Основная проблема Masked Language Model (MLM)

Проблема: В BERT мы маскируем некоторые токены и учим модель их предсказывать. Но функция потерь (loss) считается только для замаскированных позиций. Получается парадокс: а что происходит с представлениями (representations) для всех остальных, незамаскированных токенов? Они могут оказаться плохого качества, потому что модель их "не учит".

Первое решение и новая проблема

Решение 1: Давайте считать cross-entropy loss не только для маскированных токенов, но и для некоторых немаскированных тоже.

Но тут возникает новая проблема: Модель может схитрить и просто копировать немаскированные токены, даже не анализируя контекст. Ведь правильный ответ уже есть на входе!

Например:

- Вход: "Кот [MASK] на диване"
- Модель для слова "кот": просто выводит "кот", не думая о контексте

Финальное решение

Решение 2: Иногда заменяем немаскированные токены на случайные слова. Тогда модель не может просто копировать и должна использовать контекст.

Denoising Autoencoder

В итоге получается **шумоподавляющий автоэнкодер**:

1. Берем исходный текст
2. Портим его: заменяем часть токенов на [MASK] или случайные слова
3. Учим модель восстанавливать исходный текст

Пример:

- Исходный текст: "Собака лает во дворе"
- Испорченный: "Собака [MASK] во автомобиль"
- Задача: восстановить "Собака лает во дворе"

Это заставляет модель действительно понимать контекст и создавать качественные представления для всех позиций, а не только маскированных.

При обучении BERT пишут что в 80% случаев слово заменяют на маску, в 10% случаев меняют на случайное слово и в 10% случаев оставлюят как есть - у меня вопросы

1. Как при обучении модель понимает, что слово заменено на случайное (как это технически делается)
2. Я занаю что в 15% случаев слово меняется на маску - как это согласуется с тем что написано выше?

Отличные вопросы! Они касаются ключевых деталей механизма Masked Language Model (MLM), который является основой предобучения BERT. Давайте разберем каждый из них по порядку.

1. Как модель понимает, что слово заменено на случайное?

Короткий ответ: Модель *не знает* заранее, было ли слово заменено на маску, случайное слово или оставлено как есть. Её задача — *угадать оригинальное слово* на этой позиции, исходя из контекста. Именно в процессе этого "угадывания" она и учится различать корректный и некорректный контекст.

Техническая реализация:

1. **Подготовка данных:** Допустим, у нас есть предложение: `"The quick brown fox jumps over the lazy dog."`
2. **Выбор слов для маскирования:** Случайным образом выбирается 15% от всех токенов в предложении. Для простоты, пусть мы выбрали слово `"fox"`.
3. **Применение стратегии замены:** Теперь к выбранному слову применяется правило 80/10/10:
   - **80% (Вероятность 0.8):** Заменяем на `[MASK]`.
     - Результат: `"The quick brown [MASK] jumps over the lazy dog."`
   - **10% (Вероятность 0.1):** Заменяем на *случайное* слово из словаря. Например, на слово `"plane"`.
     - Результат: `"The quick brown plane jumps over the lazy dog."`
   - **10% (Вероятность 0.1):** Оставляем без изменений, то есть `"fox"`.
     - Результат: `"The quick brown fox jumps over the lazy dog."`
4. **Задача для модели:** Модель BERT получает на вход *искаженное* предложение (с [MASK], случайным словом или оригинальным). Её задача — для каждой из позиций, которые мы изначально выбрали (в нашем случае это позиция слова `"fox"`), предсказать **оригинальное слово**, каким оно было до любых наших manipulations.
   - **В случае с [MASK]:** Модель видит явный признак, что здесь что-то missing. Она использует контекст (`"quick brown ... jumps"`) чтобы предсказать, что здесь должно быть `"fox"`.
   - **В случае со случайным словом ("plane"):** Модель видит слово, которое абсолютно не подходит по контексту (`"quick brown plane jumps"` — это бессмыслица). Она должна понять, что это слово здесь некорректно, и исправить его, предсказав `"fox"`.
   - **В случае с неизмененным словом ("fox"):** Модель видит корректное слово в корректном месте. Её задача — понять, что это слово *было выбрано для маскирования*, и ей всё равно нужно предсказать `"fox"` для этой позиции. Если она просто "перескажет" входное слово `"fox"`, она не aprenderá ничего полезного. Заставляя её предсказывать то же самое слово, мы учим её понимать, что и это слово в данном контексте является уместным и правильным.

Для чего это нужно? Эта хитрость создает mismatch между этапом предобучения и этапом тонкой настройки (fine-tuning). На fine-tuning вы не увидите токен `[MASK]`, так как задача уже другая (классификация, NER и т.д.). Если бы модель during pre-training видела только `[MASK]`, она бы плохо адаптировалась к реальным данным, где все слова настоящие. Случайные и неизмененные замены

заставляют модель всегда быть "начеку" и анализировать каждое слово в контексте, а не слепо доверять входным данным.

