

NLP Assignment 3 - WordWeavers


1. We imported the 'Bert-base-uncased' model from Hugging Face.


2. In this work, we denote the number of layers (i.e., Transformer blocks) as L , the hidden size as H , and the number of self-attention heads as A .³ We primarily report results on two model sizes: **BERT_{BASE}** ($L=12$, $H=768$, $A=12$, Total Parameters=110M) and **BERT_{LARGE}** ($L=24$, $H=1024$, $A=16$, Total Parameters=340M).

Number of Parameters for BERT-base from the paper ~ 110M

Number of Parameters for BERT-base from the code = 110,106,428

Hence, the two values match. The code-implementation based calculation matches the value reported in the paper.

```
 d = 0
for param in pretrained.parameters():
    d+=param.numel()
print(d)
```

 110106428

3. We imported the model, **randomized** its weights and then trained it for 5 epochs on the train split of the WikiText dataset.

This pretrained model is pushed to Hugging Face and can be found here- <insert link>

Steps

1. Prepare the dataset:
 - a. The dataset is cleaned as it contains many empty strings and just heading.
 - b. Dataset after cleaning:

```
(DatasetDict({
  test: Dataset({
    features: ['text'],
    num_rows: 2183
  })
  train: Dataset({
    features: ['text'],
    num_rows: 17556
  })
  validation: Dataset({
    features: ['text'],
    num_rows: 1841
  })
}),
datasets.dataset_dict.DatasetDict)
```

c.

2. Train the tokenizer:

- a. To be able to train our model we need to convert our text into a tokenized format. Most Transformer models come with a pre-trained tokenizer, but since we are pre-training our model from scratch we also need to train a Tokenizer on our data. Transformer models often use subword tokenization, so a tokenizer needed training. The trained tokenizer is pushed on the 🤗 and can be found here [Dhairya/nlp-bert-wordWeavers-tokenizer](https://huggingface.co/Dhairya/nlp-bert-wordWeavers-tokenizer)
- b. Credits: <https://huggingface.co/blog/pretraining-bert>
- c. The tokenized dataset has three element:
 - i. 'input_ids'
 - ii. 'token_type_ids'
 - iii. 'attention_mask'

3. Preprocessing the tokized dataset

- a. Block Size Sequence Dataset:
 - i. BERT usually accepts sequences of 512 tokens.
 - ii. We created a sequence of 255 tokens. We will concatenate the data such that we have a list of 255 tokens for each element in the tokenized dataset. ([CLS] + 255 (Sentence 1) + [SEP] 255 (Sentence 2)).
- b. Next Sentence Prediction (NSP):
 - i. NSP dataset is created such that 50% times sentence 2 comes after sentence 1. Rest 50% is random. To implement this logic we did when index is even the sentence 2 (which is the next sentence) comes after sentence 1 and for the odd index the two sentences are chosen at random.
 - ii. It added one extra feature in the dataset, 'next_sentence_label'
- c. Masked Language Modelling (MLM):

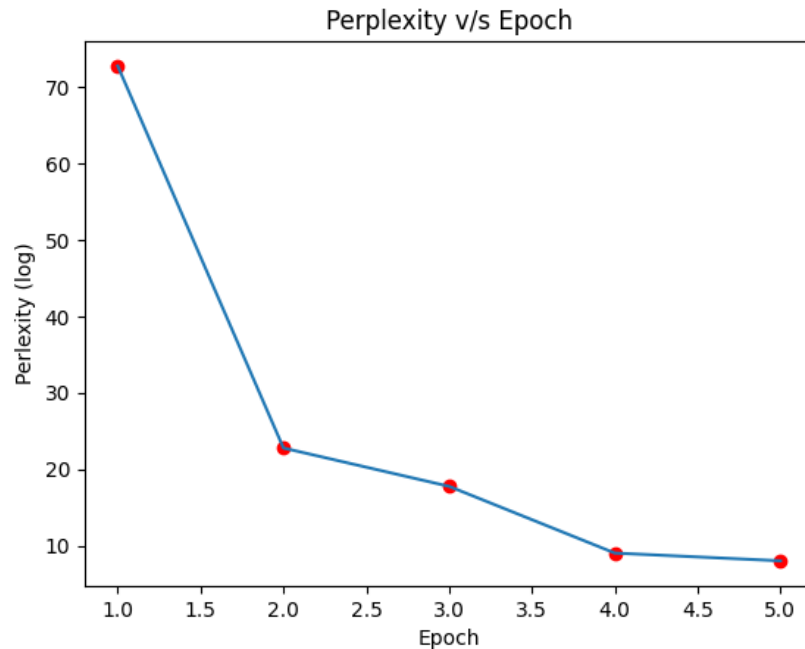
- i. The transformers library offers a streamlined solution for the MLM task. This is done using 'DataCollatorForLanguageModeling' with mlm probability 0.15 (similar in paper). [Reference](#)
4. Pretraing our model:
 - a. It was done for 5 epoch learning rate 0.01 using the Trainer class of hugging face
 - b. Here we loaded the bert architecture with random weights and this was pretrained on our dataset only.
4. We calculated the log (to the base e) of perplexity scores over the test split of the WikiText dataset for every one of the 5 epochs. This is done by using the following reference-
(<https://medium.com/@priyankads/perplexity-of-language-models-41160427ed72>)
The log-perplexity values are equal to the eval_loss over the testing dataset.

