МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ

Ярославский государственный университет им. П.Г. Демидова

Факультет информатики и вычислительной техники

**ОБУЧЕНИЕ НЕЙРОННОЙ СЕТИ ДЛЯ СОЗДАНИЯ ИССКУСТВЕННОГО ИНТЕЛЕКТА В ИГРАХ**

**Студент:**

Данилов Максим Владимирович

3 курс, группа ПИЭ-32БО

**Руководитель:**

Старший преподаватель

кафедры информационных и сетевых технологий

Легков Николай Васильевич

г. Ярославль

2022 г.

Оглавление

[ВВЕДЕНИЕ 3](#_Toc138017062)

[1. ТЕХНОЛОГИИ НЕЙРОННЫХ СЕТЕЙ 5](#_Toc138017063)

[2. ГЕНЕТИЧЕСКИЙ АЛГОРИТМ ДЛЯ ОБУЧЕНИЯ НЕЙРОННОЙ СЕТИ 12](#_Toc138017064)

[2.1. Основные понятия 12](#_Toc138017065)

[2.2. Операторы генетического алгоритма 13](#_Toc138017066)

[2.3. Обучение нейронной сети с помощью генетического алгоритма 14](#_Toc138017067)

[2.4. Преимущества и недостатки генетического алгоритма 17](#_Toc138017068)

[3. ТОПОЛОГИИ НЕЙРОННЫХ СЕТЕЙ. 20](#_Toc138017069)

[3.1. Многослойный перцептрон. 20](#_Toc138017070)

[3.2. Рекуррентные нейронные сети. 21](#_Toc138017071)

[3.3. Сверточные нейронные сети 23](#_Toc138017072)

[4. ПРИМЕНЕНИЕ НЕЙРОННЫХ СЕТЕЙ В КОМПЬЮТЕРНЫХ ИГРАХ 27](#_Toc138017073)

[4.1. Возможности нейронных сетей в игровой индустрии. 27](#_Toc138017074)

[4.2. Улучшение графики с помощью ИИ. 27](#_Toc138017075)

[4.3. Создание игровых миров и уровней с помощью нейросетей. 29](#_Toc138017076)

[5. МОЙ ПРОЕКТ 31](#_Toc138017077)

[5.1. Общее представление. 31](#_Toc138017078)

[5.2. Общий вид программы и интерфейс 33](#_Toc138017079)

[5.3. Проделанная работа по обучению нейронной сети 34](#_Toc138017080)

[5.4. Описание классов и функций. 35](#_Toc138017081)

[Список использованных источников 41](#_Toc138017082)

[Приложение А 42](#_Toc138017083)

[Листинг 1. Файл NeuralNetwork.cs 42](#_Toc138017084)

[Листинг 2. Файл Manager.cs 47](#_Toc138017085)

[Листинг 3. Файл Bot.cs 50](#_Toc138017086)

**ВВЕДЕНИЕ**

Искусственная нейронная сеть — это совокупность связей между нейронами. Концепция нейронной сети берет свое начало в биологии. Термин "нейронная сеть" появился в работах исследователей в начале 1940-х годов. Исследователи пришли к выводу, что проблемы можно решать с помощью алгоритмов, основанных на работе человеческого мозга, и в 1960-х годах начали появляться первые реализации этих алгоритмов.

Нейронные сети — это инструменты, используемые в искусственном интеллекте для обработки данных подобно человеческому мозгу. Этот тип машинного обучения, известный как глубокое обучение, основан на взаимодействии узлов или нейронов в слоистой структуре, которая приближается к организации мозга. Этот механизм позволяет компьютерам создавать адаптивные системы, которые могут учиться на своих ошибках и улучшать свою работу.

Нейронную сеть можно описать как систему простых искусственных нейронов, которые связаны между собой и взаимодействуют. Эти нейроны обычно довольно просты.

Каждый нейрон принимает на вход некоторые данные, точнее, он считает взвешенную сумму своих входов, добавляет смещение, обрабатывает ее с помощью функции активации, отбрасывает или дорабатывает результат и передает его на выходной слой, и так далее. Каждый нейрон имеет свой собственный вес для оценки значения входного сигнала. После обработки входных данных нейронная сеть выдает полученную информацию через выходной слой.

**Объектом исследования** является процесс обучения нейронных сетей в играх.

**Предметом** **исследования** является нейронные сети, ориентированные на игры с использованием искусственного интеллекта.

**Цель работы** состоит в разработке алгоритма обучения нейронной сети для создания искусственного интеллекта в играх.

**Задачи:**

* Изучить работу нейронных сетей.
* Исследовать виды нейронных сетей, чтобы найти подходящую для проекта.
* Создать игру со встроенным искусственным интеллектом на основе нейронной сети.

В результате проделанной работы был разработан эффективный искусственный интеллект. Было создано и проведено тестирование игровой среды, в которой в качестве оппонента выступала нейронная сеть, обученная с использованием генетического алгоритма. Также была разработана игровая среда для обучения нейронных сетей, позволяющая контролировать процесс обучения и анализировать результаты. Разработанный алгоритм способен обучать нейронную сеть на основе выполнения заданных команд и целей, что позволяет создавать более эффективный искусственный интеллект для игр.

**Обоснование важности применения нейронных сетей в компьютерных играх**

В настоящее время нейронные сети используются в различных областях, включая медицину, финансы, производство и технологии, а также науку. В медицине, например, нейронные сети используются для анализа медицинских данных и выявления тенденций, которые могут помочь улучшить уход за пациентами.

В финансовой сфере нейронные сети помогают принимать решения о капиталовложениях и определять потенциальные риски для инвесторов. В науке нейронные сети используются для прогнозирования погоды, изучения механизмов генетических заболеваний и создания искусственного интеллекта для управления спутниками.

В компьютерных играх нейронные сети используются для создания искусственного интеллекта для персонажей, управления игровыми объектами и принятия решений.

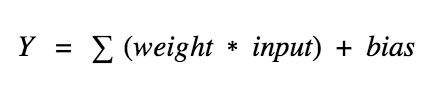
Применение нейронных сетей в компьютерных играх - важный шаг в развитии игровой индустрии. Нейронные сети можно использовать для создания персонажей и ботов с более реалистичным поведением и развитым искусственным интеллектом. Это позволяет игрокам испытать не только полное погружение в игровой процесс, но и почти реалистичное представление поведения окружающих их объектов.

В целом, использование нейронных сетей становится все более популярным в игровой индустрии, поскольку они позволяют создавать более интерактивные и реалистичные игровые сцены и более сложные игровые механики.

1. **ТЕХНОЛОГИИ НЕЙРОННЫХ СЕТЕЙ**

Функция активации определяет выходное значение нейрона в зависимости от результата взвешенной суммы входов и порогового значения.

Рассмотрим нейрон:



Значение *Y* может быть любым в диапазоне от -бесконечности до +бесконечности. В действительности нейрон не знает границу, после которой следует активация. Ответим на вопрос, как решается, должен ли нейрон быть активирован. Для этой цели решили добавлять активационную функцию. Она проверяет произведенное нейроном значение *Y* на предмет того, должны ли внешние связи рассматривать этот нейрон как активированный, или его можно игнорировать.

* Ступенчатая функция активации

Начнем с того, как определить границы активации для активационной функции. Если значение переменной Y превышает определенный пороговый уровень, то мы можем считать нейрон активированным. В противоположном случае, когда значение Y ниже порога, мы можем утверждать, что нейрон неактивен. Необходимо теперь проанализировать эту схему и формализовать ее для дальнейшего применения.

-Функция А = активирована, если Y > границы, иначе нет.

-Другой способ: A = 1, если Y > границы, иначе А = 0.

Такая функция называется ступенчатой. Она представлена на рисунке 1.1.

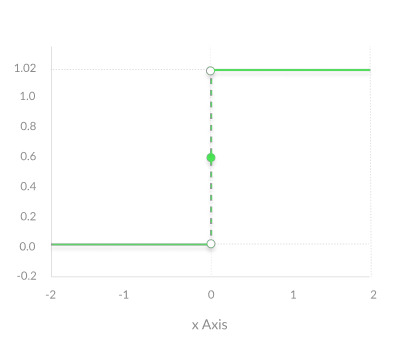


Рисунок 1.1. Ступенчатая функция активации

Функция принимает значение *1* (активирована), когда *Y > 0* (граница), и значение 0 (не активирована) в противном случае.

Это простой способ, однако в нём есть недостатки. Рассмотрим следующую ситуацию.

Допустим, мы разрабатываем бинарный классификатор, то есть модель, способную дать ответ "да" или "нет". Такой результат может быть легко получен благодаря использованию ступенчатой функции, которая вернет 1 или 0 в зависимости от обработанных входных данных.

Теперь представим случай, когда требуется большее количество нейронов для классификации многих классов: класс1, класс2, класс3 и так далее. Однако, что произойдет, если более одного нейрона будет активировано? В таком случае, все нейроны в функции активации перейдут в состояние "1". И возникает вопрос о том, какой класс окажется конечным для данного объекта.

