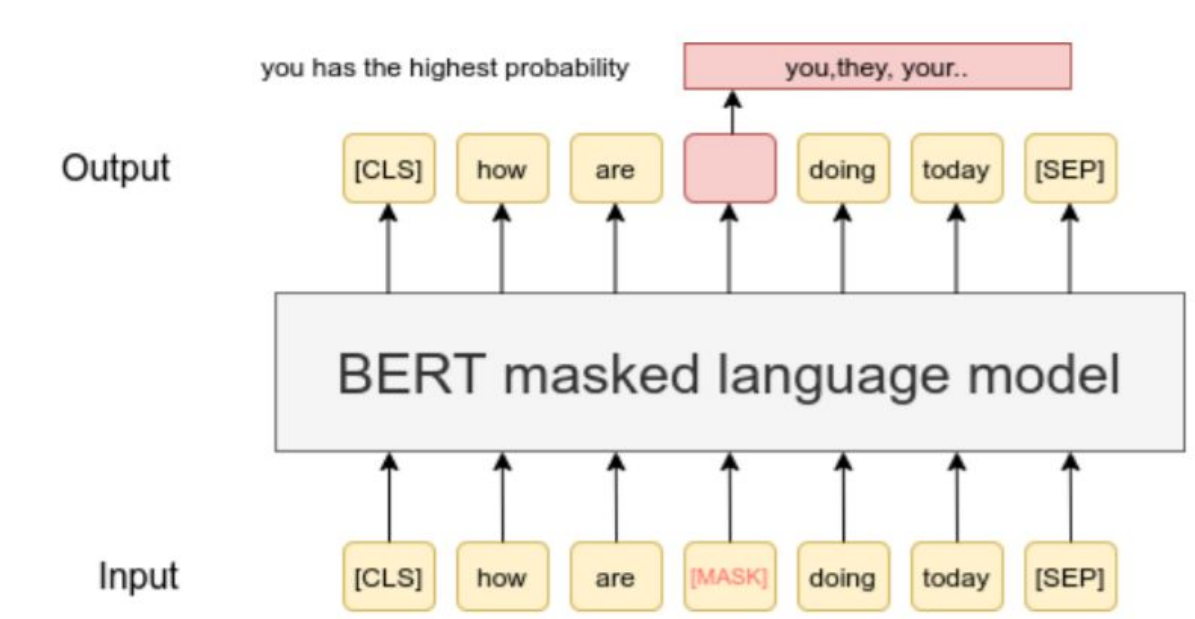
ELECTRA

Efficiently Learning an Encoder that Classifies Token Re-placements Accurately

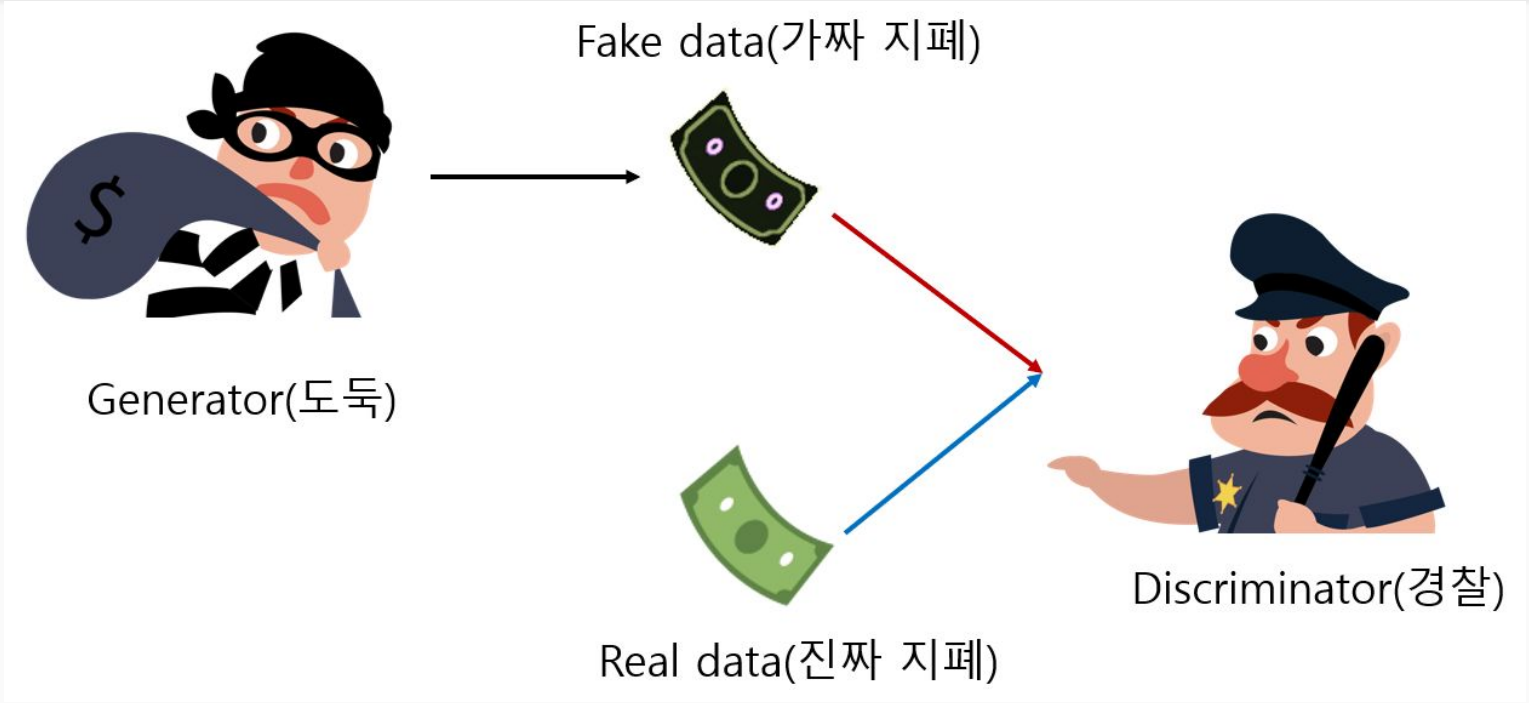
BERT: Denoising Autoencoder



