Многозадачное обучение языковых моделей, основанных на механизме внутреннего внимания

Погорельцев С. А., н.р. Полякова И. Н.

МГУ им. Ломоносова, ф-т ВМК, каф. АЯ

2022

Что такое многозадачное обучение?

TODO

Формализация понятия задача

$$T_i = \{p_i(x), p_i(y|x), L_i\} \tag{1}$$

- $p_i(x)$ - распределение входных данных
- $p_i(y|x)$ - распределение меток (классификация) / значений (регрессии) в зависимости от входных данных
- L_i - функция потерь

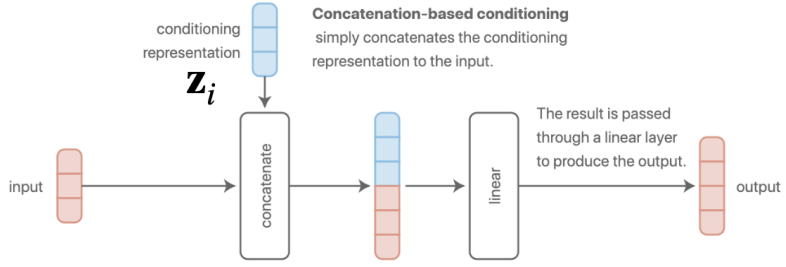
Выборки для задачи

- D_i^{train} - обучающая
- D_i^{val} - валидационная
- D_i^{test} - тестовая

Гипотезы многозадачной модели (Hard Parameter Sharing)

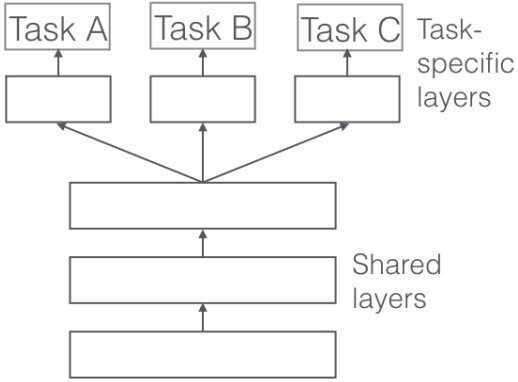
Основной вопрос: **Как организовать "переключение" на уровне модели?**

- Конкатенация one-hot вектора задачи к скрытому представлению



- Добавление вектора смещения - смещение данных задачи
- Умножение - изменение масштаба данных задачи
- Архитектура с несколькими головами (популярна, хорошо работает на практике, в дальнейшем буду говорить о ней, т. к. использую её в работе)

Архитектура с несколькими головами



Введём некоторые обозначения и уточним понятие задачи

- H_i - специфичные для задачи слои - голова модели
- E - общие слои, формирующие представления для голов H_i

Тогда i-ая задача:

$$T_i = \{p_i(x), p_i(y|x), H_i, L_i\} \qquad (2)$$

Forward pass для задачи T_i

$$F_i(x) = H_i \circ E \circ x = H_i(E(x)) \qquad (3)$$

Обучение многоголовой модели (популярные подходы)

- **Одна задача на батч** Данные всех задач разбиваются на батчи и эти батчи перемешиваются вместе. На каждой эпохе выбирается случайный j-ый батч соотвествующий задаче T_i $batch_j^i$, а затем всё (почти) как обычно.
- **Оптимизация (взвешенной) суммы функций потерь**
  - Собираются "метабатчи" $\{\text{batch}_{j_1}^1, \text{batch}_{j_2}^2, ..., \text{batch}_{j_n}^n\}$ - множество по 1 случайному батчу для каждой задачи (здесь n штук).
  - Для каждой i-ой задачи вычисляется $fwd_i = F_i(\text{batch}_{j_i}^i)$, $loss_i = L_i(f_i)$ и оптимизируется $loss = \sum_{i=1}^n w_i \, \text{loss}_i$, где $\forall i \; w_i > 0$.
  - w_i регулирует важность i-ой задачи при оптимизации

Я экспериментирую с обоими и кажется, что оптимизация суммы работает лучше ~~что в целом логично, т. к. первый вариант с точки зрения метоптов - тихий ужас~~

Алгоритм обучения с оптимизацией по одной задаче

for epochNum $\leftarrow \overline{0, N}$ **do**
  for all $\text{batch}_j^i \in$ *batches.shuffle*() **do**
    $outputs \leftarrow F_i(batch_j^i)$
    $\text{loss} \leftarrow L_i(outputs)$
    $\text{F}_\text{i} \leftarrow \text{F}_\text{i} - \alpha \nabla \text{loss}$ ▷ Тут может быть любой метод оптимизации
  end for
end for

