POE Práctica 3

Sonia Castro Paniello
Eduard Ramon Aliaga

April 2023

1 Assignment tasks:

Hyperparameter optimization: try incrementing embedding size, RNN hidden size, and batch size. You can also compare different optimizers. Combination of the output of the max-pool layer with the output of a mean-pool layer (concatenation or addition).Add a dropout layer after the pooling layer.Comparative analysis of the performance with other RNNs or number of layers.

We have conducted various experiments, varying the embedding size, RNN hidden size, and batch size. Additionally, we have tried different RNN models, including both uni- and bidirectional LSTM and GRU models. We have also varied the number of layers and evaluated the effects of using max pooling or combining it with mean pooling, either by summing

```
max_pool, _ = padded.max(dim=0)
mean_pool = padded.mean(dim=0)
pooled = max_pool + mean_pool
```

or concatenating

```
pooled = torch.cat([max_pool, mean_pool], dim=1)
```

In some experiments, we applied a dropout of 0.1, some after pooling and some in the RNN as indicated in the following tables.

Uni directionals:

	Type	Layers	Pooling	Dropout	RNN dropout	Emb. size	Hid. size
Trial 1	LSTM	2	Max	No	No	64	256
Trial 2	GRU	2	Max	No	No	64	256
Trial 3	GRU	2	Max-Mean concat	No	No	64	256
Trial 4	GRU	2	Max-Mean concat	Yes	No	64	256
Trial 5	GRU	2	Max-Mean sum	No	No	64	256

Table 1: Hyperparameters in each trial.

	Type	Layers	Pooling	Dropout	RNN dropout	Emb. size	Hid. size
Trial 6	GRU	2	Max-Mean sum	Yes	No	64	256
Trial 7	GRU	2	Max-Mean sum	Yes	Yes	64	256
Trial 8	LSTM	2	Max-Mean sum	Yes	No	64	256
Trial 9	GRU	2	Max	No	Yes	64	256
Trial 10	GRU	1	Max	No	No	64	256
Trial 11	GRU	3	Max	No	No	64	256
Trial 12	GRU	2	Max	Yes	No	64	256

Table 2: Hyperparameters in each trial.

	Tain accuracy	Val. accuracy	Train Final Model	Num Param	Test acc. (Submission)
Trial 1	99.7	93.1	99.712	1608171	0.93739
Trial 2	99.8	93.3	99.765	1394155	0.94031
Trial 3	99.8	92.7	99.762	1454315	0.93341
Trial 4	99.7	92.2	99.762	1454315	0.93404
Trial 5	99.8	93.0	99.798	1394155	0.93531
Trial 6	99.7	93.1	99.666	1394155	0.93577
Trial 7	99.6	93.1	99.485	1394155	0.93509
Trial 8	99.6	93.1	99.597	1608171	0.93449
Trial 9	99.6	93.1	99.504	1394155	0.93988
Trial 10	98.7	93.0	98.625	999403	Not subm.
Trial 11	99.6	93.2	99.514	1788907	0.93588
Trial 12	99.6	93.6	99.566	1394155	0.94292

Table 3: Number of parameters, train, validation and test accuracy.

Upon concluding the multiple experiments, it has been observed that the RNN-GRU model with two unidirectional layers, combined with max pooling and dropout techniques to prevent overfitting, exhibits superior performance. It may be worth investigating the effects of further increasing the dropout rate or altering the batch size, embedding size, and hidden size for potential improvements.

Bidirectionals:

	Type	Layers	Pooling	Dropout	Emb. size	Batch size
Trial 1	LSTM	1	Max	No	64	256
Trial 2	GRU	1	Max	No	64	256
Trial 3	LSTM	1	Max-Mean concat	No	64	256
Trial 4	LSTM	1	Max-Mean concat	Yes	64	256
Trial 5	LSTM	1	Max-Mean sum	No	64	256
Trial 6	GRU	1	Max	Yes	156	300
Trial 7	LSTM	1	Max-Mean sum	Yes	64	256
Trial 8	LSTM	1	Max-Mean concat	Yes	64	256
Trial 9	LSTM	1	Max	Yes	64	256

Table 4: Hyperparameters in each trial.

	Tain accuracy	Val. accuracy	Train Final Model	Num Param	Test acc. (Submission)
Trial 1	99.8	93.7	99.817	1471723	0.94482
Trial 2	99.7	93.3	99.610	1306859	0.94001
Trial 3	99.8	93.5	99.833	1592043	0.94109
Trial 4	99.6	93.3	99.628	1592043	0.94024
Trial 5	99.9	93.6	99.847	1471723	0.94285
Trial 6	99.8	93.5	99.700	2459147	Not subm.
Trial 7	99.9	93.8	99.929	1471723	Not subm.
Trial 8	99.6	93.3	99.628	1592043	Not subm.
Trial 9	99.4	93.9	99.404	1471723	94.641

Table 5: Number of parameters, train, validation and test accuracy.

