шаге, и может остановиться, если производительность падает.

**Приложение А: Листинг кода**

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Net.Sockets;

using System.Net;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

using Newtonsoft.Json;

using System.IO;

using Newtonsoft.Json.Linq;

using static System.Net.WebRequestMethods;

using static Zubareva\_221\_327\_lab2\_neuro.Form1;

using static System.Windows.Forms.LinkLabel;

using System.Drawing.Imaging;

using System.Runtime.Serialization.Formatters.Binary;

using static System.Windows.Forms.VisualStyles.VisualStyleElement.Button;

using static System.Windows.Forms.VisualStyles.VisualStyleElement;

using System.Threading;

namespace Zubareva\_221\_327\_lab2\_neuro

{

public partial class Form1 : Form

{

public class NeuralNetwork

{

public int inputSize, hiddenSize, outputSize;

public float[,] weightsInputHidden;

public float[,] weightsHiddenOutput;

public Random rand = new Random();

public NeuralNetwork(int inputSize, int hiddenSize, int outputSize)

{

this.inputSize = inputSize;

this.hiddenSize = hiddenSize;

this.outputSize = outputSize;

weightsInputHidden = new float[inputSize, hiddenSize];

weightsHiddenOutput = new float[hiddenSize, outputSize];

InitializeWeights();

}

public void TrainOnDataset(List<(float[] input, int correctAction)> dataset, int epochs = 1000)

{

for (int e = 0; e < epochs; e++)

{

foreach (var (input, action) in dataset)

{

Train(input, action);

}

}

}

private void InitializeWeights()

{

for (int i = 0; i < inputSize; i++)

for (int j = 0; j < hiddenSize; j++)

weightsInputHidden[i, j] = (float)(rand.NextDouble() \* 2 - 1);

for (int i = 0; i < hiddenSize; i++)

for (int j = 0; j < outputSize; j++)

weightsHiddenOutput[i, j] = (float)(rand.NextDouble() \* 2 - 1);

}

private float[] ReLU(float[] x)

{

float[] output = new float[x.Length];

for (int i = 0; i < x.Length; i++)

output[i] = Math.Max(0, x[i]);

return output;

}

private float[] Softmax(float[] x)

{

float sum = 0;

float[] output = new float[x.Length];

for (int i = 0; i < x.Length; i++)

sum += (float)Math.Exp(x[i]);

for (int i = 0; i < x.Length; i++)

output[i] = (float)Math.Exp(x[i]) / sum;

return output;

}

private float[] MatrixMultiply(float[] input, float[,] weights)

{

int rows = weights.GetLength(0);

int cols = weights.GetLength(1);

float[] output = new float[cols];

for (int j = 0; j < cols; j++)

{

for (int i = 0; i < rows; i++)

output[j] += input[i] \* weights[i, j];

}

return output;

}

public int Predict(float[] input)

{

float[] hiddenLayer = ReLU(MatrixMultiply(input, weightsInputHidden));

float[] outputLayer = Softmax(MatrixMultiply(hiddenLayer, weightsHiddenOutput));

int bestAction = 0;

float maxVal = outputLayer[0];

for (int i = 1; i < outputLayer.Length; i++)

{

if (outputLayer[i] > maxVal)

{

maxVal = outputLayer[i];

bestAction = i;

}

}

return bestAction;

}

public string SendCommand(int action)

{

string command = "FORWARD"; // Значение по умолчанию

if (action == 0)

command = "LEFT";

else if (action == 1)

command = "FORWARD";

else if (action == 2)

command = "RIGHT";

return command;

}

public void SaveWeights(string filePath)

{

using (StreamWriter writer = new StreamWriter(filePath))

{

for (int i = 0; i < inputSize; i++)

{

for (int j = 0; j < hiddenSize; j++)

{

writer.Write(weightsInputHidden[i, j] + " ");

}

writer.WriteLine();

}

writer.WriteLine("----");

for (int i = 0; i < hiddenSize; i++)

{

for (int j = 0; j < outputSize; j++)

{

writer.Write(weightsHiddenOutput[i, j] + " ");

}

writer.WriteLine();

}

}

}

public (int action, float[] probabilities) PredictWithConfidence(float[] input)

{

float[] hiddenLayer = ReLU(MatrixMultiply(input, weightsInputHidden));

float[] outputLayer = Softmax(MatrixMultiply(hiddenLayer, weightsHiddenOutput));

int bestAction = 0;

float maxVal = outputLayer[0];

for (int i = 1; i < outputLayer.Length; i++)

