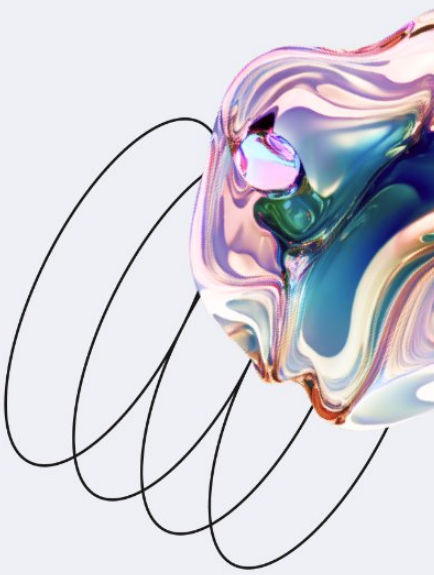
GeekBrains



Способы работы с данными с помощью Pytorch

Оглавление

Словарь терминов

Computer Vision (CV) — это область искусственного интеллекта, которая занимается обработкой и анализом изображений и видео.

Dataset — это коллекция данных, используемых для обучения и тестирования моделей машинного обучения.

DataLoader — это инструмент в PyTorch, который позволяет загружать данные из датасета пакетами (batches) и выполнять их предобработку.

Natural Language Processing (NLP) — это область искусственного интеллекта, которая занимается обработкой и анализом текстовых данных

Аугментация данных — это процесс создания измененных версий исходных данных (например, изображений или текста) для увеличения объема обучающего набора.

Батчи — это подмножества данных, на которые разбивается обучающий набор для более эффективного обучения модели.

Векторизация — это процесс преобразования данных (например, текста) в числовые представления (векторы), которые могут быть использованы в моделях машинного обучения.

Лейблы (метки) — это выходные значения, которые модель должна предсказать.

Недообучение — это явление, при котором модель не может достаточно хорошо обучиться на обучающих данных.

Переобучение — это явление, при котором модель слишком хорошо обучается на обучающих данных, но плохо обобщает знания на новых данных.

Признаки (Features) — это входные данные, которые используются для обучения модели.

Сверточные нейронные сети (CNN) — это тип нейронных сетей, специально разработанный для работы с изображениями.

Токенизация — это процесс разбиения текста на более мелкие единицы, называемые токенами (например, слова, подслова или символы).

Введение

Работа с данными является ключевым этапом в любом проекте машинного обучения. Качество и подготовка данных напрямую влияют на эффективность модели. PyTorch предоставляет мощные инструменты для загрузки, предобработки и трансформации данных, что значительно упрощает работу с различными типами информации, такими как изображения, тексты или числовые данные.

На этом занятии мы подробно рассмотрим основные аспекты работы с данными в PyTorch. Мы начнем с изучения DataLoader — инструмента, который позволяет эффективно загружать и обрабатывать данные пакетами (batches). Затем перейдем к предобработке данных и их трансформациям, которые помогают привести данные к формату, пригодному для обучения модели. Уделим внимание работе с изображениями и текстами, так как эти типы данных требуют специфической обработки. Наконец, мы разберем, как организовать наборы данных и выполнить их разделение на обучающую, валидационную и тестовую выборки.

Загрузка данных с использованием DataLoader в PyTorch

DataLoader является неотъемлемой частью любой программы глубокого машинного обучения, так как он позволяет эффективно организовать процесс загрузки данных, их обработки и использования для обучения модели.

Зачем нужен DataLoader?

В реальных задачах машинного обучения данные часто бывают объемными и сложными:

- Изображения могут занимать гигабайты памяти.
- Текстовые данные требуют предобработки (токенизации, векторизации).
- Числовые данные могут быть огромных размеров.

Прямая работа с такими данными может привести к проблемам. Например, нехватка памяти: если все данные загружены сразу, это может перегрузить оперативную память. Или медленной обработке, тогда обучение модели на всех данных одновременно может быть неэффективным. Также может потребоваться ручное управление потоками данных, что усложняет код.

DataLoader решает эти проблемы, предоставляя:

- Пакетную обработку данных, через загрузку небольшими порциями (batches), что экономит память.
- Автоматическое перемешивание данных помогает избежать переобучение модели.
- Параллельная загрузка данных ускоряет процесс подготовки данных за счет многопоточности.

Чтобы использовать DataLoader, важно понимать его основные компоненты:

Dataset - это объект, который хранит ваши данные. В PyTorch есть два типа датасетов:

- класс `torch.utils.data.Dataset` – базовый класс для создания собственных датасетов.
- готовые датасеты, например, `torchvision.datasets` для изображений (например, CIFAR10) или `torchtext.datasets` для текста (например, MNIST).

DataLoader – это обертка над Dataset, которая предоставляет итератор для удобной загрузки данных. Параметры DataLoader позволяют настроить процесс загрузки. Перед использованием DataLoader необходимо создать объект Dataset. Разберем как это сделать.

