Project Description

Deep Learning for Natural Language Processing

University of Göttingen

Teaching Staff:

Lecturer: Dr. Terry Ruas

Teaching Assistants: Finn Schmidt, Niklas Bauer, Ivan Pluzhnikov, Jan Philip Wahle, Jonas Lührs

Table of Contents

Note. This project was adapted from the DNLP course (Summer 2023) at the University of Göttingen, which was based in the default project at Stanford University's CS 224n Natural Language Processing with Deep Learning, by Gabriel Poesia, John Hewitt, Amelie Byun, John Cho, and their team. Originally, the project was developed as the "minbert" assignment developed at Carnegie Mellon University's CS11-711 Advanced NLP, created by Shuyan Zhou, Zhengbao Jiang, Ritam Dutt, Brendon Boldt, Aditya Veerubhotla, and Graham Neubig. Thank you all for the great resource.

1 Project Description

The goal of this project is for you first to implement some of the key aspects of the original BERT model, including multi-head self-attention and a Transformer layer. You will then use your completed BERT model on several tasks (e.g., sentiment analysis, question similarity, semantic similarity). In addition, you will also implement a BART model for paraphrase type generation and detection. Finally, in the latter half of this project, you will fine-tune and extend the models to solve the same downstream tasks. We describe several techniques commonly used to create more robust and semantically-rich sentence embeddings – most from recent research papers. We provide these suggestions to help you get started. They should all improve over the baseline if implemented correctly (and note that there is usually more than one way to implement something correctly).

In other words, the project will have two parts:

- **Part 01:** Baseline implementations of BERT (sentiment analysis, question similarity, semantic similarity) and BART (paraphrase type generation and detection)[1]. Each task must have <u>at least one</u> baseline.
- **Part 02:** Improvements for all baselines. Each task baseline must have <u>at least one</u> improvement.

Though you are not required to implement something original, the best projects will pursue some form of originality (and may become research papers in the future). Originality does not necessarily have to be a completely new approach – small but well-motivated changes to existing models are valuable, especially if followed by sound analysis. If you can show quantitatively and qualitatively that your small but original change improves a state-of-the-art model (and even better, explain what particular problem it solves and how), then you will have done exceptionally well. It will be up to you to figure out what to do. In many cases, there will not be one correct answer for doing something – it will take experimentation to determine which is best. We expect you to exercise the judgment and intuition you have gained from the class and self-study to build your models.

[1] You can also implement the BERT model for paraphrase type detection. Please read Section 7.2 for more details.

2 Basic Setup

For this project, you will need a machine with GPUs to train your models efficiently. For this, you have access to a server provided by the University of Göttingen (GWDG and NHR@Göttingen)[2]. The servers and GPUs offered are restricted and should not be used as a playground or debug environment for your code. We advise that you develop your code on your local machine, using PyTorch without GPUs, and move to your Server only once you have debugged your code and you are ready to train. We advise that you use GitHub to manage your codebase and sync it between the two machines (and between team members). When you work through this section for the first time, do so on your local machine. You will then repeat the process on the GWDG cluster. Once you are on an appropriate machine, clone the project GitHub repository with the following command.

```
git clone https://github.com/GippLab-DNLP-Team/dnlp-final-project
```

Alternatively, use the GitHub template feature to create your repository copy first. This repository contains the starter code and a minimalist implementation of the BERT model (minBERT) we will use. We also provide a small guide to the use of the BART model. We encourage you to fork our repository rather than simply download it so you can easily integrate any bug fixes we make into the code. You should periodically check whether there are any new fixes that you need to download. To do so, navigate to the `dnlp-final-project` directory and run the `git pull` command. If you use GitHub to manage your code, you must keep your repository private until the semester ends.

2.1 Code overview

The repository dnlp-final-project/ contains the following files:

- **base_bert.py**: A base BERT implementation that can load pre-trained weights against which you can test your implementation.
- **bert.py**: This file contains the BERT Model. Several sections of this implementation need to be completed. This includes the self-attention and the Transformer (BERT) layer.
- **config.py**: This is where the configuration class is defined. You won't need to modify this file in this assignment.
- **multitask_classifier.py**: A classifier pipeline where you will train your minBERT implementation to perform sentiment analysis, paraphrase detection, and semantic textual similarity tasks (and paraphrase type detection).
- **bart_detection.py:** A pipeline to train a BART model for the paraphrase types detection task. The main parts of this pipeline remain to be completed.

[2] Make sure to participate in the workshop offered by GWDG and NHR on 2024-05-30 during the exercise session.

- **bart_generation.py:** A pipeline to train a BART model for the paraphrase generation task. The main parts of this pipeline remain to be completed.
- **datasets.py**: A dataset handling script.
- **evaluation.py**: An evaluation handling script.
- **optimizer.py**: An implementation of the Adam Optimizer. The `step()` function of the Adam optimizer needs to be completed.
- **tokenizer.py**: This is where BertTokenizer is implemented. You won't need to modify this file in this assignment.
- **utils.py**: Utility functions and classes. You won't need to modify this file in this assignment.

In addition, there are two directories:

- **data/.** This directory contains the train, dev, and test-student splits of the SST, STS, QQP and ETPC datasets as `csv` files that you will be using in the first half of this project.
- **predictions/.** This directory contains the output predictions of your models for each of the datasets provided. When you have completed model training, the supplied pipeline components should place predictions here.
- **sanity_test/.** This folder contains tests for your completed implementations of Adam and BERT. Reminder to run these files only from their directory.

2.2 Setup

Once you are on an appropriate machine and have cloned the project repository, it's time to run the setup commands.

```
cd dnlp-final-project
./setup.sh
```

Make sure you have Anaconda or Miniconda installed. For information about how to install them, please see here.

This creates a conda environment called `dnlp`.

For the first part of the project, you are only allowed to use libraries installed by `setup.` No other external libraries are allowed. Do not use `transformers` except where indicated. You shouldn't change any existing command options (including defaults) or add new required parameters. **The first part of the project is just to establish an equal baseline for everyone.**

```
 conda activate dnlp
```

This activates the `dnlp` environment. Remember to do this each time you work on your code.

(Optional) If you would like to use PyCharm, select the `dnlp` environment. Example instructions:

1. Open the `dnlp-final-project directory` in PyCharm.
2. Go to PyCharm Preferences Project Project interpreter.
3. Click the gear in the top-right corner, then Add.
4. Select Conda environment Existing environment Click '...' on the right.
5. Select …`/miniconda3/envs/dnlp/bin/python`.
6. Select OK then Apply.

(Optional) If you would like to use VSCode, install the Python extension and select the `dnlp` environment. Example instructions:

1. Open the `dnlp-final-project` directory in VSCode.
2. Open the command palette (CTRL/CMD + SHIFT + P)
3. Select "Python: Select Interpreter". Click "Python 3.10 (`'dnlp'`)"

2.3 Setup on GWDG cluster

First, you have to log into the frontend node of the GWDG cluster. See here for more details, and check out their Playlist on YouTube. Note that some modules or binaries must be loaded using `module load`, but most of your code should work without. A modern `git` version is available this way. After logging in and cloning your repository, you must create a conda environment. Note that this script may take a long time to run, resolve the environment, and download the packages.

```
cd dnlp-final-project
./setup_gwdg.sh
```

The computing nodes of the GWDG cluster have no internet access. Since the project code requires several external files to run, e.g., `config.json` and `pytorch_model.bin`, we have to download them beforehand on the frontend node with internet access. This script should do that as well. You should run it once on the frontend node of the GWDG cluster. The necessary files will be saved in the cached path `~/.cache/huggingface/transformers/` on the frontend node where the computing nodes can access them.

3 BERT & BART

3.1 BERT

Bidirectional Encoder Representations from Transformers or BERT is a transformer-based model that generates contextual word representations. With its backbone being the Transformer, and by using the deeply bidirectional word representations, released in 2018, BERT took a large leap forward for contextual word embeddings/large language models/foundational models. Here, we will walk through the BERT model and give an overview of how the original BERT model was trained.

The original version of BERT was trained using two unsupervised tasks on Wikipedia articles.

Masked Language Modeling (MLM). To train BERT to extract deep bidirectional representations, the training procedure masks some percentage (15% in the original paper) of the word piece tokens and attempts to predict them. Specifically, the final hidden vectors corresponding to the masked tokens are fed into an output softmax layer over the vocabulary and are subsequently predicted. To prevent a mismatch between initial pre-training and later fine-tuning, the "masked" tokens are not always replaced by the [MASK] token in the training procedure. Instead, the training data generator chooses 15% of the token positions at random for prediction; then in 80% of these cases, the token is replaced `[MASK]`; in 10% of cases, the token is replaced with a random token, and in another 10% of cases, the token will remain unchanged.

