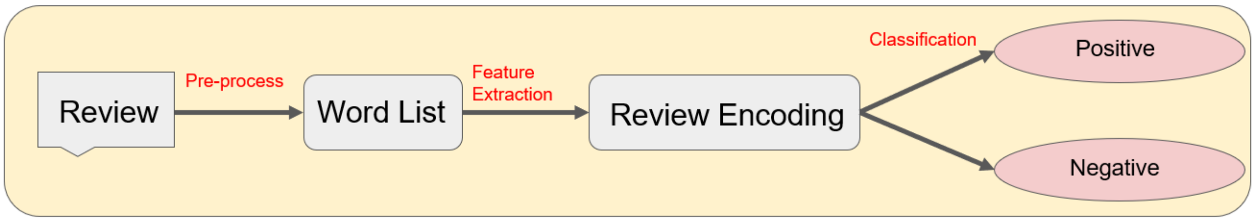
Group 7 **Movie Review Sentiment Analysis** COM 526000 Deep Learning

劉美忻 105061807 ELiu, 邱繼賢 110024516 EdwardChiu, 張藺云 111061530 erhyun January 2023



Goal: Implement a neural network that will read a movie review and classify it as positive or negative. Maximize the F1 score on the test set.

The movie review is 1) pre-processed, 2) the features are extracted, and then 3) the features are used in classification. We experimented with different methods in each of the 3 stages. First, we tried traditional machine learning methods for the feature extraction and classifier, then tried the state-of-the art BERT encoder (bert-base by huggingface), then we 'looked inside' to try to improve on the BERT results by freezing parameters for fine-tuning the classifier and augmenting the feature vector, and we finally tried other BERT models. We also tried the effect of keeping capitalization in pre-processing.

We show our best models first, and then describe our various experiments in the sections following.

**Best Models**

BERT-large-cased gave our best test set F1 score = 94.03841%
BERT-base-uncased gave F1 = 93.3%, and we were stuck there for a while and used it as our basis for the experiments described later

| | **base** | **large** |
|---|---|---|
| uncased | 93.3034 | 93.9276 |
| cased | 92.6927 | 94.0384 |

**Pre-processing**:

First, we read the review text, and 1) lowercase 2) spell out abbreviations 3) remove non-alphabetic characters such as punctuation and symbols. We also removed tags such as <br />

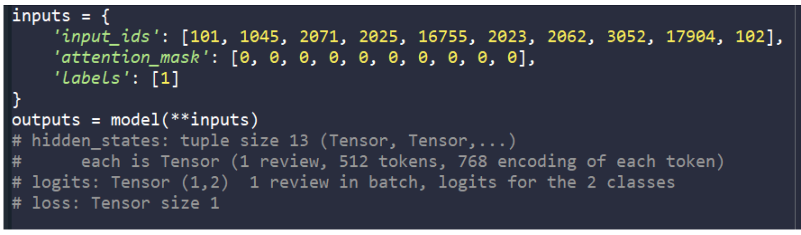
Then, the BERT tokenizer further breaks words into wordpieces (tokens), pads short texts with 0s and truncates long texts to make a 512-long list of wordpiece ids. [CLS] is the special token for the start of a sentence (here we treat the entire review as a long sentence), [SEP] indicates the end of the review (it was also for next-sentence prediction in another task that bert used in pretraining). BERT requires as input: the wordpiece input_ids (each wordpiece has an id in the vocabulary of size ~30,500) and the attention_mask (1 to indicate those tokens are important, and 0 for the paddings). Here is an example:

'I couldn't recommend this more. Breathtaking!!' →

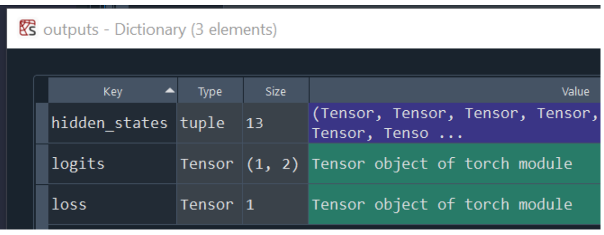
| token wordpieces | [CLS] | i | could | not | recommend | this | more | breath | ##taking | [SEP] | [PAD] |
|---|---|---|---|---|---|---|---|---|---|---|---|
| input_ids | 101 | 1045 | 2071 | 2025 | 16755 | 2023 | 2062 | 3052 | 17904 | 102 | 0 |
| attention_mask | 1 | 1 | 1 | 1 | 1 | | 1 | 1 | 1 | 1 | 0 |

**Feature extraction and classification:**

BERT takes as input the above input_ids and attention_masks, and outputs the loss, logits, and hidden_states. It has the encoding and classifier layers in one model. (In our experiments section, we explored if we could improve default bert by replacing the built-in bert classifier layers with our own.)