---

2. Как согласуется 15% с правилом 80/10/10?

Здесь нет противоречия. Эти проценты работают на двух разных этапах.

1. **15%:** Это **общая доля слов во всем предложении**, которые будут отобраны для возможной замены. Из 100% всех слов в батче выбирается 15%.
    - *Пример: В предложении из 10 слов будет выбрано 1.5 слова. На практике это округляется, и в одних предложениях будет выбрано 1 слово, в других — 2.*
2. **80/10/10:** Это **распределение стратегий замены для тех самых 15% отобранных слов**.
    - **80% от 15%** (то есть **12%** от всех слов в предложении) будут заменены на `[MASK]`.
    - **10% от 15%** (то есть **1.5%** от всех слов) будут заменены на случайное слово.
    - **10% от 15%** (то есть **1.5%** от всех слов) останутся неизменными.

Итоговое распределение для всего набора данных выглядит так:

- **85% слов:** Не трогаются вообще. Модель не должна их предсказывать, она их использует как контекст.
- **15% слов:** Отбираются для маскирования. Далее с ними происходит:
    - `[MASK]`: 80% из 15% = **12%** всех слов.
    - Случайное слово: 10% из 15% = **1.5%** всех слов.
    - Оригинальное слово: 10% из 15% = **1.5%** всех слов.

Таким образом, правила прекрасно согласуются: сначала выбираем 15%, а затем внутри этих 15% применяем правило 80/10/10.

Не совсем понял - вот модель получает на вход предложение, какие-то слова заменены на MASK. Для тех слов, которые заменены на MASK мы решаем задачу классификации - укадываем нужное слово из словаря. Но для тех слов, которые просто заменены что мы делаем, тоже угадываем слово мз словаря? Тогда как на этапе обучения модель понимает что данное слово изменено на случаейное и для данного токена надо решить задачу классификации?

Отличный и очень важный вопрос! Вы правильно уловили ключевую сложность. Давайте разберемся.

Короткий ответ: Модель не должна сама догадываться, какое слово было заменено. Мы, на этапе подготовки данных, *явно указываем модели*, для каких именно позиций в предложении она должна предсказать оригинальное слово. Это делается с помощью **маскирования выходов (output masking)**.

Техническое объяснение

Процесс можно разбить на следующие шаги:

1. **Создание маски для предсказания (MLM Mask):**

- Для каждого предложения в батче мы случайным образом выбираем 15% токенов. Давайте создадим бинарную маску (например, тензор из 0 и 1) такой же длины, как и предложение. Единицы (`1`) будут стоять на тех позициях, которые мы отобрали для маскирования. Нули (`0`) — на всех остальных.
- *Пример:* Для предложения `[CLS] The quick brown fox jumps [SEP]` маска может выглядеть так: `[0, 0, 1, 0, 0, 0]`. Здесь мы выбрали третье слово (`brown`).

2. **Искажение входных данных:**
   - Теперь мы проходим по всем позициям, отмеченным `1` в нашей маске, и к каждому из этих слов применяем правило 80/10/10.
   - *Результат (допустим, применилось правило 10% - замена на случайное слово):* `[CLS] The quick **red** fox jumps [SEP]`
3. **Прямой проход (Forward Pass):**
   - Это искаженное предложение подается на вход BERT.
   - Модель обрабатывает его и на выходе выдает **вектор скрытого состояния (hidden state)** для *каждого* токена в последовательности. Каждый этот вектор можно преобразовать в распределение вероятностей по всему словарю (логиты).
4. **Вычисление потерь (Loss Calculation) — Самый важный шаг:**
   - Вот здесь мы и используем созданную на первом шаге **маску для предсказания**.
   - Мы вычисляем функцию потерь **только для тех позиций, которые отмечены единицей (1) в маске**. Для всех остальных позиций (отмеченных нулем) мы просто игнорируем предсказание модели.
   - **Технически:** Мы берем логиты (предсказания) модели только с позиций, указанных в маске, и вычисляем кросс-энтропийную потерю между этим предсказанием и истинным (оригинальным) словом.
   - *В нашем примере:* Мы возьмем предсказание модели для третьего токена (на месте которого сейчас стоит `red`). Модель, глядя на контекст `The quick ... fox jumps`, должна предсказать, что оригинальным словом было `brown`. Мы не смотрим на предсказания модели для `The`, `quick`, `fox` и т.д.

Аналогия

Представьте, что вы учитель, а модель — ученик.