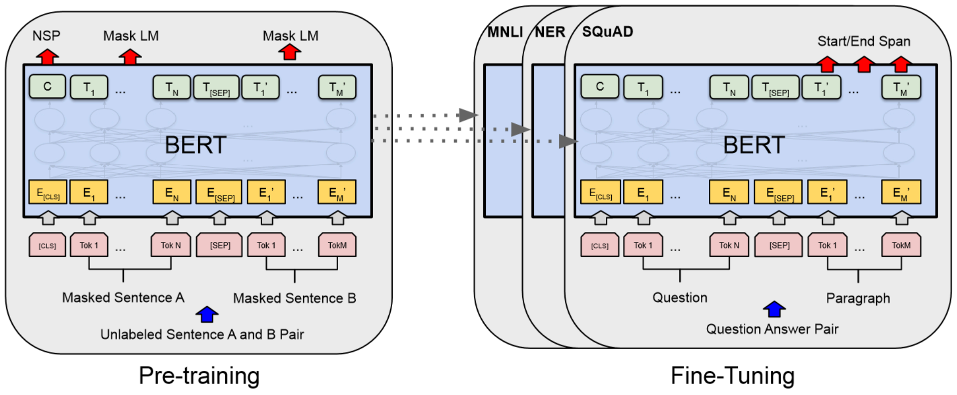

Figure 1: The original BERT model was trained on two unsupervised tasks, masked token prediction and next sentence prediction. Figure from [Devlin et al., 2019].

Next Sentence Prediction (NSP). To allow BERT to understand the relationships between two sentences, BERT is further fine-tuned on the Next Sentence Prediction task. Specifically, across training with these sentence pairs, the BERT model, 50% of the time, the actual next sentence, and 50% of the time, it is shown as a random sentence. The BERT model then predicts whether the second sentence was the next across these pairs.

3.2 BART

Bidirectional and Auto-Regressive Transformers (BART) is a model that combines the best of both encoder and decoder architectures from transformer models. BART was introduced by Facebook in 2019 and has achieved state-of-the-art performance in various natural language processing tasks, such as text summarization, translation, and question-answering.

BART has a unique approach to pre-training. It involves corrupting text with an arbitrary noising function and then learning to reconstruct the original text. The process consists of two main stages. Firstly, the encoder receives corrupted text, such as missing words or sentences, and processes it through its layers to understand the context and structure. Second, the decoder attempts to generate the original, uncorrupted text. BART has a dual mechanism that enables it to comprehend context, grammar, and semantics, making it a potent tool for both text comprehension and generation.

The model is trained using various noising strategies, including token deletion, text infilling, and sentence permutation, to ensure its ability to handle a broad range of language tasks. Its flexible architecture excels in tasks requiring deep understanding and text generation.

4 Implementing minBERT

We have provided you with several of the building blocks for implementing minBERT. In this section, we will describe the baseline BERT model and the sections that you must implement.

4.1 Tokenization (`tokenizer.py`)

The BERT model converts sentence input into tokens before performing any additional processing. Specifically, the BERT model utilizes a `WordPiece` tokenizer that splits sentences into individual words into word pieces. BERT has a predefined set of 30K different word pieces. These word pieces are then converted into `ids` for the rest of the BERT model. For example, the following words are converted into respective following word pieces.

Word	Word Piece
snow	[snow]
snowing	[snow,##ing]
fight	[fight]
fighting	[fight,##ing]
snowboard	[snow,##board]

Table 1: An overview of considered paraphrase types and their occurrences in the ETPC dataset.

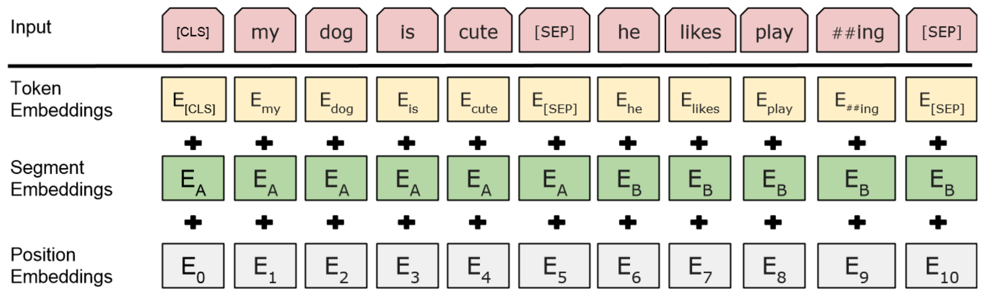
Input
[CLS]
my
dog
is
cute
[SEP]
he
likes
play
##ing
[SEP]
Token Embeddings
$E_{[CLS]}$ E_{my} E_{dog} E_{is} E_{cute} $E_{[SEP]}$ E_{he} E_{likes} E_{play} $E_{\#\#ing}$ $E_{[SEP]}$
Segment Embeddings
E_A E_A E_A E_A E_A E_A E_B E_B E_B E_B E_B
Position Embeddings
E_0 E_1 E_2 E_3 E_4 E_5 E_6 E_7 E_8 E_9 E_{10}

Figure 2: BERT embedding layer. The input embeddings that are utilized later in the model are the sum of the token embeddings, the segmentation embeddings, and the position embeddings. Figure from [Devlin et al., 2019].

In addition to separating each sentence into its constituent word pieces tokens, word pieces that have previously not been seen (i.e., that are not part of the original 30K word pieces) will be set as the `[UNK]` token. To ensure that all input sentences have the same length, each input sentence is further padded to a given `max_length` (512) with the `[PAD]` token. Finally, for many downstream tasks, BERT represents sentence embeddings with the hidden state of the first token. As a result, in BERT's implementation, `[CLS]` is prepended to the token representation of each input sentence. In this first part of this project, you will be working with the hidden state of this token.

Note that a part of vanilla BERT's training was a next-sentence prediction task. To help differentiate the first sentence from the second sentence input (given that BERT does not take in two distinct inputs as other models), the `[SEP]` token was further added to introduce an artificial separation between sentences.

4.2 Embedding Layer (`bert.BertModel.embed`)

After tokenizing and converting each token to `ids`, the BERT model uses a trainable embedding layer across each token. The input embeddings used in later portions of BERT are the sum of the token embeddings, the segmentation embeddings, and the position embeddings. Each embedding layer in the base version of BERT has a dimensionality of 768.

The learnable token embeddings map the individual input `ids` into vector representation for later use. More concretely, given some input word piece indices[3] $w_1, ..., w_k \in N$, the embedding layer performs an embedding lookup to convert the indices into token embeddings $v_1, ..., v_k \in \mathbb{R}^D$.

The learnable segmentation embeddings are used to differentiate between different sentences inputted into the model. We note that for this project, we do not consider the segmentation embeddings (we only consider individual sentences and not next-sentence prediction tasks[4]). They are implemented as placeholders only within the code base we provide.

Finally, the positional embeddings encode the position of different words within the input. Like the token embeddings, position embeddings are learned embeddings that are learned for each of the 512 positions in a given BERT input.

[3] A token index is an integer that tells you which row (or column) of the embedding matrix contains the word's embedding.

[4] Take a look at the literature to know why the next sentence prediction task is not used here.

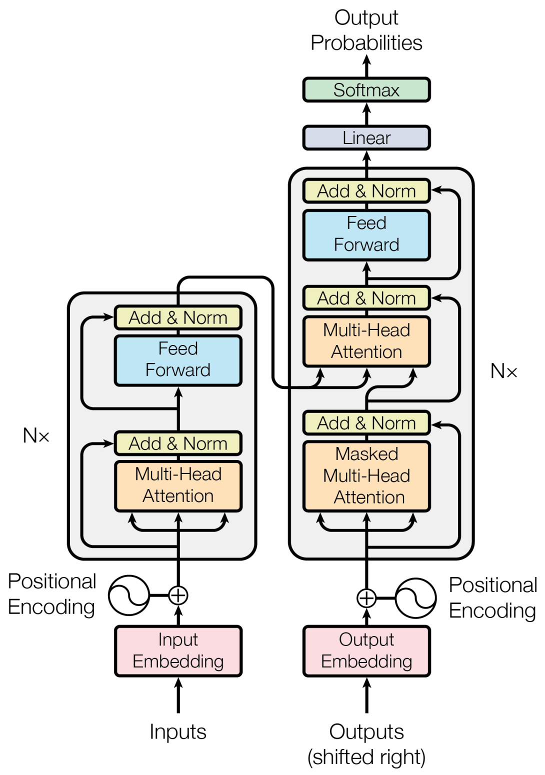

Figure 3: Encoder and Decoder Layers of Transformer used in BERT. Figure from [Vaswani et al., 2017].