The perplexity can be gotten by taking the exponent of the log(perplexity) scores.

- a. The log perplexities of the model after each epoch are shown below-

Epoch	Log(Perplexity)
1	72.70
2	22.77
3	17.76
4	9.04
5	8.0283

- b. Our model is very confused, as we started with random weights thus perplexity is very high initially.
- c. The perplexity is reducing with each epoch as the model's loss on the testing data is decreasing. The model is becoming less confused. The values are still very high even after 5 epochs- $\exp(8.0283) = 3067$. This means that the model is still quite confused.
- d. Due to the small number of epochs, the perplexity is still high. If we pretrain the model for more epochs, we will be getting smaller perplexity scores.



5. Our Pretrained model pushed to Hugging Face hub. It can be found here- <https://huggingface.co/Dhairya/nlp-bert-wordWeavers-pretrained>

We also trained our Tokenizer and pushed it to the Hugging Face hub. It can be found here- <https://huggingface.co/Dhairya/nlp-bert-wordWeavers-tokenizer>

6. a) We imported our pretrained model from Hugging Face and fine tuned it for Classification on the SST-2 Dataset for sentiment analysis.

We created an 80:20 split of the SST-2 Dataset. To ensure a balanced dataset, we performed stratified sampling with random seed=1.

The fine-tuning was performed on the train split.

The following were the steps that were used to fine tune the model on binary classification using the sst-2 dataset:

1. The pretrained model and the tokenizer are first loaded into the system using the transformers library and the AutoModelForSequenceClassification module and the number of classes was set to 2.

2. We then tokenize the dataset using a helper function and the loaded pre-trainer tokenizer.
3. We then configured the training arguments which will be used to initialize the trainer. We appropriately choose the hyperparameters to fine-tune the model in the most efficient way possible.
4. A `compute_metrics` function was defined, which again will be fed into the trainer and will be used to compute the precision, recall, f1 and accuracy scores of the validation set at the end of each epoch during training.
5. We then created a trainer instance, feeding the tokenized train and test dataset into the instance.
6. We then ran the trainer and set up a function in order to print the final returns of the fine-tuned model on the test dataset.

b) We imported our pretrained model from Hugging Face and fine tuned it for Question-Answering on the SQuAD dataset.

We created an 80:20 split of the SQuAD dataset again by using stratified sampling on the titles and combining the train and validation dataset. The fine-tuning was performed on the train split.

The following steps were used to fine-tune the pre-trained model on SQUAD for context based question answering:

1. We first tokenize the data by passing it through our loaded pre-trained tokenizer.
2. Since the dataset does not contain the end token indices of the answers, we calculate and add them to the answers list using the start index and the answer sentence.
3. We correct any off-set errors that might be there in the dataset and remove any data points which have missing questions or answers.
4. We define a dataset trainer argument(hyperparameter) including the number of epochs, learning rates etc and feed it into the trainer.
5. Lastly we run a preliminary evaluation on the validation data set using the trainer instance.

7. Metrics Calculated over the Test Split

a) Classification on SST2-

These results are obtained by evaluating our fine tuned model on the fully unseen Validation Split of the SST2 dataset. The Test split was corrupted with all labels marked as -1 in a 0 vs 1 classification problem. So, we went ahead with evaluation on the Validation set.

```
Accuracy: 0.5092
Precision: 0.2593
Recall: 0.5092
F1 Score: 0.3436
```

Metric	Score
Accuracy	0.5092
Precision	0.2593
Recall	0.5092
F1 Score	0.3436

The above metrics are computed by taking the average of class-wise scores for these metrics. The weighted average is on the basis of distribution of each class in the evaluation dataset.

b) Question-Answering on SQuAD

Evaluation on 400 examples from Test Set

The F1 score, BLEU, METEOR and Exact Match scores are listed below-

```
✓ 0s [?] print(f'F1 Score: {f1_score:.4f}')  
print(f'BLEU Score: {bleu_score:.4f}')  
print(f'Meteor Score: {meteor_score:.4f}')  
print(f'Exact Match Score: {exact_match:.4f}')
```

➡ F1 Score: 0.0092
BLEU Score: 0.0118
Meteor Score: 0.0195
Exact Match Score: 0.0075