Использование готовых датасетов

PyTorch предоставляет множество готовых датасетов. Например, загрузка датасета MNIST:

```
from torchvision import datasets, transforms # Преобразование данных (например, нормализация)
transform = transforms.Compose([
    transforms.ToTensor(),  # Преобразование изображений в тензоры
    transforms.Normalize((0.5,), (0.5,))  # Нормализация значений
])
# Загрузка датасета
train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)
```

Создание собственного Dataset

Если у вас есть собственные данные, вы можете создать собственный класс, унаследованный от `torch.utils.data.Dataset`:

```
from torch.utils.data import Dataset
class CustomDataset(Dataset):
    def __init__(self, data, labels, transform=None):
        self.data = data
        self.labels = labels
        self.transform = transform
    def __len__(self):
        return len(self.data)
    def __getitem__(self, idx):
        sample = self.data[idx]
        label = self.labels[idx]
        if self.transform:
            sample = self.transform(sample)
        return sample, label
```

Теперь, когда у нас есть Dataset, мы можем создать DataLoader. Основные параметры:

```
from torch.utils.data import DataLoader
train_loader = DataLoader(
    dataset=train_dataset,  # Используемый датасет
    batch_size=64,          # Размер батча
    shuffle=True,           # Перемешивание данных перед каждой эпохой
    num_workers=4           # Количество потоков для загрузки данных
)
```

Разберем подробнее параметры.

- **batch_size** задает размер одного батча (пакета). Например, если batch_size=64, то данные будут загружаться по 64 элемента за раз. Меньший размер батча требует меньше памяти, но обучение может быть менее стабильным. Больший размер батча ускоряет обучение, но требует больше памяти.
- **shuffle** задает необходимость перемешивания данных. Если True, данные перемешиваются перед каждой эпохой. Это помогает избежать переобучения. Для тестовых данных обычно задают shuffle=False.
- **num_workers** определяет количество потоков для параллельной загрузки данных. Увеличение этого параметра ускоряет загрузку, но требует больше ресурсов.
- **drop_last**. Если True, последний неполный батч будет отброшен. Полезно, если размер данных не делится нацело на batch_size.

После создания DataLoader мы можем легко итерироваться по данным. Например:

```
for batch_idx, (data, labels) in enumerate(train_loader):
   print(f"Batch {batch_idx}:")
   print(f"Data shape: {data.shape}, Labels shape: {labels.shape}")
```

Пример вывода

```
Batch 0:
Data shape: torch.Size([64, 1, 28, 28]), Labels shape: torch.Size([64])
Batch 1:
Data shape: torch.Size([64, 1, 28, 28]), Labels shape: torch.Size([64])
...
```

Здесь `data` — тензор с данными (например, изображения), а `labels` — тензор с метками.

Практические советы

- Оптимизация загрузки данных. Используйте num_workers для ускорения загрузки. Убедитесь, что данные помещаются в память.
- Работа с большими датасетами. Если данные не помещаются в память, используйте ленивую загрузку (lazy loading).
- Трансформации данных. Всегда применяйте необходимые преобразования данных (например, нормализацию) через transform.
- Отладка. Проверяйте форму данных и меток на каждом шаге, чтобы избежать ошибок.

Предобработка данных и их трансформации

Предобработка данных является фундаментальным для успешной работы с моделями, так как качество данных напрямую влияет на результаты обучения. В классическом машинном обучении данные проходят такие этапы:

- Обработка пропущенных значений через удаление или заполнение пропусков.
- Кодирование категориальных данных. Модели машинного обучения работают только с числовыми данными, поэтому категориальные признаки нужно преобразовать.
- Масштабирование и нормализация. Разные признаки могут иметь разные масштабы (например, возраст от 0 до 100 и доход от 0 до 1 млн). Это может негативно повлиять на обучение модели.
- Балансировка данных.

В обучении нейросетей данные тоже нужно предобработать.

Для изображений важны следующие преобразования:

- Приведение всех изображений к одному размеру.
- Нормализация. Пиксельные значения обычно нормализуются в диапазон [0, 1] или [-1, 1].
- Аугментация данных, т.е добавление измененных версий изображений (поворот, отражение, обрезка) для увеличения объема данных.

Текстовые данные требуют специфической обработки:

- Токенизация – разделение текста на слова или символы.
- Удаление стоп-слов, например, "и", "в".
- Векторизация, т.е преобразование текста в числовые представления/

Разберем эти методы подробнее.

Работа с изображениями

Работа с изображениями в задачах машинного обучения требует особого внимания к их предобработке. Это связано с тем, что модели, такие как сверточные нейронные сети (CNN), чувствительны к формату и содержанию данных, поэтому предобработка становится критически важной.

Первый шаг – **приведение изображений к одному размеру.**

Модели машинного обучения требуют, чтобы входные данные имели одинаковую форму. Например, CNN ожидает, что все изображения будут иметь размер (H, W, C), где H — высота, W — ширина, а C — количество каналов (например, 3 для RGB).

