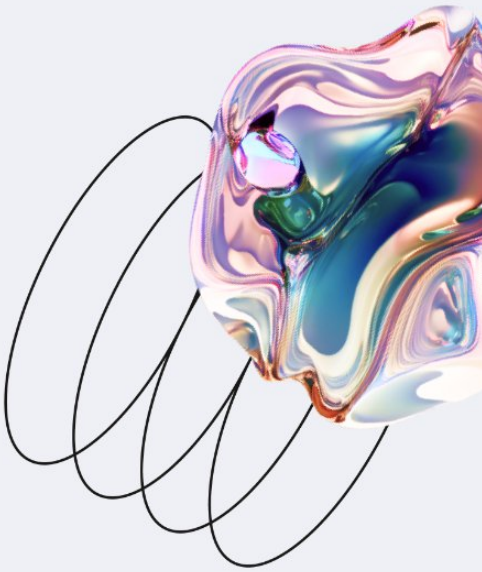
GeekBrains



Расширенное обучение с Pytorch

Архитектура нейронных сетей

Оглавление

Введение

Всем привет! Это наша девятая лекция на курсе архитектура нейронных сетей Сегодня мы обсудим способы расширенного обучения с использованием PyTorch. PyTorch — это мощный и гибкий фреймворк для создания и обучения нейронных сетей, широко используемый в исследовательской и прикладной разработке машинного обучения. Он обеспечивает простоту и удобство при разработке, предоставляя динамическое вычислительное графическое представление и обширную экосистему инструментов для работы с нейронными сетями. PyTorch активно применяется как в научных исследованиях, так и в индустрии для построения систем компьютерного зрения, обработки естественного языка и других областей.

Цель этой лекции — углубить ваше понимание работы с PyTorch и научить решать сложные задачи машинного обучения, включая:

1. Обучение нейронной сети: ключевые шаги, связанные с разработкой, настройкой и обучением моделей.
2. Перенос обучения (Transfer Learning): использование предварительно обученных моделей для новых задач.
3. Работа с собственными наборами данных (Custom Datasets): создание и обработка пользовательских наборов данных для специфичных задач.
4. Деплоймент моделей PyTorch: экспорт, интеграция и оптимизация моделей для использования в реальных приложениях.

Эти темы являются важными компонентами профессиональной работы с PyTorch, и их освоение поможет вам эффективно разрабатывать и внедрять решения на базе нейронных сетей в своих проектах.

Давайте начинать!

Словарь терминов

Перенос обучения — это метод, который позволяет моделям применять ранее полученные знания к новым, задачам связанным с предыдущими.

Пользовательский набор данных – это набор данных, относящихся к конкретной проблеме, над которой вы работаете.

Деплоймент модели — это процесс предоставления обученной модели на производство для использования в реальном времени.

Перенос обучения (Transfer Learning)

Как люди, мы преуспеваем в переносе знаний с одной задачи на другую. Когда мы сталкиваемся с новой проблемой или задачей, мы часто полагаемся на прошлый опыт, чтобы он нас направлял. Аналогично, в мире машинного обучения **перенос обучения** — это метод, который позволяет моделям применять ранее полученные знания к новым, задачам связанным с предыдущими.

Вместо того чтобы начинать с нуля, модель повторно использует то, чему она уже научилась, что позволяет ей решать новые задачи более эффективно и действенно. Этот подход особенно выгоден, когда для новой проблемы доступны ограниченные данные.

Перенос обучения (Transfer Learning) — это подход, при котором предварительно обученная модель используется для решения новой задачи. Вместо того чтобы обучать модель с нуля, мы адаптируем уже обученную на большой выборке модель для конкретного применения. Это позволяет значительно сократить время обучения и улучшить производительность моделей на небольших выборках данных.

Такой подход особенно полезен в области глубокого обучения, где большие объемы данных и вычислений требуются для обучения сложных моделей. Transfer learning также снижает риск переобучения , поскольку модель уже включает в себя обобщаемые признаки, полезные для второй задачи.

Transfer learning в основном используется в задачах компьютерного зрения и обработки естественного языка, таких как анализ настроений, из-за огромного объема требуемой вычислительной мощности.

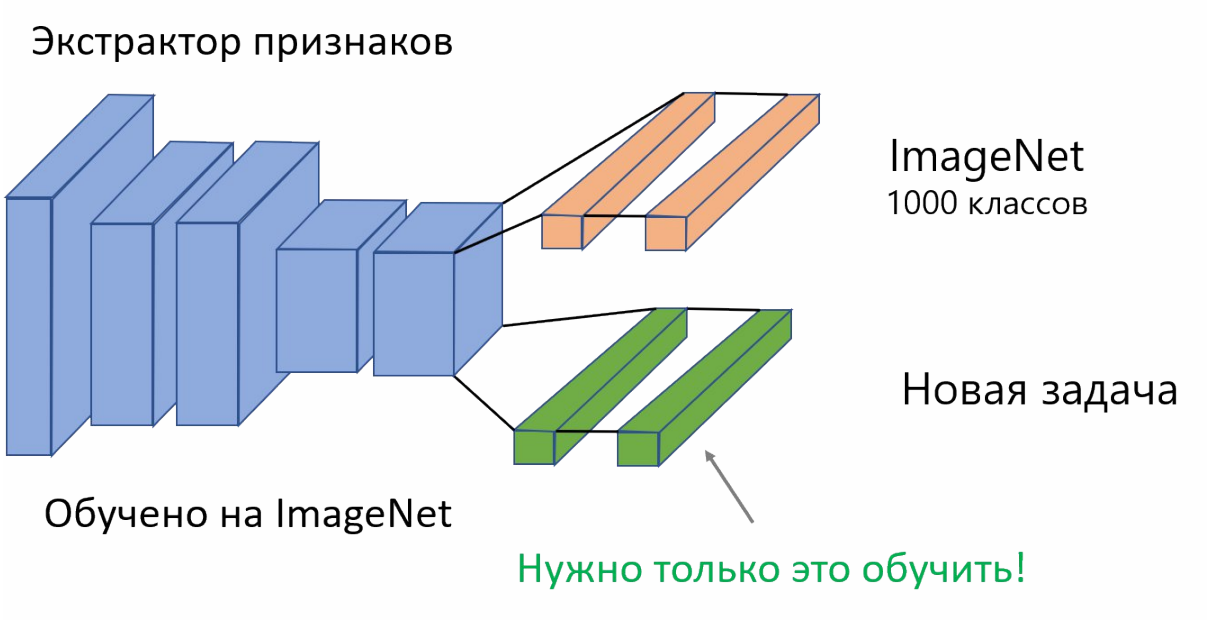
При этом обучение нейронной сети требует большого объёма данных, например, в ImageNet это миллионы картинок. **ImageNet** — база данных аннотированных изображений, предназначенная для отработки и тестирования методов распознавания образов и машинного зрения. ImageNet использует краудсорсинг для аннотирования изображений. ImageNet — это dataset, организованный в соответствии с иерархией WordNet.