Metric	Score
F1	0.0092
BLEU	0.0118
METEOR	0.0195
Exact Match	0.0075

The Rouge-1, Rouge-2 and Rouge-L scores are shown below

Metric	Recall	Precision	F1
Rouge-1	0.0919	0.0444	0.046
Rouge-2	0.0341	0.0107	0.0130
Rouge-L	0.0919	0.0444	0.004

The squad_v2 metrics are shown below

Metric	Score
exact	0.0
f1	2.7265
total	400
HasAns_exact	0.0
HasAns_f1	2.7265
HasAns_total	400
best_exact	0.0
best_f1_thresh	0.0
best_f1	2.7265

Evaluation on Full Test Set (18.000+)

The F1 score, BLEU, METEOR and Exact Match scores are listed below-

Metric	Score
F1	0.0024
BLEU	0.0174
METEOR	0.0244
Exact Match	0.0013

The Rouge-1, Rouge-2 and Rouge-L scores are shown below

Metric	Recall	Precision	F1
Rouge-1	0.0911	0.0530	0.0525
Rouge-2	0.0322	0.0158	0.0172
Rouge-L	0.0896	0.0518	0.0513

The squad_v2 metrics are shown below

Metric	Score
exact	0.0
f1	3.2668
total	18550
HasAns_exact	0.0
HasAns_f1	3.2668
HasAns_total	18550
best_exact	0.0
best_f1_thresh	0.0
best_f1	3.2668

8. We imported our pre trained and fine tuned models from Hugging Face, and instantiated our models as follows-

```
from transformers import BertForPreTraining
from transformers import AutoModelForSequenceClassification
from transformers import AutoModelForQuestionAnswering

pretrained = BertForPreTraining.from_pretrained("Dhairya/nlp-bert-wordWeavers-pretrained")
classi = AutoModelForSequenceClassification.from_pretrained("Dhairya/bert-wordweavers-sst2-v5")
squad = AutoModelForQuestionAnswering.from_pretrained("Dhairya/bert-wordweavers-squad")
```

Number of Parameters of Pretrained Model = 110,106,428 = X

```
[▶] d = 0
    for param in pretrained.parameters():
        d+=param.numel()
    print(d)
```

➞ 110106428

Number of Parameters after fine tuning for Classification = 109,483,778 = Y

```
[25] p = 0
      for param in classi.parameters():
          p+=param.numel()
      print(p)
```

109483778

Number of Parameters after fine tuning for Question-Answering = 108,893,186 = Z

```
✓ 0s [▶] t = 0
      for param in squad.parameters():
          t+=param.numel()
      print(t)
```

➞ 108893186

In both the fine tuning tasks (Classification and Question-Answering), the number of parameters in the pretrained model are more than the number of parameters in the fine tuned model. A detailed explanation of this is provided in 10.b

X>Y and X>Z.

9. Pushed the Classification fine-tuned model to Hugging Face-
<https://huggingface.co/Dhairya/bert-wordweavers-ft-sst2>

Pushed the Question-Answering fine-tuned model to Hugging Face -
<https://huggingface.co/Dhairya/bert-wordweavers-ft-squad>

10. a) Why does the model perform good/bad?

The reasoning for high perplexity in the case of pretraining has already been described in q4. This high perplexity (bad performance) occurs because we have randomly initialized the weights and trained for very few epochs too (just 5).

In the case of the classification task, we observe low precision, recall, accuracy and F1 score values. This is because the performance of downstream tasks depends heavily on how well the model has been pre-trained. Due to poor knowledge and insufficient training of the LM, the downstream performance on classification is also bad. Also, due to computational cost, we have run the finetuning for only 5 epochs. This is too little to expect very high performance.

For the case of classification, we observe low accuracy and precision, recall as well. This is because of all the reasons related to random initialization, insufficient pretraining and insufficient finetuning outlined above.

Question-Answering is a hard task, even for a trained model. High-performing models need the ability to pick out the correct answer from a very large number of spans in the context- this makes question-answering hard, in general.

We have gotten poor performance on our fine tuned model as well. Again, the performance of downstream question-answering tasks depends heavily on how well the pre trained model has been trained. Due to poor knowledge and insufficient training of the LM, the downstream performance on question answering is also bad. Again, due to computational cost, we have run the finetuning for only 5 epochs. Again, this is too little to expect very high performance.

For the case of question-answering, we observe higher Rouge-1 scores as compared to Rouge-2 scores in both tests- (400 and 18550). This is expected, as fewer bigrams will overlap as compared to unigrams. We observe low Exact Match, BLEU and METEOR scores for the reasons outlined above. The performance is better for 400 vs 18550 test points as more answers are being reported wrong in the case of greater number of test points.

b) Inference from the number of parameters differing between pre training and fine tuning?

