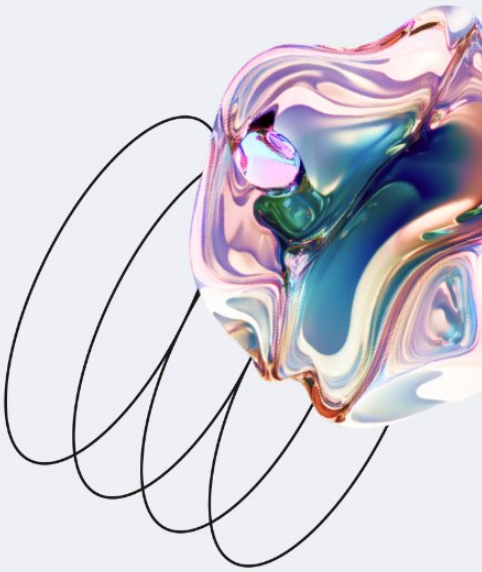
GeekBrains



Основы PyTorch

Оглавление

Введение

Добро пожаловать на лекцию по основам PyTorch! На предыдущих занятиях мы изучали различные типы нейронных сетей, такие как полносвязные сети, свёрточные нейронные сети (CNN) и рекуррентные нейронные сети (RNN). Обсуждали их архитектуры, особенности и примеры применения. Сегодня сделаем важный шаг к практическому применению этих знаний, изучив один из самых популярных инструментов для разработки нейронных сетей — PyTorch.

PyTorch — это открытая библиотека машинного обучения, разработанная и поддерживаемая Facebook AI Research (FAIR) (Facebook – продукт компании Meta, которая признана экстремистской организацией в России).

PyTorch особенно ценится исследователями и разработчиками за свою интуитивно понятную структуру, динамическое вычислительное графическое ядро и богатый функционал.

В рамках лекции рассмотрим следующие ключевые аспекты PyTorch:

1. **Основные компоненты PyTorch**: что такое тензоры, как их создавать и манипулировать ими.
2. **Автоматическое дифференцирование**: как PyTorch вычисляет градиенты, что позволяет эффективно обучать нейронные сети.
3. **Создание и обучение моделей**: как определить модель, задать функцию потерь и использовать оптимизаторы для её обучения.
4. **Примеры применения**: рассмотрим несколько примеров, чтобы закрепить полученные знания и увидеть, как PyTorch используется на практике.

В результате вы получите общее представление о том, как использовать PyTorch для разработки и обучения нейронных сетей, и будете готовы к выполнению практических заданий

Давайте начнем наш путь в мир PyTorch и увидим, как он может облегчить нам работу с нейронными сетями!

Словарь терминов

PyTorch — это оптимизированная тензорная библиотека глубокого обучения, основанная на Python и Torch, которая в основном используется для приложений, использующих графические и центральные процессоры.
Градиент — это значение, используемое для корректировки внутренних весов сети, позволяющее сети обучаться

Основы Pytorch

С момента своего создания в 2017 году PyTorch стал очень популярной и эффективной средой для создания модели глубокого обучения. Эта библиотека машинного обучения с открытым исходным кодом основана на Torch и предназначена для обеспечения большей гибкости и повышенной скорости реализации глубоких нейронных сетей . В настоящее время PyTorch является наиболее популярной библиотекой для исследователей и практиков искусственного интеллекта во всем мире в промышленности и научных кругах.

Что такое PyTorch и как он работает?

PyTorch — это оптимизированная тензорная библиотека глубокого обучения, основанная на Python и Torch.
Torch — проект с открытым исходным кодом для глубокого обучения, написанный на языке C и обычно используемый через интерфейс Lua. Это был проект-предшественник PyTorch, который больше не разрабатывается активно. Новая версия ориентирована на Python.
API PyTorch прост и гибок, что делает его любимым для ученых и исследователей, занимающихся разработкой новых моделей и приложений глубокого обучения. Широкое использование привело к появлению множества расширений для конкретных приложений, таких как текст, компьютерное зрение и аудиоданные. Также есть возможность использовать предобученные модели, которые можно использовать напрямую.
PyTorch предпочтительнее других фреймворков глубокого обучения, таких как TensorFlow и Keras, поскольку он использует динамические графы вычислений и полностью Pythonic (написан на Python с использованием парадигм языка и предназначенный для использование в Python скриптах). Он позволяет ученым, разработчикам и отладчикам нейронных сетей запускать и тестировать части кода в режиме реального времени. Таким образом, пользователям не нужно ждать реализации всего кода, чтобы проверить, работает ли часть кода или нет.
Двумя основными особенностями PyTorch являются:

- Тензорные вычисления (аналогично NumPy) с мощной поддержкой ускорения графического процессора.
- Автоматическая дифференциация для создания и обучения глубоких нейронных сетей

Основы PyTorch

Основные операции PyTorch очень похожи на Numpy. Давайте сначала разберемся с основами.

- Введение в тензоры

В машинном обучении, когда мы представляем данные, нам нужно делать это в числовом виде. Тензор — это просто контейнер, который может хранить данные в нескольких измерениях. Однако с математической точки зрения тензор — это фундаментальная единица данных, которую можно использовать в качестве основы для сложных математических операций. Это может быть число, вектор, матрица или многомерный массив, например массивы Numpy. Тензоры также могут обрабатываться процессором или графическим процессором, чтобы ускорить операции. Существуют различные типы тензоров, такие как Float Tensor, Double Tensor, Half Tensor, Int Tensor и Long Tensor, но PyTorch использует 32-битный Float Tensor в качестве типа по умолчанию.

- Математические операции

Коды для выполнения математических операций в PyTorch такие же, как и в Numpy. Пользователям необходимо инициализировать два тензора, а затем выполнять над ними такие операции, как сложение, вычитание, умножение и деление.

- Инициализация матрицы и матричные операции

Чтобы инициализировать матрицу случайными числами в PyTorch, используйте функцию randn(), которая дает тензор, заполненный случайными числами из стандартного нормального распределения. Установка случайного начального числа в начале будет генерировать одни и те же числа каждый раз, когда вы запускаете этот код. Основные матричные операции и операции транспонирования в PyTorch также аналогичны NumPy.

Почему стоит использовать PyTorch для глубокого обучения?

Использование PyTorch для глубокого обучения предоставляет множество преимуществ, которые сделали его одним из наиболее популярных инструментов в этой области. Рассмотрим основные причины, по которым PyTorch стоит использовать для глубокого обучения:

1. Интуитивность и простота использования:
    - Динамическое вычисление графов (Dynamic Computation Graphs): В отличие от некоторых других библиотек, например, TensorFlow, PyTorch использует динамические вычислительные графы. Это означает, что граф вычислений строится "на лету" во время выполнения кода, что упрощает отладку и позволяет гибко изменять архитектуру модели.
    - Понятный синтаксис: PyTorch имеет синтаксис, близкий к обычному Python, что делает его легким для изучения и использования. Это особенно удобно для новичков в области глубокого обучения.
2. Автоматическое дифференцирование:
    - PyTorch включает мощную систему автоматического дифференцирования (autograd), которая автоматически вычисляет градиенты для оптимизации моделей. Это значительно упрощает процесс обучения нейронных сетей.
3. Гибкость и модульность:
    - Модульная структура: PyTorch позволяет легко создавать сложные модели, комбинируя различные слои и функции. Это делает его отличным выбором как для исследований, так и для промышленного применения.
    - Поддержка CUDA: PyTorch обеспечивает удобное использование графических процессоров (GPU) для ускорения вычислений. Это особенно важно для обучения больших моделей на больших наборах данных.
4. Сообщество и поддержка:
    - PyTorch активно поддерживается и развивается сообществом исследователей и инженеров. Большое количество документации, обучающих материалов и форумов позволяет быстро находить ответы на возникающие вопросы.
    - Экосистема и интеграция: PyTorch легко интегрируется с другими инструментами и библиотеками, такими как NumPy, SciPy и другие. Кроме того, существует множество вспомогательных библиотек, таких как torchvision для работы с изображениями и torchtext для обработки текста.
5. Практическое применение и поддержка индустрии:
    - Многие ведущие компании и исследовательские организации используют PyTorch для разработки и внедрения решений на основе глубокого обучения. Это подтверждает его надежность и эффективность.
    - PyTorch используется в различных приложениях, от обработки изображений и видео до обработки естественного языка и биоинформатики.
6. Обучающие ресурсы и примеры:
    - PyTorch предоставляет множество учебных ресурсов, примеров и готовых моделей, которые можно использовать в своих проектах. Это облегчает процесс обучения и внедрения новых идей.

В итоге, PyTorch сочетает в себе удобство использования, мощные возможности и гибкость, что делает его отличным выбором для широкого круга задач в области глубокого обучения.

Установка PyTorch

Установка PyTorch проста и может быть выполнена несколькими способами в зависимости от вашей операционной системы и предпочтений по установке. Вот пошаговое руководство по установке PyTorch на различные платформы:

Установка через командную строку

1. Определите конфигурацию вашей системы

- Операционная система: Убедитесь, что вы знаете, какую операционную систему используете (Windows, macOS, Linux).
- Версия Python: PyTorch поддерживает Python 3.7 и выше.
- Поддержка CUDA: Если вы планируете использовать PyTorch с GPU, определите версию CUDA, установленную на вашей системе. Если у вас нет GPU или вы не хотите использовать его, можно установить версию для CPU.

2. Откройте командную строку или терминал

На Windows это может быть PowerShell или Command Prompt, на macOS и Linux — Terminal.

3. Используйте команду установки с официального сайта

Перейдите на официальный сайт PyTorch и выберите нужные параметры в разделе "Install PyTorch". Сайт автоматически сгенерирует нужную команду для установки.

