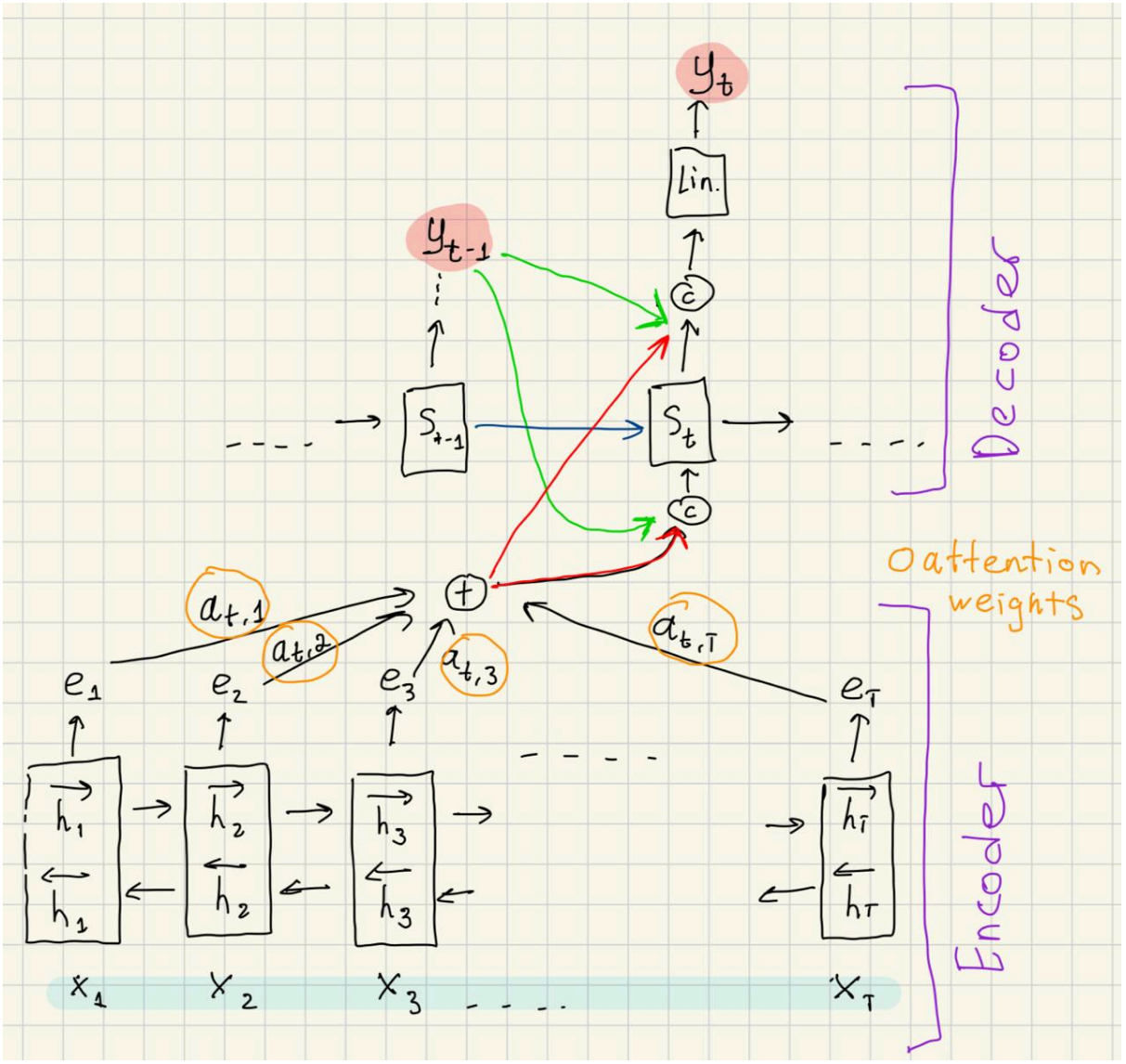
1. Рекуррентные модели

Я решила начать с чего-то попроще и вначале использовала рекуррентную архитектуру (вдохновлялась https://pytorch.org/tutorials/beginner/torchtext_translation_tutorial.html) - она состояла из энкодера в виде рекуррентной сети (bidirectional gru) с предшествующим dropout примененный к эмбеддингам и последующим линейным слоем для каждого из скрытых состояний - получаем закодированное входное предложение, декодера который реализован как рекуррентная сеть (gru) с линейным слоем- для предсказания следующего токена. Также в этой архитектуре используется механизм внимания между закодированной энкодером входной последовательностью и скрытым состоянием в декодере. Таким образом мы на каждом шаге генерации выходного предложения формируем так называемый контекстный вектор который является взвешенной суммой векторов из энкодера, где веса как раз получаются из применения attention к скрытому текущему состоянию в декодере и векторов полученных из энкодера. Этот контекстный вектор мы высчитываем на каждом шаге генерации и конкатенируем его с эмбеддингом входного токена для отправки в gru. Для предсказания следующего токена к выходу реккурентной сети конкатенируется вот этот контекстный вектор а также эмбеддинг текущего токена и подается в линейный слой. Также надо упомянуть что я использовала для этой архитектуры `torch.nn.utils.clip_grad_norm_` для стабильного обучения чтобы градиенты не становились гигантскими. Веса модели я инициализировала семплируя из нормального распределения с 0 матожиданием и стандартным отклонением в 0.01.

Попыталась изобразить все вышеописанное



2. Датасет

Валидационный датасет я обрезала до 500 предложений Эксперименты проводила с обучением на 90К предложений из обучающей выборки (и в таком варианте использовала размер батча = 128) а потом хорошие комбинации учила заново на полной тренировочной выборке (в таком случае использовала размер батча 256). Что касается словаря, то я везде использовала токенизацию по словам (деля предложение по пробелам). Размер словаря регулировала параметром минимальной частоты встречаемости токена в

обучающей выборке. В исходном варианте минимальная частота была 25 (словарь получался примерно 4000 на урезанной обучающей выборке и 6000 на полной). Генерацию английского текста я ограничивала максимальным количеством слов равным количеству слов в немецком тексте + 10

3. Немного экспериментов с рекуррентками (*изменения пишу относительно предыдущей версии)

1. `gru_bid_enc_attention_gru_with_skip1:`

размер эмбеддинга для немецкого = 64
размер эмбеддинга для английского = 64
размерность скрытых векторов в энкодере = 128
размерность скрытых векторов в декодере = 128
размерность в attention = 32
encoder dropout = 0.5
decoder dropout = 0.5
количество эпох = 15
clip = 1
teacher_forcing_ratio = 0.5

Оптимизатор - Adam
лосс - CrossEntropyLoss с игнорированием токенов паддинга

В этой модели я использовала teacher_forcing с 50% вероятностью - то есть с 50% вероятностью во время обучения брала или сгенерированный моделью выходной токен или же токен из таргета.

