Final Project Deliverables

November 10, 2024

Since I am doing the NLP and none of my 3 paper give me a github repo to reference so I am using the BERT pre-trained model to help with me. I have the own training method and also I am using the trainer function to help with the fine-tune of my model. And since BERT is a pre-trained model that trained with a insanely huge datasest, so when I am training on specific small set, some responses are really hard to change from the pre-trained model. And due to the limitation of computer resource and dataset size, the performance of the intended BERT may not be that ideal, so I compare the difference of output on BERT to implies that when we get enough dataset and computer source the model will work as intended. For all of the model, you can download them here, so you don't need to wait for days for them to finish. Important: Remember to unzip. https://drive.google.com/drive/folders/147NJfxJCJ7RkCEabP-zTO7GlbTitVMzt?usp=sharing

[1]:
```
import warnings
warnings.filterwarnings('ignore')
warnings.simplefilter('ignore')
```

[2]:
```
import torch
from datasets import load_dataset
import torch.nn.functional as F
import random
import math
import numpy as np
from torch.utils.data import DataLoader
from tqdm import tqdm
from transformers import BertTokenizer, DataCollatorForLanguageModeling,␣
 ↪BertForMaskedLM, BertForNextSentencePrediction, Trainer, TrainingArguments,␣
 ↪DataCollatorForLanguageModeling, AdamW, get_linear_schedule_with_warmup
```

[3]:
```
# Check If I can use GPU or not
print(f'Can I can use GPU now? -- {torch.cuda.is_available()} \nThe Graphics␣
 ↪Card Model is {torch.cuda.get_device_name(0)}')
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
Can I can use GPU now? -- True
The Graphics Card Model is NVIDIA GeForce RTX 4080
```

This Part below is pre-trained BERT model

[4]:
```
# load bert model
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
```

```
model = BertForMaskedLM.from_pretrained("bert-base-uncased")
model.to(device);
```

```
BertForMaskedLM has generative capabilities, as `prepare_inputs_for_generation`
is explicitly overwritten. However, it doesn't directly inherit from
`GenerationMixin`. From v4.50 onwards, `PreTrainedModel` will NOT inherit from
`GenerationMixin`, and this model will lose the ability to call `generate` and
other related functions.
  - If you're using `trust_remote_code=True`, you can get rid of this warning by
loading the model with an auto class. See
https://huggingface.co/docs/transformers/en/model_doc/auto#auto-classes
  - If you are the owner of the model architecture code, please modify your
model class such that it inherits from `GenerationMixin` (after
`PreTrainedModel`, otherwise you'll get an exception).
  - If you are not the owner of the model architecture class, please contact the
model code owner to update it.
Some weights of the model checkpoint at bert-base-uncased were not used when
initializing BertForMaskedLM: ['bert.pooler.dense.bias',
'bert.pooler.dense.weight', 'cls.seq_relationship.bias',
'cls.seq_relationship.weight']
- This IS expected if you are initializing BertForMaskedLM from the checkpoint
of a model trained on another task or with another architecture (e.g.
initializing a BertForSequenceClassification model from a BertForPreTraining
model).
- This IS NOT expected if you are initializing BertForMaskedLM from the
checkpoint of a model that you expect to be exactly identical (initializing a
BertForSequenceClassification model from a BertForSequenceClassification model).
```

[5]:
```
def print_prediction(model_use, sentence):
    inputs = tokenizer(sentence, return_tensors="pt")
    print(inputs.input_ids)
    inputs.to(device)
    with torch.no_grad():
        outputs = model_use(**inputs)
        predictions = outputs.logits
    mask_token_index = torch.where(inputs["input_ids"] == tokenizer.
 ↪mask_token_id)[1]
    predicted_token_id = predictions[0, mask_token_index].argmax(axis=-1)
    predicted_word = tokenizer.decode(predicted_token_id)
    print(f"Original sentence: {sentence}")
    print(f"Predicted word: {predicted_word}")
```

Some explanation on the input_ids, for the tensor with value 101, it is [CLS] means the start of the sentence, for the tensor with value 102 it is [SEP] means the sentence separator, for the tensor with value 103, it is [MASK], which is the word being hidden. And that is what we are going to predict.

