Отчёт EFDL ДЗ1.

Александр Куцаков

11 февраля 2024 г.

1 Исправление багов

Для удобства я буду перечислять исправления по файлам. В процессе я столкнулся с тем, что диффузионка генерирует изображения с чёрными или белыми пятнами, после чего 90% времени решения дз я пытался исправить эту проблему. Многие модификации были связаны с ней.

Также я не считаю нужным подробно комментировать рефакторинг кода связанный с логированием (например, изменение параметров функций или возвращаемых значений). По логируемым переменным я пройдусь в следующей секции без привязки к коду.

1.1 tests/test_model.py

Функция *test_unet.py* вопросов не вызвала.

Функция *test_diffusion.py* оказалась достаточно странной, поскольку даже после казалось бы полного исправления модели в ней часто не проходил $1.0 \leq output \leq 1.2$. Этот тест задаёт некоторые границы, между которыми результат лежит с определённой вероятностью. Можно было бы переписать его более правильно и проверять, что *assert* проходит достаточно часто (то есть в цикле делать это несколько раз или увеличить размер батча), однако мне показалось достаточным перезапускать тест несколько раз, чтобы убедиться в корректности.

1.2 tests/test_pipeline.py

Функция *test_train_on_one_batch* очень полезна для поиска ошибок при обучении (до этапа генерации). Так мы проверяем, что модель хотя бы может обучаться. Исправлять ничего не потребовалось.

Функция *test_training* была переписана мной так, чтобы работать только с одним семплом из датасета, при этом симулируя полноценный *torch.utils.data.Dataset*, в котором он повторялся несколько раз. Я сделал эту модификацию, чтобы поддерживать случаи $batch\_size > 1, len(dataloader) > 1$ уже в тестах. Один семпл же нужен, чтобы модель быстрее переобучилась и можно было понять, насколько хорошо она восстанавливает изображение. Результаты можно видеть ниже, однако для них потребовалось 20000 итераций на небольшом *batch_size*, поэтому я не стал делать обязательный тест на CPU в *test_training*, выбирая *device* в зависимости от наличия видеокарты

Рис. 1: Визуализация работы *test_training*. Исходные семплы (слева) и полученные диффузионкой (справа).

Name	Stmts	Miss	Cover
modeling/diffusion.py	38	0	100%
modeling/training.py	37	2	95%
modeling/unet.py	68	0	100%
TOTAL	143	2	99%

Таблица 1: Покрытие тестами директории *modeling*

1.3 modeling/diffusion.py

Метод $forward$ потребовал исправления

- (?) Timestep я семплировал с нуля, просто потому что индексация массивов начинается с этого числа. Потом я выяснил, что это не так важно, поскольку в $get\_schedules$ массив создаётся с запасом.
- Конечно же, мы хотим использовать $randn\_like$, это опечатка, которая обнаруживается методом пристального взгляда.
- Также был забыт $sqrt$ в коэффициент перед eps, там должно быть $sqrt\_one\_minus\_alpha\_prod$. Очень хотелось, чтобы диффузионку не пришлось исправлять, однако в итоге пришлось залезть в байесы и вспомнить формулы.

Метод *sample* также претерпел несколько модификаций

- Я добавил копию инициализации семплов, чтобы можно было их логировать.
- Также индексы *timesteps* в этом методе нужно согласовывать с использованием их в $forward$.

Я где-то час внимательно смотрел на $get\_schedules$, когда у меня плохо работало семплирование, в итоге ограничился добавлением условия $beta1 > 0$ в *assert*, других ошибок не нашёл.

1.4 modeling/training.py

В $train\_step$ я решил сразу возвращать $loss.item()$, чтобы не таскать весь граф вычислений следом.

В функции $train\_epoch$ багов обнаружено не было.

В $generate\_samples$ я стал сохранять/возвращать оба значения – начальное (шум) и конечное для семпла. Из содержательного я добавил сюда *processing*, который обращал нормализацию из $torchvision.transforms$, которая применялась к элементам перед попаданием в модель. Поскольку модель работала с афинным пребразованием пространства картинок, то необходимо было вернуть результаты семплирования обратно.

1.5 modeling/unet.py

В методе $forward$ не сходились размеры объектов *thro* и *temb*, поэтому нужно было добавлять размерности к эмбеддингам для правильного броадкастинга с картинками. Также в процессе дебага я убрал прибавление этих эмбеддингов к $up1$, решив, что это может быть багом. Остальные функции, к счастью, оказались написаны без ошибок.

1.6 Coverage

Наконец, результаты покрытия тестами

2 Настройка логирования

При тестировании работу логирования я не проверял, поскольку это скорее два разных способа отловить проблемы в коде, а логирование можно подебажить 20 перезапусками модели.

Для запуска обучения можно использовать

```
python main.py
```

Изменения параметров можно осуществить в файле $config.yaml$.

Логируемые значения

– (*loss*) Значение функции потерь после каждой итерации. EMA я логировать не стал, поскольку smoothing можно сделать в самом wandb.

– (*lr*) Динамика learning rate по ходу обучения, который в моём случае был константным.