Мы хотим, чтобы активировался только один нейрон, а функции активации других нейронов были равна нулю (только в этом случае можно быть уверенным, что сеть правильно определяет класс). Такую сеть труднее обучать и добиваться сходимости. Если активационная функция не бинарная, то возможны значения “активирован на 50%”, “активирован на 20%” и так далее. Если активированы несколько нейронов, можно найти нейрон с наибольшим значением активационной функции.

Но в таком случае, как и ранее, если более одного нейрона говорят “активирован на 100%”, проблема по-прежнему остается. Так как существуют промежуточные значения на выходе нейрона, процесс обучения проходит более гладко и быстро, а вероятность появления нескольких полностью активированных нейронов во время тренировки снижается по сравнению со ступенчатой функцией активации

* Линейная функция активации

Линейная функция представляет собой прямую линию и пропорциональна входу (то есть взвешенной сумме на этом нейроне).

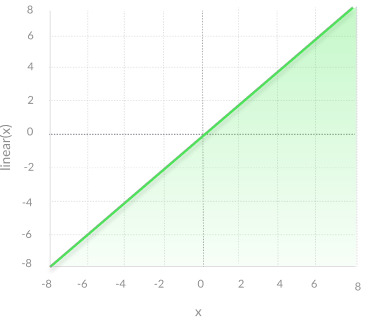


Рисунок 1.2. Линейная функция активации.

Такой выбор активационной функции позволяет получать спектр значений, а не только бинарный ответ. Можно соединить несколько нейронов вместе и, если более одного нейрона активировано, решение принимается на основе применения операции max (или softmax).

Но у линейной функции есть две основных проблемы:

1. Невозможность использования метода обратного распространения ошибки. Так как в основе этого метода обучения лежит градиентный спуск, а для того, чтобы его найти, нужно взять производную, которая для данной функции активации — константа и не зависит от входных значений. То есть при обновлении весов нельзя сказать улучшается ли эмпирический риск на текущем шаге или нет.
2. Рассмотрим нейронную сеть с несколькими слоями с данной функцией активации. Так как для каждого слоя выходное значение линейно, то они образуют линейную комбинацию, результатом которой является линейная функция. То есть финальная функция активации на последнем слое зависит только от входных значений на первом слое. А это значит, что любое количество слоев может быть заменено всего одним слоем, и, следовательно, нет смысла создавать многослойную сеть.

Главное отличие линейной функции от остальных в том, что ее область значений не ограничена: (−∞; +∞) (−∞; +∞). Следовательно, ее нужно использовать, когда выходное значение нейрона должно ∈R, а не ограниченному интервалу.

* Сигмоидная функция

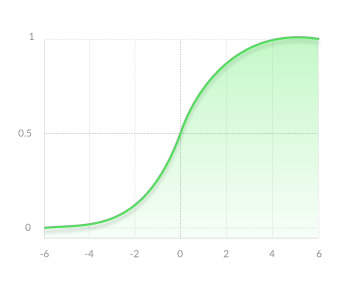


Рисунок 1.3. Сигмоидная функция активации

Сигмоидная функция, которую также называет логистической является гладкой монотонно возрастающей нелинейной функцией:

И так как эта функция не линейна, то ее можно использовать в нейронных сетях с множеством слоев, а также обучать эти сети методом обратного распространения ошибки. Сигмоида ограничена двумя горизонтальными асимптотами y = 1 и y = 0, что дает нормализацию выходного значения каждого нейрона. Кроме того, для сигмоидной функции характерен гладкий градиент, который предотвращает "прыжки" при подсчете выходного значения. Помимо всего этого, у этой функции есть еще одно преимущество, для значений x > 2 и x < −2, y "прижимается" к одной из асимптот, что позволяет делать четкие предсказания классов.

Несмотря на множество сильных сторон сигмоидной функции, у нее есть значительный недостаток. Производная такой функции крайне мала во всех точках, кроме сравнительно небольшого промежутка. Это сильно усложняет процесс улучшения весов с помощью градиентного спуска. Более того, эта проблема усугубляется в случае, если модель содержит много слоев. Данная проблема называется проблемой исчезающего градиента.

Что касается использования сигмоидной функции, то ее преимущество над другими — в нормализации выходного значения. Иногда, это бывает крайне необходимо. К примеру, когда итоговое значение слоя должно представлять вероятность случайной величины. Кроме того, эту функцию удобно применять при решении задачи классификации, благодаря свойству "прижимания" к асимптотам.

Таким образом, функции активации являются ключевым элементом в построении нейронных сетей, и правильный выбор функции активации может значительно повлиять на производительность, точность и скорость обучения сети. Как показано в примерах, разные функции активации обладают различными свойствами, и выбор определенной функции зависит от задачи, которую мы хотим решить. Однако, существуют и другие функции активации, которые мы не рассмотрели в этой главе, и которые также могут быть полезны в решении определенных задач. Кроме того, исследования в области функций активации продолжаются, и в будущем мы можем увидеть новые, более эффективные функции, которые помогут нам решить сложные задачи в машинном обучении

# **2. ГЕНЕТИЧЕСКИЙ АЛГОРИТМ ДЛЯ ОБУЧЕНИЯ НЕЙРОННОЙ СЕТИ**

## 2.1. Основные понятия

Генетический алгоритм – это эволюционный алгоритм поиска решений в задачах оптимизации и адаптации. ГА относится к большому классу эволюционных алгоритмов (ЭА), в котором выделяются:

• Генетический алгоритм

• Генетическое программирование

• Эволюционное программирование

• Дифференциальная эволюция

Основными принципами алгоритма являются:

• Принцип естественного отбора, предложенный Ч. Дарвином в своей теории эволюции.

• Принцип скрещивания, открытый Г.Менделем в 1865г.

• Принцип мутации, открытый в 1900г. де Вре.

Описанные принципы являются основными в генетическом подходе, благодаря им происходит эволюция особей и поиск лучших решений. 19 Генетический алгоритм отличается от других методов оптимизации следующими особенностями:

• Поиск решения осуществляется из нескольких позиций (особи в популяции)

• Использование целевой функции в чистом виде, без использования её производной или другой дополнительной информации

• Применение вероятностных методов

• Устойчивость к попаданию в локальные экстремумы

Популяция представляет собой набор особей или потенциальных решений искомой задачи. Каждая особь состоит из набора хромосом. В свою очередь, хромосома представляет собой совокупность генов. В генах закодирована генетическая информация. Классический генетический алгоритм состоит из следующей последовательности этапов:

1. Генерация начальной популяции

2. Оценка приспособленности каждой особи в популяции

3. Отбор родителей для процесса скрещивания

4. Скрещивание пар родителей

5. Мутация новых особей

6. Увеличение размера популяции за счет порожденных потомков

7. Выбор особей и сокращение популяции для следующего поколения

8. Проверка условия завершения алгоритма.

9. Выбор наилучшей особи

10. Результат

## 2.2. Операторы генетического алгоритма

Основными строительными блоками генетического алгоритма являются операторы. Оператор представляет собой функционально законченный блок, реализующий один шаг алгоритма. К основным операторам ГА относятся: создание начальной популяции, скрещивание, мутация, селекция, редукция, оценка приспособленности. Каждый такой оператор частично или полностью покрывает соответствующий механизм генетики. От выбора типа каждого оператора, напрямую зависит эффективность алгоритма. Схема алгоритма представлена на рисунке 2.1.

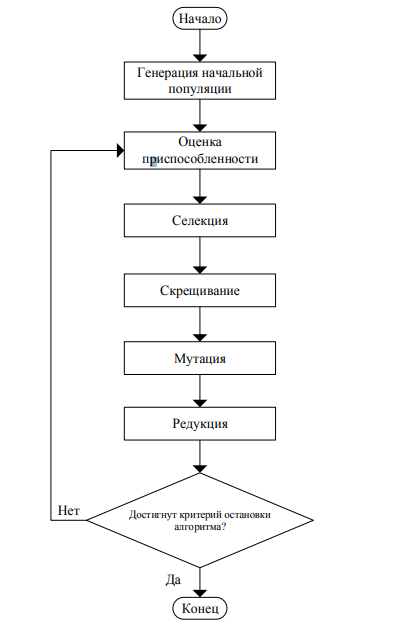


Рисунок 2.1. Пример генетического алгоритма.

В зависимости от класса решаемой задачи, структура и набор операторов довольно сильно отличается. Существует большое количество видов операторов и их вариаций.

## 2.3. Обучение нейронной сети с помощью генетического алгоритма

Генетический алгоритм как метод оптимизации может применяться для настройки и обучения ИНС. Применение ГА обосновывается его устойчивостью к попаданию в локальные оптимумы.