Пример команды для установки PyTorch с поддержкой CUDA 11.8 через pip:

pip install torch==2.0.1+cu118 torchvision==0.15.2+cu118 torchaudio==2.0.2 -f https://download.pytorch.org/whl/torch_stable.html

Для версии только для CPU:

pip install torch torchvision torchaudio

Установка через Anaconda

Если вы используете Anaconda (рекомендуется для удобства управления зависимостями и изоляции окружений), то установка PyTorch также очень проста.

1. Создайте новое окружение (опционально)
conda create -n myenv python=3.9

conda activate myenv

2. Установите PyTorch
Команда для установки PyTorch с поддержкой CUDA 11.8:

conda install pytorch torchvision torchaudio pytorch-cuda=11.8 -c pytorch -c nvidia

Для версии только для CPU:

conda install pytorch torchvision torchaudio cpuonly -c pytorch

Проверка установки

После установки PyTorch, откройте Python интерпретатор и выполните следующие команды, чтобы убедиться, что PyTorch установлен правильно:

```
import torch
print(torch.__version__)

print(torch.cuda.is_available())
```

Команда print(torch.__version__) должна вывести установленную версию PyTorch, а print(torch.cuda.is_available()) вернет True, если CUDA доступна, и False, если используется версия только для CPU.

Эти шаги помогут вам успешно установить PyTorch и начать его использование для ваших проектов по глубокому обучению.

Тензоры

Тензоры — это основные структуры данных, которые используются в нейронных сетях.
Все входы, выходы и промежуточные представления представляют собой тензоры.
Программирование нейронных сетей активно использует тензоры.
Тензоры делятся на три группы:

1. Числа (скаляры). Термин «число» используется в информатике, «скаляр» — в математике.
2. Массивы (векторы). Термин «массив» используется в информатике, «вектор» — в математике.
3. Матрицы (двумерные массивы). Используются в линейной алгебре и программировании.

Число индексов	**Информатика**	**Математика**
0	Число	Скаляр
1	Массив	Вектор
2	Двумерный массив	Матрица
n	Многомерный массив	n-мерный тензор

Тензоры — это многомерные массивы.

- **Скаляр** — это 0-мерный тензор.
- **Вектор** — это 1-мерный тензор.
- **Матрица** — это n-мерный тензор.

```
a = [2, 1, 2]  # Индексы
print(a[1])      # Вывод: 2
dd = [
   [1, 2, 3],
   [2, 3, 4]
]
print(dd[0][1])  # Вывод: 2
```

Размерности тензоров

Векторное пространство характеризуется размерностью, которая определяет число компонентов в векторе. Но у тензора размерность не указывает количество его компонентов.
Например, трёхмерный вектор из евклидового пространства описывается тройкой (x, y, z), то есть имеет три компонента.
Трёхмерный тензор, напротив, может иметь гораздо больше компонентов.
Например, двумерный тензор **dd** имеет девять компонентов:

```
dd = [
        [1, 2, 3],
        [4, 5, 6],
```

```
        [7, 8, 9]
]
```

Ранг, оси и форма тензоров тесно связаны с индексами.

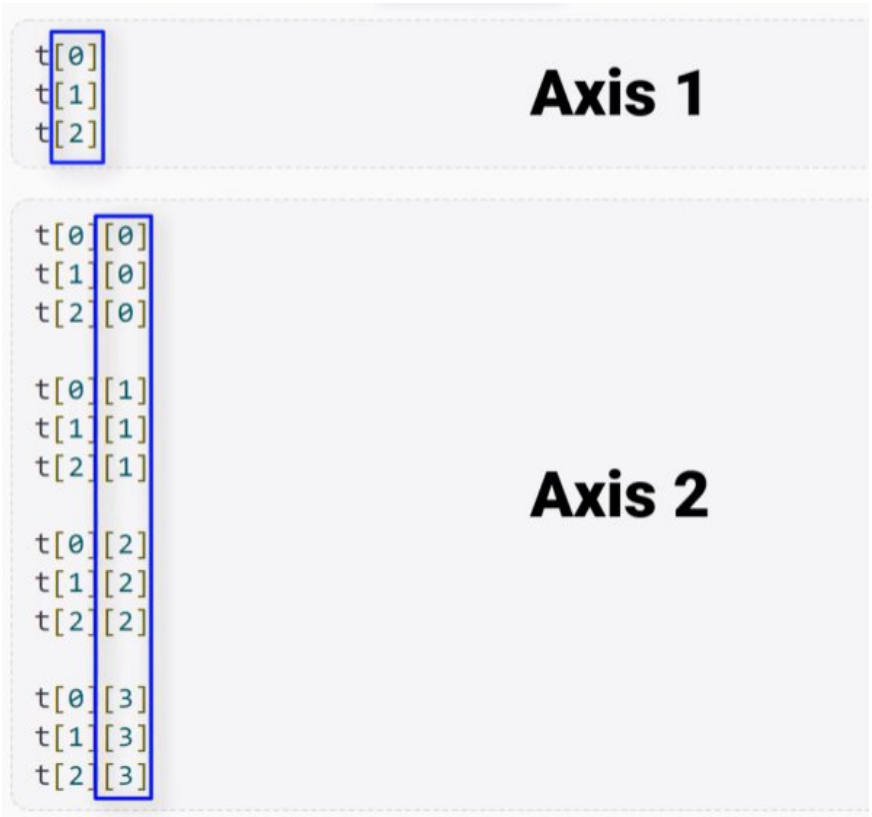
Ранг тензора указывает количество измерений, которые присутствуют в тензоре. Фактически это число осей (или размерностей), которыми характеризуется структура тензора.
Пример. Если есть тензор ранга 2, это означает, что есть матрица, двумерный массив и тензор.
Ранг указывает, сколько индексов нужно для работы с данными (например, для их нарезки или выборки).
Пример. Если тензор имеет ранг 2, для доступа к данным потребуется два индекса:

```
tensor = [
        [1, 2, 3],
        [4, 5, 6]
]

# Индексация по двум измерениям
print(tensor[0][1])  # Вывод: 2
```

Ранг тензора определяет его размерность, это важно при работе с многомерными данными.

Если есть тензор ранга 2, это означает, что в тензоре присутствуют две оси.
Когда в глубоком обучении нужно сослаться на определённое измерение тензора, используем термин «ось».
Ось тензора — это конкретное измерение тензора. Если тензор имеет ранг 2, это означает, что у тензора есть два измерения и две оси (что эквивалентно).



Kaggle / Kaggle Inc.

Элементы тензора располагаются вдоль осей. «Движение» элементов вдоль оси ограничивается длиной этой оси (числом элементов в данном измерении).

Пример. Рассмотрим двумерный тензор (матрицу):

```
tensor = [
        [1, 2, 3],
        [4, 5, 6]
]

# Первая ось (ось 0): строки
print(tensor[0])  # [1, 2, 3]
print(tensor[1])  # [4, 5, 6]

# Вторая ось (ось 1): столбцы
print(tensor[0][1])  # 2 (вторая колонка первой строки)
print(tensor[1][0])  # 4 (первая колонка второй строки)
```

- Ось 0 описывает строки.
- Ось 1 описывает столбцы.

Длина оси — это количество элементов вдоль этой оси.

В этом примере:

- длина оси 0 = 2 (две строки);
- длина оси 1 = 3 (три столбца).

Таким образом, оси тензора определяют его структуру и способ обращения к элементам.

Ранг тензора указывает, сколько осей (или измерений) у тензора.
Длина осей приводит к важному понятию — **форме (shape) тензора**.
Форма тензора определяется длиной его осей. Она указывает количество элементов, доступных вдоль каждой оси, и описывает структуру данных внутри тензора.
Пример:

```
import torch
tensor = torch.tensor([1, 2, 3])  # Одномерный тензор (вектор)
print(tensor.shape)  # Вывод: torch.Size([3])
```

В этом примере:

- у тензора одна ось (ось 0);
- длина этой оси = 3 (три элемента).

Для многомерных тензоров форма описывает размерность по каждой оси:

```
tensor = torch.tensor([[1, 2, 3], [4, 5, 6]])  # Матрица (двумерный тензор)
print(tensor.shape)  # Вывод: torch.Size([2, 3])
```

- Длина оси 0 (число строк) = 2.
- Длина оси 1 (число столбцов) = 3.

Форма и размер в PyTorch

В PyTorch форма (shape) и размер (size) означают одно и то же.

```
print(tensor.size())
# Вывод: torch.Size([2, 3])
```

Переформатирование тензоров позволяет изменять их форму, не изменяя данные:

```
reshaped_tensor = tensor.reshape(3, 2)
# Изменяем форму с (2, 3) на (3, 2)
print(reshaped_tensor)
```

Вывод:

```
tensor([[1, 2],
        [3, 4],
        [5, 6]])
```

Форма тензора предоставляет важную информацию:

- ранг (количество осей);
- размер (длина каждой оси);
- индексы (способ обращения к элементам).

Переформатирование тензоров часто используется для приведения данных к нужной структуре без изменения их содержимого.

Входные данные изображения в тензорах

В нейронных сетях изображения обрабатываются в виде тензоров. Давайте разберём, как формируется тензор для изображения и как интерпретировать его размерности.
Для обработки изображения используется тензор ранга 4, где каждая ось (ранг) представляет определённую характеристику:

```
image_input_size = [Batch size, RGB, Height, Width]
```

- **Batch size** — количество изображений в партии (например, 3 изображения).
- **RGB** — количество цветовых каналов (1 для чёрно-белого изображения или 3 для RGB).
- **Height** — высота изображения (например, 23 пикселя).
- **Width** — ширина изображения (например, 23 пикселя).

```
image_input_size = [3, 1, 23, 23]  # Три чёрно-белых изображения размером 23 x 23 пикселя.
```

Основы работы с тензорами в PyTorch

Тензоры могут быть разного ранга:

- **1D (вектор)** — одномерный тензор,
- **2D (матрица)** — двумерный тензор,
- **nD (многомерный)** — для сложных данных.

