

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

Многозадачное обучение

Многозадачное обучение (MultiTask Learning) - подполе машинного обучения, в котором одновременно решаются несколько задач, при этом используются общие черты и различия между задачами. Это может привести к повышению эффективности обучения и точности прогнозирования для моделей, специфичных для конкретных задач, по сравнению с отдельным обучением моделей.

Языковые модели, основанные на механизме внутреннего внимания

Да да, я про Трансформеры и в частности BERT-подобные модели

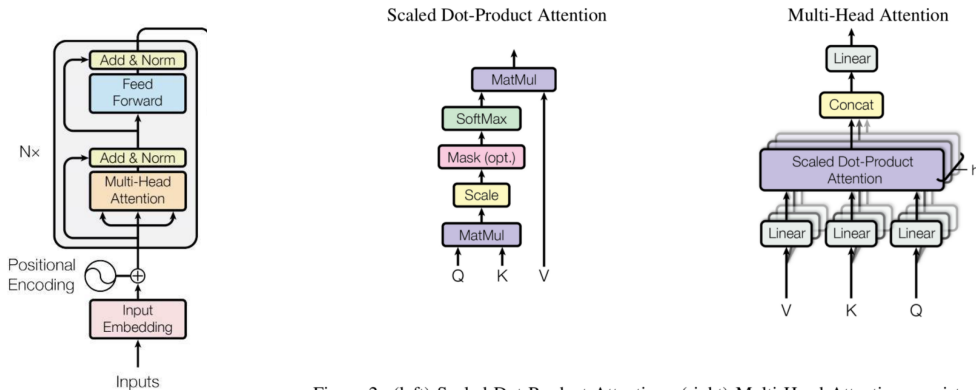


Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

Формализация понятия задача

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

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

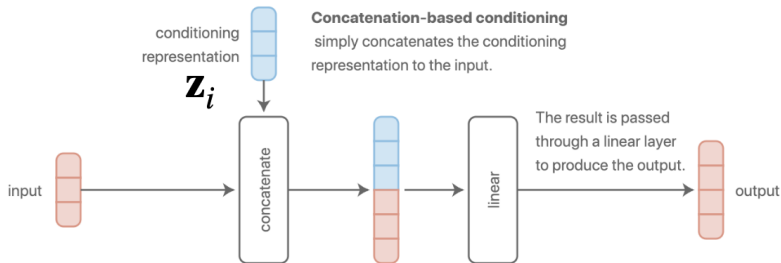
Выборки для задачи

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

Гипотезы многозадачной модели (Hard Parameter Sharing)

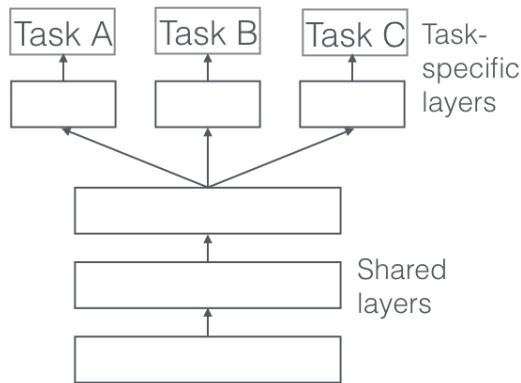
Основной вопрос: Как организовать "переключение" на уровне модели?

- 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:

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

Forward pass для задачи T_i

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

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 соответствующий задаче T_i $batch_j^i$, а затем всё (почти) как обычно.
- **Оптимизация (взвешенной) суммы функций потерь**
 - Собираются "метабатчи" $\{batch_{j_1}^1, batch_{j_2}^2, \dots, batch_{j_n}^n\}$ - множество по 1 случайному батчу для каждой задачи (здесь n штук).
 - Для каждой i -ой задачи вычисляется $fwd_i = F_i(batch_{j_i}^i)$, $loss_i = L_i(f_i)$ и оптимизируется $loss = \sum_{i=1}^n w_i loss_i$, 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  $batch_j^i \in batches.shuffle()$  do
     $outputs \leftarrow F_i(batch_j^i)$ 
     $loss \leftarrow L_i(outputs)$ 
     $F_i \leftarrow F_i - \alpha \nabla loss$ 
  end for
end for
```

▷ Тут может быть любой метод оптимизации

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 metabatchj  $\in$  metabatches.shuffle() do ▷ Итерация по "метабатчам"  
    loss  $\leftarrow 0$   
    for all batchji  $\in$  metabatchj do ▷ Итерация по задачам и их батчам  
      outputsji  $\leftarrow F_i(\text{batch}_j^i)$   
      lossj  $\leftarrow \text{loss}_j + w_i L_i(\text{outputs}_j^i)$   
    end for  
    lossj.backward(); optimizer.step(); optimizer.zero_grad() ▷ *  
  end for  
end for
```

