딥러닝 프로젝트 보고서

20172605 컴퓨터학부 김도현

1. 요약

이 프로젝트는 숭실대학교의 딥러닝 수업에서 진행한 수업을 바탕으로, 딥러닝을 최소한의 dependancy만 가지고 Rust언어를 사용하여 구현하였습니다. 구현한 코드로 MNIST를 학습시키는데 성공하였고, 에세이 평가 데이터를 학습시켰습니다.

2. 서론

배경

이 프로젝트는 숭실대학교의 딥러닝 수업에서 진행한 수업을 바탕으로, 에세이 평가 데이터를 직접 딥러닝을 구현하여 학습시키는 프로젝트가 나와서 진행하게 되었습니다.

문제정의

딥러닝을 직접 구현하고, 에세이 평가 데이터를 활용하여 딥러닝하기

목표설명

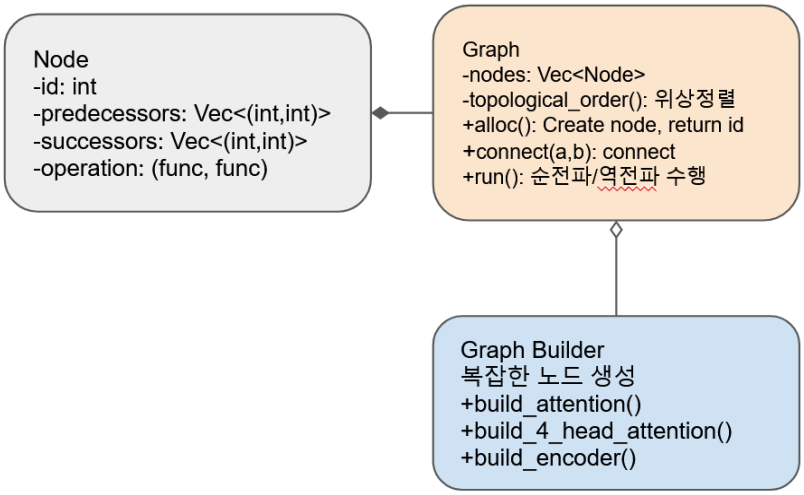
딥러닝을 Rust로 재구현하고, 먼저 MNIST를 학습시킨 이후, 최종적으로 에세이 평가 데이터를 활용해 에세이 평가 AI를 학습시키기

3. 방법

프로젝트 코드(GitHub)

https://github.com/wolfrev0/dl.rs

프로그램 구조



[그림1] 프로그램 구조도 (UML like)

먼저 계산그래프가 반드시 필요하다고 생각하여 Graph구조를 설계한 이후, attention을 작성하다 보니 복잡한 연산(Attention, Multi-head Attention, Encoder)은 한번에 코딩하는게 어렵기 때문에 작은 연산들로 쪼개서 구성하는게 좋다는 것을 발견했고, Builder를 따로 분리했습니다.

계산그래프는 임의의 그래프를 받고, 이후에 assert를 사용해서 의도한 형태의 그래프(DAG이면서 snk노드가 하나)가 아닐경우 메시지와 함께 프로그램을 종료하도록 했습니다.

Input은 4D tensor이며, 축 순서대로 batch, feature, y-axis, x-axis입니다.

Backpropagation

계산그래프로 표현된 뉴럴네트워크는 손으로 계산하여 하드코딩하기에는 상당히 복잡해지기 때문에 자동미분을 구현하기로 했습니다. MNIST를 학습할 때 까진 계산그래프가 Tree형태여서 처음에는 snk에서 진행하는 BFS로 처리했습니다. 그러나 이후 residual connection이 존재하는 Transformer Encoder를 구현하면서 DAG형태의 계산그래프에서도 자동미분이 가능하도록 위상정렬과 Dynamic Programming을 사용해 자동미분의 구현을 개선하였습니다.

