МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА - Российский технологический университет»

РТУ МИРЭА

Институт Информационных Технологий
Кафедра Вычислительной Техники (ВТ)

ОТЧЁТ ПО ПРАКТИЧЕСКИМ РАБОТАМ

по дисциплине

«Проектирование и обучение нейронных сетей»

Выполнил студенты группы: Оганнисян Г.А.
ИКБО-15-22 Кудинов А.В.

Принял старший преподаватель кафедры ВТ Семенов Р.Э.

Практическая работа выполнена «__»_____2024 г.

«Зачтено» «__»_____2024 г.

Москва 2024 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Нейронные сети являются ключевым элементом современных технологий искусственного интеллекта и машинного обучения. Их способность эффективно моделировать сложные нелинейные зависимости и адаптироваться к разнообразным задачам делает их незаменимыми в таких областях, как обработка изображений, распознавание речи, прогнозирование и многое другое. С каждым годом увеличивается объем данных и вычислительных ресурсов, что способствует развитию более сложных и мощных архитектур нейронных сетей.

Основой успешного применения нейронных сетей являются методы их обучения, позволяющие оптимизировать параметры модели для достижения высокой точности и надежности. Существуют различные подходы к обучению нейронных сетей, начиная от классических алгоритмов, таких как правила Хебба и дельта-правило, и заканчивая современными методиками глубокого обучения, включая алгоритмы обратного распространения ошибки и специализированные архитектуры, такие как сверточные и рекуррентные нейронные сети.

Изучение различных алгоритмов обучения позволяет выбрать наиболее подходящий метод для конкретной задачи, учитывая особенности данных и требования к модели. Кроме того, понимание принципов работы различных методов обучения способствует разработке более эффективных и устойчивых нейронных сетей, способных справляться с разнообразными вызовами в реальных приложениях.

Таким образом, исследование и разработка методов обучения нейронных сетей продолжают оставаться важным направлением в области искусственного интеллекта, открывая новые возможности для создания интеллектуальных систем, способных решать сложные задачи и адаптироваться к изменяющимся условиям.

1 ОБУЧЕНИЕ НЕЙРОННОЙ СЕТИ ПО ПРАВИЛАМ ХЕББА

1.1 Описание алгоритма

Принцип обучения по правилу Хебба можно описать как процесс, при котором нейронная сеть или отдельный нейрон изменяет свою структуру на основе опыта, чтобы улучшить распознавание и обработку информации. Основная идея заключается в том, что нейрон «запоминает» связи, которые были полезными в прошлом. Это делает его более чувствительным к повторяющимся или значимым паттернам.

Алгоритм функционирует на основе механизма усиления, похожего на то, как человеческий мозг укрепляет связи между нейронами через повторяющееся совместное возбуждение. Например, если определенные сенсорные сигналы часто активируют нейрон одновременно, их взаимосвязь становится сильнее, что улучшает способность сети распознавать те же сигналы в будущем. По сути, это имитация процесса обучения, при котором правильные ответы становятся более вероятными благодаря усилению связей, ассоциированных с успешной активацией.

Правило Хебба помогает создавать ассоциативные связи, что делает алгоритм полезным для задач, где требуется запоминание и усиление определенных реакций или распознавание устойчивых шаблонов.

1.2 Программная реализация обучения нейронной сети по правилам Хебба

Выполним программную реализацию нейронной сети по правилам Хебба на языке GoLang (Приложение А). В результате работы программы необходимо показать, как нейронная сеть обучается на основе правила Хебба, и продемонстрировать, как обновляются веса после каждой итерации обучения. В

процессе выполнения программа отображает начальные веса, изменения весов после каждой эпохи, а также итоговые результаты прогнозов для всех возможных входных данных. Результат представлен на рисунке 1.1

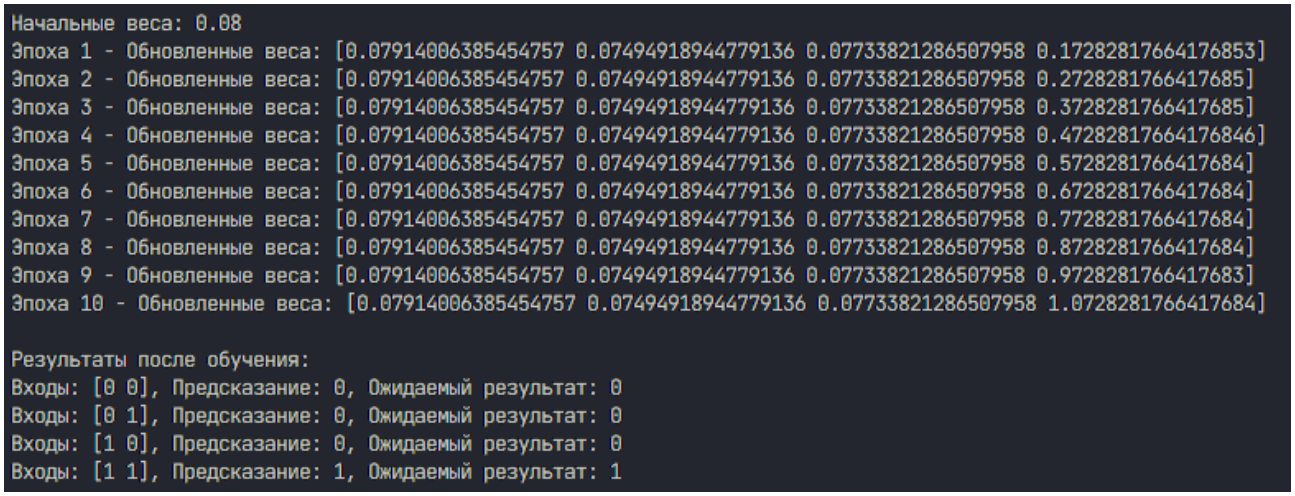


Рисунок 1.1 – Результат работы программы обучения нейронной сети по правилам Хебба

2 ОБУЧЕНИЕ НЕЙРОННОЙ СЕТИ С ПОМОЩЬЮ ДЕЛЬТА-ПРАВИЛА

2.1 Описание алгоритма

Алгоритм обучения с использованием дельта-правила основан на минимизации ошибки путем корректировки весов однослойного перцептрона. Идея заключается в том, чтобы постепенно изменять весовые коэффициенты таким образом, чтобы уменьшить разницу между предсказанными и целевыми значениями.

В начале весы и порог инициализируются случайными значениями. В процессе обучения перцептрон проходит через тренировочные примеры, вычисляет взвешенную сумму входных сигналов и применяет активационную функцию, которая возвращает бинарный результат (0 или 1). Разница между предсказанным и истинным значениями называется ошибкой. Дельта-правило корректирует весовые коэффициенты, используя эту ошибку и заданную скорость обучения, чтобы улучшить точность модели.

Корректировка происходит до тех пор, пока ошибка не станет минимальной, что позволяет модели лучше классифицировать входные данные. Алгоритм эффективно справляется с задачами, где данные можно разделить линейной границей, что делает его фундаментальной техникой для обучения простых нейронных сетей.

2.2 Программная реализация обучения нейронной сети с помощью дельта-правила

Выполним программную реализацию нейронной сети с помощью дельта-правила на языке GoLang (Приложение Б). В результате работы программы необходимо отобразить процесс обучения нейронной сети, показывая изменения весов и порога после каждой эпохи, а также оценить эффективность обучения,

анализируя уменьшение общей ошибки в процессе итераций. Важно проверить корректность работы обученной модели, сравнивая ее предсказания с целевыми значениями, и зафиксировать итоговые параметры сети, чтобы сделать вывод о способности модели выполнять задачу классификации. Результат представлен на рисунке 2.1