{

if (outputLayer[i] > maxVal)

{

maxVal = outputLayer[i];

bestAction = i;

}

}

return (bestAction, outputLayer);

}

public void LoadWeights(string filePath)

{

if (!System.IO.File.Exists(filePath))

{

Console.WriteLine("Файл весов не найден. Используются случайные значения.");

return;

}

using (StreamReader reader = new StreamReader(filePath))

{

string line;

int i = 0, j = 0;

bool readingHiddenLayer = true;

while ((line = reader.ReadLine()) != null)

{

if (line == "----")

{

readingHiddenLayer = false;

i = 0;

continue;

}

string[] values = line.Split(' ');

if (readingHiddenLayer)

{

for (j = 0; j < hiddenSize; j++)

weightsInputHidden[i, j] = float.Parse(values[j]);

}

else

{

for (j = 0; j < outputSize; j++)

weightsHiddenOutput[i, j] = float.Parse(values[j]);

}

i++;

}

}

}

public float[] GetWeights()

{

int weightCount = (inputSize \* hiddenSize) + (hiddenSize \* outputSize);

float[] weights = new float[weightCount];

int index = 0;

// Копируем веса между входом и скрытым слоем

for (int i = 0; i < inputSize; i++)

{

for (int j = 0; j < hiddenSize; j++)

{

weights[index++] = weightsInputHidden[i, j];

}

}

// Копируем веса между скрытым слоем и выходом

for (int i = 0; i < hiddenSize; i++)

{

for (int j = 0; j < outputSize; j++)

{

weights[index++] = weightsHiddenOutput[i, j];

}

}

return weights;

}

public void SetWeights(float[] weights)

{

int index = 0;

// Устанавливаем веса между входом и скрытым слоем

for (int i = 0; i < inputSize; i++)

{

for (int j = 0; j < hiddenSize; j++)

{

weightsInputHidden[i, j] = weights[index++];

}

}

// Устанавливаем веса между скрытым слоем и выходом

for (int i = 0; i < hiddenSize; i++)

{

for (int j = 0; j < outputSize; j++)

{

weightsHiddenOutput[i, j] = weights[index++];

}

}

}

public void Train(float[] input, int correctAction, float learningRate = 0.01f)

{

// Прямое распространение

float[] hiddenInput = MatrixMultiply(input, weightsInputHidden);

float[] hiddenOutput = ReLU(hiddenInput);

float[] finalInput = MatrixMultiply(hiddenOutput, weightsHiddenOutput);

float[] finalOutput = Softmax(finalInput);

// Ошибка на выходе

float[] target = new float[outputSize];

target[correctAction] = 1; // только правильный класс = 1

float[] outputErrors = new float[outputSize];

for (int i = 0; i < outputSize; i++)

outputErrors[i] = target[i] - finalOutput[i];

// Обновление весов: hidden → output

for (int i = 0; i < hiddenSize; i++)

{

for (int j = 0; j < outputSize; j++)

{

weightsHiddenOutput[i, j] += learningRate \* outputErrors[j] \* hiddenOutput[i];

}

}

// Ошибка на скрытом слое

float[] hiddenErrors = new float[hiddenSize];

for (int i = 0; i < hiddenSize; i++)

{

float error = 0;

for (int j = 0; j < outputSize; j++)

{

error += outputErrors[j] \* weightsHiddenOutput[i, j];

}

// Применяем производную ReLU

hiddenErrors[i] = hiddenInput[i] > 0 ? error : 0;

}

// Обновление весов: input → hidden

for (int i = 0; i < inputSize; i++)

{

for (int j = 0; j < hiddenSize; j++)

{

weightsInputHidden[i, j] += learningRate \* hiddenErrors[j] \* input[i];

}

}

}

}

public class Genome

{

public NeuralNetwork Network { get; set; }

public double Fitness { get; set; }

}

public List<Genome> SelectTop(List<Genome> population, int count)

{

return population.OrderByDescending(g => g.Fitness).Take(count).ToList();

}

// Метод для создания начальной популяции нейросетей

private List<Genome> CreateInitialPopulation(int size, int inputSize, int hiddenSize, int outputSize)

{

var population = new List<Genome>();

for (int i = 0; i < size; i++)

{

var net = new NeuralNetwork(inputSize, hiddenSize, outputSize);

population.Add(new Genome { Network = net });

}

return population;

