Multi-label Text Classification using BERT – The Mighty Transformer

Kaushal Trivedi
Jan 28, 2019 · 7 min read ★

. . .

The past year has ushered in an exciting age for Natural Language Processing using deep neural networks. Research in the field of using pre-trained models have resulted in massive leap in state-of-the-art results for many of the NLP tasks, such as text classification, natural language inference and question-answering.

Some of the key milestones have been **ELMo**, **ULMFiT** and **OpenAI Transformer**. All these approaches allow us to pre-train an unsupervised language model on large corpus of data such as all wikipedia articles, and then fine-tune these pre-trained models on downstream tasks.

Perhaps the most exciting event of the year in this area has been the release of **BERT**, a multilingual transformer based model that has achieved state-of-the-art results on various NLP tasks. BERT is a bidirectional model that is based on the transformer architecture, it replaces the sequential nature of RNN (LSTM & GRU) with a much faster Attention-based approach. The model is also pre-trained on two unsupervised tasks, masked language modeling and next sentence prediction. This allows us to use a pre-trained BERT model by fine-tuning the same on downstream specific tasks such as sentiment classification, intent detection, question answering and more.

Okay, so what's this about?

In this article, we will focus on application of BERT to the problem of multi-label text classification. Traditional classification task assumes that each document is assigned to one and only on class i.e. label. This is sometimes termed as multi-class classification or sometimes if the number of classes are 2, binary classification.

On other hand, multi-label classification assumes that a document can simultaneously and independently assigned to multiple labels or classes. Multi-label classification has many real world applications such as categorising businesses or assigning multiple genres to a movie. In the world of customer service, this technique can be used to identify multiple intents for a customer's email.

We will use Kaggle's ***Toxic Comment Classification Challenge*** to benchmark BERT's performance for the multi-label text classification. In this competition we will try to build a model that will be able to determine different types of toxicity in a given text snippet. The types of toxicity i.e. toxic, severe toxic, obscene, threat, insult and identity hate will be the target labels for our model.

Where do we start?

Google Research recently open-sourced the tensorflow implementation of BERT and also released the following pre-trained models:

1. `BERT-Base, Uncased`: 12-layer, 768-hidden, 12-heads, 110M parameters
2. `BERT-Large, Uncased`: 24-layer, 1024-hidden, 16-heads, 340M parameters
3. `BERT-Base, Cased`: 12-layer, 768-hidden, 12-heads , 110M parameters
4. `BERT-Large, Cased`: 24-layer, 1024-hidden, 16-heads, 340M parameters
5. `BERT-Base, Multilingual Cased (New, recommended)`: 104 languages, 12-layer, 768-hidden, 12-heads, 110M parameters
6. `BERT-Base, Chinese`: Chinese Simplified and Traditional, 12-layer, 768-hidden, 12-heads, 110M parameters

We will use the smaller Bert-Base, uncased model for this task. The Bert-Base model has 12 attention layers and all text will be converted to lowercase by the tokeniser. We are running this on an AWS p3.8xlarge EC2 instance which translates to 4 Tesla V100 GPUs with total 64 GB GPU memory.

I personally prefer using PyTorch over TensorFlow, so we will use excellent PyTorch port of BERT from HuggingFace available at https://github.com/huggingface/pytorch-pretrained-BERT. We have converted the pre-trained TensorFlow checkpoints to PyTorch weights using the script provided within HuggingFace's repo.

Our implementation is heavily inspired from the ***run_classifier*** example provided in the original implementation of BERT.

Data representation

The data will be represented by class *InputExample*.