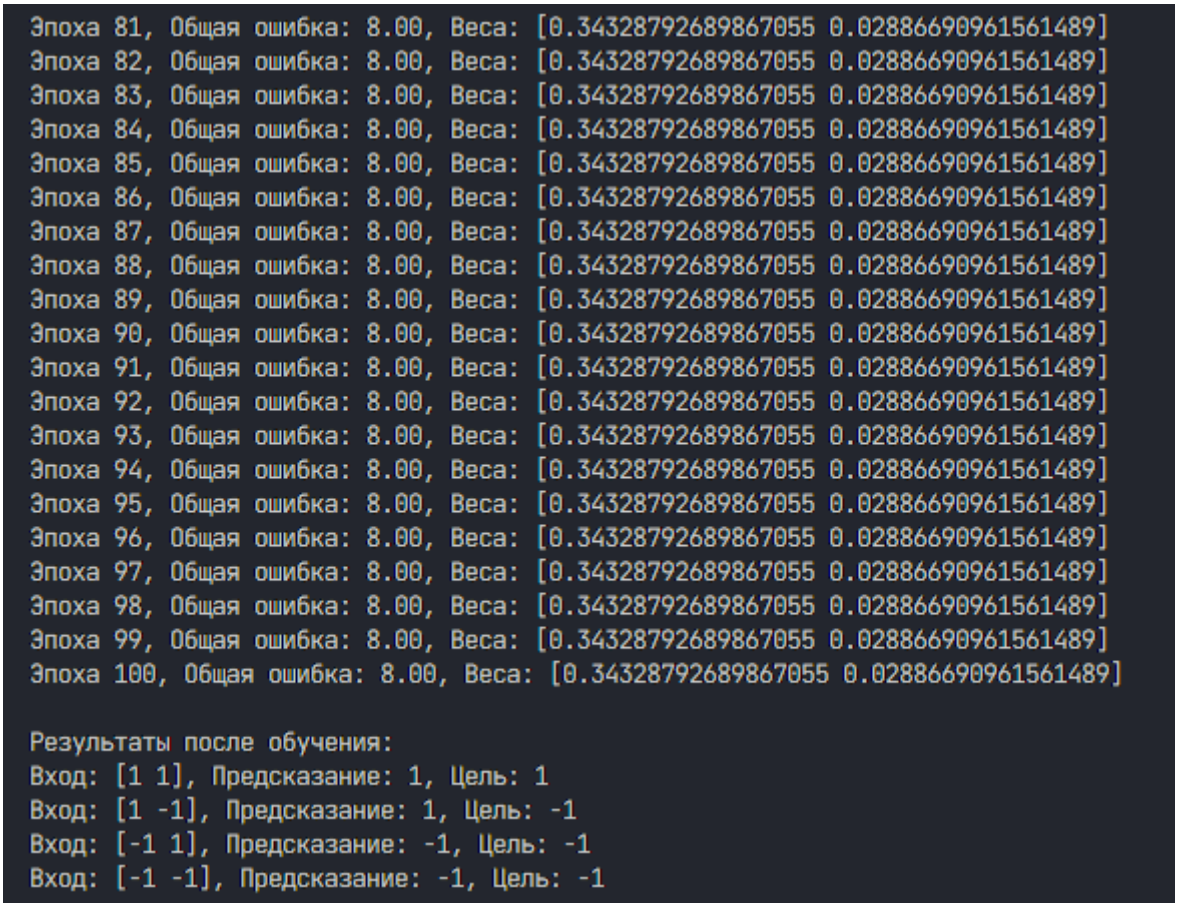


Рисунок 2.1 – Результат работы программы обучения нейронной с помощью дельта-правила

3 ОБУЧЕНИЕ НЕЙРОННОЙ СЕТИ С ИСПОЛЬЗОВАНИЕМ АЛГОРИТМА ОБРАТНОГО РАСПРОСТРАНЕНИЯ ОШИБКИ

3.1 Описание алгоритма

Алгоритм обратного распространения ошибки используется для обучения многослойных нейронных сетей. Основная идея заключается в минимизации разницы между фактическими выходными значениями сети и целевыми значениями, путем корректировки весов с учетом этой ошибки. Это достигается за счет вычисления градиентов и использования их для обновления параметров сети.

Процесс обучения начинается с прямого распространения: входные данные передаются через слои нейронной сети, чтобы получить предсказание. Сначала входные значения преобразуются с помощью весов и смещений и активируются функцией активации (сигмоида в данном случае). Затем сигналы передаются к следующему слою, и выходное значение рассчитывается аналогичным образом.

Затем происходит обратное распространение ошибки. Вычисляется ошибка как разница между ожидаемым выходом и предсказанным значением. Для минимизации этой ошибки применяется метод градиентного спуска, с использованием производной сигмоидальной функции для корректировки весов. Сначала рассчитываются градиенты для выходного слоя, затем ошибка передается назад к скрытому слою, и градиенты вычисляются также для него.

Веса и смещения обновляются пропорционально градиентам и скорости обучения, что позволяет сети постепенно становиться более точной в своих предсказаниях. Алгоритм повторяется в течение многих эпох, пока ошибка не станет минимальной или пока не достигнется заданное количество итераций. В результате сеть обучается ассоциировать входные данные с правильными выходами, что позволяет ей эффективно решать задачи классификации.

3.2 Программная реализация обучения нейронной сети с использованием алгоритма обратного распространения ошибки

Выполним программную реализацию нейронной сети используя алгоритм обратного распространения ошибки на языке GoLang (Приложение В). В результате работы программы необходимо продемонстрировать процесс обучения сети, показав, как уменьшается ошибка с течением времени, и оценить эффективность обученной модели, сравнив предсказания с ожидаемыми значениями. Итоговые параметры сети помогут сделать вывод о ее способности решать задачу классификации. Результат представлен на рисунке 3.1.

```
Веса ур. 0: [[0.755545036904078 1.1259447982470339] [1.215762368334523 0.9773741684276432]]
Веса ур. 1: [1.5059764655035626 0.7499827013719051]
Результат: [0 0.9497143795251333 0.991845389695178 0.7872119801775673]
Цель: [0 1 1 1]
Новые веса ур. 0: [[0.7544719192365712 1.1247459211100124] [1.2146892506670162 0.9761752912906219]]
Новые веса ур. 1: [1.5046564678240824 0.7470123345571686]
Итерация: 0 ----------
Веса ур. 0: [[0.7544719192365712 1.1247459211100124] [1.2146892506670162 0.9761752912906219]]
Веса ур. 1: [1.5046564678240824 0.7470123345571686]
Результат: [0 0.9497143795251333 0.991845389695178 0.7872119801775673]
Цель: [0 1 1 1]
Новые веса ур. 0: [[0.7544719192365712 1.1247459211100124] [1.2146892506670162 0.9761752912906219]]
Новые веса ур. 1: [1.5046564678240824 0.7470123345571686]
Итерация: 1 ----------
Веса ур. 0: [[0.7544719192365712 1.1247459211100124] [1.2146892506670162 0.9761752912906219]]
Веса ур. 1: [1.5046564678240824 0.7470123345571686]
Результат: [0 0.9495226135854262 0.991845389695178 0.7872119801775673]
Цель: [0 1 1 1]
Новые веса ур. 0: [[0.7567800014909039 1.1254407450719985] [1.2169973329213488 0.9768701152526079]]
Новые веса ур. 1: [1.5061888667849506 0.748729898011334]
Итерация: 2 ----------
Веса ур. 0: [[0.7567800014909039 1.1254407450719985] [1.2169973329213488 0.9768701152526079]]
Веса ур. 1: [1.5061888667849506 0.748729898011334]
Результат: [0 0.9495226135854262 0.991982731477506 0.7872119801775673]
Цель: [0 1 1 1]
Новые веса ур. 0: [[0.7570670208599675 1.1256643727806328] [1.2172843522904124 0.9770937429612421]]
Новые веса ур. 1: [1.5065414757028657 0.7490284550382801]
Итерация: 3 ----------
Веса ур. 0: [[0.7570670208599675 1.1256643727806328] [1.2172843522904124 0.9770937429612421]]
Веса ур. 1: [1.5065414757028657 0.7490284550382801]
Результат: [0 0.9495226135854262 0.991982731477506 0.7876106951768103]
Цель: [0 1 1 1]
Новые веса ур. 0: [[0.7559648752006283 1.1244725216456817] [1.2161822066310732 0.9759018918262908]]
Новые веса ур. 1: [1.5051862490584855 0.7460717732648221]
```

Рисунок 3.1 – Результат работы программы обучения нейронной сети с использованием алгоритма обратного распространения ошибки

4 ОБУЧЕНИЕ НЕЙРОННОЙ СЕТИ С ГАУССОВОЙ РАДИАЛЬНО-БАЗИСНОЙ ФУНКЦИЕЙ

4.1 Описание алгоритма

Алгоритм обучения нейронной сети с радиально-базисными функциями (RBF) включает несколько ключевых этапов, которые обеспечивают эффективную обработку данных и классификацию. Сеть RBF состоит из входного слоя, скрытого слоя с радиально-базисными функциями, которые определяют активации нейронов на основе расстояния до заданных центров, и выходного слоя, который выполняет линейную комбинацию этих активаций.

Описание алгоритма начинается с инициализации центров базисных функций с помощью кластеризации K-средних, что позволяет сети выделить основные группы данных. Затем уточнение центров выполняется с помощью метода, который корректирует их позиции, чтобы лучше представлять структуру данных. Для каждой базисной функции вычисляется ширина (параметр σ), которая определяет, как быстро убывает значение функции при увеличении расстояния от центра.

Сеть обучается минимизировать ошибку с помощью метода псевдообратной матрицы, который позволяет оптимально подобрать веса выходного слоя. После завершения обучения модель оценивается на тестовой выборке, и результаты представляются в виде показателей точности, матрицы путаницы и графической визуализации. Алгоритм обеспечивает высокую точность классификации, адаптируясь к особенностям распределения данных.