Эволюция весов сети. Для оптимизации весов ИНС, с заранее заданной архитектурой, также логично использовать ГА. На первом этапе происходит генерация начальной популяции весовых коэффициентов (ВК). Каждая особь популяции представляет собой набор весовых коэффициентов ИНС. Затем происходит оценка значения приспособленности каждой особи. Вычисление фитнес-функции основан на результате работы ИНС и состоит из следующих этапов:

• Инициализация ВК ИНС

• Запуск ИНС

• Оценка правильных ответов сети

На основании значения пригодности каждой особи, происходит выполнение операторов селекции, скрещивания и мутации. В результате операций появляется новая популяция ВК, более пригодная для дальнейшего развития. Таким образом происходит настройка весов сети. Схема эволюция весов ИНС представлена на рисунке 2. Корректировка ВК происходит после предъявления сети всех обучающих выборок. Такой подход обучения называется пакетным.

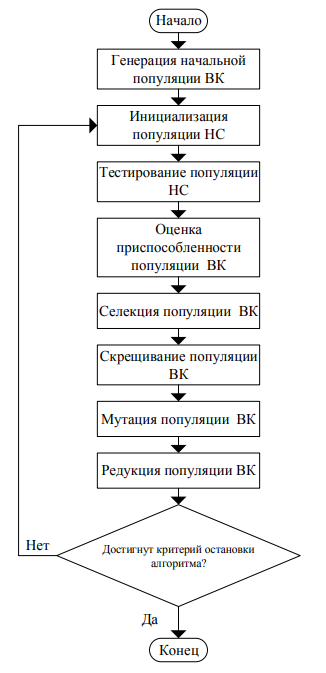


Рисунок 2.3. Настройка весов ИНС

Эволюционный подход поиска архитектуры и настройки ИНС является мощным инструментом, но требует достаточно много времени, а в замен предлагает более качественное обучение.

## 2.4. Преимущества и недостатки генетического алгоритма

Естественно, эффективность генетического алгоритма различается от класса и параметров решаемых задач. В некоторых областях генетический алгоритм лучше других справляется с поставленной задачей, а в некоторых может быть крайне неэффективен. Чаще всего, в чистом виде генетический алгоритм не используется, а используется совместно с другими поисковыми или оптимизационными алгоритмами. Это позволяет подчеркнуть сильные стороны ГА и свести к минимуму его недостатки.

Ниже приведены преимущества и недостатки ГА.

Преимущества:

• Простота реализации – отличительной особенностью ГА от других является его простота в реализации. Количество применяемых операторов невелико, сами операторы не содержат сложных и трудоемких операций, а сама идея эволюции легко усваивается.

• Универсальность алгоритма заключается в том, что он может быть применен к широкому кругу задач, если задачу можно свести к задаче оптимизации. Универсальность также заключается в том, что ГА может оперировать с параметрами задачи не только напрямую, но и с их закодированными значениями.

• Параллелизм. ГА так устроен, что в нем большинство операций происходят параллельно, не требуя результата других операций. Развитие особей в популяции ведется параллельно. В операции скрещивания участвует 2 -3 особи, в операции мутации – чаще всего 1 особь, это позволяет распараллелить данные операции. Основное время тратится на вычисление функции пригодности, которая для каждой особи может выполняться параллельно. Также ГА хорошо адаптирован к распределённым вычислительным системам.

• Высокая устойчивость к изменениям условий задачи, обусловлена тем, что суть ГА заключается в постоянной эволюции особей, что способствует быстрой адаптации особей к новым условиям.

• Совместное использование с другими алгоритмами. Очень часто генетический алгоритм используется как вспомогательный, например, для грубого поиска начальных решений, а основной алгоритм выбирается из числа более подходящих в решении поставленной задачи. Например, для обучения искусственных нейронных сетей часто используют пару ГА – метод обратного распространения ошибки.

• Возможность использования данных без преобразования кодов

• Осуществление поиска из множества точек (особей)

• Возможность выбивания из локального оптимума

Недостатки:

• Проблема выбора операторов ГА - выбор операторов не является очевидной задачей и может сильно изменяться в зависимости от класса решаемой задачи.

• Неоднозначность выбора параметров ГА – объясняется невозможностью подобрать такие параметры как: численность популяции, вероятность скрещивания, вероятность мутации, коэффициент масштабируемости фитнес-функции и другие, которые были бы универсальными для решения широкого круга задач. Для каждой конкретной задачи параметры ГА определяются экспериментально или на основе предыдущего опыта.

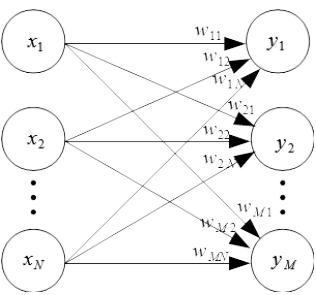
• Низкая эффективность в поиске локальных оптимумов – так как операторы алгоритма нацелены на выбивание особей из локальных точек экстремума, то задача поиска локальных решений становится затруднительной.

• Неэффективность использования в задачах с целевыми функциями, требующими длительного времени на вычисление. Пожалуй, один из главных недостатков ГА. Генетический алгоритм основан на принципе эволюции, что влечет за собой большое число итеративных операций вычисления целевой функции. Даже если популяция содержит 30 особей и длительность эволюции равна 10 эпохам, а целевая функция вычисляется за ~1с, то для работы алгоритма потребуется ~5мин.

• Неэффективность использования в задачах с большим размером хромосом. Большой размер хромосом также влечет за собой увеличение времени выполнения и снижает эффективность операторов скрещивания и мутации. В задачах обучения нейронных сетей, этот пункт очень важен. Так как с увеличением слоёв и количества нейронов в каждом слое, растет общее число элементов поиска ГА.

• Высокое взаимовлияние весовых коэффициентов на результат при обучении ИНС, требует увеличения количества эпох и размера популяции, что влечет за собой увеличение времени обучения.

1. **ТОПОЛОГИИ НЕЙРОННЫХ СЕТЕЙ.**
   1. **Многослойный перцептрон.**

 Сначала рассмотрим однослойный перцептрон, представленный на схеме рис. 3.1. Каждый входной элемент

подключен к каждому нейрону в однослойном перцептроне через связь с весами— количество нейронов слоя, j = — количество входных сигналов нейронной сети. Выход нейронного слоя вычисляется по следующим формулам:

(1.1) Рис. 3.1 Схема однослойного перцептрона

Либо в матричной форме:

(1.2)

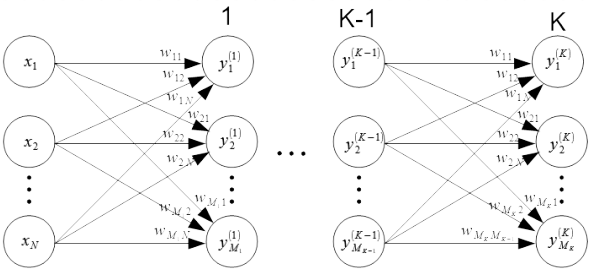
где — функция активации.

Рис. 3.2. Схема многослойного перцептрона.

На рисунке 3.2 показана схема полностью подключенной многослойной искусственной нейронной сети (многослойного перцептрона). В ней каждый нейрон первого слоя соединен с каждым элементом входного вектора, а каждый последующий слой соединен со всеми нейронами предыдущего слоя. Пусть — вход нейросети размера ; — выход -го слоя, состоящего из нейронов, , где — количество слоев; — матрица весов-го слоя. Выход последнего слоя, который также является выходом перцептрона, рассчитывается по итерационной формуле:

… (1.3)

На данный момент многослойные перцептроны используются в широком спектре приложений, включая компьютерное зрение, распознавание речи, обработку естественного языка, рекомендательные системы, прогнозирование временных рядов, анализ финансовых данных, кибербезопасность и игры.

* 1. **Рекуррентные нейронные сети.**

Рекуррентные нейронные сети (RNN) — это тип искусственных нейронных сетей, которые могут анализировать и предсказывать непрерывные данные, такие как временные ряды, текст и речь. RNN являются наиболее мощным и эффективным инструментом для анализа временных рядов, машинного перевода, распознавания речи, обработки текстов, компьютерного зрения, генерации музыки и т.д. Они имеют широкий спектр применения в различных областях.

RNN обладают значительными преимуществами перед другими нейросетевыми архитектурами, такими как многослойные перцептроны (MLP) и сверточные нейронные сети (CNN) при анализе последовательных данных. RNN обладают способностью запоминать предыдущие состояния и использовать эту информацию для определения текущего состояния. Это позволяет RNN анализировать последовательные данные, принимая во внимание контекст и зависимости между различными элементами последовательности.

РНС работают по принципу обратной связи. Это означает, что выход каждого узла сети передается на вход следующего узла. Это позволяет принимать решения с учетом предыдущего состояния системы; общий вид RNN показан на рисунке 1.3:



Рисунок 3.3 общий вид RNN

На схеме показана односторонняя RNN, где каждый узел имеет входные данные и выходные данные. Каждый узел также получает входных данных от предыдущего узла. Эта обратная связь позволяет узлам RNN запоминать предыдущее состояние системы и использовать эту информацию для анализа текущих данных.

Для обучения RNN используется алгоритм обратного распространения для адаптации весов в сети в соответствии с результатами; процесс обучения RNN требует выбора соответствующей функции потерь для оценки качества прогнозирования сети.

