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

Институт информационных технологий (ИТ)
Кафедра инструментального и прикладного программного обеспечения
(ИиППО)

ОТЧЕТ ПО ВСЕМ ЛАБОРАТОРНЫМ РАБОТАМ

по дисциплине

«Проектирование и обучение нейронных сетей»

Выполнил студент группы ИКБО-05-20 Мочалов А.Д.

Принял старший преподаватель Семёнов Р.Э.

Практические работы выполнены «__» _____ 2023 г.

(подпись студента)

«Зачтено» «__» _____ 2023 г.

(подпись студента)

Москва 2023

СОДЕРЖАНИЕ

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Правила Хебба

Правила Хебба являются одним из первых и наиболее известных правил обучения для нейронных сетей. Они были предложены Дональдом Хеббом в 1949 году и стали основой для развития множества других правил и алгоритмов обучения.

Основная идея правил Хебба состоит в том, что связи между нейронами усиливаются, если активация одного нейрона совпадает с активацией другого нейрона. То есть, если два нейрона активируются одновременно, их связь усиливается. Это правило основывается на концепции "клетки, которая активирует вместе с другой клеткой, усиливает связь с ней".

Формально, правило Хебба может быть описано следующим образом:

- Если активация пресинаптического нейрона совпадает с активацией постсинаптического нейрона, то вес связи между ними увеличивается.
- Если активация пресинаптического нейрона не совпадает с активацией постсинаптического нейрона, то вес связи между ними не изменяется.

Правила Хебба могут быть применены как для обучения нейронных сетей с прямыми связями (когда каждый нейрон связан со всеми нейронами следующего слоя), так и для обучения нейронных сетей с обратными связями (когда нейроны могут быть связаны с нейронами предыдущих слоев).

Однако, следует отметить, что правила Хебба могут привести к проблеме известной как "зацикливание обучения", когда веса связей продолжают увеличиваться, и нейроны становятся слишком чувствительными к определенным паттернам.

Основная проблема персептрона Хебба, которая ограничивает его применимость, называется "линейная разделимость". Персептрон Хебба может эффективно обучаться только в случае, когда классы данных могут быть линейно разделены гиперплоскостью.

Это означает, что персептрон Хебба не может обработать данные, которые не могут быть точно разделены прямой линией или гиперплоскостью. Если данные не являются линейно разделимыми, персептрон Хебба может не сойтись к оптимальному решению или может зациклиться в процессе обучения.

Дельта правило

Дельта-правило (Delta rule) является одним из основных алгоритмов обучения в нейронных сетях. Оно также известно как правило коррекции ошибки (error correction rule) или правило Видроу-Хоффа (Widrow-Hoff rule), по именам его создателей.

Дельта-правило используется для обновления весовых коэффициентов (весов) нейронов в нейронной сети с целью уменьшения ошибки (разницы между ожидаемым и фактическим выходом сети). Это достигается путем корректировки весовых коэффициентов на основе градиента функции потерь.

Процесс обучения с использованием дельта-правила состоит из следующих шагов:

1. Инициализация весов: В начале обучения веса нейронной сети инициализируются случайными значениями или некоторым другим способом.
2. Прямое распространение: Входные данные пропускаются через нейронную сеть, и каждый нейрон вычисляет свой выход на основе активационной функции и текущих весов.
3. Вычисление ошибки: Выход нейронной сети сравнивается с ожидаемым выходом, и вычисляется ошибка, которая представляет собой разницу между фактическим и ожидаемым выходом.
4. Обратное распространение: Ошибка распространяется обратно через сеть, и каждый нейрон вычисляет свою частную производную ошибки по своим входам (весам). Это позволяет определить, как веса влияют на ошибку.
5. Обновление весов: Веса нейронов обновляются в направлении, обратном градиенту ошибки. Обновление выполняется путем умножения градиента

на некоторый коэффициент обучения (learning rate) и вычитания этого значения из текущих весов.

6. Повторение: Шаги 2-5 повторяются для каждого обучающего примера в наборе данных до достижения некоторого критерия остановки, например, определенного количества эпох или сходимости ошибки.

Дельта-правило является одним из простых и широко используемых алгоритмов обучения в нейронных сетях. Оно позволяет сети корректировать свои веса на основе ошибки и, таким образом, улучшить свою способность к предсказанию и классификации данных.

Обратное распространение ошибки

Обратное распространение ошибки (Backpropagation) является одним из основных алгоритмов обучения в нейронных сетях. Оно позволяет определить, как веса нейронов влияют на ошибку сети и обновлять их соответствующим образом.

Процесс обучения с использованием обратного распространения ошибки состоит из следующих шагов:

1. Инициализация весов: В начале обучения веса нейронной сети инициализируются случайными значениями или некоторым другим способом.
2. Прямое распространение: Входные данные пропускаются через нейронную сеть, и каждый нейрон вычисляет свой выход на основе активационной функции и текущих весов.
3. Вычисление ошибки: Выход нейронной сети сравнивается с ожидаемым выходом, и вычисляется ошибка, которая представляет собой разницу между фактическим и ожидаемым выходом.
4. Обратное распространение: Ошибка распространяется обратно через сеть для вычисления градиента функции потерь по весам. Это выполняется с использованием алгоритма цепного правила (chain rule), который

позволяет вычислять частные производные ошибки по весам каждого нейрона.

5. Обновление весов: Веса нейронов обновляются в направлении, обратном градиенту ошибки. Обновление выполняется путем умножения градиента на некоторый коэффициент обучения (learning rate) и вычитания этого значения из текущих весов.
6. Повторение: Шаги 2-5 повторяются для каждого обучающего примера в наборе данных до достижения некоторого критерия остановки, например, определенного количества эпох или сходимости ошибки.

основные различия Дельта-правила от Обратного распространения ошибки:

Дельта-правило:

- Дельта-правило, также известное как правило Видроу-Хоффа, является одним из простейших алгоритмов обучения нейронных сетей.
- Оно используется для обновления весов только в одном слое нейронов, а именно выходном слое.
- Ошибка, вычисленная между фактическим и ожидаемым выходом, умножается на производную активационной функции выходного слоя и на коэффициент обучения. Это значение используется для обновления весов нейронов в выходном слое.
- Дельта-правило не учитывает влияние весов на ошибку внутренних слоев нейронной сети, поэтому оно не может эффективно обучать сети с несколькими скрытыми слоями.

Обратное распространение ошибки:

- Обратное распространение ошибки является более сложным алгоритмом обучения нейронных сетей и широко применяется для обновления весов во всех слоях нейронов.
- Оно использует алгоритм цепного правила (chain rule) для вычисления градиента ошибки по весам каждого нейрона в сети.

- Ошибка распространяется обратно через сеть, начиная с выходного слоя и заканчивая входным слоем. Каждый нейрон вычисляет свой градиент ошибки и использует его для обновления своих весов.
- Обратное распространение ошибки позволяет учитывать влияние весов на ошибку внутренних слоев, что позволяет эффективно обучать сети с несколькими скрытыми слоями.

В итоге, обратное распространение ошибки обеспечивает более глубокое обучение нейронных сетей, чем дельта-правило, и позволяет эффективно обновлять веса во всех слоях сети.

Обратное распространение ошибки позволяет нейронной сети адаптироваться к обучающим данным и улучшать свою способность к предсказанию и классификации. Оно основывается на идее, что ошибка сети может быть разделена на части, которые можно проследить обратно через сеть и использовать для обновления весов. Этот процесс повторяется до достижения оптимальных весов, при которых ошибка минимальна.

Радиально-базисные функции

Радиально-базисные функции (RBF) - это класс функций, которые широко используются в области нейронных сетей для аппроксимации и анализа данных. Они являются одним из методов моделирования нелинейных зависимостей между входными и выходными данными.

Основная идея RBF состоит в том, чтобы использовать радиально-симметричные функции, которые вычисляются на основе расстояния от входных данных до определенных центров. Эти центры представляют собой точки в пространстве признаков, которые определяются во время обучения нейронной сети.

Основные шаги в процессе работы с RBF:

1. Выбор центров: В начале процесса обучения нейронной сети RBF нужно выбрать центры функций. Центры могут быть выбраны случайным образом или с использованием алгоритмов кластеризации, таких как k-средних.
2. Вычисление ширины радиальной функции: После выбора центров, необходимо вычислить ширину радиальной функции для каждого центра. Ширина определяет, насколько далеко от центра будут влиять входные данные на выходную функцию. Различные подходы могут использоваться для определения ширины, например, можно использовать среднее расстояние от центра до ближайших соседей.
3. Вычисление выхода: После выбора центров и определения ширины, вычисляется выходная функция RBF. Это происходит путем вычисления расстояния между входными данными и центрами, а затем применения радиально-симметричной функции к этим расстояниям. Результаты суммируются и передаются через нелинейную функцию активации для получения окончательного выхода.
4. Обучение сети: Во время обучения сети RBF, веса центров и ширина радиальной функции оптимизируются с использованием методов оптимизации, таких как градиентный спуск или метод наименьших квадратов. Цель состоит в том, чтобы минимизировать ошибку между предсказанными и фактическими значениями.

Преимущества использования RBF включают:

- Способность аппроксимировать сложные нелинейные зависимости между входными и выходными данными.
- Быстрое обучение, особенно в сравнении с некоторыми другими типами нейронных сетей.
- Хорошая интерпретируемость результатов.

Недостатки RBF включают:

- Необходимость правильного выбора числа и расположения центров.
- Чувствительность к выбору ширины радиальной функции.
- Ограниченные возможности работы с данными высокой размерности.

В целом, радиально-базисные функции представляют собой мощный инструмент для аппроксимации и анализа данных и находят широкое применение в различных областях, таких как распознавание образов, прогнозирование временных рядов и контроль процессов.

Карты Кохонена

Карты Кохонена (или Self-Organizing Maps, SOM) - это один из алгоритмов машинного обучения, который используется для визуализации и анализа многомерных данных. Они были разработаны финским ученым Теуво Кохоненом в 1980-х годах.

Основная идея карт Кохонена заключается в создании двумерной сетки нейронов, которые группируются и организуются в соответствии с схожестью входных данных. Эти нейроны называются прототипами или векторами весов.

Процесс обучения карт Кохонена состоит из нескольких шагов:

1. Инициализация: Начальные значения весов каждого нейрона выбираются случайным образом или с использованием других методов инициализации.
2. Выбор образца: Из обучающего набора данных выбирается случайный образец.
3. Вектор наибольшего сходства: Для каждого нейрона вычисляется расстояние между его весами и входным образцом. Нейрон, чьи веса наиболее близки к входному образцу, считается победителем.
4. Обновление весов: Веса победившего нейрона и его ближайших соседей обновляются, чтобы быть ближе к входному образцу. Это позволяет картам Кохонена сжимать многомерные данные в двумерное пространство.
5. Повторение: Шаги 2-4 повторяются для каждого образца из обучающего набора данных несколько раз.

6. Завершение: После завершения обучения карт Кохонена можно использовать для кластеризации и визуализации данных. Каждый нейрон на карте представляет определенный кластер или группу данных.