To understand why the number of parameters in the pretrained model (X) are more than the number of parameters in the Classification-fine tuned model (Y) and Question-Answering fine tuned model (Z), we need to look at their architectures as well as the task for which they are trained.

Pretrained Model-

```
print(pretrained)

BertForPreTraining(
  (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(
        (0-11): 12 x 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()
    )
  )
  (cls): BertPreTrainingHeads(
    (predictions): BertLMPredictionHead(
      (transform): BertPredictionHeadTransform(
        (dense): Linear(in_features=768, out_features=768, bias=True)
        (transform_act_fn): GELUActivation()
        (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      )
      (decoder): Linear(in_features=768, out_features=30522, bias=True)
    )
    (seq_relationship): Linear(in_features=768, out_features=2, bias=True)
  )
)
```

The BertPooler module applies a linear layer in front of the [CLS] token's last-layer activation. This module has $768 \times 768 + 768$ parameters. It takes the last-layer activation of the [CLS] token and multiplies it by a weight matrix and adds a non-linearity. The output of this module is then used in the (seq_relationship) module for the Next-Sentence Prediction task (2 possible answers). There are additional parameters for the Masked Language Modeling Task as well. The module called BertPredictionHeadTransform is used in pre-training for the MLM task. It takes the last-layer activation of the tokens (768 hidden-state), and applies a linear transformation, followed by a mapping to the vocabulary space (30522) because we need to predict a token. Hence because of the nature of the NSP and MLM tasks in pre-training, these are the additional parameters required.

Fine tuned (Classification)

```
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(
        (0-11): 12 x 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)
)
```

In this case, all the modules except for the “BertPretrainingHeads” are common. We need a BertPooler here too because the [CLS] token’s last-layer activation is later used for classification (positive or negative). But we do not need to perform Masked Language Modeling anymore. Hence, we don’t require the “BertPredictionHeadTransform” module anymore and this reduces the number of parameters greatly. After pooling, we still need a Classifier network to make the prediction of positive or negative (2 classes), but the parameters here are the same as those in the (seq_relationship) module--hence, the difference is contributed mainly by the absence of the parameters required for MLM task.

So, we end up using the weights up to the BertPooler, remove the BertPreTrainingHead (as we don’t need it anymore), and add a Linear Classifier for 2 classes in front. Hence, the number of parameters in this fine tuned model are lesser than those of the pretrained model.

Hence $X > Y$.

Fine tuned (Question-Answering)

```
print(squad)

BertForQuestionAnswering(
  (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(
        (0-11): 12 x 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)
          )
        )
      )
    )
  )
  (qa_outputs): Linear(in_features=768, out_features=2, bias=True)
)
```

In the case of Question-Answering, we need to predict the probability of each token either being a start or end token of the answer span. We do not need access to the [CLS] token's last-layer activation. Hence, there is no BertPooler module that we use here. This decreases the number of parameters as compared to the pretrained model. Also, there is no need for a "BertPredictionHeadTransform" module anymore as we do not need to perform Masked Language Modeling anymore. We only need a network to take as input the last-layer activations of each token, and then predict the probability of that token being the start or end token. Hence, there is only one "qa_outputs" layer which is used for every token.

So, we end up using the weights up to the encoder stack, remove the BertPreTrainingHead (as we don't need it anymore) and BertPooler (we don't need it anymore), and add a Linear layer for 2 output (start and end probability) in front. This reduces the number of required parameters as compared to the pretrained model.

Hence $X > Z$.

a) Github Repository (with all notebooks) -

<https://github.com/DhairyaShah981/nlp-assignment-wordweavers>

b) Evaluation can be done by running the Evaluation.ipynb notebook (added to the above repository)

Work Distribution

1. Sukruta Midigeshi - Wrote the code for finetuning for question-answering in SQUAD.
2. Chirag Sarda - Wrote the code for pretraining the bert model, and calculated the perplexities with each epoch.
3. Dhairya Shah - Helped in writing the code for next sentence prediction in pretraining BERT. Worked on setting up GCP GPUs to optimize the running time.
4. Rahul Chembakasseril - Calculated the parameters and documented understanding and the changes in number of parameters after fine tuning, and the differences between finetuning and pre training, helped analyze the results obtained from evaluation of the fine tuned models on the test splits.
5. Arun Mani - Wrote the code for fine tuning for classification in SST2 dataset.
6. Shubh Singhal - Wrote the code for evaluation of the fine tuned models.
7. Yashraj Deshmukh - Helped write the code for evaluation of the finetuned models.
8. Mihika Jadhav - Helped make the plots and charts for the results obtained from evaluation of fine tuned models.
9. Bhavesh Jain - Helped analyze the results obtained from evaluation of fine tuned models.