1. Вы даете ученику текст с пропусками: "Москва — ??? России".
2. Но чтобы усложнить задачу, вы в некоторых местах не оставили пропуск, а написали заведомо неверное слово: "Столица ??? — Москва".
3. **Ключевой момент:** Вы заранее говорите ученику: "Я буду спрашивать у тебя только слова, которые подчеркнуты". Вы подчеркиваете слово `???` в первом предложении и слово `Столица` во втором.
4. Ученик видит весь текст, но знает, что отвечать нужно только про подчеркнутые слова. Его задача — для подчеркнутого слова `???` сказать "столица", а для подчеркнутого слова `Столица`

сказать "России" (потому что он должен восстановить оригинальный текст "Столица России — Москва").

Итог

Модель **не понимает**, что слово заменено на случайное. Ей это и не нужно. **Мы заставляем ее решать задачу предсказания оригинального слова для заранее заданного набора позиций**, независимо от того, что сейчас находится на этих позициях на входе (`[MASK]`, случайное слово или оригинал).

- **Вход модели (Input):** Искаженная последовательность (*после* применения 80/10/10).
- **Цель для предсказания (Target):** Оригинальная, неискаженная последовательность.
- **Маска потерь (Loss Mask):** Указывает, для каких позиций сравнивать предсказание с целью.

Таким образом, правило 80/10/10 — это просто способ **создания искусственно искаженного входа**, а маска для предсказания — это способ **сфокусировать модель на исправлении именно этих искажений**.

Мы же используем токеницацю BPE - получается мы часто будем маскировать не целые слова, а какую то часть слова и модель должна угадать эту часть слова исходя из окружения?

Отличный вопрос! Вы абсолютно правы, и это очень важный нюанс работы BERT и других моделей, использующих субсловную токенизацию (как BPE или WordPiece).

Да, именно так. Поскольку BERT использует токенизацию WordPiece (очень похожую на BPE), одно слово часто разбивается на несколько субсловных токенов. Например:

- Слово **"playing"** может быть разбито на `["play", "##ing"]`
- Слово **"unexpected"** может быть разбито на `["un", "##expect", "##ed"]`

Когда алгоритм выбирает 15% токенов для маскирования, он выбирает именно эти **субсловные токены**, а не целые слова. Это приводит к нескольким сценариям:

1. Маскируется только часть слова

Это самый частый случай. Допустим, в предложении `"She is playing chess."` мы выбрали для маскирования токен `"##ing"`.

- **Вход модели (после применения правила 80/10/10, допустим, сработало 80%):** `"She is play [MASK] chess."`
- **Задача модели:** Предсказать оригинальный токен `"##ing"` на позиции `[MASK]`, используя контекст. В данном случае контекст включает в себя начало того же самого слова — токен `"play"`.

Модель должна понять, что комбинация `"play" + ?` образует осмысленное слово в данном контексте (`"She is ... chess"`), и что наиболее вероятным завершением является `"##ing"`.

2. Маскируется всё слово

Если все субтокены, составляющие одно слово, были выбраны в рамках тех самых 15% (что маловероятно, но возможно), то слово будет замаскировано полностью.

- **Слово "unexpected" разбито на `["un", "##expect", "##ed"]`.**
- Алгоритм случайным образом выбрал для маскирования все три токена.
- **Вход модели (по правилу 80%):** `[MASK] [MASK] [MASK] situation arose.`
- **Задача модели:** Предсказать все три оригинальных токена поочередно. Это гораздо более сложная задача, так как модель лишена всей информации о слове. Ей приходится опираться исключительно на более широкий контекст (`"... situation arose"`), чтобы восстановить целое слово "unexpected".

3. Случайная замена части слова

Это самый интересный и сложный случай. Допустим, для токена `"##ing"` сработало правило 10% и его заменили на случайное слово, например, на `"##ly"`.

- **Вход модели:** `"She is play ##ly chess."`
- **Задача модели:** Посмотреть на эту позицию, увидеть там бессмысленную комбинацию `"play ##ly"` и, несмотря на это, предсказать, что изначально здесь был корректный токен `"##ing"`.

Именно в таких сценариях модель по-настоящему оттачивает свое понимание языка, потому что ее заставляют исправлять явные несоответствия в контексте.

Выводы и почему это важно

1. **Изучение морфологии:** Такая задача заставляет модель изучать морфологию языка. Она учится, что `"play" + "##ing"` означает действие в процессе, а `"play" + "##er"` означает человека.
2. **Устойчивость к опечаткам:** Этот механизм делает модель устойчивой к опечаткам и редким словам. Если модель на инференсе встречает незнакомое слово, разбитое на части, она все равно может понять его значение из контекста и составляющих его субтокенов.
3. **Более сложная задача:** Предсказание частей слов — это более сложная и тонкая задача, чем предсказание целых слов. Это заставляет модель формировать более глубокие и nuanced представления о языке.