}

// Метод для оценки фитнеса нейросетей в популяции

private void EvaluateFitness(List<Genome> population, List<(float[] input, int correctAction)> dataset)

{

foreach (var genome in population)

{

genome.Fitness = TestNetwork(genome.Network, dataset);

}

}

// Метод для тестирования нейросети и расчёта её фитнеса

private double TestNetwork(NeuralNetwork network, List<(float[] input, int correctAction)> dataset)

{

double totalFitness = 0;

foreach (var (input, correctAction) in dataset)

{

int predictedAction = network.Predict(input);

if (predictedAction == correctAction)

{

totalFitness += 1;

}

}

return totalFitness;

}

// Метод для выбора лучших нейросетей в популяции

// Метод для кроссовера двух нейросетей (смешивание их весов)

// Метод для кроссовера двух нейросетей (смешивание их весов)

// Метод для кроссовера двух нейросетей

private NeuralNetwork Crossover(NeuralNetwork parent1, NeuralNetwork parent2)

{

var child = new NeuralNetwork(parent1.inputSize, parent1.hiddenSize, parent1.outputSize);

var weights1 = parent1.GetWeights();

var weights2 = parent2.GetWeights();

var newWeights = new float[weights1.Length];

var rand = new Random();

for (int i = 0; i < newWeights.Length; i++)

{

newWeights[i] = rand.NextDouble() < 0.5 ? weights1[i] : weights2[i];

}

child.SetWeights(newWeights);

return child;

}

// Метод для мутации нейросети

private void Mutate(NeuralNetwork net, double mutationRate = 0.05, double mutationAmount = 0.1)

{

var rand = new Random();

var weights = net.GetWeights();

for (int i = 0; i < weights.Length; i++)

{

if (rand.NextDouble() < mutationRate)

{

weights[i] += (float)((rand.NextDouble() \* 2 - 1) \* mutationAmount);

}

}

net.SetWeights(weights);

}

// Метод для создания нового поколения на основе старого

private List<Genome> NextGeneration(List<Genome> current)

{

var next = new List<Genome>();

var top3 = SelectTop(current, 3);

next.AddRange(top3);

var best5 = SelectTop(current, 5);

var rand = new Random();

for (int i = 0; i < 5; i++)

{

var parent1 = best5[rand.Next(best5.Count)];

var parent2 = best5[rand.Next(best5.Count)];

var childNet = Crossover(parent1.Network, parent2.Network);

Mutate(childNet);

next.Add(new Genome { Network = childNet });

}

for (int i = 0; i < 2; i++)

{

var net = new NeuralNetwork(current[0].Network.inputSize, current[0].Network.hiddenSize, current[0].Network.outputSize);

net.GetWeights();

next.Add(new Genome { Network = net });

}

return next;

}

public Form1()

{

InitializeComponent();

nn = new NeuralNetwork(8, 10, 3);

nn.TrainOnDataset(dataset, epochs: 100);

RunEvolution(generations: 50, populationSize: 30, dataset);

}

IPAddress ipAddress;

public static int port, received, sec = 0;

byte[] data;

NeuralNetwork nn;

private float[] NormalizeDistances(float d0, float d1, float d2, float d3, float d4, float d5, float d6, float d7, float maxDistance = 1500f)

{

return new float[]

{

d0 / maxDistance,

d1 / maxDistance,

d2 / maxDistance,

d3 / maxDistance,

d4 / maxDistance,

d5 / maxDistance,

d6 / maxDistance,

d7 / maxDistance

};

}

private void RunEvolution(int generations, int populationSize, List<(float[] input, int correctAction)> dataset)

{

int inputSize = dataset[0].input.Length;

int hiddenSize = 16; // по желанию

int outputSize = 3;

var population = CreateInitialPopulation(populationSize, inputSize, hiddenSize, outputSize);

for (int gen = 0; gen < generations; gen++)

{

EvaluateFitness(population, dataset);

var bestFitness = population.Max(g => g.Fitness);

Console.WriteLine($"Поколение {gen + 1}: Лучшая пригодность = {bestFitness}");

// (опционально) отобрази результат или сохрани лучшую сеть

population = NextGeneration(population);

}

var best = SelectTop(population, 1).First();

best.Network.SaveWeights("best\_weights.txt");

}

private void TestBestNetwork(List<(float[] input, int correctAction)> dataset)

{

var nn = new NeuralNetwork(dataset[0].input.Length, 16, 3);

nn.LoadWeights("best\_weights.txt");

double fitness = TestNetwork(nn, dataset);