4.3 BERT Transformer Layer (`bert.BertLayer`)

As described in the original BERT paper [Devlin et al., 2019], the base BERT uses 12 Encoder Transformer layers. These layers were defined initially in the work *Attention is All You Need*. The Transformer layer of the BERT transformer consists of multi-head attention, followed by an additive and normalization layer with a residual connection, a feed-forward layer, and a final additive and normalization layer with a residual connection. We briefly cover each of these layers here; we recommend that you read Section 3 of both cited papers for additional details.

4.4 Multi-Headed Self-Attention (`bert.BertSelfAttention.attention`)

Multi-head Self-Attention consists of a scaled-dot product applied across multiple different heads. Specifically, the input to each head is to a scaled-dot product consisting of queries, keys of dimension d_k, and values of dimension d_v. BERT computes the dot products of the query with all keys, divides each by $\sqrt{d_k}$, and applies a softmax function to obtain the weights on the values. In practice, BERT computes the attention function on a set of queries simultaneously, packed together into a matrix Q. The keys and values are also packed together into matrices K and V. Scaled dot-product attention is thus computed as

$$Attention(Q, K, V) = Softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

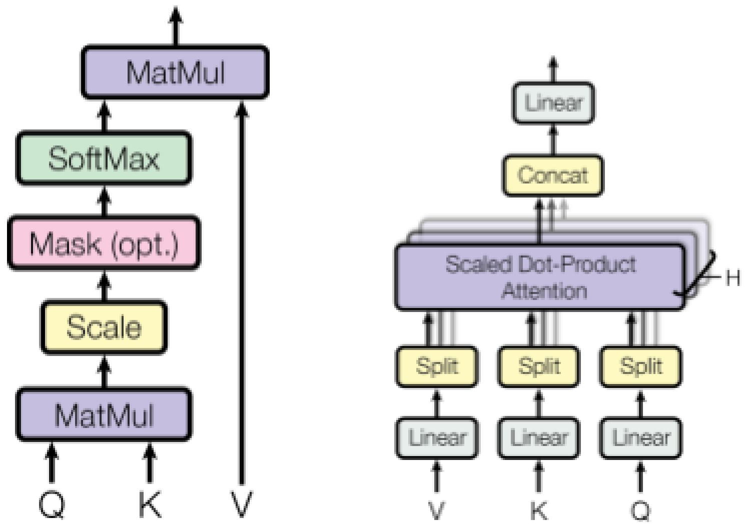

Figure 4: Scaled Dot-Product and Multi-Head Self-Attention. Figure from [Vaswani et al., 2017]

Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. With a single attention head, averaging inhibits this. Multi-head attention is computed as

$$MultiHead(Q, K, V) = concat\left(head_1, ..., head_h\right)W^O$$

Where $head_i = Attention\left(QW_i^Q, KW_i^K, VW_i^V\right)$ and where the projections are parameter matrices:

$$W_i^Q \in R^{d_{model} \times d_k}, W_i^K \in R^{d_{model} \times d_k}, W_i^V \in R^{d_{model} \times d_k} \text{ and } W_i^O \in R^{hd_v \times d_{model}}.$$

4.5 Position-wise Feed-Forward Networks

In addition to the attention sublayer, each transformer layer includes two linear transformations with a ReLU activation function [Agarap, 2019].

$$FFN(x) = max\left(0, xW_1 + b_1\right)W_2 + b_2$$

The feed-forward linear layers are followed by normalization layers. Thus, as specified in this section, multi-head attention consists of:

- Linearly projecting the queries, keys, and values with their corresponding linear layers. Namely, for each word piece embedding, BERT creates a query vector of dimension d_k, a key vector also of dimension d_k, and a value vector d_v.
- Splitting the vectors for multi-head attention.
- Following the Attention equation to compute the attended output of each head.
- Concatenating multi-head attention outputs to recover the original shape

Dropout We note that BERT applies Dropout to the output of each sub-layer before it is added to the sub-layer input and normalized. BERT also applies dropout to the sums of the embeddings and the positional encodings. BERT uses a default setting of $p_{drop} = 0.1$.

4.6 BERT output (`bert.BertModel.forward`)

As specified throughout this section, BERT consists of:

- An embedding layer that consists of token embedding **`token_embedding`** and positional embedding **`pos_embedding`.**
- BERT encoder layers which are a stack of 12 **`config.num_hidden_layers`** **`BertLayer`.**

After going through the respective layers, the outputs consist of:

- **`last_hidden_state`**: the contextualized embedding for each word piece of the sentence from the last **`BertLayer`** (i.e., the output of the BERT encoder).
- **`pooler_output`:** the `[CLS]` token embedding (The first token).

4.7 Code To Be Implemented: Multi-head Self-Attention and the Transformer Layer

We have provided you with much of the code for a BERT baseline model. Having gone over the basic structure of the BERT Transformer model, we will now describe the sections that need to be implemented:

4.7.1 BERT Multi-head Self-Attention (bert.SelfAttention.attention)

The first function that you should implement is the multi-head attention layer of the transformer. This layer maps a query and a set of key-value pairs to an output. The output is calculated as the weighted sum of the values, where the weight of each value is computed by a function that takes the query and the corresponding key. You can implement this attention function within **`bert.SelfAttention.attention`**.

4.7.2 BERT Transformer Layer (bert.BertModel and bert.BertLayer)

After implementing the BERT multi-head self-attention layer, you can implement the sections to realize the full BERT transformer layer. These functions can be filled in at **`bert.BertLayer.add_norm`**, **`bert.BertLayer.forward`**, and **`bert.BertModel.embed`.**

After finishing these steps, note that we provide a sanity check function at **`sanity_check.py`** in the `sanity_test` folder to test your implementation. It will reload two embeddings we computed with our reference implementation and check whether your implementation outputs match ours.

```
python sanity_check.py
```

5 NLP Tasks

In this project, you will implement some of the most important components of the BERT model [Devlin et al., 2019] so that you can better understand its architecture. Using pre-trained weights loaded into your BERT model, you will then perform sentence and paraphrase classification with the BERT model.

After this initial exercise, you will examine how to fine-tune BERT's contextualized embeddings to perform well on multiple sentence-level tasks (sentiment analysis, paraphrase detection, semantic textual similarity, and paraphrase classification). This section allows you to experiment with different options to obtain robust and generalizable sentence embeddings that perform well in different settings.

5.1 Paraphrase Detection

Paraphrase Detection is the task of finding paraphrases of texts in a large corpus of passages. Paraphrases are "rewordings of something written or spoken by someone else"; thus, paraphrase detection seeks to determine whether particular words or phrases convey the same semantic meaning [Fernando and Stevenson, 2008]. From a research perspective, paraphrase detection is an interesting task because it measures how well systems can "understand" fine-grained notions of semantic meaning.

As a concrete dataset example, the website Quora often receives duplicates of other questions. To better redirect users and prevent unnecessary work, Quora released a dataset that labeled whether different questions were paraphrased from each other.

Question Pair: (1) "What is the step-by-step guide to investing in the share market in India?", (2) "What is the step-by-step guide to investing in share market?" **Is Paraphrase**: No.

Question Pair: (1) "I am a Capricorn Sun Cap moon and cap rising...what does that say about me?", (2) "I'm a triple Capricorn (Sun, Moon, and ascendant in Capricorn) What does this say about me?" **Is Paraphrase**: Yes.

5.2 Semantic Textual Similarity (STS)

The semantic textual similarity (STS) task seeks to capture that some texts are more similar than others; STS aims to measure the degree of semantic equivalence [Agirre et al., 2013]. STS differs from paraphrasing in that it is not a yes or no decision; rather, STS allows for degrees of similarity. For example, on a scale from 5 (same meaning) to 0 (not at all related), the following sentences have the following relationships to each other[5]:

[5] These sentences and labels come from https://aclanthology.org/S131004.pdf

(5) The sentences are completely equivalent, as they mean the same thing:

The bird is bathing in the sink

Birdie is washing itself in the water basin.

(4) The two sentences are mostly equivalent, but some unimportant details differ:

In May 2010, the troops attempted to invade Kabul.

The US army invaded Kabul on May 7th last year, 2010.

(3) The two sentences are roughly equivalent, but some important information differs:

John said he is considered a witness but not a suspect.

He is not a suspect anymore

(2) The two sentences are not equivalent, but do share some details:

They flew out of the nest in groups.

They flew into the nest together.

(1) The two sentences are not equivalent, but are on the same topic:

The woman is playing the violin.

The young lady enjoys listening to the guitar.

(0) The two sentences are on different topics:

John went horseback riding at dawn with a whole group of friends.

