**Вопросы по теоретической части**

**Основы глубокого обучения**

1. Модель перцептрона. Проблема линейно неразделимых множеств и ее решение. Логика построения многослойных ИНС.

Изображение выглядит как текст, снимок экрана, линия, диаграмма

Автоматически созданное описаниеИзображение выглядит как текст, снимок экрана, диаграмма, круг

Автоматически созданное описание

Проблема: При помощи однослойного перцептрона невозможно решить задачу линейного разделения XOR.

Изображение выглядит как текст, снимок экрана, диаграмма, линия

Автоматически созданное описание

Решение – построение многослойных ИНС. Проблема состояла в том, что тогда не умели обучать двухслойные перцептроны. Тогда не был изобретен подход обратного распространения ошибки. Вся проблема решается с помощью обратного распространения ошибки.

Изображение выглядит как текст, снимок экрана, диаграмма, линия

Автоматически созданное описание

1. Функции активации. Требования к функциям активации Популярные функции активации.

В искусственных нейронных сетях функция активации нейрона определяет выходной сигнал, который определяется входным сигналом или набором входных сигналов. В качестве аргумента принимает сигнал, получаемый на выходе входного сумматора.

Основные требования к ФА:

● функция должна быть монотонной (обычно монотонно не убывающая)

● иметь первую производную почти всюду (необходимо для обратного распространения ошибки при обучении нейронной сети)

Примеры ФА:

● Сигмоид (логистическая функция): 

● Гиперболический тангенс: 

● Единичная ступенчатая функция (функция Хевисайда): 

● 

● Изображение выглядит как текст

Автоматически созданное описание

Слои нелинейной активации в PyTorch — это компоненты, которые добавляют нелинейность к моделям глубокого обучения, позволяя им учить более сложные закономерности в данных. В PyTorch есть несколько популярных функций нелинейной активации:

1. ReLU (Rectified Linear Unit): Это одна из самых часто используемых функций активации. Она заменяет все отрицательные значения в выходных данных нейронов на ноль. Пример использования: torch.nn.ReLU().

2. Sigmoid: Эта функция преобразует значения в диапазон от 0 до 1, что делает ее полезной для моделей, где необходимо предсказать вероятность. Пример использования: torch.nn.Sigmoid().

3. Tanh (Гиперболический тангенс): Преобразует данные так, чтобы они находились в диапазоне от -1 до 1. Это может быть полезно, если данные центрированы вокруг нуля. Пример использования: torch.nn.Tanh().

4. Leaky ReLU: Это вариация ReLU, которая вместо того, чтобы устанавливать отрицательные значения в ноль, умножает их на небольшой коэффициент. Это помогает избежать проблемы "умирающих нейронов", когда нейроны перестают обучаться. Пример использования: torch.nn.LeakyReLU().

5. Softmax: Часто используется в последнем слое многоклассовых классификационных моделей. Эта функция преобразует входные данные в распределение вероятностей по классам. Пример использования: torch.nn.Softmax(dim=1) для применения к выводам каждого образца.

Для использования этих функций активации в PyTorch, их обычно добавляют в модели как слои или применяют напрямую к данным в процессе определения модели.

1. Глубокое обучение. «Вторая весна искусственного интеллекта» и ее причины.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Зима ИИ – проблема линейно неразделимых множеств.

Первая весна ИИ – обратное распространение ошибки.

Изображение выглядит как текст, снимок экрана, Шрифт, диаграмма

Автоматически созданное описание

В 2012г AlexNet стал яркой причиной к тому, что всё сообщество ML повернулось лицом к глубоким нейронным сетям

1. Линейное отображение. Векторно-матричное дифференцирование.

Линейное отображение в контексте ИНС обычно относится к операциям, таким как взвешенная сумма входных данных.

Линейное отображение — обобщение линейной числовой функции на случай более общего множества аргументов и значений.

Векторно-Матричное Дифференцирование – это процесс нахождения скорости изменения одного вектора относительно другого. Векторно-матричное дифференцирование важно для вычисления градиентов в нейронных сетях.

Применение в ИНС:

В контексте нейронных сетей, линейное отображение часто используется в слоях сети для преобразования входных данных перед передачей их в функции активации. Векторно-матричное дифференцирование важно для процесса обучения нейронных сетей, так как оно используется в алгоритме обратного распространения ошибки для оптимизации весов сети.

Для того, чтобы выполнять операции с градиентами и тд, мы должны уметь брать производные сложных функций.

Изображение выглядит как текст, снимок экрана, Шрифт, письмо

Автоматически созданное описание

1. Проблема поиска градиента в общей логике обучения нейронной сети. Градиент функции многих переменных. Методы вычисления.

Градиент функции – это вектор, состоящий из частных производных функции по каждой из ее переменных. Он обычно используется для нахождения точек минимума (или максимума) функции, а также для получения гессиана функции.

В нейронных сетях градиенты часто используются для обучения с помощью градиентного спуска. В этом случае градиенты используются для определения, в какую сторону нужно изменить веса нейронной сети, чтобы уменьшить значение функции потерь.

Проблема: как рассчитать градиент от функции нескольких переменных.