Существуют разные методы изменения размера (resizing):

- Изменение размера без сохранения пропорций, т.е. простое масштабирование до заданных значений.
- Изменение размера с сохранением пропорций. В этом случае добавляются «пустые» области (padding) для сохранения соотношения сторон.

```
from torchvision import transforms
# Изменение размера изображений до 224x224 пикселей
resize_transform = transforms.Resize((224, 224))
# Пример применения
transformed_image = resize_transform(image)
```

Затем идет **нормализация**.

Пиксельные значения изображений обычно находятся в диапазоне [0, 255]. Такой широкий диапазон может замедлить обучение модели, так как градиенты становятся слишком большими или маленькими. Нормализация помогает ускорить сходимость модели.

При нормализации значения пикселей приводятся к стандартному диапазону [0,1] через деление значений на 255 или [-1,1] через вычитание среднего и деление на стандартное отклонение.

```
from torchvision import transforms

# Нормализация в диапазон [0, 1]
normalize_0_1 = transforms.Compose([
    transforms.ToTensor()  # Преобразование в тензор и нормализация в [0, 1]
])

# Нормализация с использованием среднего и стандартного отклонения
normalize_custom = transforms.Normalize(
    mean=[0.485, 0.456, 0.406],  # Средние значения для RGB каналов
    std=[0.229, 0.224, 0.225]    # Стандартные отклонения для RGB каналов
)



# Полный пайплайн
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    normalize_custom
])
```

И еще один важный ваг предобработки – **аугментация**.

Аугментация данных — это преобразование существующих тренировочных данных таким образом, чтобы обучающий набор данных стал более разнообразным и обширным без сбора новых экземпляров.

Впервые идеи схожие с аугментацией начали применяться для расширения обучающих наборов данных еще в 1990-х годах. Однако широкое распространение она получила с развитием глубокого обучения и конкретно с такими работами, как исследование Алекса Кшижевского (Alex Krizhevsky) и других, которые использовали аугментацию данных для улучшения результатов на ImageNet с помощью сети AlexNet в 2012 году.

Аугментация данных играет ключевую роль в повышении устойчивости модели к переобучению, особенно когда доступный объем данных ограничен. Путем введения различных вариантов одного и того же изображения, модель учится игнорировать незначительные детали и фокусироваться на основных характеристиках объектов. Это повышает способность модели обобщать, то есть правильно работать с новыми, невиданными ранее данными. Это особенно актуально в задачах компьютерного зрения и обработки аудио, где небольшие изменения в изображениях или звуках (например, повороты, масштабирование, шум) могут существенно увеличить объем и разнообразие данных без изменения их качественной принадлежности к тому или иному классу.

Основные техники аугментации включают:

- геометрические преобразования: повороты, масштабирование, сдвиги, искажение (например, перспективное преобразование).
- цветовые модификации: изменение яркости, контраста, насыщенности, добавление шума.
- случайное обрезание (random cropping) и перевороты изображений (flipping).
- эластичные преобразования: для создания искажений, которые имитируют реальные физические изменения объектов.
- смешивание изображений: комбинирование частей различных изображений для создания новых примеров.

На практике аугментация данных часто реализуется с помощью различных библиотек, таких как torchvision, Augmentor, Albumentations и imgaug. Эти инструменты предоставляют широкий спектр функций для манипулирования данными, поддерживая как стандартные, так и более сложные методы аугментации.

```
from torchvision import transforms

# Аугментация данных
augmentation_transform = transforms.Compose([
   transforms.RandomHorizontalFlip(p=0.5),    # Случайное горизонтальное отражение
   transforms.RandomRotation(degrees=30),     # Случайный поворот на ±30 градусов
```

```
    transforms.ColorJitter(brightness=0.2,  contrast=0.2),     # Изменение яркости и контраста
    transforms.RandomResizedCrop(224, scale=(0.8, 1.0))  # Случайная обрезка и изменение размера
])

# Полный пайплайн с аугментацией и нормализацией
transform = transforms.Compose([
    augmentation_transform,
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485,  0.456,  0.406],  std=[0.229,  0.224,  0.225])
])
```

Работа с текстом

С чего начать решение задачи обработки естественного языка (NLP) в глубоком обучении?

Для начала необходимо собрать данные и предварительно их обработать, например, удалить спецсимволы и стоп-слова, привести текст к нижнему регистру и токенизировать.

Затем извлечь признаки и выбрать необходимые функции, используя векторизацию. Векторизация – это процесс преобразования текстовых данных в числовые векторы. Этот процесс необходим для того, чтобы модели машинного обучения, могли эффективно работать с текстовой информацией, которая изначально представлена в виде последовательности слов.

После выбрать модель, обучить ее и оценить полученный результат.