Что делать, если в текущей задаче не так много данных? Оказывается, в решении новой задачи можно переиспользовать часть нейросети, которая училась на ImageNet.

Такую сеть рассматривают, как состоящую из двух частей:

- Первая часть (на рисунке синяя) состоит из конволюционных слоёв и извлекает из картинки признаки;
- Вторая часть (на рисунке оранжевая) с помощью многослойного персептрона (MLP) по признакам определяет, к какому из тысячи классов принадлежит

картинка.



При решении новой задачи сохраняют первую часть сети, которая отвечает за извлечение признаков, так как общие визуальные признаки могут быть полезны в любом контексте. Вторая часть адаптируется или заменяется, чтобы решать новую задачу, используя доступные данные. Такой подход называется transfer learning. Таким образом, чтобы обучить сеть на новой задаче, необходимо доучить только крайний полносвязный слой.

Transfer learning на самом деле не является методом машинного обучения, но может рассматриваться как «методология проектирования» в этой области. Оно также не является эксклюзивным для машинного обучения. Тем не менее, оно стало довольно популярным в сочетании с нейронными сетями, которым требуются огромные объемы данных и вычислительная мощность.

Применение transfer learning

Transfer learning широко используется в различных областях, включая:

1. **Компьютерное зрение**. Transfer learning широко распространен в задачах распознавания изображений, где модели, предварительно обученные на больших наборах данных изображений, адаптируются к конкретным задачам, таким как медицинская визуализация, распознавание лиц и обнаружение объектов.

2. **Обработка естественного языка (NLP)**. В NLP такие модели, как BERT, GPT и ELMo, предварительно обучаются на больших текстовых корпусах, а затем настраиваются для выполнения конкретных задач, таких как анализ настроений, машинный перевод и ответы на вопросы.

3. **Здравоохранение**. Transfer learning помогает разрабатывать медицинские диагностические инструменты, используя знания из общих моделей

распознавания изображений для анализа медицинских изображений, таких как рентгеновские снимки или МРТ.

4. **Финансы**. Передача обучения в сфере финансов помогает выявлять мошенничество, оценивать риски и оценивать кредитоспособность путем переноса закономерностей, изученных из соответствующих наборов финансовых данных.

Как работает Transfer learning?

При использовании метода переноса обучения предобученная на большом наборе данных для решения определенной задачи модель адаптируется для решения новой задачи. При этом часть или все слои модели могут быть заморожены, чтобы сохранить изученные характеристики из первоначальной задачи, в то время как другие слои могут быть дообучены на новом наборе данных.

Transfer learning представляет собой структурированный процесс использования существующих знаний из предварительно обученной модели для решения новых задач:

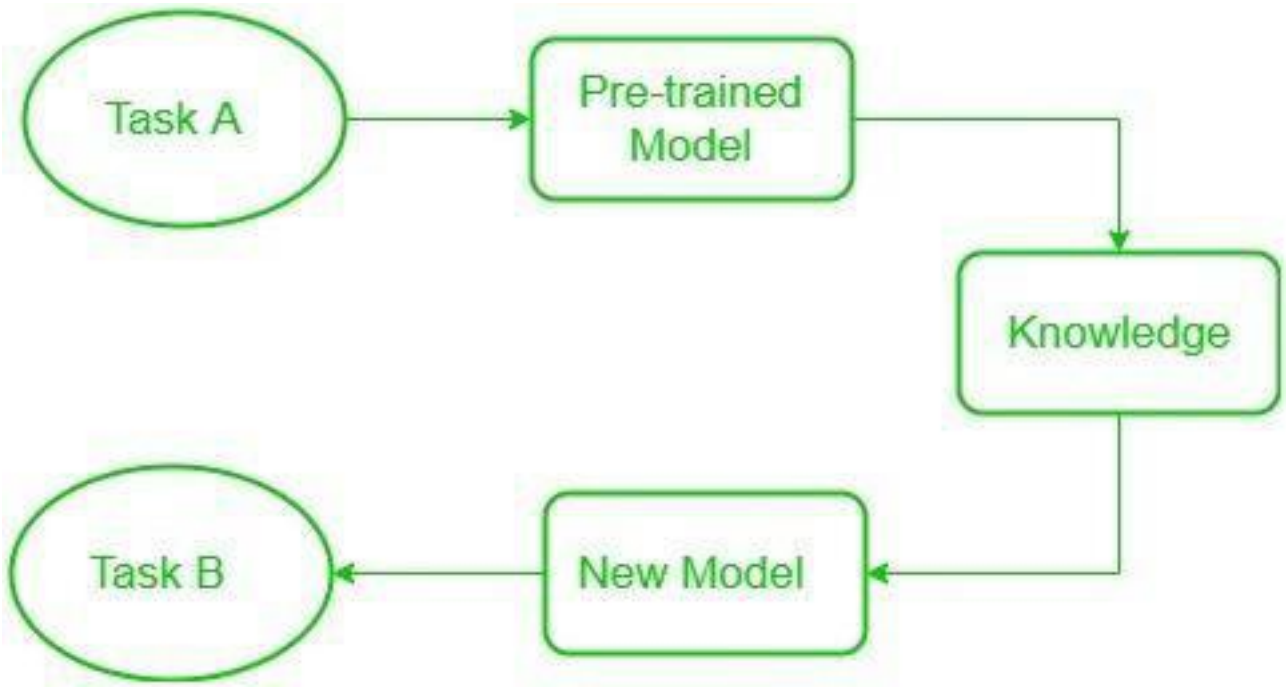
1. **Предварительно обученная модель.** Начните с модели, уже обученной на большом наборе данных для определенной задачи. Эта предварительно обученная модель изучила общие особенности и закономерности, которые актуальны для связанных задач.