[6]:
```
# test with the pre-trained model
sentence = "I am happy today so I will [MASK] video games."
print_prediction(model, sentence)
```

```
tensor([[ 101, 1045, 2572, 3407, 2651, 2061, 1045, 2097,  103, 2678, 2399, 1012,
          102]])
Original sentence: I am happy today so I will [MASK] video games.
Predicted word: play
```

[7]:
```
sentence = "The committee decided to [MASK] the proposal during the session."
print_prediction(model, sentence)
```

```
tensor([[ 101, 1996, 2837, 2787, 2000,  103, 1996, 6378, 2076, 1996, 5219, 1012,
          102]])
Original sentence: The committee decided to [MASK] the proposal during the
session.
Predicted word: consider
```

[8]:
```
sentence = "The scientist aims to [MASK] a deeper understanding of the␣
 ↪phenomenon."
print_prediction(model, sentence)
```

```
tensor([[ 101, 1996, 7155, 8704, 2000,  103, 1037, 6748, 4824, 1997, 1996, 9575,
         1012,  102]])
Original sentence: The scientist aims to [MASK] a deeper understanding of the
phenomenon.
Predicted word: gain
```

[9]:
```
sentence = "During the late 1970s , plans [MASK] to create a new highway link␣
 ↪on the south side of Lake Simcoe to connect Highway 400 and Highway 12"
print_prediction(model, sentence)
```

```
tensor([[  101,  2076,  1996,  2397,  3955,  1010,  3488,   103,  2000,  3443,
          1037,  2047,  3307,  4957,  2006,  1996,  2148,  2217,  1997,  2697,
         21934, 16288,  2000,  7532,  3307,  4278,  1998,  3307,  2260,   102]])
Original sentence: During the late 1970s , plans [MASK] to create a new highway
link on the south side of Lake Simcoe to connect Highway 400 and Highway 12
Predicted word: developed
```

[10]:
```
# Load Datasets, I use wikitext here for now
dataset = load_dataset("wikitext", "wikitext-2-raw-v1", split="train")

def tokenize_function(examples):
    return tokenizer(examples["text"], padding="max_length", truncation=True,␣
 ↪max_length=128)
```

For this part of the code, I use the trainer utilities for easier training, I will have my own implementation of train on next section

[11]:
```
# Prepare for training
tokenized_dataset = dataset.map(tokenize_function, batched=True)
data_collator = DataCollatorForLanguageModeling(tokenizer=tokenizer, mlm=True,␣
 ↪mlm_probability=0.15)
training_args = TrainingArguments(
    output_dir="./mlm_results",
    overwrite_output_dir=True,
    num_train_epochs=10,
    per_device_train_batch_size=16,
    save_steps=10000,
    save_total_limit=2,
    logging_dir="./logs",
)
```

[12]:
```
trainer = Trainer(
    model=model,
    args=training_args,
    data_collator=data_collator,
    train_dataset=tokenized_dataset,
)
# I have trained the model and upload it to the google drive, you can download␣
 ↪it and run it from here, you might don't want to run the code for 14 mins
# The training loss output is lost because I rerun this cell, you can find it␣
 ↪in the final term paper report.
# You can uncomment the this code if u wish to train, it takes about 35mins for␣
 ↪my pc to run.
# Also the num of epoch is 10 for this one
# trainer.train()
```

[13]:
```
# training_history = trainer.state.log_history
# for entry in training_history:
#     if "loss" in entry:
#         print(f"Step {entry['step']}: Loss = {entry['loss']}")
```

[14]:
```
# trainer.save_model("./mlm_trained_model");
# tokenizer.save_pretrained("./mlm_trained_model");
```

Important: It is meaning less to test the accuracy of the MLM, because for missing word of each sentences, it can have multiple choices. For example: The food was really [MASK]. The words “good”, “delicious”, “tasty” all make sense here. So it is just pointless to measure the accuracy of the BERT MODEL. So, instead, we use the perplexity instead of accuracy. Perplexity is a more commonly used indicator for evaluating the rationality and fluency of language model generation, especially for language modeling tasks. Perplexity measures the model’s prediction confidence for all words in the test set, reflecting the model’s grasp of the overall language structure and semantics. Compared with a single accuracy rate, perplexity provides a more comprehensive evaluation standard.

