Language Identification

Spoken and Written Language Processing

Data Science and Engineering
Polytechnic University of Catalonia

Claudia Len Manero - 53869554
Alina Castell Blasco - 49188689S

April 2024

1 Introduction

This assignment focuses on the Language Identification (LI) task utilizing a dataset derived from Wikipedia paragraphs. The primary objective is to enhance a character-based Recurrent Neural Network (RNN) Baseline through analysis, optimization, and comparative studies. This involves implementing modifications such as exploring different hyperparameters, combining output layers, incorporating dropout layers, and comparing various RNN architectures. As well as exploring alternative deep neural network (DNN) architectures and using different input features such as words, character n-grams, or character counts.

The base model considered throughout all the study has $embedding\_size = 64$, $hidden\_size = 256$ and $batch\_size = 256$, and its results are:

Emb. size	Hid. size	Batch size	Train time	Train Acc	Valid Acc
64	256	256	1406.57s	98,742%	92.7%

2 Output Layer Modification: Combine the output of the max-pool layer with the output of a mean-pool layer.

In this section, the main modification to the model is the combination of the outputs of the max-pool layer and the mean-pool layer; which can be implemented by addition or concatenation, as shown in the following code implementation.

```python
class CharRNNClassifier(torch.nn.Module):
    ...
    def forward(self, input, input_lengths):
    ...
    max_pool, _ = padded.max(dim=0)
    padded, _ = torch.nn.utils.rnn.pad_packed_sequence(output, padding_value=float(0))
    mean_pool = padded.mean(dim=0)
    # ADDITION
    output = max_pool + mean_pool
    # CONCATENATION
    output = torch.cat([max_pool, mean_pool], dim=1)
    output = self.h2o(output)
    return output
```

In addition to the output layer modification, other changes have been made to the RNN architecture (LSTM and GRU), as well as to the number of layers (1 and 2), and the input direction. The embedding size and batch size are 64 and 256, respectively, as the baseline moddel. The results of the different experiments are shown in the next table:

Unidirectional						
RNN	Layers	Pooling	Hid. size	Train time	Train Acc	Valid Acc
LSTM	1	max-mean sum	256	1538s	99.585%	92.8%
LSTM	1	max-mean concat	256	1547s	99.378%	92.6%
LSTM	2	max-mean sum	256	1750s	99.737%	**93.1%**
LSTM	2	max-mean concat	256	1361s	99.660%	92.8%
GRU	1	max-mean sum	256	1456s	99.779%	92.8%
GRU	1	max-mean concat	256	1443s	99.711%	92.6%
GRU	2	max-mean sum	256	2177s	99.728%	**93.1%**
GRU	2	max-mean concat	256	2182s	99.723%	92.8%

Table 1: Modifications on output layer, unidirectional RNN architectures, and number of layers.

Bidirectional						
RNN	Layers	Pooling	Hid. size	Train time	Train Acc	Valid Acc
LSTM	1	max-mean sum	256	2719s	99.860%	93.7%
LSTM	1	max-mean concat	256	2724s	99.806%	93.3%
LSTM	2	max-mean sum	256	4928s	99.826%	93.7%
LSTM	2	max-mean concat	256	4948s	99.804%	**93.8%**
GRU	1	max-mean sum	256	2549s	99.895%	93.7%
GRU	1	max-mean concat	256	2537s	99.849%	93.5%
GRU	2	max-mean sum	256	4275s	99.848%	93.8%
GRU	2	max-mean concat	256	4237s	99.841%	**94.0%**

Table 2: Modifications on output layer, bidirectional RNN architectures, and number of layers.

The best results across unidirectional, bidirectional and LSTM, GRU are highlighted in bold in both tables. The unidirectional LSTM model with single-layer has a slightly better performance using a max-mean sum pooling layer than using a max-mean concatenation, and an increased hidden layer size of 512. The same pattern can be observed with the two-layered LSTM model, as well as with the GRU architecture. Note that the models' performance and results are independent of the RNN architecture, with exception of the two best models. As for the bidirectional models, they show a similar structure although with an improved accuracy overall, as expected. Bidirectional models tend to have higher accuracy since the input flows in both directions, thus, making every component of the input sequence have information from both the past and present. Since the bidirectional model has a forward and backward layer, it takes twice the training time.