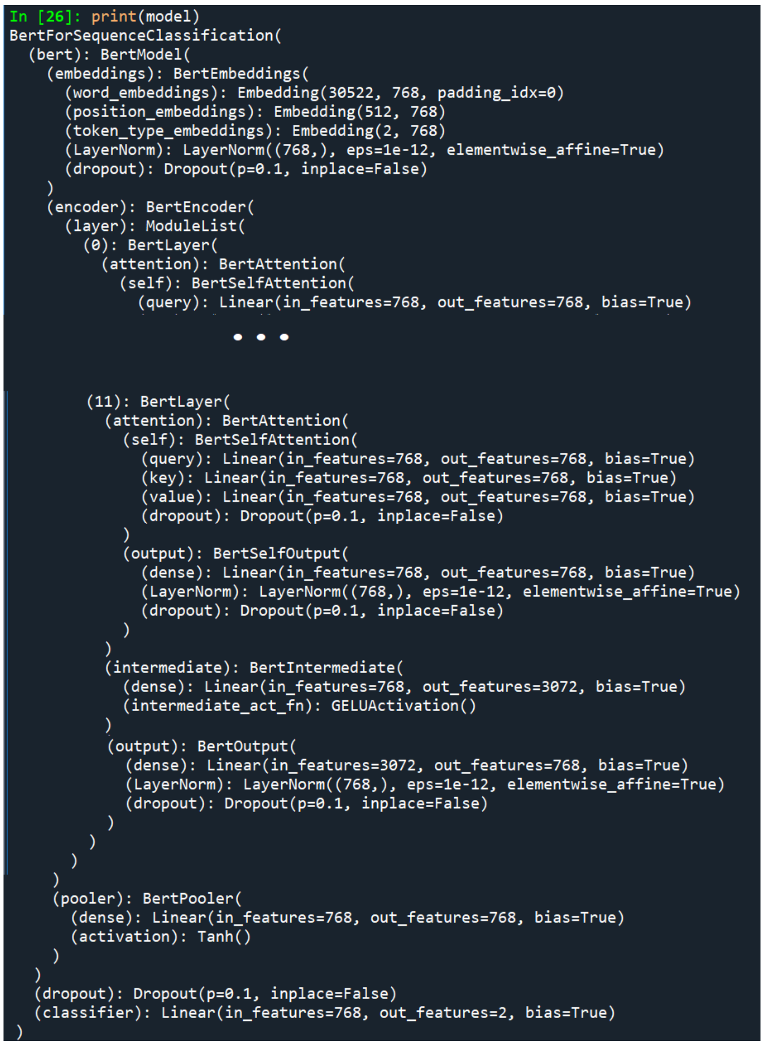
```
inputs = {
    'input_ids': [101, 1045, 2071, 2025, 16755, 2023, 2062, 3052, 17904, 102],
    'attention_mask': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    'labels': [1]
}
outputs = model(**inputs)
# hidden_states: tuple size 13 (Tensor, Tensor,...)
#      each is Tensor (1 review, 512 tokens, 768 encoding of each token)
# logits: Tensor (1,2)  1 review in batch, logits for the 2 classes
# loss: Tensor size 1
```



outputs - Dictionary (3 elements)

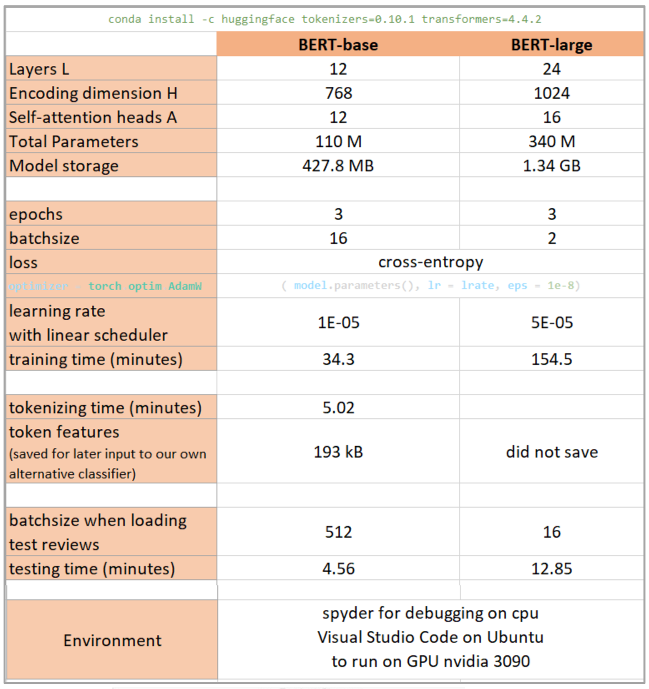
| Key | Type | Size | Value |
|---|---|---|---|
| hidden_states | tuple | 13 | (Tensor, Tensor, Tensor, Tensor, Tensor, Tenso ... |
| logits | Tensor | (1, 2) | Tensor object of torch module |
| loss | Tensor | 1 | Tensor object of torch module |