2. **Базовая модель –** эта предварительно обученная модель, известная как базовая модель, включает слои, которые обработали данные для изучения иерархических представлений, фиксируя низкоуровневые и сложные признаки.

3. **Уровни передачи.** Определите слои в базовой модели, которые содержат общую информацию, применимую как к исходным, так и к новым задачам. Эти слои, часто находящиеся в верхней части сети, захватывают широкие, повторно используемые функции.

4. **Тонкая настройка.** Тонкая настройка этих выбранных слоев с данными из новой задачи. Этот процесс помогает сохранить предварительно обученные знания, одновременно корректируя параметры для соответствия конкретным требованиям новой задачи, улучшая точность и адаптивность.

Блок-схема представлена ниже следующим образом:



Перенос обучения

Низкоуровневые признаки, изученные для задачи A, должны быть полезны для изучения модели для задачи B.

Например, в компьютерном зрении нейронные сети обычно пытаются обнаружить края в более ранних слоях, формы в среднем слое и некоторые специфические для задачи особенности в более поздних слоях. При transfer learning используются ранние и средние слои, и мы только переобучаем последние слои. Это помогает использовать помеченные данные задачи, на которой она была изначально обучена.

Этот процесс переобучения моделей известен как тонкая настройка. Однако в случае transfer learning нам необходимо изолировать определенные слои для переобучения. Затем при применении transfer learning следует иметь в виду два типа слоев:

- **Замороженные слои** – слои, которые остаются нетронутыми во время повторного обучения и сохраняют знания из предыдущей задачи, на основе которых может строиться модель.
- **Изменяемые слои** – слои, которые переобучаются в процессе тонкой настройки, чтобы модель могла адаптировать свои знания к новой, связанной задаче.

В transfer learning мы пытаемся перенести как можно больше знаний из предыдущей задачи, на которой обучалась модель, в новую задачу. Эти знания могут быть в разных формах в зависимости от проблемы и данных. Например, это может быть то, как составлены модели, что позволяет нам легче идентифицировать новые объекты.

Архитектура предварительно обученных моделей

Предварительно обученные модели — это архитектуры, обученные на больших и общих наборах данных (например, ImageNet). Рассмотрим ключевые аспекты их структуры:

Слои для извлечения признаков:

- **Низкоуровневые слои** извлекают простые признаки, такие как границы, текстуры и базовые формы. Эти слои чувствительны к базовым элементам изображений и включают свёрточные фильтры небольшого размера.
- **Среднеуровневые слои** реагируют на более сложные паттерны, такие как комбинации границ и текстур, формируя базовые элементы объектов (например, углы, контуры, цвета).
- **Высокоуровневые слои** способны извлекать семантически богатые признаки, такие как формы, объекты и сложные паттерны, которые связаны с конкретными классами. Эти слои обычно обучаются представлять абстрактные характеристики данных.
- **Полносвязные слои (FC)** чаще всего располагаются в конце архитектуры и предназначены для выполнения задач классификации или регрессии. Эти слои преобразуют признаки, извлечённые предыдущими слоями, в окончательное предсказание.

Примеры архитектур:

- **ResNet** использует остаточные блоки (residual blocks) для облегчения обучения глубоких сетей.
- **VGG.** Структура сети состоит из последовательных свёрточных слоёв с фиксированными размерами фильтров.
- **EfficientNet:** Применяет оптимизацию баланса между глубиной, шириной и разрешением модели.

Какие слои заморозить или изменить

Теперь можно спросить, как определить, какие слои нам нужно заморозить, а какие слои нужно обучить. Ответ прост: чем больше вы хотите унаследовать признаков от предварительно обученной модели, тем больше вам нужно заморозить слоев.

Степень наследования признаков из предварительно обученной модели зависит от размера и сходства целевого набора данных с исходным набором данных:

- **Маленький и похожий целевой набор данных.** Когда набор данных ограничен, но похож на базовый набор данных, тонкая настройка только нескольких слоев может привести к переобучению. В этом случае последние один или два полносвязанных слоя удаляются и заменяются новым слоем, который соответствует целевым классам. Остальная часть модели замораживается, и обучаются только вновь добавленные слои.

- **Большой и похожий целевой набор данных.** Если целевой набор данных большой и похожий, переобучение менее вероятно. Последний полносвязный слой удаляется, заменяется для соответствия новым классам, и вся модель настраивается на новом наборе данных, сохраняя архитектуру и адаптируясь к большему набору данных.
- **Маленький и разный целевой набор данных.** Когда целевой набор данных небольшой и разный, высокоуровневые признаки исходной модели менее полезны. Здесь удаляют большую часть верхних слоев и добавляют новые слои для размещения целевых классов. Модель обучают с этих нижних слоев, чтобы соответствовать уникальному набору данных.
- **Большой и разный целевой набор данных.** Для больших и разных наборов данных модель выигрывает больше всего от удаления последних слоев и добавления новых слоев, адаптированных к новой задаче. Затем вся модель переобучается без замороженных слоев, чтобы обеспечить полную адаптацию.

Transfer learning обеспечивает эффективную и действенную отправную точку, часто приводящую к высокоточным результатам и более быстрой адаптации к новым задачам.

Зачем использовать перенос обучения

Основными преимуществами transfer learning являются экономия времени обучения, повышение производительности нейронных сетей (в большинстве случаев) и отсутствие необходимости в большом количестве данных.

Обычно для обучения нейронной сети с нуля требуется много данных, но доступ к этим данным не всегда возможен. С помощью transfer learning можно построить надежную модель машинного обучения с относительно небольшим количеством данных для обучения, поскольку модель уже предварительно обучена. Это особенно ценно при обработке естественного языка, поскольку для создания больших наборов маркированных данных требуются в основном экспертные знания. Кроме того, время обучения сокращается, поскольку иногда может потребоваться несколько дней или даже недель, чтобы обучить глубокую нейронную сеть с нуля для сложной задачи.

Когда использовать перенос обучения

Как это всегда бывает в машинном обучении, сложно сформулировать правила, которые будут применимы в целом, но вот несколько рекомендаций относительно того, когда можно использовать transfer learning:

- Недостаток размеченных данных для обучения сети с нуля.
- Уже существует сеть, предварительно обученная для решения аналогичной задачи, которая обычно обучается на огромных объемах данных.
- Когда задача 1 и задача 2 имеют одинаковые входные данные.