Even though it is true that both unidirectional and bidirectional models tend to benefit from deeper architecture, there isn't a clear dominance of max-mean addition over concatenation, since unidirectional model seems to enhance its performance with the first and the bidirectional with the last. It is important to consider the trade-off between training time and model performance, with deeper architectures (two-layered and bidirectional) generally requiring almost twice the training time but resulting in higher accuracy.

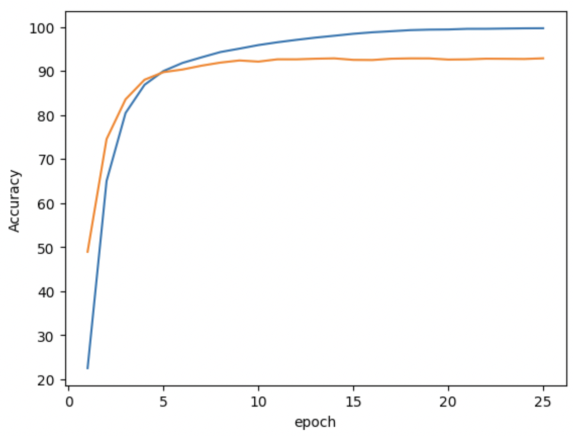

Figure 1: Training (blue) and validation (orange) curves of two-layered unidirectional LSTM model with max-mean addition pooling.

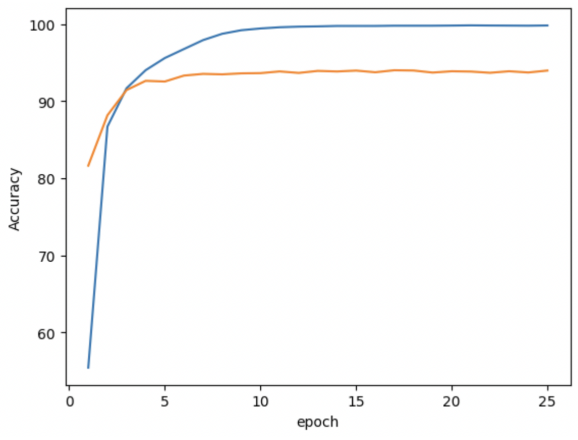

Figure 2: Training (blue) and validation (orange) curves of two-layered bidirectional GRU model with max-mean concatenation pooling.

The graphics shows that the LSTM model seems to achieve validation accuracy closer to the training one than the GRU model. The GRU model achieves greater validation accuracy in the end, which could be explained by the fact that the LSTM's valid accuracy starts from lower values.

3 Dropout Layer: Adding dropout layer after the pooling layer.

As for this section, the best combination of each architecture and type of output layer modification have been taken to observe the effect of the addition of a dropout layer. It generally helps prevent overfitting, leading to improved generalization performance.The dropout factor used is of 0.1, since it has been proved that it is the one with greater accuracy results. The implementation of the dropout layer is the following:

```python
class CharRNNClassifier(torch.nn.Module):
    def __init__(...):
        ...
        self.dropout = torch.nn.Dropout(0.1)
    def forward(self, input, input_lengths):
        # T x B
        encoded = self.embed(input)
        encoded = self.dropout(encoded)
        ...
```

The results of the experiments are shown in the next table:

Dropout							
RNN	Layers	Pooling	Direction	Train time	Valid Acc	Previous Acc	Epochs
LSTM	2	max-mean sum	Uni	2444s	**93.6%**	93.1%	25
GRU	2	max-mean sum	Uni	1490s	**93.5%**	93.1%	17
LSTM	2	max-mean concat	Bi	3763s	**94.0%**	93.8%	19
GRU	2	max-mean concat	Bi	2391s	**94.1%**	94.0%	14

Table 3: Addition of dropout layer to the best combination of RNN, layers and max-mean pooling.

