assignment11

November 12, 2023

[1]:
```
import pandas as pd
import numpy as np
from transformers import BertTokenizer, BertForSequenceClassification
import torch
from sklearn.model_selection import train_test_split
from torch.utils.data import DataLoader, RandomSampler, SequentialSampler,␣
↪TensorDataset
```

[2]:
```
file_path = '/Users/pan/Desktop/dataanalytics/bodybuilding_nutrition_products.
↪csv'
df = pd.read_csv(file_path)
```

[3]:
```
threshold_for_favorite = 100

# Create the binary target column
df['is_favorite'] = np.where(df['number_of_reviews'] >= threshold_for_favorite,␣
↪1, 0)
```

[4]:
```
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

input_ids = []
attention_masks = []

# Assuming 'Recipe_name' is the text column
for sentence in df['brand_name']:
    encoded_dict = tokenizer.encode_plus(
                        sentence,
                        add_special_tokens = True,
                        max_length = 64,
                        pad_to_max_length = True,
                        return_attention_mask = True,
                        return_tensors = 'pt',
                   )

    input_ids.append(encoded_dict['input_ids'])
    attention_masks.append(encoded_dict['attention_mask'])
```

```
input_ids = torch.cat(input_ids, dim=0)
attention_masks = torch.cat(attention_masks, dim=0)
labels = torch.tensor(df['is_favorite'])
```

```
Truncation was not explicitly activated but `max_length` is provided a specific
value, please use `truncation=True` to explicitly truncate examples to max
length. Defaulting to 'longest_first' truncation strategy. If you encode pairs
of sequences (GLUE-style) with the tokenizer you can select this strategy more
precisely by providing a specific strategy to `truncation`.
/Users/pan/anaconda3/lib/python3.11/site-
packages/transformers/tokenization_utils_base.py:2418: FutureWarning: The
`pad_to_max_length` argument is deprecated and will be removed in a future
version, use `padding=True` or `padding='longest'` to pad to the longest
sequence in the batch, or use `padding='max_length'` to pad to a max length. In
this case, you can give a specific length with `max_length` (e.g.
`max_length=45`) or leave max_length to None to pad to the maximal input size of
the model (e.g. 512 for Bert).
  warnings.warn(
```

[5]:

```
train_inputs, validation_inputs, train_labels, validation_labels = train_test_split(input_ids, labels, random_state=2018, test_size=0.1)
train_masks, validation_masks, _, _ = train_test_split(attention_masks, labels, random_state=2018, test_size=0.1)

train_data = TensorDataset(train_inputs, train_masks, train_labels)
train_sampler = RandomSampler(train_data)
train_dataloader = DataLoader(train_data, sampler=train_sampler, batch_size=32)

validation_data = TensorDataset(validation_inputs, validation_masks, validation_labels)
validation_sampler = SequentialSampler(validation_data)
validation_dataloader = DataLoader(validation_data, sampler=validation_sampler, batch_size=32)
```

[6]:

```
model = BertForSequenceClassification.from_pretrained(
    "bert-base-uncased",
    num_labels = 2,
    output_attentions = False,
    output_hidden_states = False,
)
```

```
Some weights of BertForSequenceClassification were not initialized from the
model checkpoint at bert-base-uncased and are newly initialized:
['classifier.bias', 'classifier.weight']
You should probably TRAIN this model on a down-stream task to be able to use it
for predictions and inference.
```

[7]:
```python
import pandas as pd
import numpy as np
import torch
from transformers import BertTokenizer, BertForSequenceClassification, AdamW,␣
 ↪get_linear_schedule_with_warmup
from sklearn.model_selection import train_test_split
from torch.utils.data import DataLoader, RandomSampler, SequentialSampler,␣
 ↪TensorDataset
```

```
2023-11-12 15:15:33.524143: I tensorflow/core/platform/cpu_feature_guard.cc:182]
This TensorFlow binary is optimized to use available CPU instructions in
performance-critical operations.
To enable the following instructions: AVX2 FMA, in other operations, rebuild
TensorFlow with the appropriate compiler flags.
```

[9]:
```python
threshold_for_favorite = 100

# Create the binary target column
df['is_favorite'] = np.where(df['number_of_reviews'] >= threshold_for_favorite,␣
 ↪1, 0)
```

[10]:
```python
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

input_ids = []
attention_masks = []

# Assuming 'Recipe_name' is the text column
for sentence in df['brand_name']:
    encoded_dict = tokenizer.encode_plus(
                        sentence,
                        add_special_tokens = True,
                        max_length = 64,
                        pad_to_max_length = True,
                        return_attention_mask = True,
                        return_tensors = 'pt',
                   )

    input_ids.append(encoded_dict['input_ids'])
    attention_masks.append(encoded_dict['attention_mask'])

input_ids = torch.cat(input_ids, dim=0)
attention_masks = torch.cat(attention_masks, dim=0)
labels = torch.tensor(df['is_favorite'])
```

```
Truncation was not explicitly activated but `max_length` is provided a specific
value, please use `truncation=True` to explicitly truncate examples to max
length. Defaulting to 'longest_first' truncation strategy. If you encode pairs
```

of sequences (GLUE-style) with the tokenizer you can select this strategy more precisely by providing a specific strategy to `truncation`.