[15]:
```
# eval_dataset = load_dataset("wikitext", "wikitext-2-raw-v1",␣
↪split="validation")
# tokenized_eval_dataset = eval_dataset.map(tokenize_function, batched=True)
# trainer.eval_dataset = tokenized_eval_dataset
# eval_result = trainer.evaluate()
# print(eval_result)
# loss = eval_result["eval_loss"]
# perplexity = math.exp(loss)
# print(f"Perplexity: {perplexity:.2f}")
```

The output of perplexity is also showed in the report directly, if you train the model locally, then you can uncomment to see the perplexity

This part below is self-trained BERT model

[16]:
```
model_path = "./mlm_trained_model"
tokenizer = BertTokenizer.from_pretrained(model_path)
model = BertForMaskedLM.from_pretrained(model_path)
model.eval()
model.to(device);
```

[17]:
```
sentence = "I am happy today so I will [MASK] game"
print_prediction(model, sentence)
```

```
tensor([[ 101, 1045, 2572, 3407, 2651, 2061, 1045, 2097,  103, 2208,  102]])
Original sentence: I am happy today so I will [MASK] game
Predicted word: play
```

[18]:
```
sentence = "The scientist aims to [MASK] a deeper understanding of the␣
↪phenomenon."
print_prediction(model, sentence)
```

```
tensor([[ 101, 1996, 7155, 8704, 2000,  103, 1037, 6748, 4824, 1997, 1996, 9575,
         1012,  102]])
Original sentence: The scientist aims to [MASK] a deeper understanding of the
phenomenon.
Predicted word: gain
```

[19]:
```
sentence = "The committee decided to [MASK] the proposal during the session."
print_prediction(model, sentence)
```

```
tensor([[ 101, 1996, 2837, 2787, 2000,  103, 1996, 6378, 2076, 1996, 5219, 1012,
          102]])
Original sentence: The committee decided to [MASK] the proposal during the
session.
Predicted word: consider
```

[20]:
```
sentence = "During the late 1970s , plans [MASK] to create a new highway link␣
↪on the south side of Lake Simcoe to connect Highway 400 and Highway 12"
print_prediction(model, sentence)
```

```
tensor([[  101,  2076,  1996,  2397,  3955,  1010,  3488,   103,  2000,  3443,
          1037,  2047,  3307,  4957,  2006,  1996,  2148,  2217,  1997,  2697,
         21934, 16288,  2000,  7532,  3307,  4278,  1998,  3307,  2260,   102]])
Original sentence: During the late 1970s , plans [MASK] to create a new highway
link on the south side of Lake Simcoe to connect Highway 400 and Highway 12
Predicted word: began
```

This parts below shows my own training function instead of using trainer

[21]:
```
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
model = BertForMaskedLM.from_pretrained("bert-base-uncased")
model.to(device)
dataset = load_dataset("wikitext", "wikitext-2-raw-v1", split="train")
tokenized_dataset = dataset.map(tokenize_function, batched=True,␣
 ↪remove_columns=["text"])
data_collator = DataCollatorForLanguageModeling(tokenizer=tokenizer, mlm=True,␣
 ↪mlm_probability=0.15)
train_dataloader = DataLoader(tokenized_dataset, batch_size=16, shuffle=True,␣
 ↪collate_fn=data_collator)
```

```
Some weights of the model checkpoint at bert-base-uncased were not used when
initializing BertForMaskedLM: ['bert.pooler.dense.bias',
'bert.pooler.dense.weight', 'cls.seq_relationship.bias',
'cls.seq_relationship.weight']
- This IS expected if you are initializing BertForMaskedLM from the checkpoint
of a model trained on another task or with another architecture (e.g.
initializing a BertForSequenceClassification model from a BertForPreTraining
model).
- This IS NOT expected if you are initializing BertForMaskedLM from the
checkpoint of a model that you expect to be exactly identical (initializing a
BertForSequenceClassification model from a BertForSequenceClassification model).
```

[22]:
```
def train(model, dataloader, optimizer, scheduler, device):
    model.train()
    total_loss = 0
    for batch in tqdm(dataloader, desc="Training"):
        input_ids = batch["input_ids"].to(device)
        attention_mask = batch["attention_mask"].to(device)
        labels = batch["labels"].to(device)
        outputs = model(input_ids=input_ids, attention_mask=attention_mask,␣
 ↪labels=labels)
        loss = outputs.loss
        total_loss += loss.item()
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        scheduler.step()
```

```
    avg_loss = total_loss / len(dataloader)
    return avg_loss
```

