ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators

120220225 조유빈

1. Introduction

   A. 기존 denoising autoencoder 방법론의 문제점

      i. 입력 시퀀스의 토큰 중 약 15% 정도를 마스킹하여 학습함으로써 전체 토큰 중 15%에 대해서만 loss 발생

         ✓ 즉, 하나의 example에 대해서 15%만 학습

         ✓ 학습하는데 비용이 많이 들어 감

      ii. 학습 시에는 [MASK] 토큰을 사용하여 예측하지만 실제 inference에서는 [MASK] 토큰 사용 안 함

   B. Proposed method

      i. Replaced Token Detection (RTD)

         ✓ Generator를 이용해 실제 입력 토큰들 중 일부를 가짜 토큰으로 바꾸고, 각 토큰이 실제 입력에 있는 진짜(original) 토큰인지 generator가 생성한 가짜(replaced) 토큰인지 discriminator가 구분

         ✓ 입력된 모든 토큰에 대해 학습하기 때문에 효율적이면서도 효과적

2. Method

   Generator와 discriminator 두 네트워크 모두 transformer encoder 구조

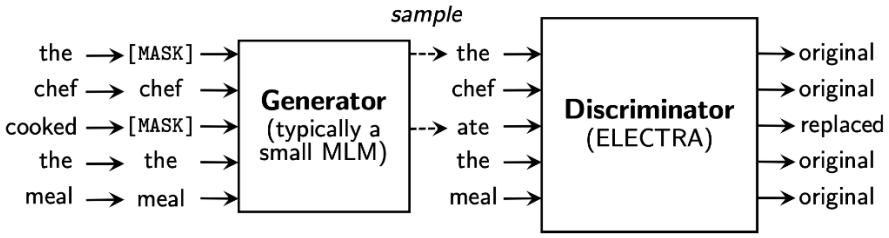

Figure 2: An overview of replaced token detection. The generator can be any model that produces an output distribution over tokens, but we usually use a small masked language model that is trained jointly with the discriminator. Although the models are structured like in a GAN, we train the generator with maximum likelihood rather than adversarially due to the difficulty of applying GANs to text. After pre-training, we throw out the generator and only fine-tune the discriminator (the ELECTRA model) on downstream tasks.

A. Generator

i. BERT의 Masked Language Modeling (MLM)과 동일

ii. 입력 $\boldsymbol{x} = [x_1, x_2, \cdots, x_n]$ 에 대하여 마스킹할 위치의 집합 $\boldsymbol{m} = [m_1, m_2, \cdots, m_k]$을 결정한 후 해당 위치의 입력 토큰을 [MASK]로 치환

```
# Mask the input
unmasked_inputs = pretrain_data.features_to_inputs(features)
masked_inputs = pretrain_helpers.mask(
    config, unmasked_inputs, config.mask_prob)
```

✓ 모든 마스킹 위치는 1과 n사이의 정수로 아래와 같이 표현

$$m_i \sim \text{unif}\{1, n\} \text{ for } i = 1 \text{ to } k \qquad \boldsymbol{x}^{\text{masked}} = \text{REPLACE}(\boldsymbol{x}, \boldsymbol{m}, \texttt{[MASK]})$$

✓ 마스팅할 개수 k는 보통 전체 토큰의 15%인 $0.15n$을 사용

iii. 마스킹 된 입력 $\boldsymbol{x}^{\text{masked}}$에 대해서 generator는 원래 토큰이 무엇인지 예측

✓ t번째 토큰에 대한 예측은 아래 식으로 표현됨

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

iv. Loss function

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

```
  def _get_masked_lm_output(self, inputs: pretrain_data.Inputs, model):
    """Masked language modeling softmax layer."""
    with tf.variable_scope("generator_predictions"):
      if self._config.uniform_generator:
        logits = tf.zeros(self._bert_config.vocab_size)
        logits_tiled = tf.zeros(
            modeling.get_shape_list(inputs.masked_lm_ids) +
            [self._bert_config.vocab_size])
        logits_tiled += tf.reshape(logits, [1, 1, self._bert_config.vocab_size])
        logits = logits_tiled
      else:
        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)
```

v. Code

```
# Generator
embedding_size = (
    self._bert_config.hidden_size if config.embedding_size is None else
    config.embedding_size)
cloze_output = None
if config.uniform_generator:
  # simple generator sampling fakes uniformly at random
  mlm_output = self._get_masked_lm_output(masked_inputs, None)
elif ((config.electra_objective or config.electric_objective)
      and config.untied_generator):
  generator_config = get_generator_config(config, self._bert_config)
  if config.two_tower_generator:
    # two-tower cloze model generator used for electric
    generator = TwoTowerClozeTransformer(
        config, generator_config, unmasked_inputs, is_training,
        embedding_size)
    cloze_output = self._get_cloze_outputs(unmasked_inputs, generator)
    mlm_output = get_softmax_output(
        pretrain_helpers.gather_positions(
            cloze_output.logits, masked_inputs.masked_lm_positions),
        masked_inputs.masked_lm_ids, masked_inputs.masked_lm_weights,
        self._bert_config.vocab_size)
  else:
    # small masked language model generator
    generator = build_transformer(
        config, masked_inputs, is_training, generator_config,
        embedding_size=(None if config.untied_generator_embeddings
                        else embedding_size),
        untied_embeddings=config.untied_generator_embeddings,
        scope="generator")
    mlm_output = self._get_masked_lm_output(masked_inputs, generator)
else:
  # full-sized masked language model generator if using BERT objective or if
  # the generator and discriminator have tied weights
  generator = build_transformer(
      config, masked_inputs, is_training, self._bert_config,
      embedding_size=embedding_size)
  mlm_output = self._get_masked_lm_output(masked_inputs, generator)
```