*: магия автоматического дифференцирования PyTorch, а на самом деле

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

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 (или я плохо искал). В итоге, была разработана (и продолжает развиваться) небольшая библиотека, которая позволяет декларативно описывать задачи и сама собирает модель, готовит данные с учётом многозадачности

- Высокоуровневые готовые конфигурации для типичных задач
- "Ручки" для низкоуровневой настройки задач
- Средства для обучения в обычном цикле PyTorch
- Обучение с многозадачным аналогом Trainer (сейчас перерабатывается)

Пример простой конфигурации задач

```
tasks = Tasks([
    SequenceClassificationTask(
        name="danetqa",
        dataset_dict=load_dataset(...),
        preprocessor=Preprocessor([preprocess_danetqa]),
        tokenizer_config=TokenizerConfig(max_length=512),
    ),
    SequenceClassificationTask(
        name="headline_cause",
        num_labels=3,
        dataset_dict=load_dataset(...),
        preprocessor=Preprocessor([preprocess_headline_cause]),
        tokenizer_config=cfg
    )], model_path = "DeepPavlov/rubert-base-cased")
```

Низкоуровневая конфигурация задачи

```
Task(  
    name = "название задачи (для индексации и логов)",  
    head = <объект подкласса torch.nn.Module> | HFHead,  
    data = Data(  
        dataset_dict = <датасет в формате datasets.DatasetDict>,  
        # другое (не обязательно, задано по-умолчанию)  
        configured_tokenizer = ConfiguredTokenizer(  
            model_path, padding = False, truncation = True, max_length  
            ... # другие параметры токенизатора Hugging Face  
        ),  
        preprocessor = Preprocessor(...),  
        columns, # поля, которые нужно приводить к формату PyTorch  
        collator_class, collator_config  
    )  
)
```

Конфигурация головы модели

- Готовая из transformers

```
head = HFHead(class_ = AutoModelForSequenceClassification,  
               config_params = {"num_labels": 2})
```

- Свой torch.nn.Module

```
class TwoLinears(nn.Module):  
    def __init__(self, ...):  
        layers = [nn.Linear(768, 2048), F.relu, nn.Linear(768, 2)]  
        self.layers = nn.ModuleList(layers)  
        self.loss = nn.CrossEntropyLoss()  
    def forward(self, encoder_outputs, labels, ...):  
        x = encoder_outputs[1]  
        for layer in self.layers: x = layer.forward(x)  
        return {"logits": x,  
                "loss": self.loss(logits.view(-1, num_labels),  
                                   labels.view(-1))}  
  
head = TwoLinears(...)
```

Обучение с циклом на PyTorch

```
train_sampler = MultitaskBatchSampler(tasks.data, "train", batch_size=12)
model = MultitaskModel(encoder_path, tasks.heads)
# ... инициализация метода оптимизации, перенос модели на GPU ...
for epoch_num in range(num_epochs):
    for batch in train_sampler:
        batch.data.to(device)
        outputs = model.forward(batch.name, **batch.data)
        loss = outputs.loss
        print(f"Training loss: {loss} on task {batch.name}")
        loss.backward()
        optimizer.step()
        lr_scheduler.step()
        optimizer.zero_grad()
```


Готовность и открытый доступ

- Код в открытом доступе
`https://github.com/s1m0000n/multitask-transformers`
- Буду рад новым контрибьюторам
- Нормальное состояние кодовой базы (pylint > 8)
- Документация в зачаточном состоянии
- Новые вещи реализуются по мере того, что я использую для ВКР
- Планируется публикация статьи