Такая модель выбила на валидации 12.2 bleu мы обсуждали на лекции что учить rnn подавая на вход ее же

ответы с предыдущего шага (пусть и с какой-то вероятностью) тяжело, поэтому в следующей версии я отказалась от этой идеи и подавала токены предложения из обучающей выборки.

2. `gru_bid_enc_attention_gru_with_skip2:`
   teacher_forcing_ratio = 1
   количество эпох = 20

Качество на валидации: 17.75 BLEU, кажется что модель достаточно маленькая так что попробуем ее увеличить немного

3. `gru_bid_enc_attention_gru_with_skip3:`

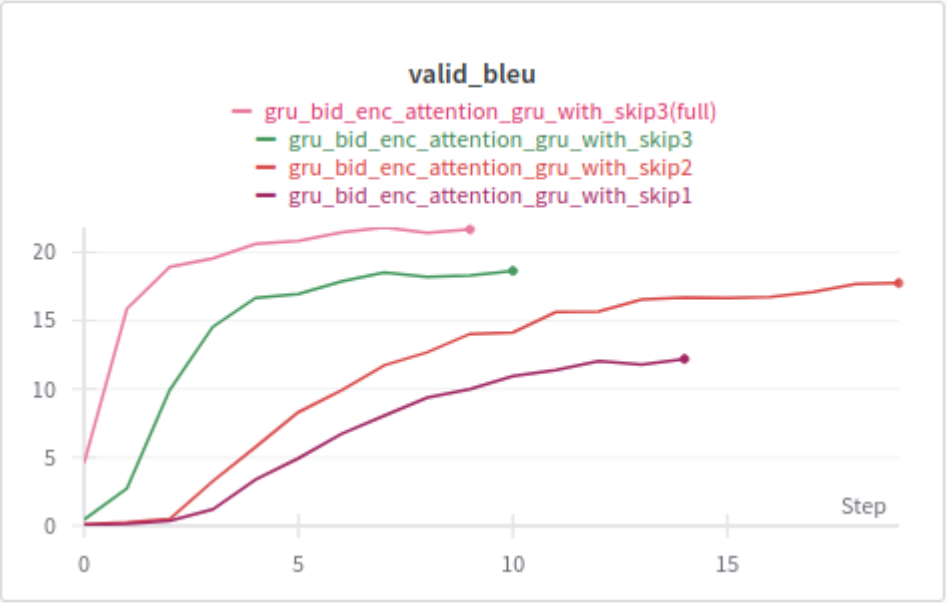
размер эмбеддинга для немецкого = 128
размер эмбеддинга для английского = 128
размерность скрытых векторов в энкодере = 256
размерность скрытых векторов в декодере = 256
размерность в attention = 64
количество эпох = 11

Качество на валидации: 18.64 что круто для 11 эпох так что можно попробовать обучить такую модель на всем обучающем датасете

4. `gru_bid_enc_attention_gru_with_skip3(full):`

количество эпох = 10

Качество на валидации: 21.67.



После этих небольших экспериментов я решила что пора переходить к чему-то серьезному и начала эксперименты с трансформером

4. Эксперименты с трансформером

Я использовала самую базовую архитектуру трансформера (https://pytorch.org/tutorials/beginner/translation_transformer.html), ничего глубоко не меняя.

1. `transformer 1`

размерность эмбеддингов = 128
количество голов = 2
размерность внутри feed-forward = 128
количество слоев энкодера = 1
количество слоев декодера = 1
количество эпох = 38

лосс - CrossEntropyLoss с игнорированием токенов паддинга
оптимизатор - Адам с lr = 1e-4, betas=(0.9, 0.98), eps=1e-9

Получилось достаточно грустно: на валидации 16.86 всего, но в целом такой результат объясняется тем что модель маленькая - увеличим !

2. `transformer 2`

размерность эмбеддингов = 256
количество голов = 4
размерность внутри feed-forward = 512
количество слоев энкодера = 3
количество слоев декодера = 3
количество эпох = 30

Качество естественно улучшилось: 20.23

Дальше я подумала о том - а почему модель учится предсказывать <unk> токены, почему мы в лоссе как бы утверждаем что предсказывать <unk> это правильно если это заведомо плохой вариант ведь такого слова не существует, не лучше ли было бы исключить при подсчете лосса позиции где следующий токен - <unk> в обучающей выборке, и тогда если на тесте реально нужно будет предсказать слово которого нет в словаре - то мы просто заменим его чем-то (возможно каким-то синонимом попроще и это поможет в дальнейшей генерации лучше понять контекст чем просто <unk> который под собой непонятно что подразумевает и не дает никакой информации особо).

3. `transformer 3`

mask_unk = True
количество эпох = 46

Качество реально улучшилось : на валидации 24.12
однако если смотреть на генерацию то лично мне редко попадались случаи когда место <unk> заменялось чем-то значимым, часто это были какие-то повторы, но тем не менее это бустит качество, видимо действительно модели так легче обучаться.