- text_a: text comment
- text_b: Not used

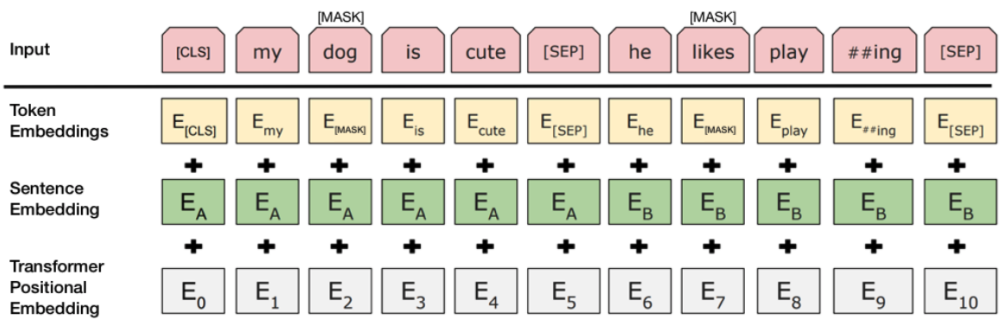
- labels: List of labels for the comment from the training data (will be empty for test data for obvious reasons)

```python
class InputExample(object):
    """A single training/test example for sequence classification."""

    def __init__(self, guid, text_a, text_b=None, labels=None):
        """Constructs a InputExample.

        Args:
            guid: Unique id for the example.
            text_a: string. The untokenized text of the first sequence. For single
            sequence tasks, only this sequence must be specified.
            text_b: (Optional) string. The untokenized text of the second sequence.
            Only must be specified for sequence pair tasks.
            labels: (Optional) [string]. The label of the example. This should be
            specified for train and dev examples, but not for test examples.
        """
        self.guid = guid
        self.text_a = text_a
        self.text_b = text_b
        self.labels = labels
```

input_example.py hosted with ♡ by GitHub view raw

```python
class InputFeatures(object):
    """A single set of features of data."""

    def __init__(self, input_ids, input_mask, segment_ids, label_ids):
        self.input_ids = input_ids
        self.input_mask = input_mask
        self.segment_ids = segment_ids
        self.label_ids = label_ids
```

input_features.py hosted with ♡ by GitHub view raw



We will convert the InputExample to the feature that is understood by BERT. The feature will be represented by class InputFeatures.

- input_ids: list of numerical ids for the tokenised text
- input_mask: will be set to 1 for real tokens and 0 for the padding tokens
- segment_ids: for our case, this will be set to the list of ones
- label_ids: one-hot encoded labels for the text

Tokenisation

BERT-Base, uncased uses a vocabulary of 30,522 words. The processes of tokenisation involves splitting the input text into list of tokens that are available in the vocabulary. In

order to deal with the words not available in the vocabulary, BERT uses a technique called BPE based WordPiece tokenisation. In this approach an out of vocabulary word is progressively split into subwords and the word is then represented by a group of subwords. Since the subwords are part of the vocabulary, we have learned representations an context for these subwords and the context of the word is simply the combination of the context of the subwords. For more details regarding this approach please refer Neural Machine Translation of Rare Words with Subword Unitshttps://arxiv.org/pdf/1508.07909.

P.S. This in my opinion is as important a breakthrough as BERT itself.

Model Architecture

We will adapt *BertForSequenceClassification* class to cater for multi-label classification.

```
class BertForMultiLabelSequenceClassification(PreTrainedBertModel):
    """BERT model for classification.
    This module is composed of the BERT model with a linear layer on top of
    the pooled output.
    """
    def __init__(self, config, num_labels=2):
        super(BertForMultiLabelSequenceClassification, self).__init__(config)
        self.num_labels = num_labels
        self.bert = BertModel(config)
        self.dropout = torch.nn.Dropout(config.hidden_dropout_prob)
        self.classifier = torch.nn.Linear(config.hidden_size, num_labels)
        self.apply(self.init_bert_weights)

    def forward(self, input_ids, token_type_ids=None, attention_mask=None, labels=None):
        _, pooled_output = self.bert(input_ids, token_type_ids, attention_mask, output_all_encod
        pooled_output = self.dropout(pooled_output)
        logits = self.classifier(pooled_output)

        if labels is not None:
            loss_fct = BCEWithLogitsLoss()
            loss = loss_fct(logits.view(-1, self.num_labels), labels.view(-1, self.num_labels))
            return loss
        else:
            return logits

    def freeze_bert_encoder(self):
        for param in self.bert.parameters():
            param.requires_grad = False

    def unfreeze_bert_encoder(self):
        for param in self.bert.parameters():
            param.requires_grad = True
```