4.2 Программная реализация обучения нейронной сети с гауссовой радиально-базисной функцией

Выполним программную реализацию обучения нейронной сети с гауссовой радиально-базисной функцией на языке GoLang (Приложение Г). В

результате работы программы необходимо оценить, как радиально-базисные функции с различными центрами и ширинами позволяют сети эффективно классифицировать данные. Нужно продемонстрировать процесс инициализации и уточнения центров базисных функций, а также показать, как сеть адаптируется к данным, минимизируя ошибку. Важно представить показатели точности модели на обучающей и тестовой выборках. Результат представлен на рисунке 4.1.

```
Новые веса ур. 0: [[-0.013147732508099882 3.8496250157898073] [0.586517049032803 3.870166043750523]]
Новые веса ур. 1: [1.0173127400934248 -2.5747827777239856]
----------------------------
Итерация: 0 ----------
Веса ур. 0: [[-0.013147732508099882 3.8496250157898073] [0.586517049032803 3.870166043750523]]
Веса ур. 1: [1.0173127400934248 -2.5747827777239856]
Результат: [0.15094522055230106 0.9496947104693563 0.8369793411285439 0.9939462788254393]
Цель: [0 1 1 1]
Новые веса ур. 0: [[-0.027921118745051196 3.8877826232992305] [0.5717436627958516 3.9083236512599466]]
Новые веса ур. 1: [1.002577331893997 -2.5895181859234135]
----------------------------
Итерация: 1 ----------
Веса ур. 0: [[-0.027921118745051196 3.8877826232992305] [0.5717436627958516 3.9083236512599466]]
Веса ур. 1: [1.002577331893997 -2.5895181859234135]
Результат: [0.15094522055230106 0.9500977128145014 0.8369793411285439 0.9939462788254393]
Цель: [0 1 1 1]
Новые веса ур. 0: [[-0.027613610303666228 3.887945477955572] [0.5720511712372366 3.9084865059162883]]
Новые веса ур. 1: [1.0028839560448088 -2.5896345783905863]
----------------------------
Итерация: 2 ----------
Веса ур. 0: [[-0.027613610303666228 3.887945477955572] [0.5720511712372366 3.9084865059162883]]
Веса ур. 1: [1.0028839560448088 -2.5896345783905863]
Результат: [0.15094522055230106 0.9500977128145014 0.8380273146553014 0.9939462788254393]
Цель: [0 1 1 1]
Новые веса ур. 0: [[-0.025291769077986692 3.8909857524604345] [0.5743730124629162 3.9115267804211507]]
Новые веса ур. 1: [1.007150745553153 -2.590808063430939]
----------------------------
Итерация: 3 ----------
Веса ур. 0: [[-0.025291769077986692 3.8909857524604345] [0.5743730124629162 3.9115267804211507]]
Веса ур. 1: [1.007150745553153 -2.590808063430939]
Результат: [0.15094522055230106 0.9500977128145014 0.8380273146553014 0.9917735853295583]
Цель: [0 1 1 1]
Новые веса ур. 0: [[-0.025287109509816774 3.8909857524873486] [0.5743776720310861 3.9115267804480647]]
Новые веса ур. 1: [1.007159308252158 -2.590808063450166]
----------------------------
```

Рисунок 4.1 – Результат нейронной сети с гауссовой радиально-базисной функции

5 ОБУЧЕНИЕ НЕЙРОННОЙ СЕТИ С ИСПОЛЬЗОВАНИЕМ КАРТЫ КОХОНЕНА

5.1 Описание алгоритма

Карты Кохонена, также известные как самоорганизующиеся карты (SOM), – это нейронные сети, которые выполняют кластеризацию и визуализацию данных, сохраняя топологические связи в упрощенном, двумерном пространстве. Алгоритм обучения SOM начинается с инициализации весов нейронов случайными значениями, чтобы создать начальное распределение на карте. Затем, в процессе обучения, сеть проходит через множество итераций, в каждой из которых случайно выбирается входной вектор из данных. Для этого входного вектора находится ближайший нейрон на карте, называемый Best Matching Unit (BMU), определяемый на основе евклидового расстояния.

После того как BMU идентифицирован, веса этого нейрона, а также его соседей на карте, обновляются так, чтобы они стали ближе к выбранному входному вектору. Обновление весов осуществляется с использованием функции соседства, которая определяет степень влияния на каждый нейрон в зависимости от его расстояния до BMU. Обычно это влияние убывает по гауссовому закону, а радиус соседства и скорость обучения уменьшаются со временем, чтобы сначала сеть могла крупными шагами адаптироваться к структуре данных, а затем постепенно тонко настраивать распределение весов.

В результате карта Кохонена самоорганизуется, так что похожие входные данные оказываются сгруппированными в соседние области карты. Это позволяет эффективно визуализировать многомерные данные и находить кластеры, сохраняя пространственную структуру данных в более удобной для анализа форме. SOM широко применяются для анализа данных, выявления паттернов и создания топологических карт данных в различных приложениях, включая обработку изображений, анализ данных и биоинформатику.

5.2 Программная реализация обучения нейронной сети по с использованием карты Кохонена

Выполним программную реализацию обучения нейронной сети с использованием карты Кохонена на языке Golang (Приложение Д). В результате работы программы необходимо визуализировать процесс самоорганизации карты, показывая, как нейроны на карте постепенно адаптируются к структуре входных данных. Следует продемонстрировать изменения в весах нейронов в ходе обучения и оценить, насколько карта эффективно группирует и отображает данные. Финальная визуализация карты должна показать, как данные распределены и какие кластеры сформировались. Это позволит сделать выводы о способности карты Кохонена упрощать и структурировать сложные многомерные данные, сохраняя топологическую целостность. Результат представлена на рисунке 5.1.

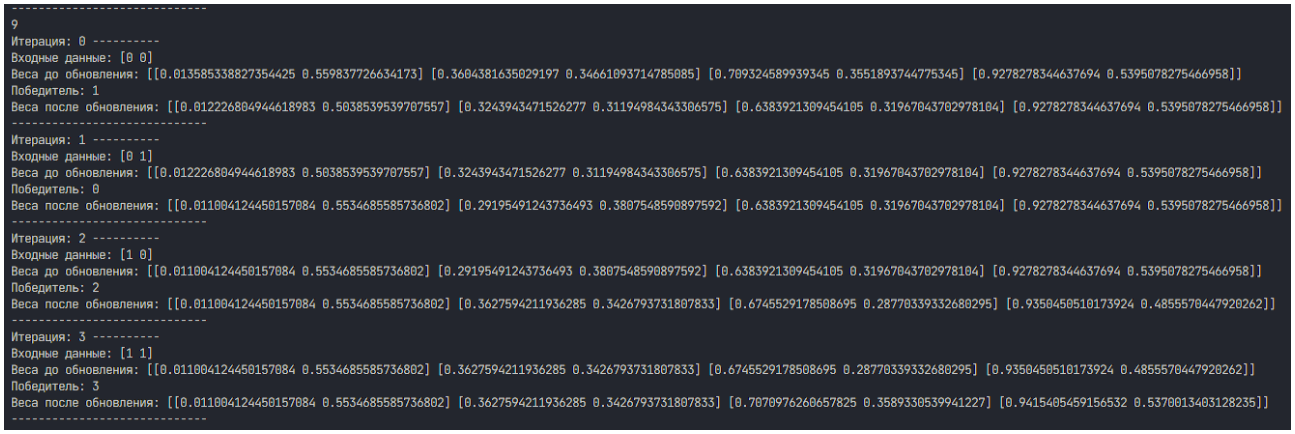


Рисунок 5.1 – Результат работы нейронной сети с картой Кохонена

6 ОБУЧЕНИЕ НЕЙРОННОЙ СЕТИ С ИСПОЛЬЗОВАНИЕМ АЛГОРИТМА ВСТРЕЧНОГО РАСПРОСТРАНЕНИЯ

6.1 Описание алгоритма

Алгоритм встречного распространения (Counter-Propagation) сочетает в себе карты Кохонена и слой Гросберга для решения задачи классификации и ассоциативного обучения. Этот алгоритм объединяет преимущества самоорганизующихся карт для кластеризации и нейронного слоя для ассоциации кластеров с целевыми классами.

В данном алгоритме карта Кохонена отвечает за кластеризацию входных данных. На каждом шаге обучения случайно выбирается входной вектор, и для него определяется ближайший нейрон на карте Кохонена, называемый Best Matching Unit (BMU). Затем веса BMU и его соседей корректируются, чтобы сделать их ближе к входному вектору, что способствует самоорганизации карты. Эта часть алгоритма напоминает классическое обучение карты Кохонена, где топологическая структура данных сохраняется на двумерной карте.

После этого слой Гросберга связывает активированный нейрон на карте Кохонена с целевым классом. Для BMU обновляются веса слоя Гросберга в направлении целевого вектора (one-hot encoded), что позволяет каждому нейрону ассоциироваться с определенной категорией. Со временем это обучение создает ассоциативные связи между кластерами на карте Кохонена и классами, заданными целевыми метками.

В результате карта Кохонена самоорганизуется, группируя похожие входные данные вместе, а слой Гросберга создает связи, позволяющие сети классифицировать новые входные данные по соответствующим классам.