Раз уж эта модель такая классная – обучим ее на полной обучающей выборке

4. `transformer 3(full)`

количество эпох = 32
Качество: 28.41

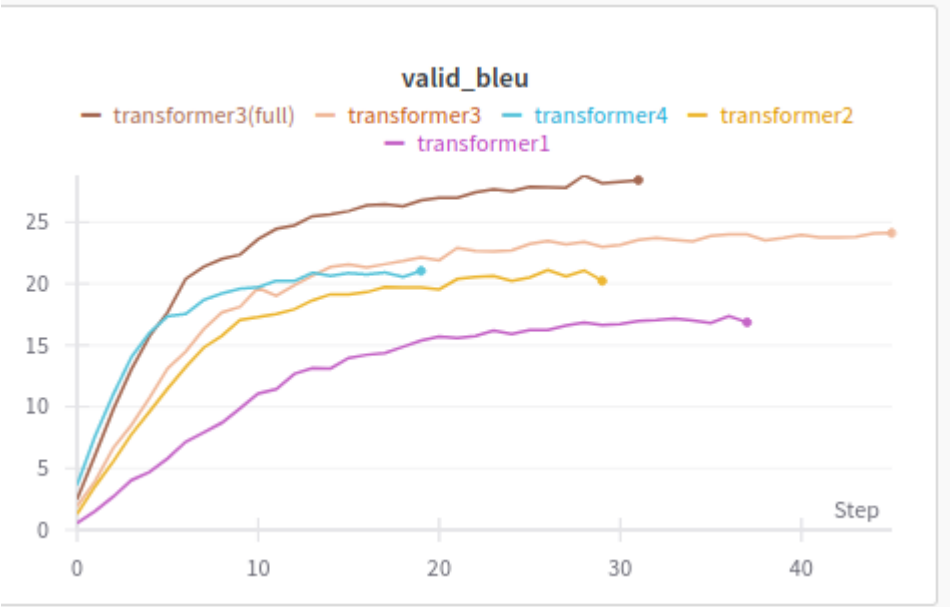
Такой подход с тем чтобы не обращать внимание на <unk> показал прекрасный результат но меня все еще смущали предложения которые такая модель генерирует - а именно вот эти повторы, поэтому я рассматривала 2 варианта обучения (mask_unk = True/False) параллельно. Особых идей как улучшать второй вариант кроме как наращивать модель я не видела поэтому я решила нарастить размер эмбеддинга и количество голов

5. `transformer 4`

размерность эмбеддингов = 512
количество голов = 8

mask_unk = False
количество эпох = 20 (далее вышел на плато)

Качество на валидации: 21.04 что хуже чем меньшая модель обученная с mask_unk = True и лучше чем модель меньше с mask_unk = False



Далее я решила еще увеличить модель с mask_unk = True, но при этом по слоям - увеличила только количество слоев в декодере а не в энкодере, так как в целом интуитивно кажется что для декодера может понадобиться больше этапов чем для сжатия. Также я подумала что возможно размер словаря слишком маленьких и можно поднять минимальную частоту до 15 слов (получается размер словаря где-то 6000 при маленькой обучающей выборке и 10000 при полной обучающей выборке) в дальнейших моделях буду использовать такую частоту.

6. `transformer 6`

размерность эмбеддингов = 384

количество голов = 6
количество слоев энкодера = 3
количество слоев декодера = 5
количество эпох = 31
mask_unk = True
min_freq = 15

Качество на валидации: 26.31 - стало значительно лучше

Следовательно можно попробовать эту модель обучить на полной выборке

7. `transformer 6(full)`

количество эпох = 40
Качество на валидации 30.38

Получилось очень круто но нужно было дать шанс трансформерам с mask_unk = False, поэтому я решила поулучшать немного их попробовав ту же конфигурацию как и в трансформере 6. Еще дополнительно я решила попробовать использовать label smoothing, в надежде что это поспособствует лучшим результатам, и модель улучшит обобщающую способность за счет вот такого вида регуляризации

8. `transformer 7`

количество эпох = 37
mask_unk = False
label_smoothing = 0.1

Качество на валидации: 24.03

Но я подумала что этого мне тоже мало и поэтому решила еще увеличить размер по сравнению с transformer 7. Ну и для того чтобы это хорошо училось я добавила scheduler = ReduceLROnPlateau с параметрами patience = 4, factor = 0.3, min_lr = 1e-7
Если честно, в этой комбинации я пробовала также использовать weight_decay, также увеличивать в Adam lr до 1e-3 однако после первых пары эпох я останавливала эти эксперименты потому что качество было ужасным, лосс прыгал, и BLEU было супер низким.

9. `transformer 8`

количество слоев энкодера = 6
количество слоев декодера = 6
количество эпох = 33

scheduler = ReduceLROnPlateau с параметрами patience = 4, factor = 0.3, min_lr = 1e-7

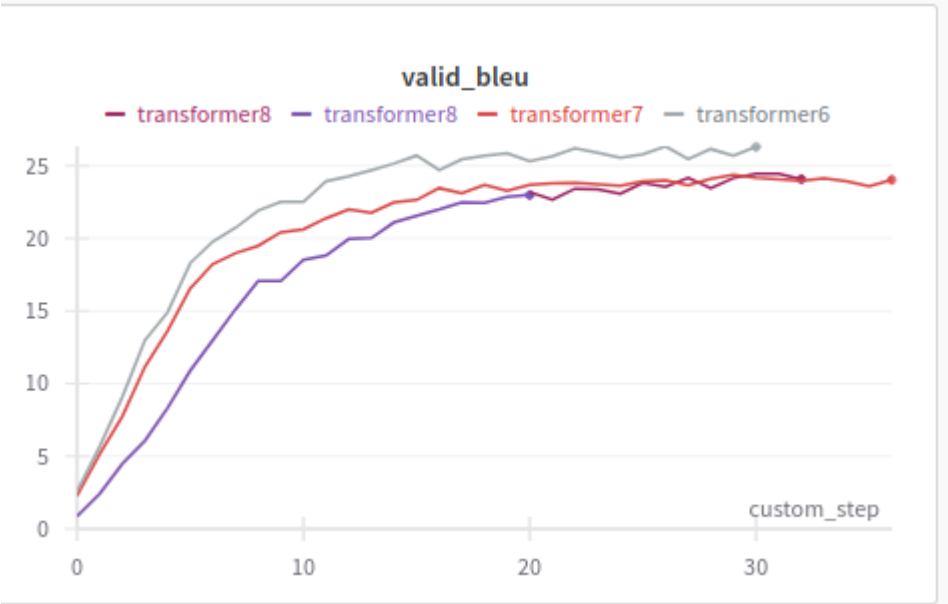
Качество 24.08, в целом особо ничего не поменялось относительно transformer7, но я поверила в то что если больше слоев то модель может больше в себя впитать особенно когда я ей скормлю всю обучающую выборку поэтому из этих 2 моделей я выбрала transformer 8 (может быть и зря)