Regarding bidirectional RNNs, it has been observed that the most optimal model is a single-layer LSTM combined with max pooling and dropout techniques. Similarly to the previous case, further experiments could be conducted to potentially enhance the model's performance.

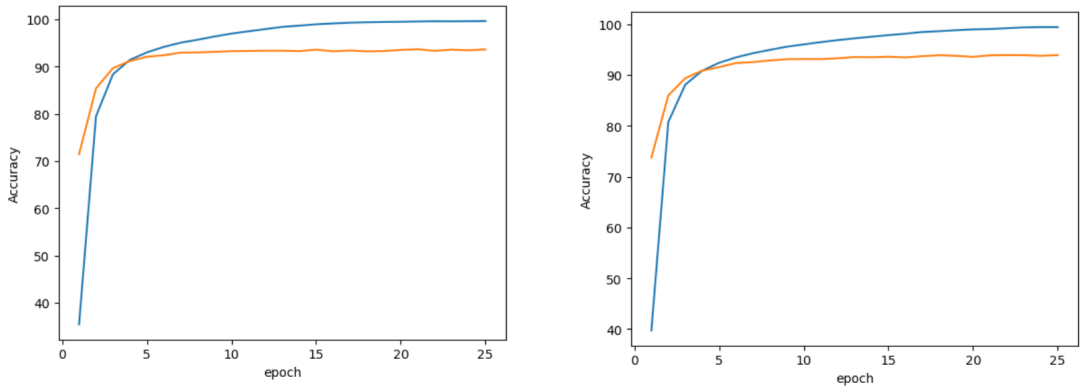

Figure 1: Blue-Trainning accuracy & Orange-Validation accuracy

From epoch 5 onwards, it is apparent from the plots that the validation accuracy remains stagnant, whereas the training accuracy exhibits marginal improvement. This suggests that alternative regularization techniques could be employed to mitigate overfitting.

2 Other optional tasks:

Comparative analysis with other DNN architectures such as convolutional neural networks.

For this optional part we implemented a Convolutional neural network creating a new class:

```
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.convs = torch.nn.ModuleList([
            torch.nn.Conv1d(in_channels=embedding_size, out_channels=num_filters,
                kernel_size=fs)
            for fs in filter_sizes
        ])
        self.dropout = torch.nn.Dropout(p=0.1)
        self.fc = torch.nn.Linear(len(filter_sizes) * num_filters, output_size)

    def forward(self, input, input_lengths):
        # B x T
        embedded = self.embed(input)
        # B x T x E
        embedded = embedded.permute(1, 2, 0)
        # B x E x T
        conv_outs = []
        for conv in self.convs:
            conv_out = conv(embedded)
            # B x F x (T - filter_size + 1)
            pool_out = torch.nn.functional.max_pool1d(conv_out,
                kernel_size=conv_out.size(2))
            # B x F x 1
            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
```

We define in the init method the layers of the network: an embedding layer, a list of convolutional layers, a dropout layer, and a fully connected layer.

The forward method defines the forward pass of the network. The input sequence is first passed through the embedding layer, and then the resulting tensor is permuted so that the time dimension is in the correct position for the convolutional layers. Then is passed through each convolutional layer, and the resulting tensors are max-pooled across the time dimension. The pooled tensors are concatenated and passed through the dropout layer

and the fully connected layer to obtain the final class prediction.

It can be seen that an input of the model is filter sizes, which is a list of integers with length equal to the number of filters we want to use in the network (aka the number of layers). Here are the executions we have done with this model:

CNN:

	Type	Layers	Pooling	Dropout	Filters size	Emb. size	Batch size
Trial 1	CNN	32	Max	Yes	2	64	256
Trial 2	CNN	32	Max	Yes	5	64	256
Trial 3	CNN	32	Max	Yes	5	128	256
Trial 4	CNN	40	Max	Yes	3	64	256
Trial 5	CNN	40	Max	Yes	5	64	256
Trial 6	CNN	40	Max	Yes	5	256	256
Trial 7	CNN	40	Max	Yes	8	64	256
Trial 8	CNN	50	Max	Yes	5	64	256
Trial 9	CNN	60	Max	Yes	3	64	256
Trial 10	CNN	60	Max	Yes	5	64	256

Table 6: Hyperparameters in each trial.

