BERT

Abstract

BERT : Bidirectional Encoder Representations from Transformers

Bert는 unlabled text로 양방향 pre-trained되었기에 작업별 구조 수정을 크게 할 필요 없이 하나의 출력 레이어만 추가만 해도 파인튜닝 가능하다.

Introduction

GPT1 ELMo는 pre-train과 finetuning의 조합 : 왼쪽에서 오른쪽으로의 아키텍처만 사용하고, 이전과 현재 token에만 attention할 수 있다.

BUT 몇몇 task들은 양방향의 맥락을 통합하는 것이 중요

(1)마스크된 언어 모델"(MLM) 사전 학습[왼쪽과 오른쪽 맥락을 융합하는 표현을 가능케 함]

(2)"다음 문장 예측(Next Sentence Prediction)" [텍스트 쌍 표현을 공동으로 사전 학습]

BERT

Model Architecture

우리 프레임워크의 두 단계: 사전 훈련과 미세 조정.

pre-training 동안, 모델은 다양한 pre-training 작업에서 레이블이 없는 데이터로 훈련됨. 미세 조정을 위해, BERT 모델은 먼저 pre-trained parameters로 초기화되고, 모든 parameters는 downstream task에서 레이블이 지정된 데이터를 사용하여 finetuning됨. 각 downstream task은 동일한 pretrained parameters로 초기화되지만, 각각 별도의 fine-tuning된 모델을 가지고 있음.

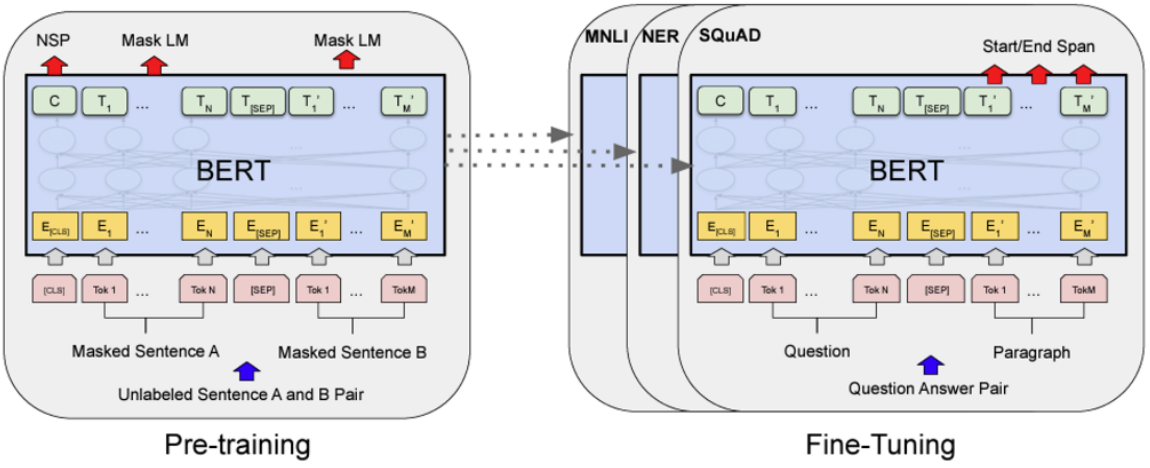


Figure 1: Overall pre-training and fine-tuning procedures for BERT. Apart from output layers, the same architectures are used in both pre-training and fine-tuning. The same pre-trained model parameters are used to initialize models for different down-stream tasks. During fine-tuning, all parameters are fine-tuned. `[CLS]` is a special symbol added in front of every input example, and `[SEP]` is a special separator token (e.g. separating questions/answers).

사전학습된 BERT가 QA문제를 해결하기 위해 초기화됨. [cls]는 input token이전에 추가되고, [sep]은 2개의 문장을 분리하기 위한 토큰임.

BERT는 GPT와의 비교를 위해 2가지 모델 사이즈를 제공하였다. BERT_BASE의 경우는 12개의 laer와 12개의 head, hid_dim은 768로 설정하였음.