Карты Кохонена обладают рядом преимуществ, таких как способность обрабатывать и визуализировать большие объемы данных, а также сохранение топологической структуры исходных данных. Они широко используются в разных областях, включая анализ данных, распознавание образов, компьютерное зрение и т.д.

Более подробную информацию о картах Кохонена и их применении можно найти в специализированной литературе и научных статьях.

Встречного распространения

Встречное распространение (Counter propagation) - это один из ключевых алгоритмов для обучения нейронных сетей. Он позволяет оптимизировать веса нейронов в сети, чтобы достичь желаемого выхода при заданных входных данных.

Основная идея встречного распространения заключается в передаче ошибки от выходного слоя к входному слою сети и обновлении весов нейронов на основе этой ошибки. Алгоритм состоит из нескольких шагов:

1. Прямой проход (forward pass): Входные данные подаются на входной слой нейронной сети, и сигналы проходят через скрытые слои до выходного слоя. Каждый нейрон вычисляет свой выход на основе взвешенной суммы своих входов и функции активации.
2. Вычисление ошибки: Выходные значения сравниваются с желаемыми значениями, и вычисляется ошибка. Эта ошибка может быть определена с помощью различных функций потерь, в зависимости от типа задачи (например, среднеквадратичная ошибка для задач регрессии или кросс-энтропия для задач классификации).
3. Обратное распространение ошибки (backward pass): Ошибка распространяется обратно через сеть, начиная с выходного слоя. Каждый

нейрон вычисляет свою локальную градиентную ошибку на основе производной функции активации и ошибки предыдущего слоя.

4. Обновление весов: Для каждого нейрона вычисляется градиент функции ошибки по его весам. Затем веса обновляются с использованием метода оптимизации, например, градиентного спуска или его вариаций. Это позволяет постепенно уменьшать ошибку и настраивать веса для лучшей производительности сети.
5. Повторение: Шаги 1-4 повторяются для каждого обучающего примера из обучающего набора данных. Обычно обучение происходит по эпохам, где каждая эпоха представляет собой полный проход по всем обучающим примерам.
6. Завершение: Обучение продолжается до достижения критерия остановки, например, определенного количества эпох или достижения достаточно малой ошибки.

Встречное распространение является эффективным методом обучения нейронных сетей, позволяющим сети адаптироваться к сложным задачам и извлекать полезные признаки из данных. Однако, он также может столкнуться с проблемой градиентного затухания или взрыва, когда градиенты становятся слишком малыми или большими. Различные модификации алгоритма, такие как использование скользящего среднего градиента или методы оптимизации с моментом, могут помочь преодолеть эти проблемы.

Рекуррентные сети

Рекуррентные нейронные сети (RNN) - это класс нейронных сетей, способных обрабатывать последовательности данных, где информация из предыдущих шагов времени сохраняется и передается в следующие шаги. Это делает RNN особенно подходящими для моделирования и обработки данных, имеющих последовательную структуру, таких как тексты, временные ряды или речь.

Основная идея RNN состоит в том, что каждый нейрон в сети имеет свое состояние (hidden state), которое обновляется на каждом шаге времени. Это состояние является своего рода памятью, которая запоминает информацию из предыдущих шагов.

Одна из самых популярных архитектур RNN - это "сеть Хопфилда" (Hopfield network), которая представляет собой полносвязную рекуррентную сеть без скрытых слоев. Она используется для решения задачи ассоциативной памяти и восстановления искаженных образов. Однако, более сложные задачи требуют более глубоких и сложных архитектур.

Одной из самых популярных и мощных архитектур RNN является долгая краткосрочная память (LSTM, Long Short-Term Memory). LSTM сети имеют дополнительные механизмы, которые позволяют им запоминать и забывать информацию на более длительные временные интервалы. Это позволяет им более эффективно работать с долгосрочными зависимостями в данных.

Еще одной распространенной архитектурой RNN является Gated Recurrent Unit (GRU). GRU сети имеют более простую структуру, но все еще обладают возможностью моделирования долгосрочных зависимостей.

Обучение RNN происходит с использованием алгоритма обратного распространения ошибки (backpropagation through time). Он аналогичен обратному распространению, но применяется для последовательностей данных. Обновление весов происходит на каждом шаге времени, чтобы учитывать информацию из предыдущих шагов.

Рекуррентные сети широко применяются в задачах обработки естественного языка (Natural Language Processing), машинного перевода, генерации текста, анализа временных рядов и других задач, где последовательная структура данных играет важную роль.

Конечная цель рекуррентных нейронных сетей (RNN) - обрабатывать и моделировать последовательности данных, сохраняя и передавая информацию из предыдущих шагов. Ниже представлен пошаговый процесс работы RNN:

1. Инициализация весов:

- На первом шаге необходимо инициализировать веса нейронной сети. Это может быть случайная инициализация или использование предобученных весов.

2. Подготовка входных данных:
    - Входные данные представляют собой последовательность значений, таких как слова в предложении или временные точки во временном ряде. Они могут быть представлены в виде векторов или матриц.
3. Проход по временным шагам:
    - RNN обрабатывает входные данные пошагово, начиная с первого временного шага и заканчивая последним.
    - На каждом временном шаге RNN принимает входные данные и текущее состояние (hidden state), которое обычно инициализируется в начале.
    - Входные данные и текущее состояние передаются через нейронные слои RNN, где происходит вычисление активации нейронов и передача информации в следующий временной шаг.
4. Обновление состояния:
    - На каждом временном шаге RNN обновляет свое состояние (hidden state), используя входные данные и предыдущее состояние.
    - Обновление состояния выполняется с помощью активационной функции, такой как гиперболический тангенс или сигмоида.
5. Передача информации:
    - Обновленное состояние передается в следующий временной шаг и используется вместе с текущими входными данными для обработки и передачи информации.
    - Таким образом, RNN сохраняет информацию из предыдущих шагов и передает ее в следующие шаги, что позволяет сети учитывать контекст и зависимости в последовательности данных.
6. Выходные данные:

- После обработки всех временных шагов RNN генерирует выходные данные, которые могут быть использованы для классификации, прогнозирования или генерации последовательностей.

7. Обучение сети:
    - Для обучения RNN используется алгоритм обратного распространения ошибки (backpropagation through time), который аналогичен обратному распространению, но применяется к последовательностям данных.
    - В процессе обучения веса нейронной сети обновляются с помощью градиентного спуска, чтобы минимизировать ошибку между предсказанными и ожидаемыми выходными данными.

Это общий обзор шагов работы RNN. Стоит отметить, что есть различные архитектуры RNN, такие как LSTM и GRU, которые имеют дополнительные механизмы для более эффективной работы с долгосрочными зависимостями в данных.

Сверточные сети

Сверточные сети являются одним из наиболее популярных типов нейронных сетей и широко используются в области компьютерного зрения. Одной из первых сверточных сетей была модель Когнитрон, предложенная Фукушимой в 1960-х годах.

Когнитрон был разработан для моделирования визуальной перцепции и имел иерархическую структуру. Он состоял из нескольких слоев, включая слои рецептивных полей, слои нормализации и слои субдискретизации. Сверточные операции, такие как свертка и пулинг, использовались для обработки входных данных.

Однако, Когнитрон имел некоторые ограничения, включая низкую устойчивость к вариациям входных данных и сложность обучения. В 1980-х годах был предложен улучшенный вариант Когнитрона, известный как Неокогнитрон.

Неокогнитрон внес ряд изменений в структуру Когнитрона, чтобы улучшить его производительность. Основным изменением было введение биологических механизмов, таких как боковое подавление и ингибиция. Это позволило сети стать более устойчивой к шуму и изменениям в данных.

Сверточные операции, такие как свертка и пулинг, остались важными компонентами Неокогнитрона. Свертка позволяет сети выделять локальные особенности и шаблоны в данных, а пулинг сокращает размерность и устраняет некоторые избыточности.

Сверточные сети, включая Когнитрон и Неокогнитрон, имеют множество применений в компьютерном зрении. Они могут использоваться для классификации изображений, детектирования объектов, сегментации изображений и многого другого.

Когнитрон

Процесс обучения на Когнитроне включает несколько шагов, которые позволяют сети извлекать иерархические признаки из входных данных. Ниже перечислены основные этапы обучения на Когнитроне:

1. Инициализация весов: Перед началом обучения веса в Когнитроне инициализируются случайными значениями. Веса отвечают за силу связей между нейронами и определяют, какие признаки будут извлекаться из данных.
2. Прямое распространение: В процессе прямого распространения входные данные проходят через слои Когнитрона. Каждый слой выполняет операции свертки, нормализации и субдискретизации для обработки данных. Результаты применения операций передаются на следующий слой до достижения выходного слоя.
3. Вычисление ошибки: После прямого распространения сравнивается результат работы сети с ожидаемыми значениями. На основе этого сравнения вычисляется ошибка, которая представляет собой разницу между прогнозируемыми и ожидаемыми значениями.

4. Обратное распространение ошибки: Чтобы обновить веса и улучшить производительность сети, необходимо распространить ошибку обратно через слои Когнитрона. Это делается с помощью алгоритма обратного распространения ошибки, который вычисляет градиент ошибки по весам и корректирует их соответственно.
5. Обновление весов: После вычисления градиента ошибки и обратного распространения, веса в Когнитроне обновляются с использованием оптимизационного алгоритма, такого как стохастический градиентный спуск. Алгоритм оптимизирует веса, направляя их в сторону минимума ошибки.
6. Итерации обучения: Процессы прямого и обратного распространения ошибки, а также обновления весов, повторяются на протяжении нескольких итераций обучения. Каждая итерация помогает улучшить производительность сети и снизить ошибку.

Процесс обучения на Когнитроне является итеративным и требует большого количества данных и вычислительных ресурсов. Он позволяет сети извлекать иерархические признаки из входных данных и улучшать свою производительность по мере обучения.

Неокогнитрон

Процесс обучения на Неокогнитроне схож с процессом обучения на Когнитроне, но имеет некоторые отличия. Ниже перечислены основные этапы обучения на Неокогнитроне:

1. Инициализация весов: Аналогично Когнитрону, веса в Неокогнитроне инициализируются случайными значениями перед началом обучения. Веса определяют силу связей между нейронами и влияют на выделение признаков из данных.
2. Прямое распространение: Входные данные проходят через слои Неокогнитрона в процессе прямого распространения. Каждый слой выполняет операции свертки, нормализации и субдискретизации для

обработки данных. Результаты передаются на следующий слой до достижения выходного слоя.