	Tain accuracy	Val. accuracy	Train Final Model	Num Param	Test acc. (Submission)
Trial 1	99.6	94.3	99.589	1130219	0.94885
Trial 2	99.7	94.4	99.631	1261291	0.95162
Trial 3	99.5	94.4	99.501	2280683	0.94927
Trial 4	99.6	94.4	99.593	1376747	0.94962
Trial 5	99.6	94.7	99.587	1581547	0.95170
Trial 6	99.3	94.0	99.368	5192683	0.94909
Trial 7	99.6	94.4	99.581	1888747	0.94385
Trial 8	99.6	94.6	99.549	2081947	0.95207
Trial 9	99.5	94.1	99.495	2232747	0.95107
Trial 10	99.5	94.7	99.510	2693547	0.95041

Table 7: Number of parameters, train, validation and test accuracy.

As it can be seen the best accuracy is achieved with 50 layers and a kernel size of 5. Taking a look to all the trials, the intuition is that as we increase the number of layers, the accuracy also should improve but that is not the case. The prove is that with 60 layers and the same kernel size the value decreases due to overfitting.

We also tried to increase the embedding size in some cases but we confronted the same result explained before: trials with a minor embedding size had more accuracy. Having said this, we did not even try to increase the batch size as it would sure end up with the same problem.

Another thing to highlight of the executions is that having more parameters to optimize

doesn't always lead to better results. Although trial 8 has a pretty huge amount of them, trial 6 doubles this number and only gave us more computation time and complexity to achieve less accuracy.

Here we plot the training accuracy (blue) vs validation acccuracy (orange) of the best model:

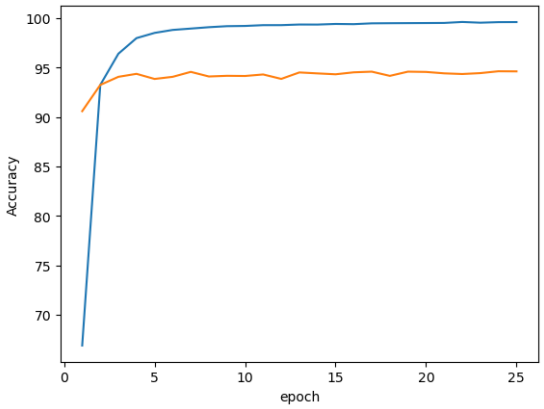

Figure 2: Blue-Trainning accuracy & Orange-Validation accuracy

From epoch 5 to 25 the change in the values is nearly unnoticeable and there is obvious overfitting (a liitle bit) as training accuracy is higher than validation accuracy.

Other input features: modify the code to use other features as input such as words, character n-grams, or character counts.

We have tried the use of bigrams instead of individual characters as the basic unit of analysis. Bigrams refer to pairs of consecutive letters or characters in a sentence. By analyzing bigrams, the model can capture more complex patterns and relationships between the words in a sentence. Since the use of words significantly increased the size of the vocabulary and the number of parameters required by the model.

```
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))
```

The code provided uses the NLTK library to tokenize each sentence in the x_train_full, extract bigrams from the sentences, and add these bigrams to a character-level vocabulary called $char\_vocab$.

```
import nltk
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])
```

Takes each sentence inx_train_full, converts it to bigrams using the $make\_bigrams()$ function, and then converts each bigram to its corresponding index in $char\_vocab$.

We conducted an RNN experiment with bigrams instead of characters as inputs. Specifically, we used a bidirectional GRU model with a single layer.The training accuracy was 99.9, the validation accuracy was 93.1, and the final model's training accuracy was 99.831 and a test accuracy of 93.5. The model had 20,521,387 parameters.

We conducted a second experiment because it seemed that the model was overfitting. We added a dropout layer with a value of 0.1, which resulted in a training accuracy of 99.6 and a validation accuracy of 93.2. The final model's training accuracy was 99.334, and it had 20,521,387 parameters. The test accuracy was of 93.302

After trying it out with RNN, we wanted to see how it would work with the best model of CNN (trial 8) and it resulted in 99.6 train accuracy, 94.2 validation accuracy, 82657195 parameters, 99.593 final model's training accuracy and 0.94747 test accuracy. After seeing that the result was below the one obtained before and that the execution time was scarce (this executions lasted between 4 and 5 hours) we did not try more models for this kind of input.

Description and comparative analysis with other classical language identification systems. You can use an existing implementation, but you will have to describe it in your own words.