Пример:

```
import torch
import numpy as np
# Создание пустого тензора
t = torch.Tensor()
print(type(t))  # Вывод: torch.Tensor
```

PyTorch предоставляет важные атрибуты для анализа тензоров:

```
print(t.dtype)   # Тип данных в тензоре (например, float32).
print(t.device)  # Устройство, на котором размещён тензор (CPU или GPU).
print(t.layout)  # Указывает расположение данных в памяти.
```

Пример:

```
a = torch.Tensor([2, 2])  # Создание тензора
print(a.dtype)   # Вывод: torch.float32
print(a.device)  # Вывод: cpu
```

Для работы с тензорами **на GPU** требуется указать устройство.

Пример:

```
device = torch.device('cuda:0')  # Указание устройства GPU с индексом 0
tensor_on_gpu = torch.tensor([1, 2, 3], device=device)
print(tensor_on_gpu.device)  # Вывод: cuda:0
```

Типы данных

В PyTorch тензоры могут использовать разные типы данных (dtype), которые зависят от используемого устройства (CPU или GPU).
Ниже представлены основные типы данных.

Тип данных	dtype	Тензор (CPU)	Тензор (GPU)
32-битное число с плавающей точкой	torch.float32	torch.FloatTensor	torch.cuda.FloatTensor
64-битное число с плавающей точкой	torch.float64	torch.DoubleTensor	torch.cuda.DoubleTensor
16-битное число с плавающей точкой	torch.float16	torch.HalfTensor	torch.cuda.HalfTensor

8-битное целое число (без знака)	torch.uint8	torch.ByteTensor	torch.cuda.ByteTensor
8-битное целое число (со знаком)	torch.int8	torch.CharTensor	torch.cuda.CharTensor
16-битное целое число (со знаком)	torch.int16	torch.ShortTensor	torch.cuda.ShortTensor
32-битное целое число (со знаком)	torch.int32	torch.IntTensor	torch.cuda.IntTensor
64-битное целое число (со знаком)	torch.int64	torch.LongTensor	torch.cuda.LongTensor

Ошибки при работе с устройствами (CPU и GPU)

PyTorch не позволяет напрямую складывать тензоры, которые находятся на разных устройствах.

Пример:

```
import torch
t1 = torch.Tensor([1, 2, 3])  # Тензор на CPU
t2 = t1.cuda()  # Переносим на GPU
# t1 + t2  # Это вызовет ошибку, так как t1 и t2 находятся на разных устройствах
```

Чтобы операции были успешными, оба тензора должны быть на одном устройстве.

Создание тензоров из данных

PyTorch предоставляет несколько способов создания тензоров.

Пример:

```
import torch
import numpy as np
data = np.array([1, 2, 3])
# 1. С помощью factory-функции `torch.tensor`
a = torch.tensor(data)
print('tensor:', a)
# 2. С помощью конструктора класса `torch.Tensor` (возвращает float)
b = torch.Tensor(data)
print('Tensor:', b)
# 3. С помощью функции `torch.as_tensor`
c = torch.as_tensor(data)
print('torch.as_tensor:', c)
# 4. С помощью функции `torch.from_numpy`
d = torch.from_numpy(data)
```

```
print('torch.from_numpy:', d)
```

Вывод:

```
tensor: tensor([1, 2, 3])
Tensor: tensor([1., 2., 3.])
torch.as_tensor: tensor([1, 2, 3])
torch.from_numpy: tensor([1, 2, 3])
```

Создание пустых тензоров (без исходных данных)

PyTorch позволяет создавать тензоры с заранее заданными значениями:
Пример:

```
# Единичная матрица
a = torch.eye(3)
print('eye:', a)

# Матрица из нулей
b = torch.zeros(2, 2)
print('zeros:', b)

# Матрица из единиц
c = torch.ones(2, 2)
print('ones:', c)

# Матрица случайных чисел
d = torch.rand(2, 2)
print('random:', d)
```

Вывод:

```
eye: tensor([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]])
zeros: tensor([[0., 0.],
               [0., 0.]])
ones: tensor([[1., 1.],
        [1., 1.]])
random: tensor([[0.6641, 0.4687],
               [0.2988, 0.3515]])
```

Тензоры в PyTorch — экземпляры класса torch.Tensor, они могут быть созданы разными способами. Используйте удобный метод в зависимости от источника данных (NumPy, случайные значения или матрицы нулей / единиц). Это позволяет эффективно готовить данные для обучения нейронных сетей.

Операции с тензорами

В PyTorch над тензорами можно производить операции, включая изменение формы, поэлементные вычисления, сравнения, объединение и многое другое. Рассмотрим основные типы операций и их использование.

Форма тензора (shape) определяет его размерности. Важные методы:

- .size() или .shape — узнать размерность;
- .numel() — узнать общее количество элементов;
- .reshape() — изменить форму тензора.

```
t = torch.tensor([
        [1, 2, 3, 3],
        [2, 2, 2, 2],
        [3, 3, 3, 3]
], dtype=torch.float32)

print(t.size())  # torch.Size([3, 4])
print(t.numel())  # 12

# Примеры reshape
print(t.reshape(1, 12))  # [[1., 2., ...]]
print(t.reshape(6, 2))   # [[1., 2.], [3., 3.], ...]
print(t.reshape(2, 2, 3))  # 3-мерный тензор
```

Метод **.squeeze()** удаляет оси размерности 1, например, из [1, 12] делает [12].

Пример:

```
t = t.reshape(1, 12)
print(t.squeeze())  # [1., 2., 3., ...]
```

Поэлементные операции выполняются между соответствующими элементами тензоров одинаковой формы.

Пример:

```
t1 = torch.tensor([[1, 2], [3, 4]], dtype=torch.float32)
t2 = torch.tensor([[9, 8], [7, 6]], dtype=torch.float32)

print(t1 + t2)  # Сложение
print(t1 - t2)  # Вычитание
print(t1 * t2)  # Умножение
```

Broadcasting позволяет проводить операции между тензорами разного размера, автоматически расширяя их до совместимой формы.

Пример:

```
t1 = torch.tensor([[1, 2], [3, 4]], dtype=torch.float32)
print(t1 + 2)  # [[3., 4.], [5., 6.]] (2 "расширяется" до [[2, 2], [2, 2]])

t2 = torch.tensor([2, 4], dtype=torch.float32)
```

```
print(t1 + t2)  # [[3., 6.], [5., 8.]]
```
Сравнения возвращают булевый тензор, где каждый элемент — результат операции.
Пример:
```
t = torch.tensor([
        [0, 5, 0],
        [6, 0, 7],
        [0, 8, 0]
], dtype=torch.float32)

print(t.eq(0))  # Равно 0
print(t.gt(5))  # Больше 5
print(t.le(7))  # Меньше или равно 7
```

PyTorch поддерживает такие операции, как модуль числа, квадратный корень, отрицание.
```
print(t.abs())   # Модуль
print(t.sqrt())  # Квадратный корень
print(t.neg())   # Отрицание
```

Объединение тензоров
Конкатенация — метод .cat() объединяет тензоры вдоль указанной оси.

```
t1 = torch.tensor([[1, 2], [1, 2]])
t2 = torch.tensor([[3, 4], [3, 4]])
print(torch.cat((t1, t2), dim=1))  # Объединение по оси 1
```

Стекирование — метод .stack() создаёт дополнительное измерение.

```
t1 = torch.tensor([[1, 1], [1, 1]])
t2 = torch.tensor([[2, 2], [2, 2]])
t3 = torch.tensor([[3, 3], [3, 3]])

t = torch.stack((t1, t2, t3))
print(t.size())  # torch.Size([3, 2, 2])
```
Индексация позволяет извлекать данные по определённым координатам.

```
t = torch.tensor([
        [[1, 1], [2, 2]],
        [[3, 3], [4, 4]],
        [[5, 5], [6, 6]]
])

print(t[0])      # Первый «батч»
print(t[0][0])   # Первый канал
print(t[0][0][0])  # Первый элемент
```

«Разворачивание» тензоров (Flatten)
Метод .flatten() превращает тензор в одномерный.

```
print(t.flatten())  # Все элементы в одном измерении
print(t.flatten(start_dim=1))  # «Разворачивание» по осям, начиная с оси 1
```

Вычисление градиента

Вычисление градиента играет ключевую роль в нейросетях. Разберемся как происходит процесс и как он связан с тензорами

Тензор в контексте нейронных сетей и глубокого обучения обозначает данные, то есть многомерные массивы, или тензоры, которые перемещаются через вычислительный граф модели.

Вычислительный граф представляет архитектуру нейронной сети как набор связанных операций. Вершины этого графа — это операции, такие как матричное умножение, сложение или применение нелинейностей (например, ReLU), а рёбра указывают направление потока данных, то есть как результаты одной операции передаются на вход следующей. Входы и выходы этих операций — это и есть тензоры. В каждом узле графа тензоры обрабатываются или передаются дальше по вычислительному графу модели от входного слоя к выходному.

Слово «тензор» подчёркивает, что данные в нейронных сетях — это не просто одномерные векторы или двумерные матрицы, а структуры, у которых может быть произвольная размерность. Это удобно для представления сложных данных, например изображений, звуковых сигналов или текстовых последовательностей.