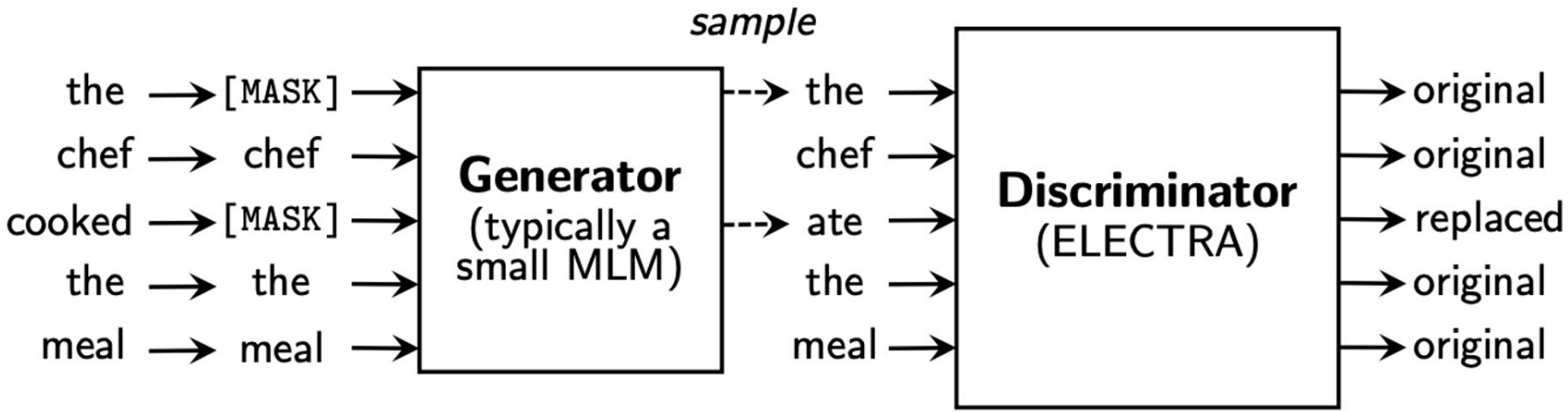
입력 시퀀스의 토큰 중 약 15% 정도를 latent vector([MASK] 토큰)로 바꾸고 원래 데이터로 복원하면서 학습함

-> 하나의 example에 대해서 전체 토큰 중 고작 15%만 학습
-> 학습이 오래걸리고 비용이 비쌈

경찰과 도둑: Replaced Token Detection(RTD)