Таким образом, маскирование на уровне субсловных токенов — это не недостаток, а мощный механизм, который позволяет модели эффективно работать с огромным словарным запасом и изучать не только синтаксис, но и морфологию языка.

Понятно, а еще вопрос - есть эмбединг самого токена, позиционный эмбединг и эмбединг сегмента. Можешь пожалуйста объяснить последний случай, т.е. у нас есть два предложения - первое и второе и для каждого предложения у нас просто отдельный вектор? Этот вектор меняетя от случая к случаю? Т.е. на одной эпохе это постоянный вектор для каждого сегмента или как?

Отличный вопрос! Вы затронули одну из ключевых архитектурных особенностей BERT. Давайте разберемся с эмбеддингом сегментов (Segment Embeddings или Token Type Embeddings).

Короткий ответ: Да, для каждого токена из первого предложения используется один обучаемый вектор (например, условно "0"), а для каждого токена из второго предложения — другой обучаемый вектор (условно "1"). Эти векторы **фиксированы** для всех пар предложений во время обучения и не меняются от батча к батчу или от эпохи к эпохе в рамках одного запуска обучения. Они являются частью параметров модели и обучаются вместе со всеми остальными весами.

Подробное объяснение

1. Для чего это нужно? BERT предобучается на двух задачах:

1. **Masked Language Model (MLM)** — та самая задача с угадыванием слов.
2. **Next Sentence Prediction (NSP)** — задача предсказания, является ли второе предложение следующим по смыслу за первым.

Эмбеддинг сегментов критически важен для второй задачи. Он позволяет модели легко различать, где заканчивается первое предложение и начинается второе, просто глядя на векторное представление токенов.

2. Как это технически устроено?

- **Эмбеддинг сегментов — это просто еще один lookup-таблица (слой эмбеддингов)**, очень похожий на таблицу для токенов и позиций.
- Размерность этой таблицы обычно `[2, hidden_size]`. То есть в ней всего **два вектора**:
  - Вектор с индексом `0` для всех токенов **первого предложения (Sentence A)**.
  - Вектор с индексом `1` для всех токенов **второго предложения (Sentence B)**.
- Эти два вектора инициализируются случайным образом и затем обучаются в процессе предобучения.

3. Как это выглядит на практике?

Рассмотрим пример пары предложений для задачи NSP:

- **Sentence A:** `[CLS] The cat sat on the mat [SEP]`
- **Sentence B:** `It was sleepy [SEP]`

После токенизации и добавления служебных токенов `[CLS]` и `[SEP]` модель создает массив `token_type_ids` (или `segment_ids`), который указывает, к какому предложению принадлежит токен:

Токен:	**[CLS]**	**The**	**cat**	**sat**	**on**	**the**	**mat**	**[SEP]**	**It**	**was**	**sleepy**	**[SEP]**
Segment ID:	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**1**

4. Процесс создания финального эмбеддинга:

Для каждого токена его финальный входной вектор формируется **поэлементным сложением** трех компонентов:

1. **Token Embedding:** Вектор самого слова (из словаря).
2. **Position Embedding:** Вектор позиции слова в последовательности (например, позиция 0 для `[CLS]`, 1 для `The`, 2 для `cat` и т.д.).

3. **Segment Embedding:** Вектор сегмента (берётся из таблицы по `segment_id`: 0 или 1).

```
Final_Embedding = Token_Embedding + Position_Embedding + Segment_Embedding
```

Этот объединенный вектор затем подается на вход первому слою трансформера.

5. Важные уточнения:

- **Фиксированность:** Вектора для сегментов 0 и 1 — это **обучаемые параметры модели**, такие же, как веса в полносвязных слоях. Они не меняются случайным образом от итерации к итерации. Они обновляются градиентным спуском, стремясь к некоторому оптимальному значению, которое помогает модели лучше решать свою задачу. После обучения они полностью фиксируются.
- **Универсальность:** Хотя изначально эмбеддинг сегментов был введен для задачи NSP, он оказался полезным и для других задач, где на вход подается два текста (например, вопрос-ответ, пара предложений для определения их семантической близости). В таких случаях модель по-прежнему использует сегмент `0` для первого текста и сегмент `1` для второго.
- **Один сегмент:** Для задач, где на вход подается всего одно предложение (например, классификация тональности текста), все токены получают сегмент `0`.

Итог

Эмбеддинг сегментов — это простой и эффективный способ дать модели структурную информацию о том, к какой части входных данных принадлежит токен. Это не случайные вектора, генерируемые на лету, а **статические, но обучаемые параметры**, которые становятся частью модели после предобучения.