Если исходная модель была обучена с использованием библиотеки с открытым исходным кодом, такой как TensorFlow или PyTorch, вы можете просто восстановить ее и переобучить некоторые слои для вашей задачи. Однако помните, что transfer learning работает только в том случае, если признаки, изученные в первой задаче, являются общими, то есть они могут быть полезны и для другой связанной задачи. Кроме того, входные данные модели должны иметь тот же размер, с которым они изначально обучались. Если у вас его нет, добавьте шаг предварительной обработки, чтобы изменить размер входных данных до необходимого размера.

Способы применения переноса обучения

1. Обучение модели для ее повторного использования

Представьте, что вы хотите решить задачу **A**, но у вас недостаточно данных для обучения глубокой нейронной сети. Один из способов обойти это — найти связанную задачу **B** с обилием данных. Обучите глубокую нейронную сеть на задаче **B** и используйте модель в качестве отправной точки для решения задачи **A**. Необходимо ли вам использовать всю модель или только несколько слоев, во многом зависит от проблемы, которую вы пытаетесь решить.

Если у вас одинаковые входные данные в обеих задачах, возможно, можно повторно использовать модель и делать прогнозы для новых входных данных. В качестве альтернативы можно исследовать метод изменения и повторного обучения различных слоев, специфичных для задач, и выходного слоя.

2. Использование предварительно обученной модели

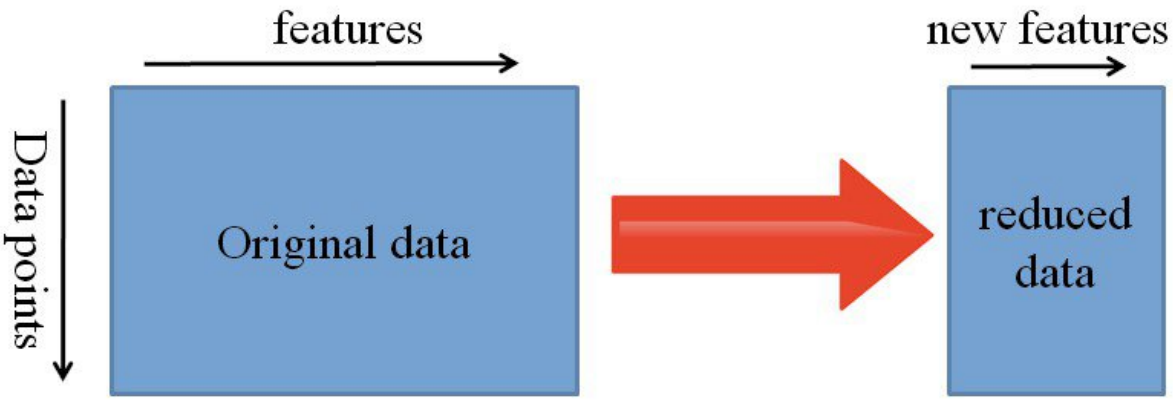
Второй подход — использовать уже предварительно обученную модель. Существует множество таких моделей, поэтому обязательно проведите небольшое исследование. Сколько слоев использовать повторно и сколько переобучать, зависит от проблемы.

Например, Keras и PyTorch предоставляет множество предварительно обученных моделей, которые можно использовать для transfer learning, прогнозирования, извлечения признаков и тонкой настройки. Существует также много исследовательских институтов, которые выпускают обученные модели.

Этот тип transfer learning чаще всего используется в глубоком обучении.

3. Извлечение признаков

Другой подход заключается в использовании глубокого обучения для обнаружения наилучшего представления вашей проблемы, что означает нахождение наиболее важных признаков. Этот подход также известен как обучение представлению, и часто может привести к гораздо лучшей производительности, чем та, которая может быть получена с помощью вручную разработанного представления.



В машинном обучении признаки обычно вручную создаются исследователями и экспертами в предметной области. К счастью, глубокое обучение может извлекать признаки автоматически. Конечно, вам все равно придется решать, какие признаки вы поместите в свою сеть. Тем не менее, нейронные сети способны узнавать, какие признаки действительно важны, а какие нет. Алгоритм обучения представлению может обнаружить хорошую комбинацию признаков в течение очень короткого периода времени, даже для сложных задач, которые в противном случае потребовали бы много человеческих усилий.

Полученное представление затем может быть использовано и для других проблем. Просто используйте первые слои, чтобы определить правильное представление признаков, но не используйте выход сети, поскольку он слишком специфичен для задачи. Вместо этого подайте данные в свою сеть и используйте один из промежуточных слоев в качестве выходного слоя. Затем этот слой можно интерпретировать как представление необработанных данных.

Этот подход в основном используется в компьютерном зрении, поскольку он позволяет уменьшить размер набора данных, что сокращает время вычислений и делает его более подходящим для традиционных алгоритмов.

Преимущества переноса обучения

Одним из ключевых преимуществ переноса обучения является возможность использовать предварительно обученные модели, что позволяет сэкономить время и ресурсы, необходимые для обучения модели с нуля. Это особенно важно в случаях, когда у вас ограниченный набор данных или ограниченные вычислительные ресурсы.

Кроме того, перенос обучения позволяет улучшить обобщающую способность модели на новом наборе данных путем передачи характеристик и знаний из предыдущей задачи.

Основные преимущества переноса обучения:

- **Ускорьте процесс обучения.** Используя предварительно обученную модель, модель может быстрее и эффективнее обучаться при выполнении второй задачи, поскольку она уже хорошо понимает особенности и закономерности в данных.

- **Лучшая производительность.** Transfer learning может привести к лучшей производительности при выполнении второй задачи, поскольку модель может использовать знания, полученные при выполнении первой задачи.
- **Обработка небольших наборов данных.** Если для второй задачи доступно ограниченное количество данных, transfer learning может помочь предотвратить переобучение, поскольку модель уже изучит общие характеристики, которые, вероятно, будут полезны во второй задаче.

Недостатки переноса обучения

- **Несоответствие доменов.** Предварительно обученная модель может не подходить для второй задачи, если две задачи существенно различаются или распределение данных между двумя задачами сильно различается.
- **Переобучение**. Transfer learning может привести к переобучению, если модель слишком точно настроена на вторую задачу, поскольку она может изучить специфические для задачи особенности, которые плохо обобщаются на новые данные.
- **Сложность**. Предварительно обученная модель и процесс тонкой настройки могут потребовать больших вычислительных затрат и специализированного оборудования.