Изображение выглядит как текст, диаграмма, снимок экрана, Шрифт

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описаниеИзображение выглядит как текст, снимок экрана, линия, Шрифт

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

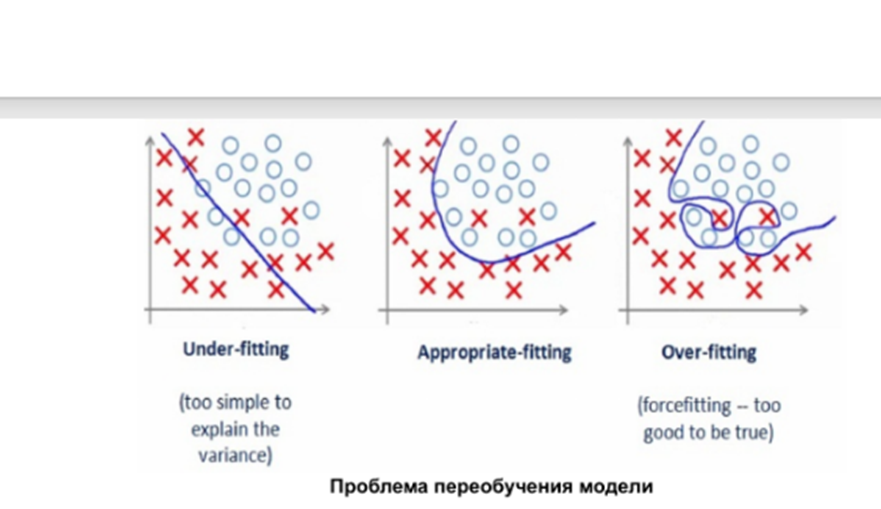
1. Кросс-валидация. Выборки train, validation, test. Проблема переобучения. Ранняя остановка.

Кросс-валидация - это метод оценки обобщающей способности модели, который включает разделение данных на тренировочный, валидационный и тестовый наборы. Это помогает предотвратить переобучение и оценить производительность модели.

Обучающая выборка (train) — это набор примеров, используемых в процессе обучения.

Валидационная выборка (validation) — это набор данных примеров, используемых для настройки гиперпараметров.

Тестовая выборка (test) – это набор данных, который не зависит от набора данных для обучения, но имеет то же распределение вероятностей, что и набор данных для обучения. Используется только для оценки производительности модели.



Изображение выглядит как текст, диаграмма, линия, число

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, Шрифт, документ

Автоматически созданное описание

1. Преобразование Softmax и функция потерь Cross Entropy loss.

Softmax используется в задачах многоклассовой классификации. Softmax преобразует логиты (выходы последнего слоя нейронной сети) в вероятности.

Выходные значения находятся в диапазоне [0,1], что хорошо, потому что мы можем чтобы избежать двоичной классификации и учесть как можно больше классов или измерений в нашей модели нейронной сети.

Преимущество Softmax перед обычным максимумом – дифференцируемость.

Изображение выглядит как текст, Шрифт, снимок экрана, линия

Автоматически созданное описание

Cross Entropy loss (или логарифмическая функция потерь – log loss) – это функция потерь, часто используемая в задачах классификации для измерения разницы между предсказанными (вероятности, выданные Softmax) и истинными распределениями (реальные метки классов). Цель состоит в том, чтобы минимизировать эту разницу.

Изображение выглядит как текст

Автоматически созданное описание

1. Механизм обратного распространения ошибки.

Изображение выглядит как текст, диаграмма, План, Шрифт

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, Шрифт, дизайн

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

1. Дифференцируемое программирование и реализация обратного распространения ошибки.

Дифференцируемое программирование позволяет компьютерным программам делать именно это: автоматически вычислять, как изменения входных данных влияют на выходные.

В контексте нейронных сетей, дифференцируемое программирование используется для автоматического вычисления градиентов в процессе обратного распространения ошибки. Это как если бы вы давали компьютеру задачу (например, минимизировать ошибку нейронной сети), и он сам находил бы наилучший способ решения, определяя, как нужно изменять веса сети.

Процесс и Применение:

1. Прямое распространение: Как и ранее, данные проходят через сеть, и делается предсказание.

2. Вычисление ошибки: Ошибка вычисляется путём сравнения предсказания с истинными значениями.

3. Автоматическое дифференцирование: С помощью дифференцируемого программирования автоматически вычисляются градиенты функции потерь по каждому параметру (весу) сети.

4. Обновление весов: На основе этих градиентов обновляются веса сети.

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

1. Стохастический градиентный спуск. Батчи обучающей выборки.

Пакетный градиентный спуск вычисляет один шаг, используя сразу весь набор данных, тогда как стохастический за шаг использует только 1 элемент. Можно два этих варианта скрестить, получив мини-пакетный (mini-batch) спуск, который за раз обрабатывает, к примеру, 100 элементов, а не все или один.

Два этих варианта ведут себя похоже, но не одинаково. Пакетный спуск действительно следует в направлении наискорейшего спуска, тогда как стохастический, используя только один элемент из обучающей выборки, не может верно вычислить градиент для всей выборки.

В жизни используются минипакетный спуск. Использование мини-батчей ускоряет обучение и помогает в оптимизации. Нормализация по мини-батчам помогает улучшить стабильность обучения.

Процесс и Применение:

1. Выбор батча: На каждом шаге алгоритма выбирается небольшой набор образцов (батч) из обучающей выборки.

2. Вычисление градиента: Для этого батча вычисляется градиент функции потерь.

3. Обновление весов: Веса в нейронной сети обновляются в направлении, противоположном градиенту, чтобы минимизировать функцию потерь.

1. Адаптивные методы градиентного спуска. Метод импульсов. Метод Нестерова.

Существует стохастический градиентный спуск, его можно улучшить.

Изображение выглядит как текст, снимок экрана, Шрифт, документ

Автоматически созданное описание

В этой ситуации могут помочь адаптивные методы градиентного спуска.

Изображение выглядит как текст, документ, снимок экрана, Шрифт

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

1. Проблема инициализации весов при обучении ИНС. Инициализация Ксавье.

В популярных фреймворках машинного обучения TensorFlow и PyTorch при инициализации весов нейросети используются случайные числа.