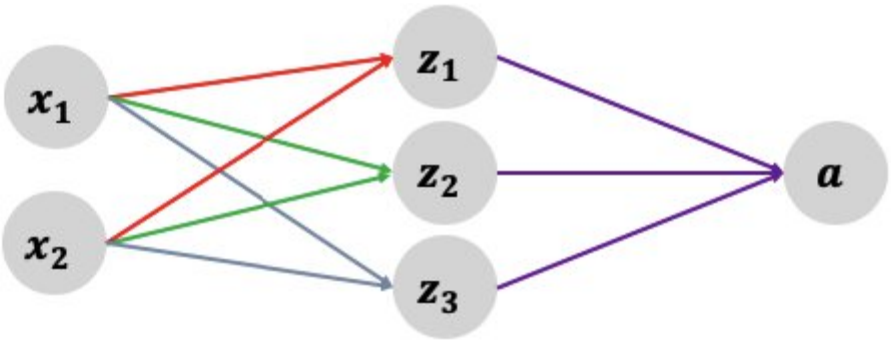
Ключевая особенность PyTorch— это динамический граф вычислений. Граф вычислений создаётся прямо во время выполнения программы. Это делает его интуитивно понятным, гибким и удобным для отладки программы.
Благодаря динамическому построению графа, отладка в PyTorch осуществляется проще. Разработчики могут запускать код по шагам и визуализировать промежуточные результаты даже с помощью простой команды print.

Посмотрим на нейронную сеть как на граф вычислений. Такой подход помогает визуализировать и структурировать вычисления, происходящие при прохождении данных через сеть.

Что такое граф вычислений

Граф вычислений — это структура, состоящая из **вершин** и **рёбер**.

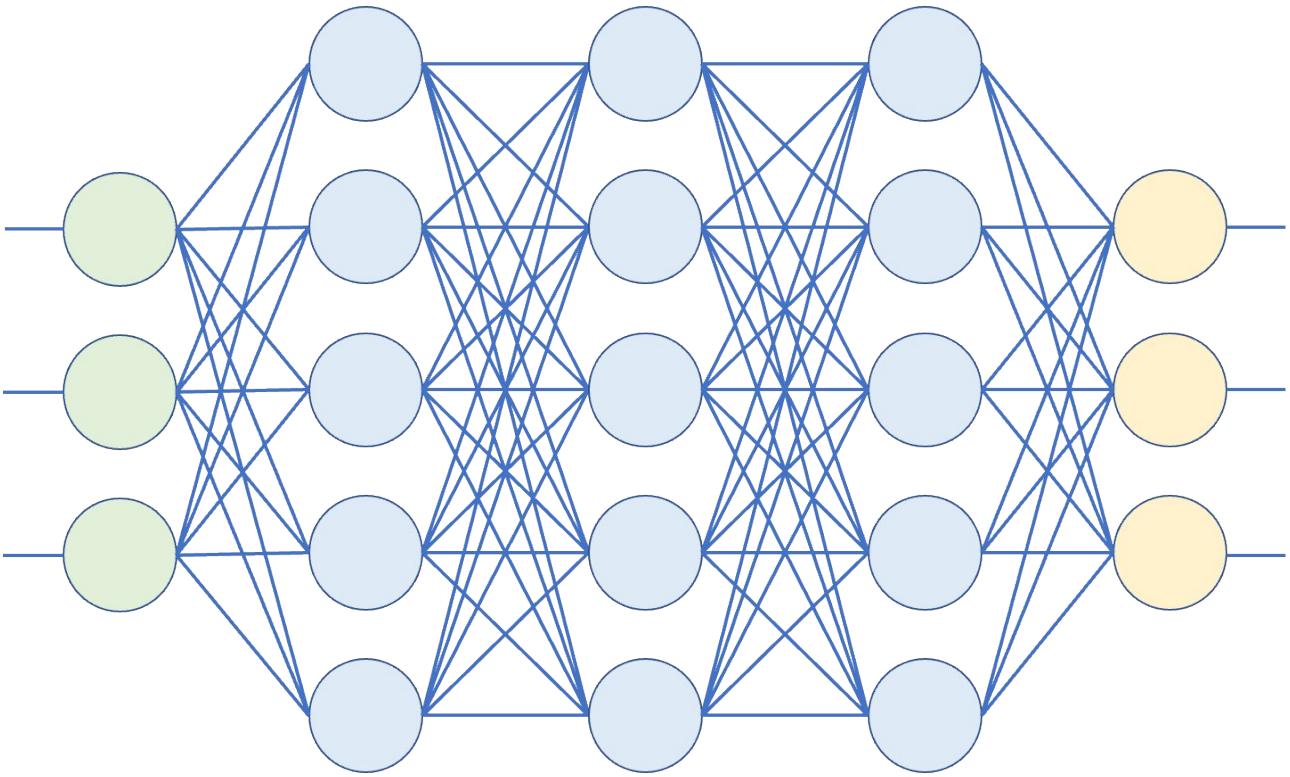
- **Вершины** (узлы) — вычисляемые значения. Например, выходы нейронов.
- **Рёбра** — направленные стрелки, показывающие, какие значения нужны для вычисления других значений. Ребро идёт от $\mathbf{x_1}$ к $\mathbf{z_1}$ в случае, если для того, чтобы вычислить z_1, необходим x_1. Это граф зависимости между вычисляемыми значениями.



Итак, граф соответствует комбинации функций. Такой граф называют **многослойным перцептроном**, который включает в себя следующие слои:

- **входной слой** содержит в себе признаки;
- **скрытый слой** — промежуточные вычисления, где каждая вершина представляет собой результат линейной комбинации входных признаков, к которой применена нелинейная функция.
- **Выходной слой** — результат финального вычисления, то есть предсказание.

Многослойный перцпетрон – простейшее представление нейросети.



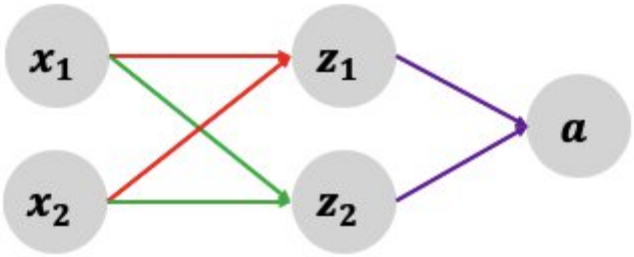
По структуре на картинке видно, что после преобразования входов через скрытые слои информация от исходных признаков **x** передаётся через узлы **z** и **прямых связей между входом и выходом больше нет**. Это упрощает и структурирует модель.

Нелинейности в нейронах

Почему важны нелинейности

Если бы у нас не было нелинейных функций, например сигмоида, сеть была бы ограничена в своей выразительности. Давайте на примере рассмотрим, что это значит.

Возьмём следующий граф:



Если убрать нелинейности, модель станет очень простой: можно подставить выражения для $\mathbf{z_1}$ и $\mathbf{z_2}$ в модель **a**.

$$z_1 = w_{1,1}x_1 + w_{2,1}x_2$$

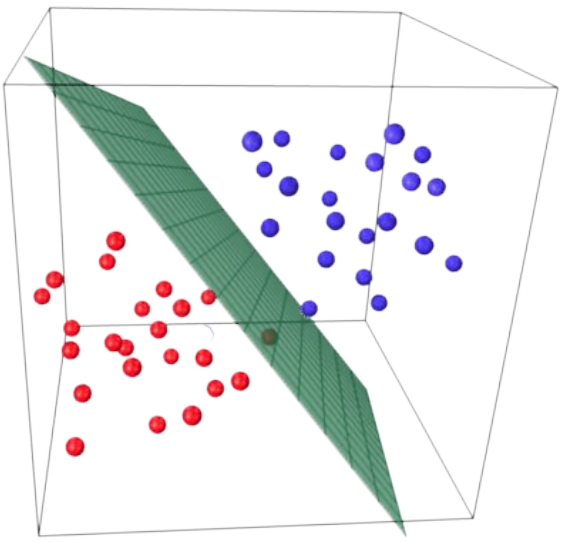
$$z_2 = w_{1,2}x_1 + w_{2,2}x_2$$

$$a = w_1z_1 + w_2z_2$$

Если раскрыть скобки и привести подобные слагаемые, получится линейная комбинация $\mathbf{x_1}$ и $\mathbf{x_2}$. При этом модель сложнее не становится.

$$a = (w_1w_{1,1} + w_2w_{1,2})x_1 + (w_1w_{2,1} + w_2w_{2,2})x_2)$$

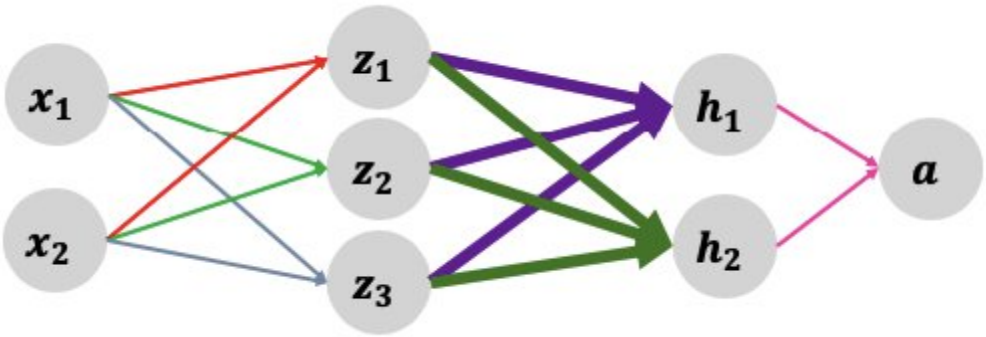
В результате даже через несколько скрытых слоёв без нелинейностей мы снова получим **линейную модель**, которая не может эффективно решать сложные задачи, так как способна только разделить пространство с помощью линии или гиперплоскости.



Нелинейности позволяют сети обучаться **сложным зависимостям**, превращая линейные комбинации на каждом слое в более гибкие преобразования. Таким образом, сеть может обучаться решать более сложные задачи классификации и регрессии.

MLP

MLP — это простейший пример нейросети.