Minibatch градиентный спуск здесь для упрощения примера, чаще используется Adam или AdamW для обучения Трансформеров (в т.ч. многозадачных)

Алгоритм обучения с оптимизацией по сумме функций потерь

for epochNum $\leftarrow \overline{0, N}$ **do** ▷ Итерация по эпохам
  for all metabatch$_j \in$ *metabatches.shuffle*() **do** ▷ Итерация по "метабатчам"
    loss $\leftarrow 0$
    for all batch$_j^i \in$ metabatch$_j$ **do** ▷ Итерация по задачам и их батчам
      $outputs_j^i \leftarrow F_i(\text{batch}_j^i)$
      $loss_j \leftarrow loss_j + w_i L_i(outputs_j^i)$
    end for
    loss$_j$.*backward*(); *optimizer.step*(); *optimizer.zero_grad*() ▷ *
  end for
end for

*: магия автоматического дифференцирования PyTorch, а на самом деле

$$\nabla loss_j(x^{i,j}) = \nabla \sum_{i=1}^{n} w_i L_i(H_i(E(x^{i,j}))$$ (продолжение на след. слайде)

Подробнее про обновление весов для суммы

Рассмотрим частную производную для k-ой координате

$$(\nabla loss_j(x^{i,j}))_k = \frac{\partial}{\partial x_k^{i,j}} \sum_{i=1}^{n} w_i L_i(H_i(E(x^{i,j})) = \sum_{i=1}^{n} w_i \frac{\partial L_i(H_i(E(x^{i,j}))}{\partial x_k^{i,j}}$$

Дальше считается как обычно, как и в предыдущем случае градиент от конкретной функции потерь с использованием правила цепочки $\frac{\partial y}{\partial x} = \frac{\partial y}{\partial z}\frac{\partial z}{\partial x}$
После вычисления градиента делается шаг метода оптимизации обновляются все веса и для общих слоёв, и для **каждой головы** (backpropagation)

- Полученный метод - обобщение первого (если w - one-hot вектор, то получаем первый метод буквально)
- Снижается проблема забывания, поскольку оптимизация всего сразу
- Возможно, стоит совместить первый и второй подходы
  - Сначала сделать базовую оптимизацию для всех задач
  - Затем заниматься более тонкой настройкой позадачных весов

Некоторые важные особенности

- **Transfer Learning**: знания для одной задачи помогают в решении другой
- **Representation Learning**: перенос знаний часто связан с выучиванием общими слоями хороших векторных представлений входных признаков
- **Регуляризация**: многозадачное обучение работает как регуляризация и в случае переобучения всегда можно добавить задач для улучшения генерализации или сделать общими больше слоёв
- **Эффективное использование**: модель одна, пусть и крупнее

Сложности

- **Negative Transfer**: бывает, что задачи отрицательно влияют на решение друг друга. Наиболее вероятные причины:
  - Недостаточно выразительная модель (часто многозадачные модели больше)
  - Сложности с оптимизацией (межзадачная интерференция, задачи могут обучаться с разной скоростью)
- **Совместность задач**: какие задачи будут хорошо работать вместе? На этот вопрос ответ пока можно искать только экспериментами (за 1 эпоху понятно)

Проблема со средствами многозадачного обучения

Во время первых экспериментов обнаружил, что приходится писать много однообразного и сложного кода, потому, что хороших библиотек для многозадачного обучения Трансформеров пока нет
В итоге, была разработна (и продолжает развиваться) небольшая библиотека вокруг экосистемы предобученных моделей и датасетов Hugging Face и PyTorch, которая позволяет декларативно описывать задачи и сама собирает модель, готовит данные с учётом многозадачности

- Высокоуровневые готовые конфигурации для типичных задач и удобные "ручки" для низкоуровневой настройки задачи
  - Базовая модель - предобученный трансформер из Hugging Face Transformers
  - Голова - torch.nn.Module, Hugging Face или готовая
  - Функция потерь - из модели Hugging Face, готовые или torch.nn.Module
  - Данные - Hugging Face Datasets (готовый или свой)
  - Метрики - свой легко расширяемый велосипед
- Обучение с многозадачным аналогом Trainer или обычный цикл PyTorch

Пример конфигурации (3 классификации из Russian SuperGLUE)

TODO

Низкоуровневые возможности конфигурации

TODO

Обучение с циклом на PyTorch

TODO

Высокоуровневое обучение (сейчас временно недоступно)

TODO

Готовность и открытый доступ

TODO

Цели и планы

TODO

Работоспособность метода: сходимость

TODO

Работоспособность метода: достижимость хорошего качества

TODO

Эксперименты с оптимизацией

TODO

Эксперименты с архитектурой голов

TODO

Что ещё хочется успеть

TODO