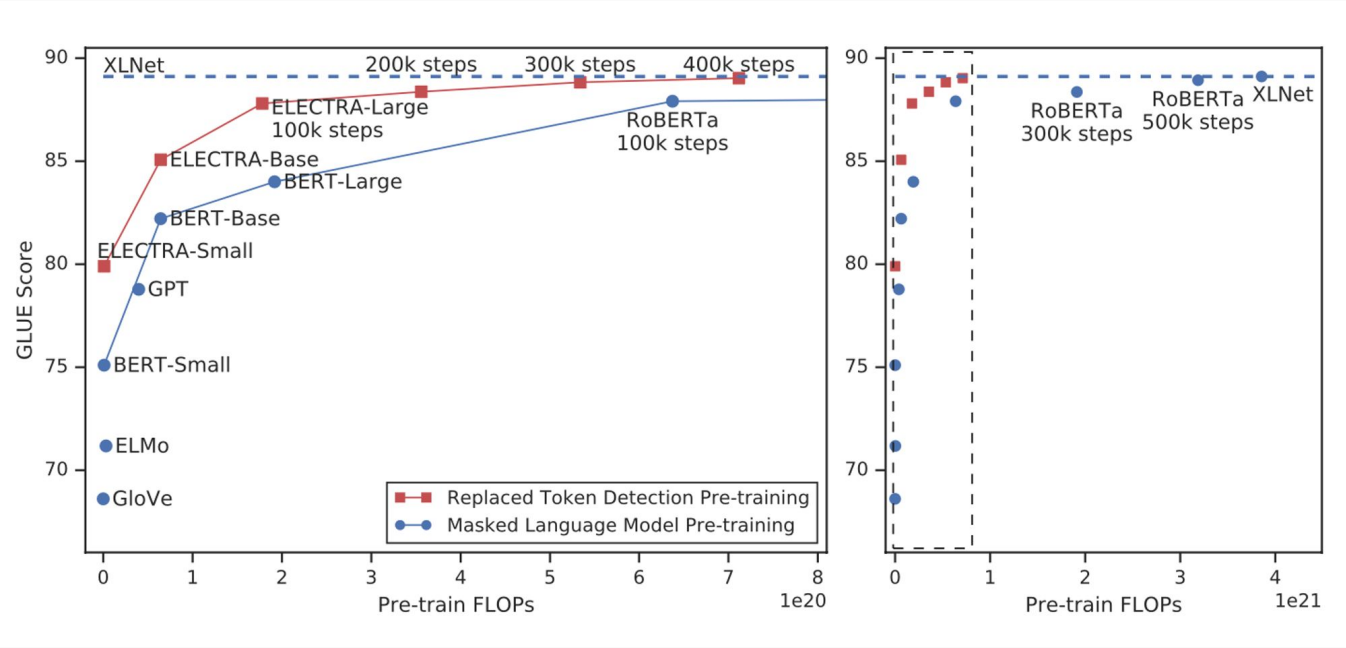


RTD: Replaced Token Detection



기존 BERT 모델을 Generator로 쓰고 각 토큰을 이진 분류하는 Discriminator를 도입

적은 연산량으로도 좋은 성능을 내는 것

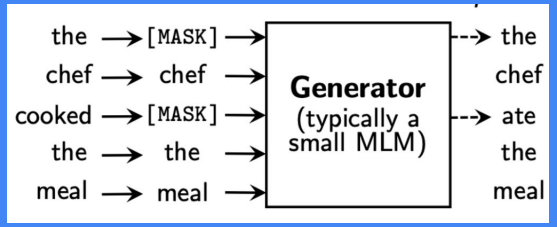


ELECTRA-Small > GPT2
ELECTRA > BERT

-> 특히, 작은 모델에서 계산량 대비 성능이 뛰어나다

-> BERT는 추론 시 [MASK] 토큰을 사용하지 않아서 성능 저하로 이어질 수 있다. [MASK] 토큰 의존.

Generator



전체 토큰 n개 중 마스킹할 집합 m을 결정

$$m_i \sim \text{unif}\{1, n\} \text{ for } i = 1 \text{ to } k$$

입력토큰을 [MASK]로 치환

$$\boldsymbol{x}^{\text{masked}} = \text{REPLACE}(\boldsymbol{x}, \boldsymbol{m}, \texttt{[MASK]})$$