Получившаяся структура — это **многослойный перцептрон (MLP)**. Он состоит:

- из **входного слоя**, передающего исходные признаки;
- одного или нескольких **скрытых слоёв**, где каждый нейрон выполняет линейное и нелинейное преобразования (например, сигмоид).
- **выходного слоя**, который выдаёт финальное предсказание.

Использование графов вычислений позволяет эффективно описать, как данные проходят через сеть и как происходит **обратное распространение ошибки (backpropagation)** для обновления весов.

Итак, многослойный перцептрон (MLP) может включать в себя несколько скрытых слоёв. Их **добавление** позволяет модели постепенно переходить в новые, более сложные **пространства признаков**. Это приводит к тому, что в каждом следующем слое задача классификации или регрессии становится более простой для **решения**, так как на каждом уровне сеть выделяет более высокоуровневые и полезные признаки.

В чём польза многослойности

Каждый слой в MLP выполняет нелинейные преобразования, которые представляют собой новые координаты. Чем больше слоёв, тем сложнее зависимости и паттерны, которые модель может захватить и описать. Так, наслоение помогает MLP решать более трудные задачи, и в итоге даже нелинейные задачи (когда данные линейно неразделимы) могут быть решены **линейной моделью на финальном слое** за счёт преобразований, сделанных предыдущими слоями.

Полносвязные слои (Dense Layers)

В MLP у **каждого слоя** есть название: **плотный** (Dense), **полносвязанный** (Fully Connected) или **коннектор**. Полносвязный слой предполагает, что каждый нейрон предыдущего слоя связан со **всеми нейронами** следующего. Это даёт возможность каждому нейрону видеть все входные признаки, что помогает ему учитывать полную информацию на каждом шаге.

Архитектура MLP

Архитектура MLP — это набор параметров, которые мы задаём вручную, чтобы определить структуру модели. Ключевые параметры включают:

- **количество слоёв** определяет, как много промежуточных уровней используется в модели;
- **количество нейронов в каждом слое** определяет размер слоя и влияет на выразительность модели;
- **функции активации** определяют тип нелинейности, добавляемой к модели (сигмоида, ReLU и другое). Выбор функции активации влияет на то, какие зависимости может захватить каждый слой.

Эти параметры позволяют управлять **сложностью** и **возможностями** модели. Чем больше слоёв и нейронов, тем сложнее задачи, которые может решать MLP.

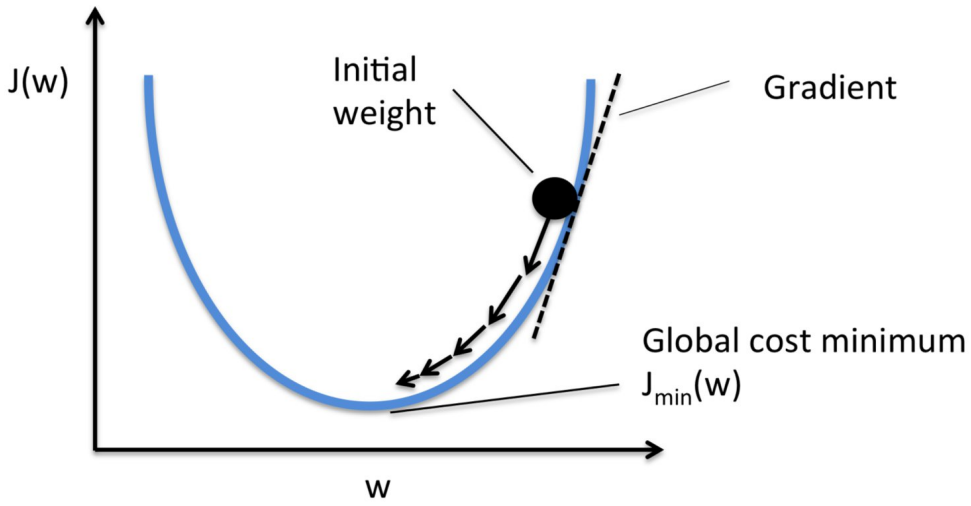
Функция потерь

Для обучения нейронных сетей используется метод градиентного спуска. В его основе лежит минимизация функции потерь, которая показывает, насколько далеки предсказанные результаты от истинных.

Как правило, функция потерь дифференцируемая, то есть от неё можно вычислить производную в любой точке. Это позволяет определить, какие параметры модели и насколько нужно корректировать.
Производная функции потерь в точке показывает направление наибольшего роста функции, а её противоположное направление указывает на уменьшение. Таким образом, при каждом шаге градиентного спуска параметры модели немного изменяются, что исправить ошибку.

Допустим, у нас есть параметр *w*, который нужно оптимизировать. Чтобы сделать это, можно начать оптимизацию с любой точки (initial weight), посчитав в ней производную. В нашем примере это тангенс угла наклона касательной.

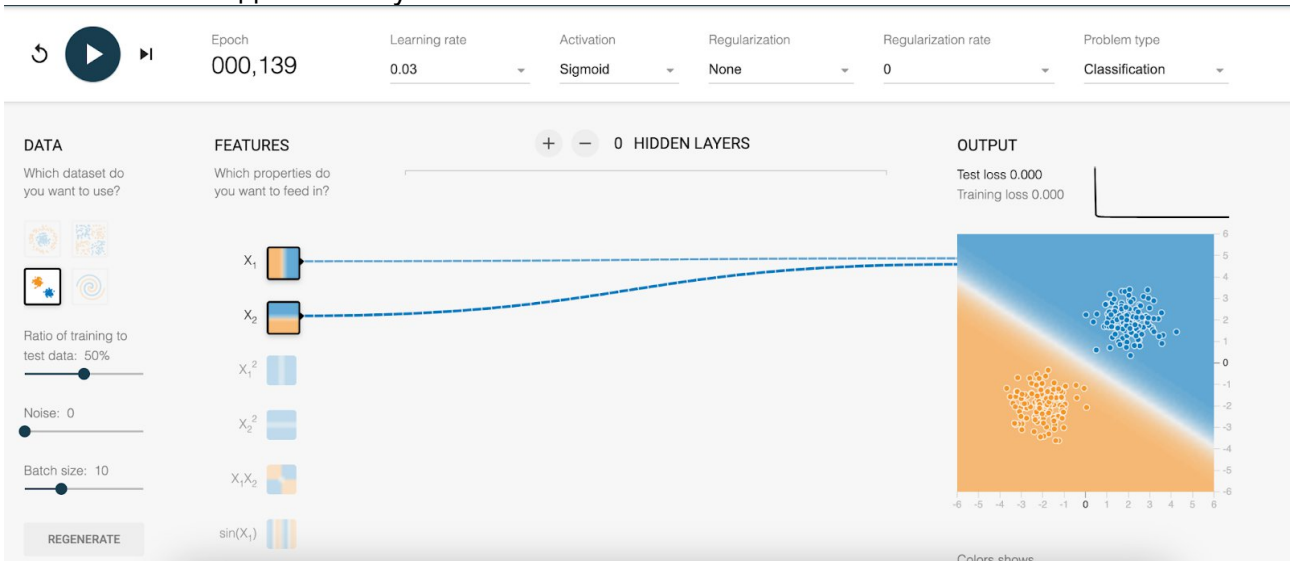
Визуализация процесса: представление о тангенсе и минимизации



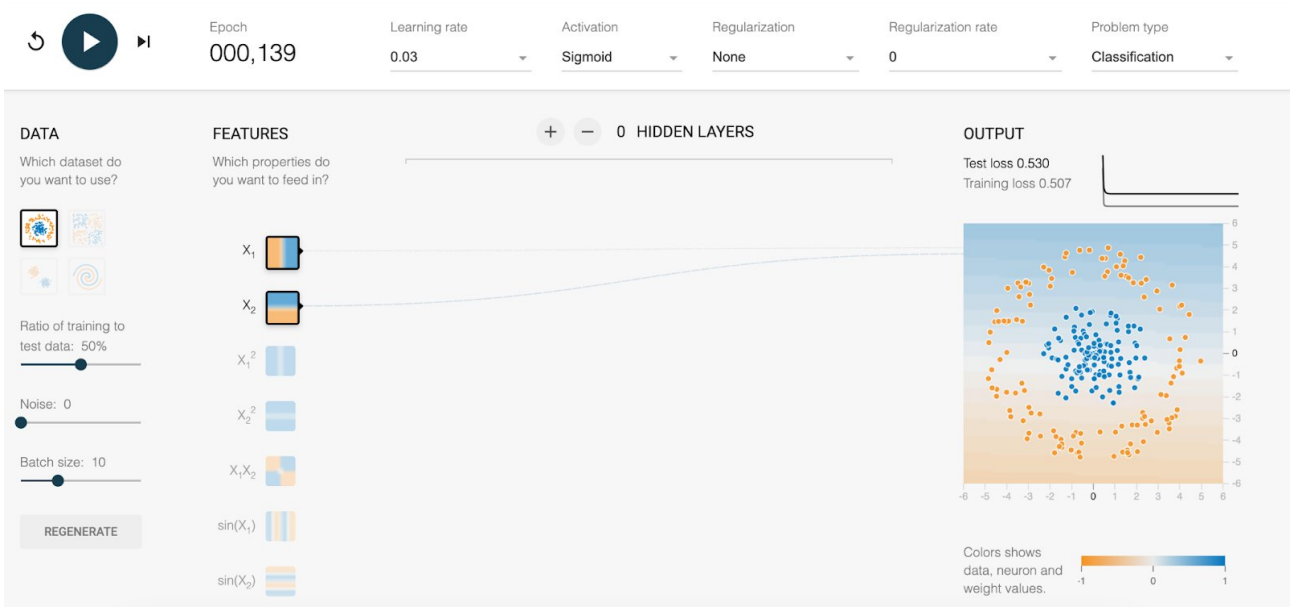
Представьте функцию потерь в виде холма, а параметры — в виде мяча на нём. Если мяч оставить на вершине, он начнёт катиться вниз, выбирая путь наименьшего сопротивления.
На каждом шаге градиентного спуска мы считаем производную в текущей точке и смещаемся в сторону, противоположную наклону линии тангенса, постепенно исправляя ошибку. Повторяя этот процесс, мы приближаемся к минимуму функции потерь.

