CS 6957: NLP with Neural Networks Fall 2023

Mini Project 4

Handed out: November 7, 2023
Due date: November 22, 2023

General Instructions

- Feel free to discuss the mini project with the instructor or the TA.
- Handwritten solutions will not be accepted.
- The homework is due by midnight of the due date. Please submit the mini project on Canvas. You should upload three files: a report with answers to the questions below, a compressed file (`.zip` or `.tar.gz`) containing your code, and two results files.
- Your code should be your own. You may use any libraries which may provide ease of implementation. However, built-in library methods for producing the output directly should not be used. Plagiarism of code will result in a zero.

1 BERT for Sentence/Sentence-Pair Classification

BERT (**?**) is a transformer-based language model that became widely popular a few years back. The promise comes from the fact that it has been pre-trained on large amounts of data. BERT has shown to be, especially, useful since training even a simple linear classifier on BERT representations has shown to have promising results on a wide array of NLP tasks. BERT also offers the ability to tweak its model weights while training this linear classifier. This process is known as "fine-tuning".

In this mini-project, you will be training sentence and sentence-pair classifiers. BERT can classify up to two sentences. For sentence classification, every input sentence (<`sent`>) is prepended by the classifier token(`[CLS]`) and followed by a separator token(`[SEP]`) to look like: "`[CLS]` <`sent`> `[SEP]`". For the sentence-pair classification, task the first sentence is prepended and appended by the classifier and separator tokens respectively. This is then followed by the second sentence which is, in turn, followed by the separator token. Hence, the processed sentence pair looks like: "`[CLS]` <$\texttt{sent}_1$ > `[SEP]` <$\texttt{sent}_2$ > `[SEP]`". The representation corresponding to the `[CLS]` token is, generally, considered a representation for the sentence or sentence-pairs.

The processed inputs are fed to the BERT which produces contextual representations of every input tokens (more technically, sub-tokens). To obtain sentence or sentence-pair

predictions, the representation corresponding to the [CLS] token is fed to a linear classifier which produces a distribution over the output labels.

$$\hat{y} = w.h_{[CLS]} + b$$

where w and b are the weights and bias terms of the linear classifier, and $h_{[CLS]}$ is the contextual representation corresponding to the [CLS] token.

For more details on the BERT models, refer the class slides and Jay Alammar's "The Illustrated BERT" blogpost(https://jalammar.github.io/illustrated-bert/). In this mini-project, you will be familiarizing yourself with HuggingFace's transformers[1] library — a library which houses BERT and other transformer models, and a toolkit on which contemporary NLP hinges.

2 Classifiers for the Mini-Project

In this mini-project, you will be training BERT-based classifiers for two tasks: (i) textual entailment and (ii) sentiment classification.

2.1 Data

As you might have noticed, we have not included any train, dev, or test data in the project files. You will be using the HuggingFace datasets (https://huggingface.co/docs/datasets/index) to obtain these splits. In order to get started, we highly recommend the datasets tutorial: https://huggingface.co/docs/datasets/tutorial. You will be using two datasets, one for the textual entailment task and one for the sentiment classification:

1. **RTE.** Recognizing Textual Entailment (RTE) is a combination of datasets from early textual entailment challenges (earliest being **?**). Textual Entailment is a sentence-pair classification task of assessing whether a certain "premise" text entails a "hypothesis" text. The label is binary, i.e., the premise entails the hypothesis ("entailment") or it does not ("not entailment"). You can get a glimpse of the data at https://huggingface.co/datasets/yangwang825/rte. Here, "text1" corresponds to the premise and "text2" to the hypothesis. You may use the "label" column which assigns 0 to entailments and 1 to non-entailments as targets. To load the RTE dataset with the datasets library, you may use the path "yangwang825/rte".

2. **SST-2.** The Stanford Sentiment Treebank (SST) **?** contains sentiment annotations for movie reviews. The SST-2 variant classifies reviews in a binary (positive or negative) fashion. We will be using a subset of SST-2. You can view the data at https://huggingface.co/datasets/gpt3mix/sst2. The review text is under the column header "sentence" while the label is in the column "label" (0 for negative and 1 for positive). To load the SST-2 dataset with the datasets library, you may use the path "gpt3mix/sst2".

[1] https://huggingface.co/docs/transformers/index

2.2 The Model

Once you have the dataset ready, you are ready to use the BERT models to build your classifiers. Particularly, you will be using two smaller variants of BERT — tiny and mini.