MessageBox.Show($"Точность сети: {fitness}/{dataset.Count} ({(fitness / dataset.Count \* 100):F2}%)");

}

float score = 0.0f;

float previousDistance = -1f;

bool isStopped = false;

float stopThreshold = -10f;

float goalX = 15.15f;

float goalY = 9.15f;

float maxDistance = 20f; // максимально возможное расстояние в лабиринте

float timePenaltyFactor = 0.1f;

float startTime; // для подсчёта времени

private float Distance(float x1, float y1, float x2, float y2)

{

return (float)Math.Sqrt(Math.Pow(x2 - x1, 2) + Math.Pow(y2 - y1, 2));

}

List<(float[] input, int correctAction)> dataset = new List<(float[], int)>

{

//(new float[] {0.1f, 0.2f, 0.4f, 0.9f, 0.7f, 0.6f, 0.2f, 0.1f}, 1),

(new float[] {0.1f, 0.2f, 0.4f, 0.9f, 0.7f, 0.6f, 0.2f, 0.9f}, 1),

(new float[] {0.1f, 0.2f, 0.7f, 0.9f, 0.7f, 0.3f, 0.2f, 0.1f}, 1),

(new float[] {0.1f, 0.1f, 0.9f, 0.9f, 0.9f, 0.2f, 0.2f, 0.1f}, 1),

(new float[] {0.2f, 0.3f, 0.2f, 0.1f, 0.1f, 0.6f, 0.8f, 0.9f}, 2),

(new float[] {0.2f, 0.3f, 0.1f, 0.1f, 0.7f, 0.8f, 0.8f, 0.9f}, 2),

(new float[] {0.2f, 0.3f, 0.2f, 0.1f, 0.1f, 0.9f, 0.9f, 0.9f}, 2),

(new float[] {0.1f, 0.1f, 0.1f, 0.1f, 0.9f, 0.9f, 0.9f, 0.9f}, 2),

(new float[] {0.9f, 0.8f, 0.7f, 0.1f, 0.2f, 0.2f, 0.1f, 0.1f}, 0),

(new float[] {0.9f, 0.8f, 0.5f, 0.1f, 0.1f, 0.2f, 0.3f, 0.2f}, 0),

(new float[] {0.9f, 0.5f, 0.5f, 0.1f, 0.1f, 0.2f, 0.3f, 0.2f}, 0),

(new float[] {0.9f, 0.8f, 0.9f, 0.9f, 0.1f, 0.1f, 0.1f, 0.1f}, 0),

};

IPEndPoint iPEndPoint;

Thread thread;

public static float n, s, c, le, re, az, b, d0, d1, d2, d3, d4, d5, d6, d7, l0, l1, l2, l3, l4, sp, X, Y, T;

private void BtnTrainNetwork\_Click(object sender, EventArgs e)

{

int[,] map = LoadMapToArray("labyrinth.txt");

TestBestNetwork(dataset);

}

UdpClient udpClient;

UdpClient udpCommands = new UdpClient();

Dictionary<string, float> decodeText;

Dictionary<string, string> decodeTextstr = new Dictionary<string, string>();

public static Dictionary<string, int> commands = new Dictionary<string, int>

{

{ "N", 0 },

{ "M", 0 },

{ "F", 0 },

{ "B", 0 },

{ "T", 0 },

};

public static string jsonString, jsonString2, message;

private void DecodingData(byte[] data)

{

var message = Encoding.ASCII.GetString(data);

decodeText = JsonConvert.DeserializeObject<Dictionary<string, float>>(message);

var lines = decodeText.Select(kv => kv.Key + ": " + kv.Value.ToString());

textBoxData.Text = "IoT: " + string.Join(Environment.NewLine, lines);

AnalyzeData(decodeText);

}

private void AnalyzeData(Dictionary<string, float> pairs)

{

if (pairs.ContainsKey("n"))

{

n = pairs["n"];

s = pairs["s"];

c = pairs["c"];

le = pairs["le"];

re = pairs["re"];

az = pairs["az"];

b = pairs["b"];

d0 = pairs["d0"];

d1 = pairs["d1"];

d2 = pairs["d2"];

d3 = pairs["d3"];

d4 = pairs["d4"];

d5 = pairs["d5"];

d6 = pairs["d6"];

d7 = pairs["d7"];

l0 = pairs["l0"];

l1 = pairs["l1"];

l2 = pairs["l2"];

l3 = pairs["l3"];

l4 = pairs["l4"];

sp = pairs["sp"];