Токенизация

Токенизация решает несколько важных задач:

- Преобразование текста в числовое представление. Модели машинного обучения работают только с числами, поэтому текст нужно преобразовать в формат, который они смогут понять.
- Упрощение анализа текста. Разделение текста на токены позволяет выделить ключевые элементы, такие как слова или подслова.
- Унификация данных. Стандартизация текста через токенизацию помогает устранить различия в форматах (например, регистр букв, пунктуация)

Существуют разные типа токенизации

Word tokenization (токенизация по словам)

Текст разбивается на отдельные слова. Это наиболее распространенный тип токенизации.

```
text = "Это пример текста для токенизации."
tokens = text.split()  # Простой способ токенизации
print(tokens)
# Output: ['Это', 'пример', 'текста', 'для', 'токенизации.']
```

Этот метод простой, но может создавать проблемы с пунктуацией, например считать словом «токенизации.» вместо «токенизации».

Character Tokenization (Токенизация по символам)

Текст разбивается на отдельные символы.

```
text = "текст"
tokens = list(text)
print(tokens)
# Output: ['т', 'е', 'к', 'с', 'т']
```

Такой метод подходит для языкой с богатым морфологическим измерением, например, китайский. При этом размер входных данных увеличивается и это может замедлить обучение.

Subword Tokenization (Токенизация по подсловам)

Текст разбивается на подслова (части слов). Этот подход объединяет преимущества токенизации по словам и символам.

Например, слово "невероятно" может быть разбито на подслова: ["не", "вер", "##оят", "##но"].

Этот способ эффективно работает с редкими словами и используется в современных моделях, таких как BERT, GPT и других.

Кроме разных способов токенизации существуют и различные инструменты. Например:

- **Библиотека NLTK**

NLTK (Natural Language Toolkit) предоставляет множество инструментов для работы с текстом, включая токенизацию.

```
import nltk
from nltk.tokenize import word_tokenize

nltk.download('punkt')  # Загрузка необходимых данных
text = "Это пример текста для токенизации."
tokens = word_tokenize(text, language='russian')
print(tokens)
```

```
# Output: ['Это', 'пример', 'текста', 'для', 'токенизации', '.']
```

- **Библиотека spaCy**

spaCy — это мощная библиотека для NLP, которая включает встроенную токенизацию.

```
import spacy

nlp = spacy.load("ru_core_news_sm")  # Загрузка модели для русского языка
text = "Это пример текста для токенизации."
doc = nlp(text)
tokens = [token.text for token in doc]
print(tokens)
# Output: ['Это', 'пример', 'текста', 'для', 'токенизации', '.']
```

- **Библиотека Hugging Face Transformers**

Hugging Face предоставляет готовые токенизаторы для современных моделей, таких как BERT, GPT и других.

```
from transformers import BertTokenizer

tokenizer = BertTokenizer.from_pretrained('bert-base-multilingual-cased')
text = "Это пример текста для токенизации."
tokens = tokenizer.tokenize(text)
print(tokens)
# Output: ['Это', 'при', '##мер', 'текста', 'для', 'то', '##кен', '##иза',
'##ции', '.']
```

Токенизация состоит из нескольких этапов.

1. Предварительная обработка. Перед токенизацией текст часто очищается. Из него удаляются лишние пробелы, слова приводятся к нижнему регистру и удаляются специальные символов (например, эмодзи, HTML-теги).

   ```
   import re

   text = "Пример текста с лишними   пробелами и @спецсимволами!"
   cleaned_text = re.sub(r"[^\w\s]", "", text)  # Удаляем спецсимволы
   cleaned_text = re.sub(r"\s+", " ", cleaned_text)  # Удаляем лишние
   пробелы
   print(cleaned_text)
   # Output: "Пример текста с лишними пробелами и спецсимволами"
   ```

2. Разбиение текста. Текст разбивается на токены с использованием выбранного метода (по словам, символам или подсловам).
3. Постобработка. После токенизации можно выполнить дополнительные шаги:
   - удалить стоп-слова, например, часто встречающихся слова, такие как "и", "в".

- провести лемматизация/стемминг, т.е. привести слова к базовой форме (например, "бегал" → "бег").

Пример с NLTK:

```
from nltk.corpus import stopwords
from nltk.stem import SnowballStemmer

nltk.download('stopwords')

stop_words = set(stopwords.words('russian'))
stemmer = SnowballStemmer('russian')

filtered_tokens = [stemmer.stem(token) for token in tokens if token not in
stop_words]
print(filtered_tokens)
# Output: ['пример', 'текст', 'токенизации']
```

Векторизация

Векторизация - процесс преобразования текстовых данных в числовые векторы. Это необходимо, чтобы модели машинного обучения могли обрабатывать текст.