테스트코드

Rust에서 제공하는 test기능을 사용했습니다. 각종 연산의 Rust구현체의 동작을 검증하기 위해 먼저 pytorch로 해당 연산에 대한 Reference Code를 작성한 다음, input/output/gradient를

계산한 값을 assert를 사용해 검증하였습니다. 또한 GitHub CI에 rust build와 rust test를 추가하여 변경사항 발생시 자동으로 빌드 및 테스트를 진행하도록 하였습니다.
https://github.com/wolfrev0/dl.rs/blob/main/src/test.rs

단어임베딩

Word2vec, fasttext, glove 등의 단어임베딩 라이브러리가 Rust에는 마땅한 crate가 없어서 사용법을 한참 찾다가 실패했는데, .vec파일을 직접 열어서 구조를 확인해보니 라인별로 단어 하나의 벡터를 저장하고 있는것을 알게 되어 그냥 직접 파싱해서 쓰기로 결정했습니다.
한국어 .vec파일은 아래 링크에서 확보할 수 있었습니다.
https://github.com/Kyubyong/wordvectors
https://fasttext.cc/docs/en/crawl-vectors.html

전자는 전자는 3만라인정도에 hidden=200이고 형태소분석이 된 상태였습니다.
후자는 200만라인정도인데, 100만라인 이상 읽으면 실패하는현상이 있습니다. 아마도 특수문자 등으로 데이터가 깨져있는것으로 추측합니다. hidden=512이며 형태소분석은 별도로 되어있지 않은듯 합니다.
컴퓨팅 자원이 열악한 관계로 첫번째 임베딩을 사용하기로 결정했습니다.

OOV(Out-of-Vocabulary)에 대한 고찰

한국어의 특성상 띄어쓰기만으로 단어를 구분하면, 거의 동일한 의미인데 형태소에 의해 살짝만 다른 단어가 많이 존재할 것입니다. 그렇기 때문에 형태소분석이 들어가야 OOV의 빈도가 낮아지고 좋은 성능이 나올 것 입니다. 그렇기 때문에 konlpy의 kkoma를 사용하여 형태소분석을 수행하였습니다.

다음의 글은 OOV가 발생한 경우에 해당 OOV단어와 문장을 가지고 비슷한 의미의 단어를 추론하여 해결합니다.
https://medium.com/analytics-vidhya/handling-out-of-vocabulary-words-in-natural-language-processing-based-on-context-4bbba16214d5 OOV가 자주 발생하지 않는다면 훌륭한 해결책인것 같고 재밌어 보였지만, 프로젝트의 규모가 커져 시간내에 구현하기 힘들 것같았습니다.

좀 더 고민해보니 한글은 형태소에 의한 단어형태의 변화 때문에 OOV가 많이 생기는 편인데, 이 형태소는 많은 경우 앞부분+뒷부분으로 자연스럽게 나눌 수 있습니다. 그래서 prefix로 정렬된 단어목록에서 Longest Common Subsequence가 제일 큰 것을 선택하기로 했습니다. OOV가 단어 뒷부분인 경우를 처리하기 위해 뒤집은 단어도 후보로 추가해주면 됩니다.

에세이 데이터에 대한 고찰

일단 데이터가 글짓기,대안제시,설명글,주장,찬성반대 5가지 카테고리로 나누어져 있었습니다. 서로 다른 맥락의 작문을 혼합하여 한번에 학습하는것은 어려울듯 하여 글짓기 데이터만 사용하기로 결정했습니다. 하나의 Json파일에도 여러가지 데이터가 있는데, 글주제와 본문

그리고 글의 점수를 제외한 나머지 모든 데이터는 에세이 작문의 점수와는 전혀 상관없거나 심지어 학습을 방해하는 데이터이기 때문에 학습에 사용하지 않았습니다. 예를들어 에세이를 평가할 때 저자의 학교(초/중/고)정보가 있다면 단순히 고등학생의 작문을 좋다고 평가하도록 학습할 것이고, 글의 길이나 작성일자 같은 경우는 작문의 점수와 전혀 상관없기 때문입니다.

학습 데이터 전처리

paragraph가 여러개 존재하는 데이터가 있는데, 별로 중요한것 같지 않아서 하나로 합쳐서 사용했습니다.
'#@문장구분#'은 형태소분석시에 다 쪼개지는데, 이를 막기위해 #하나로 대체하였습니다. 글짓에서 #을 사용하지 않기 때문에 학습에는 영향이 없을 것입니다.
형태소분석은 Konlpy의 kkma를 사용하였고, 다음과 같은 출력을 얻습니다.

```
익숙함에 속아 소중함을 잊지 말자는
```

```
[('익숙', 'XR'), ('하', 'XSA'), ('ㅁ', 'ETN'), ('에', 'JKM'), ('속', 'W'), ('아', 'ECD'), ('소중', 'XR'), ('하', 'XSA'), ('ㅁ', 'ETN'),
```