Sunrise at dawn is a magnificent view to take in if you wake up early enough for it.

5.3 Sentiment Analysis

A basic task in understanding a given text is classifying its polarity (i.e., whether the expressed opinion in a text is positive, negative, or neutral). Sentiment analysis can be used to determine individual feelings towards particular products, politicians, or news reports.

As a concrete dataset example, the Stanford Sentiment [6] [Socher et al., 2013] consists of 11,855 single sentences extracted from movie reviews. The dataset was parsed with the

[6] https://nlp.stanford.edu/sentiment/treebank.html

Stanford parser[7] and includes 215,154 unique phrases from those parse trees, each annotated by three human judges. Each phrase has a label of negative, somewhat negative, neutral, somewhat positive, or positive.

Movie Review: Light, silly, photographed with colour and depth, and rather a good time.
Sentiment: 4 (Positive)

Movie Review: Opening with some contrived banter, cliches and some loose ends, the screenplay only comes into its own in the second half.
Sentiment: 2 (Neutral)

Movie Review: ... a sour little movie at its core; an exploration of the emptiness that underlay the relentless gaiety of the 1920's ... The film's ending has a "What was it all for?"
Sentiment: 0 (Negative)

5.3 Paraphrase Type Detection

While the QQP dataset (5.1 Paraphrase detection) only differentiates between two sentences being paraphrased or not, the paraphrase type detection task is a bit more advanced. Instead of asking if two sentences are paraphrases, the task is to find the correct type of paraphrase. There are seven different types of paraphrases.

Note that one sentence pair can belong to multiple paraphrase types.

The two columns `sentence1(2)_segment_location` contain information about which token belongs to which paraphrase type. The two columns `sentence1(2)_tokenized` contain each token of the sentence in the original sentence ordering.[8]

Below is an example of a data point from this dataset. Note that you must convert the labels to a binary format to train on them.

Sentence1: Amrozi accused his brother, whom he called "the witness", of deliberately distorting his evidence.
Sentence2: Referring to him as only "the witness", Amrozi accused his brother of deliberately distorting his evidence.

Paraphrase_types: [2,6,7,0,0,0,0]

Sentence1_segment_location: [7, 7, 7, 7, 0, 2, 0, 2, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7]

Sentence2_segment_location: [2, 2, 2, 0, 7, 0, 0, 0, 0, 0, 7, 7, 7, 7, 0, 0, 0, 0, 0, 0]

[7] https://nlp.stanford.edu/software/lex-parser.shtml

[8] For more details see regarding the different paraphrase types see here.

Sentence1_tokenized: ['Amrozi', 'accused', 'his', 'brother', ',', 'whom', 'he', 'called', '``', 'the', 'witness', "''", ',', 'of', 'deliberately', 'distorting', 'his', 'evidence', '.']

Sentence2_tokenized: ['Referring', 'to', 'him', 'as', 'only', '``', 'the', 'witness', "''", ',', 'Amrozi', 'accused', 'his', 'brother', 'of', 'deliberately', 'distorting', 'his', 'evidence', '.\n']

5.4 Paraphrase Type Generation

In addition to the four classification tasks above, you will perform one generation task. The paraphrase generation task is closely related to the paraphrase type detection task. Instead of detecting the paraphrase types, you will give the model one sentence and a list of paraphrase types as input. The model should generate a paraphrased version of the given sentence using the given paraphrase types.

The data for this task is the same as the data for the paraphrase type detection task.

For a better understanding of the Paraphrase type detection and generation task, you can look into the original paper, which proposes "Paraphrase Types for Generation and Detection" by Jan Philip Wahle, Bela Gipp, and Terry Ruas.

6 Deliverable: BERT and BART Model for multiple tasks

Having implemented a working minBERT model, you will now utilize pre-trained model weights and the outputted embeddings from your implemented BERT model to perform sentiment analysis, paraphrase prediction, semantic textual similarity analysis, and paraphrase classification on corresponding datasets. In addition, you will use a pre-trained BART model to perform paraphrase classification and generation.

Furthermore, you will fine-tune your models on each dataset to achieve better results. In the end, you will have three different versions of the minBERT model and two different versions of the BART model; each fine-tuned for a different task for the first part of the project. For the second one (Section 7), you must implement at least one improvement for each baseline developed in the first part. The SST, STS, QQP, and ETPC datasets are in the `data` subfolder.

6.1 Datasets

6.1.1 Quora Paraphrase Dataset

The Quora paraphrase dataset consists of 400,000 question pairs with labels indicating whether particular instances are paraphrases of one another.

We have provided you with a subset of this dataset with the following splits. For the Quora dataset, we have the following splits:

- train (141,506 examples)
- dev (20,215 examples)
- test (40,431 examples)

Given the binary labels of this dataset, the metric that we utilize to test this dataset is accuracy.

6.1.2 SemEval Semantic Textual Similarity Benchmark Dataset (STS)

The SemEval STS Benchmark dataset consists of 8,628 different sentence pairs of varying similarity on a scale from 0 (unrelated) to 5 (equivalent meaning). For some examples of these pairs, see Section 5.2.

For the STS dataset, we have the following splits:

- train (6,041 examples)
- dev (864 examples)
- test (1,726 examples)

When testing this dataset, we will (as in the original SemEval [Agirre et al., 2013] paper) calculate the Pearson correlation of the true similarity values against the predicted similarity values across the test dataset.

6.1.3 Stanford Sentiment Treebank Dataset (SST)

The Stanford Sentiment Treebank[9] [Socher et al., 2013] consists of 11,855 single sentences from movie reviews extracted from movie reviews. The dataset was parsed with the Stanford parser[10] and includes 215,154 unique phrases from those parse trees, each annotated by three human judges. Each phrase has a label of negative, somewhat negative, neutral, somewhat positive, or positive. Within this project, you will utilize BERT embeddings to predict these sentiment classification labels. To summarize, for the SST dataset, we have the following splits:

- train (8,544 examples)
- dev (1,101 examples)
- test (2,210 examples)

6.1.4 Extended Typology Paraphrase Corpus (ETPC)

The Extended Typology Paraphrase Corpus[11] consists of 3799 sentence pairs with labels indicating which paraphrase types the sentence pair belongs. It is the largest corpus to date annotated with atomic paraphrase types.

Since the dataset is small compared to the other datasets and is used for two different tasks, we have split the dataset only into a train and test set. It is **up to you** to find and implement good train/validation splits for the two tasks.

We have the following splits:

- train (2525 examples)
- test for detection (573 examples)
- test for generation (701 examples)

6.2 Code To Be Implemented

6.2.1 Training and predicting with BERT

Within the `multitask_classifer.py` file you will find a pipeline that

[9] https://nlp.stanford.edu/sentiment/treebank.html

[10] https://nlp.stanford.edu/software/lex-parser.html

[11] https://huggingface.co/datasets/jpwahle/etpc

- Calls the BERT model to encode the sentences for their contextualized representations.
- Feeds in the encoded representations for the different tasks
- Fine-tunes the BERT model on the downstream tasks (e.g., sentence classification, paraphrase prediction and detection, semantic textual similarity)

We have provided function definitions that predict the sentiment scores of sentences, predict whether a sentence pair are paraphrases of each other, and predict the similarity of two input texts.

We similarly provide you with ready-made code that loads in the training data of the STS, SemEval, and Quora datasets and evaluates your model on the provided `dev` sets. Whether you wish to keep this formulation for training is up to you. Here, we give a brief overview of this code.

- **`multitask_classifier.MultitaskBERT`**: A class that imports the weights of a pre-trained BERT model and can predict sentiment, paraphrases, and semantic textual similarity.
- **`multitask_classifier.MultitaskBERT.forward`**: The output of your BERT model. You can choose to experiment with the contextual word embeddings of particular word pieces or extract just the **`pooler_output`**.
- **`multitask_classifier.MultitaskBERT.predict_paraphrase`**: Predicts whether two sentences are paraphrases of each other based on BERT embeddings.
- **`multitask_classifier.MultitaskBERT.predict_similarity:`** Predicts the similarity of two sentences based on BERT embeddings.
- **`multitask_classifier.MultitaskBERT.predict_sentiment`**: Predicts the sentiment of a sentence based on BERT embeddings. As a baseline, you should call the new **`forward()`** function followed by a dropout and linear layer.
- **`multitask_classifier.MultitaskBERT.predict_paraphrase_types`**:

  Predicts which kind of paraphrases two sentences are based on BERT embeddings.
- **`multitask_classifier.train_multitask`**`()`: A function for training your model. It is largely your choice how to train your model. As a baseline, you will already find code to train your model on the SST sentiment dataset in this function.
- **`datasets.SentenceClassificationDataset`**: A class for handling the SST sentiment dataset.
- **`datasets.SentencePairDataset`**: A class for handling the other three datasets.
- **`evaluations.test_multitask`**`()`: A function for testing your model. You should call this function after loading in an appropriate checkpoint of your model.