BERT가 다양한 down stream task를 잘 타파하기 위해서 input representation이 single sentence랑 a pair of sentences를 하나의 token seqence로 표현하였다. “sequence”의 의미는 BERT의 INPUT에 사용되는 token으로 singel sentence일수도 2개의 sentences가 packed된 것일수도 있음. [cls]토큰은 마지막 hidden layer에서 어떤 task인지를 나타내는 토큰이 된다.

또 만약 2문장에 대한 한 sequence token일 경우에는 두 문장 사이에 [sep] token을 추가하고, 어떤 문장인지를 나타내는 학습된 embedding을 추가한다.

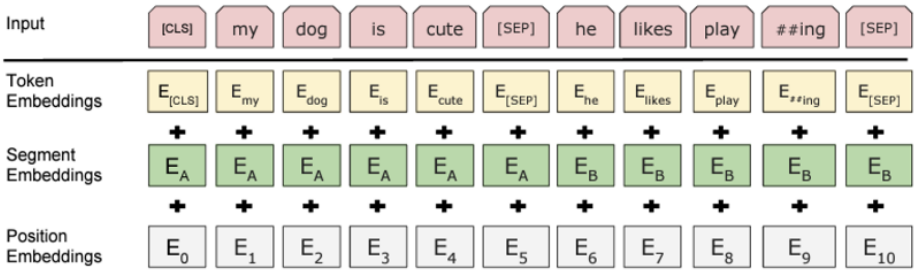


Figure 2: BERT input representation. The input embeddings are the sum of the token embeddings, the segmentation embeddings and the position embeddings.

위 그림에서처럼 기존의 Tranformer보다 segment embedding이 추가된 것을 알 수 있다.

Pre-training BERT

Task #1: Masked LM(MLM)

a deep bidirectional representation을 학습하기 위해서 input token들의 일부를 masking하고, 이를 예측하도록 하였다.

Masking Rates			Dev Set Results		
MASK	SAME	RND	MNLI Fine-tune	NER Fine-tune	NER Feature-based
80%	10%	10%	84.2	95.4	94.9
100%	0%	0%	84.3	94.9	94.0
80%	0%	20%	84.1	95.2	94.6
80%	20%	0%	84.4	95.2	94.7
0%	20%	80%	83.7	94.8	94.6
0%	0%	100%	83.6	94.9	94.6

15% of the token positions at random for prediction을 80%는 masking, 10%는 동일, 10%는 random화함.

Task #2: Next Sentence Prediction (NSP)

QA나 NLI는 두가지 문장 사이의 관계를 추론하는 task이다.

A와 A 관련있는 문장 B를 두고 관련있는(ISNEXT) 50%, 관련없는(NOTNEXT) 50%로 사전학습을 하여 위 task들에 높은 성능 향상을 보여주었다.

Pre-training data

BooksCorpus(8억 단어)와 영어 위키피디아(25억 단어)를 사용

위키피디아에서는 텍스트 문단만 추출하고 리스트, 표, 헤더는 무시

문서 수준의 코퍼스를 사용하는 것이 중요한데, 이는 긴 연속적인 시퀀스를 추출하기 위함

Fine-tuning BERT

텍스트 쌍을 독립적으로 인코딩한 후 bidirectional cross attention를 적용하는 것이 일반적.

