NeuralPoet.h5

Философствующий нейропоэт

Наверно в снах нам не нужны лишь два минутенья любви

Идея: Обучить нейросеть писать стихи и сделать её доступной в виде Telegram-бота

Шаги реализации:

1. Сбор и очистка данных
2. Выбор архитектуры нейросети и обучение
3. Тестирование и валидация нейросети
4. Подготовка готового сервиса
5. Деплой и тестирование

1.1 Сбор данных

Обучение на корпусе классической поэзии влечёт за собой две проблемы:

Во-первых сложно собрать данные, так как открытых корпусов мало, а датасеты Taiga и RussianCorpoa предоставляются только по запросу.

Во-вторых в этом мало смысла - хороших стихов всё равно не выйдет, а обижать классиков не хочется.

Поэтому принято решение собрать корпус любительской поэзии с сайта stihi.ru
Был написан простой однопоточный парсер на библиотеке requests с возможностью сохранять прогресс и начинать парсинг с того места, где остановился.

1.2 Очистка данных

Из собранных стихов были удалены лишние пробелы, ссылки, строки с цифрами и все стихи, содержащие латинские буквы.

```
df['poem'] = df['poem'].apply(clear_residual_spaces)
df['poem'] = df['poem'].apply(clear_lines_with_links)
df['poem'] = df['poem'].apply(clear_lines_with_numbers)
df['poem'] = df['poem'].apply(clear_short_lines)
df.dropna(inplace=True)
```

Также удалены слишком длинные и слишком короткие стихи. Смысла в них не много:

```
1  df[df['line_count']==2]['poem'].sample(1).values[0]
```

```
['жизнь коротка и может так сложится', 'что не успеть морально раз ложиться']
```

```
1  df[df['line_count']==2]['poem'].sample(1).values[0]
```

```
['попугайчики любовью занимаются', 'а моё воображение дурью мается']
```

Распределение слов в получившемся корпусе достаточно предсказуемо, но всё равно интересно

Средняя длина строки: 8-10 слов

80% корпуса содержит кратное четырём количество строк. Размер АБАБ самый популярный.

2.1 Подготовка данных для обучения нейросети

Все данные были пропущены через токенайзер предоставляемый библиотекой YouTokenToMe. Выбор обусловлен тем, что данный токенайзер не требует значительной предобработки и идеально подходит для обучение генеративной нейросети.

Размер словаря - 3500 токенов. Этого оказалось достаточно, чтобы качественно токенизировать датасет, к тому же сеть с выходным слоем размером 3500 легко может быть запущена на CPU. Это будет очень важно для последующего деплоя в облаке.

```
yttm_model_path = os.path.join(DRIVE_FOLDER, 'yttm.model')
yttm.BPE.train(data=yttm_txt_path, vocab_size=3500, model=yttm_model_path)

bpe = yttm.BPE(model=yttm_model_path)
```

2.2 Выбор архитектуры нейросети

Не зная как поведут себя современные генеративные архитектуры с использованием transformers в облаке, было принято решение взять классические рекуррентные сети.

Между LSTM и GRU решил не выбирать и взял обе.