Для этих целей не используют ноль или какую-нибудь константу, так как при такой инициализации нейросеть может не обучаться совсем или обучаться плохо.

При инициализации весов нулями нейронная сеть со скрытым слоем теряет часть пространства возможных выходных значений, поскольку становится эквивалентна нейросети с одним нейроном на скрытом слое.

Если веса связей между каждыми двумя последовательными слоями инициализированы некоторой константой (вообще говоря, различной для каждой пары последовательных слоёв), то пространство возможных выходных значений нейронной сети сужается по сравнению со случаем, когда все веса инициализированы различными случайными значениями.

Изображение выглядит как текст, снимок экрана, Шрифт, документ

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

1. Гиперпараметры. Скорость обучения и размер батча.

Гиперпараметры - это настройки, которые необходимо выбрать перед обучением нейронной сети. Они оказывают влияние на процесс обучения и качество обученной модели. Два важных гиперпараметра - это скорость обучения (learning rate) и размер батча (batch size).

Некоторые примеры гиперпараметров в нейронных сетях:

1. Число слоев: глубина сети, может влиять на сложность модели и ее способность обобщать на новые данные

2. Число нейронов в слое: число нейронов в каждом слое сети, может влиять на качество модели и ее способность запоминать функцию

3. Функция активации: функция, которая используется для вычисления выхода нейрона в каждом слое, некоторые примеры функций: ReLU, sigmoid, tanh, etc.

4. Размер батча: это количество примеров, которые используются для обновления весов в каждой итерации обучения. Больший размер батча обычно позволяет ускорить обучение, так как уменьшает число обновлений весов, но может привести к ухудшению качества модели, так как увеличивает разброс градиента. Маленький размер батча, с другой стороны, может улучшить качество модели, но может уменьшить скорость обучения. Как и скорость обучения, размер батча также зависит от конкретной задачи и модели, и его также нужно подбирать экспериментально.

5. Скорость обучения: это темп, с которым модель обновляет веса в зависимости от градиента. Если скорость слишком высока, то модель может перескочить экстремум функции потерь и не найти оптимальное решение. Если скорость слишком низкая, то обучение может занять слишком много времени. Оптимальная скорость обучения зависит от конкретной задачи и модели, и ее нужно подбирать экспериментально.

1. Переобучение модели и регуляризация. Dropout.

Изображение выглядит как текст, Шрифт, линия, снимок экрана

Автоматически созданное описание

Изображение выглядит как текст, бумага, письмо, вышивка

Автоматически созданное описание

1. Минбатчи – причина использования. Нормализация по мини-батчам.

Изображение выглядит как текст, снимок экрана, Шрифт, документ

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, Шрифт, чек

Автоматически созданное описание

Изображение выглядит как текст, Шрифт, чек, снимок экрана

Автоматически созданное описание

1. Многослойные сети. Граф потока вычислений.

Изображение выглядит как текст, снимок экрана, диаграмма, линия

Автоматически созданное описаниеИзображение выглядит как текст, Шрифт, диаграмма, линия

Автоматически созданное описание

Изображение выглядит как текст, Шрифт, снимок экрана, алгебра

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

**Специфические задачи глубокого обучения**

1. Специфика задач машинного обучения на изображениях. Принцип работы сверточных сетей. Преимущества сверточных сетей при решении этих задач.

Задачи машинного обучения на изображениях:

- Изображения являются двумерными или трехмерными массивами данных, что усложняет их обработку.

- Изображения могут содержать различные шумы и искажения, что затрудняет их интерпретацию.

- Изображения могут быть очень большими, что требует больших вычислительных ресурсов.

Изображение выглядит как текст, снимок экрана, Шрифт, чек

Автоматически созданное описание

Принцип работы сверточных сетей:

- Сверточные сети используют сверточную операцию для извлечения признаков из изображений.

- Сверточная операция объединяет соседние пиксели в более крупные элементы данных, называемые \*\*фильтрами\*\*.

- Фильтры могут быть настроены для обнаружения определенных признаков, таких как края, текстуры или объекты.

Преимущества сверточных сетей при решении задач машинного обучения на изображениях:

- Сверточные сети хорошо подходят для обработки изображений, поскольку они учитывают пространственную структуру данных.

- Сверточные сети устойчивы к шумам и искажениям, что делает их более надежными, чем другие методы машинного обучения.

- Сверточные сети могут обрабатывать большие изображения эффективно, что позволяет им решать более сложные задачи.

1. Архитектура многослойной ИНС распознавания изображений на основе сверточных сетей.

Изображение выглядит как диаграмма, снимок экрана, текст, План

Автоматически созданное описание

- Входной слой:\*\* Входной слой принимает входные изображения в виде матрицы пикселей.

- Сверточные слои: Сверточные слои используются для извлечения признаков из изображений. Сверточная операция объединяет соседние пиксели в более крупные элементы данных, называемые \*\*фильтрами\*\*. Фильтры могут быть настроены для обнаружения определенных признаков, таких как края, текстуры или объекты.

- Пуллинговые слои: Пулинговые слои используются для уменьшения размерности выходных карт признаков сверточных слоев. Это помогает уменьшить вычислительную сложность сети и уменьшить переобучение.

- Полносвязные слои: Полносвязные слои используются для классификации изображений. Полносвязные нейроны связаны со всеми нейронами предыдущего слоя, что позволяет сети учитывать все признаки, извлеченные из изображения.

- Выходной слой: Выходной слой представляет собой один нейрон для каждого класса изображений. Выходной сигнал нейрона представляет собой вероятность принадлежности изображения к определенному классу.

1. Приемы для глубокого обучения на небольших наборах изображений.