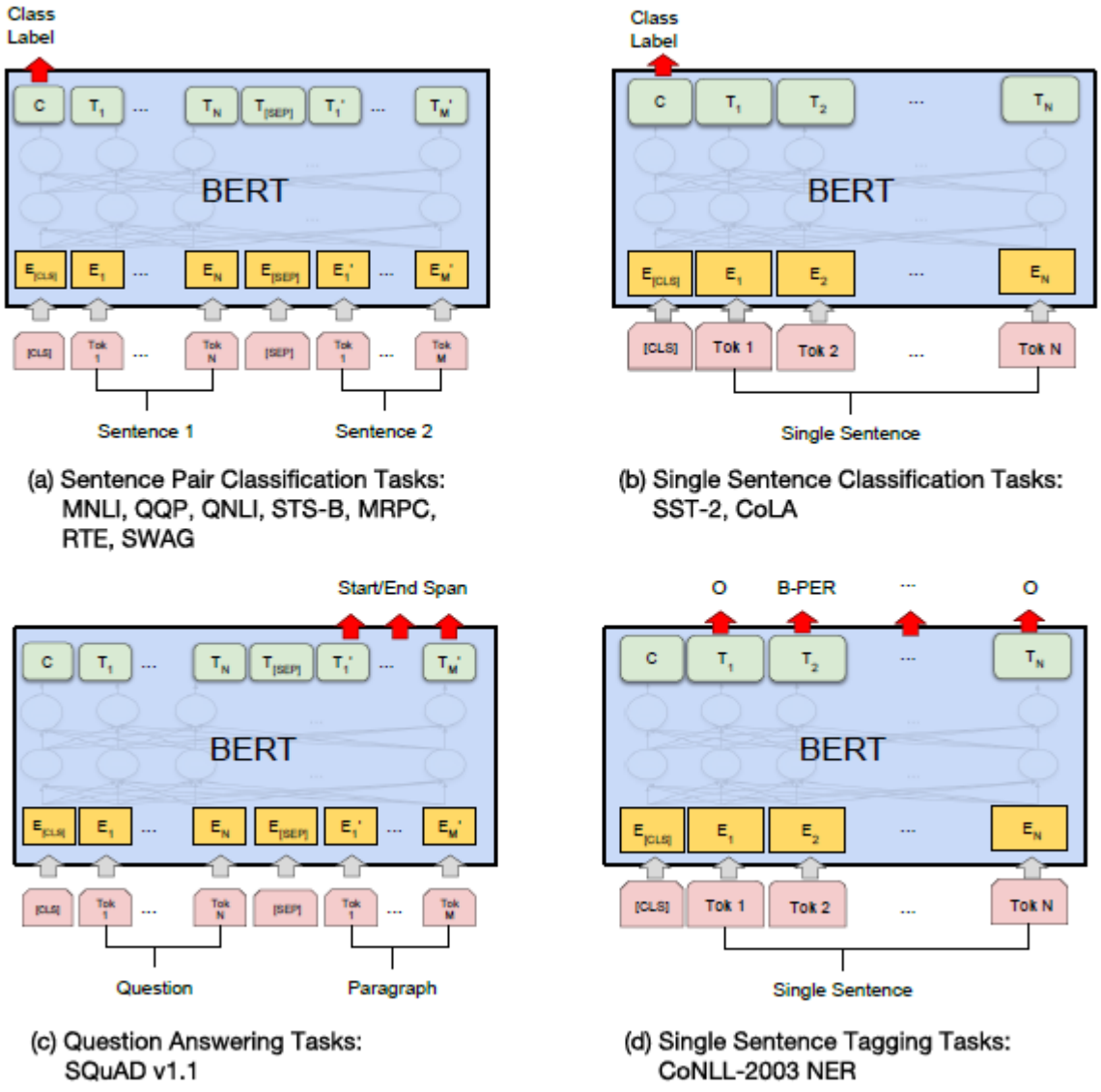
그러나 BERT는 self-attention mechanism to unify these two stages.

각각의 task에 대해 단순히 input과 output을 BERT에 plug in하고, 모든 parameter를 end-to-end로 fine-tuning한다.

사전 학습에 사용된 A와 B는 fine tuning시 task별로 paraphrasing을 위한 문장쌍, 가설과 전제의 쌍, QA의 쌍, 문장분류의 쌍으로서 작용한다. 이 때. [cls]는 어떤 task인지에 대한 설명을 제공해준다.