[MASK]토큰 중 특정 위치 t에서 토큰 xt를 생성

$$p_G(x_t|\mathbf{x}^{masked}) = \exp(e(x_t)^T h_G(\mathbf{x}^{masked})_t) / \sum_{x'} \exp(e(x')^T h_G(\mathbf{x}^{masked})_t)$$

$$\boldsymbol{x}^{\text{corrupt}} = \text{REPLACE}(\boldsymbol{x}, \boldsymbol{m}, \hat{\boldsymbol{x}})$$

e(xt)T hG(x^masked)t 는 토큰 xt의 임베딩 벡터와 생성자 G의 인코더 출력 벡터 사이의 내적을 계산

e(·)는 임베딩을 의미

토큰 xt가 해당 위치 t에 얼마나 적합한지 나타내는 점수

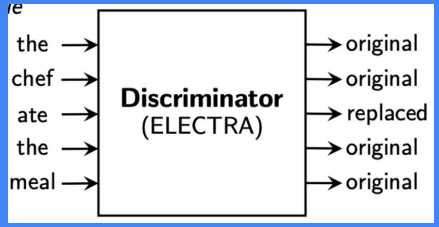
$$\mathcal{L}_{\text{MLM}}(\boldsymbol{x}, \theta_G) = \mathbb{E}\left(\sum_{i \in \boldsymbol{m}} -\log p_G(x_i|\boldsymbol{x}^{\text{masked}})\right)$$

Implement generator

```
def _get_masked_lm_output(self, inputs: pretrain_data.Inputs, model):
    # ...
    relevant_reprs = pretrain_helpers.gather_positions(
        model.get_sequence_output(), inputs.masked_lm_positions)
    logits = get_token_logits(
        relevant_reprs, model.get_embedding_table(), self._bert_config)
    # ...
    return get_softmax_output(
          logits, inputs.masked_lm_ids, inputs.masked_lm_weights,
          self._bert_config.vocab_size)
```

- model.get_sequence_output() 함수를 사용하여 G의 트랜스포머 인코더 출력을 가져온다.
- pretrain_helpers.gather_positions 함수를 사용하여 마스킹된 토큰 위치에 해당하는 벡터 표현을 추출
- get_token_logits 함수를 사용하여 추출된 벡터 표현으로부터 각 토큰에 대한 로짓(확률)을 계산
- get_softmax_output 함수를 사용하여 로짓을 소프트맥스 함수에 통과시켜 확률 분포를 얻는다.

Discriminator



각 토큰이 original인지 replaced인지 이진 분류로 학습
-> 마스킹 된 토큰이 아닌 전체 토큰을 학습할 수 있다.

$$D(\mathbf{x}^{corrupt}, t) = \text{sigmoid}(w^T h_D(\mathbf{x}^{corrupt})_t)$$

최종적으로 두 loss의 합을 최소화 시키도록 학습

$$\min_{\theta_G, \theta_D} \sum_{\boldsymbol{x} \in \mathcal{X}} \mathcal{L}_{\text{MLM}}(\boldsymbol{x}, \theta_G) + \lambda \mathcal{L}_{\text{Disc}}(\boldsymbol{x}, \theta_D)$$

xt가 생성된 토큰이라면 D가 1이라고 추론하고,
생성되지 않은 토큰이라면 0이라고 추론하도록 학습

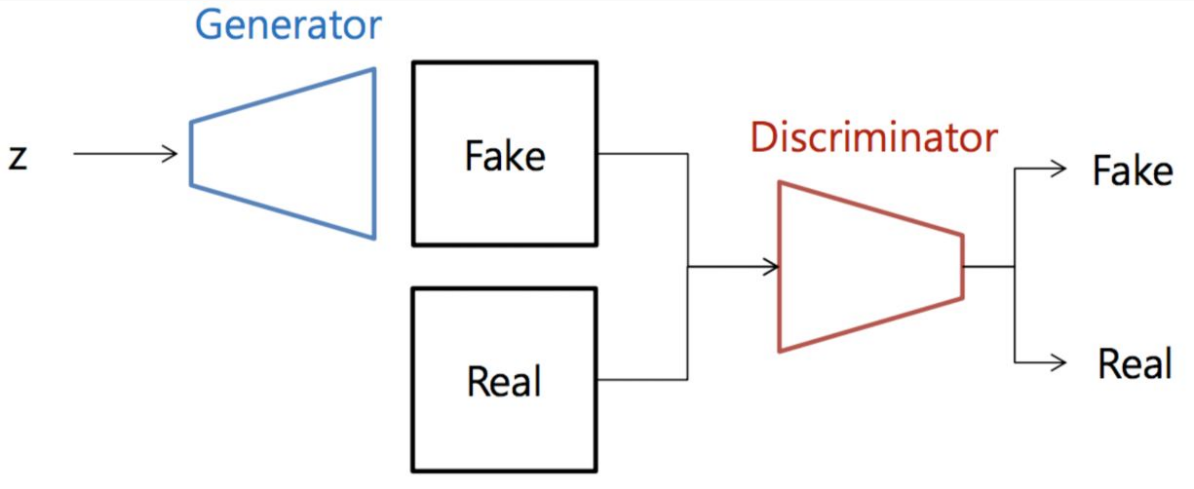
$$\mathcal{L}_{\text{Disc}}(\boldsymbol{x}, \theta_D) = \mathbb{E}\left(\sum_{t=1}^{n} -\mathbb{1}(x_t^{\text{corrupt}} = x_t) \log D(\boldsymbol{x}^{\text{corrupt}}, t) - \mathbb{1}(x_t^{\text{corrupt}} \neq x_t) \log(1 - D(\boldsymbol{x}^{\text{corrupt}}, t))\right)$$