Изображение выглядит как текст, снимок экрана, Шрифт, документ

Автоматически созданное описание

Изображение выглядит как текст, Шрифт, снимок экрана, документ

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, диаграмма, Шрифт

Автоматически созданное описание

Изображение выглядит как текст, Шрифт, снимок экрана, алгебра

Автоматически созданное описание

1. Схема работ слоя сверточной сети. Пулинг. Гиперпараметры: padding, kernel size, stride, dilation.

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, документ, Шрифт

Автоматически созданное описание

1. Задачи обработки текста: дистрибутивная семантика, матрица совместной встречаемости, представление слов в виде векторов малой размерности.

Обработка текста - это область искусственного интеллекта, которая занимается автоматическим анализом и пониманием текстовой информации. Она включает в себя множество задач, таких как:

- Классификация текста: определение темы или категории текста.

- Извлечение информации: поиск и извлечения конкретных данных из текста.

- Синтез текста: создание нового текста на основе существующего текста.

Дистрибутивная семантика - это подход к обработке текста, который использует частоту совместной встречаемости слов в тексте для представления слов в виде векторов.

Матрица совместной встречаемости - это таблица, которая показывает, как часто два слова встречаются вместе в тексте.

Представление слов в виде векторов малой размерности - это процесс преобразования матрицы совместной встречаемости в набор векторов меньшей размерности. Это делается для того, чтобы уменьшить вычислительную сложность обработки текста и улучшить его понимание.

Существует множество различных методов для представления слов в виде векторов малой размерности. Один из наиболее популярных методов - это метод главных компонент. Метод главных компонент находит набор из нескольких векторов, которые сохраняют основную информацию из матрицы совместной встречаемости.

1. Word2vec: модель CBOW.

Word2vec - это нейросетевая модель, которая используется для обучения векторных представлений слов. Векторные представления слов - это наборы чисел, которые описывают смысл слов.

CBOW - это сокращение от Continuous Bag of Words. Модель CBOW предсказывает слово в центре предложения, используя контекст, который его окружает.

Модель CBOW принимает на вход контекст, который окружает слово, которое нужно предсказать. Контекст может состоять из одного или нескольких слов. Модель CBOW затем использует эти слова для предсказания слова в центре предложения.

То есть CBOW превращает контекст в слово

Изображение выглядит как диаграмма, линия, План, Технический чертеж

Автоматически созданное описание

1. Word2vec: модель Skip-Gram.

Word2vec - это модель машинного обучения, которая используется для представления слов в виде векторов в многомерном пространстве. Это позволяет измерять семантическое сходство между словами.

Skip-gram - это один из двух основных типов моделей Word2vec. Он работает, обучая модель предсказывать контекстные слова для данного слова. Например, модель может научиться предсказывать, что слово "кот" будет окружено словами "мяу", "лапа" и "хвост".

То есть Skip-Gram превращает слово в контекст

Изображение выглядит как диаграмма, линия, План, Технический чертеж

Автоматически созданное описание

1. Рекуррентная нейронная сеть, принципы ее обучения. Сложности применения рекуррентных нейронных сетей.

Изображение выглядит как текст, снимок экрана, Шрифт, диаграмма

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, Шрифт, алгебра

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, диаграмма, Шрифт

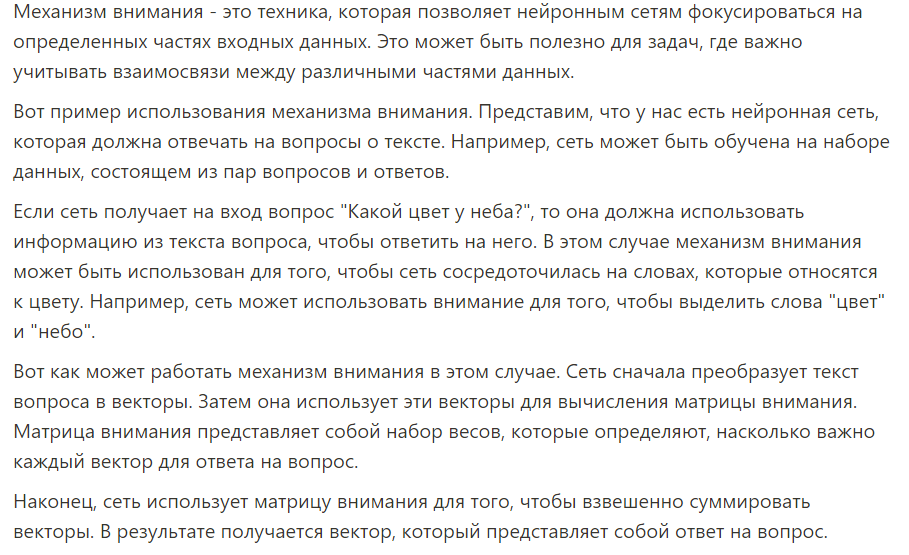
Автоматически созданное описание

1. Модуль LSTM.

Изображение выглядит как текст, снимок экрана, письмо

Автоматически созданное описание

1. Механизм Attention. Пример использования Attentinon.



1. Архитектура Transformer.

[Изображение выглядит как снимок экрана, текст, диаграмма, дизайн

Автоматически созданное описание](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A4%D0%B0%D0%B9%D0%BB:TransformerSimpleArchitecture.png)

Изображение выглядит как текст, снимок экрана, Шрифт, документ

Автоматически созданное описание

1. Модель BERT.

Модель BERT - это языковая модель, основанная на архитектуре Transformer. Она была представлена в 2018 году исследователями Google и быстро стала одним из самых эффективных инструментов для обработки естественного языка.

Модель BERT состоит из нескольких слоев Transformer, которые последовательно обрабатывают текст. На каждом слое Transformer используется механизм внимания, который позволяет модели учитывать взаимосвязи между различными словами в тексте.