The evaluation of the addition of dropout to the model is done by observing the number of epochs it takes to reach an improvement of performance, compared to the same models without dropout. The effect of the addition of dropout is clearly positive since all models increase their validation accuracy notably. In fact, they manage to improve the previous results with few epochs; with exception of the two-layered LSTM max-mean sum that needs 25 epochs to achieve its maximum accuracy.

After these results, we can safely assume that the addition of a dropout layer of 0.1 will probably improve the models performance.

This graphic shows the training and validation behaviour of the model with a dropout layer, after the 5th epoch both seem to be more constant in the values although the validation curve is more constant than the training one.

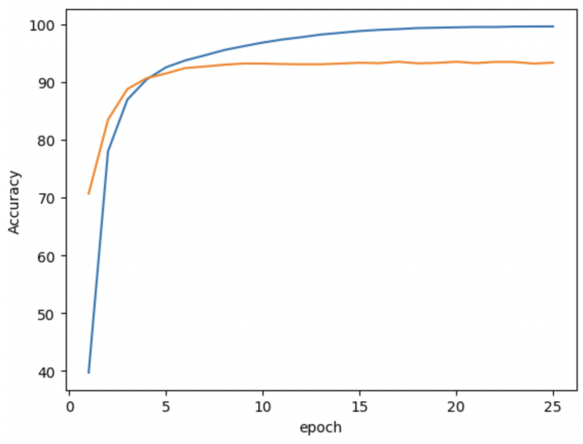

Figure 3: Training (blue) and validation (orange) curves of two-layered bidirectional GRU model with dropout layer and max-mean concatenation pooling.

4 Hyperparameter Optimization: Embedding size, RNN hidden size, batch size to improve model performance.

For hyperparameter optimization we tried augmenting the size of the embeddings, the hidden layer of the RNN and the batch size.

As mentioned earlier, the base model had $embedding\_size = 64$, $hidden\_size = 256$ and $batch\_size = 256$, and gave us a validation accuracy of 92.7%. To find the perfect combination of values we tested with $embedding\_size = [64, 128, 256]$, $hidden\_size = [256, 512, 1024]$ and $batch\_size = [256, 512, 1024]$ and modified our code to avoid the need of trying each combination separately.

```
batch_size = [256, 512, 1024]
```

```
hidden_size = [256, 512, 1024]
embedding_size = [64, 128, 256]
epochs = 25

params_grid = {
    'hidden_size': hidden_size,
    'embedding_size': embedding_size,
    'batch_size': batch_size
}
train_accuracy = []
valid_accuracy = []
for embedding_size in params_grid['embedding_size']:
    for hidden_size in params_grid['hidden_size']:
        for batch_size in params_grid['batch_size']:
            model = CharRNNClassifier(ntokens, embedding_size, hidden_size, nlabels,
    bidirectional=bidirectional, pad_idx=pad_index).to(device)
            optimizer = torch.optim.Adam(model.parameters())
            print(f'Training cross-validation model for {epochs} epochs with
    embedding_size {embedding_size}, hidden_size {hidden_size} and batch_size {
    batch_size}')
            t0 = time.time()
            for epoch in range(1, epochs + 1):
                ...
```

Bellow you can find a table with the metrics of each combination, except for $embedding\_size = 64$ and $hidden\_size = 1024$ as it gave memory problems and were discarted.

Emb. size	Hid. size	Batch size	Train time	Train Acc	Valid Acc
64	256	512	1406s	97,706%	92.9%
64	256	1024	1138s	97.031%	92.3%
64	512	256	2917s	99.887%	94.1%
64	**512**	**512**	**2554s**	**99.888%**	**94.2%**
64	512	1024	2543s	99.620%	93.8%
128	256	256	1469s	98.726%	92.8%
128	256	512	1255s	98.169%	92.6%
128	256	1024	1227s	97,4%	92.7%
128	**512**	**256**	**3053s**	**99,88%**	**94.2%**
128	512	512	2693s	99,859%	93.9%
128	512	1024	2684s	99,685%	93.6%
128	**1024**	**256**	**7369s**	**99,923%**	**94.8%**
128	**1024**	**512**	**6720s**	**99.911%**	**94.7%**
128	**1024**	**1024**	**5938s**	**99.9%**	**94.6%**
256	256	256	1590s	98.240%	92.9%
256	256	512	1376s	98.094%	92.6%
256	256	1024	1343s	97.488%	92.6%
256	512	256	3337s	99.753%	94.0%
256	512	512	2972s	99.689%	93.6%
256	512	1024	2980s	99.453%	93.6%
256	**1024**	**256**	**7799s**	**99.910%**	**94.7%**
256	**1024**	**512**	**7199s**	**99.894%**	**94.8%**

Table 4: Hyperparameter tuning results.