bert_modelling.py hosted with ♡ by GitHub view raw

The primary change here is the usage of **Binary cross-entropy with logits** (*BCEWithLogitsLoss*) loss function instead of vanilla cross-entropy loss (*CrossEntropyLoss*) that is used for multiclass classification. Binary cross-entropy loss allows our model to assign independent probabilities to the labels.

The model summary is shows the layers of the model alongwith their dimensions.

```
BertForMultiLabelSequenceClassification(
  (bert): BertModel(
```

```
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(28996, 768)
      (position_embeddings): Embedding(512, 768)
      (token_type_embeddings): Embedding(2, 768)
      (LayerNorm): FusedLayerNorm(torch.Size([768]), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
#       12 BertLayers
      (11): BertLayer(
        (attention): BertAttention(
          (self): BertSelfAttention(
            (query): Linear(in_features=768, out_features=768, bias=True)
            (key): Linear(in_features=768, out_features=768, bias=True)
            (value): Linear(in_features=768, out_features=768, bias=True)
            (dropout): Dropout(p=0.1)
          )
          (output): BertSelfOutput(
            (dense): Linear(in_features=768, out_features=768, bias=True)
            (LayerNorm): FusedLayerNorm(torch.Size([768]), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1)
          )
        )
        (intermediate): BertIntermediate(
          (dense): Linear(in_features=768, out_features=3072, bias=True)
        )
        (output): BertOutput(
          (dense): Linear(in_features=3072, out_features=768, bias=True)
          (LayerNorm): FusedLayerNorm(torch.Size([768]), eps=1e-12, elementwise_affine=True)
          (dropout): Dropout(p=0.1)
        )
      )
    )
    (pooler): BertPooler(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (activation): Tanh()
    )
  )
  (dropout): Dropout(p=0.1)
  (classifier): Linear(in_features=768, out_features=6, bias=True)
)
```

model_summary.py hosted with ♡ by GitHub view raw

1. BertEmbeddings: Input embedding layer
2. BertEncoder: The 12 BERT attention layers
3. Classifier: Our multi-label classifier with out_features=6, each corresponding to our 6 labels

Training

The training loop is identical to the one provided in the original BERT implementation in *run_classifier.py*. We trained the model for 4 epochs with batch size of 32 and sequence length as 512, i.e. the maximum possible for the pre-trained models. The learning rate was kept to 3e-5, as recommended in the original paper.

We had the opportunity to use multiple GPUs. so we wrapped the Pytorch model inside DataParallel module. This allows us to spread our training job across all the available GPUs.

We did not use half precision FP16 technique as for some reason, binary crosss entropy with logits loss function did not support FP16 processing. This doesn't really affect the end result, it simply takes a bit longer to train.

Evaluation Metrics

We adapted the accuracy metric function to include a threshold, which is set to 0.5 as default.

```python
def accuracy_thresh(y_pred:Tensor, y_true:Tensor, thresh:float=0.5, sigmoid:bool=True):
    "Compute accuracy when `y_pred` and `y_true` are the same size."
    if sigmoid: y_pred = y_pred.sigmoid()

    return np.mean(((y_pred>thresh)==y_true.byte()).float().cpu().numpy(), axis=1).sum()
```

accuracy_thresh_metric.py hosted with ♡ by GitHub view raw

```python
from sklearn.metrics import roc_curve, auc

# Compute ROC curve and ROC area for each class
fpr = dict()
tpr = dict()
roc_auc = dict()

for i in range(num_labels):
    fpr[i], tpr[i], _ = roc_curve(all_labels[:, i], all_logits[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area
fpr["micro"], tpr["micro"], _ = roc_curve(all_labels.ravel(), all_logits.ravel())
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])
```

roc_auc.py hosted with ♡ by GitHub view raw

For multi-label classification, a far more important metric is the ROC-AUC curve. This is also the evaluation metric for the Kaggle competition. We calculate ROC-AUC for each label separately. We also use micro-averaging on top of individual labels' roc-auc scores.