Сегодня RNN и их производные находят широкое применение в таких различных областях, как речевые помощники, машинный перевод, анализ текста и речи, генерация музыки, генерация описания изображения. RNN являются одним из наиболее эффективных инструментов для анализа и прогнозирования временных рядов и имеют большой потенциал для применения во многих других областях

* 1. **Сверточные нейронные сети**

Свёртка является операцией, которая применяется к двум последовательностям f и g и порождает третью последовательность.

, где

Формула для двумерной свёртки:

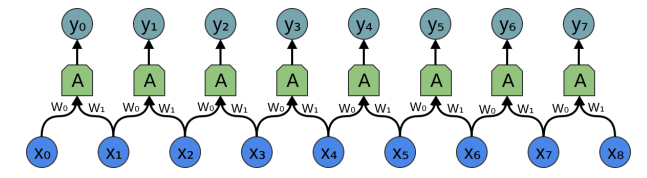
 Рассмотрим одномерный свёрточный слой с входами и выходами (см. рис. 3.4). Тогда функция для выходов будет представлена следующим образом:

Рисунок 3.4 — Пример одномерного свёрточного слоя.

В свёрточном слое находится множества копий одного и того же нейрона, поэтому многие веса появляются в нескольких позициях.

Стандартная матрица весов соединяет каждый вход с каждым нейроном с разным весом. Матрицы свёрточного слоя отличаются тем, что разные веса могут появляться в нескольких местах, и, поскольку нейроны не подключены ко всем возможным входам, матрица содержит много нулевых элементов:

То есть умножение на матрицу выше — то же самое, что и свёртка с . Ядро свёртки, скользящее по разным частям изображения, соответствует наличию нейронов в этих частях.

Свертку можно проиллюстрировать на примере обработки изображений. Представьте, что изображение — это двумерная функция и что различные преобразования изображения — это не что иное, как свертки между функцией изображения и локальной функцией, называемой ядром свертки.

Каждый новый пиксель изображения представляет собой взвешенную сумму пикселей, через которые прошло ядро до этого момента времени; двумерный слой свертки показан на рисунке 3.5.

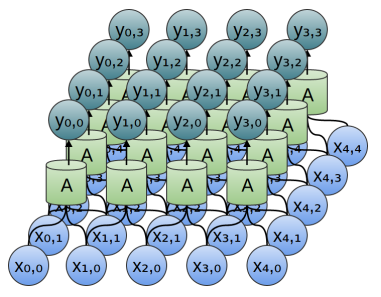


Рисунок 3.5 — Двумерный свёрточный слой.

Свёрточная нейронная сеть — архитектура нейронных сетей, изначально созданная и использованная для эффективного распознавания изображений: чередуются свёрточные слои (convolutions) с нелинейными активационными функциями (ReLU или гиперболический тангенс tanh) и слои объединения (pooling layers).

В отличие от сетей прямого распространения, где каждый входной нейрон соединен с выходным нейроном в следующем слое, сверточные сети производят выходные значения путем свертки на каждом входном слое. Операция свертки использует небольшую матрицу весов, которые перемещаются между слоями обработки.

Эта матрица смещается по всем слоям обработки для формирования сигнала активации при каждом смещении нейрона в ту же позицию в следующем слое.

Затем формируется первый слой. Эта матрица называется ядром свертки и используется для нейронов в различных выходных слоях.

Она используется для различных нейронов в выходном слое.

В операции свертки каждый сегмент (например, изображение) поэлементно умножается на матрицу свертки, результат суммируется и записывается в ту же позицию в выходном изображении. Матрица свертки является графическим представлением признаков. Следующий слой, полученный в результате операции свертки, представляет наличие данного признака. В сверточной нейронной сети существует несколько наборов весов, которые кодируют элементы изображения. Ядра свертки создаются в процессе обучения сети. При прохождении каждого набора весов формируется карта признаков; если в слое появляется много отдельных карт признаков, сеть становится многоканальной.

Многоканальная.

Каждый конволюционный слой имеет свой фильтр для каждого канала, и конволюционное ядро выполняет побитовую обработку предыдущего слоя. Результаты различных фильтров объединяются вместе. Таким образом, создаются объединенные слои. Операция субдискретизации уменьшает размерность результирующей карты признаков. В структуре этой сети информация о наличии искомого признака считается более важной, чем точное знание его координат, поэтому максимальное значение выбирается из нескольких соседних нейронов в карте признаков и используется в качестве нейрона в сжатой карте признаков меньшей размерности. Затем максимальное значение используется в качестве нейрона в сжатой карте признаков меньшей размерности. Помимо дальнейшего увеличения скорости вычислений, эта операция оставляет неизменным масштаб сети по отношению к входному изображению.

После начального слоя сигнал проходит через ряд слоев свертки, в которых чередуются операции свертки и объединения. С помощью чередования слоев можно построить карту признаков. На практике это означает способность распознавать сложные иерархические признаки.

После прохождения через несколько слоев карта признаков вырождается в вектор или скаляр, но таких карт признаков сотни. На выходе сверточного слоя сети устанавливается еще несколько слоев полностью связанных нейронных сетей (например, перцептронов), в которые подаются конечные карты признаков.

1. **ПРИМЕНЕНИЕ НЕЙРОННЫХ СЕТЕЙ В КОМПЬЮТЕРНЫХ ИГРАХ**
   1. **Возможности нейронных сетей в игровой индустрии.**

Нейронные сети уже давно перешагнули границы лабораторий и проникли в повседневную жизнь. Их применение широко распространено в игровой индустрии и во многих других областях человеческой деятельности. От настольных игр, таких как шахматы и го, до видеоигр. Применение нейронных сетей в компьютерных играх открывает возможности для создания новых уникальных продуктов.

Примеры практического применения нейронных сетей в играх включают улучшение графики и фотореалистичных изображений, создание новых виртуальных миров, повышение реалистичности игровых персонажей, улучшение систем искусственного интеллекта, оптимизацию скорости и качества обработки текстур и звуковых эффектов и многое другое.

Нейронные сети также особенно полезны для создания новых игровых сценариев и инновационных видеоигр в целом. Они особенно полезны для создания персонализированных игровых сценариев, учитывающих поведение и привычки каждого игрока.

Кроме того, использование нейронных сетей для анализа поведения игроков может улучшить удобство игрового интерфейса. Это позволяет создавать более удобные и простые игровые интерфейсы, которые упрощают игровой процесс и облегчают управление персонажами.

* 1. **Улучшение графики с помощью ИИ.**

Задача улучшения графики в играх - одно из главных применений нейронных сетей. Быстро развивающиеся технологии позволяют создавать все более реалистичные и потрясающие визуальные эффекты.

Сейчас большинство современных игр достаточно «тяжелые». И их становится всё сложнее оптимизировать. ИИ можно использовать для более качественной обработки окружения, "апскейла" картинки, что уже сейчас делают представители NVIDIA, или же оптимизации эффектов с учётом вашей конфигурации, где необходимое количество кадров в секунду (FPS) будет служить выходным значением для нейронной сети.

Deep Learning Super Sampling (DLSS), или суперсэмплинг на основе глубокого обучения, — технология NVIDIA, которая улучшает быстродействие и качество изображения в играх благодаря искусственному интеллекту. Результат использования технологии можно увидеть на рис. 4.1

Рисунок 4.1 – Использование технологии NVIDIA – DLSS 3.0

Упрощенно работу алгоритма DLSS можно описать так.

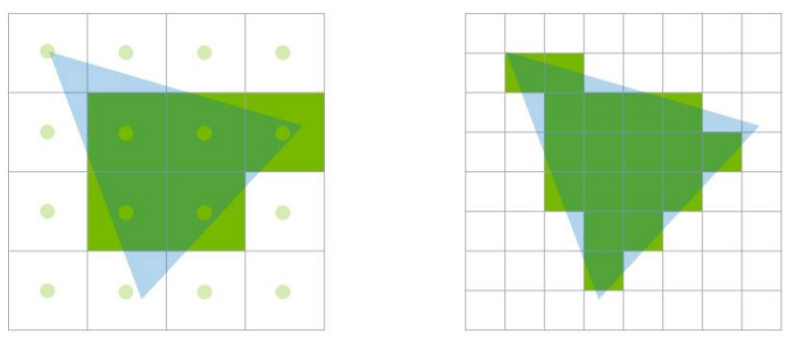
При рендеринге простых геометрических фигур (в примере используется треугольник рис 4.2) из исходных кадров малого размера определяющим фактором качества конечного результата является субпиксельная маска. К примеру, с использованием маски 4х4 при отрисовке треугольника конечный результат мало напоминает исходную фигуру. При увеличении же сетки ресемплирования всего в 4 раза — до размера 8х8 — конечное изображение уже больше напоминает исходник.