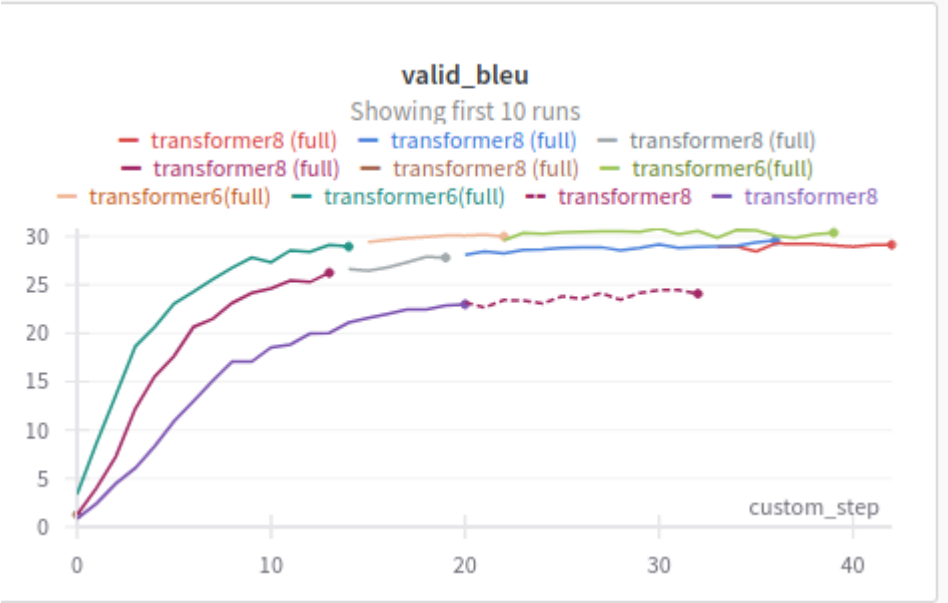
Соответственно обучила его на полной обучающей выборке

10. `transformer 8 (full)`

количество эпох = 43
Качество: 29.13 что хуже чем у модели transformer 6 (full) однако надо помнить что здесь мы по факту никак не

обрабатывает <unk> и учимся их генерировать там где действительно стоит слово которого нет в словаре, поработав с этими <unk> на пост-процессинге мы можем улучшить качество в то время как в transformer 6(full) мы не учили модель генерировать <unk> поэтому она вообще их не предсказывает поэтому тут меньше свободы для улучшения результата пост-процессингом.