Таким образом, на всех эпохах обучения для всех первых предложений во всех примерах будет использоваться один и тот же (но постепенно обучающийся) вектор сегмента `0`, а для всех вторых предложений — вектор сегмента `1`.

А как используя BERT можно решить задачу question answering? Интересует техническая составляющаяя

Отличный вопрос! Задача Question Answering (QA), а именно **извлечение ответа из текста** (Extractive QA), — это одна из тех задач, где BERT показал выдающиеся результаты. Техническая составляющая здесь очень элегантна.

Классический пример датасета для такой задачи — **SQuAD** (Stanford Question Answering Dataset). В нем есть:

- **Контекст** (passage) — большой текст.
- **Вопрос** по этому контексту.
- **Ответ** — это не отдельный факт, а **спан (span)** из самого контекста, то есть начало и конец слова/слов, которые являются ответом.

Например:

- **Контекст:** "Кошка сидела на коврике и спала..."
- **Вопрос:** "Где спала кошка?"
- **Ответ:** "на коврике" (который является непрерывным отрезком в контексте).

Вот как BERT решает такую задачу на этапе **fine-tuning**.

---

1. Подготовка входа (Энкодинг)

Модель на вход получает склеенную последовательность из вопроса и контекста. Между ними ставится служебный токен `[SEP]`, чтобы модель понимала границу.

Формат входных данных: `[CLS] [Вопрос] [SEP] [Контекст] [SEP]`

- `[CLS]` — специальный токен, который используется для классификации всей последовательности. В QA из него не всегда извлекают ответ.
- **Эмбеддинги:** Как мы обсуждали, к каждому токену добавляется его токенный, позиционный и сегментный эмбеддинг. Для QA:
  - Все токены **вопроса** и первый `[SEP]` получают **сегментный ID = 0**.
  - Все токены **контекста** и последний `[SEP]` получают **сегментный ID = 1**.

---

2. Архитектура головы для QA (Output Layer)

Вся мощь BERT используется для получения контекстуализированных векторных представлений каждого токена. Задача головы — **найти начало и конец ответа** среди токенов контекста.

Для этого поверх выходных векторов BERT (последнего слоя) добавляются **два простых линейных слоя** (иногда их называют "указателями"):

1. **Слой для начала ответа (Start Token Classifier)**
2. **Слой для конца ответа (End Token Classifier)**

Оба этих слоя — это независимые линейные преобразования, которые проецируют выходной вектор BERT для каждого токена (`T_i`) из размерности `hidden_size` в одно число (`logit`).

Технически:

- `start_logits = Linear_layer_start(T_i)`
- `end_logits = Linear_layer_end(T_i)`

Эти числа (логиты) затем пропускаются через функцию `softmax`, чтобы получить распределение вероятностей **для каждого токена в последовательности** быть началом или концом ответа.

Важное ограничение: Модель ищет ответ **только внутри контекста**. Токены вопроса и `[SEP]` не могут быть началом или концом ответа. На практике это учитывается при вычислении потерь или во время предсказания.

---

3. Процесс обучения (Fine-Tuning)

На этапе обучения у нас есть правильные ответы — индексы начального и конечного токена ответа в контексте.

Функция потерь (Loss Function) — это сумма двух функций **кросс-энтропии**:

1. **Loss_start:** Между предсказанным распределением вероятностей `softmax(start_logits)` и истинным индексом начала ответа (это one-hot vector, где 1 стоит на правильном токене).
2. **Loss_end:** Между предсказанным распределением `softmax(end_logits)` и истинным индексом конца ответа.

Общая потеря: `Total Loss = Loss_start + Loss_end`

Модель учится одновременно предсказывать оба положения так, чтобы они были согласованы и указывали на верный отрезок текста.

---

4. Процесс предсказания (Inference)

Когда модель обучена, мы проделываем следующие шаги:

1. Подаем на вход модель пару "вопрос + контекст".
2. Получаем два массива логитов: `start_logits` и `end_logits` для каждого токена.
3. Применяем `softmax` к этим логитам, чтобы получить вероятности.
4. Теперь нам нужно найти лучшую пару `(start, end)`. Мы не можем просто взять токены с максимальной вероятностью начала и конца независимо друг от друга, так как они могут быть несогласованными (например, конец раньше начала).

Стратегия поиска:

- Мы перебираем **все возможные пары** `(i, j)`, где:
  - `i` — индекс кандидата на начало (токен из контекста).
  - `j` — индекс кандидата на конец (токен из контекста, причем `j >= i`).
  - Ответ не может быть длиннее определенного предела (например, 30 токенов), чтобы сократить перебор.
- Для каждой пары `(i, j)` мы вычисляем **совместный скоринг**: `score = start_logits[i] + end_logits[j]`.
- Выбираем ту пару `(i, j)`, у которой этот скоринг **максимален**.