3. Вычисление ошибки: После прямого распространения сравнивается результат работы сети с ожидаемыми значениями, а затем вычисляется ошибка. Ошибка представляет собой разницу между прогнозируемыми и ожидаемыми значениями.
4. Обратное распространение ошибки: Для обновления весов и улучшения производительности сети, ошибку необходимо распространить обратно через слои Неокогнитрона. Алгоритм обратного распространения ошибки вычисляет градиент ошибки по весам и корректирует их соответствующим образом.
5. Обновление весов: После вычисления градиента ошибки и обратного распространения, веса в Неокогнитроне обновляются с использованием оптимизационного алгоритма, такого как стохастический градиентный спуск. Алгоритм оптимизирует веса, направляя их в сторону минимума ошибки.
6. Итерации обучения: Процессы прямого и обратного распространения ошибки, а также обновления весов, повторяются на протяжении нескольких итераций обучения. Каждая итерация помогает улучшить производительность сети и снизить ошибку.

Процесс обучения на Неокогнитроне требует большого объема данных и вычислительных ресурсов, а также может быть итеративным. Подобно Когнитрону, Неокогнитрон способен извлекать иерархические признаки из входных данных и улучшать свою производительность в процессе обучения.

Ниже перечислены характеристики каждого из этих слоев:

Слой Когнитрона:

- Структура: Когнитрон состоит из одного или нескольких сверточных слоев, нормализации и подвыборки (subsampling).
- Операции: Когнитрон выполняет операцию свертки, которая позволяет нейронной сети извлекать локальные признаки из входных данных.

- Локальная связность: Когнитрон имеет свойство локальной связности, при котором каждый нейрон связан только с небольшой областью входных данных.
- Иерархическая структура: Когнитрон образует иерархическую структуру слоев, где каждый последующий слой извлекает более абстрактные и сложные признаки.
- Инвариантность к трансляции: Когнитрон способен обнаруживать признаки независимо от их положения в изображении, что делает его инвариантным к трансляции.

Слой Неокогнитрона:

- Структура: Неокогнитрон состоит из сверточных слоев, нормализации, подвыборки и дополнительных слоев, таких как сети конкуренции и латерального подавления.
- Операции: Неокогнитрон выполняет операции свертки, нормализации и подавления для обработки входных данных.
- Входные связи: Неокогнитрон имеет более широкие входные связи, что позволяет ему обрабатывать информацию из большего количества источников.
- Сети конкуренции: Неокогнитрон использует сети конкуренции для выбора наиболее активных нейронов на каждом слое, что помогает улучшить распознавание и классификацию данных.
- Рекуррентные связи: Неокогнитрон может иметь рекуррентные связи, которые позволяют обратное распространение информации и улучшают способность сети к адаптации и обучению.

Трансформер

Трансформер (Transformer) - это архитектура нейронной сети, разработанная для обработки последовательностей данных, таких как текст или звук. Она была предложена в 2017 году и стала широко применяться в области обработки естественного языка.

Трансформер состоит из нескольких слоев, каждый из которых выполняет определенные функции. Основные слои в архитектуре трансформера включают:

1. Слой кодирования (Encoder Layer): Слой кодирования преобразует входные данные (например, слова в предложении) во внутреннее представление, называемое вектором признаков. Это достигается за счет применения механизма самовнимания (self-attention) и полносвязного слоя (feed-forward layer) на каждую позицию в последовательности.
2. Механизм самовнимания (Self-Attention): Это ключевой механизм трансформера. Он позволяет модели оценивать важность каждого слова в контексте других слов в предложении. Самовнимание позволяет модели сосредоточиться на релевантных информационных фрагментах.
3. Слой декодирования (Decoder Layer): Слой декодирования используется в архитектуре трансформера, предназначенной для задач генерации последовательностей, таких как машинный перевод. Он имеет дополнительный слой самовнимания, который помогает модели учиться использовать предыдущие прогнозы для генерации следующих элементов последовательности.
4. Механизм внимания (Attention): Это механизм, используемый для объединения информации из разных частей предложения или последовательности. В архитектуре трансформера используется внимание со знаком единицы (Scaled Dot-Product Attention), которое позволяет модели сосредоточиться на наиболее релевантных элементах данных.

Кроме этих основных слоев, трансформер может содержать дополнительные слои, такие как слои нормализации (Normalization) и слои

резидуальных связей (Residual Connections), которые помогают модели эффективнее обучаться и делать более стабильные прогнозы.

Значение каждого слоя в архитектуре трансформера связано с его способностью извлекать и агрегировать информацию из последовательности данных, а также с его способностью моделировать зависимости и контекст между различными элементами последовательности. В целом, трансформерная архитектура обеспечивает более параллельные и эффективные методы работы с последовательностями, чем традиционные рекуррентные нейронные сети.

Свёрточные графовые сети (Graph Convolutional Networks, GCNs) и сети-трансформеры (Transformer networks) являются двумя различными архитектурами глубоких нейронных сетей, которые используются для обработки и анализа графовых структур или последовательностей данных.

Графовые свёрточные сети (GCNs) разработаны специально для анализа графовых данных, где узлы графа представляют сущности, а ребра - связи между этими сущностями. GCNs работают путем агрегации информации из соседних узлов, применяя операции свёртки на уровне графа. Они позволяют эффективно использовать графовую структуру для принятия решений, и часто применяются в задачах, связанных с анализом социальных сетей, рекомендациями товаров и прогнозированием графовых свойств, таких как классификация узлов или предсказание связей.

Сети-трансформеры (Transformer networks) являются архитектурой, изначально представленной для обработки последовательностей данных, таких как тексты. Они глубоко рекурсивны, позволяют моделировать долгосрочные зависимости и добились больших успехов в задачах машинного перевода, генерации текста и других задачах обработки естественного языка. Сети-трансформеры работают за счёт взаимодействия и обмена информацией между различными "внимательностями" (self-attention) на уровне входных последовательностей, что позволяет им учить более сложные взаимодействия и зависимости между элементами последовательностей.

В трансформере информация проходит через несколько слоев, называемых "трансформерными блоками". Каждый трансформерный блок состоит из двух основных компонентов: многоканальный механизм внимания (multi-head attention) и позиционно-сетевая прямая связь (position-wise feed-forward network). Многоканальный механизм внимания обрабатывает и кодирует зависимости между элементами последовательности, позволяя моделировать контекст и учитывать токеновых на разных позициях. Позиционно-сетевая прямая связь - это слой с полносвязными нейронами, применяемый к каждой позиции независимо. Этот слой помогает модели вносить нелинейности в процесс обработки информации и приспособить ее к конкретным задачам. Повторяя эти блоки, модель может улавливать все более и более сложные зависимости во входных данных.

Название "трансформер" происходит от использования механизма внимания, который позволяет модели распределять внимание на различные элементы внутри последовательности данных, обрабатывая и преобразуя их. Этот механизм позволяет модели делать "трансформации" над данными в предсказательных задачах обработки последовательностей, и, следовательно, архитектура получила название "трансформер".

Графовые нейронные сети

Графовые нейронные сети (Graph Neural Networks, GNN) - это класс нейронных сетей, специально разработанных для работы с графами. Графы представляют собой совокупность вершин, соединенных ребрами, которые используются для моделирования сложных взаимосвязей и структур, таких как социальные сети, биологические сети, молекулярные структуры и т. д. Графовые нейронные сети позволяют эффективно обрабатывать и анализировать информацию на основе таких графовых структур.

Особенностью графовых нейронных сетей является их способность учитывать локальную информацию о вершинах и их связях при обработке

графов. В традиционных нейронных сетях каждый входной элемент рассматривается независимо от остальных, в то время как GNN учитывают взаимодействия между элементами графа. Это делает GNN особенно эффективными при анализе данных, где важны не только отдельные элементы, но и их контекст и взаимодействия.

Графовые нейронные сети могут быть применены в различных задачах, включая:

1. Классификацию вершин и графов: GNN могут классифицировать вершины или графы на основе их свойств и структуры. Например, они могут определить категорию социального профиля или классифицировать молекулы на основе их химических свойств.
2. Прогнозирование свойств графов: GNN могут предсказывать различные свойства графов, такие как цена акций, эмоциональная окраска графа или прогнозирование взаимодействий между молекулами в химической реакции.
3. Рекомендательные системы: GNN могут использоваться для создания рекомендательных систем, которые учитывают взаимосвязи и влияние пользователей друг на друга в социальных сетях или других графовых данных для предсказания персонализированных рекомендаций.
4. Анализ социальных сетей: GNN могут быть применены для анализа социальных сетей, идентификации влиятельных пользователей, выявления сообществ и анализа распространения информации в сети.
5. Компьютерное зрение на графах: GNN могут использоваться для анализа и обработки графической информации, такой как изображения или трехмерные модели, представленные в виде графов.

ПРАКТИЧЕСКАЯ ЧАСТЬ

Правила Хебба

```
# Импортируем библиотеку NumPy для работы с массивами и матрицами
import numpy as np

# Определяем функцию сигмоида для использования в нейронной сети
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Определяем класс NeuralNetwork, который представляет нейронную сеть
class NeuralNetwork:
    # Конструктор класса, инициализирующий параметры нейронной сети
    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size  # Количество нейронов во входном
слое
        self.hidden_size = hidden_size  # Количество нейронов в скрытом
слое
        self.output_size = output_size  # Количество нейронов в выходном
слое
        # Инициализируем веса между входным и скрытым слоями случайными
значениями
        self.weights_input_hidden = np.random.uniform(size=(self.in-
put_size, self.hidden_size))
        # Инициализируем веса между скрытым и выходным слоями случайными
значениями
        self.weights_hidden_output = np.random.uniform(size=(self.hid-
den_size, self.output_size))

    # Метод forward выполняет прямое распространение сигнала по сети
    def forward(self, input_data):
        # Вычисляем входной сигнал в скрытый слой, умножая входные дан-
ные на веса
        self.hidden_layer_input = np.dot(input_data, self.weights_in-
put_hidden)
        # Применяем функцию активации сигмоида к выходу скрытого слоя
        self.hidden_layer_output = sigmoid(self.hidden_layer_input)
        # Вычисляем входной сигнал в выходной слой, умножая выход скры-
того слоя на веса
        self.output_layer_input = np.dot(self.hidden_layer_output,
self.weights_hidden_output)
        # Применяем функцию активации сигмоида к выходу выходного слоя
        self.output_layer_output = sigmoid(self.output_layer_input)
        # Возвращаем выходной сигнал сети
        return self.output_layer_output
```

```
    # Метод hebbian_update обновляет веса сети с использованием правила
Хэбба
    def hebbian_update(self, input_example, learning_rate):
        # Обновляем веса между скрытым и выходным слоями
        self.weights_hidden_output += learning_rate * np.outer(self.hid-
den_layer_output, self.output_layer_output) * (1 - self.out-
put_layer_output)
        # Обновляем веса между входным и скрытым слоями
        self.weights_input_hidden += learning_rate * np.outer(input_ex-
ample, self.hidden_layer_output) * (1 - self.hidden_layer_output)

    # Метод train выполняет обучение сети на предоставленных данных
    def train(self, input_data, target, learning_rate, epochs):
        self.input_data = input_data  # Обучающие входные данные
        self.target = target  # Целевые значения
        # Выполняем обучение сети в течение заданного числа эпох
        for _ in range(epochs):
            # Проходим по всем обучающим примерам
            for i in range(len(input_data)):
                # Выполняем прямое распространение и обновление весов
                output = self.forward(input_data[i:i+1])
                self.hebbian_update(input_data[i:i+1], learning_rate)

# Пример обучающих данных и целевых значений (решение задачи OR)
input_data = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
target = np.array([[0], [1], [1], [1]])

# Создаем экземпляр нейронной сети с заданными параметрами
nn = NeuralNetwork(input_size=2, hidden_size=4, output_size=1)

# Обучаем сеть, передавая обучающие данные, целевые значения, скорость
обучения и число эпох
nn.train(input_data, target, learning_rate=0.1, epochs=10000)

# Тестируем сеть, выводим результаты
for i in range(len(input_data)):
    print(f"Input: {input_data[i]}, Output: {nn.forward(in-
put_data[i:i+1])}")
```