Пример переноса обучения

Подход с transfer learning может значительно ускорить обучение и повысить точность, особенно когда данные или вычислительные ресурсы ограничены. В этом примере мы будем использовать предобученную модель ResNet-18 из библиотеки `torchvision.models` для классификации изображений на два класса (например, «кошки» и «собаки»). Мы заменим последний слой модели для нашей задачи и дообучим её на новых данных.

Шаг 1: Загрузка и предобработка данных

Мы используем `datasets.ImageFolder` для загрузки изображений из папок (train/ и val/). Для каждого набора данных определены преобразования: аугментация для обучающих данных и нормализация для обоих наборов.

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, models, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import numpy as np

# Определяем преобразования для обучающего и тестового наборов данных
data_transforms = {
    'train': transforms.Compose([
```

```
        transforms.RandomResizedCrop(224),   # Случайное изменение размера
и обрезка
        transforms.RandomHorizontalFlip(),    # Случайное горизонтальное
отражение
        transforms.ToTensor(),               # Преобразование в тензор
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
# Нормализация
    ]),
    'val': transforms.Compose([
        transforms.Resize(256),              # Изменение размера
        transforms.CenterCrop(224),          # Центральная обрезка
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
}

# Загружаем данные из папок (например, train/ и val/)
data_dir = './data'  # Путь к данным
image_datasets = {x: datasets.ImageFolder(data_dir + '/' + x,
data_transforms[x]) for x in ['train', 'val']}
dataloaders = {x: DataLoader(image_datasets[x], batch_size=32,
shuffle=True) for x in ['train', 'val']}

# Размеры наборов данных
dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'val']}
class_names = image_datasets['train'].classes  # Имена классов


# Визуализация образцов изображений
def imshow(inp, title=None):
    """Отображение образцов изображений."""
    inp = inp.numpy().transpose((1, 2, 0))    # Перестановка осей для
matplotlib
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    inp = std * inp + mean  # Обратная нормализация
    inp = np.clip(inp, 0, 1)  # Ограничение значений пикселей в диапазоне
[0, 1]
    plt.imshow(inp)
    if title is not None:
        plt.title(title)
    plt.axis('off')
    plt.show()

# Получаем один батч изображений
inputs, classes = next(iter(dataloaders['train']))
out = torchvision.utils.make_grid(inputs)  # Создаем сетку изображений
imshow(out, title=[class_names[x] for x in classes])
```

Шаг 2: Загрузка предобученной модели

Модель `resnet18` загружается с флагом `pretrained=True`, что означает использование весов, обученных на ImageNet. Мы заменяем последний слой модели (`fc`) на новый слой с двумя выходами (для двух классов).

```
model = models.resnet18(pretrained=True)
# Модификация последнего слоя модели
# У ResNet-18 последний слой — это fully connected layer (fc)
# Мы заменяем его на новый слой с количеством выходов, равным числу наших классов
num_ftrs = model.fc.in_features  # Получаем количество входных признаков
model.fc = nn.Linear(num_ftrs, 2)  # Заменяем fc на новый слой с 2 выходами (для двух классов)

# Перемещаем модель на GPU, если доступно
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model = model.to(device)
```

Шаг 3: Настройка функции потерь и оптимизатора

Используем функцию потерь `CrossEntropyLoss`, которая подходит для задач классификации. Оптимизатор — это стохастический градиентный спуск (SGD) с моментом. Планировщик уменьшает скорость обучения каждые 7 эпох, что помогает улучшить сходимость

```
# Настройка функции потерь и оптимизатора
criterion = nn.CrossEntropyLoss()  # Функция потерь для задачи классификации
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)  # Оптимизатор SGD

# Планировщик скорости обучения (уменьшает LR каждые 7 эпох)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=7, gamma=0.1)
```

Шаг 4: Обучение модели

Модель обучается в течение заданного числа эпох. В каждой эпохе есть две фазы: обучение и валидация. Для обучения используется механизм обратного распространения ошибки, а для валидации — только прямой проход.

```
def train_model_with_visualization(model, criterion, optimizer, scheduler, num_epochs=10):
    best_model_wts = model.state_dict()
    best_acc = 0.0

    # Списки для хранения потерь и точности
    train_losses, val_losses = [], []
    train_accuracies, val_accuracies = [], []

    for epoch in range(num_epochs):
        print(f'Epoch {epoch+1}/{num_epochs}')
        print('-' * 10)
```

```
    for phase in ['train', 'val']:
        if phase == 'train':
            model.train()
        else:
            model.eval()

        running_loss = 0.0
        running_corrects = 0

        for inputs, labels in dataloaders[phase]:
            inputs = inputs.to(device)
            labels = labels.to(device)

            optimizer.zero_grad()

            with torch.set_grad_enabled(phase == 'train'):
                outputs = model(inputs)
                _, preds = torch.max(outputs, 1)
                loss = criterion(outputs, labels)

                if phase == 'train':
                    loss.backward()
                    optimizer.step()

            running_loss += loss.item() * inputs.size(0)
            running_corrects += torch.sum(preds == labels.data)

        if phase == 'train':
            scheduler.step()

        epoch_loss = running_loss / dataset_sizes[phase]
        epoch_acc = running_corrects.double() / dataset_sizes[phase]

        print(f'{phase} Loss: {epoch_loss:.4f} Acc: {epoch_acc:.4f}')

        # Сохраняем потери и точность
        if phase == 'train':
            train_losses.append(epoch_loss)
            train_accuracies.append(epoch_acc)
        else:
            val_losses.append(epoch_loss)
            val_accuracies.append(epoch_acc)

        if phase == 'val' and epoch_acc > best_acc:
            best_acc = epoch_acc
            best_model_wts = model.state_dict()

# Визуализация потерь и точности
plt.figure(figsize=(12, 5))
```

```
    plt.subplot(1, 2, 1)
    plt.plot(train_losses, label='Train Loss')
    plt.plot(val_losses, label='Validation Loss')
    plt.title('Loss over epochs')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot([acc.cpu().numpy() for acc in train_accuracies], label='Train
Accuracy')
    plt.plot([acc.cpu().numpy()     for     acc     in     val_accuracies],
label='Validation Accuracy')
    plt.title('Accuracy over epochs')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()

    plt.show()

    print(f'Best val Acc: {best_acc:.4f}')
    model.load_state_dict(best_model_wts)
    return model


# 6. Запуск обучения
model = train_model(model, criterion, optimizer, scheduler, num_epochs=10)
```

Шаг 5: Сохранение модели :

После завершения обучения сохраняются веса лучшей модели.