A transformer has been implemented for language prediction. The input sequence is first passed through an embedding layer, after which a mask is created to exclude padded tokens during training. The transformer consists of the specified number of layers and processes the masked embeddings. The outputs of the transformer are summed across the sequence dimension, and a linear layer is used to project the result into the output dimension (number of languages) for language prediction.

To avoid out-of-memory errors during training, the sequence length was limited to 200 and the batch size was reduced to 64 using gradient accumulation with a step of 4. The bigram model was not used due to the same memory limitations, and the number of layers could not be increased or modified for the same reason.

```
class TransformerClassifier(torch.nn.Module):

    def __init__(self, input_size, embedding_size, hidden_size, output_size,
        num_layers=1, num_heads=4, dropout=0.1, pad_idx=0):
```

```
        super().__init__()
        self.embed = torch.nn.Embedding(input_size, embedding_size,
            padding_idx=pad_idx)
        self.encoder_layer = torch.nn.TransformerEncoderLayer(embedding_size,
            num_heads, hidden_size, dropout)
        self.encoder = torch.nn.TransformerEncoder(self.encoder_layer, num_layers)
        self.h2o = torch.nn.Linear(embedding_size, output_size)

    def forward(self, input, input_lengths):
        # input: [seq_len, batch_size]
        embedded = self.embed(input)
        mask = (input == 0).transpose(0, 1) #mask out the padding tokens
        encoded = self.encoder(embedded, src_key_padding_mask=mask)
        output = encoded.sum(dim=0)
        output = self.h2o(output)
        return output
```

```
def train(model, optimizer, data, batch_size, token_size, max_norm=1,
    accumulation_steps=4, log=False):
    model.train()
    total_loss = 0
    ncorrect = 0
    nsentences = 0
    ntokens = 0
    niterations = 0
    max_T = 300

    for batch_idx, batch in enumerate(pool_generator(data, batch_size,
        token_size, shuffle=True)):
        # Get input and target sequences from batch
        X = [torch.from_numpy(d[0]) for d in batch]
        X = [x[:max_T] for x in X]
        X_lengths = [x.numel() for x in X]
        ntokens += sum(X_lengths)
        X_lengths = torch.tensor(X_lengths, dtype=torch.long)
        y = torch.tensor([d[1] for d in batch], dtype=torch.long, device=device)

        # Pad the input sequences to create a matrix
        X = torch.nn.utils.rnn.pad_sequence(X).to(device)
        model.zero_grad()

        output = model(X, X_lengths)
        loss = criterion(output, y)

        loss = loss / accumulation_steps # divide loss by accumulation steps

        loss.backward()

        if (batch_idx + 1) % accumulation_steps == 0: # perform weight update
            after accumulation_steps iterations
            torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm) #
                Gradient clipping
            optimizer.step()
            model.zero_grad()
```

```
        # Training statistics
        total_loss += loss.item() * accumulation_steps
        ncorrect += (torch.max(output, 1)[1] == y).sum().item()
        nsentences += y.numel()
        niterations += 1

    # Perform final weight update if there are remaining gradients
    if accumulation_steps > 1 and (batch_idx + 1) % accumulation_steps != 0:
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm) # Gradient
            clipping
        optimizer.step()
        model.zero_grad()

    total_loss = total_loss / nsentences
    accuracy = 100 * ncorrect / nsentences

    if log:
        print(f'Train: wpb={ntokens//niterations}, bsz={nsentences//niterations},
            num_updates={niterations}')

    return accuracy
```

	Layers	Heads	Pooling	Epoch	Train acc.	Val acc.	Train Final Model
Trial 1	1	4	Sum	25	87.7	87.0	87.927
Trial 2	2	4	Sum	25	87.5	86.8	87.794
Trial 3	1	4	Max	25	86.9	86.0	87.082
Trial 4	1	4	Sum	50	89.2	87.7	89.165

Table 8: Hyperparameters and accuracy in each trial.

One can observe that the model's optimal performance was achieved with a single layer and summation pooling. As a result, we proceeded to train this model for 50 epochs in order to enhance its performance. Despite our efforts, it is worth noting that the other models employed in this exercice have yielded superior outcomes. As depicted in the subsequent plot, in this instance, overfitting is not as pronounced.