X = pairs["x"];

Y = pairs["y"];

T = pairs["t"];

}

else

{

MessageBox.Show("No data");

}

}

private void Receive()

{

while (true)

{

try

{

data = udpClient.Receive(ref iPEndPoint);

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

}

private void Send()

{

IPEndPoint pointServer = new IPEndPoint(ipAddress, 8888);

InformTextbox.Text = message;

SerializeCommands(commands);

decodeText["n"] = commands["N"];

string oldCommands = JsonConvert.SerializeObject(decodeText, Formatting.None);

byte[] data = Encoding.ASCII.GetBytes(oldCommands + "\n");

udpCommands.Send(data, data.Length, pointServer);

byte[] dataForRobot = Encoding.ASCII.GetBytes(jsonString + "\n");

udpCommands.Send(dataForRobot, dataForRobot.Length, pointServer);

float[] inputs = NormalizeDistances(d0, d1, d2, d3, d4, d5, d6, d7);

// Получаем предсказание и вероятности

var (action, probabilities) = nn.PredictWithConfidence(inputs);

string command = nn.SendCommand(action);

// Логируем вероятности и решение

InformTextbox.Text += $"\r\n[AI] Command: {command} ({action})";

InformTextbox.Text += $"\r\n[AI] Confidence: LEFT={probabilities[0]:0.00}, FORWARD={probabilities[1]:0.00}, RIGHT={probabilities[2]:0.00}";

Robot.Moving(command);

// Автообучение — опционально, можно обучить на "разумной" логике

int correctAction = DetermineCorrectAction(inputs); // <- ты можешь сам написать этот метод!

nn.Train(inputs, correctAction); // обучение на лету

float distanceToGoal = Distance(X, Y, goalX, goalY);

float timeSpent = (Environment.TickCount / 1000f) - startTime;

float fitness = maxDistance - distanceToGoal - (timePenaltyFactor \* timeSpent);

InformTextbox.Text += $"\r\n[AI] Fitness: {fitness:0.00}";

// Робот двигается

if (isStopped)

{

InformTextbox.Text += "\r\n[STOPPED] Робот остановлен. Нажми Start заново.";

return;

}

float currentDistance = Distance(X, Y, goalX, goalY);

float timeNow = Environment.TickCount / 1000f;

float deltaTime = 1f / 2f; // таймер тикает примерно 2 раза в секунду

// Проверка на столкновение

if (b == 1)

{

isStopped = true;

InformTextbox.Text += "\r\n[ALERT] Столкновение! Нажми Start, чтобы попробовать заново.";

decodeTextstr["TEXT"] = "RESTART";

return;

}

// Первая итерация

if (previousDistance < 0)

previousDistance = currentDistance;

// Вычисляем награду

float reward = CalculateReward(currentDistance, timeSpent, deltaTime);

score += reward;

InformTextbox.Text += $"\r\n[REWARD] ΔScore: {reward:0.00}, Total: {score:0.00}";

// Сохраняем дистанцию на след. тик

previousDistance = currentDistance;

// Проверка на провал

if (score < stopThreshold)

{

isStopped = true;

textBoxData.Text += "\r\n[FAIL] Робот слишком долго не может пройти. Очки в минусе.";

decodeTextstr["TEXT"] = "RESTART";

return;

}

}

private void SerializeCommands(Dictionary<string, int> pairs)

{

jsonString = JsonConvert.SerializeObject(pairs, Formatting.None);

}

private float CalculateReward(float currentDistance, float timeSpent, float deltaTime)

{

float speed = (previousDistance - currentDistance) / deltaTime;

float reward = 0f;

if (speed > 0)

{

reward += speed \* 10; // награда за приближение

}

else

{

reward += speed \* 20; // штраф за удаление

}

reward -= timePenaltyFactor \* deltaTime; // небольшой штраф за время

return reward;

}

private void btnConnect\_Click(object sender, EventArgs e)

{

ipAddress = IPAddress.Parse(textBoxIP.Text);

port = Int32.Parse(textBoxLocalPort.Text);

iPEndPoint = new IPEndPoint(ipAddress, port);

udpClient = new UdpClient(port);

thread = new Thread(() => Receive());

thread.Start();

Thread.Sleep(1000);

timer1.Enabled = true;

timer1.Start();

startTime = Environment.TickCount / 1000f;

}

private async void timer1\_Tick(object sender, EventArgs e)

