3. Intent Classification with BERT

Trying to predict the scenario (or the scenario group) mainly involves a semantic analysis of the question. This means that it is feasible to reach a high accuracy by only looking at the presence of some words, not considering at all the syntactic role of these words. To predict the intent, though, this approach is no longer optimal, since the range of words present in the questions is considerably restricted. This fact makes it paramount to draw additional information from the syntactic roles of the words present in a question.
Questions like "*Do I have any alarms set?*" and "*Remove all set alarms*" share a significant portion of their vocabulary, and knowing that both *set* and *alarms* are in the question no longer helps if we cannot determine if *set* refers to the *alarms* or if it is a verb acting on the *alarms* .

In other words, we need a **Contextual Model**, i.e. a model which takes into consideration the context of the sentence and, most importantly, the **syntactical relantionships** between the words. We chose BERT for this task, as it is one of the most advanced models for text classification, having undergone a contextual training fit of the whole Wikipedia and Books Corpus (>10,000 books of different genres).

The pre-trained BERT is already available in the PyTorch package as a *BertForSequenceClassification* model, all we need to do is fine-tune the last layer of the model in order for it to be able to predict an intent among our intended range.

The first step is importing the dataset.

In [ ]:
```
import tensorflow as tf

device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
  raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))

# install
!pip install pytorch-pretrained-bert pytorch-nlp

# BERT imports
import torch
from torch.utils.data import TensorDataset, DataLoader, RandomSampl
er, SequentialSampler
from keras.preprocessing.sequence import pad_sequences
from sklearn.model_selection import train_test_split
from pytorch_pretrained_bert import BertTokenizer, BertConfig
from pytorch_pretrained_bert import BertAdam, BertForSequenceClassi
fication
from tqdm import tqdm, trange
```

```
import pandas as pd
import io
import numpy as np
import matplotlib.pyplot as plt
% matplotlib inline

# specify GPU device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu
")
n_gpu = torch.cuda.device_count()
torch.cuda.get_device_name(0)
import pandas as pd
df_complete = pd.read_csv("/content/dataset_intent_train.csv", sep=
";")

def group(groups: list):
    for group in groups:
        to_group = group.split("/")
        for scen in to_group:
            df_complete["scenario"][df_complete["scenario"] == scen
] = group
    return sorted(list(set(df_complete["scenario"])))

scenarios = group(['alarm',
                   'email',
                   'iot',
                   'music/audio/play',
                   'recommendation/lists/datetime/calendar',
                   'weather/cooking/transport/general/social/news/t
akeaway/qa'])

df = df_complete[df_complete["scenario"]=='alarm']  ## predicting h
ere for ALARM
```

Now we modify our questions to meet BERT's requirements for input text. Each sentence must begin with a "[CLS]" token and must end with a "[SEP]" token. We can then tokenize the sentences with the builtin *BertTokenizer*, which will tokenize the words in a way that BERT can understand, and for which has already undergone extensive fitting. We also need to encode the intent labels.

In [ ]:
```
sentences = ["[CLS] " + question + " [SEP]" for question in df["que
stion"]]

# Tokenize with BERT tokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased', do_l
ower_case=True)
tokenized_texts = [tokenizer.tokenize(sent) for sent in sentences]

MAX_LEN = 128
# Pad our input tokens
input_ids = pad_sequences([tokenizer.convert_tokens_to_ids(txt) for
txt in tokenized_texts],
                          maxlen=MAX_LEN, dtype="long", truncating=
"post", padding="post")
# Use the BERT tokenizer to convert the tokens to their index numbe
rs in the BERT vocabulary
input_ids = [tokenizer.convert_tokens_to_ids(x) for x in tokenized_
texts]
input_ids = pad_sequences(input_ids, maxlen=MAX_LEN, dtype="long",
truncating="post", padding="post")

intent_labels = {intent: i for (i, intent) in enumerate(list(set(df
["intent"])))}
labels = np.asarray(df["intent"].apply(lambda intent: intent_labels
[intent]))
```

Now it is time to create all the tensor datasets we will need for fitting the model.

In [ ]:
```
attention_masks = []
# Create a mask of 1s for each token followed by 0s for padding
for seq in input_ids:
  seq_mask = [float(i>0) for i in seq]
  attention_masks.append(seq_mask)


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_ma
sks, input_ids,
                                                       random_state=2018, tes
t_size=0.1)

# Convert all of our data into torch tensors, the required datatype
for our model
train_inputs = torch.tensor(train_inputs)
validation_inputs = torch.tensor(validation_inputs)
train_labels = torch.tensor(train_labels)
validation_labels = torch.tensor(validation_labels)
train_masks = torch.tensor(train_masks)
validation_masks = torch.tensor(validation_masks)

# Select a batch size for training.
batch_size = 32

# Create an iterator of our data with torch DataLoader
train_data = TensorDataset(train_inputs, train_masks, train_labels)
train_sampler = RandomSampler(train_data)
train_dataloader = DataLoader(train_data, sampler=train_sampler, ba
tch_size=batch_size)
validation_data = TensorDataset(validation_inputs, validation_masks
, validation_labels)
validation_sampler = SequentialSampler(validation_data)
validation_dataloader = DataLoader(validation_data, sampler=validat
ion_sampler, batch_size=batch_size)

model = BertForSequenceClassification.from_pretrained("bert-base-un
cased", num_labels=len(intent_labels))
```

Now we are ready to fit and evaluate the model.