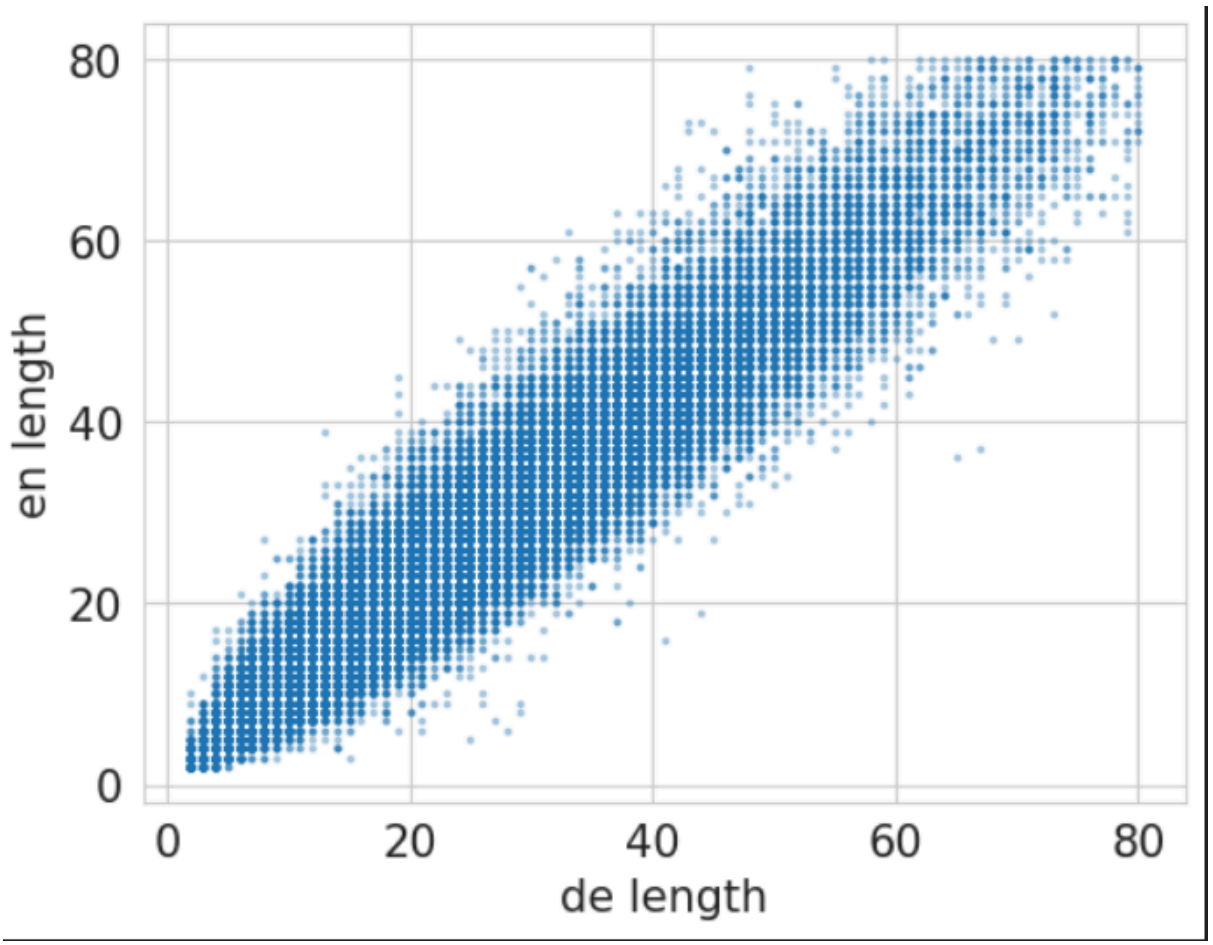


5. Эксперименты с пост-процессингом

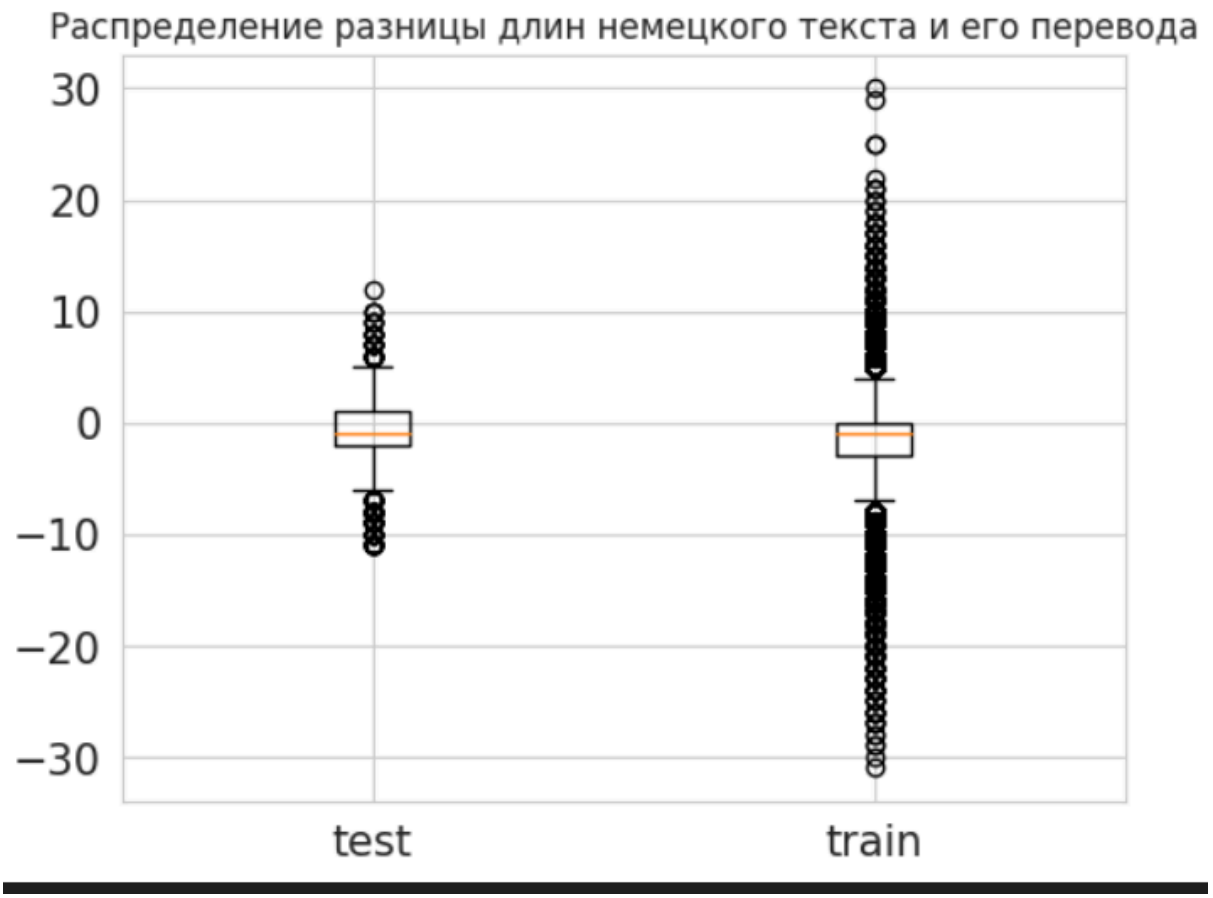
Я попробовала несколько вариантов пост - процессинга:

1. without_unk = True/False этот параметр пост-процессинга отвечал за то предсказывать ли unk или заменять их на следующий по вероятности токен.
2. left_unk_for_gen = True/False - в том случае когда without_unk = True, у нас есть 2 варианта как поступить: использовать все-таки unk для генерации следующих токенов или использовать токены которые будут в итоге в предсказанном тексте
3. beam_search = True/False с параметром k - использовать или нет beam_search, ограничивающее количество кандидатов на каждом шаге задано параметром k

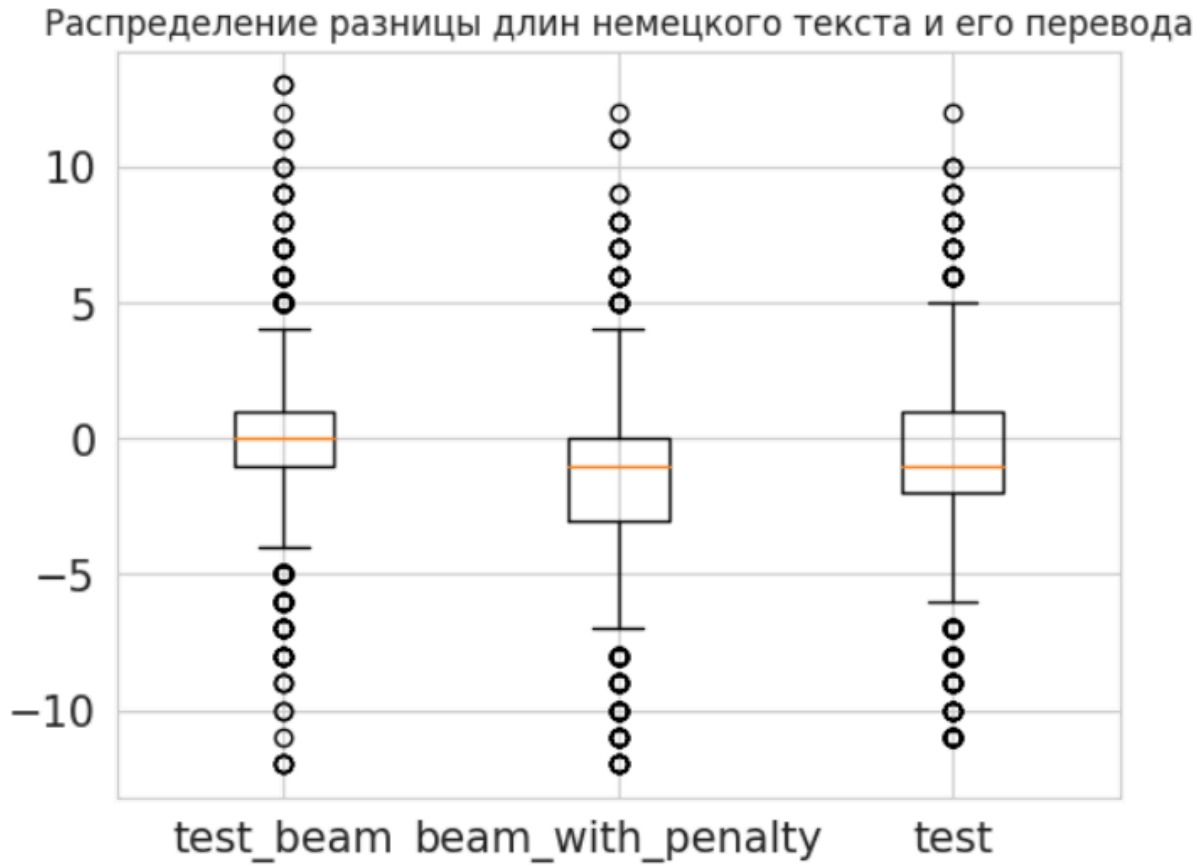
Также я анализировала зависимость длины английского перевода от немецкого (думала может как-то изменить этот порог в +10 слов), однако посмотрев на графики поняла что мой порог подходит идеально так как разница в длинах оригинала и перевода не зависит от длины оригинала и в целом в большинстве своем разница зажата в -10 и 10