5. Взяв из исходного текста отрезок от `i`-го до `j`-го токена, мы получаем текстовый ответ.

Краткий итог технической реализации:

Компонент	Реализация в BERT для QA
Входной формат	`[CLS] Вопрос [SEP] Контекст [SEP]`
Сегментные эмбеддинги	`0` для вопроса, `1` для контекста
Выходная головка	Два линейных слоя поверх выходов BERT: один для начала ответа, один для конца.
Функция потерь	`Loss = CrossEntropy(start_logits, true_start) + CrossEntropy(end_logits, true_end)`

Компонент	Реализация в BERT для QA
Предсказание	Поиск пары `(start, end)` с максимальной суммой `start_logit + end_logit`.

Таким образом, BERT действует как мощный **энкодер**, который понимает взаимосвязь между вопросом и контекстом, а простая головка-указатель использует это понимание, чтобы буквально **"указать" на ответ** в тексте.

Давайте разберем этот текст по частям! Здесь описываются две важные модели - ELECTRA и DeBERTa, которые являются улучшенными версиями BERT.

ELECTRA

Основная идея: Вместо того чтобы просто маскировать (скрывать) случайные токены как в BERT, ELECTRA **заменяет** их на другие токены с помощью небольшой генеративной модели (MLM - Masked Language Model). Затем основная модель должна предсказать для каждого токена: "был ли этот токен заменен или он оригинальный?"

Почему это лучше:

- **Более эффективное обучение**: в BERT модель учится только на замаскированных токенах (обычно 15% от текста), а в ELECTRA - на всех 100% токенов
- **Меньше различий** между предобучением и дообучением на конкретных задачах

Архитектура:

- Есть два компонента: **генератор** (маленький transformer) и **дискриминатор** (основная модель)
- Генератор предлагает замены для токенов
- Дискриминатор определяет, какие токены были заменены
- Они обучаются совместно, но не состязательно (в отличие от GAN)

Результаты:

- Обучение занимает больше времени (две модели вместо одной)
- Но сходится быстрее к лучшему результату
- Особенно эффективен для маленьких моделей

DeBERTa

Это модель, которая улучшает и RoBERTa, и ELECTRA.

Ключевые улучшения:

1. Разделенное внимание (Disentangled Attention)

Проблема: В обычных трансформерах внимание зависит и от содержания токена, и от его позиции одновременно.

Решение: Разделить это на:

- Внимание к **содержанию** (H)

- Внимание к **относительной позиции** (P)

Это упрощает задачу для модели - она может отдельно понимать "что это за слово" и "где оно находится".

2. Улучшенный декодер масок

Проблема с BERT: абсолютные позиции добавляются к входным эмбеддингам сразу.

Решение DeBERTa:

- Использовать абсолютные позиции только на последнем слое
- Пропустить последний слой через себя 2 раза подряд
- Сначала используя позиционные эмбеддинги как запросы, потом - выход предыдущей итерации

Логика: Абсолютные позиции важнее всего на последнем слое, потому что там формируется понимание синтаксиса, который критичен для предсказания замаскированных слов.

3. Scale Invariant Fine-Tuning (SIFT)

Это техника регуляризации при дообучении:

- Добавляет небольшой "враждебный шум" к нормализованным эмбеддингам
- Штрафует модель за резкие изменения в распределении
- Делает модель более "гладкой" и стабильной
- Дает примерно +1% точности

Практический смысл для NLP-специалиста:

1. ELECTRA показывает, что можно обучаться эффективнее, используя все токены, а не только замаскированные
2. DeBERTa демонстрирует важность правильной работы с позиционной информацией
3. Разделение задач (содержание vs позиция) часто дает лучшие результаты
4. Правильная регуляризация при дообучении критически важна

Эти модели стали важными вехами в развитии архитектур до появления современных LLM вроде GPT.

Отлично! Давайте разберем XLNet по частям. Это действительно интересная и инновационная модель в области NLP.

Что такое XLNet?

XLNet - это модель, которая попыталась взять лучшее от двух подходов к предобучению:

- **Language Modeling (LM)** - классические языковые модели
- **Masked Language Modeling (MLM)** - подход BERT

Проблема, которую решает XLNet

Сравним два подхода:

Autoregressive LM (как GPT): ✅ Естественная авторегрессивная зависимость (предсказываем слово за словом) ✅ Одинаковое распределение данных при обучении и файн-тюнинге ❌ Нет двунаправленного контекста (видим только предыдущие слова)

MLM (как BERT): ✅ Естественный двунаправленный контекст ❌ Маскированные токены предсказываются независимо друг от друга ❌ Разное распределение данных (токены [MASK] есть при обучении, но нет при использовании)

Решение: Permutation Language Model

XLNet использует **пермутационную языковую модель**:

```
Пример: предложение "Я люблю изучать NLP"
Обычный порядок: [Я] → [люблю] → [изучать] → [NLP]
Пермутация 1: [люблю] → [NLP] → [Я] → [изучать]
Пермутация 2: [изучать] → [Я] → [NLP] → [люблю]
```

Ключевая идея: Модель обучается на всех возможных перестановках слов в предложении. Каждая позиция не знает заранее, какие другие позиции будут доступны, поэтому учится использовать информацию от всех позиций.