1. **BERT-tiny**[2]. This is the smallest version of the BERT model with two layers of transformer encoders and an embedding dimension of 128. You can use the model path `prajjwal1/bert-tiny`. (We will show how to use this path a bit later.)

2. **BERT-mini**[3]. This is the smallest version of the BERT model with four layers of transformer encoders and an embedding dimension of 256. You can use the model path `prajjwal1/bert-mini`.

Training the BERT classifiers largely differs in two steps from the conventional NLP pipeline :

1. **Tokenization.** There are several aspects of pre-processing and tokenization involved when working with pre-trained transformer models. In §**??**, we mentioned adding the classifier and separator tokens. Further, BERT breaks down tokens into word pieces or sub-words. Since BERT handles a maximum input size of 512 sub-tokens, sequences longer than 512 need to be truncated. Similarly, shorter sequences need to be appropriately padded to enable batching (just like you did in the previous mini-project) and padded tokens should not contribute to any attention or loss computations. Fortunately for us, the transformers library allows us to do all of this in a couple of lines. In this mini-project, you will be using the `BERTTokenizer`[4] or the `AutoTokenizer`[5] class instances to instantiate the tokenizer. Here's an illustration:

```
tokenizer = BERTTokenizer.from_pretrained(<path>)

# Data for sentence classification
sentences = ["sent1", "sent2"]

# Data for pair classification
sentence_pairs_text1 =  ["sent3", "sent4"]
sentence_pairs_text2 =  ["sent5", "sent6"]

# Processed sentences for sentence classification
tokenized_inp = tokenizer(sentences, truncation=True, padding=
                                    True, return_tensors='pt')
# AND,
# Processed sentences for sentence classification
tokenized_inp = tokenizer( sentence_pairs_text1,
                                    sentence_pairs_text2,
                                    truncation=True, padding=
                                    True, return_tensors='pt')
```

[2]https://huggingface.co/prajjwal1/bert-tiny
[3]https://huggingface.co/prajjwal1/bert-mini
[4]`https://huggingface.co/docs/transformers/model_doc/bert#transformers.BertTokenizer`
[5]`https://huggingface.co/docs/transformers/model_doc/auto#transformers.AutoTokenizer`

Here, `truncation=True` ensures sequences longer than 512 are appropriately truncated, `padding=True` pads shorter sequences such that the sequences are of the length of the sequence in the batch. The `return_tensors='pt'` ensures that PyTorch tensors are returned. The `<path>` refers to the model path described above.

2. **Model.** This is the BERT model which you'll be using to obtain the contextual representation. In this mini-project, you will be using the `BERTModel`[6] or the `AutoModel`[7] class instances to instantiate the model. Here's an illustration:

```
model = BERTModel.from_pretrained(<path>)    #Initialization
out = model(**tokenized_inp)
```

Here, the 'out' variable contains various outputs of the model. You may use the `pooler_output` attribute of the 'out' variable to get the `[CLS]` token representation. This token representation can then be passed to a linear layer to obtain predictions. In this mini-project, for each of the two BERT variants, you'll be training a classifier with and without finetuning the BERT model.

2.3 Parameters

1. num_classes = 2. Number of output categories for both tasks
2. $d_{rep} = \{128, 256\}$. size of the embedding dimension (i.e, dimensions of $h_{[CLS]}$ representation)
3. max_eps = 10. Maximum training epochs.

2.4 Hyper-parameters

Just like the previous mini projects, the only tunable hyper-parameter is the learning rate. Train your model using the following learning rates: {0.0001, 0.00001, 0.000001}. Run the training loop for a maximum of 10 epochs. The best model across learning rate and epochs is the one that gets the highest dev accuracy. Feel free to choose the batch size. **Please fix the random seed in your implementation.**

Benchmark information. A training epoch roughly takes under 30 seconds with a batch size of 64 on a Titan X machine. It requires roughly 8 GB of GPU RAM. You can ask for a GPU on CHPC by setting the `--gres=gpu:1` flag.

3 What to Report

1. [20 points] You are provided two files with hidden splits for both tasks. Submit two results CSV files: `results_rte.csv` and `results_sst2.csv` containing the predictions for the two hidden split files. In addition to the original columns, your results file

[6] `https://huggingface.co/docs/transformers/model_doc/bert#transformers.BertModel`
[7] `https://huggingface.co/docs/transformers/model_doc/auto#transformers.AutoModel`

should have a 'prediction' column with the label predictions (0 or 1), a 'probab_0' column containing the probability of predicting the label 0, and a 'probab_1' column containing the probability of predicting the label 1.