– (*init*) Значение семпла до прогона через диффузионку, то есть белый шум. Так, например, мог быть отловлен баг с *rand* вместо *randn* в некоторых местах.

– (*sample*) Просемплированная картинка, обновляется раз в эпоху.

– (x) Первая картинка в батче на каждом шаге обучения. Это позволяет посмотреть визуализацию аугментаций и адекватность данных, что модель получает на вход.

Результат получился далеко не сразу, поскольку долгое время я оставлял в конфиге *weight_decay*. Никогда раньше не слышал, чтобы он мешал обучаться диффузионкам (точнее семплировать), однако видимо это так. Возможно, априорное предположение нормального распределение параметров не совсем уместно в такой сложной постановки задачи и сбивает параметры того нормального, которое мы рассматриваем для семплирования. Для примера прикладываю картинки с ним и без него

Рис. 2: Результаты семплирования в процессе обучения с $weight\_decay = 0.01$ (справа) и без него (слева).

3 Hydra и эксперименты

Конфигурацию я сделал подобную *pytorch_lightning*, то есть добавил отдельную пачку параметров для *trainer* (число эпох, шаг логирования, сам логгер), хоть такой сущности не было. Конструирование оптимайзера происходит с помощью поля *_target_* и метода *hydra.utils.instantiate*, поэтому для его изменения достаточно поменять $torch.optim.Adam \to torch.optim.SGD$ в конфиге.

Для экспериментов я ставил $num\_epochs := 20$, чтобы не тратить ограниченное GPU от Kaggle. Функцией *log_artifact* пользовался, ссылки на эксперименты прикрепил, поэтому изменение конфига там отследить можно.

3.1 Использование другого learning rate

Изменение в конфиг внести несложно

```
optim:
  lr: 1e-4
```

Я сделал *lr* немного выше, результат в логировании представлен ниже (wandb)

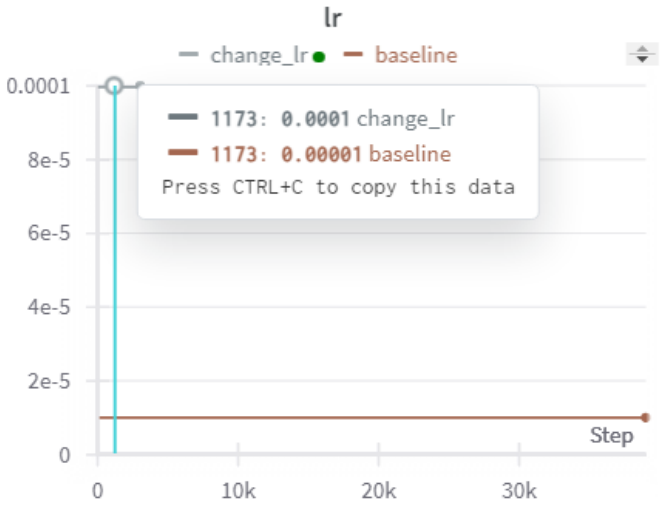

Рис. 3: Логирование разных значение lr (10^{-4} и 10^{-5}).

3.2 Подключение flip аугментаций

Для добавления flip (horizontal и vertical) я использовал отдельное поле

```
augments:
  flip: true
```

Результаты его применения представлены ниже (wandb)

Рис. 4: Входные картинки диффузионки с применением flip аугментаций (слева) и без них (справа).

3.3 Смена Adam на SGD

Здесь, как и описывалось выше, достаточно написать

```
optim:
  _target_: torch.optim.SGD
```

Результаты изменений напрямую не логируются, однако их можно видеть в графике функции потерь, которая отражает более “спокойный” метод SGD (wandb)

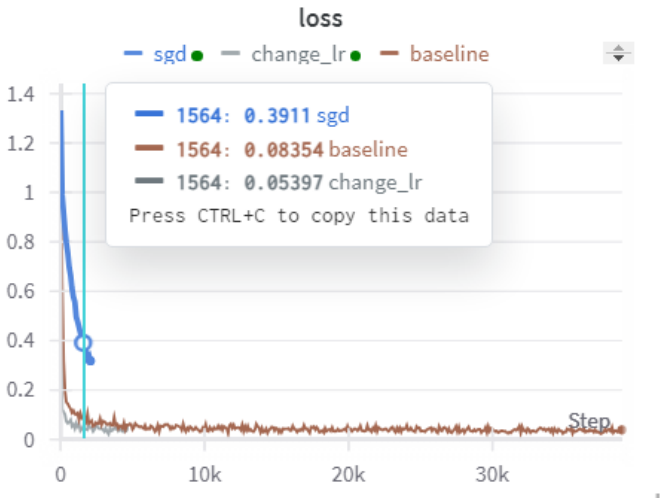

Рис. 5: Отражение смены оптимизатора на графике лосса (синий – SGD, остальные – Adam).

4 Добавление DVC

Для DVC я поставил небольшой эксперимент с тремя эпохами обучения, после чего запушил *dvc.lock* напрямую из Kaggle. Все нужные файлы можно найти в репозитории.