(Note: in the picture above, there's only 1 review. When training, we loaded reviews in batches of 16.)



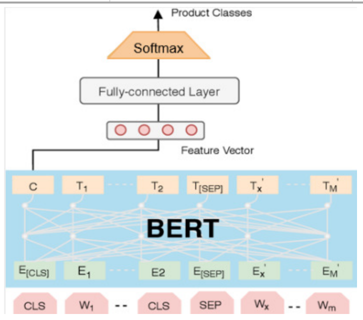
```
In [26]: print(model)
BertForSequenceClassification(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(30522, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (token_type_embeddings): Embedding(2, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
                          • • •

        (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, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (activation): Tanh()
    )
  )
  (dropout): Dropout(p=0.1, inplace=False)
  (classifier): Linear(in_features=768, out_features=2, bias=True)
)
```

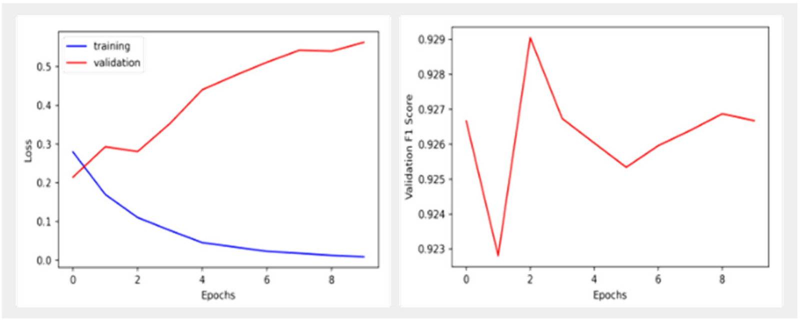


`conda install -c huggingface tokenizers=0.10.1 transformers=4.4.2`

| | BERT-base | BERT-large |
|---|---|---|
| Layers L | 12 | 24 |
| Encoding dimension H | 768 | 1024 |
| Self-attention heads A | 12 | 16 |
| Total Parameters | 110 M | 340 M |
| Model storage | 427.8 MB | 1.34 GB |
| | | |
| epochs | 3 | 3 |
| batchsize | 16 | 2 |
| loss | cross-entropy | |
| `optimizer = torch.optim.AdamW` | `( model.parameters(), lr = lrate, eps = 1e-8)` | |
| learning rate with linear scheduler | 1E-05 | 5E-05 |
| training time (minutes) | 34.3 | 154.5 |
| | | |
| tokenizing time (minutes) | 5.02 | |
| token features (saved for later input to our own alternative classifier) | 193 kB | did not save |
| | | |
| batchsize when loading test reviews | 512 | 16 |
| testing time (minutes) | 4.56 | 12.85 |
| | | |
| Environment | spyder for debugging on cpu<br>Visual Studio Code on Ubuntu<br>to run on GPU nvidia 3090 | |