Рассмотрим некоторые инструменты векторизации

- Bag of words
- Word2Vec
- FastText
- TF-IDF
- Glove
- BPE (Byte-pair encoding)

Bag of words

Мешок слов или Bag of words - метод векторизации текстовых данных, где документ представлен как неупорядоченное множество слов, игнорируя грамматическую структуру и порядок слов, но сохраняя их количество.
Несомненным плюсом такого подхода является его простота реализации. Но есть и существенные минусы, Bag of words не учитывает порядок слов, контекст, семантику
Пример BoW на основе предложений:

- The red dog is Mikhail's best friend.
- Jack the cat is Maria's best friend.

Bag of Words

best	cat	dog	friend	is	jack	maria	mikhail	red	the
1	0	1	1	1	0	0	1	1	1
1	1	0	1	1	1	1	0	0	1

The red dog is Mikhail's best friend.
Jack the cat is Maria's best friend.

TF-IDF

TF-IDF или Term Frequency-Inverse Document Frequency – это статистическая мера, которая используется для оценки важности слова в контексте документа, являющегося частью коллекции документов или корпуса.

Детальнее рассмотрим части определения:

- TF (term frequency) измеряет, насколько часто слово встречается в документе. Рассчитывается как отношение числа раз, когда слово встречается в документе, к общему числу слов в этом документе.
- IDF (inverse document frequency) измеряет, насколько важно слово во всем корпусе. Рассчитывается как логарифм отношения общего числа документов в корпусе к числу документов, содержащих слово.

Плюсы: простота использования и функция учета важности слов.

Минусы: не учитывает семантику и порядок слов.

Пример TF-IDF на основе предложений:

- The red dog is Mikhail's best friend.
- Jack the cat is Maria's best friend.

TF-IDF

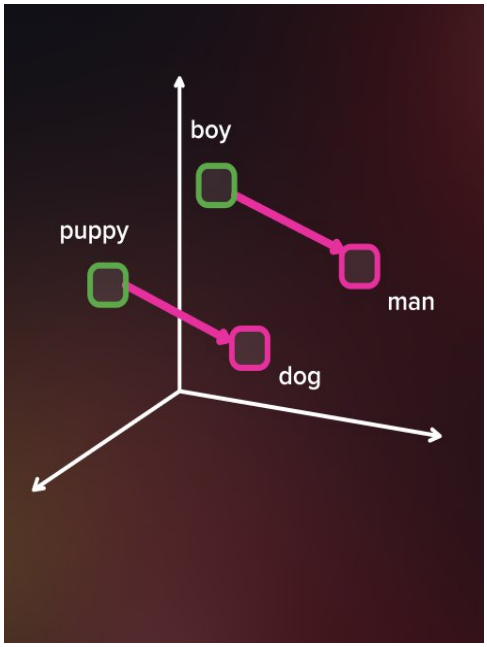
best	cat	dog	friend	is
0.3174043985887566	0.0	0.4461008073765536	0.3174043985887566	0.3174043985887566
0.3174043985887566	0.44610080737655367	0.0	0.3174043985887566	0.3174043985887566

jack	maria	mikhail	red	the
0.0	0.0	0.4461008073765536	0.4461008073765536	0.3174043985887566
0.44610080737655367	0.44610080737655367	0.0	0.0	0.3174043985887566

Word2Vec

Метод представления слов в векторном пространстве, в основе которого лежит нейронная сеть. Такой подход требует предварительного обучения модели.

Ключевая идея, лежащая в основе Word2Vec, подразумевает, что слова встречающиеся в похожих контекстах, должны иметь схожие векторные представления.



Изображение: Word2Vec / serokell.io

Существует две основные архитектуры: "Continuous Bag of Words" (CBOW) и модель "Skip-Gram".

CBOW:

- CBOW предсказывает целевое слово (центральное слово) на основе его контекста (окружающих слов).
- Входом в модель является окно контекста окружающих слов, а выходом - целевое слово.
- Модель обучается минимизировать разницу между предсказанными и фактическими целевыми словами.

Skip-gram:

- Skip-Gram, предсказывает контекстные слова на основе целевого слова.
- Входом является целевое слово, а выходом - вероятностное распределение контекстных слов в заданном окне.
- Модель обучается максимизировать вероятность контекстных слов, учитывая целевое слово.