Multiple combinations caught our attention as they gave good validation accuracy results that exceeded the baseline model ones.

If we consider training time, the best model would be the ones trained with *embedding_size* = 64, *hidden_size* = 512 and *batch_size* = 512 or *embedding_size* = 128, *hidden_size* = 512 and *batch_size* = 256 as they only take 2554s and 3053s to train and give a validation accuracy of 94,2%. Although they are not the best resulting models, they are trained in half the time or less than the rest of models with the best results, which could be a determining factor to choose it.

The actual model with the best results is the one trained with *embedding_size* = 128, *hidden_size* = 1024 and *batch_size* = 256 giving an accuracy of 94,8% on validation and a 99,923% accuracy in training. Despite being the best model of all the tried combinations, it was training for 7369s. We tried using it together with other modifications of the baseline model it gave memory errors as *hidden_size* = 1024 implies a bigger model.

We also found that the model trained with *embedding_size* = 128, *hidden_size* = 1024 and *batch_size* = 1024 had good results and a lower training time. This model gave a validation accuracy of 94,6% in a training time of 5938s. This model presents the same memory problems as the previous one.

To conclude we considered that either the *embedding_size* = 64, *hidden_size* = 512 and *batch_size* = 512 or *embedding_size* = 128, *hidden_size* = 512 and *batch_size* = 256 models were the best performing ones when trying to achieve the best model together with other modifications, but in case we just wanted to use the optimized baseline one the *embedding_size* = 128, *hidden_size* = 1024 and *batch_size* = 256 would be the best one.

5 Model configuration using max-pooling and tuned hyperparameters.

After observing the results of the hyperparameter tuning, we decided to experiment with the best models of the previous tests but using an embedding size of 128, since all of them were done with a size of 64. The results are shown in the following table:

Embedding size = 128							
RNN	Layers	Pooling	Hid. size	Direction	Train time	Valid Acc	Previous Acc
LSTM	2	max-mean sum	256	Unidirectional	1386s	93.1%	93.0%
LSTM	2	max-mean sum	512	Unidirectional	4699s	**94.0%**	93.0%
GRU	2	max-mean sum	256	Unidirectional	1684s	**93.3%**	93.1%
GRU	2	max-mean sum	512	Unidirectional	4492s	**93.8%**	93.1%
LSTM	2	max-mean concat	256	Bidirectional	3019s	93.6%	**93.8%**
GRU	2	max-mean concat	256	Bidirectional	1390s	**94.1%**	94.0%

Table 5: Modification of embedding size on the best unidirectional and bidirectional models.

These results match with the results obtained in the hyperparameter tuning study, embedding size of 128 achieves a higher accuracy yielding improved results in all experiments with exception to one. Another highlight these results show is that the two-layered bidirectional GRU model with concatenated max-mean pooling achieves a validation accuracy of 94.1%, which is the same value as the best one obtained for the moment.

In the following table, the best model configurations achieved so far are compared to some additional ones with max-pooling, instead of the max-mean pooling used previously. Baseline model configuration is taken for embedding, hidden and batch sizes.

RNN	Layers	Dropout	Direction	Valid Acc
LSTM	2	No	Uni	93.1%
LSTM	1	No	Bi	93.7%
LSTM	1	Yes	Bi	93.9%
GRU	1	No	Uni	92.7%
GRU	2	No	Uni	93.3%
GRU	2	Yes	Uni	93.6%
GRU	1	No	Bi	93.3%
GRU	1	Yes	Bi	93.5%

Table 6: Max-pooling layer for previously computed models.

None of the tried configurations seem to increase the accuracy value with respect to the previously mentioned one of 94.1%.