6.2 Программная реализация обучения нейронной сети по с использованием карты Кохонена

Выполним программную реализацию обучения нейронной сети с использованием алгоритма встречного распространения на языке GoLang (Приложение Е). В результате работы программы необходимо продемонстрировать, как карта Кохонена самоорганизуется, группируя похожие входные данные, и как слой Гросберга связывает эти группы с целевыми классами. Это позволит сделать вывод о способности сети классифицировать данные с использованием алгоритма встречного распространения и о сохранении топологических связей в ходе обучения. Результат представлена на рисунке 6.1.

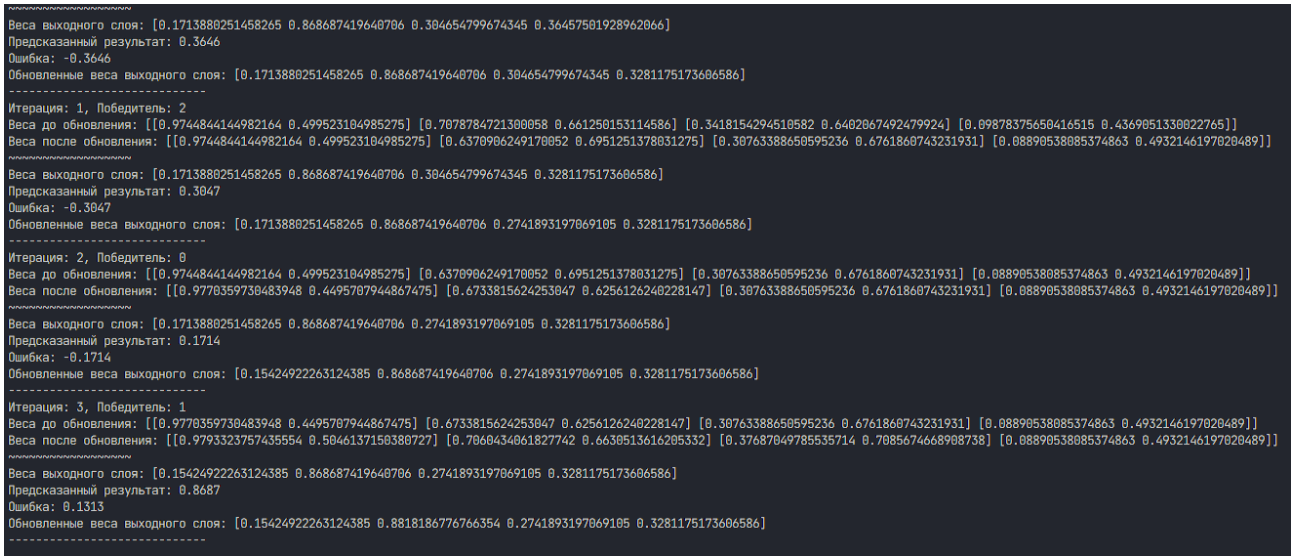


Рисунок 6.1 – Результат работы нейронной сети с использованием встречного распространнения

7 ГЛУБОКОЕ ОБУЧЕНИЕ НЕЙРОННОЙ СЕТИ С ИСПОЛЬЗОВАНИЕМ МОДЕЛИ DO-RNN

7.1 Описание алгоритма

Глубокое обучение нейронной сети с использованием модели DO-RNN (Deep Output Recurrent Neural Network) представляет собой подход, предназначенный для анализа временных рядов и предсказания их поведения. DO-RNN комбинирует преимущества рекуррентных нейронных сетей (RNN), которые хорошо справляются с последовательными данными, с глубокими выходными слоями, позволяющими более гибко обрабатывать сложные зависимости в данных.

Основной задачей такой модели является извлечение и обработка временных зависимостей из входных последовательностей. Рекуррентные слои, такие как LSTM, используются для хранения и передачи информации о предыдущих шагах, что делает модель способной предсказывать будущее значение временного ряда на основе исторических данных. Выходные слои модели дополнительно усиливают возможности DO-RNN, позволяя адаптивно преобразовывать скрытые состояния в точные предсказания.

В процессе обучения модель минимизирует разницу между предсказанными значениями и фактическими, что достигается через оптимизацию параметров с использованием функции потерь и алгоритма градиентного спуска. Входные данные, такие как финансовые временные ряды, предварительно масштабируются и структурируются в последовательности фиксированной длины, чтобы соответствовать требованиям модели. После обучения DO-RNN способна выполнять прогнозирование на новых данных, обеспечивая точные и интерпретируемые результаты.

DO-RNN находит применение в финансовом анализе, прогнозировании спроса, обработке сигналов и других задачах, где важно учитывать временную 22 динамику и сложные зависимости внутри данных.

7.2 Программная реализация глубокого обучения нейронной сети с использованием модели DO-RNN

Выполним программную реализацию нейронной сети с глубоким обучением на модели DO-RNN на языке GoLang (Приложения Ж). Представлена на рисунке 7.1. В результате работы программы необходимо обучить модель DO-RNN для анализа временных рядов и прогнозирования их будущих значений, используя данные о финансовых активах.

Модель DO-RNN продемонстрировала свою способность к анализу временных рядов и точному прогнозированию на тестовой выборке. Итоговые метрики, такие как RMSE, и визуализация графиков предсказаний подтвердили эффективность предложенного подхода.

```
Цель: [0]
Вход: [0 1]
Скрытое состояние: [0.6867694115373891 0.6421962294449367]
Выход: [0.7139841272278074]
Цель: [0]
Вход: [1 0]
Скрытое состояние: [0.6127953762932208 0.5557832807257271]
Выход: [0.690488766884867]
Цель: [0]
Вход: [1 1]
Скрытое состояние: [0.7762093870927417 0.6917809768905175]
Выход: [0.7322566054473573]
Цель: [1]
Итерация: 99
Вход: [0 0]
Скрытое состояние: [0.5 0.5]
Выход: [0.6656455231616087]
Цель: [0]
Вход: [0 1]
Скрытое состояние: [0.6867386876092445 0.6421583098717526]
Выход: [0.7133801639220387]
Цель: [0]
Вход: [1 0]
Скрытое состояние: [0.6127560181452693 0.5557372785500918]
Выход: [0.6899291024817487]
Цель: [0]
Вход: [1 1]
Скрытое состояние: [0.7761559934678478 0.6917063396281319]
Выход: [0.7316083936218172]
Цель: [1]
```

Рисунок 7.1 – Результат работы нейронной сети с использование модели DO-RNN

8 ОБУЧЕНИЕ НЕЙРОННОЙ СЕТИ С ИСПОЛЬЗОВАНИЕМ МОДЕЛИ R-CNN

8.1 Описание алгоритма

Обучение нейронной сети с использованием модели R-CNN (Region-based Convolutional Neural Network) представляет собой процесс, направленный на решение задач классификации и детектирования объектов. R-CNN является мощной архитектурой, которая сочетает сверточные нейронные сети для извлечения признаков с дополнительными методами для точного определения областей, содержащих объекты.

Основная идея алгоритма состоит в том, чтобы разделить изображение на множество возможных регионов интереса (Region Proposals), а затем пропустить каждый из них через сверточную сеть для извлечения признаков. На основе полученных признаков модель классифицирует объект, находящийся в этом регионе, и уточняет координаты его границ. Это позволяет эффективно совмещать локализацию объектов с их идентификацией.

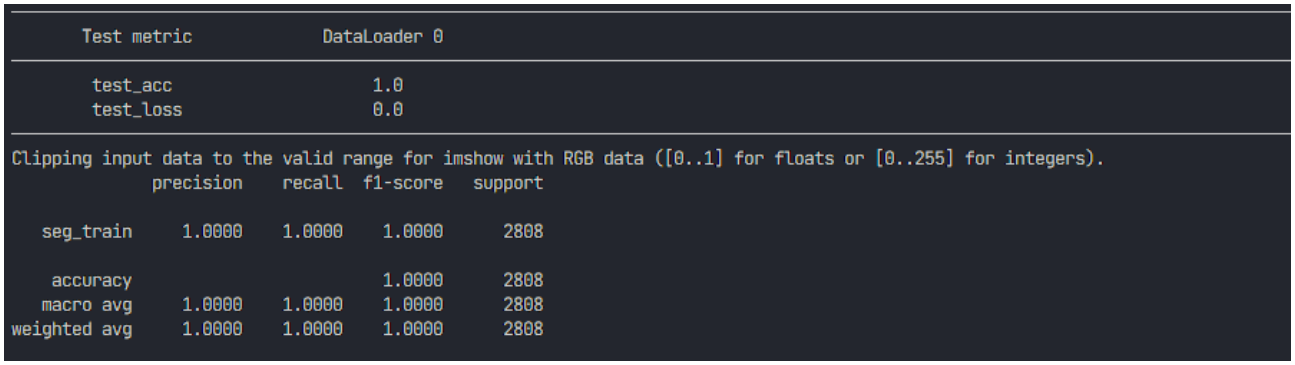
В процессе обучения модель оптимизирует свои параметры, минимизируя ошибки в классификации и регрессии координат, используя функции потерь. На тестовой выборке оценивается точность классификации и определяется качество детектирования объектов. Финальная модель позволяет обрабатывать как стандартные тестовые наборы, так и новые изображения, обеспечивая высокую точность и универсальность.