Within this file, you are to implement the `MultitaskBert` model. You will implement this class to encode sentences using BERT and obtain the pooled representation of each sentence or sentence pair[12]. The class will then classify the sentence or sentence pair by applying dropout on the pooled output and then projecting it using a linear layer. Finally (already implemented), the model must be able to adjust its parameters depending on whether we are using pre-trained weights or are fine-tuning.

Note, that the parts you must implement in this class are marked with a `NotImplementedError`.

6.2.2 Training and predicting with BART

The Python scripts `bart_generation.py` and `bart_detection.py` contain everything needed to do the two tasks: paraphrase type detection and paraphrase generation. You will also find the prepared ETPC data set required for these tasks in the data folder. Note that the training set is the same for both tasks.

Note that you should split the train data into a train and validation set for a meaningful evaluation. Finding an appropriate train/validation split ratio is up to you.

The `bart_detection.py` and `bart_generation.py` files provide you with a pipeline to load, train, and test a BART model on those two tasks:

- **`transform_data()`**: A function to transform the data frame into a Dataloader. We recommend you encode the sentences using the AutoTokenizer from the Transformer library. Further, you should change the labels into binary labels for paraphrase detection, as described in the comment in the code. This is not necessary for paraphrase generation.
- **`train_model():`** Here, you implement a function that finetunes the loaded BART model on the ETPC dataset. Return the model and save it at appropriate checkpoints during training.
- **`test_model():`** Implement a function that uses a given model to predict/generate outputs for the test_set. Make sure to return the predictions/generations as a dataframe, where the columns are named as asked in the comment of the function. This is important since our evaluation of your model's predictions requires this format.
- **`evaluate_model():`** We provide this function to you to measure your model performance. For the detection task, the function calculates and returns the accuracy of the predictions. For the generation task, the function calculates and returns the BLEU score of the generations.

[12] See the forward function in `bert.py` for how to access this representation.

- `finetune_paraphrase_detection(generation)():` This pipeline loads and transforms the data, loads and trains a BART model, and saves the model's predictions on the test_set using all the described functions above.

6.2.3 Adam Optimizer

In addition to implementing BertSentimentClassifer, you will further implement the `step()` function of the Adam Optimizer based on Decoupled Weight Decay Regularization [Loshchilov and Hutter, 2019] and Adam: A Method for Stochastic Optimization [Kingma and Ba, 2017] to train a sentiment classifier.

The Adam optimizer is a method for efficient stochastic optimization that only requires first-order gradients. The technique computes adaptive learning rates for different parameters by estimating the first and second moments of the gradients. Specifically, at each time step, the algorithm updates the exponential moving averages of the gradient m_t and the squared gradient v_t , where the hyperparameters $\beta_1, \beta_2 \in [0, 1)$ control the rate of exponential decay of these averages. Given that these moving averages are initialized at 0 at the initial time step, these averages are biased towards zero. As a result, a key aspect of this algorithm is performing bias correction to obtain $\hat{m}_t$ and $\hat{v}_t$ at each time step. We present the full algorithm below [Kingma and Ba, 2017]:

Algorithm 1 *Adam*, our proposed algorithm for stochastic optimization. g_t^2 indicates the elementwise square $g_t \odot g_t$. All operations on vectors are element-wise. Good default settings for the machine learning problems in the original paper were $\alpha = 0.0002$, $\beta_1 = 0.1$, $\beta_2 = 0.001$ and $\epsilon = 10^{-8}$.

```
Require: $\alpha$: Stepsize
Require: $\beta_1, \beta_2 \in (0, 1]$: Exponential decay rates for the first and second moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters $\theta$
Require: $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize initial 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize initial 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while $\theta^t$ not converged do
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \varepsilon)$ (Update parameters)
  end while
  return $\theta^t$ (Resulting parameters)
```

Note that the efficiency of algorithm 1 can, at the expense of clarity, be improved upon by changing the order of computation by replacing the last three lines in the loop with the following line:

$$\alpha_t \leftarrow \alpha \cdot \sqrt{1 - \beta_2^t} / \left(1 - \beta_1^t\right) \text{ and } \theta_t \leftarrow \theta_{t-1} - \alpha_t \cdot m_t / \left(\sqrt{v_t} + \varepsilon\right)$$

This eliminates the need to calculate the bias-corrected estimates manually. If you are unsure, look at the papers mentioned.

You should implement the `step()` function of the Adam Optimizer. Our reference uses the "efficient" method of computing the bias correction mentioned at the end of Section 2 "Algorithm" in [Kingma and Ba, 2017] (and at the end of the algorithm above) in place of the intermediate $\hat{m}$ and $\hat{v}$ method. Similarly, the learning rate should be incorporated into the weight decay update. You can test your implementation in the sanity_test directory by running:

```
python optimizer_test.py
```

6.2.4 Training minBERT

You should test your completed model for all datasets using pre-trained and fine-tuned embeddings. You can run training by using the following command:

```
python multitask_classifier.py --option [pretrain / finetune] task=[sst, sts,
qqp, etpc]
```

Look at the argument parsing part of the code to understand more possible arguments. If you are running the code on the GWDG cluster, you should add the `--use_gpu` and `--local_files_only` flags.

As a baseline, your implementation should have results similar to the following on the dev datasets (Mean reference accuracies and standard deviations, n=10):

```
Finetuning for SST: Dev Accuracy: 0.522 (0.006)

Finetuning for STS: Dev Correlation: 0.361 (0.010)
```

```
Finetuning for QQP: Dev Accuracy: 0.759 (0.007)
```

Keep in mind this should be achieved with default settings. We used very simple linear layers and a minimal implementation with the correct loss function on top of the given BART embeddings. For the large data set of Quora, it is sufficient if you only train for a single epoch. Please let us know if you achieve significantly higher performance with another minimal implementation. The STS score is especially bad.

6.2.5 Training BART

You can run training by running the `bart_detection.py` (for the detection task) or `bart_generation.py` (for the generation task) file:

```
python bart_generation.py
```

```
python bart_detection.py
```

Again, you should use the GWDG arguments as above. You can also implement a similar argument system as we did for the other implementation.

As a baseline, your implementation should have results similar to the following on the dev datasets:

```
Finetuning for Paraphrase Detection: Dev Accuracy: 0.829 (0.003)

Finetuning for Paraphrase Generation: BLEU Score: 46.7 (0.136)
```

Again, we used a very minimal implementation with a split ratio of `80/20`, `5` epochs, and a learning rate of `1e-5`. Please let us know if you achieve significantly higher performance with another minimal implementation.

For Part 1, you may only use the training set and our dev set to train, tune, and evaluate your models. For this section, for grading, we will largely be looking at your code and implementation. For the first part of the project, it is not necessary to exceed the baseline by a significant margin. Training for each dataset should take no more than 5 and 15 minutes (depending on your GPU).

7 Extensions and Improvements for Additional Downstream Tasks

While we have focused on implementing key aspects of BERT and BART in the first half of this project, for the rest of this project (and the part that will make up the bulk of your grade on the final deliverable), you will have free rein to explore other datasets to better fine-tune and otherwise adjust your BERT and BART embeddings so that their performances improve on the following tasks: BERT - paraphrase prediction, semantic textual similarity, and sentiment analysis; BART - paraphrase type generation and detection.

Note that this section will test you using the SST dataset for sentiment analysis, the Quora dataset for paraphrase prediction, the SemEval dataset for semantic textual analysis, and the ETPC for paraphrase type generation and detection. You will find the `train`, `dev`, and `test` datasets for each of these datasets within the `data` folder. The ETPC dataset only contains `train` and `test` sets. You **may only use our training and dev data** to train, tune, and evaluate your models. If you use the official test data of these datasets to train, tune, or evaluate your models, or if you manually modify your csv solutions in any way, you are committing an **honor code violation**[13].

In addition to embeddings extracted from pre-trained BERT weights provided to you in the prior part of this assignment, you are allowed to utilize other pre-existing NLP tools such as a POS tagger, dependency parser, Wordnet, coreference module, etc... that are not built on top of pre-trained contextual embeddings. You may further use other embeddings that you train yourself. However, for this assignment, you may not, for instance, use pre-trained embeddings from the transformers library.

For this next part of the project, you can reorganize the functions inside each class, create new classes, and otherwise retrofit your code.

7.1 Possible Extensions

Many possible extensions can improve your model's performance on our proposed tasks. We recommend finding a relevant research paper for each improvement you wish to attempt (or at least see if others have tried them). Here, we provide some suggestions, but you might look elsewhere for interesting ways of improving sentence embeddings for the selected tasks. Do not limit yourself to this list; explore further.