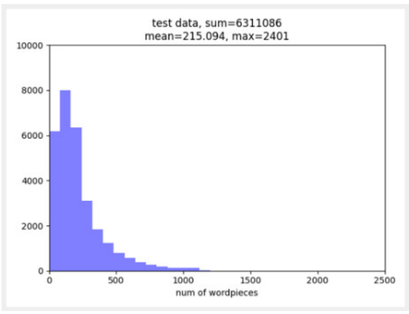


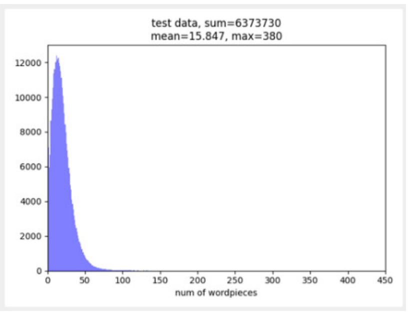
- BERT-base-uncased is the playground for most of our experiments because it's not too big. (BERT-large took over 3x longer to train and over 3x the memory. Above left shows the architecture of huggingface BERT-base-uncased.
- We chose batchsize 16 to train BERT-base because the original authors' papers [1,2] suggest 16 or 32, and our teammember using Colab Pro had resource errors at size 32. For BERT-large, we could only load batchsize 2 when training, before running into memory allocation errors on our lab server. At testing, we loaded reviews in batches due to memory considerations: 512 reviews at a time for BERT-base, and 16 reviews at a time for BERT-large.
- The flow chart at the right visualizes the BERT model. The tokenized review (pink) is passed into BERT: first through an embedding layer and then through 12 deeply-bidirectional transformer layers. The output of the last transformer layer gives the 768-dimension encoding 'C' of the CLS token (CLS stands for classifier). This CLS encoding is the feature vector of the review, and it is passed to the classifier layers which will then give our positive/negative sentiment 2-class logits. The last transformer layer also outputs each word-piece token's 768-dim encodings (Ts).
- Bert was pre-trained for 2 tasks: 1) next-sentence prediction NSP and 2) masked-word prediction. It was necessary to fine-tune BERT end-to-end, since the parameters that give the C feature vector were trained for NSP, and we need to tune them for our different task of binary classification. The deeply bidirectional nature of the attention mechanism in the transformers trained on masked word prediction allows the BERT encoding to understand the context of each word in a long sequence of movie review text. BERT has become the replacement of RNN and LSTM in this sense, since RNNs and LSTMs are very deep and require a lot of training data and time, while the powerful pre-trained BERT model was already trained on a huge corpus including Wikipedia, and can be easily fine-tuned for downstream tasks.



- When doing experiments with BERT-base, we chose to fine-tune end to end for 3 epochs. At left, we held out 25% of the AIdea ~29k dataset for validation, and the generalization gap shows overfitting. There are 110 million parameters, while our set only has around 29000 samples. BERT must have memorized a lot of the information. Our curves show that after 3 epochs, the overfitting hasn't gotten too bad, and the validation F1 score is high. This experiment confirms the suggestions in the papers [1,2], to use 3~4 epochs (not so many as 10 or 20). Also, 1 epoch of BERT-base took around 10 minutes on the GPU, so time was considered as well. After validating, we retrained the chosen model using the entire training 29k dataset (no validation hold-out).

- It is evident that the ratio of tuneable parameters vs. training dataset size leads to overfitting issues. We were under the impression that we could only train using the data provided by AIdea. In hindsight, since BERT itself was already pre-trained with other corpuses, we should've tried using the full imdb dataset (after making sure there's adequate representation of both positive and negative sentiments and no overlap with our AIdea dataset). On the plus side, by using only the AIdea training data, we got a fair comparison of how well different architectures could do under limited data constraints.





- Huggingface only allows 512 tokens max. Shorter reviews were padded, and longer reviews were truncated. We kept the last 512 tokens because the ending sentences usually contain the true sentiment. We wrote code to check the possible effect of truncation. The left-most histogram shows that the average review has 215 wordpiece tokens, and the vast majority are under our 512 limit, so truncating shouldn't be too big an issue. The right histogram shows that on average, each sentence has around 15 wordpieces. In future work, we would add a stage for analyzing the BERT features sentence by sentence per review (instead of an entire review at a time) to catch information missed by our truncation limitation.

**Experiment 1: Traditional methods for feature extraction + classifier. 87% → BERT 93.3%**

Starting the project, we tried these combinations of feature extractor + classifier:

| Feature Extractors | Classifiers |
|---|---|
| • term frequency- inverse document frequency (tf-idf)<br>78,219-dimensional vector/review<br>Each element how often each word from the 78k training set vocabulary appears in the review, downscaled if the term appears across many reviews<br><br>• word2vec : Each word becomes a vector encoding that includes very local context information.<br><br>• Bert : 利用 self-attention 考慮整句評論語義下，將每則 review 轉換成 encoding 向量 C，word tokens 轉換成encoding 向量 T<br><br>** tfidf and SVM were done by sklearn*<br>** Bert details were discussed earlier*<br>** tfidf and w2v details were in our midterm ppt* | • Support Vector Machine (SVM)<br>• RandomForests (RF)<br>• Fully connected neural network (FCN)<br>• Long Short Term Memory (LSTM) |

| **Model** | **F1 score** |
|---|---|
| tf-idf + SVM | 0.8731 |
| w2v + SVM | 0.8472 |
| w2v + RF | 0.8219 |
| w2v+ LSTM | 0.7935 |
| Bert + FCN | 0.9330 |

In the traditional methods (first 5 trials in the table at right), our preprocessing involved lowering case, spelling out abbreviations, removing punctuation, removing symbols and tags, and also the additional step of removing stop-words such as 'and', 'the', etc. We did not remove stop-words before BERT; they are important for semantics context.