R-CNN широко используется в задачах компьютерного зрения, таких как распознавание объектов на фотографиях, видеоаналитика и автономные системы, благодаря своей способности эффективно работать с изображениями, содержащими множество различных объектов.

8.2 Программная реализация обучения нейронной сети с использованием модели R-CNN

Выполним программную реализацию нейронной сети с обучением на модели R-CNN на языке GoLang (Приложения 3). Представлена на рисунке 8.1. В результате работы программы необходимо обучить нейронную сеть на одном из стандартных наборов данных, таких как CIFAR-10 или CIFAR 100, и провести оценку её производительности.

Модель R-CNN продемонстрировала высокую точность классификации и возможность визуализировать её работу, включая правильные и ошибочные предсказания. Итоговая визуализация и метрики позволили сделать выводы о качестве и применимости модели к различным задачам классификации изображений.



```
       Test metric            DataLoader 0
┌──────────────────────────┬──────────────────────────┐
        test_acc                    1.0
        test_loss                   0.0
└──────────────────────────┴──────────────────────────┘
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
              precision    recall  f1-score   support

   seg_train     1.0000    1.0000    1.0000      2808

    accuracy                         1.0000      2808
   macro avg     1.0000    1.0000    1.0000      2808
weighted avg     1.0000    1.0000    1.0000      2808
```

Рисунок 8.1 – Результат работы нейронной сети с использованием модели R-CNN

ЗАКЛЮЧЕНИЕ

В процессе выполнения практических работ были изучены и реализованы различные методы обучения нейронных сетей, начиная от классических подходов и заканчивая современными алгоритмами глубокого обучения. Каждый из рассмотренных методов обладает своими уникальными характеристиками, преимуществами и ограничениями, что позволяет выбирать наиболее подходящий алгоритм в зависимости от специфики решаемой задачи и требований к модели.

Правила Хебба и дельта-правило представляют собой фундаментальные методы обучения, обеспечивающие базовое понимание принципов адаптации весов в нейронных сетях. Эти методы просты в реализации и подходят для задач с ограниченными ресурсами, однако их возможности ограничены по сравнению с более современными подходами.

Алгоритм обратного распространения ошибки стал стандартом де-факто для обучения многослойных нейронных сетей благодаря своей эффективности и способности обучать глубокие архитектуры. Он позволяет значительно повысить точность моделей, однако требует большого объема данных и вычислительных ресурсов, а также тщательно подобранных гиперпараметров.

Использование гауссовых радиально-базисных функций расширяет возможности нейронных сетей в задачах аппроксимации и кластеризации, обеспечивая высокую гибкость и точность. Карты Кохонена, в свою очередь, эффективны для визуализации и кластеризации данных, однако их применение ограничено задачами без четкой структуры выходных данных.

Современные модели глубокого обучения, такие как DO-RNN и R-CNN, демонстрируют высокую производительность в обработке последовательных и изображенческих данных соответственно. Эти модели позволяют решать сложные задачи, такие как распознавание объектов и обработка временных рядов, однако их обучение требует значительных вычислительных мощностей и большого объема размеченных данных.

Анализ проведенных экспериментов показал, что выбор метода обучения напрямую влияет на конечные результаты нейронной сети. Классические методы подходят для простых задач и образовательных целей, тогда как современные алгоритмы глубокого обучения необходимы для решения комплексных и высокоточных задач в реальных приложениях.

В заключение можно отметить, что развитие методов обучения нейронных сетей является динамичной областью исследований, требующей постоянного совершенствования и адаптации к новым вызовам. Будущие исследования могут быть направлены на интеграцию различных подходов, улучшение алгоритмов оптимизации, а также на разработку более эффективных архитектур нейронных сетей, способных работать с ограниченными ресурсами и обеспечивать высокую точность при минимальных затратах вычислительных мощностей.

Таким образом, проделанная работа демонстрирует разнообразие существующих методов обучения нейронных сетей и подчеркивает важность выбора подходящего алгоритма для достижения наилучших результатов в конкретных задачах. Полученные знания и практические реализации могут служить основой для дальнейших исследований и разработок в области искусственного интеллекта и машинного обучения.

СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ

1. Широков, И.Б. Анализ технологий глубокого обучения с подкреплением для систем машинного зрения / И.Б. Широков, С.В. Колесова, В.А. Кучеренко, М.Ю. Серебряков / Известия ТулГУ 2022.

2. Сущеня, Р.В. Нейронные сети и их классификация. Основные виды нейронных сетей / Р.В. Сущеня, А.Э. Кокаев / Международный научный журнал «Вестник науки» 2023.

3. Паршин, А.И. Случайное мультимодальное глубокое обучение в задаче распознавания изображений / А.И. Паршин, М.Н. Аралов, В.Ф. Барабанов, Н.И. Гребенникова / Вестник Воронежского государственного университета 2021.

ПРИЛОЖЕНИЯ

Приложение А – Код реализации обучения нейронной сети по правилам Хебба GoLang.

Приложение Б – Код реализации обучения нейронной сети с помощью дельта-правила на языке GoLang.

Приложение В – Код реализации обучения нейронной сети с использованием алгоритма обратного распространения ошибки на языке GoLang.

Приложение Г – Код реализации обучения нейронной сети с гауссовой радиально-базисной функцией на языке GoLang.

Приложение Д – Код реализации обучения нейронной сети с использованием карты Кохонена на языке GoLang.

Приложение Е – Код реализации обучения нейронной сети с использованием алгоритма встречного распространения на языке GoLang.

Приложение Ж – Код реализации обучения нейронной сети с использование модели DO-RNN на языке GoLang.

Приложение З – Код реализации обучения нейронной сети с использованием модели R-CNN на языке GoLang.

Приложение А

Код реализации обучения нейронной сети по правилам Хебба GoLang.

Листинг А – Код реализации обучения нейронной сети по правилам Хебба

```
package main

import ("fmt" "math/rand")

func hebbLearn(inputs []float64, outputs []float64, w []float64, n float64)
[]float64 {
	for i := 0; i < len(inputs); i++ {
		w[i] += n * inputs[i] * outputs[i]
	}
	return w
}
func predict(input float64, weight float64) int {
	if input*weight > 0.5 { // Порог
		return 1
	}
	return 0
}
func main() {
	input1 := []float64{0.0, 0.0, 1.0, 1.0}
	input2 := []float64{0.0, 1.0, 0.0, 1.0}
	inputs := make([]float64, len(input1))
	for i := 0; i < len(input1); i++ {
		if input1[i] == 1.0 && input2[i] == 1.0 {
			inputs[i] = 1.0
		} else {
			inputs[i] = 0.0
		}
	}
	outputs := []float64{0.0, 0.0, 0.0, 1.0}
	weights := make([]float64, len(inputs))
	for i := 0; i < len(weights); i++ {
		weights[i] = rand.Float64() * 0.1 // Начальные веса
	}
	n := 0.1
	epochs := 10
	fmt.Printf("Начальные веса: %.2f\n", weights[0])
	for epoch := 1; epoch <= epochs; epoch++ {
		weights = hebbLearn(inputs, outputs, weights, n)
		fmt.Printf("Эпоха %d - Обновленные веса: %v\n", epoch, weights)
	}
	fmt.Println("\nРезультаты после обучения:")
	for i := 0; i < len(inputs); i++ {
		pred := predict(inputs[i], weights[i])
		fmt.Printf("Входы: [%d %d], Предсказание: %d, Ожидаемый результат:
%d\n",
			int(input1[i]), int(input2[i]), pred, int(outputs[i]))
	}
}
```

Приложение Б

Код реализации обучения нейронной сети с помощью дельта-правила на языке GoLang.