Implement discriminator

```
def _get_discriminator_output(self, inputs, discriminator, labels):
    # ...
    weights = tf.cast(inputs.input_mask, tf.float32)
    labelsf = tf.cast(labels, tf.float32)
    losses = tf.nn.sigmoid_cross_entropy_with_logits(
        logits=logits, labels=labelsf) * weights
    per_example_loss = (tf.reduce_sum(losses, axis=-1) /
                        (1e-6 + tf.reduce_sum(weights, axis=-1)))
    loss = tf.reduce_sum(losses) / (1e-6 + tf.reduce_sum(weights))
    # ...
```

- tf.cast(labels, tf.float32)를 통해 이를 float32 타입으로 변환합니다. 원래 토큰은 1, 대체 토큰은 0으로 표현
- weights는 패딩 토큰의 여부를 나타냅니다.(실제 토큰인지 아닌지)
- labels는 각 토큰이 원래 토큰인지 대체 토큰인지 나타내는 텐서
- tf.nn.sigmoid_cross_entropy_with_logits 함수로 로짓(logits)과 레이블(labelsf)을 기반으로 loss를 계산
- 따라서 loss는 위에서 계산한 loss를 실제 토큰의 갯수로 나눈 값이 된다.(1e-6은 0으로 나누는 것을 방지)

GAN



generator가 discriminator를 속이도록 학습(adversarial 학습)

discriminator는 가짜 토큰을 잘 구별하도록 학습

-> 실험 결과 maximum likelihood 학습이 더 성능이 좋은 것으로 확인

Weight sharing

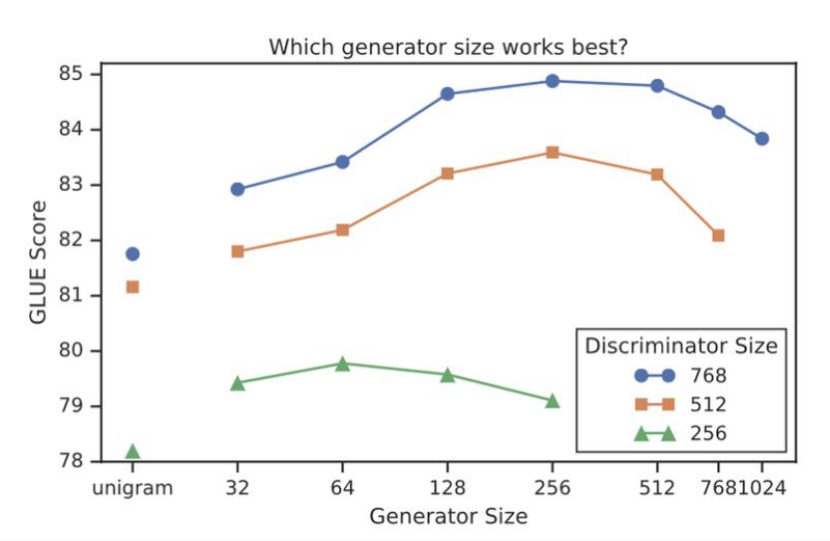
Generator와 Discriminator는 모두 Transformer 인코더 구조 -> weight sharing 가능

- 가중치를 연결하지 않았을 때 83.6점
- 토큰 임베딩만 연결했을 때 84.3점
- 모든 가중치를 연결했을 때 84.4점

모든 가중치를 연결했을 때 가장 좋은 성능을 보여주지만(D가 학습하지 못한 부분까지 G가 학습해주기 때문) G와 D의 크기가 같아야 한다는 제약조건이 있다.