2. [20 points] Report the test accuracy of a random classifier for both datasets. Fill up the following tables with the test accuracies for each of the four settings:

	w/o BERT Fine-tuning	with BERT Fine-tuning
BERT-tiny	0.4729241877256318	0.6101083032490975
BERT-mini	0.4693140794223827	0.628158844765343

Table 1: Experiment Results for RTE. Random baseline accuracy: 0.5270758122743683

	w/o BERT Fine-tuning	with BERT Fine-tuning
BERT-tiny	0.500823723228995	0.5992779783393501
BERT-mini	0.5112575507962658	0.5992779783393501

Table 2: Experiment Results for SST-2. Random baseline accuracy: 0.49917627677100496

3. [10 points] Briefly summarize the trends observed from the results in the previous question.

4. [20 points] For the following sentences/sentence pairs, write down the model predictions.

RTE

(a) Premise: The doctor is prescribing medicine.
Hypothesis: She is prescribing medicine.

(b) Premise: The doctor is prescribing medicine.
Hypothesis: He is prescribing medicine.

(c) Premise: The nurse is tending to the patient.
Hypothesis: She is tending to the patient.

(d) Premise: The nurse is tending to the patient.
Hypothesis: He is tending to the patient.

SST-2

(a) Kate should get promoted, she is an amazing employee.

(b) Bob should get promoted, he is an amazing employee.

(c) Kate should get promoted, he is an amazing employee.

(d) Bob should get promoted, they are an amazing employee.

5. [10 points] Briefly comment on your findings from the examples in the previous question.

4 Theory: Exploration of Layer Norm

[20 points] In this question, we will explore layer norms, which is added after (and sometimes before) the self attention and the fully connected networks within the transformer model. The idea of layer norms was originally introduced by **?** as an approach to speed up learning. Layer norms have been subsequently been found to lead to more stable training as well. For more information, refer to this blog post: `https://wandb.ai/wandb_fc/LayerNorm/reports/Layer-Normalization-in-Pytorch-With-Examples---VmlldzoxMjk5MTk1`.

The layer norm is an operator that maps a vector to another vector. Typically, as in the PyTorch implementation [8], the layer norm also has trainable parameters γ and β (both vectors), and is defined as:

$$LayerNorm[x] = \frac{x - \bar{x}}{\sqrt{Var[x] + \epsilon}} * \gamma + \beta$$

Here, the notation $\bar{x}$ denotes the mean of the elements of the vector x, $Var[x]$ denotes their variance and the $*$ operator is the elementwise product between the normalized vector on the left and the elements of γ. ϵ is a small constant added for numerical stability.

For simplicity, for the first two questions below, let us only consider the setting where the parameters γ consists of all ones, and the β is the zero vector.

1. [5 points] If the input to layer norm is a d-dimensional vector, show that the result of layer norm will have a norm of $\sqrt{d}$.

   $$LayerNorm[x] = \frac{x - \bar{x}}{\sqrt{Var[x] + \epsilon}} * \gamma + \beta$$

   Given that thr right-hand-side of $(*)$ will become trivial since it will just multiply the left-hand-side by 1. We focus on the left-hand-side of $(*)$.

   $$\bar{x} = \frac{1}{d}\sum_{i=1}^{d} x_i$$

   $$Var[x] = \sqrt{\frac{1}{d}\sum_{i=1}^{d}(x_i - \bar{x})^2}$$

   If we replace these into the original equation we can observe that the denominator will end up determining the norm, only when $\gamma + \beta$ are trivial.

2. [10 points] If we have two dimensional input vectors (i.e., $d = 2$), show that the layer norm operator will map any vector whose elements are *different* to either the vector $[-1, 1]$ or $[1, -1]$.

3. [5 points] Now suppose the γ and the β can be any real numbers. How will your analysis for the above questions change?

[8] `https://pytorch.org/docs/stable/generated/torch.nn.LayerNorm.html`

References

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://aclanthology.org/N19-1423.

Ido Dagan, Oren Glickman, and Bernardo Magnini. The pascal recognising textual entailment challenge. In *Machine learning challenges workshop*, pages 177–190. Springer, 2005.

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 David Yarowsky, Timothy Baldwin, Anna Korhonen, Karen Livescu, and Steven Bethard, editors, *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1631–1642, Seattle, Washington, USA, October 2013. Association for Computational Linguistics. URL https://aclanthology.org/D13-1170.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.