Experiments

we present BERT fine-tuning results on 11 NLP tasks



(a) Sentence Pair Classification Tasks: MNLI, QQP, QNLI, STS-B, MRPC, RTE, SWAG

(b) Single Sentence Classification Tasks: SST-2, CoLA

(c) Question Answering Tasks: SQuAD v1.1

(d) Single Sentence Tagging Tasks: CoNLL-2003 NER

GLUE

The General Language Understanding Evaluation (GLUE) benchmark

- **MNLI** Multi-Genre Natural Language Inference is a large-scale, crowdsourced entailment classification task

- **QQP** Quora Question Pairs is a binary classification task where the goal is to determine if two questions asked on Quora are semantically equivalent
- **QNLI** Question Natural Language Inference is a version of the Stanford Question Answering Dataset which has been converted to a binary classification task
- **SST-2** The Stanford Sentiment Treebank, 사람들의 sentiment가 주석처리된 movie reviews로부터 추출된 binary single sentence classification
- **STS-B** The Semantic Textual Similarity Benchmark로, news headlines and other sources로 부터 얻어진 문장쌍이다.
- **RTE** Recognizing Textual Entailment binary entailment task similar to MNLI, but with much less training data
- **WNLI** Winograd NLI is a small natural language inference dataset

SQuAD

The Stanford Question Answering Dataset

자연어 처리 분양에서 사용되는 질의응답 데이터셋

- SQuAD v1.1

  약 10만개의 질문으로 구성된 데이터셋.

  위키피디아의 500개 이상의 기사에서 추출된 문단을 기반으로 만듦.

  각 질문은 답을 포함하고 있는 특정문단에 대응되고, 답은 그 문단 내에서 답변이 가능한 segment로 존재함.

- SQuAD v2.0

  SQuAD v2.0은 SQuAD v1.1을 기반으로 하며, 약 5만 개의 새로운 질문을 추가하여 총 15만 개의 질문을 포함

  이전 버전은 모든 질문에 대한 답이 존재하는 데이터셋이였다면 해당 데이터셋에서는 답이 문단 내에 존재하지 않는 “불가능한” 질문이 포함되어있다.

SWAG

The Situations With Adversarial Generations (SWAG) dataset

상식적 추론을 평가하기 위한 113k sentence-pair completion examples이다.

주어진 문장이 있을 때, 네 가지 선택지 중 가장 그럴듯한 이어지는 문장을 선택하는 것이 작업의 목표

SWAG 데이터셋에 대한 미세 조정을 할 때, 우리는 주어진 문장(문장 A)과 가능한 이어지는 문장(문장 B)을 연결한 네 개의 입력 시퀀스를 구성하고, 이 작업에서 [cls]토큰이 softmax를 거친 출력을 나타내고 이 점수로 가장 그럴듯한 문장을 반환하게 된다.

Ablation Studies

pre-training

Tasks	Dev Set MNLI-m (Acc)	QNLI (Acc)	MRPC (Acc)	SST-2 (Acc)	SQuAD (F1)
$BERT_{BASE}$	84.4	88.4	86.7	92.7	88.5
No NSP	83.9	84.9	86.5	92.6	87.9
LTR & No NSP	82.1	84.3	77.5	92.1	77.8
+ BiLSTM	82.1	84.1	75.7	91.6	84.9

LTR(Left To Right)

No NSP : NSP없이 pre-train된 것을 의미.

BiLSTM의 경우는 LTR과 RTL을 2번 실행한 후 , concat을 하므로 single bidirectional model보다 2배 비싸다.