● BERT является автокодировщиком

● BERT использует трансформер-архитектуру

● В каждом слое кодировщика применяется двустороннее внимание, что позволяет учитывать контекст с обеих сторон от рассматриваемого токена

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как снимок экрана, текст, линия, Шрифт

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

**PyTorch и задачи глубокого обучения**

1. Класс Tensor. Операции, изменяющие размер тензора. Операции агрегации.

Класс Tensor является основным строительным блоком для работы с тензорами во многих библиотеках машинного обучения, таких как TensorFlow или PyTorch. Тензор представляет собой многомерный массив данных, и его размерность определяется числом осей.

import torch

# Создание тензора

tensor = torch.tensor([[1, 2, 3],

[4, 5, 6]])

Операции, изменяющие размер тензора:

* Reshape (Изменение формы): Эта операция изменяет форму тензора, сохраняя при этом общее количество элементов. Например, тензор размером (2, 3) может быть изменен в тензор размером (3, 2) при использовании операции reshape. tensor.reshape(3, 2)
* Transpose (Транспонирование): Транспонирование изменяет порядок осей в тензоре. Например, транспонирование тензора размером (2, 3) дает тензор размером (3, 2). tensor.t()
* Expand (Расширение): Эта операция увеличивает размерность тензора, добавляя новые оси с размерностью 1. tensor.unsqueeze(0)
* Squeeze (Сжатие): Сжатие удаляет размерности с размерностью 1 из тензора. tensor.squeeze(0)

Операции агрегации:

* Sum (Сумма): Эта операция вычисляет сумму значений в тензоре по определенным осям. tensor.sum(dim=1)
* Mean (Среднее): Среднее вычисляет среднее значение значений в тензоре по определенным осям. tensor.mean(dim=0)
* Max (Максимум): Операция max возвращает максимальное значение в тензоре или по указанным осям. tensor.max().item()
* Min (Минимум): Операция min возвращает минимальное значение в тензоре или по указанным осям.
* Argmax (Индекс максимального значения): Возвращает индексы максимальных значений в тензоре. tensor.argmax(dim=1)
* Argmin (Индекс минимального значения): Возвращает индексы минимальных значений в тензоре.

1. Принципиальная логика обучения нейронной сети.

Базовый цикл обучения ИНС:

Инициализация весов:

Начальные веса нейронной сети инициализируются случайным образом. Это важный шаг, так как правильная инициализация может существенно влиять на процесс обучения.

Прямой проход (Forward Propagation):

Данные передаются через нейронную сеть от входного слоя к выходному. Каждый нейрон в сети принимает входные данные, умножает их на соответствующие веса, применяет активационную функцию и передает результат следующему слою.

Вычисление потерь (Loss Computation):

Сравниваются предсказанные значения с фактическими, и вычисляется функция потерь. Функция потерь измеряет, насколько предсказания модели отличаются от фактических значений.

Обратное распространение (Backward Propagation):

Градиенты функции потерь вычисляются с помощью метода обратного распространения. Градиенты показывают, как изменится функция потерь при изменении весов. Эти градиенты затем используются для обновления весов сети.

Обновление весов (Weight Update):

Веса сети обновляются с использованием оптимизационного алгоритма, такого как градиентный спуск. Это позволяет сети постепенно улучшать свои предсказательные способности, минимизируя функцию потерь.

Итерации:

Процесс прямого и обратного распространения повторяется многократно (несколько эпох) для всего набора данных. Каждая итерация включает в себя прогон данных через сеть, вычисление потерь, обратное распространение и обновление весов.

Оценка и тестирование:

После завершения обучения модель тестируется на отложенном наборе данных для оценки ее обобщающей способности. Это позволяет определить, насколько хорошо модель обучилась и способна ли она делать предсказания на новых данных.

Настройка гиперпараметров:

Гиперпараметры, такие как скорость обучения, количество слоев и нейронов в сети, выбираются до начала обучения и могут быть подогнаны в процессе, чтобы улучшить производительность модели.

1. Автоматическое дифференцирование в PyTorch. Пример и применение в обучении ИНС.

PyTorch предоставляет мощный инструмент для автоматического дифференцирования, который может быть использован для обучения нейронных сетей. Это достигается с помощью создания графа вычислений, в котором каждый узел представляет собой операцию, а ребра представляют тензоры, которые передаются между операциями. При вычислении градиента используется обратное распространение ошибки (backpropagation).

Чтобы использовать автоматическое дифференцирование в PyTorch, необходимо создать объект torch.Tensor, связать его с графом вычислений и затем использовать функцию backward() для вычисления градиента.

Пример:

import torch

# Create tensors

x = torch.tensor([2.0], requires\_grad=True)

y = x\*\*2 + 2\*x + 1

# Compute gradients

y.backward()

# Print gradients

print(x.grad) # Output: tensor([4.])

В обучении ИНС автоматическое дифференцирование используется для вычисления градиентов по параметрам модели при обратном распространении ошибки. Это позволяет обновлять параметры модели в направлении антиградиента, что помогает уменьшить значение функции потерь.

1. Загрузка и преобразование данных. Классы Dataset, DataLoader, Transforms (и композиция трансформеров).

В мире машинного обучения, загрузка и преобразование данных - это важная часть подготовки данных перед обучением модели. В PyTorch, для этих целей часто используются три основных компонента: Dataset, DataLoader и Transforms.

Dataset - это абстрактный класс в PyTorch, предназначенный для предоставления доступа к данным.