In this next table, all best models achieved across all sections are simultaneously compared.

Best models									
RNN	Layers	Dropout	Pooling	Emb.size	Hid. size	Batch size	Direction	Train time	Valid Acc
GRU	2	Yes	concat	64	256	256	Bi	2391s	**94.1%**
LSTM	1	No	max	64	512	256	Uni	3053s	94.2%
LSTM	1	No	max	128	512	256	Uni	3053s	94.2%
LSTM	1	Yes	max	128	512	256	Uni	5730s	94.2%
LSTM	1	Yes	max	128	512	512	Uni	-	94.2%

Table 7: Addition of tuned hyperparameters to different combinations.

To conclude these sections and considering a trade-off between training time and validation accruacy, the best model configuration achieved so far is composed of a one-layered unidirectional LSTM architecture with a max-pooling layer and a hidden size of 512. Then the dropout layer of 0.1, embedding size of 64 or 128 and batch size of 256 or 512 can be interchangeable. The best validation accuracy obtained so far is of 94.2%.

6 Convolutional Neural Network (CNN) comparison with RNN

In this section a CNN is defined and compared to the previous RNNs. The implementation consists on defining a new class with some changes on the RNN, starting by the addition of a list of convolutional layers and fully connected layer. The forward method has the input sequence passe through the embedding layer similarly to the RNN, but this time the embedding tensor is permuted in order to be correctly passed through each convolutional layer. Then, they are passed through a pooling layer and concatenated, before finally being passed through the dropout and fully connected layer.

Here is the code implementation:

```
class CharCNNClassifier(torch.nn.Module):
    def __init__(self, input_size, embedding_size, num_filters, filter_sizes,
    output_size, pad_idx=0):
        super().__init__()
        self.embed = torch.nn.Embedding(input_size, embedding_size, padding_idx=
    pad_idx)
        self.dropout = torch.nn.Dropout(0.1)
        self.convs = torch.nn.ModuleList([torch.nn.Conv1d(in_channels=embedding_size,
                        out_channels=num_filters,kernel_size=fs)
            for fs in filter_sizes])
        self.fc = torch.nn.Linear(len(filter_sizes) * num_filters, output_size)
    def forward(self, input, input_lengths):
        # Ensure input tensor is on the same device as the model's parameters
        input = input.to(self.embed.weight.device)
        # B x T
        embedded = self.embed(input)
        embedded = embedded.permute(1, 2, 0)
        # B x E x T
        conv_outs = []
        for conv in self.convs:
            conv_out = conv(embedded)
            pool_out = torch.nn.functional.max_pool1d(conv_out,kernel_size=conv_out.
    size(2))
            pool_out = pool_out.squeeze(-1)
            conv_outs.append(pool_out)
        # B x (F * len(filter_sizes))
        output = torch.cat(conv_outs, dim=1)
        output = self.dropout(output)
        # B x O
        output = self.fc(output)
        return output
```

The experiments have been done by modifying the number of filters (which are the number of convolutional layers), and having a list of (2,3,4,5) kernel sizes. some tests were done in the beginning with few layers without dropout, until we saw that not using it would yield worse valid accuracy and decided to use a 0.1 dropout for the rest of experiments. The results are shown on the next table:

CNN				
Num filters	Dropout	Emb. Size	Train Time	Valid Acc
32	No	64	189s	91.7%
32	Yes	64	185s	92.7%
64	Yes	64	594s	93.8%
100	No	64	288s	93.6%
100	Yes	64	291s	94.1%
128	Yes	64	333s	94.2%
256	Yes	64	628s	94.5%
512	Yes	64	1208s	94.7%
768	Yes	64	1742s	94.8%

Table 8: CNN with different number of layers (or filters).

As it can be seen, the higher the number of filter layers given to the CNN, the higher the validation accuracy of the model, as well as, the training time. We conclude that the number of layers that return better results is 768 filters. Given those results, we decided to experiment with hyperparameter tuned previously and see if the model with most accuracy improved. The results are shown on this table:

CNN					
Num filters	Dropout	Emb. Size	Hid. size	Train Time	Valid Acc
768	Yes	64	512	1327s	**94.9%**
768	Yes	128	512	2309s	94.7%

Table 9: CNN with 748 layer and hyperparameters tuned.