```
torch.save(model.state_dict(), './best_model.pth')
```

Работа с собственными наборами данных (Custom Datasets)

При работе с глубоким обучением иногда требуется использовать собственные наборы данных, которые не входят в стандартные наборы данных библиотек. PyTorch предоставляет удобный способ работать с собственными данными путем создания пользовательских наборов данных.

Что такое пользовательский набор данных

Пользовательский набор данных — это набор данных, относящихся к конкретной проблеме, над которой вы работаете.

По сути, **пользовательский набор данных** может состоять практически из чего угодно.

Например, если бы мы создавали приложение для классификации изображений продуктов питания, такое как Nutrify , наш пользовательский набор данных мог бы состоять из изображений продуктов питания.

Для модели классификации того, является ли текстовый отзыв на веб-сайте положительным или отрицательным, пользовательский набор данных мог бы включать примеры существующих отзывов клиентов и их оценок.

Для приложения для классификации звуков, пользовательский набор данных мог бы содержать образцы звуков вместе с их метками.

В системе рекомендаций для клиентов, приобретающих товары на веб-сайте, пользовательский набор данных мог бы включать примеры товаров, купленных другими людьми.

PyTorch включает в себя множество существующих функций для загрузки различных пользовательских наборов данных в библиотеках доменов TorchVision, TorchText и TorchAudioTorchRec

Однако иногда существующих функций может быть недостаточно.

В этом случае мы всегда можем создать подкласс torch.utils.data.Dataset и настроить его по своему вкусу.

Что такое Dataset и DataLoader в PyTorch

В прошлом модуле мы уже говорили о Dataset и DataLoader. Раскроем тему нем ного подробнее.

PyTorch предоставляет два ключевых класса для работы с данными:

- **Dataset**: Абстрактный класс для представления набора данных. Пользователь может создать собственный подкласс Dataset, реализовав методы __len__ и __getitem__.
- **DataLoader**: Обеспечивает итерацию по данным, включая поддержку мини-батчей, многопоточной загрузки и перемешивания данных.

В PyTorch основным абстрактным классом для пользовательских наборов данных является torch.utils.data.Dataset. Этот класс предоставляет интерфейс, который позволяет пользовательским классам поддерживать индексирование данных и возможность получения их длины. Он позволяет вам определить, как ваши данные должны быть прочитаны, преобразованы и доступны. С другой стороны, класс DataLoader предоставляет эффективный способ итерации по вашему набору данных в пакетах, что имеет решающее значение для обучения моделей.

Для создания пользовательского класса Dataset необходимо унаследоваться от torch.utils.data.Dataset и реализовать два основных метода: __len__ для возвращения длины набора данных и __getitem__ для получения элемента по индексу.

Пример использования встроенного Dataset:

```
from torchvision import datasets
from torchvision.transforms import ToTensor

# Загрузка встроенного набора данных MNIST
train_data = datasets.MNIST(root="data", train=True, download=True, transform=

# Создание DataLoader
from torch.utils.data import DataLoader

train_loader = DataLoader(train_data, batch_size=32, shuffle=True)

# Итерация по DataLoader
for batch in train_loader:
    images, labels = batch
    print(images.shape, labels.shape)
    break
```

Реализация собственного Dataset и DataLoader

Структура пользовательских данных

Прежде чем реализовать класс Dataset, необходимо правильно организовать данные. Пример структуры для задачи классификации изображений:

```
custom_dataset/
|-- train/
|   |-- class_1/
|   |-- class_2/
|-- test/
|   |-- class_1/
|   |-- class_2/
```

Для работы с такими данными PyTorch предоставляет утилиту ImageFolder, но мы сосредоточимся на создании полностью кастомного Dataset.

Реализация набора данных и загрузчика данных в PyTorch выглядит следующим образом:

Шаг 1: Импорт необходимых библиотек

Сначала убедитесь, что у вас импортированы необходимые библиотеки:

```
import torch
from torch.utils.data import Dataset, DataLoader
import numpy as np
```

Шаг 2: Определение класса вашего пользовательского набора данных

Чтобы создать пользовательский набор данных, вам необходимо определить класс, который наследуется от torch.utils.data.Dataset. Этот класс должен реализовывать три метода: `__init__`, `__len__`, и `__getitem__`.

- **__init__** инициализирует набор данных со всеми необходимыми атрибутами, такими как пути к файлам или этапы предварительной обработки данных.
- **__len__** возвращает общее количество образцов в наборе данных.
- **__getitem__** извлекает выборку из набора данных по заданному индексу.

```
class CustomDataset(Dataset):
    def __init__(self, data, labels):
        self.data = data
        self.labels = labels

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

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

Шаг 3: Подготовка данных

Далее подготовьте данные и метки. Для демонстрационных целей мы создадим случайные данные с помощью NumPy:

```
данные = np.random.randn ( 100 , 3 , 32 , 32 )   # 100 образцов изображений 3x32x32
labels = np . random . randint ( 0 , 10 , size = ( 100 ,))   # 100 меток в диапазоне
0-9
```

Шаг 4: Создание экземпляра вашего набора данных

Создайте экземпляр вашего пользовательского набора данных с подготовленными данными:

```
dataset = CustomDataset(data, labels)
```

Шаг 5: Создание DataLoader

Класс DataLoader обрабатывает пакетирование, перемешивание и загрузку данных параллельно. Для DataLoader необходимо задать:

- **batch_size.** Указывает количество образцов в партии.
- **shuffle**. Если установлено значение True, данные будут перемешиваться в каждой эпохе.
- **num_workers**. Указывает количество подпроцессов, используемых для загрузки данных.

```
dataloader = DataLoader(dataset, batch_size=4, shuffle=True, num_workers=2)
```

Шаг 6: Итерация через DataLoader

Теперь вы можете пройтись по DataLoader в вашем цикле обучения. Каждая итерация даст пакет данных и соответствующие метки:

```
for batch_data, batch_labels in dataloader:
    print(batch_data.shape, batch_labels.shape)
    # Add your training code here
```