튜플의 첫 원소만 남기고, 형태소의 분리를 위해 다음과 같이 데이터를 저장하였습니다.

```
익숙@하@ㅁ@에@속@아@소중@하@ㅁ@
```

30점 만점인 점수는 0~1사이로 scale하였습니다.
최종적으로 전처리된 데이터는 다음과 같습니다.


```
{
  "prompt": "익숙@하@ㅁ@에@속@아@소중@하@ㅁ@을@잊@...",
  "paragraph": "나의@소@중함@은@나의@친구@이@다@.@#@하지만@내일@이@면@...",
  "score": 0.7472222
}
```

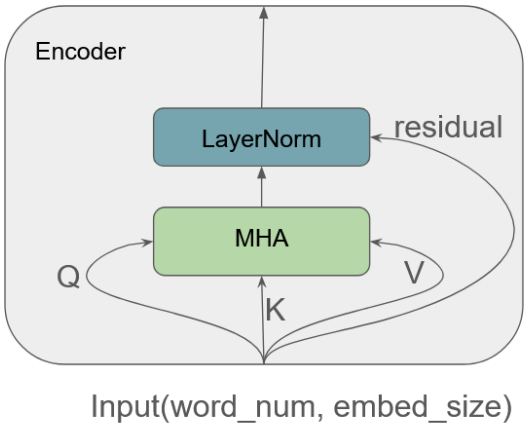

모델 선택

Transformer류 모델(GPT와 BERT)들이 대세인것 같아서 공부해보았습니다.

- BERT는 양방향 문맥을 포착하므로, 개체 인식(entity recognition), 개체 관계 추출(entity relation extraction), 감성 분석(sentiment analysis) 등 다양한 NLP 작업에서 뛰어난 성능을 보입니다.
- GPT는 생성 작업(generative tasks)에 더 적합합니다. 예를 들어, 텍스트 생성, 요약, 기계 번역 등의 작업에서 GPT는 높은 성능을 발휘합니다.

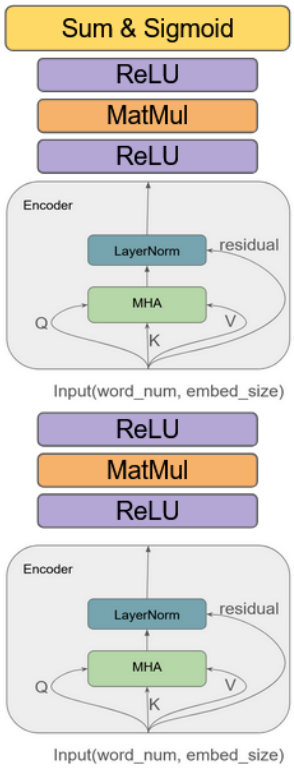
이 프로젝트에서 해결하고자 하는 에세이평가 task는 문장의 특징을 추출해서 점수를 Regression하는 NLP task이므로 BERT구조가 더 적합하다고 판단했습니다. BERT를 바닥부터 학습시키기는 무리인것 같아서 BERT의 핵심구조인 multi-head self attention은 유지하면서 파라메터를 적극적으로 줄인 모델을 만들기로 했습니다.

새로 설계할 신경망(이하 EssayNet)의 Encoder의 구조는 아래와 같습니다.



[그림 2] 간략화한 Encoder

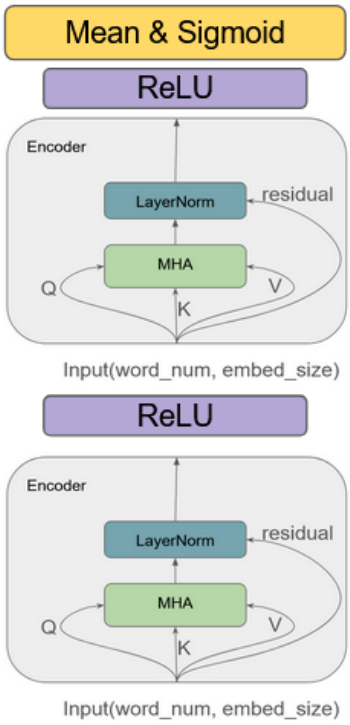
노트북 CPU만으로 학습이 가능해야 하므로 파라미터를 최대한 적게 유지하기 위해 위의 Encoder구조를 2개만 사용하였습니다. 또한 Encoder로 추출한 특징을 결합하기 위해 MatMul을 추가하고, 비선형성을 추가하기 위해 ReLU를 추가했으며, 점수는 Scalar값이므로 하나의 값으로 sum연산을 해준 다음, 점수를 0~1사이로 scale하기 위해 Sigmoid를 사용하여 아래와 같이 구성해보았습니다.