Transforms - это набор операций, которые могут быть применены к данным, как правило, во время загрузки данных с использованием Dataset.

from torchvision import transforms

custom\_transform = transforms.Compose([

transforms.ToTensor(),

transforms.Normalize(mean=[0.5], std=[0.5])

])

DataLoader - это обертка над Dataset, которая предоставляет удобный способ итерации по данным во время обучения.

from torch.utils.data import DataLoader

batch\_size = 64

train\_dataset = CustomDataset(data, labels, transform=custom\_transform)

train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True)

1. Класс nn.Module. Назначение. Основные поля, методы.

Класс nn.Module в PyTorch является базовым классом для всех нейронных сетей и обеспечивает большую часть необходимой функциональности для создания и управления моделями. Вот основные аспекты этого класса:

Назначение

1. Организация компонентов сети: nn.Module позволяет объединять различные слои и функции активации в единую сетевую структуру.

2. Управление параметрами: Автоматически управляет параметрами сети (весами, смещениями), что облегчает их оптимизацию и сохранение.

3. Поддержка GPU: Поддерживает автоматическое переключение между использованием CPU и GPU.

Основные поля

1. Параметры сети: Все параметры (веса, смещения) модулей, являющиеся его подклассами, автоматически регистрируются в списке параметров.

2. Подмодули: Можно добавлять дочерние модули, которые также являются экземплярами nn.Module.

Основные методы

1. forward: Основной метод, который определяет прямой проход (forward pass) через модель. Этот метод должен быть переопределен при создании нового класса, наследующего от nn.Module.

2. \_\_init\_\_: Конструктор класса, где определяются слои и другие компоненты модели.

3. to(device): Перемещает параметры и буферы модели на указанный устройство (cpu или cuda).

4. state\_dict(): Возвращает словарь, содержащий все параметры модели.

5. load\_state\_dict(state\_dict): Загружает параметры из словаря в модель.

6. zero\_grad(): Обнуляет градиенты параметров, что необходимо перед выполнением обратного распространения ошибки (backpropagation).

7. eval(): Переводит модель в режим оценки (evaluation mode), который отключает слои, ведущие себя по-разному во время обучения и тестирования (например, Dropout).

8. train(): Переключает модель в режим обучения.

1. Линейные слои (Linear Layers).

Линейные слои (или полносвязные слои) - это один из основных типов слоев в нейронных сетях. Они представляют собой матричные операции, где каждый входной нейрон соединен с каждым выходным нейроном с использованием весов. В PyTorch линейные слои реализуются с использованием класса torch.nn.Linear.

import torch

import torch.nn as nn

# Определение размерности входных и выходных данных

input\_size = 10

output\_size = 5

# Создание линейного слоя

linear\_layer = nn.Linear(input\_size, output\_size)

# Генерация случайного входного тензора

input\_tensor = torch.randn((3, input\_size))

# Прохождение входного тензора через линейный слой

output\_tensor = linear\_layer(input\_tensor)

# Вывод информации о слое и результатах

print("Linear Layer:")

print(linear\_layer)

print("\nInput Tensor:")

print(input\_tensor)

print("\nOutput Tensor:")

print(output\_tensor)

1. Слои нелинейной активации (Non Linear Activations).

Слои нелинейной активации в нейронных сетях используются для внесения нелинейности в модель. Они представляют собой нелинейные функции, которые применяются к выходам нейронов, добавляя гибкость и способствуя обучению сложных зависимостей в данных. В PyTorch такие слои реализованы как функции активации в модуле `torch.nn.functional`.

Вот несколько примеров популярных функций активации:

1. ReLU - это функция активации, которая возвращает входное значение, если оно положительное, и 0 в противном случае.

python

import torch

import torch.nn.functional as F

x = torch.randn(5)

output = F.relu(x)

2. Sigmoid - это функция активации, которая возвращает значения в диапазоне от 0 до 1. Она часто используется для бинарной классификации.

x = torch.randn(5)

output = F.sigmoid(x)

3. Tanh - это функция активации, которая возвращает значения в диапазоне от -1 до 1. Полезна при работе с центрированными данных.

x = torch.randn(5)

output = F.tanh(x)

4. Softmax - это функция активации, которая используется для преобразования выходов сети в вероятности. Она приводит значения к вероятностному распределению.

x = torch.randn(5)

probabilities = F.softmax(x, dim=0)

При построении нейронных сетей в PyTorch эти функции активации можно использовать как слои модели, добавляя их после линейных слоев или других операций. Например:

import torch

import torch.nn as nn

class NeuralNetwork(nn.Module):

def \_\_init\_\_(self):

super(NeuralNetwork, self).\_\_init\_\_()

self.linear = nn.Linear(10, 5)

self.relu = nn.ReLU()

def forward(self, x):

x = self.linear(x)

x = self.relu(x)

return x

Этот пример демонстрирует использование слоя ReLU после линейного слоя в составе нейронной сети.

1. Слои нормализации (Normalization Layers).

Слои нормализации в нейронных сетях помогают стабилизировать и ускорить обучение, предотвращая внутренний сдвиг распределения активаций. В PyTorch, такие слои представлены различными видами нормализации, такими как Batch Normalization (`nn.BatchNorm1d` и `nn.BatchNorm2d`), Layer Normalization (`nn.LayerNorm`), и Instance Normalization (`nn.InstanceNorm1d` и `nn.InstanceNorm2d`).

Примеры использования:

1. Batch Normalization применяется по размерности батча. Он вычисляет среднее и дисперсию активаций по батчу и использует их для нормализации.

import torch

import torch.nn as nn

# Пример для одномерных данных (например, в полносвязных слоях)

batch\_norm = nn.BatchNorm1d(5)

x = torch.randn((10, 5))

output = batch\_norm(x)

# Пример для двумерных данных (например, в сверточных слоях)

batch\_norm\_2d = nn.BatchNorm2d(3)

x\_2d = torch.randn((10, 3, 28, 28))

output\_2d = batch\_norm\_2d(x\_2d)

2. Layer Normalization применяется по размерности признаков. Он вычисляет среднее и дисперсию активаций по каждому признаку внутри каждого примера.