Полный код с моделью и загрузчиком данных выглядит так:

```python
# Define a simple CNN model
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 3, 1)
        self.conv2 = nn.Conv2d(16, 32, 3, 1)
        self.fc1 = nn.Linear(32*6*6, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.max_pool2d(x, 2, 2)
        x = torch.relu(self.conv2(x))
        x = torch.max_pool2d(x, 2, 2)
        x = torch.flatten(x, 1)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

model = SimpleCNN()

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop
num_epochs = 5

for epoch in range(num_epochs):
    for batch_data, batch_labels in dataloader:
        # Convert numpy arrays to torch tensors
        batch_data = torch.tensor(batch_data, dtype=torch.float32)
        batch_labels = torch.tensor(batch_labels, dtype=torch.long)

        # Zero the parameter gradients
        optimizer.zero_grad()

        # Forward pass
        outputs = model(batch_data)
        loss = criterion(outputs, batch_labels)

        # Backward pass and optimization
        loss.backward()
        optimizer.step()

    print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')
```

Использование классов PyTorch Dataset и DataLoader для пользовательских данных упрощает процесс загрузки и предварительной обработки данных. Определив пользовательский набор данных и используя DataLoader, вы можете эффективно обрабатывать большие наборы данных и сосредоточиться на разработке и обучении своих моделей. Независимо от того, работаете ли вы с изображениями, текстом или

другими типами данных, эти классы обеспечивают надежную структуру для обработки данных в PyTorch. Этот комплексный подход гарантирует, что ваш конвейер данных будет эффективным, масштабируемым и простым в обслуживании, что позволит вам сосредоточиться на создании и улучшении своих моделей.

Деплоймент моделей PyTorch

Деплоймент моделей PyTorch - это процесс предоставления обученной модели в продакшен для использования в реальном времени. Это важный шаг в разработке искусственного интеллекта, который позволяет применять модель для решения задач в реальном мире.

Деплоймент моделей машинного обучения — это процесс, с помощью которого обученные модели интегрируются в реальные приложения. Важность этого этапа трудно переоценить: он позволяет использовать результаты исследований и экспериментов для решения прикладных задач.

PyTorch является одним из самых популярных фреймворков для разработки моделей благодаря своей гибкости и удобству. Однако деплоймент моделей PyTorch требует понимания различных инструментов, таких как TorchScript, ONNX, и серверные платформы (например, TorchServe).

Основы деплоймента моделей

Деплоймент необходим для того, чтобы перенести модель из стадии разработки в реальный мир, где она может быть использована для решения прикладных задач. Рассмотрим основные цели и сценарии применения:

Цели деплоймента:

- Интеграция модели в производственные системы, такие как веб-приложения, мобильные приложения, IoT-устройства.
- Обеспечение доступа к модели через API, чтобы внешние системы могли отправлять запросы и получать результаты.
- Оптимизация модели для работы в условиях реального времени, включая снижение задержек и повышение пропускной способности.

Масштабирование, позволяющее обрабатывать множество одновременных запросов и обеспечивать высокую доступность системы.

Типичные сценарии использования:

- **Веб-сервисы**, например, рекомендательные системы, системы поиска изображений или чат-боты.
- **Встроенные устройства** такие как умные камеры, голосовые помощники и другие устройства с ограниченными вычислительными ресурсами.
- **Аналитические системы** на основе временных рядов, обработки больших данных или анализа текста.

Задачи и вызовы деплоймента

Деплоймент моделей машинного обучения связан с рядом технических задач и вызовов, которые необходимо учитывать для успешной реализации.

Оптимизация модели:

- Уменьшение размера модели для более быстрого выполнения и экономии ресурсов.
- Преобразование модели в формат, который совместим с целевой платформой (например, TorchScript или ONNX).

Производительность:

- Обеспечение низкой задержки (латентности), что критично для приложений реального времени.
- Высокая пропускная способность, позволяющая обрабатывать множество запросов одновременно.
- Эффективное использование вычислительных ресурсов, включая процессоры (CPU) и графические процессоры (GPU).

Масштабирование:

- Возможность горизонтального масштабирования за счёт добавления серверов или контейнеров.
- Интеграция с системами оркестрации, такими как Kubernetes, для управления инфраструктурой.

Надёжность и обновления:

- Обеспечение устойчивости системы к сбоям.
- Возможность плавного обновления модели без остановки сервиса.

Эти аспекты требуют планирования и использования специализированных инструментов, которые мы рассмотрим далее в лекции.

Процесс деплоймента моделей

1. **Выбор модели.** Первым шагом является выбор подходящей модели PyTorch для деплоя. Модель должна быть обучена на достаточном объеме данных и показывать хорошие результаты на валидационном наборе.

2. **Экспорт модели**. Для деплоя модели ее необходимо экспортировать в формат, который поддерживается на платформе, где она будет использоваться. Например, модель можно экспортировать в ONNX формат.

3. **Интеграция модели**. После экспорта модели необходимо интегрировать ее в целевое приложение или сервис. Это может включать в себя написание кода для загрузки модели, предобработки данных и вывода результатов.

4. **Оптимизация модели**. Для эффективного деплоя модели на производство ее необходимо оптимизировать. Можно провести квантизацию модели, уменьшить размер модели или использовать специальные библиотеки для оптимизации.

5. **Тестирование и внедрение**. После интеграции модели необходимо провести тестирование, чтобы убедиться, что она работает правильно и дает точные результаты. Затем происходит внедрение модели на производство для использования в реальных условиях.

Инструменты деплоймента моделей

TorchScript

TorchScript — это промежуточное представление PyTorch-моделей, которое делает их независимыми от Python.

Преимущества TorchScript:

- Ускорение выполнения.
- Возможность использования вне Python-окружения.
- Удобство интеграции с C++.

Два способа конвертации:

1. **Трассировка (Tracing)**: фиксирует вычислительный граф, используя пример входных данных.

```
import torch

class MyModel(torch.nn.Module):
    def forward(self, x):
        return x * 2

model = MyModel()
traced_model = torch.jit.trace(model, torch.randn(1))
traced_model.save("model.pt")
```

2. **Скриптинг (Scripting)**: анализирует модель на основе её структуры кода.

```
scripted_model = torch.jit.script(model)
scripted_model.save("model_scripted.pt")
```

ONNX (Open Neural Network Exchange)

ONNX — это формат для интероперабельности моделей машинного обучения.