Листинг Б – Код реализации обучения нейронной сети с помощью дельта-правила

```
package main

import (
	"fmt"
	"math/rand"
	"time"
)

// Функция активации
func activate(input, weights []float64, porog float64) (sum float64) {
	for i := 0; i < len(input); i++ {
		sum += input[i] * weights[i]
	}
	if sum > porog {
		return 1.0
	}
	return -1.0
}

// Дельта-правило для обновления весов
func delta(input, weights []float64, target, n, porog float64) (updatedWeights
[]float64, err float64) {
	output := activate(input, weights, porog)
	err = target - output // Ошибка
	for i := 0; i < len(weights); i++ {
		weights[i] += n * err * input[i] // Обновление весов
	}
	return weights, err
}

func main() {
	rand.Seed(time.Now().UnixNano()) // Для генерации случайных чисел

	// Входные данные и цели
	inputs := [][]float64{
		{1.0, 1.0},
		{1.0, -1.0},
		{-1.0, 1.0},
		{-1.0, -1.0},
	}
	targets := []float64{1.0, -1.0, -1.0, -1.0} // Цели для операции И

	// Инициализация весов
	weights := make([]float64, len(inputs[0]))
	for i := 0; i < len(weights); i++ {
		weights[i] = rand.Float64() - 0.5 // Случайные веса от -0.5 до 0.5
	}

	// Параметры обучения
	n := 0.1     // Скорость обучения
	porog := 0.0 // Порог активации
	maxEpochs := 100
	fmt.Println("Начальные веса:", weights)
```

Продолжение листинга Б

```
	// Обучение
	for epoch := 1; epoch <= maxEpochs; epoch++ {
		totalError := 0.0

		for i := 0; i < len(inputs); i++ {
			var err float64
			weights, err = delta(inputs[i], weights, targets[i], n,
porog)
			totalError += err * err // Сумма квадратов ошибок
		}

		fmt.Printf("Эпоха %d, Общая ошибка: %.2f, Веса: %v\n", epoch,
totalError, weights)

		if totalError == 0 {
			fmt.Printf("Обучение завершено на эпохе %d\n", epoch)
			break
		}
	}

	// Проверка предсказаний
	fmt.Println("\nРезультаты после обучения:")
	for i := 0; i < len(inputs); i++ {
		prediction := activate(inputs[i], weights, porog)
		fmt.Printf("Вход: %v, Предсказание: %.0f, Цель: %.0f\n",
inputs[i], prediction, targets[i])
	}
}
```

Приложение В

Код реализации обучения нейронной сети с использованием алгоритма обратного распространения ошибки на языке GoLang.

Листинг В – Код реализации обучения нейронной сети с использованием алгоритма обратного распространения ошибки

```
package main

import (
	"fmt"
	"math"
	"math/rand"
)

// Коэффициент обучения
var learningRate = 0.1

// Функция активации (sin)
func activationFunc(x float64) float64 {
	return math.Sin(x)
}

// Производная функции активации (cos)
func derivActivationFunc(x float64) float64 {
	return math.Cos(x)
}

// Прямой проход (forward pass)
func forward(inputs []float64, weights0 [][]float64, weights1 []float64)
(float64, []float64) {
	// Вычисление скрытого слоя
	inputs1 := []float64{
		activationFunc(inputs[0])*weights0[0][0] +
activationFunc(inputs[1])*weights0[1][0], // input10
		activationFunc(inputs[1])*weights0[0][1] +
activationFunc(inputs[0])*weights0[1][1], // input11
	}

	// Вычисление выхода
	input2 := activationFunc(inputs1[0])*weights1[0] +
activationFunc(inputs1[1])*weights1[1]

	// Возвращаем предсказанное значение и скрытые активации
	return activationFunc(input2), inputs1
}

// Обратный проход (backpropagation)
func backward(inputs, inputs1 []float64, output, target float64, weights0
[][]float64, weights1 []float64) {
	// Ошибка на выходе
	error := target - output
	// Вычисление сигма для выходного слоя
	sigma2 := derivActivationFunc(output) * error

	// Обновление весов выходного слоя
	weights1[0] += learningRate * sigma2 * derivActivationFunc(inputs1[0])
	weights1[1] += learningRate * sigma2 * derivActivationFunc(inputs1[1])
```

Продолжение листинга В

```
	// Вычисление сигма для скрытого слоя
	sigma1_0 := derivActivationFunc(inputs1[0]) * sigma2 * weights1[0]
	sigma1_1 := derivActivationFunc(inputs1[1]) * sigma2 * weights1[1]

	// Обновление весов скрытого слоя
	weights0[0][0] += learningRate * sigma1_0 *
derivActivationFunc(inputs[0])
	weights0[0][1] += learningRate * sigma1_1 *
derivActivationFunc(inputs[1])
	weights0[1][0] += learningRate * sigma1_0 *
derivActivationFunc(inputs[0])
	weights0[1][1] += learningRate * sigma1_1 *
derivActivationFunc(inputs[1])
}

func main() {
	// Инициализация весов
	weights0 := [][]float64{
		{rand.Float64(), rand.Float64()},
		{rand.Float64(), rand.Float64()},
	}
	weights1 := []float64{rand.Float64(), rand.Float64()}

	// Входные данные (обучающая выборка)
	inputs := [][]float64{
		{0, 0},
		{0, 1},
		{1, 0},
		{1, 1},
	}

	// Целевые значения
	targets := []float64{0, 1, 1, 1}

	// Результаты предсказания для каждого примера
	OUT := make([]float64, len(inputs))

	// Цикл обучения
	for epoch := 0; epoch < 100; epoch++ {
		for i, input := range inputs {
			fmt.Printf("Итерация: %d ----------\n", i)
			fmt.Printf("Веса ур. 0: %v\n", weights0)
			fmt.Printf("Веса ур. 1: %v\n", weights1)

			// Прямой проход: вычисление выхода сети
			output, inputs1 := forward(input, weights0, weights1)
			OUT[i] = output
			fmt.Printf("Результат: %v\n", OUT)

			// Обратный проход: обновление весов
			backward(input, inputs1, output, targets[i], weights0,
weights1)

			fmt.Printf("Цель: %v\n", targets)

			// Вывод обновлённых весов
			fmt.Printf("Новые веса ур. 0: %v\n", weights0)
			fmt.Printf("Новые веса ур. 1: %v\n", weights1)
		}
	}
}
```

Приложение Г

Код реализации обучения нейронной сети с гауссовой радиально-базисной функцией на языке GoLang.

Листинг Г – Код реализации обучения нейронной сети с гауссовой радиально-базисной функцией

```
package main

import (
	"fmt"
	"math"
	"math/rand"
)

// Коэффициент обучения
var learningRate = 0.1

// Радиально-базисная функция
func rbf(x, C, S float64) float64 {
	return math.Exp(-math.Pow(x-C, 2) / (2 * math.Pow(S, 2)))
}

// Производная радиально-базисной функции
func derivRBF(x, C, S float64) float64 {
	return (C - x) * (math.Exp(-math.Pow(x, 2)/(2*math.Pow(S,
2))+(C*x)/(S*S)) / (S * S))
}

// Функция активации (sin)
func activationFunc(x float64) float64 {
	return math.Sin(x)
}

// Производная функции активации (cos)
func derivActivationFunc(x float64) float64 {
	return math.Cos(x)
}

// Прямой проход (forward pass)
func forward(inputs []float64, weights0 [][]float64, weights1 []float64, C, S
float64) (float64, []float64) {
	// Вычисление скрытого слоя
	inputs1 := []float64{
		activationFunc(inputs[0])*weights0[0][0] +
activationFunc(inputs[1])*weights0[1][0], // input10
		activationFunc(inputs[1])*weights0[0][1] +
activationFunc(inputs[0])*weights0[1][1], // input11
	}

	// Вычисление выхода
	input2 := rbf(inputs1[0], C, S)*weights1[0] + rbf(inputs1[1], C,
S)*weights1[1]

	// Возвращаем предсказанное значение и скрытые активации
	return rbf(input2, C, S), inputs1
}

// Обратный проход (backpropagation)
```

Продолжение листинга Г

```
func backward(inputs, inputs1 []float64, output, target float64, weights0
[][]float64, weights1 []float64, C, S float64) {
	// Ошибка на выходе
	error := target - output

	// Вычисление сигма для выходного слоя
	sigma2 := derivRBF(output, C, S) * error

	// Обновление весов выходного слоя
	weights1[0] += learningRate * sigma2 * derivRBF(inputs1[0], C, S)
	weights1[1] += learningRate * sigma2 * derivRBF(inputs1[1], C, S)

	// Вычисление сигма для скрытого слоя
	sigma1_0 := derivRBF(inputs1[0], C, S) * sigma2 * weights1[0]
	sigma1_1 := derivRBF(inputs1[1], C, S) * sigma2 * weights1[1]

	// Обновление весов скрытого слоя
	weights0[0][0] += learningRate * sigma1_0 *
derivActivationFunc(inputs[0])
	weights0[0][1] += learningRate * sigma1_1 *
derivActivationFunc(inputs[1])
	weights0[1][0] += learningRate * sigma1_0 *
derivActivationFunc(inputs[0])
	weights0[1][1] += learningRate * sigma1_1 *
derivActivationFunc(inputs[1])
}

func main() {
	// Инициализация весов
	weights0 := [][]float64{
		{rand.Float64(), rand.Float64()},
		{rand.Float64(), rand.Float64()},
	}
	weights1 := []float64{rand.Float64(), rand.Float64()}

	// Входные данные (обучающая выборка)
	inputs := [][]float64{
		{0, 0},
		{0, 1},
		{1, 0},
		{1, 1},
	}

	// Целевые значения
	targets := []float64{0, 1, 1, 1}

	// Параметры радиально-базисной функции
	C := 1.0
	S := 1.0

	// Результаты предсказания для каждого примера
	OUT := make([]float64, len(inputs))

	// Цикл обучения
	for epoch := 0; epoch < 100; epoch++ {
		for i, input := range inputs {
			fmt.Printf("Итерация: %d ----------\n", i)
			fmt.Printf("Веса ур. 0: %v\n", weights0)
			fmt.Printf("Веса ур. 1: %v\n", weights1)

			// Прямой проход: вычисление выхода сети
			output, inputs1 := forward(input, weights0, weights1, C, S)
```

Продолжение листинга Г

```
                    OUT[i] = output
                    fmt.Printf("Результат: %v\n", OUT)

                    // Обратный проход: обновление весов
                    backward(input, inputs1, output, targets[i], weights0,
weights1, C, S)

                    fmt.Printf("Цель: %v\n", targets)

                    // Вывод обновлённых весов
                    fmt.Printf("Новые веса ур. 0: %v\n", weights0)
                    fmt.Printf("Новые веса ур. 1: %v\n", weights1)
                    fmt.Println("------------------------------")
                }
        }
}
```

Приложение Д

Код реализации обучения нейронной сети с использованием карты Кохонена на языке GoLang.