[그림3] EssayNet의 첫번째 계산그래프

그러나 학습이 잘 안되었습니다. 일단 분석해보니 score가 0.5 이하로 내려가지 못했는데, Sigmoid 바로 직전에 ReLU연산을 하기 때문에 모든 값이 0이상이 되버렸고, 그래서 음수가

나오지 않아서 sigmoid가 0.5이하가 되지 못하고 있었습니다. 그래서 sigmoid 직전의 ReLU는 제거하였습니다.
그래도 기울기가 0이 되버리는 현상이 있었는데, sigmoid(sum(x))를 하면 hidden이 작을때는 sum값이 작아서 잘 학습되지만, hidden이 커지면 sum값도 커지기 때문에 sigmoid의 기울기가 소실되는 문제였습니다. 그래서 sigmoid(mean(x))으로 대체하였고, mock learning이 잘 되는것을 확인했습니다.



[그림4] EssayNet의 최종 계산그래프

학습

계산그래프의 학습은 SGD를 사용했으며, Gradient가 있을때 SGD를 수행하는것은 간단한 for loop하나로 끝나기 때문에 별도의 파일 분리 없이 main함수에서 수행하였습니다.
일단 수렴성을 확인하기 위해, 모든 랜덤입력에 대해 라벨을 고정된 상수 k로 하는 Mock learning을 진행하였고, 해당 값으로 잘 수렴하는것을 확인할 수 있었습니다.
실제 데이터에 대해 학습할 때는 총 10 epoch를 수행하였고, 매 2번의 epoch마다 learning rate를 1/10으로 줄여나가며 학습하였습니다. (아래의 수식 참고)

$$lr = \frac{lr_{base}}{10^{epoch/2}}$$

성능최적화

실제 글쓰기 데이터로 실행시켜보니 너무 느렸고, 병목지점을 분석해보니 layernorm과 softmax의 backward 시간복잡도가 문제였습니다. 두 함수 모두 Vector to Vector 함수였고, Backward를 구현하기 위해 Jacobian행렬을 구해서 기울기를 구했습니다. 그런데 Jacobian을 구하는 것은 입력크기 N에 대해 시간복잡도가 O(N^2)이기 때문에 문제가 됩니다. 이를 식정리를 통해 전처리가 가능한 형태로 변형하여 해결하였습니다.
Softmax의 Jacobian을 사용해 바로 얻어지는 dx의 기울기는 다음과 같습니다.

$$\sum_j \frac{dy_j dz}{dx_i dy_j} = \sum_j y_j(\delta_{ij} - y_i)\frac{dz}{dy_j} \; where \; \delta_{ij} = if \; i == j \; then \; 1 \; else \; 0$$

정리하면 아래와 같이 쓸 수 있습니다.

$$\sum_j \frac{dy_j dz}{dx_i dy_j} = -y_i \sum_j y_j \frac{dz}{dy_j} + y_i \frac{dz}{dy_i}$$

이제 j에 대한 sigma를 전처리해두면 모든 i에 대해 O(1)에 x_i의 gradient를 구할 수 있습니다.