We conclude that the best configuration for a CNN model is 768 number of layers, dropout of 0.1, embedding size of 64 and hidden layer size of 512. Next we present a visual representation of the best model's training and validation behaviour.

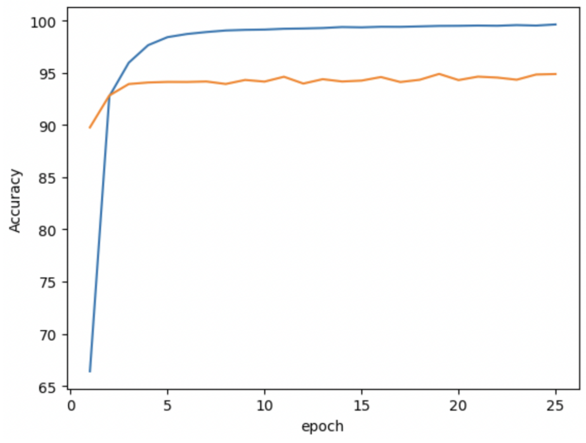

Figure 4: CNN with 768 layers model training (blue) and validation (orange) accuracy

As the behaviour in the graphic shows, the highest validation accuracy is achieved in the 19 epoch, with a fluctuation effect over all its curve. While the training accuracy starts to be constant at the 5 epoch.

7 Modification on input features: character n-grams

This time the input features have been modified to use character n-grams instead of individual characters, in specific, 2-grams or bigrams have been used. This model will be able to capture specific relationships between the words in a sentence, since bigrams consider pairs of consecutive characters. In order to implement bigrams as inputs, the library NLTK is required to tokenize input sequences, extract character 2-grams and add them to the dictionary *char_vocab* to its corresponding index. The code implementation for this input feature modification is done at the beginning and the end of the code notebook. It is as follows:

```
char_vocab = Dictionary()
(...)
import nltk
from nltk.util import ngrams
from nltk.tokenize import wordpunct_tokenize
for sentence in x_train_full:
    # Tokenize the sentence into bigrams
    tokenized_sentence = list(sentence)
    bigrams = ngrams(tokenized_sentence, 2)
    # Add the bigrams to the vocabulary
    for bigram in bigrams:
        char_vocab.add_token(''.join(bigram))
#From token or label to index
```

```python
n = 2
# create bigram function
def make_bigrams(text):
    return list(nltk.ngrams(text, n))
# convert x_train_full to bigrams
x_train_bigrams = [make_bigrams(line) for line in x_train_full]
x_train_idx = [np.array([char_vocab.token2idx[''.join(pair)] for pair in bigram]) for
    bigram in x_train_bigrams]
# convert y_train_full to indices
y_train_idx = np.array([lang_vocab.token2idx[lang] for lang in y_train_full])

(...)

def test(...)

x_test_txt = open(f'{INPUTDIR}/x_test.txt').read().splitlines()
x_test_bigrams = [make_bigrams(line) for line in x_test_txt]
x_test_idx = [np.array([char_vocab.token2idx.get(''.join(pair), unk_index) for pair in
     bigram]) for bigram in x_test_bigrams]
test_data = [(x, idx) for idx, x in enumerate(x_test_idx)]
```

The models that have been tried out to observe the change of performance when applying character n-grams as inputs are the best ones obtained from the first section. Those models did not have very good accuracy values, so we decided to try on this different technique to see if their values would increase. The model specifications and results are shown in the next table, note that the hidden size and batch size are 256 for all models:

Character n-grams								
RNN	Pooling	Layers	Emb. size	Direction	Dropout	Train Time	Valid Acc	Previous Acc
LSTM	sum	2	64	Uni	No	2360s	92.7%	**93.1%**
GRU	sum	2	128	Uni	No	2109s	93.3%	93.3%
LSTM	concat	2	64	Bi	No	4948s	93.4%	**93.8%**
GRU	oncat	2	64	Bi	No	4276s	93.0%	**94.9%**
LSTM	concat	2	64	Bi	Yes	4342s	93.6%	**94.0%**
GRU	concat	2	64	Bi	Yes	3821s	93.6%	**94.1%**

Table 10: Model combinations using n-grams.