7.1.1 Additional Pretraining

How to Fine-Tune BERT for Text Classification? [Sun et al., 2020]

As outlined, BERT was trained in a general domain with a different data distribution than the datasets we will use to grade your project. A natural way to improve your model would be to pre-train your BERT model further with target-domain data. This would involve implementing and training on the masked LM objective or predicting tokens as outlined in

[13] This will result in automatic **Fail**.

the training datasets that we provided. See [Devlin et al., 2019] for more details on BERT's pre-training.

7.1.2 Multiple Negative Ranking Loss Learning

Efficient Natural Language Response Suggestion for Smart Reply [Henderson et al., 2017]

Another effective way of improving your embeddings would be to fine-tune your model with Multiple Negative Ranking Loss[14]. With this loss function, training data consists of sets of K sentence pairs $\left[\left(a_1, b_1\right),..., \left(a_n, b_n\right)\right]$ where a_i, b_i are labelled as similar sentences and all $\left(a_i, b_j\right)$ where $i \neq j$ are not similar sentences. The loss function then minimizes the distance between a_i, b_i while it simultaneously maximizing the distance $\left(a_i, b_j\right)$ where $i \neq j$. Specifically, training is to minimize the approximated mean negative log probability of the data. See sbert[15] and [Henderson et al., 2017] for additional details.

7.1.3 Cosine-Similarity Fine-Tuning

Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks [Reimers and Gurevych, 2019]

Additional fine-tuning can further improve your BERT model on one of the pre-selected tasks. SemEval dataset the similarity between two embeddings is often computed using their cosine similarity. A way of potentially improving your embeddings would thus be to utilize CosineEmbeddingLoss[16] while fine-tuning on this dataset. In this setup, sentences that are the equivalent have a cosine similarity of 1 and those that are unrelated have a cosine similarity score of 0.

7.1.4 Fine-Tuning with Regularized Optimization

SMART: Robust and Efficient Fine-Tuning for Pre-trained Natural Language Models Principled Regularized Optimization [Jiang et al., 2020]

Aggressive fine-tuning can often cause over-fitting. This can cause the model to fail to generalize to unseen data. To combat this in a principled manner, [Jiang et al., 2020] propose (1) Smoothness-inducing regularization, which effectively manages the complexity of the model and (2) Bregman proximal point optimization, which is an instance of trust-region methods and can prevent aggressive updating. See [Jiang et al., 2020] and their repository[17] for additional details.

---

[14] https://www.sbert.net/docs/package_reference/losses.html

[15] https://www.sbert.net/examples/training/nli/README.html#multiplenegativesrankingloss

[16] https://pytorch.org/docs/stable/generated/torch.nn.CosineEmbeddingLoss.html

[17] https://github.com/namisan/mt-dnn

7.1.5 Multitask Fine-Tuning

BERT and PALs: Projected Attention Layers for Efficient Adaptation in Multi-Task Learning. [Stickland and Murray, 2019]
MTRec: Multi-Task Learning over BERT for News Recommendation. [Bi et al., 2022]
Gradient surgery for multi-task learning. [Yu et al., 2020]

Rather than fine-tuning BERT on individual tasks, you can alternatively use multi-task learning to update BERT. For example, Bi et al. [2022], use multi-task learning adding each together each loss to the tasks of category classification and named entity recognition. Using multi-task learning, however, depending on how the model is fine-tuned is not always beneficial. Gradient directions of different tasks may conflict with one another. Yu et al. [15] recommend a technique called Gradient Surgery that projects the gradient of the $i - th$ task g_i onto the normal plane of another conflicting task's gradient g_i.

7.1.6 Contrastive Learning

Simple Contrastive Learning of Sentence Embeddings. [Gao et al., 2021]

Gao et al. [2021] proposed a simple contrastive learning framework that works with unlabeled and labeled data called SimCSE. Unsupervised SimCSE simply takes an input sentence and predicts itself in a contrastive learning framework, with only standard dropout used as noise. In contrast, supervised SimCSE incorporates annotated pairs from NLI datasets into contrastive learning by using entailment pairs as positives and contradiction pairs as hard negatives. You can utilize a similar approach to improve sentence embeddings across different models.

7.1.7 Paraphrase Generation with Deep Reinforcement Learning

Paraphrase Generation with Deep Reinforcement Learning [Li, Jiang, Shang et al., 2018]

Li, Jiang, Shang et al. [2018] propose a deep reinforcement learning framework consisting of a data-trained generator and evaluator for paraphrase generation. The generator, a sequence-to-sequence model, creates paraphrases, while the evaluator, a deep matching model, judges their accuracy. The generator is first trained by deep learning and later fine-tuned with reinforcement learning using the evaluator's output as a reward. You can use a similar approach to generate more accurate paraphrases.

7.1.8 Controlled Paraphrase Generation

Quality Controlled Paraphrase Generation, [Bandel et al., 2022]

In current paraphrase generation models the quality of a generated paraphrase is not directly controlled. Bandel et al. [2022] try to overcome this limitation by introducing a quality-guided controlled paraphrase generation model that allows directly controlling the quality dimensions. The quality for a sentence **s** and a paraphrase **s′** is defined as a three-dimensional vector q(**s**,**s′**) containing the semantic similarity, the syntactic and lexical variation. You can add the quality controlling architecture, as proposed in the paper, to your model to improve the quality of your paraphrase generation.

7.1.9 Efficient finetuning

PIP: Parse-Instructed Prefix for Syntactically Controlled Paraphrase Generation [Wan et al., 2023]

Fine-tuning a large language model by updating all its parameters is computationally expensive. Thus, more efficient fine-tuning methods have been developed, which only update a small fraction of the parameters. One efficient fine-tuning method is the so-called Prefix Tuning. Prefix tuning keeps the parameters of the pretrained model fixed and only trains a small continuous "prefix" that is input to the model. Wan et al. [2023] propose two methods to instruct a model's encoder prefix to capture syntax-related knowledge.

You can use their or other efficient fine-tuning methods to improve your training of the models.

7.1.10 Additional datasets

Fine-tuning the BERT/BART model on different datasets is an additional approach you could apply. There are a lot of different datasets and tasks that you can potentially apply to your model to get more robust embeddings. See the following for some example datasets:

- https://arxiv.org/abs/1508.05326
- http://sbert.net/datasets/paraphrases
- https://arxiv.org/abs/1704.05426
- https://aclanthology.org/2022.emnlp-main.631

7.1.11 Additional input features

Although deep learning can learn end-to-end without feature engineering, using the right input features (e.g., part-of-speech tag, named entity type, etc.) can still boost performance significantly. If you implement a model like this, reflect on the tradeoff between feature engineering and end-to-end learning and comment on it in your repository.

7.1.12 Other improvements

You can improve your performance through many other things besides training changes. The suggestions in this section are just some examples; it will take time to run the necessary experiments and draw the necessary comparisons. Remember that we will be grading your experimental thoroughness, so do not neglect the hyperparameter search!

- *Regularization*. The baseline code uses dropout. You could further experiment with different values of dropout and different types of regularization.
- *Sharing weights*. The baseline code outlines a way to use different distinctive layers for predicting whether sentences are paraphrases, their semantic similarity, and each sentence's sentiment. You could potentially share some layers amongst these different tasks to improve performance.

- *Model size and the number of layers*. With any model, you can increase the number of layers utilized to predict each of the tasks.
- *Optimization algorithms*. The baseline uses the Adam optimizer. PyTorch supports many other optimization algorithms. Also, consider varying the learning rate.
- *Ensembling*. Ensembling almost always boosts performance, so try combining several of your models together for your final submission. However, ensembles are more computationally expensive to run.
- *Hyperparameter Optimization*. While we provide some defaults for various hyperparameters, these do not necessarily lead to the best results. Another approach would be to perform a hyperparameter search to find the best hyperparameters for your model.

7.1.13 Other approaches

The models and techniques we have presented here are far from exhaustive. There are many published papers on the tasks we are testing - there may be new ones we haven't seen yet! In addition, there is lots of deep learning research on many different tasks that may help improve your model[18] These papers may contain interesting ideas that you can apply to build more robust and semantically rich embeddings.

7.2 Bonus Tasks

Note that the following tasks are not mandatory. Although optional, these tasks can help you better understand the concepts and inspire new ideas and extra points. If you decide to try them, please include them in your repository promptly. These tasks **will not** affect your grades negatively. In other words, solving them can only positively affect your grade.