4. 결과

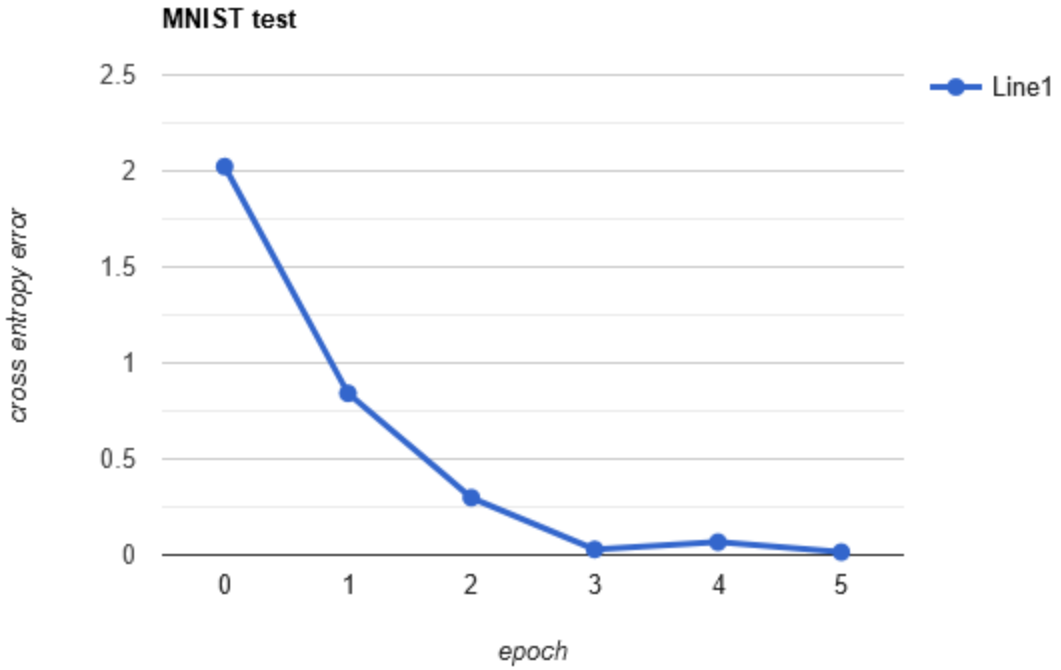
학습에 필요한 AI연산의 forward/backward 구현 성공

- identity
- matmul
- relu
- eltw_add
- eltw_mult
- softmax_xy
- softmax_y
- softmax_cross_entropy
- transpose
- layer_norm
- concat4

Multi-head Attention과 Encoder는 위의 연산들을 가지고 graph_builder.rs에서 조합하여 만들었습니다.

MNIST학습 성공

본격적인 NLP작업을 하기 전에, 프로그램이 전반적으로 잘 작동하는지 검증하는 차원에서 간단한 MatMul-ReLU-MatMul-ReLU-SoftMax 신경망을 만들고 MNIST 학습을 진행시켜보았습니다.