Вывод

```
Input: [0 0], Output: [[0.94933342]]
Input: [0 1], Output: [[0.99518491]]
Input: [1 0], Output: [[0.99539897]]
Input: [1 1], Output: [[0.99700881]]
```

Дельта правило

```
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))

class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.weights_input_hidden = np.random.uniform(size=(self.in-
put_size, self.hidden_size))
        self.weights_hidden_output = np.random.uniform(size=(self.hid-
den_size, self.output_size))

    def forward(self, input_data):
        self.hidden_layer_input = np.dot(input_data, self.weights_in-
put_hidden)
        self.hidden_layer_output = sigmoid(self.hidden_layer_input)
        self.output_layer_input = np.dot(self.hidden_layer_output,
self.weights_hidden_output)
        self.output_layer_output = sigmoid(self.output_layer_input)
        return self.output_layer_output

    def delta_update(self, learning_rate):
        output_error = self.target - self.output_layer_output
        output_delta = output_error * sigmoid_derivative(self.out-
put_layer_input)

        hidden_error = np.dot(output_delta, self.weights_hidden_out-
put.T)
        hidden_delta = hidden_error * sigmoid_derivative(self.hid-
den_layer_input)

        self.weights_hidden_output += learning_rate * np.outer(self.hid-
den_layer_output, output_delta)
        self.weights_input_hidden += learning_rate * np.outer(self.in-
put_data, hidden_delta)

    def train(self, input_data, target, learning_rate, epochs):
        self.input_data = input_data
        self.target = target
        for _ in range(epochs):
            for i in range(len(input_data)):
                self.input_data = input_data[i:i+1]
```

```
                self.target = target[i:i+1]
                output = self.forward(self.input_data)
                self.delta_update(learning_rate)

# Пример обучающих данных и целевых значений
input_data = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
target = np.array([[0], [1], [1], [0]])

# Создаем нейронную сеть
nn = NeuralNetwork(input_size=2, hidden_size=4, output_size=1)

# Обучаем сеть
nn.train(input_data, target, learning_rate=0.1, epochs=7000)

# Тестируем сеть
for i in range(len(input_data)):
    print(f"Input: {input_data[i]}, Output: {nn.forward(in-
put_data[i:i+1])}")
```

Вывод

```
Input: [0 0], Output: [[0.10604536]]
Input: [0 1], Output: [[0.91692062]]
Input: [1 0], Output: [[0.91044393]]
Input: [1 1], Output: [[0.07429269]]
```

Обратное распространение ошибки

```
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.weights_input_hidden = np.random.uniform(size=(self.in-
put_size, self.hidden_size))
        self.weights_hidden_output = np.random.uniform(size=(self.hid-
den_size, self.output_size))
```

```
    def forward(self, input_data):
        self.hidden_layer_input = np.dot(input_data, self.weights_in-
put_hidden)
        self.hidden_layer_output = sigmoid(self.hidden_layer_input)
        self.output_layer_input = np.dot(self.hidden_layer_output,
self.weights_hidden_output)
        self.output_layer_output = sigmoid(self.output_layer_input)
        return self.output_layer_output

    def backward(self, input_data, target, learning_rate):
        output_error = target - self.output_layer_output
        d_output = output_error * sigmoid_derivative(self.out-
put_layer_output)
        hidden_error = d_output.dot(self.weights_hidden_output.T)
        d_hidden = hidden_error * sigmoid_derivative(self.hid-
den_layer_output)

        self.weights_hidden_output += self.hidden_layer_out-
put.T.dot(d_output) * learning_rate
        self.weights_input_hidden += input_data.T.dot(d_hidden) * learn-
ing_rate

    def train(self, input_data, target, learning_rate, epochs):
        for _ in range(epochs):
            for i in range(len(input_data)):
                output = self.forward(input_data[i:i+1])  # Передаем
данные как двумерный массив
                self.backward(input_data[i:i+1], target[i:i+1],
learning_rate)  # Также передаем как двумерный массив

# Пример обучающих данных и целевых значений
input_data = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
target = np.array([[0], [1], [1], [0]])

# Создаем нейронную сеть
nn = NeuralNetwork(input_size=2, hidden_size=4, output_size=1)

# Обучаем сеть
nn.train(input_data, target, learning_rate=0.1, epochs=7000)

# Тестируем сеть
for i in range(len(input_data)):
    print(f"Input: {input_data[i]}, Output: {nn.forward(in-
put_data[i:i+1])}")
```

Вывод

```
Input: [0 0], Output: [[0.09147421]]
Input: [0 1], Output: [[0.9238417]]
Input: [1 0], Output: [[0.92247836]]
```

```
Input: [1 1], Output: [[0.0678134]]
```

Радиально-базисные функции

```
# Импортируем библиотеки
import numpy as np

# Определяем функцию для вычисления радиальной базисной функции Гаусса
def rbf_gaussian(x, center, sigma):
    return np.exp(-np.linalg.norm(x - center)**2 / (2 * sigma**2))

# Определяем класс для RBF нейронной сети
class RBFNeuralNetwork:
    def __init__(self, input_size, rbf_size, output_size, rbf_cen-
ters=None, sigma=None):
        # self.input_size - переменная, которая хранит размерность вход-
ных данных
        self.input_size = input_size

        # self.rbf_size - переменная, которая хранит количество радиаль-
ных базисных функций (RBF) в сети
        self.rbf_size = rbf_size

        # self.output_size - переменная, которая хранит размерность вы-
ходных данных
        self.output_size = output_size


        # Инициализируем веса между RBF и выходным слоем случайными зна-
чениями
        self.weights_rbf_output = np.random.uniform(size=(self.rbf_size,
self.output_size))

        # Если не предоставлены центры RBF, генерируем случайные центры
        self.rbf_centers = rbf_centers if rbf_centers is not None else
np.random.uniform(size=(self.rbf_size, self.input_size))

        # Если не предоставлен параметр sigma, устанавливаем значение по
умолчанию
        self.sigma = sigma if sigma is not None else 1.0

    # Метод для прямого прохода сети
    def forward(self, input_data):
        # Вычисляем выход RBF слоя
        self.rbf_layer_output = np.array([[rbf_gaussian(input_data[i],
self.rbf_centers[j], self.sigma)
                                           for j in
range(self.rbf_size)] for i in range(len(input_data))])
```

```
        # Вычисляем выход выходного слоя
        self.output_layer_output = np.dot(self.rbf_layer_output,
self.weights_rbf_output)
        return self.output_layer_output

    # Метод для обучения сети
    def train(self, input_data, target, learning_rate, epochs):
        for _ in range(epochs):
            # Вычисляем выход сети
            output = self.forward(input_data)

            # Вычисляем градиент для обновления весов
            delta_weights_rbf_output = learning_rate *
np.dot(self.rbf_layer_output.T, (target - output))

            # Обновляем веса
            self.weights_rbf_output += delta_weights_rbf_output

# Пример обучающих данных и целевых значений
input_data = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
target = np.array([[0], [1], [1], [0]])

# Создаем экземпляр нейронной сети
rbf_nn = RBFNeuralNetwork(input_size=2, rbf_size=90, output_size=1)

# Обучаем сеть
rbf_nn.train(input_data, target, learning_rate=0.01, epochs=3000)

# Устанавливаем порог для классификации
threshold = 0.5

# Тестируем сеть
for i in range(len(input_data)):
    raw_output = rbf_nn.forward(input_data[i:i+1])
    output = (raw_output >= threshold).astype(int)
    print(f"Input: {input_data[i]}, Raw Output: {raw_output},
Thresholded Output: {output}")
```

Вывод

```
Input: [0 0], Raw Output: [[0.06860478]], Thresholded Output: [[0]]
Input: [0 1], Raw Output: [[0.93155717]], Thresholded Output: [[1]]
Input: [1 0], Raw Output: [[0.93126996]], Thresholded Output: [[1]]
Input: [1 1], Raw Output: [[0.06838716]], Thresholded Output: [[0]]
```

Карты Кохонена

```
import numpy as np

class KohonenNeuralNetwork:
    def __init__(self, input_size, kohonen_size, output_size, ko-
honen_centers=None, learning_rate=0.01, epochs=5000):
        # Инициализация карты самоорганизующегося отображения (SOM) с
заданными параметрами
        self.input_size = input_size
        self.kohonen_size = kohonen_size
        self.output_size = output_size
        self.learning_rate = learning_rate
        self.epochs = epochs

        # Инициализация центров Кохонена случайными значениями, если не
предоставлены
        self.kohonen_centers = kohonen_centers if kohonen_centers is not
None else np.random.uniform(size=(self.kohonen_size, self.input_size))

        # Инициализация весов от слоя Кохонена к выходному слою случай-
ными значениями
        self.weights_kohonen_output = np.random.uniform(size=(self.ko-
honen_size, self.output_size))

    def _find_most_similar_kohonen_unit(self, x):
        # Находит индекс Кохоненской единицы с наименьшим евклидовым
расстоянием до входа 'x' тоесть выбирается победитель
        distances = np.array([np.linalg.norm(x - self.kohonen_cen-
ters[i]) for i in range(self.kohonen_size)])
        return np.argmin(distances)

    def forward(self, input_data):
        # Выполняет проход вперед через SOM
        kohonen_units = np.array([self._find_most_similar_ko-
honen_unit(input_data[i]) for i in range(len(input_data))])

        # Рассчитывает выход на основе активаций Кохоненских единиц
        output = np.dot(np.eye(self.kohonen_size)[kohonen_units],
self.weights_kohonen_output)

        # Применяет порог к выходу (например, бинарный порог на уровне
0,5)
        if output < 0.5:
            output = 0
        else:
            output = 1
        return output

    def train(self, input_data, target):
```

```
        # Обучение SOM
        for _ in range(self.epochs):
            for i in range(len(input_data)):
                most_similar_unit = self._find_most_similar_ko-
honen_unit(input_data[i])

                # Обновление весов Кохоненских центров на основе входных
данных и скорости обучения
                self.kohonen_centers[most_similar_unit] = self.ko-
honen_centers[most_similar_unit] + self.learning_rate * (input_data[i] -
self.kohonen_centers[most_similar_unit])

                # Обновление весов от слоя Кохонена к выходному слою на
основе цели и скорости обучения
                self.weights_kohonen_output[most_similar_unit] +=
self.learning_rate * (target[i] - self.weights_kohonen_output[most_simi-
lar_unit])

# Пример обучающих данных и целевых значений
input_data = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
target = np.array([[0], [1], [1], [0]])

# Создаем нейронную сеть с методом Кохонена
kohonen_nn = KohonenNeuralNetwork(input_size=2, kohonen_size=8, out-
put_size=1)

# Обучаем сеть
kohonen_nn.train(input_data, target)

# Тестируем сеть
for i in range(len(input_data)):
    print(f"Input: {input_data[i]}, Output: {kohonen_nn.forward(in-
put_data[i:i+1])}")
```