# Пример для одномерных данных (например, в полносвязных слоях)

layer\_norm = nn.LayerNorm(5)

x = torch.randn((10, 5))

output = layer\_norm(x)

# Пример для двумерных данных (например, в сверточных слоях)

layer\_norm\_2d = nn.LayerNorm((3, 28, 28))

x\_2d = torch.randn((10, 3, 28, 28))

output\_2d = layer\_norm\_2d(x\_2d)

3. Instance Normalization применяется по размерности признаков, но вычисляет среднее и дисперсию для каждого примера независимо.

# Пример для одномерных данных (например, в полносвязных слоях)

instance\_norm = nn.InstanceNorm1d(5)

x = torch.randn((10, 5))

output = instance\_norm(x)

# Пример для двумерных данных (например, в сверточных слоях)

instance\_norm\_2d = nn.InstanceNorm2d(3)

x\_2d = torch.randn((10, 3, 28, 28))

output\_2d = instance\_norm\_2d(x\_2d)

Эти слои нормализации могут быть полезны в различных архитектурах нейронных сетей и могут помочь в повышении стабильности обучения и ускорении сходимости модели.

1. Слои регуляризации (Dropout Layers).

Слои регуляризации, такие как Dropout, являются инструментами для предотвращения переобучения в нейронных сетях. Dropout случайным образом "выключает" (устанавливает в ноль) некоторые нейроны во время обучения, что помогает сети обучаться более устойчиво и лучше обобщать на новые данные. В PyTorch, слой Dropout представлен классом `torch.nn.Dropout`.

Пример использования Dropout в нейронной сети:

import torch

import torch.nn as nn

class NeuralNetworkWithDropout(nn.Module):

def \_\_init\_\_(self):

super(NeuralNetworkWithDropout, self).\_\_init\_\_()

self.fc1 = nn.Linear(10, 5)

self.relu = nn.ReLU()

self.dropout = nn.Dropout(p=0.5) # p - вероятность "выключения" нейрона

def forward(self, x):

x = self.fc1(x)

x = self.relu(x)

x = self.dropout(x)

return x

# Создание экземпляра сети с Dropout

model\_with\_dropout = NeuralNetworkWithDropout()

# Генерация случайного входного тензора

input\_tensor = torch.randn((3, 10))

# Прохождение входного тензора через сеть

output\_tensor = model\_with\_dropout(input\_tensor)

# Вывод информации о слое и результатах

print("Neural Network with Dropout:")

print(model\_with\_dropout)

print("\nInput Tensor:")

print(input\_tensor)

print("\nOutput Tensor:")

print(output\_tensor)

В этом примере создается небольшая нейронная сеть с использованием Dropout. Dropout применяется после линейного слоя и активационной функции ReLU. Заметьте, что Dropout активен только во время обучения (`model.train()`), и при инференсе (`model.eval()`) нейроны не выключаются.

Для контроля параметра `p` (вероятности "выключения" нейрона), вы можете использовать другие значения в интервале (0, 1). Обычно распространенные значения - 0.2, 0.5 или 0.7. Выбор конкретного значения может потребоваться некоторые эксперименты в зависимости от конкретной задачи и архитектуры сети.

1. Сверточные слои (Convolution Layers). Сжимающие слои (Pooling Layers).

Сверточные слои (Convolutional Layers) и сжимающие слои (Pooling Layers) являются ключевыми компонентами сверточных нейронных сетей, широко используемых в обработке изображений и других задачах, где пространственная структура данных важна.

1. \*\*Сверточные слои:\*\*

- \*\*Определение:\*\* Сверточные слои применяют фильтры (ядра) к входным данным для выделения локальных шаблонов и признаков. Эти слои обычно используются для извлечения признаков из изображений.

- \*\*Использование:\*\*

```python

import torch

import torch.nn as nn

# Пример для одномерных данных (например, временные ряды)

conv1d\_layer = nn.Conv1d(in\_channels=1, out\_channels=16, kernel\_size=3)

x\_1d = torch.randn((32, 1, 10))

output\_1d = conv1d\_layer(x\_1d)

# Пример для двумерных данных (например, изображения)

conv2d\_layer = nn.Conv2d(in\_channels=3, out\_channels=64, kernel\_size=3, padding=1)

x\_2d = torch.randn((32, 3, 28, 28))

output\_2d = conv2d\_layer(x\_2d)

```

2. \*\*Сжимающие слои (Pooling Layers):\*\*

- \*\*Определение:\*\* Сжимающие слои, также называемые слоями пулинга, уменьшают размерность данных, объединяя значения в кластеры. Это помогает уменьшить количество параметров и вычислений, сохраняя важные признаки.

- \*\*Использование:\*\*

```python

import torch

import torch.nn as nn

# Пример для одномерных данных (например, временные ряды)

maxpool1d\_layer = nn.MaxPool1d(kernel\_size=2)

x\_1d = torch.randn((32, 1, 10))

output\_1d = maxpool1d\_layer(x\_1d)

# Пример для двумерных данных (например, изображения)

maxpool2d\_layer = nn.MaxPool2d(kernel\_size=2)

x\_2d = torch.randn((32, 3, 28, 28))

output\_2d = maxpool2d\_layer(x\_2d)

```

В этих примерах `in\_channels` - количество каналов во входных данных, `out\_channels` - количество фильтров (или ядер), `kernel\_size` - размер ядра свертки или окна пулинга. Параметр `padding` используется для управления размером выхода в сверточных слоях.