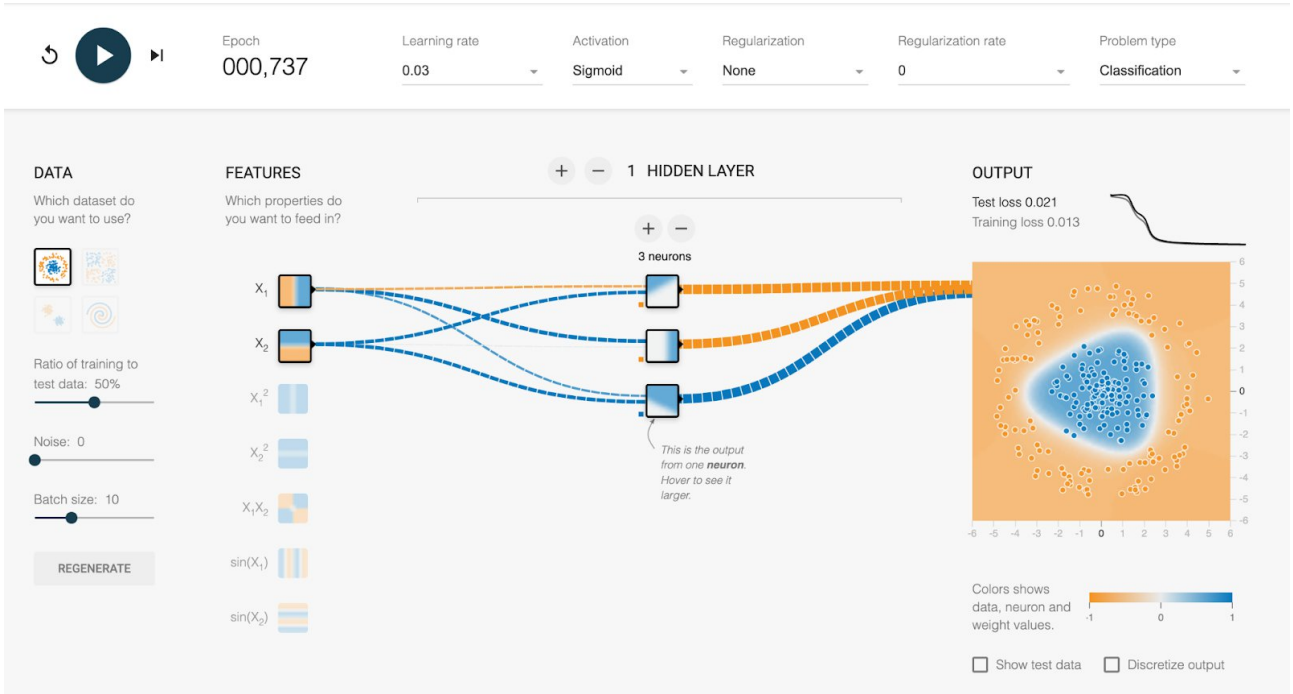
Визуализировать процесс градиентного спуска можно с помощью нейросетевых инструментов, таких как Playground. В нём можно увидеть, как меняются веса сети и как они влияют на результат. Также на платформе можно варьировать количество скрытых слоёв, нейронов и фичей, чтобы наблюдать, как сеть учится различать данные.

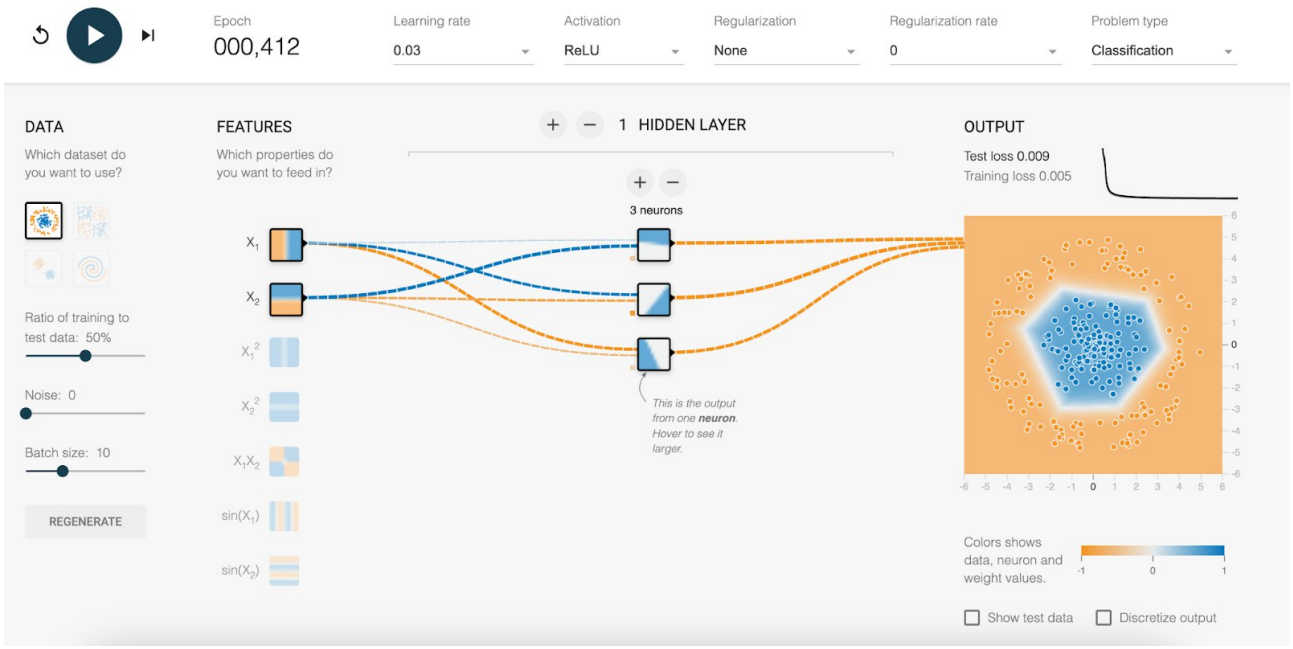
Для простых, линейно разделимых данных, которым достаточно одной прямой, линейные модели (например, один нейрон) находят разделяющую линию, минимизирующую функцию потерь. В них градиентный спуск быстро сходится к одному оптимальному решению независимо от начальных значений параметров, так как есть всего один оптимум.



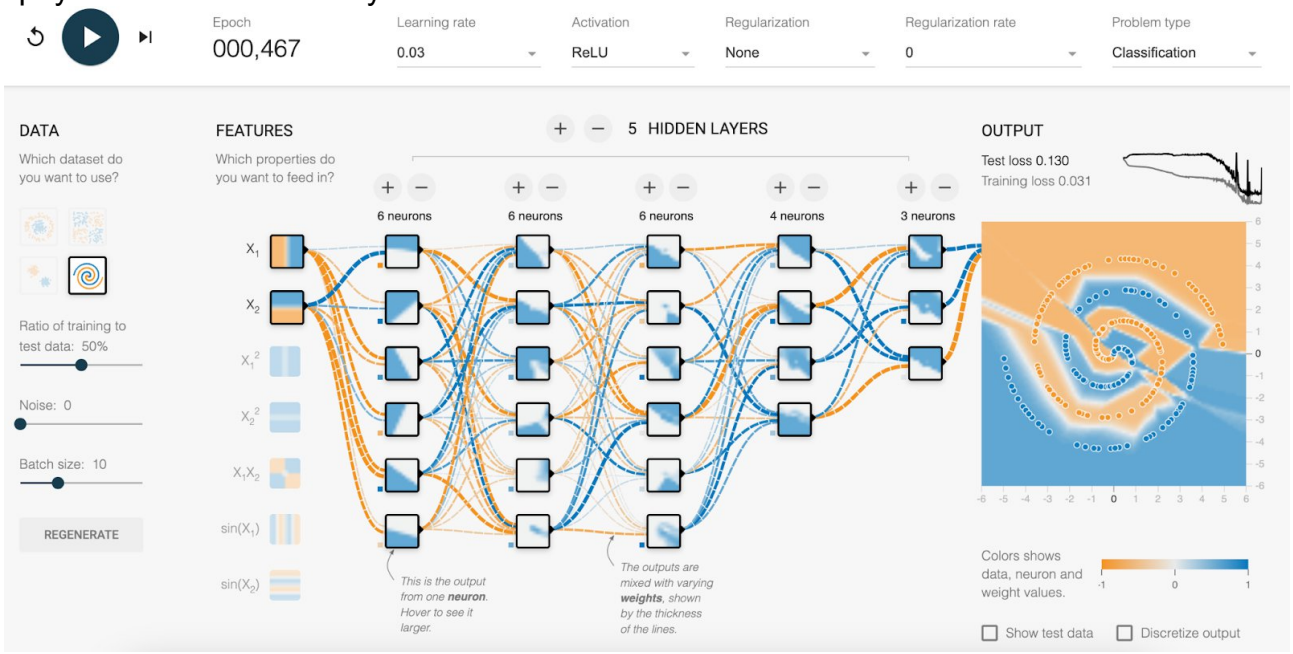
При работе с более сложными, нелинейными данными одной линии мало. В таких случаях сеть должна включать в себя ещё скрытые слои, чтобы научиться комбинировать несколько линейных моделей, которые в совокупности образуют более сложные границы разделения.