-> 하지만 G와 D가 같다면 BERT 연산의 두배가 된다.

Smaller Generators

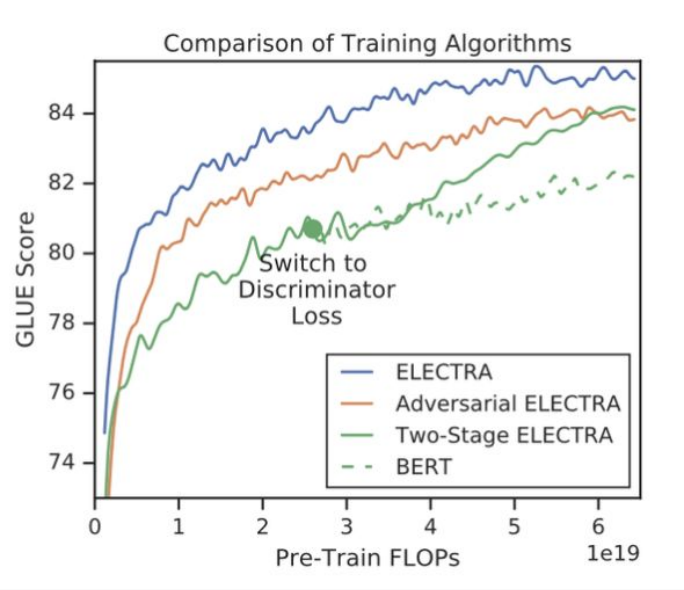


다른 하이퍼파라미터는 일정하게 유지하면서 계층 크기를 줄임.

작은 generator모델이 discriminator크기의 1/4~1/2 크기를사용할 때 가장 잘 작동한다는 것을 발견

-> 저자는 강력한 생성자는 판별자에게 너무 어려운 작업을 제기하여 효과적인 학습을 방해할 수 있다고 추측.

Training Algorithms



- Joint Training: generator와 discriminator를 **동시에 학습**하는 방법
- Two-stage training: **generator를 먼저 학습**하고 discriminator를 generator의 가중치로 초기화 한 후 discriminator를 학습시키는 방법
- Adversarial training: GAN처럼 **adversarial training**을 모사해서 학습시키는 방법

-> 생성자의 정확도가 많이 좋아지지 않기 때문에 Adversarial training 는 좋은 성능이 나오지 않는다.

-> Joint Training이 가장 좋은 성능, generator가 점점 더 어려운 작업을 학습할 수 있기 때문.

Small Models

연구의 핵심 중 하나는
pre-training의 효율성 향상

-> 하나의 GPU로도 빠르게
학습할 수 있는 모델 성능 실험

Model	Train / Infer FLOPs	Speedup	Params	Train Time + Hardware	GLUE
ELMo	3.3e18 / 2.6e10	19x / 1.2x	96M	14d on 3 GTX 1080 GPUs	71.2
GPT	4.0e19 / 3.0e10	1.6x / 0.97x	117M	25d on 8 P6000 GPUs	78.8
BERT-Small	1.4e18 / 3.7e9	45x / 8x	14M	4d on 1 V100 GPU	75.1
BERT-Base	6.4e19 / 2.9e10	1x / 1x	110M	4d on 16 TPUv3s	82.2
ELECTRA-Small	1.4e18 / 3.7e9	45x / 8x	14M	4d on 1 V100 GPU	79.9
50% trained	7.1e17 / 3.7e9	90x / 8x	14M	2d on 1 V100 GPU	79.0
25% trained	3.6e17 / 3.7e9	181x / 8x	14M	1d on 1 V100 GPU	77.7
12.5% trained	1.8e17 / 3.7e9	361x / 8x	14M	12h on 1 V100 GPU	76.0
6.25% trained	8.9e16 / 3.7e9	722x / 8x	14M	6h on 1 V100 GPU	74.1
ELECTRA-Base	6.4e19 / 2.9e10	1x / 1x	110M	4d on 16 TPUv3s	85.1

ELECTRA-Small은 BERT-Small보다 5점 높은 GLUE 점수를 얻었으며, 더 큰 GPT 모델(117M 인것으로 보아 GPT-2로 추측) 보다도 좋은 성능 달성