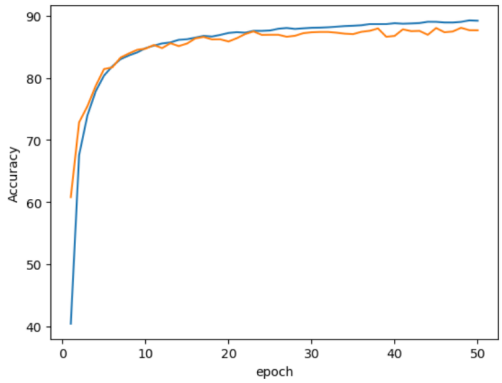

Figure 3: Blue-Trainning accuracy & Orange-Validation accuracy

Finally and nearly at the end of the deadline we implemented a Bert model using a pretrained one. Here is the code:

```
# This Python 3 environment comes with many helpful analytics libraries installed
import random
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import torch # Deep learning framework
import torch.nn.functional as F
import time
from types import SimpleNamespace
from collections import Counter
import os
import re
import pathlib
import array
import pickle
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import pandas as pd
import math
import string
import sklearn
import nltk
import tensorflow as tf

# Input data files are available in the "../input/" directory.
import os
INPUTDIR = '../input/wili6'
print(os.listdir(f'{INPUTDIR}'))

#Init random seed to get reproducible results
seed = 1111
random.seed(seed)
np.random.RandomState(seed)
torch.manual_seed(seed)

# Any results you write to the current directory are saved as output.
x_train_full = open(f'{INPUTDIR}/x_train.txt').read().splitlines()
y_train_full = open(f'{INPUTDIR}/y_train.txt').read().splitlines()
print('Example:')
print('LANG =', y_train_full[0])
print('TEXT =', x_train_full[0])

!pip install transformers

from transformers import BertTokenizer, TFBertModel, BertConfig,
    TFBertForSequenceClassification
tokenizer=BertTokenizer.from_pretrained('bert-base-uncased', do_lower_case=True)

def mask_inputs_bert(x_train_full, max_len=200):
    input_ids =[]
    attention_masks =[]
    for x in x_train_full:
```

```
        encoded_dict = tokenizer.encode_plus(x, add_special_tokens = True,
            max_length = max_len, pad_to_max_length = True, return_attention_mask
            = True)
        input_ids.append(encoded_dict['input_ids'])
        attention_masks.append(encoded_dict['attention_mask'])
    input_ids = tf.convert_to_tensor(input_ids)
    attention_masks = tf.convert_to_tensor(attention_masks)

    return input_ids, attention_masks

class Dictionary(object):
    def __init__(self):
        self.token2idx = {}
        self.idx2token = []

    def add_token(self, token):
        if token not in self.token2idx:
            self.idx2token.append(token)
            self.token2idx[token] = len(self.idx2token) - 1
        return self.token2idx[token]

    def __len__(self):
        return len(self.idx2token)
lang_vocab = Dictionary()
# use python set to obtain the list of languages without repetitions
languages = set(y_train_full)
for lang in sorted(languages):
    lang_vocab.add_token(lang)
print("Labels:", len(lang_vocab), "languages")
y_train_full = np.array([lang_vocab.token2idx[lang] for lang in y_train_full])

from sklearn.model_selection import train_test_split
train_input, val_input, train_label, val_label = train_test_split(x_train_full,
    y_train_full)

train_tokenized, train_mask = mask_inputs_bert(train_input[:15000], 200)
val_tokenized, val_mask = mask_inputs_bert(val_input[:15000])

train_label = tf.convert_to_tensor(train_label[:15000])
val_label = tf.convert_to_tensor(val_label[:15000])

bert_model =
    TFBertForSequenceClassification.from_pretrained('bert-base-uncased',
    num_labels =235)

loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
metric = tf.keras.metrics.SparseCategoricalAccuracy('accuracy')
optimizer = tf.keras.optimizers.Adam(learning_rate = 2e-5, epsilon=1e-08)
bert_model.compile(loss=loss, optimizer= optimizer, metrics=[metric])

history = bert_model.fit([train_tokenized, train_mask], train_label[:15000],
    batch_size=32, epochs=25, validation_data = ([val_tokenized, val_mask],
    val_label[:15000]))
```

The main changes of the code are: The tokenization of input as it is tokenized using the BERT tokenizer from the transformers library.

The instantiation of a BERT model using the TFBertForSequenceClassification class from the transformers library, with the number of labels set to 235. The model is compiled using the Adam optimizer, sparse categorical cross-entropy loss function, and sparse categorical accuracy metric.

Finally, the training of the model using the fit method, passing in the preprocessed training and validation data, batch size, number of epochs, and validation data for monitoring performance during training.

The performance metrics of the model can be observed as follows: the loss value is 0.0425 and the accuracy score is 0.9893. On the other hand, the validation loss is 0.5566 and the validation accuracy is 0.8823, indicating an evident overfitting. However, due to the late stage of the project delivery, we did not have sufficient time to further experiment with this architecture.