Я посмотрела также как в тренировочном датасете распределена разница длин оригинала и перевода и как в тестовом датасете (то есть тестовые оригиналы и мои переводы без всякого пост-процессинга)



Получаем что наши переводы слегка короче но в целом в медиану мы попадаем однако если мы посмотрим на результаты перевода с beam_search мы увидим что мы еще больше отдаляемся от распределения на трейне (мы еще сильнее занижаем длину перевода) что плохо, поэтому чтобы приблизиться к распределению как на обучающей выборке, мы введем в beam_search так называемый штраф за короткие предложения: наш логарифм правдоподобия на текущем шаге будет делиться на величину
(5 + cur_len)^a / 6^a, где a > 0 , так как логарифм правдоподобия это отрицательное число а мы хотим его максимизировать то получается что если мы будем делить на маленькое положительное число (что соответствует маленькой длине) то правдоподобие будет меньше чем если бы мы делили на бОльшее положительное число.



получилось действительно лучше приблизить распределение на тренировочной выборке

посмотрим на результаты моих экспериментов с пост-процессингом

в таблице представлена метрика на тестовой выборке

	transformer 4	transformer 8(full)	transformer 6(full)
without_unk = False beam_search = False	22.02	27.63	26.6
without_unk = True left_unk_for_gen = False beam_search = False	21.52	27.33	—
without_unk = True left_unk_for_gen = True	22.29	27.84	—

beam_search = False			
without_unk = False beam_search = True k = 3	22.64	—	—
without_unk = False beam_search = True k = 5	22.67	28.02	27.09
without_unk = True left_unk_for_gen = False beam_search = True k = 3	22.87	—	—
without_unk = True left_unk_for_gen = False beam_search = True k = 5	22.71	27.9	—
without_unk = False beam_search = True k = 5 a = 0.6 (c len_penalty)	—	—	25.58
without_unk = False beam_search = True k = 5 a = 0.3 (c len_penalty)	—	—	26.5

Как мы видим len_penalty в моих экспериментах показал себя не очень хорошо, обычный beam_search лучше, возможно надо было перебрать получше a но если честно так как у модели с mask_unk = True качество на тесте по сравнению с валидацией упало на 4 BLEU, я не видела перспективы в этой модели, поэтому решила продолжить работу с transformer8(full). Лучшее качество у нее удалось достичь при beam_search с k = 5, а всякие фокусы с заменами unk показали результат хуже.

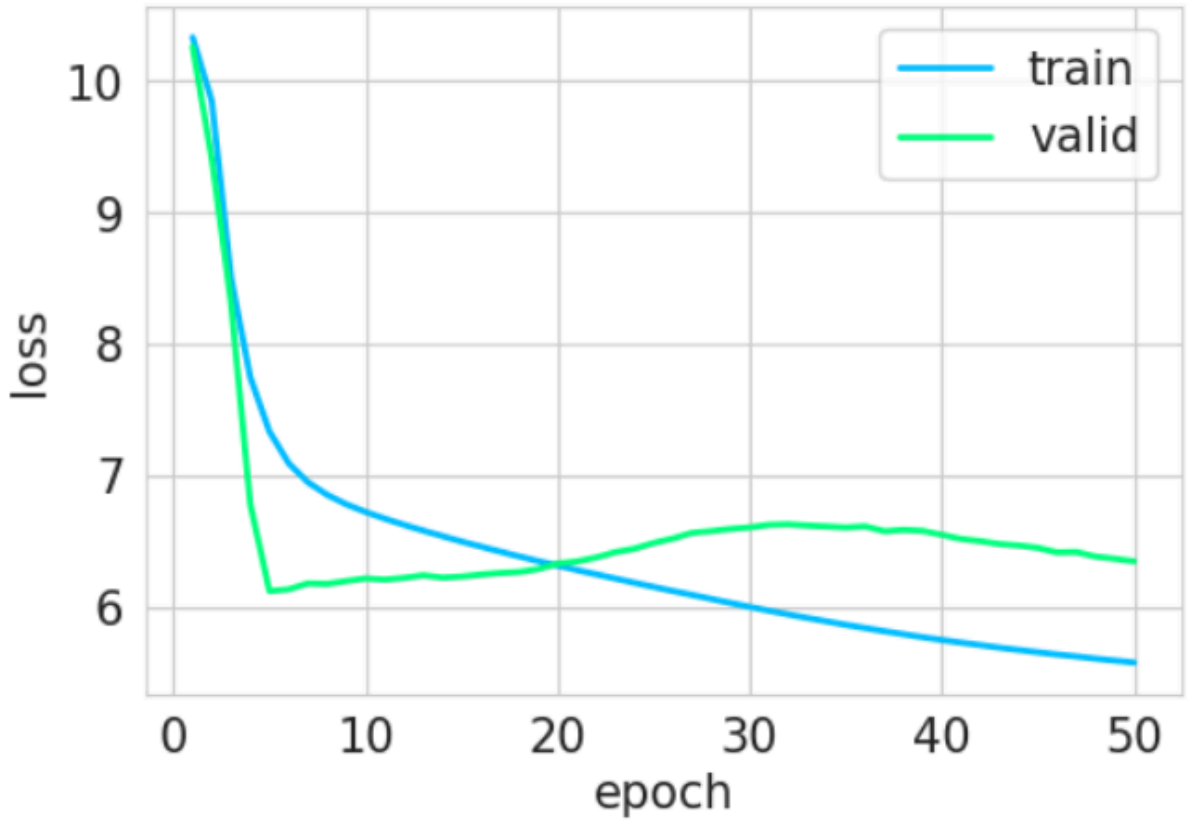
Но unk токен это на самом деле проблема - их достаточно много и понятно что их точно не может быть в истинном переводе оригинала - значит с ними надо что-то делать. Все мои прошлые подходы они не могли заменить unk на что-то что находится за пределами словаря и только лишь могли как-то повлиять на дальнейшую генерацию или в случае если unk предсказывался ложно - это поправить. Следовательно надо было как-то покрыть слова которые находятся вне словаря. Все что у нас есть это исходный текст на немецком, соответственно было бы круто понимать к какому немецкому слову относится этот unk, так как скорее всего unk это одно какое-то слово встречающееся не так часто а не часть речевой конструкции, поэтому соотношение 1 к 1 unk и немецкого слова очень бы помогла. Сама архитектура трансформера наводит на мысль что можно смотреть на attention матрицы между выходами энкодера и декодера, ведь кажется что для предсказания очередного слова больше всего вес будет у немецкого слова ему соответствующего. Тогда мы можем просто в соответствие очередному предсказанному unk ставить немецкое слово имеющее наибольший вес в момент предсказания этого unk. Я брала attention матрицы с последнего слоя декодера так как они ближе всего к предсказанию следующего токена и мне показалось логичным брать именно их. Также так как я использую multihead attention, я просто делала усреднение по всем головам.