Вывод

```
Input: [0 0], Output: 0
Input: [0 1], Output: 1
Input: [1 0], Output: 1
Input: [1 1], Output: 0
```

Встречного распространения

Задача регрессии

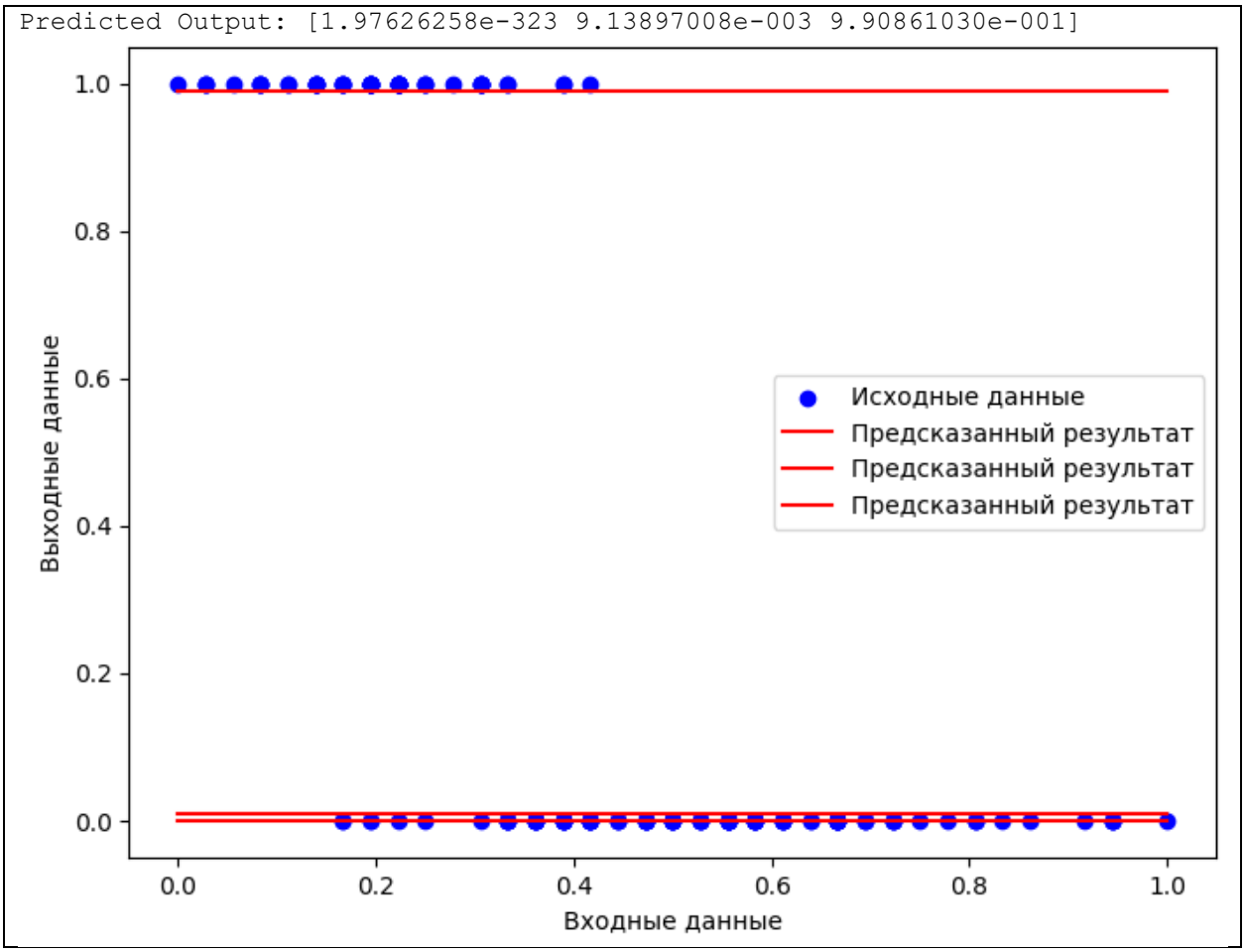
```
import numpy as np
import matplotlib.pyplot as plt

class RegressionCounterPropagationNetwork:
```

```
    def __init__(self, input_dim, hidden_dim, output_dim):
        self.input_weights = np.random.uniform(-1, 1, (input_dim, hid-
den_dim))
        self.hidden_weights = np.random.uniform(-1, 1, (hidden_dim, out-
put_dim))

    def train(self, data, target_outputs, lr, epochs):
        for _ in range(epochs):
            for input_vec, target in zip(data, target_outputs):
                hidden_activations = np.dot(input_vec, self.in-
put_weights)
                distances = np.sum((self.input_weights.T - hidden_acti-
vations[:, np.newaxis]) ** 2, axis=0)
                winner = np.argmin(distances)

                # Обновление весов для задачи регрессии
                self.hidden_weights[winner, :] += lr * (target -
self.hidden_weights[winner, :])

    def predict(self, input_vec):
        hidden_activations = np.dot(input_vec, self.input_weights)
        distances = np.sum((self.input_weights.T - hidden_activations[:,
np.newaxis]) ** 2, axis=0)
        winner = np.argmin(distances)
        return self.hidden_weights[winner, :]

# Пример использования для задачи регрессии
input_dim = 4  # Количество входных признаков
hidden_dim = 10  # Количество нейронов в скрытом слое
output_dim = 3  # Для задачи регрессии используем три выходных нейрона

# Создание сети и обучение на данных
cpn = RegressionCounterPropagationNetwork(input_dim, hidden_dim, out-
put_dim)
epochs = 1000
lr = 0.1
# data и target_outputs должны быть подготовлены для задачи регрессии
cpn.train(data, target_outputs, lr, epochs)

# Предсказание
input_data = np.array([0.5, 0.6, 0.7, 0.8])  # Пример входных данных для
предсказания
predicted_output = cpn.predict(input_data)
print("Predicted Output:", predicted_output)

# Создаем массив данных для графика
x_values = np.linspace(0, 1, 100)  # Генерируем 100 точек между 0 и 1
predicted_values = []

# Предсказываем результат для каждой точки
```

```
for x in x_values:
    input_data = np.array([x, 0.6, 0.7, 0.8])  # Фиксируем значения,
кроме первого
    predicted_output = cpn.predict(input_data)
    predicted_values.append(predicted_output)

predicted_values = np.array(predicted_values)  # Преобразуем в массив

# Построение графика
plt.figure(figsize=(8, 6))
plt.scatter(data[:, 0], target_outputs[:, 0], color='blue', la-
bel='Исходные данные')
plt.plot(x_values, predicted_values, color='red', label='Предсказанный
результат')
plt.xlabel('Входные данные')
plt.ylabel('Выходные данные')
plt.legend()
plt.show()
```

Вывод

```
Predicted Output: [1.97626258e-323 9.13897008e-003 9.90861030e-001]
```



Задача классификации

```
import numpy as np
from scipy.spatial import distance
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.preprocessing import MinMaxScaler

class CounterPropagationNetwork:
    def __init__(self, input_dim, hidden_dim, output_dim):
        self.input_weights = np.random.uniform(-1, 1, (input_dim, hid-
den_dim))
        self.hidden_weights = np.random.uniform(-1, 1, (hidden_dim, out-
put_dim))

    def train(self, data, target_outputs, lr):
        for input_vec, target in zip(data, target_outputs):
            # Competition stage
            hidden_activations = np.dot(input_vec, self.input_weights)
            distances = np.sum((self.input_weights.T - hidden_activa-
tions[:, np.newaxis]) ** 2, axis=0)
            winner = np.argmin(distances)
            # Learning stage
            self.input_weights[:, winner] += lr * (input_vec - self.in-
put_weights[:, winner])
            self.hidden_weights[winner, :] += lr * (target - self.hid-
den_weights[winner, :])

    def predict(self, input_vec):
        hidden_activations = np.dot(input_vec, self.input_weights)
        distances = np.sum((self.input_weights.T - hidden_activations[:,
np.newaxis]) ** 2, axis=0)
        winner = np.argmin(distances)
        return winner  # return the index of winning neuron

    def predict_output(self, input_vec):
        winner = self.predict(input_vec)
        return self.hidden_weights[winner, :]  # return output of win-
ning neuron

    def predict_class(self, input_vec):
        output = self.predict_output(input_vec)
        return np.argmax(output)

# Load and preprocess the Iris dataset
iris = load_iris()
data = iris.data
target = iris.target

# One-hot encode the target labels
target_outputs = np.zeros((target.shape[0], 3))
for i in range(target.shape[0]):
```

```
    target_outputs[i, target[i]] = 1

# Scale the data to [0, 1]
scaler = MinMaxScaler()
data = scaler.fit_transform(data)

# Create the CPN with 4 input dimensions, 10 neurons in the hidden
layer, and 3 output neurons
cpn = CounterPropagationNetwork(4, 10, 3)

# Train the CPN on the data
epochs = 10000
for i in range(epochs):
    cpn.train(data, target_outputs, 0.1)

# Test the network's performance on the training data and collect win-
ning neurons
winning_neurons_by_class = {0: [], 1: [], 2: []}
for i in range(len(data)):
    print("Input:", data[i], "Predicted Class:", cpn.pre-
dict_class(data[i]))
    winning_neuron_index = cpn.predict(data[i])
    winning_neurons_by_class[target[i]].append(cpn.input_weights[:, win-
ning_neuron_index])

# Plot the distribution of data in the hidden layer for each neuron
plt.figure(figsize=(10, 7))
colors = ['r', 'g', 'b']  # Red for class 0, Green for class 1, Blue for
Class 2
for class_idx, color in enumerate(colors):
    neuron_weights = np.array(winning_neurons_by_class[class_idx]).T  #
transpose to shape (2, N)
    plt.scatter(neuron_weights[0], neuron_weights[1], color=color, la-
bel=f'Class {class_idx}')

for class_idx, items in winning_neurons_by_class.items():
    print(f"Class {class_idx}: {len(items)} items")

# Move the legend outside the plot
plt.legend(loc='upper left', bbox_to_anchor=(1, 1))
plt.xlabel("Hidden Neuron 1 Weight")
plt.ylabel("Hidden Neuron 2 Weight")
plt.title("Distribution of Neuron Weights in Hidden Layer")
plt.grid(True)
plt.show()
```