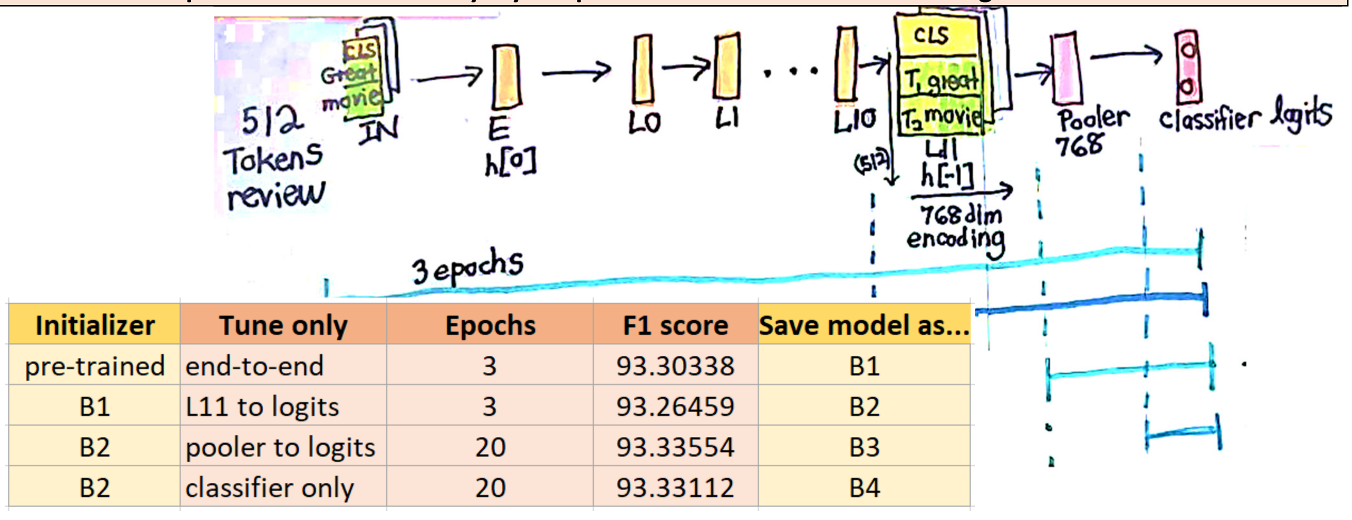
The best traditional method was to use tf-idf with SVM, obtaining a testing set F1 score = 87.31%. The traditional methods have largely been outdated by BERT, which gave a huge improvement to 93.3%. BERT can encode meaningful context information across a long sequence of text, as discussed earlier. The original 87% used the bag of words idea and found frequency of terms. BOW with tfidf does not contain context information, only term frequencies.

Interestingly, word2vec does encode some local context information (at the bi-gram, tri-gram level), but it did not perform better than tfidf. We averaged all the word vectors in the review to represent the review encoding that's passed to the classifier. This may have lost some of the frequency information, and perhaps combining the w2v and tfidf features would've resulted in some improvement. We explore combining BERT word encodings with tfidf on p5.

For the LSTM trial, we passed the w2v word encodings to an LSTM (RNN), but the results were not good. This might be because we were limited in how long the LSTM sequence could be, since length would make the network deep and have vanishing gradient problems. Also, it is slow to train and would require a huge amount of data to tune the many parameters from scratch. BERT solves this problem because it was already trained on a huge corpus, and its transformer attentions gives it the ability to learn context information in long sequences.

Thus, we continued experimenting with our then-best 93.3% BERT-base-uncased model.