Рисунок 4.2 – Пример “апскейлинга”

Такой подход к рендерингу игровых сцен дает конечному пользователю несколько очень важных преимуществ:

* Снижение нагрузки на графический процессор, ведь обрабатывать приходится меньшие по размеру и объему изображения. При этом существенно снижается нагрев видеокарты, а это залог ее длительной работы без сбоев;
* Рост FPS, поскольку аппаратной части значительно проще и легче обрабатывать несколько небольших по размеру кадров в единицу времени. Экспериментально доказано, включение режима DLSS приводит к увеличению частоты кадров в 1,5-2 раза.
  1. **Создание игровых миров и уровней с помощью нейросетей.**

Нейросеть может генерировать новые уровни в играх с помощью алгоритмов глубокого обучения. Эти алгоритмы основываются на обработке большого объема данных и выявлении в них закономерностей, которые затем используются для создания новых уровней.

Процесс работы нейросети начинается с обучения ее на уже существующих уровнях игры. Нейросеть анализирует эти уровни, выделяет их особенности, определяет, какие элементы и комбинации элементов делают уровень интересным и сложным, а затем использует эту информацию для создания новых уровней.

Таким образом, при создании нового уровня нейросеть использует основные элементы, такие как препятствия, враги, бонусы, и создает уникальную комбинацию этих элементов, которая будет интересной для игрока.

Применение нейросетей для генерации новых уровней и миров уже используется в ряде игр, таких как Super Mario Bros, Minecraft, No Man's Sky, Microsoft Flight Simulator и т.д. Это позволяет разработчикам игр создавать более сложные и увлекательные уровни, а также экономить время на создании каждого уровня вручную.

Например, Microsoft Flight Simulator воссоздан весь земной шар, но вручную разработчики создавали модели только наиболее известных объектов. Весь остальной мир был заполнен домами и деревьями при помощи алгоритмов автоматической генерации.

Для генерации объектов на поверхности Земли разработали комбинацию нейронных сетей, в качестве исходных данных получающих спутниковые и панорамные снимки из сервиса Bing.

Анализируя спутниковые снимки, нейронные сети разбивают их на фрагменты, а затем вычленяют и классифицируют здания, дороги и растительность. Для каждого из зданий искусственный интеллект определяет набор характеризующих его атрибутов.

Это может быть, например, этажность и стиль постройки, количество окон, балконов и других мелких деталей, которые можно распознать по снимкам. Затем, ориентируясь на эти атрибуты, нейронная сеть создает виртуальную копию здания.

Его внешние особенности — например, цвет фасада или дизайн окон — задаются с учетом географического расположения постройки. В общей сложности нейросети распознали и создали виртуальные копии примерно для 1.5 миллиарда зданий.

После этого здания, дороги и растения размещаются в виртуальном мире примерно там же, где они находятся в реальности. Нейронные сети могут работать с ограниченным набором данных: если нет полной информации, система может додумать вид здания.

На обработку спутниковых и панорамных снимков со всей планеты уходит около 72 часов. Нейросети запускаются на серверах Azure, облачного сервиса Microsoft, во время обработки им приходится иметь дело с несколькими петабайтами данных.

На выходе нейронные сети генерируют порядка 1.5 миллиарда цифровых отпечатков, использующихся для генерации виртуального мира.

1. **МОЙ ПРОЕКТ**

**5.1. Общее представление.**

Моя игра является демонстрацией возможностей обучения нейронной сети через генетический алгоритм на примере гоночного трека. В игре боты представляют из себя машины, которые учатся проходить трассу за наименьшее время. Полученная обученная модель может быть использована в дальнейшем против игрока или других целей для улучшения "геймплейной" составляющей.

Одной из ключевых особенностей игры является то, что процесс обучения нейронной сети происходит в режиме реального времени. Это позволяет быстро узнавать результаты работы алгоритма и вносить корректировки в процесс обучения.

Для реализации игры были использованы язык программирования C#, среда разработки JetBrains Rider и движок Unity.

Unity — это кроссплатформенный игровой движок, который широко используется для создания различных игровых проектов. Unity обеспечивает множество инструментов и ресурсов для создания игровых объектов, анимаций, эффектов и все это с высокой степенью оптимизации и производительности. В Unity есть возможность создавать игры не только для ПК, но и для мобильных платформ.

Языком программирования был выбран C# так как это объектно-ориентированный язык программирования, который широко используется в различных сферах программирования, включая разработку игр и нейросетей, а также это основной ЯП для Unity. Он предоставляет множество возможностей для написания высококачественного кода и обладает высокой степенью производительности.

Rider — это среда разработки на C# от компании JetBrains. Она обеспечивает простой и удобный интерфейс для написания кода и содержит множество инструментов для отладки, тестирования и взаимодействия с Unity.

Использование этих инструментов позволяет создавать высококачественную и продуктивную игру с возможностью обучения нейронной сети через генетический алгоритм. Одной из главных задач разработки игры была оптимизация процесса обучения и создание нейронной сети, которая могла бы проходить трассу за наименьшее время. Обучение нейросети происходило в реальном времени, что позволяло быстро узнавать результаты работы алгоритма и вносить корректировки в процесс обучения.

В ходе работы было проведено сравнение различных архитектур нейронных сетей и определены оптимальные значения для каждого параметра, чтобы повысить эффективность обучения. В результате была получена модель нейронной сети, которая успешно проходит гоночный трек за наименьшее время.

Эти инструменты подходят для создания игрового контента и имеют удобные инструменты для работы с графикой и физикой. Также они позволяют использовать разные библиотеки для реализации машинного обучения.

* 1. **Общий вид программы и интерфейс**

Интерфейс игры (рис. 5.1) содержит две панели, каждая из которых предназначена для определенной функции. Первая панель отображает информацию о лучшем результате прошлого и номер текущего поколения. Эта панель позволяет игроку отслеживать прогресс в обучении нейронной сети.



Рисунок 5.1. Интерфейс разработанного приложения.

Вторая панель обеспечивает пользователю управление обучением нейросети. Она включает в себя поле для ввода числа популяции - количество игроков, которых нужно обучать. Это число может быть установлено в соответствии с вашими предпочтениями и возможностями компьютера.

Первый слайдер на второй панели позволяет настраивать скорость игры. Этот параметр будет влиять на скорость обучения ботов и, соответственно, ускорит при достаточной вычислительной мощности.

Второй слайдер для настройки шанса мутации также доступен на второй панели интерфейса. Он позволяет настраивать вероятность мутаций ботов во время обучения. Этот параметр позволяет управлять эффективностью обучения.

Третий слайдер для настройки силы мутации - еще одно важное орудие для управления обучением нейронной сети. Он позволяет пользователю установить, насколько сильно мутации будут влиять на нейронные сети ботов. Этот параметр является ключевым фактором в определении эффективности обучения нейронной сети, и хорошо настроенный слайдер сможет максимизировать результаты обучения.

Общая конфигурация и настройка параметров нейронной сети доступны на второй панели, что делает ее легкой и удобной в использовании для игроков. Данная настройка предоставляет пользователям максимальную возможность управления в процессе обучения, что способствует оптимизации результатов.

## 5.3. Проделанная работа по обучению нейронной сети

Процесс обучения нейронной сети происходит на основе генетического алгоритма. Данный метод позволяет находить наилучшие параметры для достижения заданной цели.

Также были проведены различные эксперименты для определения влияния различных параметров на качество работы нейронной сети.

Одним из экспериментов было использование различных трасс для тренировки нейронной сети. Некоторые трассы оказались не проходимыми, так как они упирались в одну точку и не позволяли сети развиваться дальше. Это может быть связано с ошибками в моделировании гоночного трека.

Кроме того, проводились эксперименты над конфигурацией нейронной сети. Было выяснено, что лучшей силой мутации являются значения от 0.3 до 0.5, а лучшим шансом мутации является значение в 1% (0.01). Эти настройки помогли улучшить результаты обучения.

Другим важным экспериментом было добавление функции обмена генетическими данными - Crossover(). После ее добавления скорость обучения возросла на 15-20%. Это свидетельствует о том, что подобные оптимизации в значительной мере могут повлиять на эффективность обучения нейронной сети.

В итоге была получена модель нейронной сети, которая успешно проходит гоночный трек за наименьшее время. Полученная модель может быть использована в качестве искусственного интеллекта для противостояния реальным игрокам.

Таким образом, разработка игры с использованием нейронных сетей и генетических алгоритмов позволяет демонстрировать возможности машинного обучения и создавать динамичный игровой процесс.

* 1. **Описание классов и функций.**

Класс Manager – контролирует и управляет обучением нейронной сети и интерфейсом.

Start() – стандартная функция вызываемая Unity в начале игры до первого обработанного кадра. Внутри проверяется размер популяции, чтобы генетический алгоритм работал корректно, а также вызывается повторяющийся каждые “timeframe” секунд метод “CreateBots()” и инициализирует нейронные сети методом “InitNetworks()”.

InitNetworks() - функция инициализации нейронных сетей. В данной функции создаются экземпляры нейронных сетей и загружается обучение, которое было сохранено в предыдущих запусках алгоритма.

CreateBots() - функция создания новых интеллектуальных машин. Происходит итерация по списку существующих машин, новое поколение которых фигурирует в текущем этапе алгоритма. Выполняется обновление фитнес-счетчика, затем расчет новых значений интеллектуальных параметров. Сортируются нейронные сети по фитнес-счетчику, сохраняется лучшая особь, затем создаются новые нейронные сети, скрещивая лучшие особи и мутируя эти потомков.