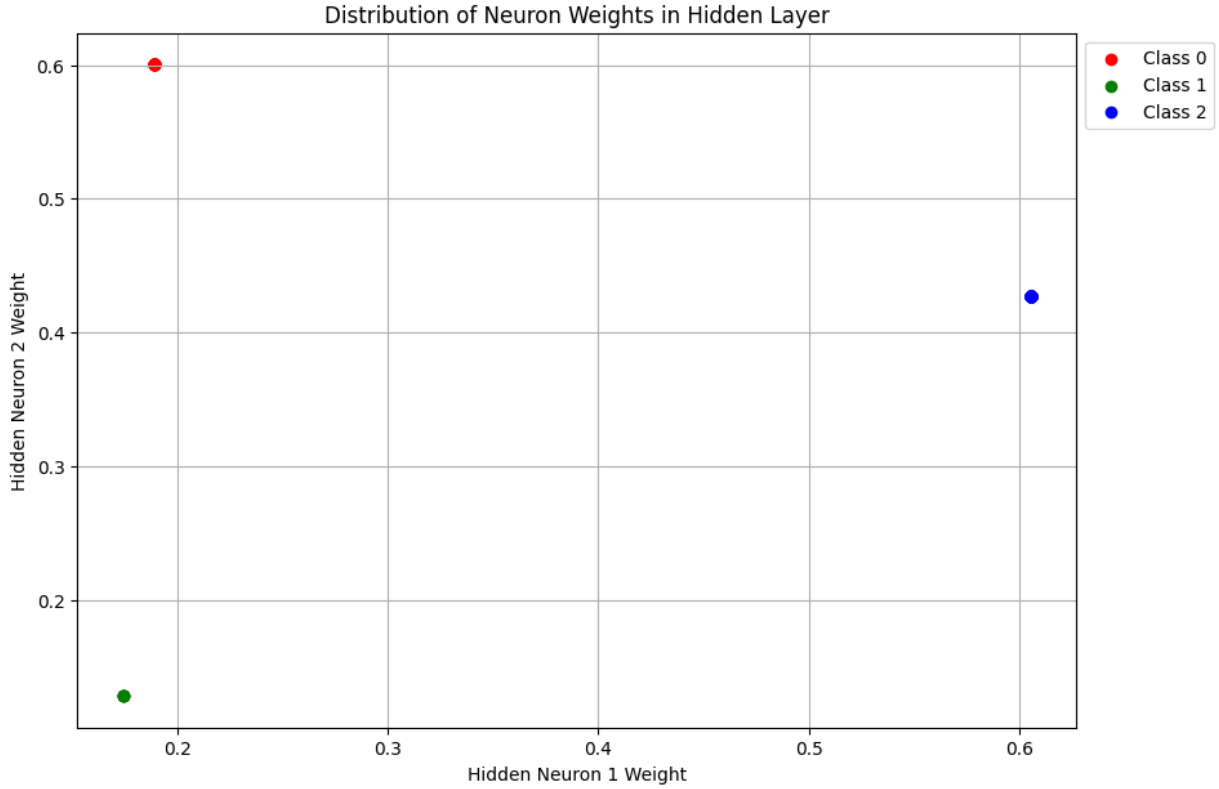
Вывод

```
Input: [0.22222222 0.625      0.06779661 0.04166667] Predicted Class: 0
```

```
Input: [0.16666667 0.41666667 0.06779661 0.04166667] Predicted Class: 0
Input: [0.11111111 0.5        0.05084746 0.04166667] Predicted Class: 0
Input: [0.08333333 0.45833333 0.08474576 0.04166667] Predicted Class: 0
Input: [0.19444444 0.66666667 0.06779661 0.04166667] Predicted Class: 0
Input: [0.30555556 0.79166667 0.11864407 0.125     ] Predicted Class: 0
Input: [0.08333333 0.58333333 0.06779661 0.08333333] Predicted Class: 0
Input: [0.19444444 0.58333333 0.08474576 0.04166667] Predicted Class: 0
Input: [0.02777778 0.375      0.06779661 0.04166667] Predicted Class: 0
Input: [0.16666667 0.45833333 0.08474576 0.        ] Predicted Class: 0
Input: [0.30555556 0.70833333 0.08474576 0.04166667] Predicted Class: 0
Input: [0.13888889 0.58333333 0.10169492 0.04166667] Predicted Class: 0
Input: [0.13888889 0.41666667 0.06779661 0.        ] Predicted Class: 0
Input: [0.         0.41666667 0.01694915 0.        ] Predicted Class: 0
Input: [0.41666667 0.83333333 0.03389831 0.04166667] Predicted Class: 0
Input: [0.38888889 1.         0.08474576 0.125     ] Predicted Class: 0
Input: [0.30555556 0.79166667 0.05084746 0.125     ] Predicted Class: 0
Input: [0.22222222 0.625      0.06779661 0.08333333] Predicted Class: 0
Input: [0.38888889 0.75       0.11864407 0.08333333] Predicted Class: 0
Input: [0.22222222 0.75       0.08474576 0.08333333] Predicted Class: 0
Input: [0.30555556 0.58333333 0.11864407 0.04166667] Predicted Class: 0
Input: [0.22222222 0.70833333 0.08474576 0.125     ] Predicted Class: 0
Input: [0.08333333 0.66666667 0.         0.04166667] Predicted Class: 0
Input: [0.22222222 0.54166667 0.11864407 0.16666667] Predicted Class: 0
Input: [0.13888889 0.58333333 0.15254237 0.04166667] Predicted Class: 0
Input: [0.19444444 0.41666667 0.10169492 0.04166667] Predicted Class: 0
Input: [0.19444444 0.58333333 0.10169492 0.125     ] Predicted Class: 0
Input: [0.25       0.625      0.08474576 0.04166667] Predicted Class: 0
Input: [0.25       0.58333333 0.06779661 0.04166667] Predicted Class: 0
Input: [0.11111111 0.5        0.10169492 0.04166667] Predicted Class: 0
Input: [0.13888889 0.45833333 0.10169492 0.04166667] Predicted Class: 0
Input: [0.30555556 0.58333333 0.08474576 0.125     ] Predicted Class: 0
Input: [0.25       0.875      0.08474576 0.        ] Predicted Class: 0
Input: [0.33333333 0.91666667 0.06779661 0.04166667] Predicted Class: 0
Input: [0.16666667 0.45833333 0.08474576 0.04166667] Predicted Class: 0
Input: [0.19444444 0.5        0.03389831 0.04166667] Predicted Class: 0
Input: [0.33333333 0.625      0.05084746 0.04166667] Predicted Class: 0
Input: [0.16666667 0.66666667 0.06779661 0.        ] Predicted Class: 0
Input: [0.02777778 0.41666667 0.05084746 0.04166667] Predicted Class: 0
Input: [0.22222222 0.58333333 0.08474576 0.04166667] Predicted Class: 0
Input: [0.19444444 0.625      0.05084746 0.08333333] Predicted Class: 0
Input: [0.05555556 0.125      0.05084746 0.08333333] Predicted Class: 1
Input: [0.02777778 0.5        0.05084746 0.04166667] Predicted Class: 0
Input: [0.19444444 0.625      0.10169492 0.20833333] Predicted Class: 0
Input: [0.22222222 0.75       0.15254237 0.125     ] Predicted Class: 0
Input: [0.13888889 0.41666667 0.06779661 0.08333333] Predicted Class: 0
Input: [0.22222222 0.75       0.10169492 0.04166667] Predicted Class: 0
Input: [0.08333333 0.5        0.06779661 0.04166667] Predicted Class: 0
Input: [0.27777778 0.70833333 0.08474576 0.04166667] Predicted Class: 0
Input: [0.19444444 0.54166667 0.06779661 0.04166667] Predicted Class: 0
Input: [0.75       0.5        0.62711864 0.54166667] Predicted Class: 2
Input: [0.58333333 0.5        0.59322034 0.58333333] Predicted Class: 2
Input: [0.72222222 0.45833333 0.66101695 0.58333333] Predicted Class: 2
Input: [0.33333333 0.125      0.50847458 0.5       ] Predicted Class: 2
Input: [0.61111111 0.33333333 0.61016949 0.58333333] Predicted Class: 2
Input: [0.38888889 0.33333333 0.59322034 0.5       ] Predicted Class: 2
Input: [0.55555556 0.54166667 0.62711864 0.625     ] Predicted Class: 2
Input: [0.16666667 0.16666667 0.38983051 0.375     ] Predicted Class: 1
Input: [0.63888889 0.375      0.61016949 0.5       ] Predicted Class: 2
Input: [0.25       0.29166667 0.49152542 0.54166667] Predicted Class: 2
Input: [0.19444444 0.         0.42372881 0.375     ] Predicted Class: 1
Input: [0.44444444 0.41666667 0.54237288 0.58333333] Predicted Class: 2
Input: [0.47222222 0.08333333 0.50847458 0.375     ] Predicted Class: 2
Input: [0.5        0.375      0.62711864 0.54166667] Predicted Class: 2
```

```
Input: [0.36111111 0.375      0.44067797 0.5       ] Predicted Class: 2
Input: [0.66666667 0.45833333 0.57627119 0.54166667] Predicted Class: 2
Input: [0.36111111 0.41666667 0.59322034 0.58333333] Predicted Class: 2
Input: [0.41666667 0.29166667 0.52542373 0.375     ] Predicted Class: 2
Input: [0.52777778 0.08333333 0.59322034 0.58333333] Predicted Class: 2
Input: [0.36111111 0.20833333 0.49152542 0.41666667] Predicted Class: 2
Input: [0.44444444 0.5        0.6440678  0.70833333] Predicted Class: 2
Input: [0.5        0.33333333 0.50847458 0.5       ] Predicted Class: 2
Input: [0.55555556 0.20833333 0.66101695 0.58333333] Predicted Class: 2
Input: [0.5        0.33333333 0.62711864 0.45833333] Predicted Class: 2
Input: [0.58333333 0.375      0.55932203 0.5       ] Predicted Class: 2
Input: [0.63888889 0.41666667 0.57627119 0.54166667] Predicted Class: 2
Input: [0.69444444 0.33333333 0.6440678  0.54166667] Predicted Class: 2
Input: [0.66666667 0.41666667 0.6779661  0.66666667] Predicted Class: 2
Input: [0.47222222 0.375      0.59322034 0.58333333] Predicted Class: 2
Input: [0.38888889 0.25       0.42372881 0.375     ] Predicted Class: 2
Input: [0.33333333 0.16666667 0.47457627 0.41666667] Predicted Class: 2
Input: [0.33333333 0.16666667 0.45762712 0.375     ] Predicted Class: 2
Input: [0.41666667 0.29166667 0.49152542 0.45833333] Predicted Class: 2
Input: [0.47222222 0.29166667 0.69491525 0.625     ] Predicted Class: 2
Input: [0.30555556 0.41666667 0.59322034 0.58333333] Predicted Class: 2
Input: [0.47222222 0.58333333 0.59322034 0.625     ] Predicted Class: 2
Input: [0.66666667 0.45833333 0.62711864 0.58333333] Predicted Class: 2
Input: [0.55555556 0.125      0.57627119 0.5       ] Predicted Class: 2
Input: [0.36111111 0.41666667 0.52542373 0.5       ] Predicted Class: 2
Input: [0.33333333 0.20833333 0.50847458 0.5       ] Predicted Class: 2
Input: [0.33333333 0.25       0.57627119 0.45833333] Predicted Class: 2
Input: [0.5        0.41666667 0.61016949 0.54166667] Predicted Class: 2
Input: [0.41666667 0.25       0.50847458 0.45833333] Predicted Class: 2
Input: [0.19444444 0.125      0.38983051 0.375     ] Predicted Class: 1
Input: [0.36111111 0.29166667 0.54237288 0.5       ] Predicted Class: 2
Input: [0.38888889 0.41666667 0.54237288 0.45833333] Predicted Class: 2
Input: [0.38888889 0.375      0.54237288 0.5       ] Predicted Class: 2
Input: [0.52777778 0.375      0.55932203 0.5       ] Predicted Class: 2
Input: [0.22222222 0.20833333 0.33898305 0.41666667] Predicted Class: 1
Input: [0.38888889 0.33333333 0.52542373 0.5       ] Predicted Class: 2
Input: [0.55555556 0.54166667 0.84745763 1.        ] Predicted Class: 2
Input: [0.41666667 0.29166667 0.69491525 0.75      ] Predicted Class: 2
Input: [0.77777778 0.41666667 0.83050847 0.83333333] Predicted Class: 2
Input: [0.55555556 0.375      0.77966102 0.70833333] Predicted Class: 2
Input: [0.61111111 0.41666667 0.81355932 0.875     ] Predicted Class: 2
Input: [0.91666667 0.41666667 0.94915254 0.83333333] Predicted Class: 2
Input: [0.16666667 0.20833333 0.59322034 0.66666667] Predicted Class: 2
Input: [0.83333333 0.375      0.89830508 0.70833333] Predicted Class: 2
Input: [0.66666667 0.20833333 0.81355932 0.70833333] Predicted Class: 2
Input: [0.80555556 0.66666667 0.86440678 1.        ] Predicted Class: 2
Input: [0.61111111 0.5        0.69491525 0.79166667] Predicted Class: 2
Input: [0.58333333 0.29166667 0.72881356 0.75      ] Predicted Class: 2
Input: [0.69444444 0.41666667 0.76271186 0.83333333] Predicted Class: 2
Input: [0.38888889 0.20833333 0.6779661  0.79166667] Predicted Class: 2
Input: [0.41666667 0.33333333 0.69491525 0.95833333] Predicted Class: 2
Input: [0.58333333 0.5        0.72881356 0.91666667] Predicted Class: 2
Input: [0.61111111 0.41666667 0.76271186 0.70833333] Predicted Class: 2
Input: [0.94444444 0.75       0.96610169 0.875     ] Predicted Class: 2
Input: [0.94444444 0.25       1.         0.91666667] Predicted Class: 2
Input: [0.47222222 0.08333333 0.6779661  0.58333333] Predicted Class: 2
Input: [0.72222222 0.5        0.79661017 0.91666667] Predicted Class: 2
Input: [0.36111111 0.33333333 0.66101695 0.79166667] Predicted Class: 2
Input: [0.94444444 0.33333333 0.96610169 0.79166667] Predicted Class: 2
Input: [0.55555556 0.29166667 0.66101695 0.70833333] Predicted Class: 2
Input: [0.66666667 0.54166667 0.79661017 0.83333333] Predicted Class: 2
Input: [0.80555556 0.5        0.84745763 0.70833333] Predicted Class: 2
Input: [0.52777778 0.33333333 0.6440678  0.70833333] Predicted Class: 2
```

```
Input: [0.5        0.41666667 0.66101695 0.70833333] Predicted Class: 2
Input: [0.58333333 0.33333333 0.77966102 0.83333333] Predicted Class: 2
Input: [0.80555556 0.41666667 0.81355932 0.625     ] Predicted Class: 2
Input: [0.86111111 0.33333333 0.86440678 0.75      ] Predicted Class: 2
Input: [1.         0.75       0.91525424 0.79166667] Predicted Class: 2
Input: [0.58333333 0.33333333 0.77966102 0.875     ] Predicted Class: 2
Input: [0.55555556 0.33333333 0.69491525 0.58333333] Predicted Class: 2
Input: [0.5        0.25       0.77966102 0.54166667] Predicted Class: 2
Input: [0.94444444 0.41666667 0.86440678 0.91666667] Predicted Class: 2
Input: [0.55555556 0.58333333 0.77966102 0.95833333] Predicted Class: 2
Input: [0.58333333 0.45833333 0.76271186 0.70833333] Predicted Class: 2
Input: [0.47222222 0.41666667 0.6440678  0.70833333] Predicted Class: 2
Input: [0.72222222 0.45833333 0.74576271 0.83333333] Predicted Class: 2
Input: [0.66666667 0.45833333 0.77966102 0.95833333] Predicted Class: 2
Input: [0.72222222 0.45833333 0.69491525 0.91666667] Predicted Class: 2
Input: [0.41666667 0.29166667 0.69491525 0.75      ] Predicted Class: 2
Input: [0.69444444 0.5        0.83050847 0.91666667] Predicted Class: 2
Input: [0.66666667 0.54166667 0.79661017 1.        ] Predicted Class: 2
Input: [0.66666667 0.41666667 0.71186441 0.91666667] Predicted Class: 2
Input: [0.55555556 0.20833333 0.6779661  0.75      ] Predicted Class: 2
Input: [0.61111111 0.41666667 0.71186441 0.79166667] Predicted Class: 2
Input: [0.52777778 0.58333333 0.74576271 0.91666667] Predicted Class: 2
Input: [0.44444444 0.41666667 0.69491525 0.70833333] Predicted Class: 2
Class 0: 50 items
Class 1: 50 items
Class 2: 50 items
```