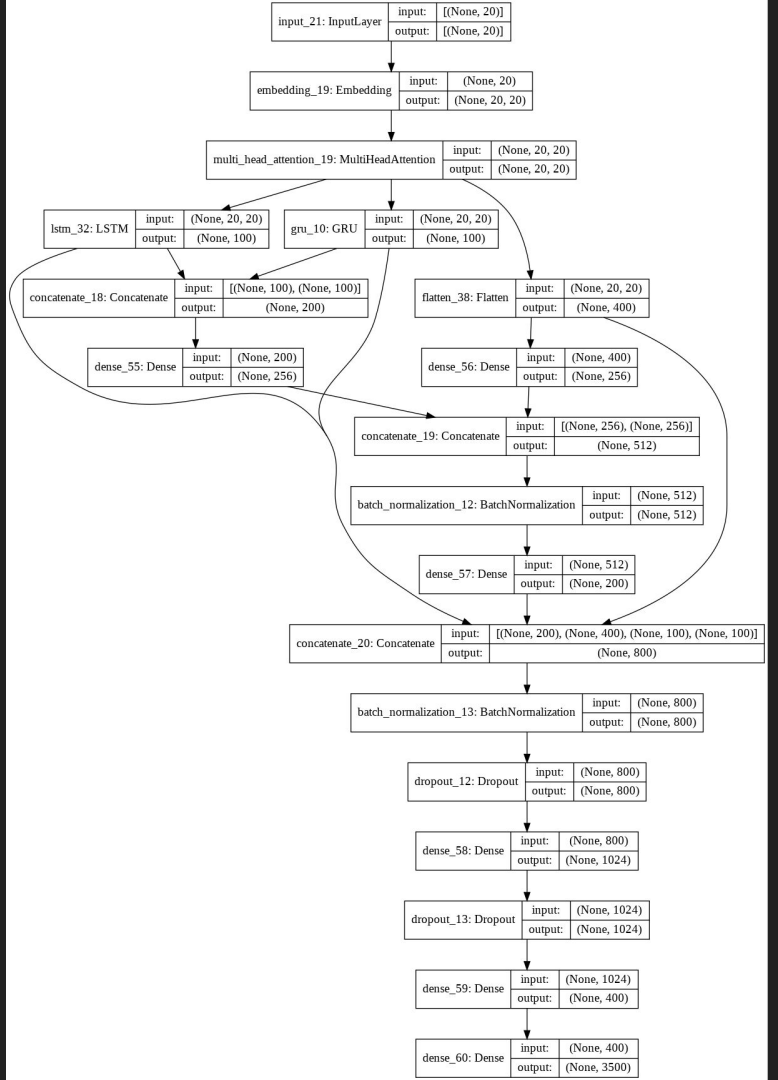
Чистые LSTM+GRU, работающие в параллели давали субъективно неплохой результат на генерации, но медленно обучались и иногда сходились к очень странному локальному минимуму, который при генерации выдавал только букву “И”.

Проблема скорости обучения и такого застревания решилась добавлением MultyHeadedAttention слоя.

Финальная архитектура выглядит так:

1. Вход нейросети (20 токенов) поступает в Embedding слой
2. На эмбеддинги смотрит восьмиголовы MultyHeadedAttention слой
3. Выход слоя внимания передается в модель, где параллельно работают LSTM+GRU слои, результат работы которых объединяется в один тензор
4. Конкатенируются тензоры слоёв MHA, LSTM, GRU и LSTM+GRU модели и поступают на вход полносвязной language modeling head
5. В конце линейный слой с количеством нейронов равным размеру словаря токенизатора.

Всё это учится минимизировать ошибку CategoricalCrossentropy по логитам (на выходе нет softmax активации)



2.3 Инференс

Обычно генерация текста работает так:

Для получения предсказания в модель подается токенизированная строка, которую следует дополнить. Для этой строки вычисляется следующий наиболее вероятный* токен. Результат этого вычисления снова подается на вход нейросети, чтобы получить следующий токен. Так повторяется N раз в цикле.

Подобный подход не совсем подходит для цели проекта: желательно чтобы бот даже на один и тот же пользовательский ввод каждый раз отдавал разные стихи.

Алгоритм beam-search с регулируемой температурой тоже не подошёл, так как работал слишком медленно на CPU.

Пришлось писать собственную функцию для предсказания.

2.3.1 Инференс поэтический

В цикл предсказания добавлены некоторые эвристики:

1. Случайно выбирать один из наиболее вероятных токенов, а не самый вероятный
2. Случайно выбирать из ещё большего пула токенов в начале каждой новой строки
3. Генерировать “вектор хаоса” состоящий такого количества из нулей и единиц, сколько шагов у цикла предсказания. Если на каждом шаге предсказания, в векторе оказывается единица - следующий токен выбирается из значительно большего пула наиболее вероятных кандидатов.