[23]:
```
learning_rate = 5e-5
num_epochs = 3
optimizer = AdamW(model.parameters(), lr=learning_rate)
total_steps = len(train_dataloader) * num_epochs
scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=0,␣
 ↪num_training_steps=total_steps)
for epoch in range(num_epochs):
    print(f"Epoch {epoch + 1}/{num_epochs}")
    avg_loss = train(model, train_dataloader, optimizer, scheduler, device)
    print(f"Average training loss for epoch {epoch + 1}: {avg_loss:.4f}")

print("Training complete.")
```

```
Epoch 1/3

Training:
100%|                                        |
2295/2295 [02:45<00:00, 13.87it/s]

Average training loss for epoch 1: 1.9247
Epoch 2/3

Training:
100%|                                        |
2295/2295 [02:43<00:00, 14.04it/s]

Average training loss for epoch 2: 1.7795
Epoch 3/3

Training:
100%|                                        |
2295/2295 [02:41<00:00, 14.19it/s]

Average training loss for epoch 3: 1.6701
Training complete.
```

[24]:
```
sentence = "During the late 1970s , plans [MASK] to create a new highway link␣
 ↪on the south side of Lake Simcoe to connect Highway 400 and Highway 12"
print_prediction(model,sentence)
```

```
tensor([[  101,  2076,  1996,  2397,  3955,  1010,  3488,   103,  2000,  3443,
          1037,  2047,  3307,  4957,  2006,  1996,  2148,  2217,  1997,  2697,
         21934, 16288,  2000,  7532,  3307,  4278,  1998,  3307,  2260,   102]])
Original sentence: During the late 1970s , plans [MASK] to create a new highway
link on the south side of Lake Simcoe to connect Highway 400 and Highway 12
Predicted word: emerged
```

The PART BELOW THIS LINE IS BERT NextSentencePrediction (NSP) This part I directly use trainer instead of implement own training function

[26]:
```
model_name = "bert-base-uncased"
tokenizer = BertTokenizer.from_pretrained(model_name)
model = BertForNextSentencePrediction.from_pretrained(model_name).to(device)
```

[27]:
```
sentence_a = "I went to the store"
sentence_b = "bought some fresh vegetables."
inputs = tokenizer.encode_plus(sentence_a, sentence_b, return_tensors='pt')
inputs = {key: value.to(device) for key, value in inputs.items()}
with torch.no_grad():
    outputs = model(**inputs)
    logits = outputs.logits
    predicted_label = torch.argmax(logits, dim=1).item()

if predicted_label == 0:
    print(f"Sentence A: {sentence_a}")
    print(f"Sentence B: {sentence_b}")
    print("Sentence B is likely the next sentence for Sentence A.")
else:
    print(f"Sentence A: {sentence_a}")
    print(f"Sentence B: {sentence_b}")
    print("Sentence B is NOT the next sentence for Sentence A.")
```

```
Sentence A: I went to the store
Sentence B: bought some fresh vegetables.
Sentence B is likely the next sentence for Sentence A.
```

[28]:
```
sentence_a = "I am happy"
sentence_b = "blamed me for an hour in school."
inputs = tokenizer.encode_plus(sentence_a, sentence_b, return_tensors='pt')
inputs = {key: value.to(device) for key, value in inputs.items()}
with torch.no_grad():
    outputs = model(**inputs)
    logits = outputs.logits
    predicted_label = torch.argmax(logits, dim=1).item()

if predicted_label == 0:
    print(f"Sentence A: {sentence_a}")
    print(f"Sentence B: {sentence_b}")
    print("Sentence B is likely the next sentence for Sentence A.")
else:
    print(f"Sentence A: {sentence_a}")
    print(f"Sentence B: {sentence_b}")
    print("Sentence B is NOT the next sentence for Sentence A.")
```

```
Sentence A: I am happy
Sentence B: blamed me for an hour in school.
```

```
Sentence B is NOT the next sentence for Sentence A.
```

[29]:
```
dataset = load_dataset("bookcorpus", split="train[:1%]")
def prepare_sentence_pairs(examples):
    sentence_pairs = {"input_ids": [], "token_type_ids": [], "attention_mask":␣
 ↪[], "labels": []}
    texts = examples["text"]
    num_texts = len(texts)
    all_texts = dataset["text"]
    total_texts = len(all_texts)

    for i in range(num_texts):
        sentence_a = texts[i]

        if random.random() < 0.5 and i < num_texts - 1:
            sentence_b = texts[i + 1]
            label = 0
        else:
            random_index = random.randint(0, total_texts - 1)
            while random_index == i or (i < num_texts - 1 and random_index == i␣
 ↪+ 1):
                random_index = random.randint(0, total_texts - 1)
            sentence_b = all_texts[random_index]
            label = 1

        encoded = tokenizer(
            sentence_a,
            sentence_b,
            padding="max_length",
            truncation=True,
            max_length=128
        )
        sentence_pairs["input_ids"].append(encoded["input_ids"])
        sentence_pairs["token_type_ids"].append(encoded["token_type_ids"])
        sentence_pairs["attention_mask"].append(encoded["attention_mask"])
        sentence_pairs["labels"].append(label)

    return sentence_pairs
```