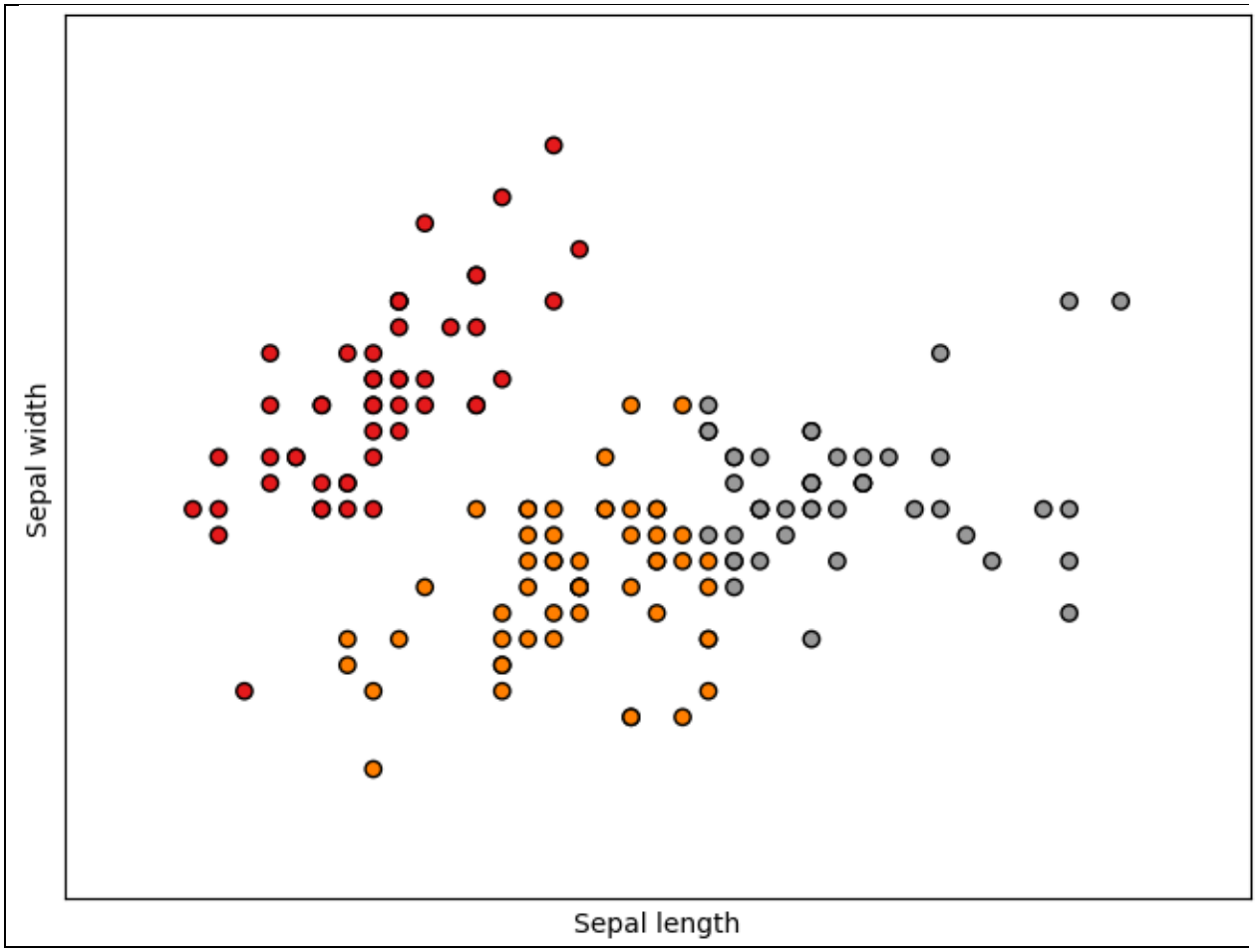


Задача кластеризации

```python
import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets
from sklearn.decomposition import PCA
import tensorflow as tf
```

```
from tensorflow import keras

# Загрузка данных
iris = datasets.load_iris()
X = iris.data[:, :2]
y = iris.target

# Создание и обучение нейронной сети
model = keras.Sequential([
    keras.layers.Dense(128, activation='relu', input_shape=(2,)),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dense(3, activation='softmax')
])
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
model.fit(X, y, epochs=1000, batch_size=50, verbose=0)

# Предсказание меток классов
predictions = model.predict(X)
predicted_classes = np.argmax(predictions, axis=1)

# Построение графика
x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5

plt.figure(2, figsize=(8, 6))
plt.clf()
plt.scatter(X[:, 0], X[:, 1], c=predicted_classes, cmap=plt.cm.Set1,
edgecolor="k")
plt.xlabel("Sepal length")
plt.ylabel("Sepal width")
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.xticks(())
plt.yticks(())

plt.show()
```