Листинг Д – Код реализации обучения нейронной сети с использованием карты Кохонена

```
package main

import (
	"fmt"
	"math"
	"math/rand"
)

// Параметры обучения
var learningRate = 0.1
var radius = 1.0

// Функция для вычисления евклидова расстояния
func euclideanDistance(v1, v2 []float64) float64 {
	sum := 0.0
	for i := range v1 {
		sum += math.Pow(v1[i]-v2[i], 2)
	}
	return math.Sqrt(sum)
}

// Обновление весов победителя и его соседей
func updateWeights(winnerIndex int, input []float64, weights [][]float64,
learningRate, radius float64) {
	for i := range weights {
		// Вычисляем расстояние между победителем и текущим нейроном
		distance := math.Abs(float64(i - winnerIndex))

		// Если нейрон находится в радиусе соседства, обновляем его веса
		if distance <= radius {
			for j := range weights[i] {
				weights[i][j] += learningRate * (input[j] -
weights[i][j])
			}
		}
	}
}

// Функция для нахождения победителя (нейрон с минимальным расстоянием)
func findWinner(input []float64, weights [][]float64) int {
	minIndex := 0
	minDist := euclideanDistance(input, weights[0])

	for i := 1; i < len(weights); i++ {
		dist := euclideanDistance(input, weights[i])
		if dist < minDist {
			minDist = dist
			minIndex = i
		}
	}
	return minIndex
}
```

Продолжение листинга Д

```
// Прямой проход (обучение сети)
func trainKohonen(inputs [][]float64, weights [][]float64) {
	for i, input := range inputs {
		// Находим победителя
		winner := findWinner(input, weights)

		fmt.Printf("Итерация: %d ----------\n", i)
		fmt.Printf("Входные данные: %v\n", input)
		fmt.Printf("Веса до обновления: %v\n", weights)
		fmt.Printf("Победитель: %d\n", winner)

		// Обновляем веса
		updateWeights(winner, input, weights, learningRate, radius)

		fmt.Printf("Веса после обновления: %v\n", weights)
		fmt.Println("-----------------------------")
	}
}

func main() {
	// Входные данные (обучающая выборка)
	inputs := [][]float64{
		{0, 0},
		{0, 1},
		{1, 0},
		{1, 1},
	}

	// Инициализация весов сети (например, 4 нейрона, каждый имеет по 2
веса)
	weights := [][]float64{
		{rand.Float64(), rand.Float64()},
		{rand.Float64(), rand.Float64()},
		{rand.Float64(), rand.Float64()},
		{rand.Float64(), rand.Float64()},
	}

	// Обучение сети
	for epoch := 0; epoch < 10; epoch++ {
		fmt.Println(epoch)
		trainKohonen(inputs, weights)
	}
}
```

Приложение Е

Код реализации обучения нейронной сети с использованием алгоритма встречного распространения на языке GoLang.

Листинг Е – Код реализации обучения нейронной сети с использованием алгоритма встречного распространения

```
package main

import (
	"fmt"
	"math"
	"math/rand"
)

// Параметры обучения
var learningRate = 0.1
var radius = 1.0

// Функция для вычисления евклидова расстояния
func euclideanDistance(v1, v2 []float64) float64 {
	sum := 0.0
	for i := range v1 {
		sum += math.Pow(v1[i]-v2[i], 2)
	}
	return math.Sqrt(sum)
}

// Обновление весов карты Кохонена
func updateKohonenWeights(winnerIndex int, input []float64, weights
[][]float64, learningRate, radius float64) {
	for i := range weights {
		// Вычисляем расстояние между победителем и текущим нейроном
		distance := math.Abs(float64(i - winnerIndex))

		// Если нейрон находится в радиусе соседства, обновляем его веса
		if distance <= radius {
			for j := range weights[i] {
				weights[i][j] += learningRate * (input[j] -
weights[i][j])
			}
		}
	}
}

// Функция для нахождения победителя на карте Кохонена
func findKohonenWinner(input []float64, kohonenWeights [][]float64) int {
	minIndex := 0
	minDist := euclideanDistance(input, kohonenWeights[0])

	for i := 1; i < len(kohonenWeights); i++ {
		dist := euclideanDistance(input, kohonenWeights[i])
		if dist < minDist {
			minDist = dist
			minIndex = i
		}
	}
	return minIndex
}
```

Продолжение листинга Е

```
// Обратное распространение ошибки для выходного слоя
func backpropagationError(output, target float64) float64 {
	return target - output
}

// Прямое распространение на выходной слой
func forwardOutput(winnerIndex int, outputWeights []float64) float64 {
	// Веса выходного слоя умножаются на победивший нейрон карты Кохонена
	return outputWeights[winnerIndex]
}

// Обновление весов выходного слоя
func updateOutputWeights(winnerIndex int, outputError float64, outputWeights
[]float64) {
	outputWeights[winnerIndex] += learningRate * outputError
}

func main() {
	// Входные данные
	inputs := [][]float64{
		{0, 0},
		{0, 1},
		{1, 0},
		{1, 1},
	}

	// Целевые значения для логической операции И
	targets := []float64{0, 0, 0, 1}

	// Инициализация весов карты Кохонена
	kohonenWeights := [][]float64{
		{rand.Float64(), rand.Float64()},
		{rand.Float64(), rand.Float64()},
		{rand.Float64(), rand.Float64()},
		{rand.Float64(), rand.Float64()},
	}

	// Инициализация весов выходного слоя
	outputWeights := []float64{rand.Float64(), rand.Float64(),
rand.Float64(), rand.Float64()}

	// Обучение сети встречного распространения
	for epoch := 0; epoch < 10; epoch++ {
		for i, input := range inputs {
			// Находим победителя на карте Кохонена
			winnerIndex := findKohonenWinner(input, kohonenWeights)
			fmt.Printf("Итерация: %d, Победитель: %d\n", i, winnerIndex)
			fmt.Printf("Веса до обновления: %v\n", kohonenWeights)

			// Обновляем веса карты Кохонена
			updateKohonenWeights(winnerIndex, input, kohonenWeights,
learningRate, radius)
			fmt.Printf("Веса после обновления: %v\n", kohonenWeights)
			// Прямое распространение на выходной слой
			output := forwardOutput(winnerIndex, outputWeights)
			fmt.Println("~~~~~~~~~~~~~~~~~~")
			fmt.Printf("Веса выходного слоя: %v\n", outputWeights)
			fmt.Printf("Предсказанный результат: %.4f\n", output)

			// Вычисление ошибки
			outputError := backpropagationError(output, targets[i])
			fmt.Printf("Ошибка: %.4f\n", outputError)
```

Продолжение листинга Е

```
                    // Обновляем веса выходного слоя
                    updateOutputWeights(winnerIndex, outputError, outputWeights)
                    fmt.Printf("Обновленные веса выходного слоя: %v\n",
outputWeights)

                    fmt.Println("------------------------------")
                }
        }
}
```

Приложение Ж

Код реализации обучения нейронной сети с использование модели DO-RNN на языке GoLang

Листинг Ж - Код реализации обучения нейронной сети с использование модели DO-RNN

```
package main

import (
	"fmt"
	"math"
	"math/rand"
)

// Параметры обучения
var learningRate = 0.01

// Функция активации (сигмоида)
func sigmoid(x float64) float64 {
	return 1 / (1 + math.Exp(-x))
}

// Производная сигмоиды
func sigmoidDerivative(x float64) float64 {
	return x * (1 - x)
}

// Прямой проход для рекуррентного слоя
func forwardPass(input, hiddenState []float64, inputWeights, recurrentWeights,
outputWeights [][]float64) ([]float64, []float64) {
	// Обновляем скрытое состояние
	newHiddenState := make([]float64, len(hiddenState))
	for i := range newHiddenState {
		sum := 0.0
		for j := range input {
			sum += input[j] * inputWeights[i][j]
		}
		for j := range hiddenState {
			sum += hiddenState[j] * recurrentWeights[i][j]
		}
		newHiddenState[i] = sigmoid(sum)
	}

	// Выход сети
	output := make([]float64, len(outputWeights))
	for i := range output {
		sum := 0.0
		for j := range newHiddenState {
			sum += newHiddenState[j] * outputWeights[i][j]
		}
		output[i] = sigmoid(sum)
	}
	return newHiddenState, output
}

// Обратное распространение ошибки и обновление весов
func backwardPass(input, hiddenState, output, target []float64, inputWeights,
recurrentWeights, outputWeights [][]float64) {
```