-> 성능과 학습 속도에 대한 향상을 강조

Large Models

Model	Train FLOPs	Params	CoLA	SST	MRPC	STS	QQP	MNLI	QNLI	RTE	Avg.
BERT	1.9e20 (0.27x)	335M	60.6	93.2	88.0	90.0	91.3	86.6	92.3	70.4	84.0
RoBERTa-100K	6.4e20 (0.90x)	356M	66.1	95.6	**91.4**	92.2	92.0	89.3	94.0	82.7	87.9
RoBERTa-500K	3.2e21 (4.5x)	356M	68.0	96.4	90.9	92.1	92.2	90.2	94.7	86.6	88.9
XLNet	3.9e21 (5.4x)	360M	69.0	**97.0**	90.8	92.2	92.3	90.8	94.9	85.9	89.1
BERT (ours)	7.1e20 (1x)	335M	67.0	95.9	89.1	91.2	91.5	89.6	93.5	79.5	87.2
ELECTRA-400K	7.1e20 (1x)	335M	**69.3**	96.0	90.6	92.1	**92.4**	90.5	94.5	86.8	89.0
ELECTRA-1.75M	3.1e21 (4.4x)	335M	69.1	96.9	90.8	**92.6**	**92.4**	**90.9**	**95.0**	**88.0**	**89.5**

ELECTRA는 RoBERTa 및 XLNet보다 훨씬 적은 FLOPs(1/4 미만)으로도 비슷한 성능을 달성

-> 같은 FLOPs 당 효율성을 입증

-> ELECTRA-1.75M은 가장 좋은 성능

Efficiency Analysis(1)

ELECTRA 15%

ELECTRA 모델과 동일하지만, discriminator 손실은 마스킹된 15%의 토큰에 대해서만 계산

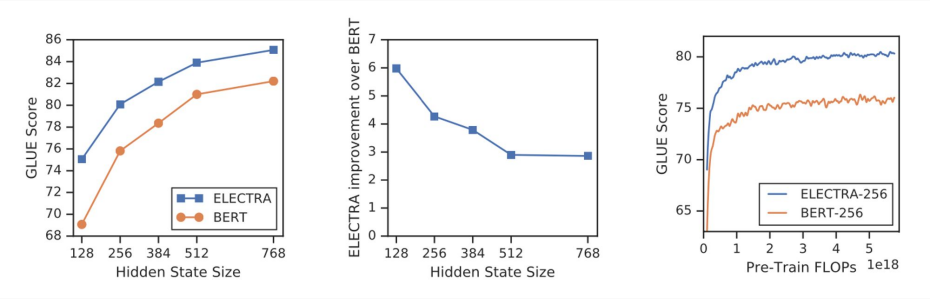
Replace MLM

discriminator를 MLM으로 학습. 마스킹된 토큰을 [MASK] 토큰 대신 생성자 모델에서 샘플링된 토큰으로 대체

All-Tokens MLM

Replace MLM과 유사하게 마스킹된 토큰을 생성자 샘플로 대체합니다. 하지만, 마스킹된 토큰뿐만 아니라 모든 입력 토큰에 대한 정체성을 예측합니다. (이 모델은 BERT와 ELECTRA를 결합한 형태)

Efficiency Analysis(2)



Model	ELECTRA	All-Tokens MLM	Replace MLM	ELECTRA 15%	BERT
GLUE score	85.0	84.3	82.4	82.4	82.2

결론적으로는 ELECTRA는 ELECTRA 15%와 Replace MLM보다 많은 성능 개선이 있는 것으로 보아 이 두 문제를 해결하여 성능이 상당히 좋아진 것을 확인할 수 있다. ELECTRA는 특히 작은 모델에서 성능 향상을 보여줬다.
(매개변수 효율성이 높다, parameter-efficient)

Conclusion

RTD가 MLM보다 더 효율적으로 학습할 수 있고 성능도 더 좋다.

저자는 향후 NLP pre-train 연구에서는 성능뿐만 아니라 효율성도 고려해야 하며,
계산량 및 매개변수 수를 평가 지표와 함께 보고해야 한다고 주장