**Experiment 2: Freeze early layers' parameters for further fine-tuning → +0.03%**



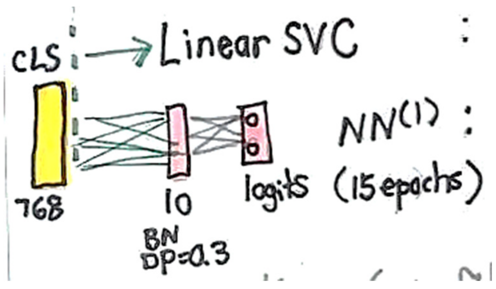
| Initializer | Tune only | Epochs | F1 score | Save model as... |
|---|---|---|---|---|
| pre-trained | end-to-end | 3 | 93.30338 | B1 |
| B1 | L11 to logits | 3 | 93.26459 | B2 |
| B2 | pooler to logits | 20 | 93.33554 | B3 |
| B2 | classifier only | 20 | 93.33112 | B4 |

The pre-trained BERT-base model was for next sentence prediction, so end-to-end tuning is a must since our task is different: binary sentiment classification. Thus, we end-to-end tuned the pre-trained BERT for 3 epochs and saved our model, named B1. Continuing end-to-end tuning until 10 epochs resulted in overfitting, as shown on page 2, since there were too many parameters and not enough data. Thus, we froze the embedding layer and first 11 transformer layers' parameters loaded from model B1, and only continued to train from the last transformer layer L11 to the output logits layer. We did this for 3 epochs, and saved the fine-tuned model as B2. This resulted in test-set F1 score= 93.26459%, a slight drop, perhaps because of the overfitting problem. However, we thought it best to include the last transformer layer in fine-tuning because it might have a lot of impact on the output 'C' review encoding and we hoped this would help fit the model better to our binary classification task.

. Next, we loaded this encoder-tuned-for-classification model (B2) and next fine-tuned only layers from pooler to logits for 20 epochs. These layers are considered the internal 'classifier' part of BERT. This resulted in the best result so far, of 93.33554% test set F1 score, and we saved this model as B3. (We chose 20 epochs because that worked well in our previous homework code involving classifier tuning, considering having 29k training samples and a 2~3 layer classifier network here, and where the homework dataset-to-parameter ratio was even more limited.)

Finally, we wondered if only tuning the very last classifier layer's weights and biases (and freezing everything including the pooler layer, loaded from B2) for 20 epochs would reduce the parameter footprint with less overfitting. The result was 93.33112%, slightly worse than including the pooler layer in fine-tuning.

**Experiment 3: Take BERT-extracted features + our own classifier → just use the BERT-included classifier**



| Classifier | F1 score |
|---|---|
| SVM | 93.07717 |
| NN (1) | 93.33046 |

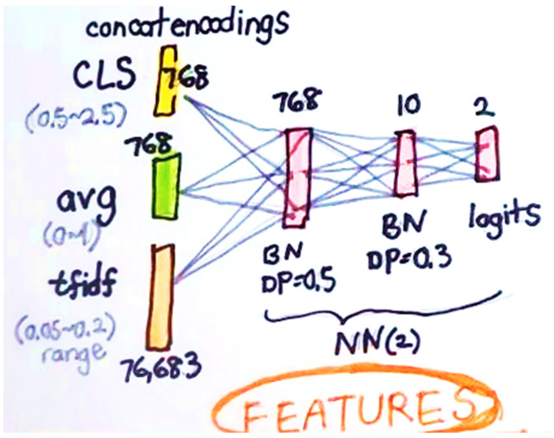
To deal with the overfitting problem, we wanted more control of the fully-connected classifier layers in training. Thus, we extracted BERT's 768-dimensional 'C' encoding for the CLS token (the output of layer 11 in saved model B2), which represents the entire review, and used this as our review feature encoding to be input into a separate classifier. We thus created our own fully-connected classifier, called NN(1), drawn at left in pink. This gave us the flexibility to experiment with regularization techniques, batch-normalization/no batch-norm, and different dropout rates in an attempt to improve generalization ability. The best NN (batch-normalized, dropout=0.3, ReLU activation) we could do was 93.33046% testing F1 score, about the same as (no improvement from) the built-in BERT classifier.

We also experimented with passing BERT's CLS review-encoding feature into a SVM classifier (sklearn's LinearSVC), which resulted in 93.07717% testing F1 score. This indicates that the nonlinearity of the activation function from NN layers is responsible for a quite-significant +0.3% of the F1 score.

No matter what we tried with the classifier, we could not get much improvement, perhaps because we just don't have enough data in the AIdea training set. Thus, we turned our attention to improving the feature extraction.