Effect of Model size

train BERT with a differing number of layers, hidden units, and attention heads

Hyperparams				Dev Set Accuracy		
#L	#H	#A	LM (ppl)	MNLI-m	MRPC	SST-2
3	768	12	5.84	77.9	79.8	88.4
6	768	3	5.24	80.6	82.2	90.7
6	768	12	4.68	81.9	84.8	91.3
12	768	12	3.99	84.4	86.7	92.9
12	1024	16	3.54	85.7	86.9	93.3
24	1024	16	3.23	86.6	87.8	93.7

larger models lead to a strict accuracy improvement.

increasing the model size will lead to continual improvements on large-scale tasks such as machine translation and language modeling,

Feature-based Approach with BERT

System	Dev F1	Test F1
ELMo (Peters et al., 2018a)	95.7	92.2
CVT (Clark et al., 2018)	-	92.6
CSE (Akbik et al., 2018)	-	**93.1**
Fine-tuning approach		
$\text{BERT}_{\text{LARGE}}$	96.6	92.8
$\text{BERT}_{\text{BASE}}$	96.4	92.4
Feature-based approach ($\text{BERT}_{\text{BASE}}$)		
Embeddings	91.0	-
Second-to-Last Hidden	95.6	-
Last Hidden	94.9	-
Weighted Sum Last Four Hidden	95.9	-
Concat Last Four Hidden	96.1	-
Weighted Sum All 12 Layers	95.5	-

다음 표는 NER task의 성능을 보여준다.

- 사전 훈련된 모델에 단순 분류 계층을 추가하고, 모든 매개변수를 다운스트림 작업에 맞게 함께 파인튜닝
- 사전 훈련된 모델에서 고정된 특징을 추출, by extracting the activations from one or more layers without fine-tuning any parameters of BERT.

BERT는 finetuning 방식과 feature based 방식 모두에서 효과적인 것으로 나타났으며, 특히 'Concat Last Four Hidden' 방식이 특징 기반 접근 방식에서 매우 경쟁력 있는 성능을 보여주었습니다. 이는 BERT를 다양한 NLP 작업에 유연하게 적용할 수 있음을 시사한다.