In [ ]:
```
torch.cuda.empty_cache()
model.cuda()
# BERT fine-tuning parameters
param_optimizer = list(model.named_parameters())
no_decay = ['bias', 'gamma', 'beta']
optimizer_grouped_parameters = [
    {'params': [p for n, p in param_optimizer if not any(nd in n fo
```

```
r nd in no_decay)],
     'weight_decay_rate': 0.01},
    {'params': [p for n, p in param_optimizer if any(nd in n for nd
in no_decay)],
     'weight_decay_rate': 0.0}
]

optimizer = BertAdam(optimizer_grouped_parameters,
                     lr=2e-5,
                     warmup=.1)

# Function to calculate the accuracy of our predictions vs labels
def flat_accuracy(preds, labels):
    pred_flat = np.argmax(preds, axis=1).flatten()
    labels_flat = labels.flatten()
    return np.sum(pred_flat == labels_flat) / len(labels_flat)

# Store our loss and accuracy for plotting
train_loss_set = []
# Number of training epochs
epochs = 4

# BERT training loop
for _ in trange(epochs, desc="Epoch"):

  ## TRAINING

  # Set our model to training mode
  model.train()
  # Tracking variables
  tr_loss = 0
  nb_tr_examples, nb_tr_steps = 0, 0
  # Train the data for one epoch
  for step, batch in enumerate(train_dataloader):
    # Add batch to GPU
    batch = tuple(t.to(device) for t in batch)
    # Unpack the inputs from our dataloader
    b_input_ids, b_input_mask, b_labels = batch
    # Clear out the gradients (by default they accumulate)
    optimizer.zero_grad()
    # Forward pass
    loss = model(b_input_ids, token_type_ids=None, attention_mask=b
_input_mask, labels=b_labels)
    train_loss_set.append(loss.item())
    # Backward pass
    loss.backward()
    # Update parameters and take a step using the computed gradient
    optimizer.step()
    # Update tracking variables
    tr_loss += loss.item()
    nb_tr_examples += b_input_ids.size(0)
    nb_tr_steps += 1
  print("Train loss: {}".format(tr_loss/nb_tr_steps))
```

```
  ## VALIDATION

  # Put model in evaluation mode
  model.eval()
  # Tracking variables
  eval_loss, eval_accuracy = 0, 0
  nb_eval_steps, nb_eval_examples = 0, 0
  # Evaluate data for one epoch
  for batch in validation_dataloader:
    # Add batch to GPU
    batch = tuple(t.to(device) for t in batch)
    # Unpack the inputs from our dataloader
    b_input_ids, b_input_mask, b_labels = batch
    # Telling the model not to compute or store gradients, saving m
emory and speeding up validation
    with torch.no_grad():
      # Forward pass, calculate logit predictions
      logits = model(b_input_ids, token_type_ids=None, attention_ma
sk=b_input_mask)
    # Move logits and labels to CPU
    logits = logits.detach().cpu().numpy()
    label_ids = b_labels.to('cpu').numpy()
    tmp_eval_accuracy = flat_accuracy(logits, label_ids)
    eval_accuracy += tmp_eval_accuracy
    nb_eval_steps += 1
  print("Validation Accuracy: {}".format(eval_accuracy/nb_eval_step
s))
```

The last piece, a little helper class with a simple `predict` method that will give us the final predictions.

In [ ]:
```
class BertForIntent:
  def __init__(self, model):
    self.model = model

  def predict(self, questions):
    # Create sentence and label lists
    # Tokenize all of the sentences and map the tokens to thier wor
d IDs.

    # For every sentence...
    sentences = ["[CLS] " + question + " [SEP]" for question in que
stions]

    # Tokenize with BERT tokenizer
    tokenizer = BertTokenizer.from_pretrained('bert-base-uncased',
do_lower_case=True)
    tokenized_texts = [tokenizer.tokenize(sent) for sent in sentenc
es]
    MAX_LEN = 128
    # Pad our input tokens
    input_ids = pad_sequences([tokenizer.convert_tokens_to_ids(txt)
for txt in tokenized_texts],
```

```
                                maxlen=MAX_LEN, dtype="long", truncat
ing="post", padding="post")
    # Use the BERT tokenizer to convert the tokens to their index n
umbers in the BERT vocabulary
    input_ids = [tokenizer.convert_tokens_to_ids(x) for x in tokeni
zed_texts]
    input_ids = pad_sequences(input_ids, maxlen=MAX_LEN, dtype="lon
g", truncating="post", padding="post")

    # Create attention masks
    attention_masks = []

    # Create a mask of 1s for each token followed by 0s for padding
    for seq in input_ids:
      seq_mask = [float(i>0) for i in seq]
      attention_masks.append(seq_mask)

    # Convert to tensors.
    prediction_inputs = torch.tensor(input_ids)
    prediction_masks = torch.tensor(attention_masks)

    # Set the batch size.
    batch_size = 1

    # Create the DataLoader.
    prediction_data = TensorDataset(prediction_inputs, prediction_m
asks)
    prediction_sampler = SequentialSampler(prediction_data)
    prediction_dataloader = DataLoader(prediction_data, sampler=pre
diction_sampler, batch_size=batch_size)


    self.model.cuda()

    self.model.eval()

    # Tracking variables
    predictions = []

    # Predict
    for batch in prediction_dataloader:
      # Add batch to GPU
      batch = tuple(t.to(device) for t in batch)

      # Unpack the inputs from our dataloader
      b_input_ids, b_input_mask = batch

      # Telling the model not to compute or store gradients, saving
memory and
      # speeding up prediction
      with torch.no_grad():
          # Forward pass, calculate logit predictions
          outputs = self.model(b_input_ids, token_type_ids=None,
                               attention_mask=b_input_mask)
```

```
        logits = outputs[0]

        # Move logits and labels to CPU
        logits = logits.detach().cpu().numpy()

        # Store predictions and true labels
        predictions.append(np.argmax(logits))
    return predictions
```

The code above is for the *alarm* group, we only need to repeat it for all the other groups.