TextChange() - функция, которая срабатывает при изменении ползунков(слайдеров) на экране. Она изменяет текстовое значение, чтобы показывать пользователю текущие, выставленные значения.

ChangeParameters() - функция, срабатывающая при нажатии на кнопку «Accept Changes», применяет введенные данные к конфигурации нейронной сети.

Класс NeuralNetwork – создает и управляет нейронной сетью для решения задач машинного обучения.

NeuralNetwork() - конструктор класса, принимающий массив целых чисел, который содержит количество нейронов в каждом слое ИНС, и инициализирующий поля класса.

InitNeurons() - функция, отвечающая за инициализацию нейронов каждого слоя ИНС. Нейроны по слоям соответствуют переданному массиву целых чисел и хранятся в массиве neurons.

InitBiases() - функция, создающая массив смещений (biases) для каждого слоя ИНС. Значения смещений выбираются случайным образом из диапазона от -0.5 до 0.5.

InitWeights() - функция, создающая массив весов (weights) для каждой связи между нейронами двух слоев. Значения весов выбираются случайным образом из диапазона от -0.5 до 0.5. Каждый элемент массива weights содержит двумерный массив, представляющий связи между нейронами текущего и предыдущего слоя ИНС. Каждый такой массив содержит веса для всех нейронов текущего слоя.

FeedForward() - функция, принимающая входное значение inputs и проходит через все слои нейронной сети, вычисляя выходные значения нейронов и возвращая их.

Activate() - функция, которая используется для применения активационной функции (в данном случае гиперболический тангенс) к вычисленному значению нейрона.

Mutate() - функция, которая изменяет веса (weights) и смещения (biases) нейронов в нейронной сети на случайные значения. Входные параметры: chance - вероятность произведения мутации, val - максимальное изменение весов и смещений. Функция проходит по всем элементам массивов biases и weights и изменяет их, если случайно выбранное число из диапазона от 0 до chance меньше или равно 5. В таком случае, текущий элемент массива увеличивается или уменьшается на случайную величину из диапазона от -val до val.

Crossover() - функция, реализующая кроссовер (обмен генетическим материалом) двух нейронных сетей. Входной параметр: other - другая нейронная сеть. Функция проходит по всем массивам biases и weights текущей нейронной сети и с определенной вероятностью (0.5) заменяет их значения на соответствующие значения из массивов biases и weights другой нейронной сети. В результате функции, текущая нейронная сеть может унаследовать некоторые характеристики от другой нейронной сети.

Метод CompareTo используется для сравнения эффективности двух нейронных сетей и помогает выбрать лучшую среди них.

Copy() - функция, используемая для создания глубокой копии нейронной сети, чтобы можно было сохранить ее состояние и использовать ее в будущем.

Load() - функция, загружающая сохраненное состояние нейронной сети из файла и восстанавливает веса и смещения.

Save() – функция, записывающая в файл по указанному пути все значения весов и смещений нейронной сети. Он начинает с создания файла, закрывает его сразу же, затем открывает его для записи с использованием объекта StreamWriter, который будет использоваться для записи значений.

Класс Bot – использует нейронную сеть, чтобы управлять машиной на гоночном треке.

FixedUpdate() – функция, где сначала проверяется, не произошло ли столкновение автомобиля с объектом, используя переменную isCrashed. Если столкновения не произошло, мы проходимся в цикле по 5 направлениям, на которые разбивается окружность вокруг автомобиля. Для каждого направления мы создаем новый вектор, поворачивая вектор transform.right на угол i \* 45 - 90 градусов вокруг оси y и сохраняем его в переменную newVector. Затем мы создаем новый луч, направленный из позиции автомобиля в направлении newVector и проверяем, есть ли столкновение с объектами на сцене. Если столкновение было, мы вычисляем input[i], который дает информацию об удаленности препятствия: значение близко к 1, если препятствие находится близко к автомобилю, и близко к 0, если препятствие находится далеко. Если столкновения не было, input[i] = 0.

Далее мы используем нейронную сеть, созданную ранее, для вычисления output[0] и output[1]. output[0] отвечает за вращение автомобиля вокруг своей оси, а output[1] отвечает за движение автомобиля вперед. Значения output[0] и output[1] зависят от значений input[0]...input[4], которые мы вычислили ранее. И наконец, мы применяем изменения к позиции и повороту автомобиля.

В методе OnCollisionEnter() мы проверяем, столкнулся ли автомобиль с объектом на сцене. Если столкновение было с объектом на слое "CheckPoint", мы проверяем, совпадает ли этот объект с тем, которым должен проехать автомобиль следующим, чтобы считаться проехавшим чекпоинт. Если совпадает, мы увеличиваем переменную checkpointIndex на 1. Если столкновение было с другим объектом, мы устанавливаем переменную isCrashed в true, чтобы остановить движение автомобиля и поставить его на паузу.

В функции UpdateFitness() мы обновляем значение фитнеса для нейронной сети, сохраняя значение checkpointIndex в переменную fitness. Фитнесное значение показывает, насколько хорошо сеть прошла текущую итерацию обучения. В данном случае, чем больше чекпоинтов проехал автомобиль, тем выше его фитнес.

**Заключение**

В ходе выполнения курсовой работы была подробно изучены нейронные сети и их возможности. Были исследованы принципы работы, а также были изучены основные типы нейронных сетей, их различия и преимущества в решении определенных задач.

Одной из главных задач курсовой работы было исследование различных видов нейронных сетей, чтобы найти подходящую для создания игры со встроенным искусственным интеллектом. В результате выбор был сделан в пользу генетического алгоритма, который обладает высокой скоростью обучения, способностью работать с большими объемами данных и отличной точностью в предсказании результатов.

В результате проделанной работы был создан эффективный искусственный интеллект. Было проведено тестирование игровой среды, в которой в качестве оппонента выступала нейронная сеть, обученная с использованием генетического алгоритма. Также была создана игровая среда для обучения нейронных сетей, позволяющая контролировать процесс обучения и анализировать результаты. Разработанный алгоритм способен обучать нейронную сеть на основе выполнения заданных команд и целей, что позволяет создавать более эффективный искусственный интеллект для игр.

Созданная игра продемонстрировала возможности нейронных сетей и способность к решению сложных задач. Она также позволила разобраться в особенностях использования генетических алгоритмов в научно-исследовательских проектах и их внедрении в реальные приложения. В данной работе была проделана тщательная работа, которая позволила увидеть потенциал использования искусственного интеллекта в различных областях и установить новые границы для развития технологий в будущем.

# **Список использованных источников**

1. The Neural Network Zoo. URL: https://www.asimovinstitute.org/neural-network-zoo/. (дата обращения 15.03.2023)

2. Вороновский Г.К., Махотило К.В., Петрашев С.Н., Сергеев С.А. Генетические алгоритмы, искусственные нейронные сети и проблемы виртуальной реальности. // Харьков: Основа, 1997. – 107 с.

3. Гладков Л.А., Курейчик В.В., Курейчик В.М. Генетические алгоритмы// Под ред. В.М. Курейчика. — 2-е изд., испр. и доп. — М.: ФИЗМАТЛИТ, 2006. — 320 с.

4. Дарвин Ч. Происхождение видов. //пер. с англ. –М.: ИД «Эксмо», 2016. – 488с.

5. Дубенко К. И. Возможное применение нейронных сетей в играх в будущем // Наука, образование и культура. 2018. №10 (34). URL: https://cyberleninka.ru/article/n/vozmozhnoe-primenenie-neyronnyh-setey-v-igrah-v-buduschem. (дата обращения: 24.04.2023).

6. Как работает нейронная сеть: алгоритмы, обучение, виды нейросетей URL: https://neurohive.io/ru/osnovy-data-science/osnovy-nejronnyh-setej-algoritmy-obuchenie-funkcii-aktivacii-i-poteri/. (дата обращения: 24.04.2023).

7. Рутковская Д., Пилиньский М., Рутковский Л. Нейронные сети, генетические алгоритмы и нечёткие системы. // Д. Рутковская, пер. с польского И. Д. Рудинского. -М.: Горячая линия–Телеком, 2007. – 452с.

8. Остроух Е. Н., Чегодарь М. Ю. Обучение нейронной сети с использованием генетических алгоритмов // Известия ЮФУ. Технические науки. 2008. №11. URL: https://cyberleninka.ru/article/n/obuchenie-neyronnoy-seti-s-ispolzovaniem-geneticheskih-algoritmov. (дата обращения: 17.05.2023).