B. Discriminator

i. 입력 토큰 시퀀스에 대해서 각 토큰이 original인지 replaced인지 구분

ii. Figure2와 같이 Generator에서 마스킹 위치의 집합 m에 해당하는 위치의 토큰을 [MASK]가 아닌 generator의 softmax 분포 $p_G(x_t|\boldsymbol{x})$에 대해 샘플링한 토큰으로 치환(corrupt)

```
 fake_data = self._get_fake_data(masked_inputs, mlm_output.logits)
 self.mlm_output = mlm_output
def _get_fake_data(self, inputs, mlm_logits):
  """Sample from the generator to create corrupted input."""
  inputs = pretrain_helpers.unmask(inputs)
  disallow = tf.one_hot(
      inputs.masked_lm_ids, depth=self._bert_config.vocab_size,
      dtype=tf.float32) if self._config.disallow_correct else None
  sampled_tokens = tf.stop_gradient(pretrain_helpers.sample_from_softmax(
      mlm_logits / self._config.temperature, disallow=disallow))
  sampled_tokids = tf.argmax(sampled_tokens, -1, output_type=tf.int32)
  updated_input_ids, masked = pretrain_helpers.scatter_update(
      inputs.input_ids, sampled_tokids, inputs.masked_lm_positions)
  if self._config.electric_objective:
    labels = masked
  else:
    labels = masked * (1 - tf.cast(
        tf.equal(updated_input_ids, inputs.input_ids), tf.int32))
  updated_inputs = pretrain_data.get_updated_inputs(
      inputs, input_ids=updated_input_ids)
  FakedData = collections.namedtuple("FakedData", [
      "inputs", "is_fake_tokens", "sampled_tokens"])
  return FakedData(inputs=updated_inputs, is_fake_tokens=labels,
                   sampled_tokens=sampled_tokens)
```

- ✓ Original sequence: [the, chef, cooked, the, meal]
- ✓ Input sequence for generator: [[MASK], chef, [MASK], the, meal]
- ✓ Input sequence for discriminator: [the, chef, ate, the, meal]
- ✓ 수식으로 표현하면 아래와 같음

$$\hat{x}_i \sim p_G(x_i|\boldsymbol{x}^{\text{masked}}) \text{ for } i \in \boldsymbol{m} \qquad \boldsymbol{x}^{\text{corrupt}} = \text{REPLACE}(\boldsymbol{x}, \boldsymbol{m}, \hat{\boldsymbol{x}})$$

iii. 치환된 입력 $\boldsymbol{x}^{\text{corrupt}}$에 대해서 discriminator는 각 토큰이 원래 입력과 동일한지 치환된 것인지 예측 (binary classification)

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

- ✓ Original: 이 위치에 해당하는 토큰은 원본 문장의 토큰

✓ Replaced: 이 위치에 해당하는 토큰은 generator에 의해 변형된 것

iv. Loss function

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

```
def _get_discriminator_output(
    self, inputs, discriminator, labels, cloze_output=None):
  """Discriminator binary classifier."""
  with tf.variable_scope("discriminator_predictions"):
    hidden = tf.layers.dense(
        discriminator.get_sequence_output(),
        units=self._bert_config.hidden_size,
        activation=modeling.get_activation(self._bert_config.hidden_act),
        kernel_initializer=modeling.create_initializer(
            self._bert_config.initializer_range))
    logits = tf.squeeze(tf.layers.dense(hidden, units=1), -1)
    if self._config.electric_objective:
      log_q = tf.reduce_sum(
          tf.nn.log_softmax(cloze_output.logits) * tf.one_hot(
              inputs.input_ids, depth=self._bert_config.vocab_size,
              dtype=tf.float32), -1)
      log_q = tf.stop_gradient(log_q)
      logits += log_q
      logits += tf.log(self._config.mask_prob / (1 - self._config.mask_prob))

    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))
    probs = tf.nn.sigmoid(logits)
    preds = tf.cast(tf.round((tf.sign(logits) + 1) / 2), tf.int32)
```

v. Code

```
# Discriminator
disc_output = None
if config.electra_objective or config.electric_objective:
  discriminator = build_transformer(
      config, fake_data.inputs, is_training, self._bert_config,
      reuse=not config.untied_generator, embedding_size=embedding_size)
  disc_output = self._get_discriminator_output(
      fake_data.inputs, discriminator, fake_data.is_fake_tokens,
      cloze_output)
  self.total_loss += config.disc_weight * disc_output.loss
```

3. GAN과의 차이점

   A. Generator가 원래 토큰과 동일한 토큰을 생성했을 때, GAN은 negative(fake) sample로 판단하지만 ELECTRA는 positive sample로 판단함

   B. Generator가 discriminator를 속이기 위해 adversarial하게 학습하는 것이 아니라 maximum likelihood로 학습함

      i. Generator에서 샘플링하는 과정으로 인해 역전파가 불가능하여 adversarial하게 generator를 학습시키기 어려움

      ii. 강화학습으로 이를 구현해보았지만, maximum likelihood로 학습시키는 것보다 성능이 좋지 않았음

   C. Generator의 입력으로 노이즈 벡터를 사용하지 않음

   D. Pre-training을 마친 후에는 generator는 사용하지 않고 discriminator만 사용하여 downstream task로 finetuning 진행