Продолжение листинга Ж

```
// Ошибка на выходе
	outputError := make([]float64, len(output))
	for i := range output {
		outputError[i] = (target[i] - output[i]) *
sigmoidDerivative(output[i])
	}

	// Ошибка скрытого состояния
	hiddenError := make([]float64, len(hiddenState))
	for i := range hiddenState {
		sum := 0.0
		for j := range outputError {
			sum += outputError[j] * outputWeights[j][i]
		}
		hiddenError[i] = sum * sigmoidDerivative(hiddenState[i])
	}

	// Обновляем веса выходного слоя
	for i := range outputWeights {
		for j := range outputWeights[i] {
			outputWeights[i][j] += learningRate * outputError[i] *
hiddenState[j]
		}
	}

	// Обновляем рекуррентные и входные веса
	for i := range inputWeights {
		for j := range inputWeights[i] {
			inputWeights[i][j] += learningRate * hiddenError[i] * input[j]
		}
		for j := range recurrentWeights[i] {
			recurrentWeights[i][j] += learningRate * hiddenError[i] *
hiddenState[j]
		}
	}
}

// Обучение сети
func trainRNN(inputs, targets [][]float64, inputWeights, recurrentWeights,
outputWeights [][]float64, epochs int) {
	hiddenState := make([]float64, len(inputWeights)) // начальное скрытое
состояние

	for epoch := 0; epoch < epochs; epoch++ {
		fmt.Printf("Итерация: %d\n", epoch)
		for i, input := range inputs {
			// Прямой проход
			hiddenState, output := forwardPass(input, hiddenState,
inputWeights, recurrentWeights, outputWeights)

			// Печать значений для отладки
			fmt.Printf("Вход: %v\n", input)
			fmt.Printf("Скрытое состояние: %v\n", hiddenState)
			fmt.Printf("Выход: %v\n", output)
			fmt.Printf("Цель: %v\n", targets[i])

			// Обратное распространение ошибки
			backwardPass(input, hiddenState, output, targets[i],
inputWeights, recurrentWeights, outputWeights)
		}
	}
}
```

Продолжение листинга Ж

```
func main() {
	// Входные данные и целевые значения
	inputs := [][]float64{
		{0, 0},
		{0, 1},
		{1, 0},
		{1, 1},
	}
	targets := [][]float64{
		{0},
		{0},
		{0},
		{1},
	}

	// Инициализация весов
	inputWeights := [][]float64{
		{rand.Float64(), rand.Float64()},
		{rand.Float64(), rand.Float64()},
	}
	recurrentWeights := [][]float64{
		{rand.Float64(), rand.Float64()},
		{rand.Float64(), rand.Float64()},
	}
	outputWeights := [][]float64{
		{rand.Float64(), rand.Float64()},
	}

	// Обучение сети
	trainRNN(inputs, targets, inputWeights, recurrentWeights, outputWeights,
100)
}
```

Приложение 3

Код реализации обучения нейронной сети с использованием модели R-CNN на языке GoLang

Листинг 3 - Код реализации обучения нейронной сети с использованием модели R-CNN

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from pathlib import Path
import opendatasets as od
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torch.utils.data import random_split
from torchvision.utils import make_grid
from torchvision import datasets, transforms
from torchvision.datasets import ImageFolder
import pytorch_lightning as pl
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from PIL import Image
import os


od.download("https://www.kaggle.com/datasets/puneet6060/intel-image-
classification")

transform = transforms.Compose([
    transforms.RandomRotation(10),
    transforms.RandomHorizontalFlip(),
    transforms.Resize((224, 224)),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[.485, .456, .406],
                         std=[.229, .224, .225])
])


class_names = sorted(os.listdir("./intel-image-classification/seg_train"))
print(class_names)

class DataModule(pl.LightningDataModule):
    def __init__(self, transform=transform, batch_size=32):
        super().__init__()
        self.root_dir = "./intel-image-classification/seg_train"  # Указываем
путь к данным
        self.transform = transform
        self.batch_size = batch_size


    def setup(self, stage=None):
        dataset = ImageFolder(root=self.root_dir, transform=self.transform)
        n_data = len(dataset)
        n_train = int(0.6 * n_data)  # 60% train
        n_val = int(0.2 * n_data)    # 20% val
        n_test = n_data - n_train - n_val  # 20% test
```

Продолжение листинга 3

```
        train_dataset, val_dataset, test_dataset = random_split(dataset,
[n_train, n_val, n_test])

        self.train_dataset = DataLoader(train_dataset,
batch_size=self.batch_size, shuffle=True)
        self.val_dataset = DataLoader(val_dataset, batch_size=self.batch_size)
        self.test_dataset = DataLoader(test_dataset, batch_size=self.batch_size)

    def train_dataloader(self):
        return self.train_dataset

    def val_dataloader(self):
        return self.val_dataset

    def test_dataloader(self):
        return self.test_dataset


class ConvolutionalNetwork(pl.LightningModule):

    def __init__(self):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=6, kernel_size=3, stride=1),
            nn.ReLU(), #активация
            nn.MaxPool2d(kernel_size=2, stride=2), #уменьшает размер признака в
2 раза шаг 2
            nn.Conv2d(in_channels=6, out_channels=16, kernel_size=3, stride=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
# in_channels=3: Входное изображение имеет 3 канала (RGB).
# out_channels=6: Слой извлекает 6 фильтров (карты признаков).
# kernel_size=3: Размер фильтра — 3x3.
# stride=1: Шаг фильтра — 1 пиксель

        self.classifier = nn.Sequential(
            nn.Linear(16 * 54 * 54, 120), #
            nn.ReLU(),
            nn.Linear(120, 84), #
            nn.ReLU(),
            nn.Linear(84, 20),
            nn.ReLU(),
            nn.Linear(20, len(class_names)),  # Количество классов = числу
моделей
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(-1, 16 * 54 * 54)
        x = self.classifier(x)
        return F.log_softmax(x, dim=1)

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.parameters(), lr=0.001)
        return optimizer

    def training_step(self, train_batch, batch_idx):
        X, y = train_batch
        y_hat = self(X)
        loss = F.cross_entropy(y_hat, y)
        acc = y_hat.argmax(dim=1).eq(y).float().mean()
```

Продолжение листинга 3

```
        self.log("train_loss", loss)
        self.log("train_acc", acc)
        return loss

    def validation_step(self, val_batch, batch_idx):
        X, y = val_batch
        y_hat = self(X)
        loss = F.cross_entropy(y_hat, y)
        acc = y_hat.argmax(dim=1).eq(y).float().mean()
        self.log("val_loss", loss)
        self.log("val_acc", acc)

    def test_step(self, test_batch, batch_idx):
        X, y = test_batch
        y_hat = self(X)
        loss = F.cross_entropy(y_hat, y)
        acc = y_hat.argmax(dim=1).eq(y).float().mean()
        self.log("test_loss", loss)
        self.log("test_acc", acc)

if __name__ == "__main__":
    datamodule = DataModule()
    datamodule.setup()
    model = ConvolutionalNetwork()
    trainer = pl.Trainer(max_epochs=10)
    trainer.fit(model, datamodule)

    datamodule.setup(stage="test")
    test_loader = datamodule.test_dataloader()
    trainer.test(dataloaders=test_loader)

    for images, labels in datamodule.train_dataloader():
        break
    im = make_grid(images, nrow=16)

    plt.figure(figsize=(12, 12))
    plt.imshow(np.transpose(im.numpy(), (1, 2, 0)))

    inv_normalize = transforms.Normalize(mean=[-0.485 / 0.229, -0.456 / 0.224, -
0.406 / 0.225],
                                         std=[1 / 0.229, 1 / 0.224, 1 / 0.225])
    im = inv_normalize(im)

    plt.figure(figsize=(12, 12))
    plt.imshow(np.transpose(im.numpy(), (1, 2, 0)))

    device = "cpu"
    model.eval()

    y_true = []
    y_pred = []
    with torch.no_grad():
        for test_data in datamodule.test_dataloader():
            test_images, test_labels = test_data[0].to(device),
test_data[1].to(device)

            pred = model(test_images).argmax(dim=1)
            y_true.extend(test_labels.cpu().numpy())
            y_pred.extend(pred.cpu().numpy())

    print(classification_report(y_true, y_pred, target_names=class_names,
digits=4))
```