**Experiment 4: Concatenate extra features + our own classifier → +0.03%**

We study the effect of extra features paired with our own shallow fully-connected NN classifier, and with the traditional SVM classifier (sklearn's LinearSVC, strictly linear hyperplane that splits the feature dimension space into 2 for binary classification). The first two rows are reprinted results from earlier experiments. Note that BERT's powerful semantically-rich encoding was responsible for a whopping +5.7% increase (87.31→ 93.08) in the test-set F1 score, compared to the BOW tfidf feature. (1) refers to the 1-hidden-layer classifier 'NN(1)' drawn in the Experiment 3 diagram above, and (2) refers to the 2-hidden-layer 'NN(2)' classifier drawn below. We used ReLU activation, batchnorm, and the dropout rates shown for our fully-connected NN classifiers.



| Features | Classifier | |
|---|---|---|
| | NN | LinearSVC |
| tfidf (76,683) | | 87.31214 |
| CLS (768) | 93.3046 (1) *15 epochs*<br>93.33554 (bert out) | 93.07717 |
| CLS+tfidf (77,451) | 93.29354<br>*15 epochs* | 93.18087 |
| CLS+avg (1,536) | | 93.13027 |
| CLS+tfidf+avg (78,219)<br>*with scaling* | 93.33289 (2)<br>*9 epochs* | 93.14432 |
| CLS+tfidf+avg<br>*no scaling* | 93.30568 (2)<br>*9 epochs* | 92.48946 |

We tried to make stronger features by concatenating more information together:

1) The yellow 'CLS' is the 768-dimensional review-encoding, output by the last transformer layer of the BERT encoder saved in model B2 (described on page 4).
2) Next, we also took BERT's 512 wordpiece encodings 'Ts' of the review and averaged them together to get the 'avg' feature shown in green above left. This is similar to how (page 3, Experiment 1, 2nd row) w2v gives each word an encoding and we average all the word encodings in a review to output to a classifier. Except here, BERT gives a much more powerful wordpiece encoding that holds long-term semantics meaning.
3) We also concatenated with the orange tfidf feature vector to capture frequency information.

The last row of the table on the bottom right of page 5 shows the result of using the 78,219-dimensional direct-concatenation of all 3 features. Next, we tried scaling the features before concatenating:

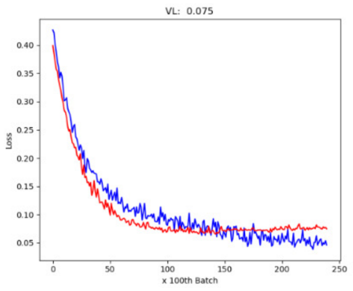
A) 'CLS' (BERT review encoding feature vector values) are scaled to be in 0.5~2.5. They must be the most powerful feature encoding information from BERT, so they were given more weight compared to the other features. This would help with SVM because those dimensions would dominate the hyperplane-tuning. For NN, we hoped larger neuron inputs well inside the ReLU active region would make the classifier pay more attention to those features.

B) 'avg' of BERT word-piece encodings are scaled to be in 0~1. We guessed they would be second-important compared to the 'CLS' feature.

C) 'tfidf' was left as-is, since the normalized frequencies are always in 0~1, and they are very tiny anyway, mainly around 0.05~0.2. (Although it dominates in number of dimensions, those elements are quite small)

The effect of scaling the features is shown in the second-to-last row (chart at bottom of page 5). For SVM, this had a direct impact of increasing F1 from 92.4 to 93.1, a significant increase of +0.65%, which confirms our prediction in A) above. However, the NN classifier with and without scaling both had F1 around 93.3%, with a slight +0.03% with scaling. This interestingly confirms that the weights and biases of the NN automatically learn which features are more important for classification during training.

With scaling, we tried different combinations of the 3 feature vectors to get the 3rd and 4th rows. With our own neural network classifier, we improved the F1 score from 93.3046% for CLS-only feature, to 93.33289% for CLS+avg+tfidf feature., a modest +0.03% improvement. However, the original end-to-end BERT encoder with its built-in classifier already gave 93.33554% at best, so our extra features didn't really help from a practical viewpoint.

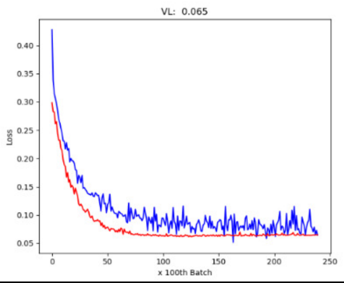


Those results were based on training on the full 29k dataset.

o We decided to train 9 epochs for CLS + tfidf + avg with classifier NN2 by looking at when overfitting started, as in the graph at left. With 10% validation hold-out, the training (blue) and validation(red) curves are shown.

o We got a similar curve for CLS + tfidf with classifier NN2, so it was also trained for 9 epochs

• The curve at right is for CLS with classifier NN1. No overfitting is indicated, so we trained for 15 epochs.