   Все значения регулируются. Достаточно изменить пару параметров чтобы перейти от стихов про любовь к чистому лингвистическому безумию.

```
def predict_on_string(my_string,
                      max_len=60,
                      seq_len=20,
                      top_k=1,
                      variant_newline=True,
                      newline_top_k = 20,
                      chaos_top_k=0,
                      chaos_rate=0.5):
```

3. Тестирование и валидация

Придумать адекватную метрику для данной задачи не удалось. Нейросеть тренировалась пока accuracy на тестовой выборке (10% данных) не начнет уменьшаться.

Валидация проводилась вручную по субъективным ощущением. Целью было получить достаточную степень осмысленности сгенерированных текстов при приемлемом уровне “удивительности”.

Параметры функции предсказания настраивались так же.

Итоговый результат слабо похож на поэзию - для сохранения рифмы требовались бы сложные эвристики или совсем другой подход к обучению.

И скорее всего совсем другой датасет.

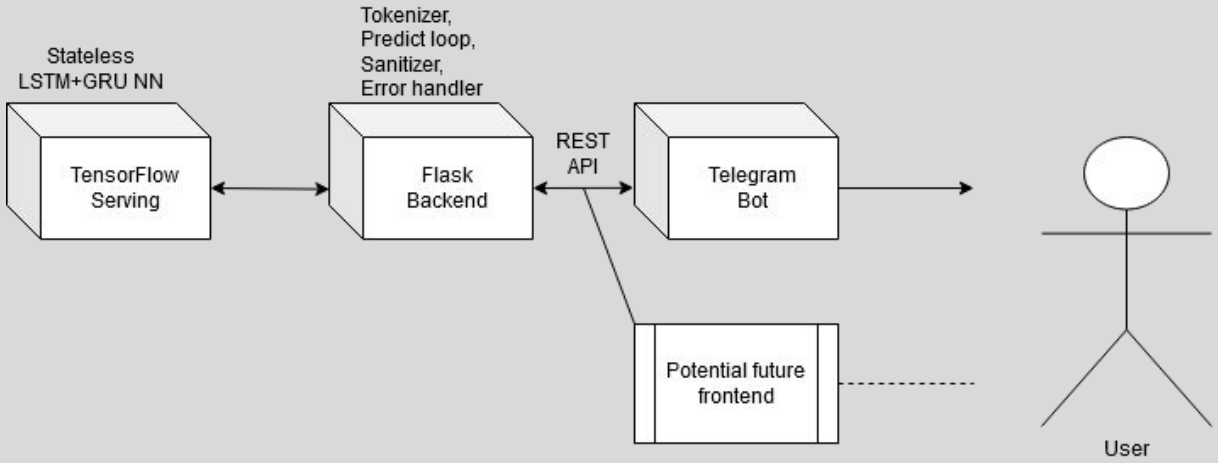
```
array([list(['остынь не приставай к атлету', 'он превратит тебя в котлету'])
```

4. Подготовка сервиса

Итоговый сервис состоит из следующих частей:

1. Нейросеть
2. Flask backend предоставляющий API-метод предсказания телеграм боту и другим потенциальным пользователям
3. Телеграм-бот

Для реализации выбрана сервисная архитектура. Каждый сервис запущен в docker-контейнере.



5. 1 Деплой

Все сервисы были обернуты в докер. Для TF-Serving контейнера отдельно протестирована возможность работы на CPU и скорость такой работы.

Контейнеры общаются внутри виртуальной локальной сети через REST-API

Для деплоя был арендован сервер облачный инстанс с 6GB RAM, хотя дальнейшие тесты показали, что достаточно будет 3GB RAM