{

if (data != null)

{

DecodingData(data);

// Отправка по UDP

Send();

}

else

{

textBoxData.Text += "\r\n[ERROR] No data";

}

}

private int DetermineCorrectAction(float[] inputs)

{

if (inputs[0] > 0.7f)

return 1; // FORWARD

else if (inputs[3] > 0.6f && inputs[4] > 0.6f)

return 2; // RIGHT

else

return 0; // LEFT

}

private int[,] LoadMapToArray(string fileName)

{

string[] lines = System.IO.File.ReadAllLines(fileName);

int rows = lines.Length;

int cols = lines[0].Length;

int[,] mapArray = new int[rows, cols];

for (int y = 0; y < rows; y++)

{

for (int x = 0; x < cols; x++)

{

mapArray[y, x] = (lines[y][x] == '#') ? 0 : 1;

}

}

return mapArray;

}

private void btnStartStop\_Click(object sender, EventArgs e)

{

if (timer1.Enabled == true)

{

timer1.Enabled = false;

timer1.Stop();

}

else

{

timer1.Enabled = true;

timer1.Start();

}

}

private void ShowWeightsInGrid(bool showInputToHidden)

{

dataGridWeights.Columns.Clear();

dataGridWeights.Rows.Clear();

if (showInputToHidden)

{

for (int j = 0; j < nn.hiddenSize; j++)

{

dataGridWeights.Columns.Add("H" + j, "H" + j);

}

for (int i = 0; i < nn.inputSize; i++)

{

var row = new DataGridViewRow();

row.CreateCells(dataGridWeights);

for (int j = 0; j < nn.hiddenSize; j++)

{

row.Cells[j].Value = nn.weightsInputHidden[i, j];

}

dataGridWeights.Rows.Add(row);

}

}

else

{

for (int j = 0; j < nn.outputSize; j++)

{

dataGridWeights.Columns.Add("O" + j, "O" + j);

}

for (int i = 0; i < nn.hiddenSize; i++)

{

var row = new DataGridViewRow();

row.CreateCells(dataGridWeights);

for (int j = 0; j < nn.outputSize; j++)

{

row.Cells[j].Value = nn.weightsHiddenOutput[i, j];

}

dataGridWeights.Rows.Add(row);

}

}

}

private void ApplyWeightsFromGrid(bool applyToInputHidden)

{

if (applyToInputHidden)

{

for (int i = 0; i < nn.inputSize; i++)

{

for (int j = 0; j < nn.hiddenSize; j++)

{

nn.weightsInputHidden[i, j] = float.Parse(dataGridWeights.Rows[i].Cells[j].Value.ToString());

}

}

}

else

{

for (int i = 0; i < nn.hiddenSize; i++)

{

for (int j = 0; j < nn.outputSize; j++)

{

nn.weightsHiddenOutput[i, j] = float.Parse(dataGridWeights.Rows[i].Cells[j].Value.ToString());

}

}

}

}

private void btnShowWeights\_Click(object sender, EventArgs e)

{

bool showInputHidden = radioInputHidden.Checked; // или comboBox.SelectedItem

ShowWeightsInGrid(showInputHidden);

}

private void btnApplyWeightsFromGrid\_Click(object sender, EventArgs e)

{

bool toInputHidden = radioInputHidden.Checked;

ApplyWeightsFromGrid(toInputHidden);

}

private void btnSaveWeightsToFile\_Click(object sender, EventArgs e)

{

SaveFileDialog sfd = new SaveFileDialog();

sfd.Filter = "Text Files|\*.txt";

if (sfd.ShowDialog() == DialogResult.OK)

{

nn.SaveWeights(sfd.FileName);

}

}

private void btnLoadWeightsFromFile\_Click(object sender, EventArgs e)

{

OpenFileDialog ofd = new OpenFileDialog();

ofd.Filter = "Text Files|\*.txt";

if (ofd.ShowDialog() == DialogResult.OK)

{

nn.LoadWeights(ofd.FileName);

}

}

public class Robot

{

public static void Moving(string command)

{

message = "Current command is" + command;

int B = 0, F = 0;

if (command == "FORWARD")

{

F = 100;

B = 0;

}

else if (command == "LEFT")

{

F = 10;

B = -100;

}

else if (command == "RIGHT")

{

F = 10;

B = 100;

}

else

{

F = 0;

B = 0;

}

commands["B"] = B;

commands["F"] = F;

commands["N"] += 1;

}

}

}

}