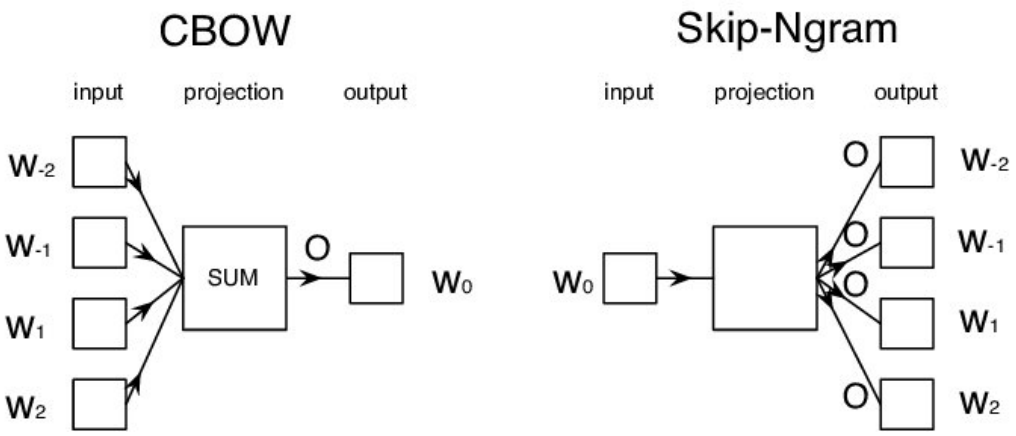


Схема: Ling, Wang & Dyer, Chris & Black, Alan & Trancoso, Isabel. (2015). Two/Too Simple Adaptations of Word2Vec for Syntax Problems. 10.3115/v1/N15-1142 / researchgate.net

Пример Word2Vec на основе предложений:

- The red dog is Mikhail's best friend.
- Jack the cat is Maria's best friend.

Word2Vec

0	1	2	3	4	5	6	7	8	9
-0.0053622723	0.0023643137	0.051033497	0.09009273	-0.0930295	-0.07116809	0.064588726	0.08972988	-0.05015428	-0.037633717
0.07380505	-0.015334713	-0.04536613	0.065540515	-0.048601605	-0.018160176	0.028765798	0.009918737	-0.08285215	-0.09448818
0.07311766	0.05070262	0.06757693	0.0076286555	0.063508905	-0.03405366	-0.009464013	0.057685733	-0.075216375	-0.039361037
-0.07511582	-0.009300423	0.095381185	-0.073191665	-0.023337686	-0.01937741	0.08077437	-0.059308957	0.0004516244	-0.047537338
-0.0960355	0.05007293	-0.08759586	-0.043918252	-0.00035099982	-0.0029618144	-0.0766124	0.09614743	0.04982058	0.09233143
-0.08157917	0.044957984	-0.04137076	0.008245361	0.084986195	-0.044621766	0.045175005	-0.0678696	-0.035484888	0.09398508
-0.015776526	0.0032137155	-0.041406296	-0.076826885	-0.015080082	0.024697948	-0.00888027	0.055336617	-0.02742977	0.02260065
0.054560937	0.083464116	-0.014538203	-0.09208648	0.043707922	0.0057181637	0.074423164	-0.008133274	-0.026385587	-0.0875349
-0.008554139	0.028285386	0.05401408	0.07051035	-0.05702481	0.018590463	0.06090821	-0.047984917	-0.031080065	0.06796093
0.016314756	0.0018991709	0.034736373	0.002177775	0.09618826	0.050606035	-0.0891739	-0.0704156	0.009014559	0.06392534
-0.08619688	0.03665738	0.051898837	0.057419382	0.07466918	-0.06167675	0.011056137	0.060472824	-0.028400505	-0.061735224

```
model = Word2Vec(tokenized_sentences, vector_size=10, window=5, min_count=1, workers=4)
```

Помимо самостоятельного обучения, существуют готовые предобученные модели. Некоторые популярные ресурсы на которых их можно найти:

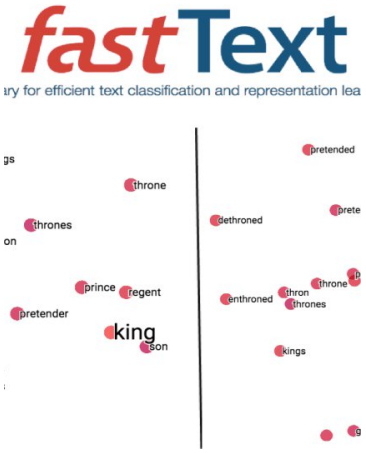
https://rusvectores.org/ - семантические модели для русского языка

https://github.com/piskvorky/gensim-data - репозиторий данных Gensim

Плюсы: учитывает семантику отношений между словами и контекст слов в заданном корпусе.

Минусы: длительное время обучения и необходимость использования большого объема данных.

FastText



Изображение: fastText / wandb.ai

FastText - это библиотека для изучения встраивания слов и классификации текста. Усовершенствованная модель Word2Vec, в которой добавлена поддержка символьных n-grams, а также некоторые улучшения для ускорения процесса обучения.

Какие преимущества дает использование n-grams в FastText:

- Учет морфологии и словообразования. N-grams позволяют FastText изучать векторы не только для отдельных слов, но и для их частей, например, буквенных комбинаций. Это полезно для учета морфологических изменений, а также для обработки новых слов или слов с опечатками.
- Работа с неизвестными словами. Благодаря использованию n-grams, FastText способен обрабатывать неизвестные слова, составленные из известных n-grams. Это особенно важно в задачах, где требуется обработка текста с ограниченным словарем или с большим количеством специфичных терминов.
- Улучшенное обобщение. Модели, учитывающие n-grams, часто лучше обобщают информацию, так как они могут учить зависимости на уровне подслов. Это может быть полезно в сценариях с ограниченными данными, где модель должна извлекать максимум информации из имеющихся текстов.
- Лучшая поддержка для языков с богатой морфологией. В языках с богатой морфологией, где слова могут изменяться по форме, использование n-grams может значительно улучшить качество представлений слов и текстов.