7.2.1 Paraphrase Types Detection with minBERT

You are required to complete the **multitask_classifier.MultitaskBERT.predict_paraphrase_type** function to establish a baseline for paraphrase detection using minBERT. After setting this baseline, explore and implement an improvement to enhance the model's performance. Additionally, investigate and compare the performance differences between the minBERT and BART models for this task. This comparison might show some limitations of the minBERT model since this paraphrase detection task is quite challenging. However, you might be able to improve the minBERT's performance with some of the suggested (chapter 7.1) improvements.

```
Finetuning for ETPC (Paraphrase Detection): Dev Accuracy: 0.253 (0.004)
```

[18] http://nlpprogress.com/

7.2.2 Multitask classification

You will train one model combined on the three datasets: QQP, SST, and STS, to evaluate multitask learning capabilities. Beyond training, implement an improvement that boosts the model's multitask performance (average performance on all three tasks). Compare the multitask model's performance against models trained on individual tasks to assess the effectiveness of multitask learning. How does training on a different dataset influence the model’s performance on another task? Make sure to discuss the obtained results. What are the benefits/limitations of this approach? Do you identify any trends?

To start, try using the multitask argument already provided, but you may need to modify your training loops.

You can systematically investigate how different training (e.g., train on one dataset after another, shuffle all datasets together, and train the model simultaneously, etc) affects its performance on all three tasks. Starting from those insights, you can come up with some ideas of how to improve the multitask performance of the model. Furthermore, you could use some ideas from the papers presented in 7.1. Implement <u>at least one idea</u> that leads to an improvement in the multitask performance.

8 Notes

Projects that partially, or even exclusively, use large language model (LLM) APIs like OpenAI's GPT-3, 3.5, and now 4, or ChatGPT are within scope for the final project. However, students should also be aware that they are expected to make a substantive scientific and/or engineering contribution on top of such APIs. Since LM APIs abstract away a lot of the challenge of building NLP systems, the relative contribution of a project using an LM API is expected to be more than a project that works directly with LMs via deep learning libraries like Transformers or PyTorch. With this in mind, projects that simply prompt an LLM like GPT-3 to generate text for a specific use case (e.g., summarizing news articles or generating song lyrics) are unlikely to be enough work for a final project. Examples of more appropriate contributions include (but are not limited to): Systematically identifying and/or benchmarking some capabilities, weaknesses, and/or biases of current LLMs Building substantial systems that interface with LLM APIs to enable new applications or workflows for end-users.

9 Deliverables

The project will have three deliverables.

1. The initial model's code for the five baseline models-tasks (Section 6).
2. The model's scores of the improved models and code for the sentiment analysis, paraphrase detection, semantic textual similarity tasks, paraphrase types detection and generation (SST, Quora, STS, and ETPC) (Sections 7).
3. The detailed repository for the fine-tuned model (Section 9.3).
4. **(optional)** the scores of the bonus models and code for the bonus task.

9.1 Part 1 - Submission Instructions for minBERT and BART

You will submit the minBERT and BART part of this project to your tutor (detailed below):

1. Verify that the following files exist at these specified paths within your repository:
   - `predictions/bert/sst-sentiment-test-output.csv`
   - `predictions/bert/sts-similarity-test-output.csv`
   - `predictions/bert/quora-paraphrase-test-output.csv`
   - `predictions/bart/etpc-paraphrase-detection-test-output.csv`
   - `predictions/bart/etpc-paraphrase-generation-test-output.csv`

   You don't have to submit any Python files; however, we should be able to find all relevant Python files in your repository.
2. Run **`prepare_submit.py`** to produce your **`dnlp_final_project_submission.zip`** file.
3. Rename your submission file to **`id_group_name_dnlp_final_project_submission.zip`**
4. Submit your `id_group_name_dnlp_final_project_submission.zip` in StudIP here.

At a high level, the submission file for the dev/test datasets should look like the following:

```
id      Predicted_Sentiment
001fefa37a13cdd53fd82f617       4
00415cf9abb539fbb7989beba       2
00a4cc38bd041e9a4c4e545ff       1
...
fffcaebf1e674a54ecb3c39df3
```

The submission file for the STS dataset should look like the following:

id	**Predicted_Similarity**
8f4d49b9f4558f9e45423e84c	1.000
1c5cd37407630a3ba19a0f2ad	0.4051
318c885e36cc9e6f6bb7de7dd	0.2138
...	
4e1ef3b635d01039a8a8f059b	0.7462

The submission file for the Paraphrase dataset should look like the following:

id	**Predicted_Is_Paraphrase**
872887985e1e0f2dd5b690ffd	1
472398907a6adb9ed2f660550	0
c3ceaaed421cc008282efdf8a	0
...	
5e10dfc4ac8ae205f3e114445	1

The submission file for the Paraphrase type detection should look like the following:

id	**Predicted_Paraphrase_Types**
872887985e1e0f2dd5b690ffd	[0,1,0,1,0,1,1]
472398907a6adb9ed2f660550	[1,1,0,0,0,0,0]
c3ceaaed421cc008282efdf8a	[1,1,0,0,0,1,1]
...	
5e10dfc4ac8ae205f3e114445	[0,0,0,1,0,0,0]

The submission file for the Paraphrase type generation should look like the following:

id	**Generated_sentence2**
2317539b4e5a4160a253d5199685955c	Referring to him as only "the witness",...
eaab3036395f4bcc9c9c93d678c622a1	On June 10, the ship's owners had published…
90fe3baf95d84645956219234f72694b	PG&E Corp. shares jumped $1.63 or 8 percent to…
...	
4c02bbf5f56c447e97a0d8f40b358777	With the scandal hanging over Stewart's company, revenue…

9.2 Part 2 - Submission Instructions for Fine-Tuned Models (BERT and BART)

You are allowed to submit your `dev` and `test` results **only once** in StudIP. In case of an extraordinary situation, please contact your tutors or the lecturer.

You may use the **`evaluations.test_multitask()`** function in the `evaluations.py` script to generate a submission file of the correct format. At a high level, the submission file for the SST dataset should look like the following:

```
id      Predicted_Sentiment
001fefa37a13cdd53fd82f617       4
00415cf9abb539fbb7989beba       2
00a4cc38bd041e9a4c4e545ff       1
...
fffcaebf1e674a54ecb3c39df3
```

The submission file for the STS dataset should look like the following:

```
id      Predicted_Similarity
8f4d49b9f4558f9e45423e84c       1.000
1c5cd37407630a3ba19a0f2ad       0.4051
318c885e36cc9e6f6bb7de7dd       0.2138
...
4e1ef3b635d01039a8a8f059b       0.7462
```

The submission file for the Paraphrase dataset should look like the following:

```
id      Predicted_Is_Paraphrase
872887985e1e0f2dd5b690ffd       1
472398907a6adb9ed2f660550       0
c3ceaaed421cc008282efdf8a       0
...
5e10dfc4ac8ae205f3e114445       1
```

The submission file for the Paraphrase type detection should look like the following:

```
id      Predicted_Paraphrase_Types
872887985e1e0f2dd5b690ffd       [0,1,0,1,0,1,1]
472398907a6adb9ed2f660550       [1,1,0,0,0,0,0]
c3ceaaed421cc008282efdf8a       [1,1,0,0,0,1,1]
...
5e10dfc4ac8ae205f3e114445       [0,0,0,1,0,0,0]
```

The submission file for the Paraphrase type generation should look like the following:

```
id      Generated_sentence2
2317539b4e5a4160a253d5199685955c    Referring to him as only "the witness",...
eaab3036395f4bcc9c9c93d678c622a1    On June 10, the ship's owners had published…
90fe3baf95d84645956219234f72694b    PG&E Corp. shares jumped $1.63 or 8 percent
to…
...
4c02bbf5f56c447e97a0d8f40b358777    With the scandal hanging over Stewart's
company, revenue…
```

The header is required as well as the first column being a 25-digit hexadecimal ID for each example (IDs defined in each of the respective test/dev files), and the last column is your predicted answer. The rows can be in any order. For the `test` and `dev` of the QQP, STS and SST data, you must submit a prediction for every example. For the ETPC dataset, you only must submit a prediction for every example from the test set.

Create a submission file the same way as in Section 9.1

9.3 Repository Instructions

In this course, we will not have a formal report. Instead, we will invest time in having a high-quality repository. Thus, all groups are required to follow the mandatory README template. The NeurIPS[19] template available at PapersWithCode[20] is also a great reference for a solid repository.

On top of all the information on these repositories, you should also include two sections, namely the “Methodology” and “Experiments”. Another required section in your repository is the description of each member’s contribution to the project. These sections should explain and highlight the contributions of your project so we can quickly assess your group. Along with the complete explanation of your project, you should also include any references used, e.g., other repositories, papers, etc. Make sure to indicate how you are using existing ideas and extending them. Later, this might be used as an initial step for a possible paper if your group is selected.