I would recommend reading this excellent blog to get a deeper insight on the roc-auc curve.

Evaluation Scores

We ran a few experiments with a few variations but more of less got similar results. The outcome is as listed below:

Training Loss: **0.022**, Validation Loss: **0.018,** Validation Accuracy: **99.31%**

ROC-AUC scores for the individual labels:

toxic: **0.9988**

severe-toxic: **0.9935**

obscene: **0.9988**

threat: **0.9989**

insult: **0.9975**

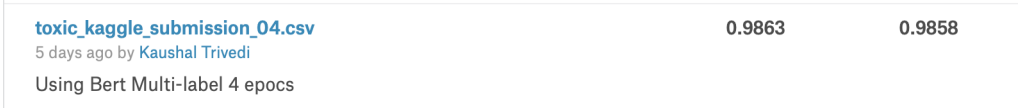
identity_hate: **0.9988**

Micro ROC-AUC: **0.9987**

The result seems to be quite encouraging as we seems to have created a near perfect model for detecting toxicity of a text comment. Now lets see how we score against the Kaggle leaderboard.

Kaggle result

We ran inference logic on the test dataset provided by Kaggle and submitted the results to the competition. The following was the outcome:



We scored **0.9863** roc-auc which landed us within top 10% of the competition. To put this result into perspective, this Kaggle competition had a price money of $35000 and the 1st prize winning score is ***0.9885***.

The top scores are achieved by teams of dedicated and highly skilled data scientists and practitioners. They use various techniques as such ensembling, data augmentation and test-time augmentation in addition to what we have done so far.

Conclusion and Next Steps

We have tried to implement the multi-label classification model using the almighty BERT pre-trained model. As we have shown the outcome is really state-of-the-art on a well-known published dataset. We were able to build a world class model that can be used in production for various industries, especially in customer service.

For us, the next step will be to fine tune the pre-trained language models by using the text corpus of the downstream task using the masked language model and next sentence prediction tasks. This will be an unsupervised task and hopefully will allow the model to learn some of our custom context and terminologies. This is similar technique used by ULMFiT. I will share The outcome in another blog so do watch out for it.

I have shared most of the code for this implementation in the code gist. However I will merge my changes back to HuggingFace's github repo.

I would encourage you all to implement this technique on your own custom datasets and would love to hear some stories.

I would love to hear back from all. Also please feel free to contact me using LinkedIn or Twitter.

. . .

Update

I have made available the jupyter notebook for this article. Note that this is an interim option and this work will be merged into HuggingFace's awesome pytorch repo for BERT.

Jupyter Notebook Viewer

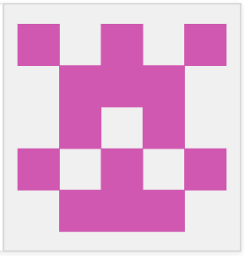
Check out this Jupyter notebook!

nbviewer.jupyter.org

kaushaltrivedi/bert-toxic-comments-multilabel

Multilabel classification for Toxic comments challenge using Bert - kaushaltrivedi/bert-toxic-comments-multilabel

github.com



. . .

References

- The original BERT paper.
- Open-sourced TensorFlow BERT implementation with pre-trained weights on github
- PyTorch implementation of BERT by HuggingFace – The one that this blog is based on.
- Highly recommended course.fast.ai. I have learned a lot about deep learning and transfer learning for natural language processing by following fast.ai.

Machine Learning | Bert | Deep Learning | NLP | AI