6. Немного бесполезных экспериментов

получается допустим у нас будет соответствие между unk и немецким словом, но нам нужно выдать вместо unk

английское слово, поэтому нам каким-то образом надо получить перевод соответствующего немецкого слова.
У нас есть достаточно неплохой корпус английских и немецких текстов и я подумала что можно обучить векторные представления для всех вообще английских слов, потом обучить векторные представления для всех немецких слов а далее перевести их в одно пространство (читала в статьях что так делали просто обучая матрицы перехода). Ну и мне хотелось завести что-то простое так как я не была уверена что это сработает и не хотела чтобы это заняло много времени поэтому я сначала решила просто обучить word2vec представления для английских слов (использовала CBOW модель), и уже из этого ничего хорошего не получилось, конечно возможно надо было поучить это все подольше покрутить гиперпараметры и так далее но я поняла что на самом деле нас интересуют редко встречающиеся слова (но я рассматривала только слова которые встречались больше 1 раза) и скорее всего именно такая модель лишь по нескольким примерам не сможет выучить хорошие вектора для таких слов.

просто оставлю это здесь:



Дальше я пошла читать статьи что можно сделать чтобы не супер долго училось (времени до конца соревнования оставалось мало) и чтобы эту модель было просто завести (не имея прям супер большого корпуса текстов).
В итоге я остановилась на Bi-Sent2vec модели (https://arxiv.org/pdf/1912.12481.pdf)
ее идея в том чтобы учить векторы для английских и немецких слов используя параллельный корпус переводов который у нас имеется, мы вычисляем вектор предложения через векторы слов в нем и потом минимизируем такой лосс (то есть мы хотим как бы предсказывать слово по предложению без этого слова, также хотим предсказывать слово по полному переводу исходного предложения с этим словом, ну и чтобы это все не скатилось к тривиальному решению добавляем негативы)

$$\min_{\boldsymbol{U},\boldsymbol{V}} \sum_{\substack{S\in\boldsymbol{C}\\ l,l'\in\{l_1,l_2\}\\ l\neq l'}} \sum_{w_t\in S_l} \Bigg(\underbrace{\ell\big(\boldsymbol{u}_{w_t}^\top \boldsymbol{v}_{S_l\setminus\{w_t\}}\big) + \sum_{w'\in N_{w_t}} \ell\big(-\boldsymbol{u}_{w_t'} \boldsymbol{v}_{S_l\setminus\{w_t\}}\big)}_{\text{monolingual loss}} + \underbrace{\ell\big(\boldsymbol{u}_{w_t}^\top \boldsymbol{v}_{S_{l'}}\big) + \sum_{w'\in N_{w_t}} \ell\big(-\boldsymbol{u}_{w_t'} \boldsymbol{v}_{S_{l'}}\big)}_{\text{cross-lingual loss}} \Bigg)$$

В процессе обучения я выводила получающиеся переводы слов (максимальный dot-product между эмбеддингами слов), ну во-первых у меня ничего не училось то есть почему-то лосс падал супер медленно переводы вообще были какие-то рандомные, в общем я достаточно времени потратила чтобы в этом разобраться и это написать а по итогу ничего не вышло(((

7. Немного полезных экспериментов (качество пишу на тесте)

После разочарований с неудавшимися моделями я подумала что уже ничего нового написать я не успею, нужно работать с тем что есть – а именно с моим обученным трансформером 8 (full). Если подумать, то мы можем использовать его attention матрицы не только для того чтобы искать соответствующее unk немецкое слово, но и чтобы составить словарь немецкое слово - английское слово, для этого я прогоняю всю обучающую выборку через модель, и в получившихся attention матрицах для каждого английского слова выбираю 2 немецких слова (максимальных по значениям в матрице) и для каждого немецкого слова выбираю 2 английских слова таким же

образом, и в словарь (где ключи это пара (de_word,en_word)) прибавляю для каждой из 4 пар соответствующие значения в attention матрице. Таким образом после у меня получается набор пар немецкое слово - английское слово и их накопленная сумма значений в attention матрицах. Далее итоговый словарь составляется так: для каждого немецкого слова ищу пару из английского слова у которого в текущем словаре самая большая накопленная сумма.

Теперь уже с этим словарем можно работать по старой схеме, пропускаем текст на немецком через нашу модель, получаем предсказания и attention матрицу и с помощью нее находим для каждого unk соответствующее немецкое слово, далее с помощью нашего словаря переводим его на английский.