How to use ELECTRA

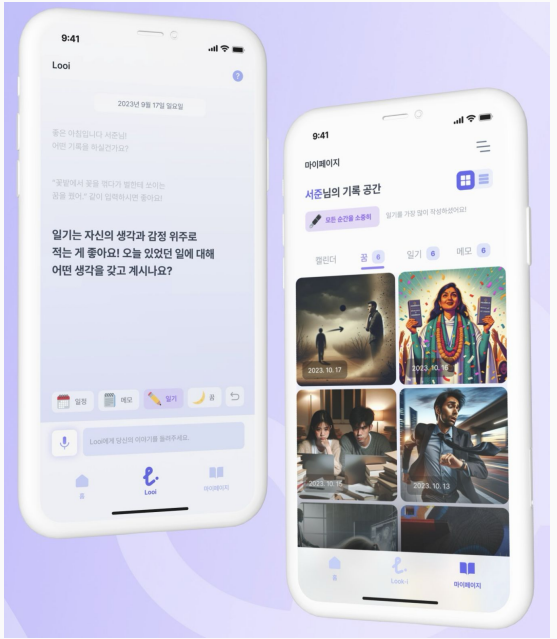
자체 텍스트 분류 모델 개발

모든 작업은 텍스트 분류 모델을 지나가기 때문에
가장 많은 토큰을 사용한다.

단순한 분류작업이기 때문에 트랜스포머의 인코더 계열
오픈 소스 모델 fine-tuning 해서 사용.

기대 효과

원가 절감 및 분류 정확성 향상



How to use ELECTRA

ELECTRA는 기본적으로 downstream 작업에 적절하지 않다. (효율적으로 pre-training)

-> 효율적으로 학습된 discriminator을 분리하여 fine-tuning하는 과정이 필요함

```
from transformers import ElectraForSequenceClassification, ElectraConfig

model_name = "beomi/KcELECTRA-base-v2022"
label_map = {'꿈': 0, '일기': 1, '일정': 2, '메모': 3, '오류': 4}

config = ElectraConfig.from_pretrained(model_name)
config.hidden_dropout_prob = 0.2 # dropout 설정
config.num_labels=len(label_map) # 라벨 개수 설정
model = ElectraForSequenceClassification.from_pretrained(
  model_name,
  config=config
).to(device)
```

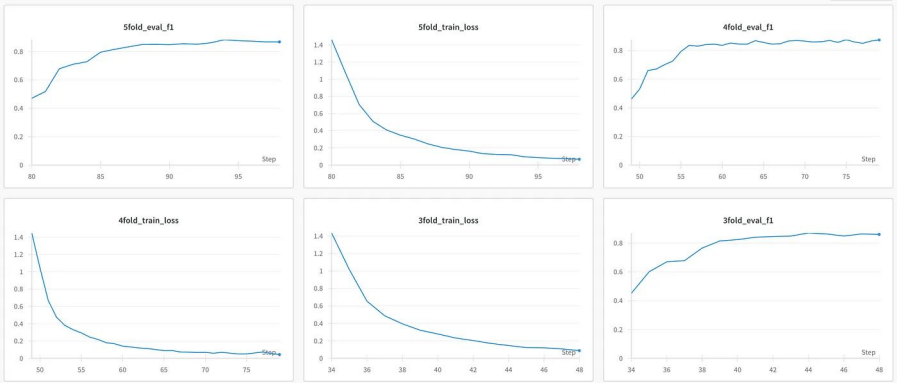
[CLS] 토큰으로 분류작업 할 수 있도록 라벨 설정

How to use ELECTRA

```
Thu Jan 25 15:18:14 2024
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 545.23.06              Driver Version: 545.23.06    CUDA Version: 12.3     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  NVIDIA GeForce RTX 4090        Off | 00000000:01:00.0 Off |                  Off |
| 43%   73C    P2             414W / 450W |  17902MiB / 24564MiB |     99%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|    0   N/A  N/A      1524      G   /usr/lib/xorg/Xorg                            72MiB |
|    0   N/A  N/A      1838      G   /usr/bin/gnome-shell                          61MiB |
|    0   N/A  N/A    329101      C   /usr/bin/python                            17748MiB |
+---------------------------------------------------------------------------------------+
```

배치사이즈 32기준, RAM 17GB 정도 사용



잘 학습되는 것을 확인할 수 있다.

How to use ELECTRA

```
Test dataset: 536
F1 Score for fold 1: 0.9579168072253672
F1 Score for fold 2: 0.9614135559040007
F1 Score for fold 3: 0.9506633558716269
F1 Score for fold 4: 0.94866333629038
F1 Score for fold 5: 0.9420034775694255
Average F1 Score: 0.9521321065721601
```

train 데이터의 크기: 2142
test 데이터의 크기: 536

코드 링크

기존 작업(gpt-4-turbo)
가격 : 16.47원 (2023.01.25 기준)
지연시간 : 1.32초

기존 f1-score: 0.673
개선된 f1-score: 0.952

-> 약 1.4배 성능 향상 (단, 기존에는 라벨이 4개)