None of them seem to achieve a greater performance on the validation accuracy with respect to their previously computed ones with individual characters instead of bigrams. Maybe further studies on this aspect could be done by analysis the behaviour on trigrams as inputs or by studying bigrams on baseline models with max pooling or hyperparameter tuning.

8 Description and comparative analysis with other classical language identification systems

For this last section we thought of using a transformer based model as studied in the previous lab. The idea was to use a BERT pre-trained model and tokenizer. To do so we changed some parts of the code as shown below.

```python
    tokenizer = BertTokenizer.from_pretrained('bert-base-multilingual-cased')

```

```python
def train_bert(model, optimizer, data, batch_size, max_seq_length, max_norm=1, log=False):
    (...)
    for batch in pool_generator(data, batch_size, max_seq_length, shuffle=True):
        # Get input and target sequences from batch
        X = [tokenizer.decode(d[0], skip_special_tokens=True) for d in batch]  # Convert tokens back to string
        X = [tokenizer.encode_plus(x, add_special_tokens=True, max_length=max_seq_length, truncation=True, pad_to_max_length=True) for x in X]  # Re-encode strings with truncation and padding
        input_ids = torch.tensor([x['input_ids'] for x in X], dtype=torch.long).to(device)
        attention_mask = torch.tensor([x['attention_mask'] for x in X], dtype=torch.long).to(device)
        token_type_ids = torch.tensor([x['token_type_ids'] for x in X], dtype=torch.long).to(device)
        y = torch.tensor([d[1] for d in batch], dtype=torch.long, device=device)
        model.zero_grad()
        output = model(input_ids, attention_mask=attention_mask, token_type_ids=token_type_ids, labels=y)
        loss = output.loss
        loss.backward()
        (...)

def validate_bert(model, data, batch_size, token_size):
    (...)
    with torch.no_grad():
        for batch in pool_generator(data, batch_size, token_size):
            # Get input and target sequences from batch
            X = [tokenizer.encode(d[0], add_special_tokens=True) for d in batch]
            X = torch.tensor([x[:token_size] for x in X], dtype=torch.long).to(device)
            y = torch.tensor([d[1] for d in batch], dtype=torch.long, device=device)
            answer = model(X, attention_mask=(X != tokenizer.pad_token_id).float())
            ncorrect += (torch.max(answer.logits, 1)[1] == y).sum().item()
            nsentences += y.numel()
        dev_acc = 100 * ncorrect / nsentences
    return dev_acc

# Define BERT model for sequence classification
bert_model = BertForSequenceClassification.from_pretrained('bert-base-multilingual-cased', num_labels=nlabels)
# Define optimizer
optimizer = AdamW(bert_model.parameters(), lr=5e-5)
# Define batch size and token size (maximum sequence length for BERT)
batch_size = 128
token_size = 512
# Training loop
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
bert_model.to(device)
epochs = 25
for epoch in range(epochs):
    train_accuracy = train_bert(bert_model, optimizer, train_data, batch_size, token_size)
    val_accuracy = validate_bert(bert_model, val_data, batch_size, token_size)
    print(f'Epoch {epoch+1}: Train Accuracy: {train_accuracy:.4f}, Validation Accuracy: {val_accuracy:.4f}')
# Save the trained model
torch.save(bert_model.state_dict(), 'bert_sequence_classification_model.pth')
```

We decided to choose for the following reasons: BERT stands out for sentence classification due to its comprehensive understanding of words within the context of a sentence, facilitated by bidirectional processing. Pre-trained on extensive text data, it acquires word and sentence representations crucial for accurate classification. Leveraging the powerful Transformer architecture, BERT efficiently captures complex linguistic patterns. Moreover, its capability for transfer learning enables easy adaptation to various tasks with minimal data. Finally, BERT's contextualized word embeddings ensure precise interpretation of word meanings within sentences, further enhancing its effectiveness in classification tasks.

Nevertheless, we were not able to obtain the model results due to lack of GPU and time restrains, the model execution takes a considerable amount of time.