Преимущества ONNX:

- Поддержка разных фреймворков (TensorFlow, PyTorch).
- Оптимизация с помощью ONNX Runtime.

Экспорт модели в ONNX:

```
torch.onnx.export(
    model,
    torch.randn(1, 3, 224, 224),
    "model.onnx",
    export_params=True,
    opset_version=11
)
```

Инструменты для работы с ONNX:

- ONNX Runtime: ускорение выполнения.
- Netron: визуализация графа модели.

TorchServe

TorchServe — это инструмент для создания серверов, обслуживающих PyTorch-модели.

Основные функции:

- Хостинг нескольких моделей.
- Обработка запросов REST API.
- Мониторинг и логирование.

Шаги развертывания с TorchServe:

1. Сохранение модели:

   ```
   torch-model-archiver --model-name my_model --version 1.0
   --serialized-file model.pt --handler handler.py
   ```

2. Запуск сервера:

   ```
   torchserve --start --model-store ./model_store --models
   my_model=my_model.mar
   ```

Практические аспекты деплоймента

Оптимизация моделей

Сжатие моделей:

- **Квантизация**: уменьшение разрядности весов и активаций. Это снижает объём памяти и ускоряет выполнение.

Пример динамической квантизации:

```
from torch.quantization import quantize_dynamic
quantized_model = quantize_dynamic(model, {torch.nn.Linear}, dtype=
```

Статическая квантизация выполняется с использованием калибровочных данных для определения диапазонов значений весов и активаций.

Пост-тренировочная квантизация позволяет применять квантизацию после обучения модели.

- **Проределение (Pruning)** – удаление наименее значимых весов модели. Это уменьшает её сложность, сохраняя при этом точность на приемлемом уровне.
- **Сжимающие алгоритмы**. Использование техник, таких как Huffman-кодирование или кластеризация весов, для уменьшения размера модели.

Конвертация форматов:

- Экспорт модели в PyTorch Mobile для интеграции в мобильные приложения.

```
pytorch_mobile_export model.pt --output-format mobile.pt
```

- Использование NVIDIA TensorRT для высокопроизводительного выполнения на GPU.

```
from torch2trt import torch2trt
model_trt = torch2trt(model, [torch.randn(1, 3, 224, 224)])
```

Обеспечение надёжности

Обработка ошибок:

- Валидация входных данных для предотвращения некорректных запросов.
- Логирование ошибок и исключений для упрощения диагностики.

Резервирование и отказоустойчивость:

- Использование систем балансировки нагрузки, таких как Nginx или AWS Elastic Load Balancer, для распределения трафика.
- Внедрение резервных копий и дублирования серверов, чтобы обеспечить бесперебойную работу системы при сбоях.

Обновления моделей:

- Реализация механизмов A/B-тестирования для безопасного внедрения новых версий моделей.
- Использование канареечных развёртываний, при которых обновления распространяются постепенно.

Мониторинг и обновления

- **Метрики производительности**: время отклика (latency), количество запросов в секунду (throughput), загрузка ресурсов (CPU, GPU, память).
- **Инструменты мониторинга**:
    - Prometheus и Grafana для визуализации метрик.
    - Использование ELK Stack (Elasticsearch, Logstash, Kibana) для централизованного логирования.
- **Автоматизация обновлений**. Интеграция с системами CI/CD (например, GitHub Actions, Jenkins) для автоматического развертывания новых версий.
- **Уведомления**. Настройка оповещений в случае сбоев или ухудшения производительности через Slack, электронную почту или PagerDuty.

Кейсы использования

Веб-приложение для классификации изображений

- Использование Flask или FastAPI для создания API.
- TorchServe для хостинга модели.

Мобильное приложение для предсказаний

- PyTorch Mobile и оптимизация для ограниченных ресурсов.

Аналитическая система

- Использование ONNX Runtime для интеграции в аналитические пайплайны.

Деплоймент моделей PyTorch — это важный этап на пути от экспериментов к реальным решениям. Понимание ключевых инструментов и подходов позволяет создавать надёжные, масштабируемые и производительные системы.

Заключение

Сегодняшняя лекция познакомила вас с продвинутыми аспектами работы с PyTorch, что позволит вам более уверенно применять этот фреймворк для решения сложных задач машинного обучения. Мы разобрали ключевые этапы работы с PyTorch:

1. **Перенос обучения:** мы рассмотрели, как использовать предварительно обученные модели для новых задач, что значительно сокращает время и ресурсы, необходимые для обучения. Вы также научились изменять и адаптировать существующие архитектуры для ваших собственных нужд.

2. **Работа с пользовательскими наборами данных:** мы углубились в процесс создания и обработки собственных наборов данных с использованием классов Dataset и DataLoader. Эти инструменты дают вам возможность работать с нестандартными форматами данных и применять разнообразные преобразования для улучшения качества обучения.

3. **Деплоймент моделей:** мы обсудили, как подготовить модель для использования в производственной среде, включая экспорт в формат TorchScript или ONNX, интеграцию через веб-приложения и оптимизацию для ускорения инференса. Эти навыки критически важны для внедрения машинного обучения в реальные приложения.

Основные выводы и рекомендации:

- PyTorch предоставляет мощный инструментарий для всех этапов работы с нейронными сетями: от разработки моделей до их внедрения.

- Регулярная практика — ключ к успеху. Экспериментируйте с настройками моделей, оптимизаторами и функциями потерь, чтобы лучше понять их влияние на результаты.

- Используйте перенос обучения, чтобы ускорить решение задач, особенно когда объем данных ограничен.

- Создание и обработка пользовательских наборов данных дает вам свободу работы с уникальными задачами и улучшает качество обучения моделей.

- Деплоймент моделей требует внимания к оптимизации, что помогает сделать решения быстрыми и эффективными.

PyTorch сочетает гибкость и производительность, предоставляя все необходимое для исследований и разработки современных систем машинного обучения. Теперь, когда вы освоили ключевые аспекты работы с этим инструментом, следующая цель — начать применять полученные знания в ваших собственных проектах.

Помните: машинное обучение — это не только математика и код, но и искусство поиска лучших решений для сложных задач. Удачи вам на этом пути!