As a rule of thumb, your repository should allow anyone to reproduce your results. Thus, the information contained in your `README.md` should be clear and detailed. To cope with the recent (and fast) use of AI assistants, please also include the AI Usage Cards[21] in your repository as a pdf file. You can find many examples of excellent repositories, methodology, and experiments on the links and papers of this project description. However, please do not limit yourself to them. The information in this document is not a comprehensive list, but a small, compiled collection. All group repositories should be private and only shared/invited with the lecturers and teaching assistants of the course via our teaching user account. After the course is over, you can make your code public. Please remember that the quality of your repository (e.g., content, organization, reproducibility, transparency) will directly reflect on your final grade. Therefore, a higher score on the solved task does not necessarily mean a higher grade. For more details on the grading criteria, please read the next section.

Note that we have created a **README template** for you. Please use this template in your repository and fill out all the sections as described.

Please keep the predictions of your final models on the used train, dev, and test sets in the predictions folder of your repository.

[19] https://neurips.cc/

[20] https://github.com/paperswithcode/releasing-research-code

[21] https://ai-cards.org/

10 Grading Criteria

The final project will be graded holistically. This means we will look at many factors when determining your grade: the creativity, complexity, and technical correctness of your approach, your thoroughness in exploring and comparing various approaches, the strength of your results, the effort you applied, and the quality of your repository, evaluation, and error analysis. Generally, implementing more complicated models represents more effort, and implementing more unusual models (e.g. ones that we have not mentioned in this handout) represents more creativity. You are not required to pursue original ideas, but the best projects in this class will go beyond the ideas described in this handout and may become published works themselves!

For the second part of this project, an aspect of your grade will include your performance relative to the leaderboard as a whole across all tasks. Note that the strength of your results on the leaderboard is only one of the many factors we consider in grading. We focus on evaluating peoples' well-reasoned research questions, explanations, and experiments that evaluate those questions.

There is no pre-defined accuracy (SST, Quora, ETPC), Pearson correlation (SemEval), or BLEU score (ETPC) to ensure a good grade. Though we have run some preliminary tests to get some ballpark scores, it is impossible to say in advance what distribution of scores will be reasonably achievable for students in the provided timeframe. For similar reasons, no pre-defined rule for which of the extensions proposed would ensure a good grade. Implementing a small number of things with good results and thorough experimentation/analysis is better than implementing many things that don't work or barely work. In addition, the quality of your repository, method, and experimentation are essential. We expect you to convincingly show that your techniques are effective and describe why they work (or the cases when they do not work).

Challenging Grades

All students are welcome to question their grades for the exam and the project. After the grade is communicated, the student will have **one week** to schedule a meeting with the tutor to review their grade.

11 FAQ

- What happens if we encounter unforeseen technical challenges with the datasets or model implementations?

  Answer: You can contact your tutors all the time. They won't give you the solution but will help you to overcome those challenges.

- Can I implement a model to solve all tasks at the same time?

  Answer: Yes. This is actually one of the Bonus tasks. Make sure to explain how this model is built and trained!

- Can I use another model to generate synthetic data?

  Answer: Yes, just make sure to explain how it was used.

- Can I include ablation studies and other metrics to support my results?

  Answer: Yes, the more robust your investigation is, the better.

- Can I just replicate a successful solution?

  Answer: You can start with this after implementing the baseline models and investigate this solution for further improvements. For the final project (including improvements), this is not enough since you have to bring your ideas.

- Can we use external libraries for the project?

  Answer: For the first part of the project, only libraries installed by setup.sh are allowed. For the second part, you are free to install additional libraries.

- Are there any specific computational resources provided for training the models?

  Answer: Yes. The computational power and your budget on the GWDG cluster are sufficient for the whole project.

- Are there any limitations on the size or architecture of the models that we can use for our improvements in Part 2 of the project?

  Answer: No. However, you have a limited budget for using the GPUs on the cluster. So, you should be able to make all the improvements with this budget.

- Are we allowed to use LLMs like ChatGPT for the project?

  Answer: Yes. However, it's not sufficient to simply call the API of ChatGPT or to only create synthetic data with those models. Instead, you should systematically investigate benchmarks, weaknesses, etc., of such an approach including a LLM. Every AI assistant use must be disclosed using the AI-Usage Cards.

- My group was composed of 5 people, but some dropped out during the semester. Do I still need to solve all five tasks?

  Answer: Yes, and their improvement in Part 02 as well.

- If too many members left my group, what should I do?

  Answer: There are many alternatives: look for another group, look for other people in the same situation as you and maybe form a new group, talk with the responsible Tutor.

- Can I still pass the course if I missed/failed the written exam?

  Answer: No. You need a minimum passing score in both modules (Lecture and Practical). However, you are more than welcome to continue in the course.

- Are there any minimum requirements for the project?

  Answer: Yes. Each task must contain at least one baseline and one improvement. Keep in mind this is the minimum; it is up to you to pursue an innovative solution, interesting technique, detailed analysis, clear description and discussion, etc. Remember, quantity is not quality, more (shallow) experiments do not necessarily translate into a higher score.

12 Honor Code

Here are some specifically relevant guidelines:

1. Unless you wrote that implementation yourself, you may not use a pre-existing implementation for the minBERT challenge as your starting point.

2. You are not allowed to use pre-trained contextual embeddings (such as ELMO, GPT, etc) for your system. You are allowed to use other pre-existing NLP tools, such as a POS tagger, dependency parser, and coreference module, that are not built on top of pre-trained contextual embeddings. We encourage you to rely on clever solutions not pre-trained models. If you used other pre-trained models, it would not be clear where the improvements would come from.

3. You are free to discuss ideas and implementation details with other teams (in fact, we encourage it!). However, under no circumstances may you look at another team's code or incorporate their code into your project.

4. It is an honor code violation to use the official SST, Quora, SemEval, ETPC training and test data, and their test sets in any way.

5. Do not share your code publicly (e.g., in a public GitHub repo) until after the class has finished.

References

Abien Fred Agarap. Deep Learning using Rectified Linear Units (ReLU), February 2019.

Eneko Agirre, Daniel Cer, Mona Diab, Aitor Gonzalez-Agirre, and Weiwei Guo. *SEM 2013 shared task: Semantic Textual Similarity. In *Second Joint Conference on Lexical and Computational Semantics (*SEM), Volume 1: Proceedings of the Main Conference and the*

Shared Task: Semantic Textual Similarity, pages 32–43, Atlanta, Georgia, USA, June 2013. Association for Computational Linguistics.

Qiwei Bi, Jian Li, Lifeng Shang, Xin Jiang, Qun Liu, and Hanfang Yang. MTRec: Multi- Task Learning over BERT for News Recommendation*. In Findings of the Association for Computational Linguistics: ACL 2022,* pages 2663–2669, Dublin, Ireland, 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.findings-acl.209.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv:1810.04805 [cs],* May 2019.

Samuel Fernando and Mark Stevenson. A Semantic Similarity Approach to Paraphrase Detection. In *Computer Science,* 2008.

Tianyu Gao, Xingcheng Yao, and Danqi Chen. SimCSE: Simple Contrastive Learning of Sentence Embeddings. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 6894–6910, Online and Punta Cana, Dominican Republic, 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.552.

Matthew Henderson, Rami Al-Rfou, Brian Strope, Yun-hsuan Sung, Laszlo Lukacs, Ruiqi Guo, Sanjiv Kumar, Balint Miklos, and Ray Kurzweil. Efficient Natural Language Response Suggestion for Smart Reply, May 2017.

Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Tuo Zhao. SMART: Robust and Efficient Fine-Tuning for Pre-trained Natural Language Models through Principled Regularized Optimization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics,* pages 2177–2190, Online, 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.197.

Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, January 2017.

Ilya Loshchilov and Frank Hutter. Decoupled Weight Decay Regularization, January 2019.

Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP),* pages 3980–3990, Hong Kong, China, 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1410.

Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing,* pages 1631–1642, Seattle, Washington, USA, October 2013. Association for Computational Linguistics.

Asa Cooper Stickland and Iain Murray. BERT and PALs: Projected Attention Layers for Efficient Adaptation in Multi-Task Learning, May 2019.

Chi Sun, Xipeng Qiu, Yige Xu, and Xuanjing Huang. How to Fine-Tune BERT for Text Classification?, February 2020.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, undefinedukasz Kaiser, and Illia Polosukhin. Attention is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems, NIPS'17,* pages 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.

Tianhe Yu, Saurabh Kumar, Abhishek Gupta, Sergey Levine, Karol Hausman, and Chelsea Finn. Gradient Surgery for Multi-Task Learning, December 2020.