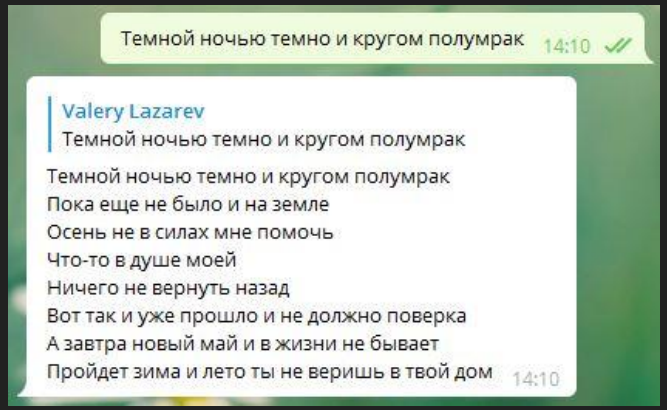
Запуск всего сервиса осуществляется командой docker-compose up

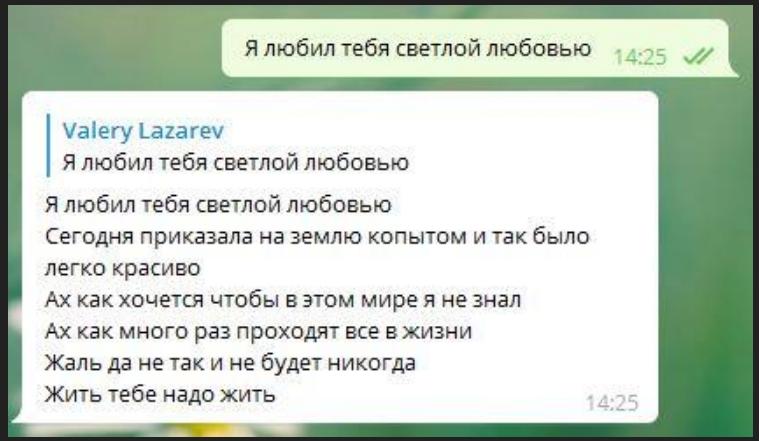
```
root@37-140-198-103:~# cd lstm-poet/
root@37-140-198-103:~/lstm-poet# docker-compose up
Creating network "lstmpoet_default" with the default driver
Creating backend ...
Creating telegram_bot ...
Creating tf-serving ...
Creating tf-serving
Creating telegram_bot
Creating tf-serving ... done
Attaching to backend, telegram_bot, tf-serving
```

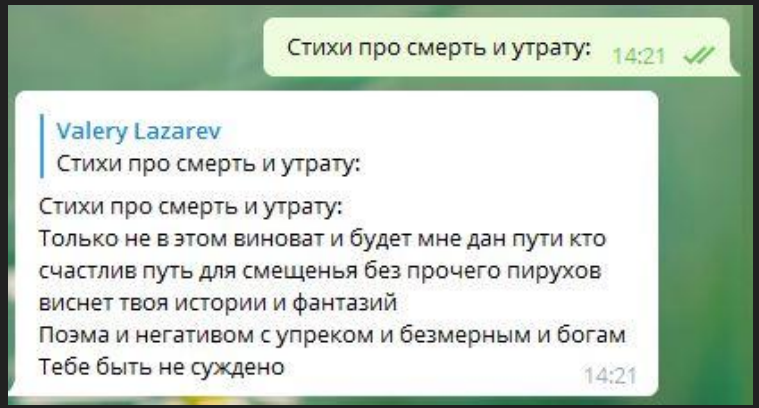
← (Запускать под рутом лучше, конечно, не надо)

5.2 Тестирование

Telegram @neural_poet_bot







Можно попробовать прямо сейчас

Заключение

Основной целью проекта была end-to-end реализация продукта на основе нейросетей. От сбора данных до готового результата доступного всем.

Данную цель для себя считаю выполненной на 100%.

У этого конкретного проекта нет способа монетизации и нет потенциала для развития, но такой потенциал есть у сотен других моделей машинного обучения и сервисов, которые я теперь могу создавать и доводить до готового изделия.

Именно этот, личный, результат мне кажется самым важным итогом моей дипломной работы.

```
backend  | GENERATED:
backend  |  Загадочна душа поэта
backend  | Измучилась и не заявила в мозги и даже разбирая стихи прост да
backend  | И до сих раз на свете жить
backend  | И не нахожу себя
backend  | Что я теперь о том о себе пророслющу себя обряженный в этом суть вселенной козырь
```