Сверточные и сжимающие слои обычно используются вместе для построения глубоких сверточных нейронных сетей, способных эффективно извлекать признаки из сложных данных.

1. Слои функций потерь (Loss Functions).

Слои функций потерь (Loss Functions) в нейронных сетях используются для измерения разницы между прогнозами модели и фактическими метками (целевыми значениями). Целью обучения является минимизация значения функции потерь. В PyTorch множество различных функций потерь реализованы в модуле `torch.nn`.

Вот несколько примеров функций потерь:

1. \*\*Кросс-энтропия (CrossEntropyLoss):\*\*

- \*\*Определение:\*\* Используется для задач классификации. Кросс-энтропия измеряет расхождение между предсказанными вероятностями классов и истинными метками.

- \*\*Использование:\*\*

```python

import torch

import torch.nn as nn

loss\_fn = nn.CrossEntropyLoss()

# Пример использования

logits = torch.randn((64, 10)) # Пример вывода модели

labels = torch.randint(0, 10, (64,)) # Пример истинных меток

loss = loss\_fn(logits, labels)

```

2. \*\*MSE (Mean Squared Error):\*\*

- \*\*Определение:\*\* Используется для задач регрессии. MSE измеряет среднеквадратичную разницу между предсказанными значениями и истинными метками.

- \*\*Использование:\*\*

```python

import torch

import torch.nn as nn

loss\_fn = nn.MSELoss()

# Пример использования

predictions = torch.randn((64, 1)) # Пример предсказаний модели

targets = torch.randn((64, 1)) # Пример истинных значений

loss = loss\_fn(predictions, targets)

```

3. \*\*Бинарная кросс-энтропия (BCELoss):\*\*

- \*\*Определение:\*\* Используется для задач бинарной классификации. BCELoss измеряет расхождение между предсказанными вероятностями и бинарными метками.

- \*\*Использование:\*\*

```python

import torch

import torch.nn as nn

loss\_fn = nn.BCELoss()

# Пример использования

logits = torch.randn((64, 1)) # Пример вывода модели

labels = torch.randint(0, 2, (64, 1)).float() # Пример бинарных меток

loss = loss\_fn(torch.sigmoid(logits), labels)

```

1. Слои эмбеддингов nn.Embedding и их применение.

Слои эмбеддингов (`nn.Embedding`) в нейронных сетях используются для отображения категориальных данных, таких как индексы слов, пользователей или товаров, в плотные векторные представления (эмбеддинги). Эти слои обычно применяются в задачах обработки естественного языка (NLP) или рекомендательных системах, где важно представление категориальных данных в непрерывном пространстве.

Пример использования `nn.Embedding`:

```python

import torch

import torch.nn as nn

# Определение слоя эмбеддингов

embedding\_layer = nn.Embedding(num\_embeddings=1000, embedding\_dim=50)

# Пример использования

input\_indices = torch.LongTensor([1, 5, 9, 3, 7]) # Индексы категориальных данных

embeddings = embedding\_layer(input\_indices)

# Вывод результатов

print("Embeddings Shape:", embeddings.shape)

```

В этом примере `num\_embeddings` - это общее количество уникальных категорий, а `embedding\_dim` - размерность эмбеддинга для каждой категории. В результате работы `nn.Embedding` для каждого входного индекса мы получаем соответствующий вектор-эмбеддинг.

1. Класс torch.nn.LSTM и torch.nn.GRU.

`torch.nn.LSTM` и `torch.nn.GRU` - это два различных класса в PyTorch, представляющих собой реализации рекуррентных слоев для обработки последовательных данных, таких как последовательности временных рядов, текста и других временных данных.

1. \*\*`torch.nn.LSTM`:\*\*

- \*\*Определение:\*\* LSTM (Long Short-Term Memory) - это тип рекуррентной нейронной сети, спроектированный для преодоления проблемы затухающих градиентов. Он содержит ячейку памяти, которая может хранить информацию в течение длительного времени.

- \*\*Пример использования:\*\*

```python

import torch

import torch.nn as nn

# Определение слоя LSTM

lstm\_layer = nn.LSTM(input\_size=10, hidden\_size=20, num\_layers=2, batch\_first=True)

# Пример использования

input\_data = torch.randn((32, 5, 10)) # Пример входных данных (batch\_size, sequence\_length, input\_size)

output, (hn, cn) = lstm\_layer(input\_data)

```

Здесь `input\_size` - размер входных данных, `hidden\_size` - размер скрытого состояния, `num\_layers` - количество слоев LSTM.

2. \*\*`torch.nn.GRU`:\*\*

- \*\*Определение:\*\* GRU (Gated Recurrent Unit) - это другой тип рекуррентной нейронной сети, который также разработан для решения проблемы затухающих градиентов. Он обладает меньшим количеством параметров по сравнению с LSTM.

- \*\*Пример использования:\*\*

```python

import torch

import torch.nn as nn

# Определение слоя GRU

gru\_layer = nn.GRU(input\_size=10, hidden\_size=20, num\_layers=2, batch\_first=True)

# Пример использования

input\_data = torch.randn((32, 5, 10)) # Пример входных данных (batch\_size, sequence\_length, input\_size)

output, hn = gru\_layer(input\_data)

```

Здесь `input\_size` - размер входных данных, `hidden\_size` - размер скрытого состояния, `num\_layers` - количество слоев GRU.

Оба слоя предоставляют возможность обрабатывать последовательности данных, сохраняя контекст в скрытом состоянии. Выбор между `LSTM` и `GRU` зависит от конкретной задачи и характеристик данных. В некоторых случаях `LSTM` может быть более мощным, но `GRU` может быть вычислительно более эффективным и быстрым в обучении.