# **Приложение А**

**Листинги программы**

## Листинг 1. Файл NeuralNetwork.cs

|  |
| --- |
| **using** System**.**Collections**.**Generic**;**  **using** System**;**  **using** System**.**Globalization**;**  **using** System**.**IO**;**  **using** UnityEngine**;**  **using** Random **=** UnityEngine**.**Random**;**  **public** class NeuralNetwork **:** IComparable**<**NeuralNetwork**>**  **{**  **private** int**[]** layers**;**  **private** float**[][]** neurons**;**  **private** float**[][]** biases**;**  **private** float**[][][]** weights**;**  **private** int**[]** activations**;**    **public** float fitness **=** 0**;**    **public** NeuralNetwork**(**int**[]** layers**)**  **{**  **this.**layers **=** **new** int**[**layers**.**Length**];**  **for** **(**int i **=** 0**;** i **<** layers**.**Length**;** i**++)**  **{**  **this.**layers**[**i**]** **=** layers**[**i**];**  **}**  InitNeurons**();**  InitBiases**();**  InitWeights**();**  **}**  **private** void InitNeurons**()**  **{**  List**<**float**[]>** neuronsList **=** **new** List**<**float**[]>();**  **for** **(**int i **=** 0**;** i **<** layers**.**Length**;** i**++)**  **{**  neuronsList**.**Add**(new** float**[**layers**[**i**]]);**  **}**  neurons **=** neuronsList**.**ToArray**();**  **}**  **private** void InitBiases**()**  **{**  List**<**float**[]>** biasList **=** **new** List**<**float**[]>();**  **for** **(**int i **=** 0**;** i **<** layers**.**Length**;** i**++)**  **{**  float**[]** bias **=** **new** float**[**layers**[**i**]];**  **for** **(**int j **=** 0**;** j **<** layers**[**i**];** j**++)**  **{**  bias**[**j**]** **=** Random**.**Range**(-**0.5f**,** 0.5f**);**  **}**  biasList**.**Add**(**bias**);**  **}**  biases **=** biasList**.**ToArray**();**  **}**  **private** void InitWeights**()**  **{**  List**<**float**[][]>** weightsList **=** **new** List**<**float**[][]>();**  **for** **(**int i **=** 1**;** i **<** layers**.**Length**;** i**++)**  **{**  List**<**float**[]>** layerWeightsList **=** **new** List**<**float**[]>();**  int neuronsInPreviousLayer **=** layers**[**i **-** 1**];**  **for** **(**int j **=** 0**;** j **<** neurons**[**i**].**Length**;** j**++)**  **{**  float**[]** neuronWeights **=** **new** float**[**neuronsInPreviousLayer**];**  **for** **(**int k **=** 0**;** k **<** neuronsInPreviousLayer**;** k**++)**  **{**  neuronWeights**[**k**]** **=** Random**.**Range**(-**0.5f**,** 0.5f**);**  **}**  layerWeightsList**.**Add**(**neuronWeights**);**    **}**  weightsList**.**Add**(**layerWeightsList**.**ToArray**());**  **}**  weights **=** weightsList**.**ToArray**();**    **}**  **public** float**[]** FeedForward**(**float**[]** inputs**)**  **{**  **for** **(**int i **=** 0**;** i **<** inputs**.**Length**;** i**++)**  **{**  neurons**[**0**][**i**]** **=** inputs**[**i**];**  **}**  **for** **(**int i **=** 1**;** i **<** layers**.**Length**;** i**++)**  **{**  int layer **=** i **-** 1**;**  **for** **(**int j **=** 0**;** j **<** neurons**[**i**].**Length**;** j**++)**  **{**  float **value** **=** 0f**;**  **for** **(**int k **=** 0**;** k **<** neurons**[**i **-** 1**].**Length**;** k**++)**  **{**  **value** **+=** weights**[**i **-** 1**][**j**][**k**]** **\*** neurons**[**i **-** 1**][**k**];**  **}**  neurons**[**i**][**j**]** **=** activate**(value** **+** biases**[**i**][**j**]);**  **}**  **}**  **return** neurons**[**neurons**.**Length **-** 1**];**  **}**    **public** float activate**(**float **value)**  **{**  **return** **(**float**)**Math**.**Tanh**(value);**  **}**  **public** void Mutate**(**int chance**,** float val**)**  **{**  **for** **(**int i **=** 0**;** i **<** biases**.**Length**;** i**++)**  **{**  **for** **(**int j **=** 0**;** j **<** biases**[**i**].**Length**;** j**++)**  **{**  biases**[**i**][**j**]** **=** **(**Random**.**Range**(**0f**,** chance**)** **<=** 5**)** **?** biases**[**i**][**j**]** **+=** Random**.**Range**(-**val**,** val**)** **:** biases**[**i**][**j**];**  **}**  **}**  **for** **(**int i **=** 0**;** i **<** weights**.**Length**;** i**++)**  **{**  **for** **(**int j **=** 0**;** j **<** weights**[**i**].**Length**;** j**++)**  **{**  **for** **(**int k **=** 0**;** k **<** weights**[**i**][**j**].**Length**;** k**++)**  **{**  weights**[**i**][**j**][**k**]** **=** **(**Random**.**Range**(**0f**,** chance**)** **<=** 5**)** **?** weights**[**i**][**j**][**k**]** **+=** Random**.**Range**(-**val**,** val**)** **:** weights**[**i**][**j**][**k**];**  **}**  **}**  **}**  **}**  **public** void Crossover**(**NeuralNetwork other**)**  **{**  **for** **(**int i **=** 0**;** i **<** weights**.**Length**;** i**++)**  **{**  **if** **(**Random**.**Range**(**0f**,** 1f**)** **<** 0.5f**)**  **{**  weights**[**i**]** **=** other**.**weights**[**i**];**  **}**  **}**  **for** **(**int i **=** 0**;** i **<** biases**.**Length**;** i**++)**  **{**  **if** **(**Random**.**Range**(**0f**,** 1f**)** **<** 0.5f**)**  **{**  biases**[**i**]** **=** other**.**biases**[**i**];**  **}**  **}**  **}**  **public** int CompareTo**(**NeuralNetwork other**)**  **{**  **if** **(**other **==** **null)**  **return** 1**;**  **if** **(**fitness **>** other**.**fitness**)**  **return** 1**;**  **if** **(**fitness **<** other**.**fitness**)**  **return** **-**1**;**    **return** 0**;**  **}**  **public** NeuralNetwork copy**(**NeuralNetwork nn**)**  **{**  **for** **(**int i **=** 0**;** i **<** biases**.**Length**;** i**++)**  **{**  **for** **(**int j **=** 0**;** j **<** biases**[**i**].**Length**;** j**++)**  **{**  nn**.**biases**[**i**][**j**]** **=** biases**[**i**][**j**];**  **}**  **}**  **for** **(**int i **=** 0**;** i **<** weights**.**Length**;** i**++)**  **{**  **for** **(**int j **=** 0**;** j **<** weights**[**i**].**Length**;** j**++)**  **{**  **for** **(**int k **=** 0**;** k **<** weights**[**i**][**j**].**Length**;** k**++)**  **{**  nn**.**weights**[**i**][**j**][**k**]** **=** weights**[**i**][**j**][**k**];**  **}**  **}**  **}**  **return** nn**;**  **}**  **public** void Load**(**string path**)**  **{**  TextReader tr **=** **new** StreamReader**(**path**);**  int numberOfLines **=** **(**int**)new** FileInfo**(**path**).**Length**;**  string**[]** listLines **=** **new** string**[**numberOfLines**];**  int index **=** 1**;**  **for** **(**int i **=** 1**;** i **<** numberOfLines**;** i**++)**  **{**  listLines**[**i**]** **=** tr**.**ReadLine**();**  **}**  tr**.**Close**();**  **if** **(new** FileInfo**(**path**).**Length **>** 0**)**  **{**  **for** **(**int i **=** 0**;** i **<** biases**.**Length**;** i**++)**  **{**  **for** **(**int j **=** 0**;** j **<** biases**[**i**].**Length**;** j**++)**  **{**  biases**[**i**][**j**]** **=** float**.**Parse**(**listLines**[**index**],** CultureInfo**.**InvariantCulture**.**NumberFormat**);**  index**++;**  **}**  **}**  **for** **(**int i **=** 0**;** i **<** weights**.**Length**;** i**++)**  **{**  **for** **(**int j **=** 0**;** j **<** weights**[**i**].**Length**;** j**++)**  **{**  **for** **(**int k **=** 0**;** k **<** weights**[**i**][**j**].**Length**;** k**++)**  **{**  weights**[**i**][**j**][**k**]** **=** float**.**Parse**(**listLines**[**index**],** CultureInfo**.**InvariantCulture**.**NumberFormat**);** **;**  index**++;**  **}**  **}**  **}**  **}**  **}**  **public** void Save**(**string path**)**  **{**  File**.**Create**(**path**).**Close**();**  StreamWriter writer **=** **new** StreamWriter**(**path**,** **true);**  **for** **(**int i **=** 0**;** i **<** biases**.**Length**;** i**++)**  **{**  **for** **(**int j **=** 0**;** j **<** biases**[**i**].**Length**;** j**++)**  **{**  writer**.**WriteLine**(**biases**[**i**][**j**]);**  **}**  **}**  **for** **(**int i **=** 0**;** i **<** weights**.**Length**;** i**++)**  **{**  **for** **(**int j **=** 0**;** j **<** weights**[**i**].**Length**;** j**++)**  **{**  **for** **(**int k **=** 0**;** k **<** weights**[**i**][**j**].**Length**;** k**++)**  **{**  writer**.**WriteLine**(**weights**[**i**][**j**][**k**]);**  **}**  **}**  **}**  writer**.**Close**();**  **}**  **}** |