Вывод



Рекуррентные сети

```
import numpy as np
from keras.models import Sequential
from keras.layers import LSTM, Dense

# Инициализация входных данных XOR
inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
# Инициализация ожидаемого вывода для каждого входа XOR
expected_output = np.array([[0], [1], [1], [0]])

# Меняем форму входных данных в формат [примеры, временные шаги, при-
знаки] для LSTM
inputs = inputs.reshape((inputs.shape[0], inputs.shape[1], 1))

# Начинаем определение модели LSTM
model = Sequential()

# Добавьте слой LSTM к модели с 2 узлами (или нейронами)
# Он использует функцию активации 'tanh'
```

```
# и ожидает, что входные данные будут в формате [временные шаги, при-
знаки]
model.add(LSTM(units=2, activation='tanh', input_shape=(2, 1)))

# Слой Dense - это обычный полносвязный слой нейронной сети.
# Он использует функцию активации 'sigmoid' и имеет 1 узел (или нейрон)
model.add(Dense(1, activation='sigmoid'))

# Модель использует среднеквадратическую ошибку в качестве функции по-
терь, оптимизатор adam
# и во время обучения оценит модель по 'accuracy'.
model.compile(loss='mean_squared_error', optimizer='adam', metrics=['ac-
curacy'])

# Количество раз, которое мы прогоняем весь набор данных
epochs = 1000
# Эпохи, на которых мы хотим оценить модель
milestone_epochs = [0, 250, 500, 750, epochs - 1]

# Проходим через эпохи в цикле
for i in range(epochs):
    # Приспособить модель для одной эпохи
    model.fit(inputs, expected_output, epochs=1, batch_size=4, ver-
bose=0)

    # На наших предустановленных вехах вычислить и вывести точность
    if i in milestone_epochs:
        _, accuracy = model.evaluate(inputs, expected_output, verbose=0)
        print(f"Epoch: {i + 1}, Accuracy: {accuracy * 100:.2f}%")

# После того как модель обучена, используем её для предсказания на вход-
ных данных
predicted_output = model.predict(inputs)

# Выводим результаты LSTM после 1,000 эпох
# Функция numpy round используется для округления до ближайшего целого
print("\nOutput from the LSTM after 1,000 epochs: ", np.round(pre-
dicted_output))
```

Вывод

```
Epoch: 1, Accuracy: 25.00%
Epoch: 251, Accuracy: 75.00%
Epoch: 501, Accuracy: 100.00%
Epoch: 751, Accuracy: 100.00%
Epoch: 1000, Accuracy: 100.00%
1/1 [==============================] - 1s 1s/step

Output from the LSTM after 1,000 epochs:  [[0.]
 [1.]
 [1.]
 [0.]]
```

Сверточные сети

```
# импортировать необходимые библиотеки
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, Flatten, Dense, Activation
from tensorflow.keras.callbacks import LambdaCallback

# создаём данные - входные значения и их ожидаемые выходные значения для
OR
inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
expected_output = np.array([[0], [1], [1], [1]])

# меняем форму входных данных для CNN
# каждый образец теперь будет двумерным тензором вместо одномерного век-
тора
inputs = inputs.reshape((inputs.shape[0], inputs.shape[1], 1))

# создаём модель CNN
model = Sequential()
model.add(Conv1D(filters=2, kernel_size=2, activation='relu', in-
put_shape=(2, 1)))
model.add(Flatten())
model.add(Dense(1))
model.add(Activation('sigmoid'))

epochs = 2000
# компилируем модель, определяя тип потерь и метод оптимизации
model.compile(loss='mean_squared_error', optimizer='adam', metrics=['ac-
curacy'])
# Обучение модели
milestone_epochs = [250, 500, 750, 1000, 1250, 1500, 1750, 1999]  # мо-
менты тренировки, на которых будем проверять точность
for i in range(epochs):
    model.fit(inputs, expected_output, epochs=1, batch_size=4, ver-
bose=0)  # тренируем модель на данных

    # проверяет точность на каждой вехе (milestone) и печатает резуль-
таты
    if i in milestone_epochs:
        _, accuracy = model.evaluate(inputs, expected_output, verbose=0)
        print(f"Epoch: {i + 1}, Accuracy: {accuracy * 100:.2f}%")


# предсказания модели
```

```
predicted_output = model.predict(inputs)
print("\nOutput from the CNN after 2,000 epochs: ", np.round(pre-
dicted_output))
```

Вывод

```
Epoch: 251, Accuracy: 75.00%
Epoch: 501, Accuracy: 75.00%
Epoch: 751, Accuracy: 75.00%
Epoch: 1001, Accuracy: 75.00%
Epoch: 1251, Accuracy: 75.00%
Epoch: 1501, Accuracy: 75.00%
Epoch: 1751, Accuracy: 100.00%
Epoch: 2000, Accuracy: 100.00%
1/1 [==============================] - 0s 69ms/step

Output from the CNN after 2,000 epochs:  [[0.]
 [1.]
 [1.]
 [1.]]
```

Трансформер

```
import numpy as np
from tensorflow.keras import layers
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam

# Подготовка данных для XOR
inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])  # входные данные XOR
expected_output = np.array([[0], [1], [1], [0]])  # ожидаемый выход XOR

# Установка размерности вложения
embed_dim = 64

# Начало построения модели
input_layer = layers.Input(shape=(2,))
x = layers.Dense(embed_dim, activation='relu')(input_layer)  # первый слой прямого
распространения
x = layers.Dense(embed_dim, activation='relu')(x)  # второй слой прямого распростра-
нения вместо блока трансформера
x = layers.Dense(embed_dim, activation='relu')(x)  # добавляем ещё один слой для
увеличения "глубины" нашей модели
output_layer = layers.Dense(1, activation='sigmoid')(x) # выходной слой с sigmoid
вместо softmax

# Финализация модели
model = Model(inputs=input_layer, outputs=output_layer)

# Компиляция модели
```

```
model.compile(loss='binary_crossentropy', optimizer=Adam(), metrics=['accuracy'])

# Обучение модели
epochs = 100
milestone_epochs = [0, 25, 50, 75, 99]
for i in range(epochs):
    model.fit(inputs, expected_output, epochs=1, batch_size=4, verbose=0)  # обуче-
ние модели

    # Расчет и вывод точности на каждом этапе
    if i in milestone_epochs:
        _, accuracy = model.evaluate(inputs, expected_output, verbose=0)
        print(f"Эпоха: {i + 1}, Точность: {accuracy * 100:.2f}%")
# Выполнение обученной модели на данных XOR, результат округляется до ближайшего це-
лого числа
predicted_output = model.predict(inputs)
print("\nВыходные данные от модели после 100 эпох:\n", np.round(predicted_output))
```

Вывод

```
Эпоха: 1, Точность: 50.00%
Эпоха: 26, Точность: 100.00%
Эпоха: 51, Точность: 100.00%
Эпоха: 76, Точность: 100.00%
Эпоха: 100, Точность: 100.00%
1/1 [==============================] - 0s 92ms/step

Выходные данные от модели после 100 эпох:
 [[0.]
 [1.]
 [1.]
 [0.]]
```

Графовые нейронные сети

```
import numpy as np  # Импортируем библиотеку numpy для работы с массивами
import tensorflow as tf  # Импортируем tensorflow для создания и обучения модели
from tensorflow.keras import layers  # Импортируем layers для создания слоев модели
from tensorflow.keras.models import Model  # Импортируем Model для создания модели
from spektral.layers import GCNConv, GlobalAvgPool  # Импортируем слои GCNConv и GlobalAvgPool из spektral
from spektral.data import Dataset, Graph  # Импортируем Dataset и Graph из spektral
from spektral.data.loaders import BatchLoader  # Импортируем BatchLoader из spektral

# Определяем данные XOR (графы)
node_features = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]])  # Вводим признаки узлов
```

```
# Создаем список графов
graph_list = [
    Graph(x=node_features, a=np.array([[1, 1], [1, 0]]), y=[0]),  # Создаем первый граф
    Graph(x=node_features, a=np.array([[1, 0], [0, 1]]), y=[1]),  # Создаем второй граф
    Graph(x=node_features, a=np.array([[0, 1], [1, 1]]), y=[1]),  # Создаем третий граф
    Graph(x=node_features, a=np.array([[0, 0], [0, 0]]), y=[0]),  # Создаем четвертый граф
]

# Создаем пользовательский набор данных
class CustomDataset(Dataset):
    def __init__(self, graphs, **kwargs):  # Инициализируем класс
        self.graphs = graphs  # Сохраняем графы в атрибуте класса
        super().__init__(**kwargs)  # Вызываем инициализатор родительского класса

    def read(self):  # Реализуем метод чтения
        return self.graphs  # Возвращаем графы

    def __getitem__(self, idx):  # Реализуем индексацию
        return self.graphs[idx]  # Возвращаем граф по индексу

    def __len__(self):  # Реализуем вычисление длины датасета
        return len(self.graphs)  # Возвращаем длину списка графов

dataset = CustomDataset(graph_list)  # Создаем экземпляр датасета
loader = BatchLoader(dataset, batch_size=4)  # Создаем загрузчик батчей

n_features = node_features.shape[1]  # Определяем количество признаков
embed_dim = 64  # Определяем размерность вложения

# Создаем модель графовой нейронной сети
F = n_features  # Количество признаков
A_in = tf.keras.Input(shape=(None, None), sparse=True, name="A")  # Вход для матрицы смежности
X_in = tf.keras.Input(shape=(None, F), name="X")  # Вход для признаков узлов
x = GCNConv(embed_dim, activation='relu')([X_in, A_in])  # Первый слой GCN
x = GCNConv(embed_dim, activation='relu')([x, A_in])  # Второй слой GCN
x = GlobalAvgPool()(x)  # Усреднение по узлам
output_layer = layers.Dense(1, activation='sigmoid')(x)  # Выходной слой

model = Model([X_in, A_in], output_layer)  # Создаем модель
model.compile(loss='binary_crossentropy', optimizer=tf.keras.optimizers.Adam(), metrics=['accuracy'])  #
Компилируем модель

# Обучаем модель GNN
epochs = 350  # Количество эпох
milestone_epochs = [0, 50, 100, 150, 200, 250, 300, 349]  # Создаем список мильных камней эпох

for i in range(epochs):  # Итерируем по эпохам
    model.fit(loader.load(), steps_per_epoch=loader.steps_per_epoch, verbose=0)  # Обучаем модель
    if i in milestone_epochs:  # Если текущая эпоха в списке мильных камней
```

```
        _, accuracy = model.evaluate(loader.load(), steps=loader.steps_per_epoch, verbose=0)  # Вычисляем
точность
        print(f"Epoch: {i + 1}, Accuracy: {accuracy * 100:.2f}%")  # Печатаем номер эпохи и точность
```

Вывод

```
Epoch: 1, Accuracy: 25.00%
Epoch: 51, Accuracy: 75.00%
Epoch: 101, Accuracy: 75.00%
Epoch: 151, Accuracy: 75.00%
Epoch: 201, Accuracy: 75.00%
Epoch: 251, Accuracy: 75.00%
Epoch: 301, Accuracy: 100.00%
Epoch: 350, Accuracy: 100.00%
```

ВЫВОД

Мы изучили 10 видов построения нейронных сетей и научились понимать принцип их работы.