В примере с треугольником для создания сложных границ мы можем добавить несколько скрытых слоёв. Эти слои трансформируют данные, позволяя сети находить нужные закономерности и улавливать сложные формы, такие как треугольники или шестиугольники.



Для решения сложных задач и придания модели нелинейности используются функции активации, такие как ReLU, сигмоида или tanh. Например, функция ReLU ускоряет процесс обучения и увеличивает вероятность нахождения оптимального решения, что особенно важно для глубоких сетей.

Несмотря на теоретические сложности, градиентный спуск показал высокую эффективность на практике и успешно применяется в обучении сложных моделей.

Детали градиентного спуска: цепное правило

Вы уже знаете, как дифференцировать функцию:

$\frac{dx^2}{dx} = 2x$, $\frac{de^x}{dx} = e^x$, $\frac{dln(x)}{dx} = \frac{1}{x}$.

Предположим, что мы берём сложную функцию p, которая зависит от z_1 и z_2. А они, в свою очередь, зависят от x_1 и x_2.

$$z_1 = z_1(x_1, x_2)$$
$$z_2 = z_2(x_1, x_2)$$
$$p = p(z_1, z_2)$$

В этом случае мы получим цепное правило: нужно вычислить производную p по z и умножить её на производную z по x.
Как найти производную p по x_1:

1) вычислить производную p по z_1 и умножить её на производную z_1 по x_1;
2) вычислить производную p по z_2 и умножить её на производную z_2 по x_1;
3) сложить произведения.

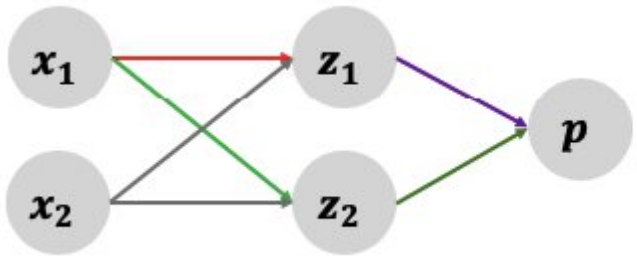
$$\frac{\partial p}{\partial x_1} = \frac{\partial p}{\partial z_1}\frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial z_2}\frac{\partial z_2}{\partial x_1}$$

Пример для h(x) = f(x)g(x):

$$\frac{\partial h}{\partial x} = \frac{\partial h}{\partial f}\frac{\partial f}{\partial x} + \frac{\partial h}{\partial g}\frac{\partial g}{\partial x} = g\frac{\partial f}{\partial x} + f\frac{\partial g}{\partial x}$$

Граф вычислений

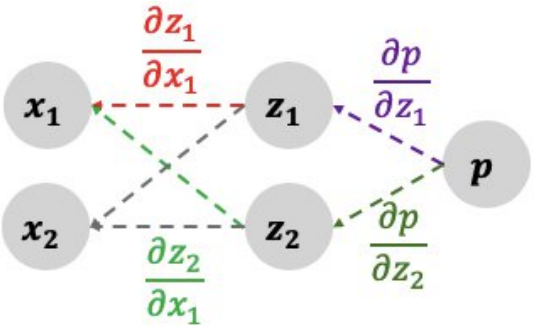
Итак для простой функции можно построить **граф вычислений**, который представляет собой стандартную структуру с направленными рёбрами, показывающими зависимости между переменными.



$$z_1 = z_1(x_1, x_2)$$
$$z_2 = z_2(x_1, x_2)$$
$$p = p(z_1, z_2)$$

Помимо этого, можно построить **граф производных**, который похож на граф вычислений, но все стрелки направлены в противоположную сторону. Этот граф используется для вычисления производных.

В графе производных каждому ребру назначена производная. Например, если ребро идёт от p к z_1, на нём будет указана производная $\frac{\partial p}{\partial z_1}$.



$$\frac{\partial p}{\partial x_1} = \frac{\partial p}{\partial z_1}\frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial z_2}\frac{\partial z_2}{\partial x_1}$$

Применение цепного правила на более глубоком графе

Рассмотрим более сложный случай с несколькими скрытыми слоями. Здесь цепное правило применяется по аналогии с простым случаем, но включает в себя дополнительные промежуточные вычисления.

Например, если нужно вычислить производную $\frac{\partial p}{\partial x_1}$, мы сначала проходим через промежуточные переменные h_1 и h_2, а затем применяем к каждой из них цепное правило.

$$\frac{\partial p}{\partial x_1} = \frac{\partial p}{\partial h_1}\frac{\partial h_1}{\partial x_1} + \frac{\partial p}{\partial h_2}\frac{\partial h_2}{\partial x_1}$$

$$\frac{\partial p}{\partial x_1} = \frac{\partial p}{\partial h_1}\frac{\partial h_1}{\partial x_1} + \frac{\partial p}{\partial h_2}\frac{\partial h_2}{\partial x_1}$$

$$\frac{\partial h_1}{\partial x_1} = \frac{\partial h_1}{\partial z_1}\frac{\partial z_1}{\partial x_1} + \frac{\partial h_1}{\partial z_2}\frac{\partial z_2}{\partial x_1} \qquad \frac{\partial h_2}{\partial x_1} = \frac{\partial h_2}{\partial z_1}\frac{\partial z_1}{\partial x_1} + \frac{\partial h_2}{\partial z_2}\frac{\partial z_2}{\partial x_1}$$

$$z_1 = z_1(x_1, x_2) \qquad h_1 = h_1(z_1, z_2)$$
$$z_2 = z_2(x_1, x_2) \qquad h_2 = h_2(z_1, z_2)$$

Производные $\frac{\partial h_1}{\partial x_1}$ и $\frac{\partial h_2}{\partial x_1}$ также вычисляются с использованием цепного правила.

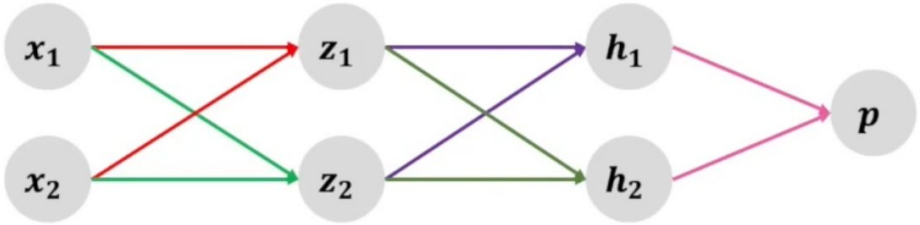
$$\frac{\partial p}{\partial x_1} = \frac{\partial p}{\partial h_1}\frac{\partial h_1}{\partial x_1} + \frac{\partial p}{\partial h_2}\frac{\partial h_2}{\partial x_1}$$

$$\frac{\partial h_1}{\partial x_1} = \frac{\partial h_1}{\partial z_1}\frac{\partial z_1}{\partial x_1} + \frac{\partial h_1}{\partial z_2}\frac{\partial z_2}{\partial x_1} \qquad \frac{\partial h_2}{\partial x_1} = \frac{\partial h_2}{\partial z_1}\frac{\partial z_1}{\partial x_1} + \frac{\partial h_2}{\partial z_2}\frac{\partial z_2}{\partial x_1}$$

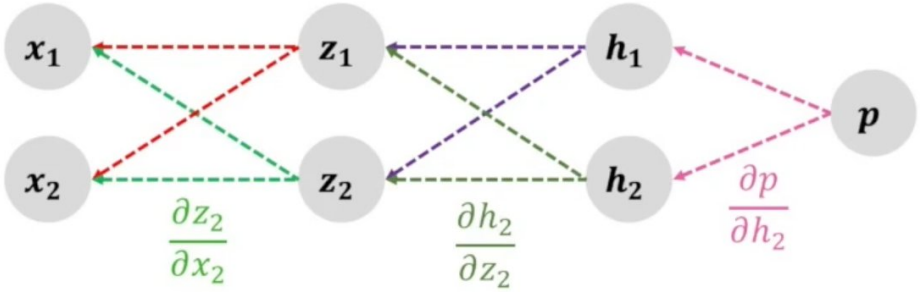
$$\frac{\partial p}{\partial x_1} = \frac{\partial p}{\partial h_1}\left(\frac{\partial h_1}{\partial z_1}\frac{\partial z_1}{\partial x_1} + \frac{\partial h_1}{\partial z_2}\frac{\partial z_2}{\partial x_1}\right) + \frac{\partial p}{\partial h_2}\left(\frac{\partial h_2}{\partial z_1}\frac{\partial z_1}{\partial x_1} + \frac{\partial h_2}{\partial z_2}\frac{\partial z_2}{\partial x_1}\right)$$

$$\frac{\partial p}{\partial x_1} = \frac{\partial p}{\partial h_1}\frac{\partial h_1}{\partial z_1}\frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial h_1}\frac{\partial h_1}{\partial z_2}\frac{\partial z_2}{\partial x_1} + \frac{\partial p}{\partial h_2}\frac{\partial h_2}{\partial z_1}\frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial h_2}\frac{\partial h_2}{\partial z_2}\frac{\partial z_2}{\partial x_1}$$

Каждый путь в графе вычислений соответствует одному слагаемому в разложении по цепному правилу.



$$\frac{\partial p}{\partial x_1} = \frac{\partial p}{\partial h_1}\frac{\partial h_1}{\partial z_1}\frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial h_1}\frac{\partial h_1}{\partial z_2}\frac{\partial z_2}{\partial x_1} + \frac{\partial p}{\partial h_2}\frac{\partial h_2}{\partial z_1}\frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial h_2}\frac{\partial h_2}{\partial z_2}\frac{\partial z_2}{\partial x_1}$$



В графе производных стрелочки направлены в другую сторону, что позволяет работать с вычислением производных через обратное распространение.