[11]:
```
# Creating DataLoaders for Training and Validation:
# Split data into train and validation sets
train_inputs, validation_inputs, train_labels, validation_labels =␣
 ↪train_test_split(input_ids, labels, random_state=2018, test_size=0.1)
train_masks, validation_masks, _, _ = train_test_split(attention_masks, labels,␣
 ↪random_state=2018, test_size=0.1)

# Create the DataLoader for our training set
train_data = TensorDataset(train_inputs, train_masks, train_labels)
train_sampler = RandomSampler(train_data)
train_dataloader = DataLoader(train_data, sampler=train_sampler, batch_size=32)

# Create the DataLoader for our validation set
validation_data = TensorDataset(validation_inputs, validation_masks,␣
 ↪validation_labels)
validation_sampler = SequentialSampler(validation_data)
validation_dataloader = DataLoader(validation_data, sampler=validation_sampler,␣
 ↪batch_size=32)
```

[12]:
```
# Setting Up Optimizer and Scheduler:
# Note: AdamW is a class from the huggingface library (as opposed to pytorch)
optimizer = AdamW(model.parameters(),
                  lr = 2e-5, # args.learning_rate - default is 5e-5
                  eps = 1e-8 # args.adam_epsilon  - default is 1e-8
                 )

# Number of training epochs (authors recommend between 2 and 4)
epochs = 4

# Total number of training steps is [number of batches] x [number of epochs]
total_steps = len(train_dataloader) * epochs

# Create the learning rate scheduler
scheduler = get_linear_schedule_with_warmup(optimizer,
                                            num_warmup_steps = 0, # Default␣
 ↪value in run_glue.py
                                            num_training_steps = total_steps)
```

```
/Users/pan/anaconda3/lib/python3.11/site-
packages/transformers/optimization.py:411: FutureWarning: This implementation of
AdamW is deprecated and will be removed in a future version. Use the PyTorch
implementation torch.optim.AdamW instead, or set `no_deprecation_warning=True`
to disable this warning
  warnings.warn(
```

[13]:
```python
# Check if GPU is available and set the device accordingly
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Also, make sure to move the model to the device
model.to(device)
```

```
Using device: cpu
```

[13]:
```
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)
  )
```

[14]:

```python
import time

def format_time(elapsed):
    '''
    Takes a time in seconds and returns a string hh:mm:ss
    '''
    # Round to the nearest second
    elapsed_rounded = int(round(elapsed))

    # Format as hh:mm:ss
    return str(datetime.timedelta(seconds=elapsed_rounded))
```

[ ]:

```python
import datetime
import random

# Training Loop:
# Seed value for reproducibility
seed_val = 42

random.seed(seed_val)
np.random.seed(seed_val)
torch.manual_seed(seed_val)
torch.cuda.manual_seed_all(seed_val)

# Store the average loss after each epoch so we can plot them
loss_values = []

for epoch_i in range(0, epochs):
    start_time = time.time()
    # Perform one full pass over the training set
    print('======== Epoch {:} / {:} ========'.format(epoch_i + 1, epochs))
    print('Training...')

    total_loss = 0
    model.train()

    # For each batch of training data...
    for step, batch in enumerate(train_dataloader):
        b_input_ids = batch[0].to(device)
```

```
        b_input_mask = batch[1].to(device)
        b_labels = batch[2].to(device)

        # Clear any previously calculated gradients
        model.zero_grad()

        # Perform a forward pass (evaluate the model on this training batch)
        outputs = model(b_input_ids, token_type_ids=None,␣
↪attention_mask=b_input_mask, labels=b_labels)
        loss = outputs.loss
        total_loss += loss.item()

        # Perform a backward pass to calculate the gradients
        loss.backward()

        # Clip the norm of the gradients to 1.0 to prevent "exploding gradients"
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

        # Update parameters and take a step using the computed gradient
        optimizer.step()

        # Update the learning rate
        scheduler.step()

    # Calculate the average loss over the training data
    avg_train_loss = total_loss / len(train_dataloader)
    loss_values.append(avg_train_loss)

    time_elapsed = time.time() - start_time
    print("  Average training loss: {0:.2f}".format(avg_train_loss))
    print("  Training epoch took: {:}".format(format_time(time_elapsed)))

    # Validation step
    print("Running Validation...")

    model.eval()

    # Tracking variables
    eval_loss, eval_accuracy = 0, 0
    nb_eval_steps, nb_eval_examples = 0, 0
    #
```

```
======== Epoch 1 / 4 ========
Training…
```

[ ]:
```
model = BertForSequenceClassification.from_pretrained("bert-base-uncased",␣
↪num_labels=2)
```

[ ]:
```python
output_dir = '/Users/yaoyaoliu/Documents/Graduate Class/2023/Fall 2023/293C/
 ↪model_save/'
model.save_pretrained(output_dir)
tokenizer.save_pretrained(output_dir)
```

[ ]:
```python
model = BertForSequenceClassification.from_pretrained(output_dir)
tokenizer = BertTokenizer.from_pretrained(output_dir)
```

[ ]:
```python
# Classify New Sentences
def classify_sentence(sentence):
    # Tokenize the sentence
    inputs = tokenizer.encode_plus(
        sentence,
        add_special_tokens=True,
        max_length=64,
        pad_to_max_length=True,
        return_attention_mask=True,
        return_tensors='pt',
    )

    # Move tensors to the same device as the model
    input_ids = inputs['input_ids']
    attention_mask = inputs['attention_mask']

    # Get model predictions
    with torch.no_grad():
        outputs = model(input_ids, attention_mask=attention_mask)

    # Convert output logits to softmax probabilities
    probs = torch.nn.functional.softmax(outputs.logits, dim=1)

    # Get the predicted class (the one with the highest probability)
    predicted_class = torch.argmax(probs, dim=1).item()

    return predicted_class

# Example usage
test_sentence = "The Best Vegan Breakfast Sandwich"
prediction = classify_sentence(test_sentence)
print(f"Predicted class for '{test_sentence}': {prediction}")
```

[ ]:
```python

```