CODE REVIEW

```
import torch.nn as nn
import torch.nn.functional as F
import torch

import math


class Attention(nn.Module):
    """
    Compute 'Scaled Dot Product Attention
    """

    def forward(self, query, key, value, mask=None, dropout=N
        scores = torch.matmul(query, key.transpose(-2, -1)) \
                 / math.sqrt(query.size(-1))
```

```
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)

        p_attn = F.softmax(scores, dim=-1)

        if dropout is not None:
            p_attn = dropout(p_attn)

        return torch.matmul(p_attn, value), p_attn
```

```
import torch.nn as nn
from .single import Attention


class MultiHeadedAttention(nn.Module):
    """
    Take in model size and number of heads.
    """

    def __init__(self, h, d_model, dropout=0.1):
        super().__init__()
        assert d_model % h == 0

        # We assume d_v always equals d_k
        self.d_k = d_model // h
        self.h = h

        self.linear_layers = nn.ModuleList([nn.Linear(d_model
        self.output_linear = nn.Linear(d_model, d_model)
        self.attention = Attention()

        self.dropout = nn.Dropout(p=dropout)

    def forward(self, query, key, value, mask=None):
        batch_size = query.size(0)

        # 1) Do all the linear projections in batch from d_mo
```

```
        query, key, value = [l(x).view(batch_size, -1, self.h
                             for l, x in zip(self.linear_laye

        # 2) Apply attention on all the projected vectors in
        x, attn = self.attention(query, key, value, mask=mask

        # 3) "Concat" using a view and apply a final linear.
        x = x.transpose(1, 2).contiguous().view(batch_size, -

        return self.output_linear(x)
```

```
import torch.nn as nn
from .token import TokenEmbedding
from .position import PositionalEmbedding
from .segment import SegmentEmbedding


class BERTEmbedding(nn.Module):
    """
    BERT Embedding which is consisted with under features
        1. TokenEmbedding : normal embedding matrix
        2. PositionalEmbedding : adding positional informatio
        2. SegmentEmbedding : adding sentence segment info, (

        sum of all these features are output of BERTEmbedding
    """

    def __init__(self, vocab_size, embed_size, dropout=0.1):
        """
        :param vocab_size: total vocab size
        :param embed_size: embedding size of token embedding
        :param dropout: dropout rate
        """
        super().__init__()
        self.token = TokenEmbedding(vocab_size=vocab_size, em
        self.position = PositionalEmbedding(d_model=self.toke
        self.segment = SegmentEmbedding(embed_size=self.token
```

```
        self.dropout = nn.Dropout(p=dropout)
        self.embed_size = embed_size

    def forward(self, sequence, segment_label):
        x = self.token(sequence) + self.position(sequence) +
        return self.dropout(x)
```

```
import torch.nn as nn
import torch
import math

class TokenEmbedding(nn.Embedding):
    def __init__(self, vocab_size, embed_size=512):
        super().__init__(vocab_size, embed_size, padding_idx=

class SegmentEmbedding(nn.Embedding):
    def __init__(self, embed_size=512):
        super().__init__(3, embed_size, padding_idx=0)

class PositionalEmbedding(nn.Module):

    def __init__(self, d_model, max_len=512):
        super().__init__()

        # Compute the positional encodings once in log space.
        pe = torch.zeros(max_len, d_model).float()
        pe.require_grad = False

        position = torch.arange(0, max_len).float().unsqueeze
        div_term = (torch.arange(0, d_model, 2).float() * -(m

        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)

        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)
```

```
    def forward(self, x):
        return self.pe[:, :x.size(1)]
```

```
import torch.nn as nn

from .transformer import TransformerBlock
from .embedding import BERTEmbedding


class BERT(nn.Module):
    """
    BERT model : Bidirectional Encoder Representations from T
    """

    def __init__(self, vocab_size, hidden=768, n_layers=12, a
        """
        :param vocab_size: vocab_size of total words
        :param hidden: BERT model hidden size
        :param n_layers: numbers of Transformer blocks(layers
        :param attn_heads: number of attention heads
        :param dropout: dropout rate
        """

        super().__init__()
        self.hidden = hidden
        self.n_layers = n_layers
        self.attn_heads = attn_heads

        # paper noted they used 4*hidden_size for ff_network_
        self.feed_forward_hidden = hidden * 4

        # embedding for BERT, sum of positional, segment, tok
        self.embedding = BERTEmbedding(vocab_size=vocab_size,

        # multi-layers transformer blocks, deep network
        self.transformer_blocks = nn.ModuleList(
            [TransformerBlock(hidden, attn_heads, hidden * 4,
```

```
    def forward(self, x, segment_info):
        # attention masking for padded token
        # torch.ByteTensor([batch_size, 1, seq_len, seq_len)
        mask = (x > 0).unsqueeze(1).repeat(1, x.size(1), 1).u

        # embedding the indexed sequence to sequence of vecto
        x = self.embedding(x, segment_info)

        # running over multiple transformer blocks
        for transformer in self.transformer_blocks:
            x = transformer.forward(x, mask)

        return x
```

```
import torch
import torch.nn as nn