Качество у такого подхода получилось: 29.58

Но иногда случалось такое что слова в словаре не оказалось (например мы смапили unk в какое-то немецкое слово которого не было у нас в обучении) и в таком случае ну не оставлять же просто unk, можно попробовать прогнать еще раз предложение на немецком через модель но уже со старым пост-процессингом с without_unk = True, left_unk_for_gen = True, beam_search = False и тогда unk которые мы не смогли заменить на английские слова заменять соответствующем словом из построенного с помощью старого пост-процессинга перевода.

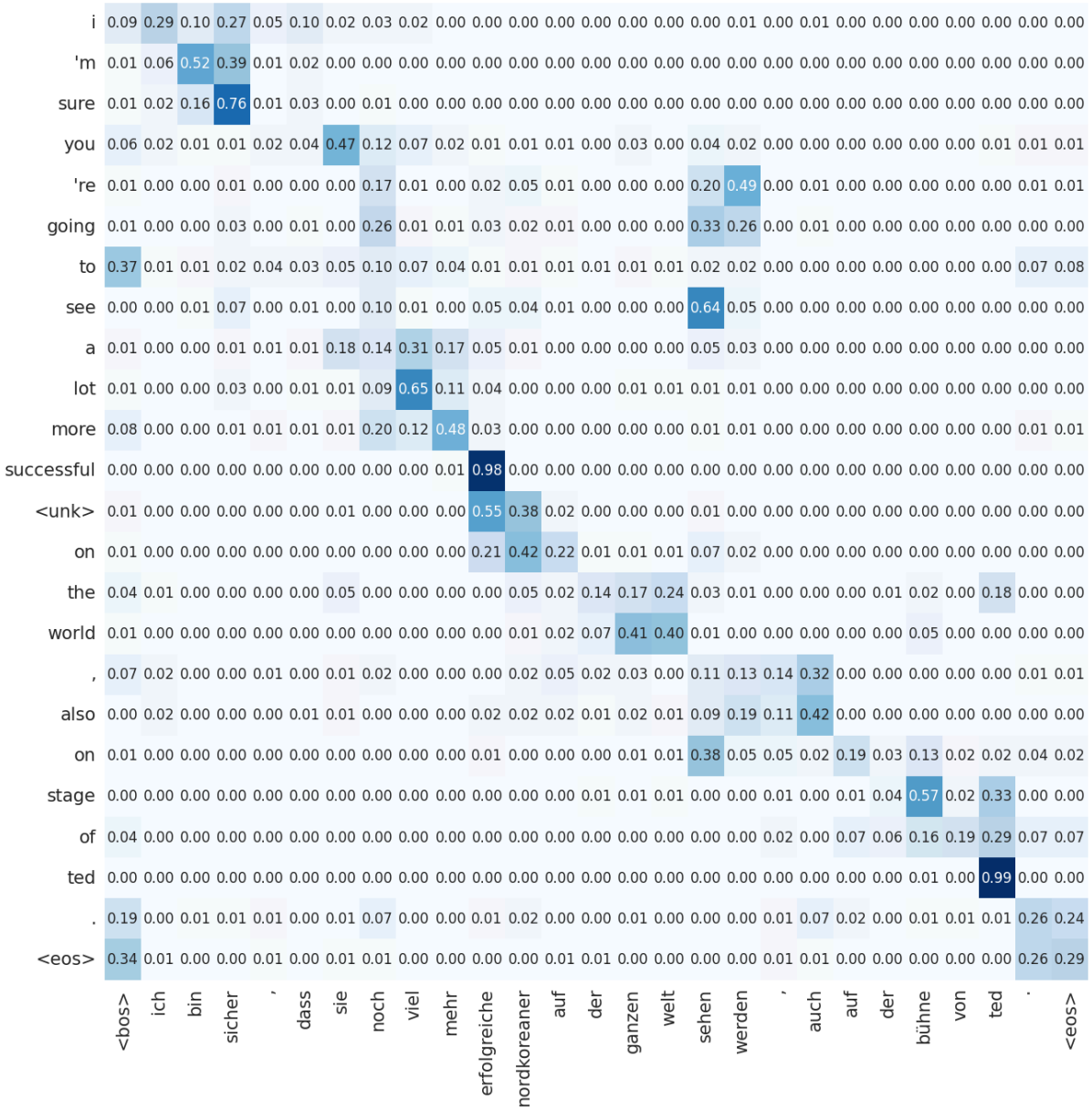
Качество чуть-чуть совсем улучшилось : 29.59

Также можно не просто прогонять немецкий текст через модельку чтобы получить первый вариант перевода но и например навесить beam_search, не трогая никак в нем unk.
при k = 5 и без второго прогона предложения для покрытия непокрытых unk получилось : 30.06

Ну и теперь можно покрыть непокрытые unk в последнем варианте: для этого сгенерированный перевод с помощью beam_search надо прогонять постепенно через модель предсказывая следующий токен именно по имеющейся сгенерированной последовательности (а не генерировать ее заново) и заменять unk на максимально вероятный токен отличный от unk, таким образом мы просто заменим имеющиеся unk в переводе никак не затронув остальные токены, назовем получившийся перевод with_removed_unk
ну и тут опять тогда пользуемся той же схемой - меняем unk через словарь если не получается то меняем на соответствующий токен в with_removed_unk.

Качество: 30.1

Хотелось больше поэтому я подумала что надо как-то поменять маппинг unk в немецкие слова, потому что я визуализировала некоторые attention матрицы и мне попадались вот такие ситуации



Тут логически понятно что unk должен относиться к nordkoreaner но тем не менее это лишь второе по максимальности слово, но если мы посмотрим на немецкое слово в которое мы бы мапили unk, то тут сразу понятно что оно относится к слову successful, потому что у successful 0.98 для этого слова, в то время как у unk всего лишь 0.55. Для того чтобы это учитывать я решила брать не просто максимум по значениям в attention матрице но чуть изменить скор для каждого немецкого слова по которому я и буду максимизировать. Он считается так:

score_ij (скор для английского слова на позиции i относительно немецкого слова на позиции j) = a_ij - (max a_:j - a_ij) то есть мы вводим штраф который равен разнице между максимальным attention весом для данного немецкого слова и attention весом между английским словом i и данным немецким словом.

И именно этот пост-процессинг оказался лучшим: мы используем все тот же beam search с k=5 строим для непокрытых unk with_removed_unk перевод, но теперь просто меняем алгоритм маппинга предсказанного unk в немецкое слово

Это выбило: 30.26 и стало моей лучшей моделью)