Finally, we decided what we needed was a more powerful BERT feature encoder…

## Experiment 5: Use different BERT models → + 0.7%

| | base | large |
|---|---|---|
| uncased | 93.3034 | 93.9276 |
| cased | 92.6927 | 94.0384 |

The table is reprinted from page 1. We had explored the inside of BERT and conducted many experiments with the not-too-big BERT-base-uncased model. Now, we decided to try the gargantuan BERT-large models. Having been stuck at 93.3% for a good while, just switching to BERT-large-uncased improved F1 by a nice 0.6% to 93.9276%. (This was at the expense of over 3x the training time and 3x the memory requirements. We ran into memory allocation errors and had to reduce the batch size.)

Next, we considered if keeping capitalization would help, so we removed the pre-processing step of lowercasing everything, and tried BERT-base-cased and BERT-large-cased models. For the large model, we improved F1 by another 0.1% to our final best model of 94.0384%. Interestingly, capitalization reduced the BERT-base accuracy quite a bit. We suspect this may be because the complexity of BERT-large allows it to recognize the extra patterns capitalization introduces, but BERT-base might instead overfit. BERT-large probably had the capacity to notice starts of sentences based on capitalization (since we removed punctuation marks). Just as it's easier for a human to read and understand text when we know where each sentence starts, this must have made it easier for BERT-large to extract information.

## Conclusion

• The main power of our final model is in BERT's pre-trained encoder. Pre-training already did the heavy lifting of training on huge databases of BooksCorpus and English Wikipedia ( 1 week on 8 GPUs). Thus, the encoder can already recognize how to get semantics information from English text. These powerful features can then be paired with a simple classifier.

- BERT is taking the place of traditional features like BOW+tfidf and word2vec.
- Transformers are replacing RNNs and LSTMs. Fine-tuning the pre-trained BERT model is inexpensive compared to training a deep network from scratch. It is useful when we have a limited training dataset, as in our case.
- Further concatenating BERT features with tfidf features only gave very slight improvement.
- Overfitting was an issue because of the complexity of BERT compared to our limited AIdea training set. We tried to freeze earlier layers and fine-tune just the later layers' parameters in an attempt to reduce number of parameters.
- We also tried regularization techniques. In the end, the main problem was not enough data. Finding more training data from imdb would have helped a lot.
- Switching from BERT-base to BERT-large helped a bit with improving the testing F1 score. However, this came at quite the expense of training time and memory requirements.
- A further step for future work would be to combine other good-performance models and use voting or ensemble learning.

## References

[0] Huggingface BERT pytorch implementation https://huggingface.co/docs/transformers/installation

(their documentation is a bit confusing)

[1] Bert google paper https://arxiv.org/pdf/1810.04805.pdf

[2] Attention is all you need https://arxiv.org/pdf/1706.03762.pdf

[3] Traditional methods ideas https://iust-projects.ir/post/pr01/

[4] Introductory BERT articles

https://medium.com/@jakubvonovsk/sentiment-analysis-with-bert-in-pytorch-7905d7f1e618

[4] How to start coding BERT with pytorch (the first 2 helped a lot with framework)

https://www.kaggle.com/code/chayan8/sentiment-analysis-using-bert-pytorch/notebook

https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

https://colab.research.google.com/drive/15hMu9YiYjE_6HY99UXon2vKGk2KwugWu#scrollTo=CwOGtRWHVaVF

https://github.com/vonsovsky/bert-sentiment/blob/master/model.py

[5] How to freeze BERT layers

https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/

https://jimmy-shen.medium.com/pytorch-freeze-part-of-the-layers-4554105e03a6#:~:text=In%20PyTorch%20we%20can%20freeze,setting%20the%20requires_grad%20to%20False

[6] To understand transformers better… (future reading)

https://coaxsoft.com/blog/building-bert-with-pytorch-from-scratch

https://neptune.ai/blog/how-to-code-bert-using-pytorch-tutorial

https://towardsdatascience.com/illustrated-self-attention-2d627e33b20a

https://jalammar.github.io/illustrated-transformer/

https://github.com/Nielspace/BERT/blob/master/BERT%20Text%20Classification%20fine-tuning.ipynb