## Листинг 2. Файл Manager.cs

|  |
| --- |
| **using** System**.**Collections**;**  **using** System**.**Collections**.**Generic**;**  **using** System**.**IO**;**  **using** TMPro**;**  **using** UnityEngine**;**  **using** UnityEngine**.**UI**;**  **public** class Manager **:** MonoBehaviour  **{**  **[**SerializeField**]** **private** float timeframe**;**  **[**SerializeField**]** **private** int populationSize**;**  **[**SerializeField**]** **private** GameObject prefab**;**  **[**SerializeField**]** **private** int**[]** layers **=** **new** int**[**3**]** **{** 5**,** 3**,** 2 **};**    **[**SerializeField**,** Range**(**0.0001f**,** 1f**)]** **private** float mutationChance **=** 0.01f**;**  **[**SerializeField**,** Range**(**0f**,** 1f**)]** **private** float mutationStrength **=** 0.5f**;**  **[**SerializeField**,** Range**(**0.1f**,** 10f**)]** **private** float gamespeed **=** 1f**;**  **[**SerializeField**]** **private** TextMeshProUGUI scoreText**;**  **[**SerializeField**]** **private** TextMeshProUGUI genText**;**  **[**SerializeField**]** **private** TMP\_InputField InputPopulationSize**;**  **[**SerializeField**]** **private** Slider InputGameSpeed**;**  **[**SerializeField**]** **private** Slider InputMutationChance**;**  **[**SerializeField**]** **private** Slider InputMutationStrength**;**  **[**SerializeField**]** **private** TextMeshProUGUI InputGameSpeedText**;**  **[**SerializeField**]** **private** TextMeshProUGUI InputMutationChanceText**;**  **[**SerializeField**]** **private** TextMeshProUGUI InputMutationStrengthText**;**    **private** List**<**NeuralNetwork**>** networks **=** **new** List**<**NeuralNetwork**>();**  **private** List**<**Bot**>** cars**;**  **private** int bestScore**;**  **private** int generation**;**  **private** string AppPath**;**  **private** void Start**()**  **{**  **if** **(**populationSize **%** 2 **!=** 0**)**  **{**  populationSize **=** 50**;**  **}**  InvokeRepeating**(**"CreateBots"**,** 0.1f**,** timeframe**);**  InitNetworks**();**  **}**  **private** void InitNetworks**()**  **{**  networks**.**Clear**();**  **for** **(**int i **=** 0**;** i **<** populationSize**;** i**++)**  **{**  NeuralNetwork net **=** **new** NeuralNetwork**(**layers**);**  net**.**Load**(**Path**.**Combine**(**Application**.**streamingAssetsPath**,** "Pre-trained.txt"**));**  networks**.**Add**(**net**);**  **}**  **}**  **private** void CreateBots**()**  **{**    Time**.**timeScale **=** gamespeed**;**  **if** **(**cars **!=** **null)**  **{**  **for** **(**int i **=** 0**;** i **<** cars**.**Count**;** i**++)**  **{**  cars**[**i**].**UpdateFitness**();**  Destroy**(**cars**[**i**].**gameObject**);**  **}**  networks**.**Sort**();**  networks**[**populationSize **-** 1**].**Save**(**Path**.**Combine**(**Application**.**streamingAssetsPath**,** "/Save.txt"**));**  generation **+=** 1**;**  scoreText**.**text **=** "Best gen score: " **+** networks**[**populationSize **-** 1**].**fitness**;**  genText**.**text **=** "Current generation: " **+** generation**;**  var newNetworks **=** **new** List**<**NeuralNetwork**>();**  **for** **(**int i **=** 0**;** i **<** populationSize **/** 2**;** i**++)**  **{**  NeuralNetwork parent1 **=** networks**[**i**];**  NeuralNetwork parent2 **=** networks**[**i **+** populationSize **/** 2**];**  NeuralNetwork child1 **=** parent1**.**copy**(new** NeuralNetwork**(**layers**));**  NeuralNetwork child2 **=** parent2**.**copy**(new** NeuralNetwork**(**layers**));**  child1**.**Crossover**(**child2**);**  child1**.**Mutate**((**int**)(**1 **/** mutationChance**),** mutationStrength**);**  child2**.**Mutate**((**int**)(**1 **/** mutationChance**),** mutationStrength**);**  newNetworks**.**Add**(**child1**);**  newNetworks**.**Add**(**child2**);**  **}**  networks**.**Clear**();**  networks**.**AddRange**(**newNetworks**);**  **}**  cars **=** **new** List**<**Bot**>();**  **for** **(**int i **=** 0**;** i **<** populationSize**;** i**++)**  **{**  Bot car **=** **(**Instantiate**(**prefab**,** **new** Vector3**(**0**,** 1.6f**,** **-**16**),** **new** Quaternion**(**0**,** 0**,** 1**,** 0**))).**GetComponent**<**Bot**>();**  car**.**network **=** networks**[**i**];**  cars**.**Add**(**car**);**  **}**  **}**    **public** void TextChange**()**  **{**  InputGameSpeedText**.**text **=** "x" **+** InputGameSpeed**.value;**  InputMutationChanceText**.**text **=** InputMutationChance**.value.**ToString**();**  InputMutationStrengthText**.**text **=** InputMutationStrength**.value.**ToString**();**  **}**    **public** void ChangeParameters**()**  **{**  bool isNumeric **=** int**.**TryParse**(**InputPopulationSize**.**text**,** **out** \_**);**  **if** **(**isNumeric**)**  populationSize **=** int**.**Parse**(**InputPopulationSize**.**text**);**    mutationChance **=** InputMutationChance**.value;**  mutationStrength **=** InputMutationStrength**.value;**  gamespeed **=** InputGameSpeed**.value;**  **}**    **}** |

## Листинг 3. Файл Bot.cs

|  |
| --- |
| **using** System**;**  **using** System**.**Collections**;**  **using** System**.**Collections**.**Generic**;**  **using** UnityEngine**;**  **public** class Bot **:** MonoBehaviour  **{**  **private** float**[]** input **=** **new** float**[**5**];**  **public** NeuralNetwork network**;**  **public** LayerMask raycastMask**;**  **public** float speedFactor**;**  **public** float rotationFactor**;**  **private** int checkpointIndex**;**  **private** bool isCrashed**;**  **[**SerializeField**]** **private** List**<**GameObject**>** checkPoints**;**  **private** void Start**()**  **{**  **for** **(**int i **=** 2**;** i **<** GameObject**.**FindGameObjectsWithTag**(**"CheckPoint"**).**Length **+** 2**;** i**++)**  **{**  checkPoints**.**Add**(**GameObject**.**Find**(**"CheckPoint (" **+** i **+** ")"**));**  **}**  **}**  void FixedUpdate**()**  **{**  **if** **(!**isCrashed**)**  **{**  **for** **(**int i **=** 0**;** i **<** 5**;** i**++)**  **{**  Vector3 newVector **=** Quaternion**.**AngleAxis**(**i **\*** 45 **-** 90**,** **new** Vector3**(**0**,** 1**,** 0**))** **\*** transform**.**right**;**  RaycastHit hit**;**  Ray Ray **=** **new** Ray**(**transform**.**position**,** newVector**);**    **if** **(**Physics**.**Raycast**(**Ray**,** **out** hit**,** 10**,** raycastMask**))**  **{**  input**[**i**]** **=** **(**10 **-** hit**.**distance**)** **/** 10**;**  **}**  **else**  **{**  input**[**i**]** **=** 0**;**  **}**  **}**  float**[]** output **=** network**.**FeedForward**(**input**);**  transform**.**Rotate**(**0**,** output**[**0**]** **\*** rotationFactor**,** 0**,** Space**.**World**);**  transform**.**position **+=** **this.**transform**.**right **\*** **(**output**[**1**]** **\*** speedFactor**);**  **}**  **}**  void OnCollisionEnter**(**Collision collision**)**  **{**    **if** **(**collision**.**collider**.**gameObject**.**layer **==** 7**)**  **{**  **for** **(**int i **=** 0**;** i **<** checkPoints**.**Count**;** i**++)**  **{**  **if** **(**collision**.**collider**.**gameObject **==** checkPoints**[**i**].**gameObject**)**  **{**  **if** **(**i **==** **(**checkpointIndex **+** 1 **+** checkPoints**.**Count**)** **%** checkPoints**.**Count**)**  **{**  checkpointIndex**++;**  **break;**  **}**  **}**  **}**  **}**  **else** **if** **(**collision**.**collider**.**gameObject**.**layer **!=** 6**)**  **{**  isCrashed **=** **true;**  **}**  **}**  **public** void UpdateFitness**()**  **{**  network**.**fitness **=** checkpointIndex**;**  **}**  **}** |

Ссылка на исходный проект -