На практике вычисляют функцию потерь только для последних 14-17% позиций в каждой перестановке.

Архитектура: Two-Stream Attention

Это самая сложная часть. XLNet использует **два потока внимания**:

1. Content Stream (h)

- Похож на обычный декодер трансформера
- Видит содержимое токенов

2. Query Stream (g)

- Видит только **позиционную информацию**, но не содержимое текущей позиции
- Используется для предсказания содержимого

Зачем это нужно? В пермутационной модели нам нужно предсказать содержимое определенной позиции, не видя это содержимое, но зная позицию.

Дополнительные улучшения

XLNet также включает идеи из **Transformer-XL**:

- **Relative positional embeddings** - относительные позиционные кодировки
- **Segment recurrence** - механизм для работы с длинными последовательностями

Технические детали

- **Параметры:** 110M (base) / 340M (large) - как у BERT
- **Обучение:** 126GB текста, 2048 batch size, 500K шагов
- **Железо:** 512 TPU v3 чипов, 2.5 дня
- **Словарь:** 32K подслов (SentencePiece)

Почему это важно?

XLNet показал, что можно создать модель, которая:

1. **Двунаправленная** (как BERT) - видит весь контекст
2. **Авторегрессивная** (как GPT) - естественно предсказывает следующие токены
3. **Без артефактов обучения** - нет токенов [MASK], которых не будет при использовании

Это был значительный шаг вперед в понимании того, как можно обучать языковые модели!

Можешь пожалуйста рассказать про модель XLNet - я слышал что там использвуется что-то вроде перестановки слов и берут лучшее из подходом MLM и LLM. Расскажи пожалуйста подробно и понятно про эту модель

Конечно! XLNet — это действительно очень интересная и мощная модель, которая cleverly объединяет преимущества двух подходов: **авторегрессивного (AR) моделирования, как у GPT (Left-to-Right)**, и **двунаправленного контекста, как у BERT (Masked Language Model, MLM)**.

Давайте разберемся по порядку, в чем была проблема и как XLNet её решила.

1. Проблемы, которые XLNet решает

Чтобы понять XLNet, нужно увидеть недостатки его предшественников:

- **Проблема BERT (MLM):**
    - BERT использует маскирование (`[MASK]`). Во время обучения модель видит эти искусственные токены, которых **нет в реальных данных** на этапе тонкой настройки (fine-tuning). Это создает несоответствие между предобучением и применением модели.
    - BERT предполагает, что маскированные токены **независимы** друг от друга. Он предсказывает каждое `[MASK]` отдельно, не учитывая возможные зависимости между ними. Например, в предложении "Играть в [MASK] с [MASK]" он может предсказать "футбол" и "мяч", но с той же вероятностью и "футбол" и "сетку", что менее осмысленно.
- **Проблема классических авторегрессивных моделей (вроде GPT):**
    - Они предсказывают слова строго последовательно (слева направо или справа налево). Это означает, что при кодировании токена модель видит только предыдущие слова, но не будущие. Контекст является **односторонним**, что не идеально для задач понимания языка.

Цель XLNet: Получить двунаправленный контекст, как у BERT, но без использования искусственных токенов `[MASK]` и с сохранением свойства авторегрессивности, которое позволяет моделивать зависимости между словами.

---

2. Ключевая идея: Permutation Language Modeling (Моделирование языка через перестановки)

Вместо того чтобы маскировать слова, XLNet использует принципиально другой подход:

1. **Перебираем все возможные порядки (permutations):** Для данной последовательности слов `[x1, x2, x3, x4]` модель рассматривает все возможные способы (перестановки) их упорядочивания. Например:
    - Порядок 1: `x1 -> x2 -> x3 -> x4`
    - Порядок 2: `x3 -> x1 -> x2 -> x4`
    - Порядок 3: `x4 -> x3 -> x1 -> x2`
    - ... и так далее для всех 24 (4!) вариантов.
2. **Авторегрессия на основе перестановки:** Для каждой выбранной перестановки модель обучается предсказывать каждое слово в последовательности, используя **только те слова, которые находятся до него в *данной перестановке***. Важно: это **не** физическая перестановка слов в предложении, а **перестановка порядка зависимости**.
    - **Пример:** Возьмем перестановку `[x3, x1, x2, x4]`.
    - Задача модели:
        - Предсказать `x3`? У нее нет контекста (первое слово в порядке).
        - Предсказать `x1`? Она может использовать только `x3`.
        - Предсказать `x2`? Она может использовать `x3` и `x1`.
        - Предсказать `x4`? Она может использовать `x3`, `x1`, `x2`.
3. **Как это дает двунаправленность?** После обучения на огромном количестве всевозможных перестановок для каждого предложения, в среднем, для предсказания любого слова `x_t` модель в разных перестановках видела все возможные комбинации контекстных слов — и те, что стоят слева от `x_t` в исходном предложении, и те, что справа. Таким образом, модель учится использовать **полный, двунаправленный контекст**.

Простая аналогия: Представьте, что вы изучаете книгу, но читаете её не подряд, а в случайном порядке: сначала главу 5, потом 1, потом 3 и т.д. В итоге, чтобы понять главу 1, вы сможете использовать информацию из глав 5, 3 и любой другой, которую вы уже "прочитали" в вашем случайном порядке. В итоге вы знаете книгу целиком.

---

3. Техническая магия: Two-Stream Self-Attention

Это самая сложная и гениальная часть XLNet. Возникает проблема:

- **Задача:** В перестановке `[x3, x1, x2, x4]` мы хотим предсказать `x1`, используя только контекст `x3`.
- **Проблема:** В позиции, где мы пытаемся предсказать `x1`, модель должна:
    1. Содержать информацию о том, *какое слово* мы предсказываем (его позицию и семантику).
    2. Не иметь доступа к содержимому самого слова `x1` (иначе предсказывать будет нечего).
    3. Иметь доступ к контексту (`x3`).

Решить это стандартным механизмом внимания, как в BERT, невозможно. Для этого авторы придумали **Two-Stream Self-Attention**.

- **Query Stream (Поток запроса):**
    - **Цель:** Представлять *информацию о том, какое слово мы предсказываем* (пункт 1).
    - **Особенность:** У этого потока **нет доступа к содержимому токена** в этой позиции (пункт 2). Он знает только его позицию и контекст.
    - **Использование:** Этот поток используется для вычисления предсказания.
- **Content Stream (Поток содержимого):**
    - **Цель:** Представлять *информацию о самом токене и его контексте* (пункт 3).
    - **Особенность:** Имеет доступ и к содержимому токена, и к его контексту.
    - **Использование:** Этот поток предоставляет контекстную информацию для Query Stream.

На каждом слое эти два потока обновляются, взаимодействуя друг с другом через механизм внимания. В итоге, когда модель должна предсказать слово в позиции `i` для данной перестановки, она использует **Query Stream** этой позиции, который содержит идею "здесь нужно предсказать слово, и вокруг него есть вот такой контекст", но не содержит самого слова.

---

4. Преимущества XLNet

1. **Нет разрыва между предобучением и fine-tuning:** Поскольку модель не использует `[MASK]`, она не сталкивается с незнакомыми токенами на этапе применения.
2. **Моделирование зависимостей между токенами:** Будучи авторегрессивной моделью по своей сути, XLNet естественным образом учится вероятностным зависимостям между словами. Она понимает, что если первое маскированное слово — "Нью-Йорк", то второе, скорее всего, "сити", а не "река".
3. **Мощное двунаправленное представление:** Как и BERT, она использует контекст со всех сторон для кодирования каждого слова, и даже более эффективно за счет перестановок.
4. **Наследство от Transformer-XL:** Модель также Incorporates идеи из Transformer-XL (что и отражено в названии 'XL'), а именно:
    - **Сегментированное повторное использование:** Модель может запоминать информацию из предыдущих сегментов текста, что позволяет ей работать с гораздо более длинными последовательностями, чем позволяет длина контекста одного сегмента.
    - **Относительные позиционные кодирования:** Вместо абсолютных позиций модель использует кодирования, которые указывают на относительное расстояние между токенами. Это хорошо обобщается на тексты длиннее тех, что были при обучении.

Итог

XLNet — это сложная, но элегантная модель, которая обходит ограничения BERT, используя **permutation language modeling**. Она предсказывает слова в случайном порядке, что позволяет ей использовать весь контекст, оставаясь при этом авторегрессивной моделью. Для технической реализации этого потребовалось ввести **two-stream self-attention** механизм.

На практике XLNet показала state-of-the-art результаты на многих задачах (например, на датасете SQuAD для вопросно-ответных систем), подтвердив эффективность своего подхода. Однако эта

модель значительно сложнее и требовательнее к вычислительным ресурсам для обучения, чем BERT.