Однако следует отметить, что использование n-grams также может увеличить размер модели и требования к вычислительным ресурсам. Выбор между использованием n-грамм и обычных векторных представлений зависит от конкретной задачи, размера доступных данных и требований к вычислительным ресурсам.

Пример FastText на основе предложений:

- The red dog is Mikhail's best friend.
- Jack the cat is Maria's best friend.

FastText

0	1	2	3	4	5	6	7	8	9
0.010838592	-0.002231252	-0.0055345315	0.014374214	-0.017806444	-0.011265729	0.007969982	-0.015072585	-0.024586465	0.011132429
0.021846056	0.020157635	0.0010977733	-0.013403538	-0.0136881	0.013414502	0.0009957011	0.039570637	0.03442612	0.005965072
0.017982768	0.019638294	-0.017755056	0.004040229	0.018399732	-0.0005988302	0.022805028	0.026207373	-0.032819487	-0.049018923
-0.0047462457	0.02608639	0.014323267	0.01165883	0.006617697	0.0012788499	-0.0111634275	0.0020903975	-0.01084095	0.013326192
-0.02463297	-0.019759165	-0.051404007	-0.012202988	-0.0017222116	0.0063068336	-0.029373838	-0.003030272	-0.00250046	0.017017808
-0.010032425	-0.008658199	0.003812937	-0.025558542	0.019129291	-0.034666535	0.0013554634	-0.03219368	-0.00818502	0.010680618
0.0046384367	-0.004077367	0.011436936	-0.018427484	-0.010673082	-0.0067176227	-0.032957397	-0.030632572	-0.025877498	0.0025360598
-0.001451952	-0.0042955163	-0.00730818	0.005700969	0.008874549	-0.0021199414	-0.003490732	-0.02112388	-0.017178139	0.004908032
0.0068684495	-0.015080601	0.0006813556	-0.0019015489	-0.018121915	0.009283176	0.031927057	-0.033175044	0.0045748404	0.03726239
-0.00052465283	-0.01705743	0.0009521544	0.013452346	0.025824811	0.013232388	0.01439777	0.0112588275	0.022453729	0.0017669658
2.8622802e-05	0.013872647	0.02495262	0.012630113	0.016474688	-0.025495844	0.035252664	0.008022659	-0.03732414	0.0006791651

```
model = FastText(sentences=tokenized_sentences, vector_size=10, window=5,
min_count=1, workers=4)
```

FastText также поддерживает CBOW и Skip-gram.

Плюсы: учитывает морфологию и хорошо работает на редких словах.

Минусы: обучение модели требует много ресурсов и потребляет много памяти.

Glove

Метод получения векторных представлений путем выстраивания матрицы совместной встречаемости, которая учитывает насколько часто слова появляются в одном контексте во всем корпусе.

GloVe также позволяет использовать предварительно обученные модели, что позволяет сократить время обучения модели, перенести зависимости полученные входе изучения больших данных и работать в условиях их ограниченности.

Плюсы: учитывает частоту встречаемости слов и семантические отношения между словами.

Минусы: плохо обрабатывает неизвестные и редкие слова, без использования предварительно обученной модели.

Jack	-2.887800	-0.56988	...
the	-5.104300	2.34960	...
cat	3.703200	4.19820	...
is	1.475000	6.00780	...
Maria	-2.210500	-0.58332	...
's	3.316300	9.72090	...
best	0.907800	-0.60642	...
friend	0.156420	-1.11530	...

Glove для "Jack the cat is Maria's best friend."

spacy.load("en_core_web_md")

Рассмотренные методы обработки текста преимущественно используются для передачи данных в классические модели машинного обучения. Многие современные LLM или большие языковые модели используют другой популярный подход векторизации.

BPE (Byte-pair encoding)

Byte-pair encoding - это техника токенизации подслов, которая направлена на сокращение размера словаря без потери качества. BPE работает путем итеративного слияния наиболее частых пар соседних символов в корпусе, чтобы выделять повторяющиеся шаблоны и объединять их в один токен.

```
{'The</w>': 1,
 'red</w>': 1,
 'dog</w>': 1,
 'is</w>': 2,
 "Mikhail's</w>": 1,
 'best</w>': 2,
 'friend</w>': 2,
 'Jac k </w>': 1,
 't he</w>': 1,
 'c a t</w>': 1,
 "M a ri a 's</w>": 1}
```

Плюсы: учитывает редкие и неизвестные слова, создает компактные векторные представления слов

Минусы: требует больших ресурсов

Наборы данных и разделение данных

Теперь поговорим о наборах данных. Правильная организация наборов данных и их разделение на обучающую, валидационную и тестовую выборки играют важнейшую роль в создании качественных моделей.