from torch.optim import Adam
from torch.utils.data import DataLoader

from ..model import BERTLM, BERT
from .optim_schedule import ScheduledOptim

import tqdm


class BERTTrainer:
    """
    BERTTrainer make the pretrained BERT model with two LM tr

        1. Masked Language Model : 3.3.1 Task #1: Masked LM
        2. Next Sentence prediction : 3.3.2 Task #2: Next Sen

    please check the details on README.md with simple example

    """

    def __init__(self, bert: BERT, vocab_size: int,
                 train_dataloader: DataLoader, test_dataloade
```

```
                 lr: float = 1e-4, betas=(0.9, 0.999), weight_
                 with_cuda: bool = True, cuda_devices=None, l
        """
        :param bert: BERT model which you want to train
        :param vocab_size: total word vocab size
        :param train_dataloader: train dataset data loader
        :param test_dataloader: test dataset data loader [can
        :param lr: learning rate of optimizer
        :param betas: Adam optimizer betas
        :param weight_decay: Adam optimizer weight decay para
        :param with_cuda: traning with cuda
        :param log_freq: logging frequency of the batch itera
        """

        # Setup cuda device for BERT training, argument -c, -
        cuda_condition = torch.cuda.is_available() and with_c
        self.device = torch.device("cuda:0" if cuda_condition

        # This BERT model will be saved every epoch
        self.bert = bert
        # Initialize the BERT Language Model, with BERT model
        self.model = BERTLM(bert, vocab_size).to(self.device)

        # Distributed GPU training if CUDA can detect more th
        if with_cuda and torch.cuda.device_count() > 1:
            print("Using %d GPUS for BERT" % torch.cuda.devic
            self.model = nn.DataParallel(self.model, device_i

        # Setting the train and test data loader
        self.train_data = train_dataloader
        self.test_data = test_dataloader

        # Setting the Adam optimizer with hyper-param
        self.optim = Adam(self.model.parameters(), lr=lr, bet
        self.optim_schedule = ScheduledOptim(self.optim, self

        # Using Negative Log Likelihood Loss function for pre
        self.criterion = nn.NLLLoss(ignore_index=0)
```

```
        self.log_freq = log_freq

        print("Total Parameters:", sum([p.nelement() for p in

    def train(self, epoch):
        self.iteration(epoch, self.train_data)

    def test(self, epoch):
        self.iteration(epoch, self.test_data, train=False)

    def iteration(self, epoch, data_loader, train=True):
        """
        loop over the data_loader for training or testing
        if on train status, backward operation is activated
        and also auto save the model every peoch

        :param epoch: current epoch index
        :param data_loader: torch.utils.data.DataLoader for i
        :param train: boolean value of is train or test
        :return: None
        """
        str_code = "train" if train else "test"

        # Setting the tqdm progress bar
        data_iter = tqdm.tqdm(enumerate(data_loader),
                              desc="EP_%s:%d" % (str_code, ep
                              total=len(data_loader),
                              bar_format="{l_bar}{r_bar}")

        avg_loss = 0.0
        total_correct = 0
        total_element = 0

        for i, data in data_iter:
            # 0. batch_data will be sent into the device(GPU
            data = {key: value.to(self.device) for key, value
```

```
            # 1. forward the next_sentence_prediction and mas
            next_sent_output, mask_lm_output = self.model.for

            # 2-1. NLL(negative log likelihood) loss of is_ne
            next_loss = self.criterion(next_sent_output, data

            # 2-2. NLLLoss of predicting masked token word
            mask_loss = self.criterion(mask_lm_output.transpo

            # 2-3. Adding next_loss and mask_loss : 3.4 Pre-t
            loss = next_loss + mask_loss

            # 3. backward and optimization only in train
            if train:
                self.optim_schedule.zero_grad()
                loss.backward()
                self.optim_schedule.step_and_update_lr()

            # next sentence prediction accuracy
            correct = next_sent_output.argmax(dim=-1).eq(data
            avg_loss += loss.item()
            total_correct += correct
            total_element += data["is_next"].nelement()

            post_fix = {
                "epoch": epoch,
                "iter": i,
                "avg_loss": avg_loss / (i + 1),
                "avg_acc": total_correct / total_element * 10
                "loss": loss.item()
            }

            if i % self.log_freq == 0:
                data_iter.write(str(post_fix))

        print("EP%d_%s, avg_loss=" % (epoch, str_code), avg_l
              total_correct * 100.0 / total_element)
```

```
    def save(self, epoch, file_path="output/bert_trained.mode
        """
        Saving the current BERT model on file_path

        :param epoch: current epoch number
        :param file_path: model output path which gonna be fi
        :return: final_output_path
        """
        output_path = file_path + ".ep%d" % epoch
        torch.save(self.bert.cpu(), output_path)
        self.bert.to(self.device)
        print("EP:%d Model Saved on:" % epoch, output_path)
        return output_path
```