Общий алгоритм
Вот как выглядит алгоритм для вычисления производных в нейронных сетях:

1) построение графа вычислений;
2) построение графа производных с обратными стрелками;
3) применение цепного правила для вычисления производных по всем путям графа;
4) суммирование результатов для получения конечной производной.

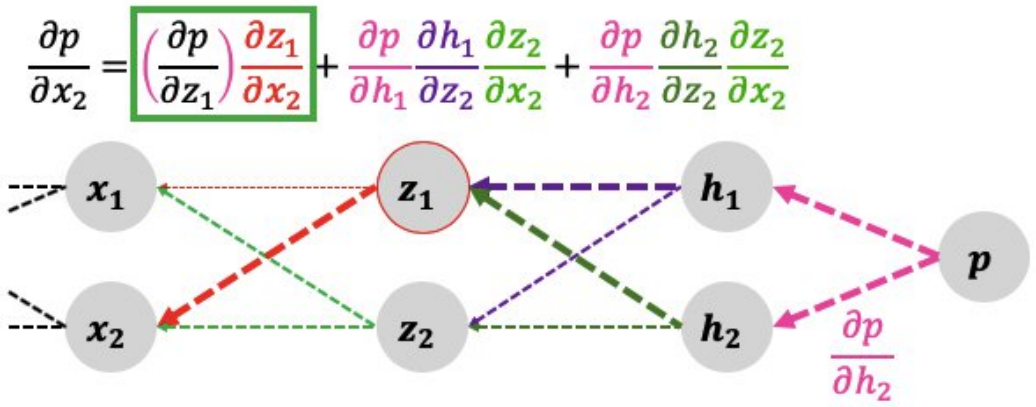
Этот подход работает на любых уровнях сложности сети. Он применим как для простых сетей с одним скрытым слоем, так и для более глубоких моделей с несколькими слоями.

Таким образом, вычисление производных в нейронных сетях — это процесс, который можно автоматизировать с помощью графов. Это упрощает обратное распространение ошибок и ускоряет обучение модели.
Для настройки нейронной сети необходимо посчитать производные функции потерь по всем параметрам модели. Чтобы это выполнить, нужно пройти по графу вычислений производных с помощью оптимального метода — **обратного распространения ошибки** (backpropagation), который упрощает и ускоряет вычисления.

Принципы обратного распространения ошибки

- Чтобы найти производные, нужно пройти по графу обратного распространения (графу производных).
- Многие пути к параметрам модели проходят через одни и те же промежуточные вершины, что позволяет повторно использовать уже вычисленные производные. Например, для поиска пути из L (потери) в x_2

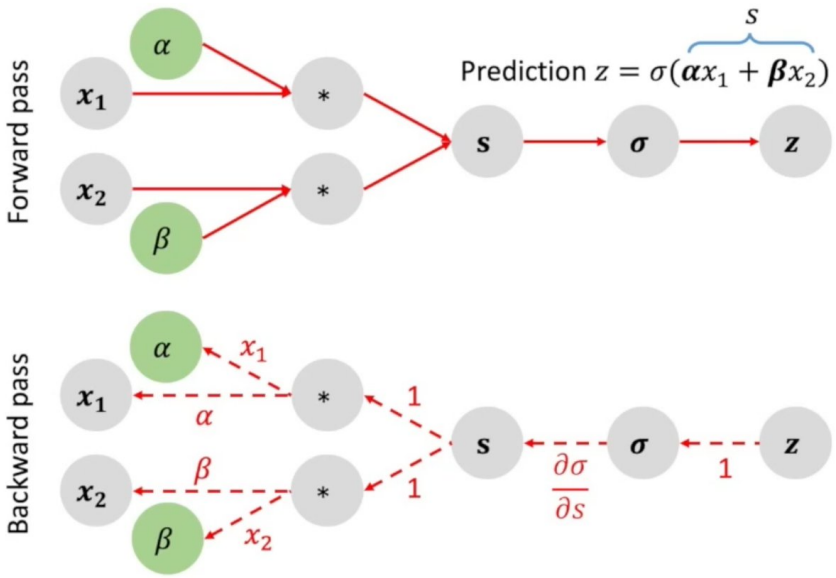
можно использовать промежуточное значение производной $\frac{\partial L}{\partial z_1}$ и умножить его на $\frac{\partial z_1}{\partial x_2}$, тем самым избегая дополнительных вычислений.



Алгоритм обратного распространения

Обратное распространение выполняется справа налево, потому что при прохождении через каждый слой нужно произвести одно умножение, чтобы получить следующую производную.
Процесс включает два прохода по графу:

- **прямой проход** (для вычисления значений узлов графа) необходим, чтобы узнать значения аргументов, которые потребуются для обратного прохода;
- **обратный проход** (собственно backpropagation) нужен для вычисления производных всех параметров с использованием значений, полученных в прямом проходе.



Необходимость двойного прохода

Прямой проход позволяет вычислить значения аргументов для функции и её производных в конкретных точках, так как градиент рассчитывается для текущего состояния параметров.

Например, для функции активации, допустим, сигмоидальной, мы рассчитываем производную в текущей точке *s* — её аргументе, который определяет значение производной.

Инициализация весов и проблема симметрии

Неправильная инициализация: если мы инициализируем все веса нулями, градиенты весов на одном слое будут одинаковыми. Это вызовет симметричное обновление весов и приведёт к тому, что нейроны будут делать идентичные предсказания.

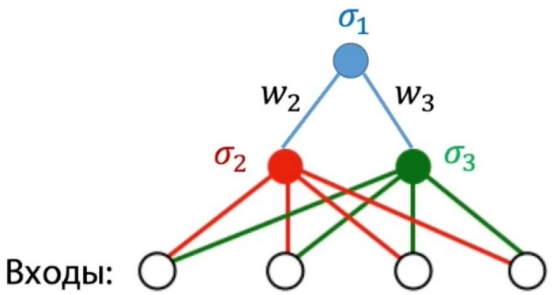
Проблема симметрии: если веса нейронов обновляются одинаково, модель не сможет обучить сложные комбинации признаков, так как нейроны будут обучаться одинаковым паттернам.

Чтобы избежать **разрушения симметрии**, нужно инициализировать веса случайным образом, добавив шум. Это позволяет нейронам обучаться разным задачам, приводя к более сложной модели и лучшим результатам.

Пример симметрии и инициализации весов

Рассмотрим сеть с четырьмя входами, двумя скрытыми нейронами σ_2 и σ_3 и одним выходным нейроном.

Если веса $\mathcal{W}_2$ и $\mathcal{W}_3$ будут нулевыми, оба нейрона σ_2 и σ_3 сделают одинаковые предсказания. Тогда производные по весам $\mathcal{W}_2$ и $\mathcal{W}_3$ также будут одинаковыми и обновления параметров произойдут симметрично, из-за чего модель не сможет обучиться сложным задачам.



$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial \sigma_1}\sigma_1(1-\sigma_1)\sigma_2$$

$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial \sigma_1}\sigma_1(1-\sigma_1)\sigma_3$$

Решение: необходимо использовать случайную инициализацию весов, чтобы каждый нейрон решал свою задачу, разбивая симметрию.

В PyTorch есть инструмент **Autograd**, который автоматически вычисляет градиенты, необходимые для обучения нейронных сетей.

torch.autograd — это механизм автоматического дифференцирования в PyTorch, который позволяет вычислять градиенты и производные функций автоматически. Он особенно полезен в обучении нейросетей, так как упрощает процесс обратного распространения ошибки (backpropagation).

Основные концепции torch.autograd:

1. **Тензоры с флагом requires_grad=True**
   PyTorch отслеживает все операции с тензорами, у которых requires_grad=True, чтобы затем можно было вычислить их градиенты.

2. **Градиенты (.grad)**
   После выполнения backward() градиенты будут сохранены в .grad каждого тензора, у которого requires_grad=True.
3. **Функция backward()**
   Вызывается для вычисления градиентов. Обычно используется в обучении нейросетей для обновления весов.
4. **Граф вычислений (Computational Graph)**
   PyTorch строит динамический граф вычислений, который записывает все операции с тензорами. Этот граф используется для обратного распространения ошибки.
5. **Контекст torch.no_grad()**
   Позволяет отключить вычисление градиентов, что полезно при инференсе (предсказаниях), когда градиенты не нужны.

```
# Создаем тензоры с отслеживанием градиентов
x = torch.tensor(2.0, requires_grad=True)
y = x**2  # Операция возведения в квадрат
# Вычисляем градиент y по x
y.backward()
# Градиент dy/dx = 2 * x
print(x.grad)  # Выведет: tensor(4.)
```

Заключение

Давайте подведем итоги нашей вводной лекции по основам PyTorch, где мы рассмотрели ключевые компоненты — тензоры и механизм автоматического дифференцирования (autograd).

Сегодня мы узнали, что:

1. **Тензоры** — это основные строительные блоки PyTorch, аналогичные массивам NumPy, но с возможностью использования GPU для ускорения вычислений. Мы рассмотрели, как создавать тензоры, выполнять с ними базовые операции и конвертировать данные между PyTorch и NumPy.

2. **Autograd** — это мощный инструмент PyTorch, который автоматически вычисляет градиенты, необходимые для обучения нейронных сетей.

3. Изучили, что такое вычислительные графы и как вычисляется градиент в нейросетях.

Освоение этих фундаментальных понятий открывает перед вами возможности для создания и обучения сложных нейронных сетей. Понимание работы с тензорами и минимизацией функции потерь является основой для более сложных тем, таких как создание архитектур нейронных сетей, настройка гиперпараметров и оптимизация моделей.

В следующих лекциях мы углубимся в практическое применение этих знаний, изучая, как создавать различные типы нейронных сетей с использованием PyTorch, как задавать функции потерь и оптимизаторы, и как настраивать процесс обучения для достижения наилучших результатов.