Наборы данных (Datasets) — это коллекции данных, которые используются для обучения и оценки нейронных сетей. В контексте машинного обучения наборы данных обычно содержат:

- Признаки (Features) – входные данные (например, пиксели изображения, текст или числовые значения).
- Метки (Labels) – выходные данные, которые модель должна предсказать (например, класс объекта или числовое значение).

Для задачи классификации изображений входные данные — это изображения, метки — это классы, например, «кошка» или «собака». Для задачи регрессии входные данные — это признаки, например, площадь дома, метки — это целевые значения, например, цена дома.

Для обучения моделей необходимо разделить данные на несколько частей.

Если данные не разделить, то модель может быть переобучена (overfitting) или недообучена (underfitting), что приведет к плохим результатам.

Основные наборы данных.

Обучающая выборка (Training Set) – подмножество данных, которое используется для обучения модели. Обычно составляет 60–80% от общего объема данных. На этих данных модель учится находить закономерности и минимизировать функцию потерь.

Валидационная выборка (Validation Set) – подмножество данных, которое используется для настройки гиперпараметров и оценки производительности модели во время обучения. Обычно составляет 10–20% от общего объема данных. На этих данных проверяется, насколько хорошо модель обобщает знания, и корректируются параметры, такие как скорость обучения или архитектура сети.

Тестовая выборка (Test Set) – подмножество данных, которое используется для финальной оценки модели после завершения обучения. Обычно 10–20% от общего объема данных. Эти данные должны быть полностью «новыми» для модели, чтобы проверить, насколько хорошо она работает в реальных условиях.

Методы разделения данных.

Самый простой способ разделить данные — использовать **случайное перемешивание** и выделить доли для каждого набора.

```
from sklearn.model_selection import train_test_split
```

```
# Исходные данные
X, y = features, labels

# Разделение на обучающую и тестовую выборки
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3,
random_state=42)

# Разделение временного набора на валидационную и тестовую выборки
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp,
test_size=0.5, random_state=42)
```

Кросс-валидация (Cross-Validation). Кросс-валидация используется, когда данных мало, и важно максимально эффективно использовать каждый пример. Данные делятся на k частей, и модель обучается k раз, каждый раз используя одну часть для валидации и остальные для обучения.

```
from sklearn.model_selection import KFold

kf = KFold(n_splits=5, shuffle=True, random_state=42)
for train_index, val_index in kf.split(X):
   X_train, X_val = X[train_index], X[val_index]
   y_train, y_val = y[train_index], y[val_index]
   # Обучение модели...
```

Стратифицированное разделение. Если данные несбалансированы (например, в задачах классификации), важно сохранить баланс классов в каждом наборе. Для этого используется стратифицированное разделение.

```
X_train, X_test, y_train, y_test = train_test_split(
   X, y, test_size=0.2, stratify=y, random_state=42
)
```

Пример полного процесса

```
from sklearn.model_selection import train_test_split
from torchvision import transforms
from torch.utils.data import DataLoader, Dataset

# Создание собственного датасета
class CustomDataset(Dataset):
   def __init__(self, data, labels, transform=None):
       self.data = data
       self.labels = labels
       self.transform = transform

   def __len__(self):
       return len(self.data)

   def __getitem__(self, idx):
       sample = self.data[idx]
```

```
        label = self.labels[idx]
        if self.transform:
            sample = self.transform(sample)
        return sample, label

# Исходные данные
X, y = features, labels

# Разделение данных
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3,
random_state=42)
X_val,   X_test,   y_val,   y_test   =   train_test_split(X_temp,   y_temp,
test_size=0.5, random_state=42)

# Преобразования
transform = transforms.Compose([
   transforms.ToTensor(),
   transforms.Normalize(mean=[0.5], std=[0.5])
])

# Создание датасетов
train_dataset = CustomDataset(X_train, y_train, transform=transform)
val_dataset = CustomDataset(X_val, y_val, transform=transform)
test_dataset = CustomDataset(X_test, y_test, transform=transform)

# Создание DataLoader
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

Заключение

Обработка данных — это важнейший этап в разработке моделей машинного обучения, который требует внимательного подхода и понимания инструментов. На этом занятии мы изучили, как использовать DataLoader для загрузки данных, а также как применять различные трансформации для их предобработки. Мы рассмотрели особенности работы с изображениями и текстами, что позволит вам адаптировать данные под специфику задачи. Кроме того, мы научились правильно организовывать наборы данных и выполнять их разделение, чтобы обеспечить корректное обучение и оценку модели.

Теперь вы обладаете необходимыми знаниями и навыками для эффективной работы с данными в PyTorch. Эти знания станут прочной основой для создания качественных моделей и успешного решения задач машинного обучения.