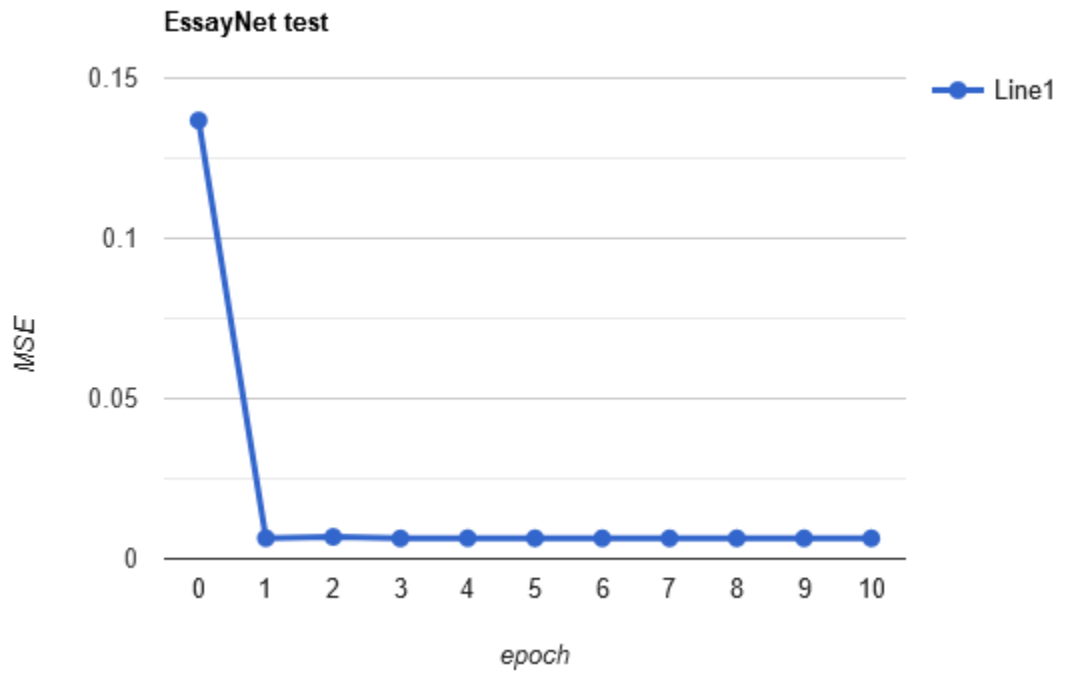


[그림5] MNIST test error
잘 학습하는것을 확인할 수 있었습니다. 학습 시간도 노트북 컴퓨터 CPU로 5 epoch를 1분 미만에 완료하였습니다. Main함수 코드는 아래에서 확인할 수 있습니다.
https://github.com/wolfrev0/dl.rs/blob/main/src/bin/mnist.rs

오픈소스 기여

MNIST를 학습시키면서 mnist crate의 개선할만한 부분(실제파일이름과 default path 불일치)을 발견하여 PR(https://github.com/davidMcneil/mnist/pull/18)을 만들었습니다.

EssayNet 학습



[그림6] EssayNet test error
결과적으로, 완전랜덤가중치(epoch 0)보다는 Error가 감소했지만, 신경망의 출력값이 score의 평균값 주변을 출력하는것으로 수렴하고 일반화는 하지 못했습니다.

5. 논의 및 결론

결과에 대한 해석

학습이 잘 안된 이유는 데이터부족이 원인인듯 합니다. 4천개의 점수라벨링된 글이 학습 데이터였는데, 이는 고작 24MB입니다. 현존하는 NLP AI들은 GB단위 혹은 그 이상 학습하는것을 생각해보면 너무 적습니다. 작은 word embedding을 사용했는데도 불구하고, embedding vector의 size가 200이었는데, 이는 encoder 내부에 200x200행렬 여러개가 존재한다는것이고, 이를 학습하기 위한 데이터로 라벨링데이터 4000개는 턱없이 부족합니다. 또 한편으로는 Encoder 2개를 쌓은 정도로 학습하기에는 모델의 파라미터 용량이 부족했던 것일 수 있습니다.
그래도 기본적인 AI연산들과 그 학습코드를 pytorch와 테스트코드로 검증했기 때문에, 추후에 더 개선된 컴퓨터 환경에서 이 프로젝트를 코드베이스로 더 많은 데이터와 더 좋은 모델로 도전해볼 가치가 있을듯 합니다.

개선할 점

- 설계상 잘못된 부분이 있었다고 생각합니다. 계산그래프에 동적할당을 사용하고 싶지 않아서 function을 사용했는데, 왜 pytorch같은 자동미분 라이브러리들이 Operation내에 Learnable Parameter를 두는지 알 수 있었습니다. 매번 연산 호출시마다 가중치를 같이 입력해주는것이 상당히 번거로운 일이어서 multi-head attention같이 가중치가 많은 연산을 구현하고 테스트 하는것 뿐만 아니라 사용하는것도 힘들었습니다.
- 시간이 부족하여 Multi Head Attention이 Single Head Attention을 여러번 하는 방식으로 작성하였는데, 한번의 (약간 수정된)Attention으로 계산할 수 있습니다.
- Attention도 간단한 연산들로 조합하지 않고 한번에 연산하면 overhead를 많이 줄일 수 있을 것으로 생각됩니다.

6. 참고문헌

Backpropagation

https://ksm2853305.tistory.com/46

Attention

https://heekangpark.github.io/nlp/attention
https://stats.stackexchange.com/questions/565196/why-are-residual-connections-needed-in-transformer-architectures
https://pytorch.org/docs/stable/generated/torch.nn.MultiheadAttention.html

Transformer & GPT & BERT

https://wikidocs.net/31379
https://wikidocs.net/103802
https://en.wikipedia.org/wiki/Generative_pre-trained_transformer#/media/File:Full_GPT_architecture.png
https://www.postech.ac.kr/ppostechian-section/2023-178%ED%98%B8-%EA%B8%B0%ED%9A%8D%ED%8A%B9%EC%A7%91-%E2%9E%81-chatgpt/
https://ratsgo.github.io/nlpbook/docs/language_model/bert_gpt/