[30]:
```
processed_dataset = dataset.map(prepare_sentence_pairs, batched=True,␣
 ↪remove_columns=["text"]);
training_args = TrainingArguments(
    output_dir="./nsp_results",
    overwrite_output_dir=True,
    num_train_epochs=3,
    per_device_train_batch_size=16,
    save_steps=10_000,
```

```
    save_total_limit=2,
    logging_dir="./logs",
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=processed_dataset,
)
```

[31]:
```
# since the training takes forever and when i restart it, the output of this column is missed
# but you can load my model by downloading it directly from the google drive
# trainer.train()
```

[32]:
```
# output_dir = "./nsp_trained_model"
# trainer.save_model(output_dir)
# tokenizer.save_pretrained(output_dir)
```

[40]:
```
model_dir = "./nsp_trained_model"
tokenizer = BertTokenizer.from_pretrained(model_dir)
model = BertForNextSentencePrediction.from_pretrained(model_dir).to(device)
```

The perplexity of NSP is actually not that ideal because the dataset is still too small for our local machine to train, the pre-trained model has better result.

[41]:
```
sentence_a = "How old is Tommy?"
sentence_b = "Tommy loves to eat banana."

inputs = tokenizer(sentence_a, sentence_b, return_tensors="pt")
inputs = {k: v.to(device) for k, v in inputs.items()}

with torch.no_grad():
    outputs = model(**inputs)
    logits = outputs.logits
    predicted_label = torch.argmax(logits, dim=1).item()

if predicted_label == 0:
    print("Sentence B is likely the next sentence for Sentence A.")
else:
    print("Sentence B is NOT the next sentence for Sentence A.")
```

```
Sentence B is likely the next sentence for Sentence A.
```

[42]:
```
sentence_a = "I am happy"
sentence_b = "blamed me for an hour in school."

inputs = tokenizer(sentence_a, sentence_b, return_tensors="pt")
```

```
inputs = {k: v.to(device) for k, v in inputs.items()}

with torch.no_grad():
    outputs = model(**inputs)
    logits = outputs.logits
    predicted_label = torch.argmax(logits, dim=1).item()

if predicted_label == 0:
    print("Sentence B is likely the next sentence for Sentence A.")
else:
    print("Sentence B is NOT the next sentence for Sentence A.")
```

```
Sentence B is likely the next sentence for Sentence A.
```

This Part of the code is for BERT-BASED-CHINESE

[39]:
```
model_name = 'bert-base-chinese'
samples = ['[CLS]       [SEP]    [MASK]     [SEP]']
model = BertForMaskedLM.from_pretrained(model_name)
tokenizer = BertTokenizer.from_pretrained('bert-base-chinese')
model.eval();
```

```
Some weights of the model checkpoint at bert-base-chinese were not used when
initializing BertForMaskedLM: ['bert.pooler.dense.bias',
'bert.pooler.dense.weight', 'cls.seq_relationship.bias',
'cls.seq_relationship.weight']
- This IS expected if you are initializing BertForMaskedLM from the checkpoint
of a model trained on another task or with another architecture (e.g.
initializing a BertForSequenceClassification model from a BertForPreTraining
model).
- This IS NOT expected if you are initializing BertForMaskedLM from the
checkpoint of a model that you expect to be exactly identical (initializing a
BertForSequenceClassification model from a BertForSequenceClassification model).
```

[36]:
```
tokenized_text = [tokenizer.tokenize(i) for i in samples]
input_ids = [tokenizer.convert_tokens_to_ids(i) for i in tokenized_text]
input_ids = torch.LongTensor(input_ids)
print(input_ids)
```

```
tensor([[ 101,  704, 1744, 4638, 7674, 6963, 3221, 1525, 7027, 8043,  102, 1266,
          776, 3221,  103, 1744, 4638, 7674, 6963,  511,  102]])
```

[37]:
```
outputs = model(input_ids)
prediction_scores = outputs[0]
prediction_scores.shape
```

[37]:
```
torch.Size([1, 21, 21128])
```

[38]:
```
sample = prediction_scores[0].detach().numpy()
pred = np.argmax(sample, axis=1)

tokenizer.convert_ids_to_tokens(pred)[14]
```

[38]:
```
' '
```