Natural Language Interaction Report

Group number 2

Anna Ricker (61287), Rodrigo Santos (60552)

1 Requirements

The aim of this project was to train a Neural network (NN) that can answer a given question with yes or no based on a question and a context as input. Instead of training a NN from the ground, another precondition was to use the pre-trained model Roberta-base [4] and fine-tune it with the superglue BoolQ dataset [5]. Elements of the dataset consist of a question string, a context string, and a binary label. This label defines the respective answer, yes or no, to the given question based on the respective context, whereas 0 stands for False and 1 means True. Furthermore, a batch size of 12 and a number of epochs of 2 were also predetermined. The implemented program should ensure that a local copy of the tuned model is saved and it prints the neural network's performance (accuracy) after training. The program must also include a method called test (passage, question), which accepts a context and a question as input. The method should then be able to answer the yes/no question with the help of the neural network.

2 Implementation

As implementation environment, we used the Google Colab platform and set up a Jupyter Notebook file. As a programming language, we used Python with its deep learning library Pytorch. Furthermore, we used the platform Hugging Face on the one hand to import the respective tools and models, on the other hand, to work with their extensive documentation to implement the given task.

Our implementation contains the following steps:

1. Install all necessary tools
2. Load the Boolq dataset
3. Load the respective tokenizer for our roberta-base model
4. Prepare the Dataset for training by tokenizing the dataset and deleting and renaming columns of the dataset so it suits the Roberta-base model.
5. Load the pre-trained model
6. Fine-tune the model by training it with the prepared dataset
7. Evaluate the training result with the validation data of the Boolq dataset and print the results.
8. Save the fine-tuned model
9. Implement the test method to use the fine-tuned model for executing the question-answering task.

Before loading the dataset we had a look at the dataset information on the hugging face webpage. That gave us the information on how to load the dataset. Following we imported the tokenizer to tokenize the dataset. It is very important to use the tokenizer that was used for the roberta-base model, so that the dataset fits the model and can be used for fine-tuning. Because the roberta-base model just expects one input, we had to implement a tokenize function for our dataset to merge our question and passage to a combined input. Following the tokenize function is displayed:

```
def tokenize_function(example):
    return tokenizer(example["question"], example["passage"], truncation=True)
```

For each element of the dataset, the tokenizer function will be applied. The tokenizer takes the question and the passage of the respective element and will tokenize them together. They will be separated by a separator. This tokenize function will then be mapped on each element of the raw dataset. The DataCollator will then form batches out of the dataset and will add the necessary padding per each batch. This is more space efficient than adding the same padding to the whole dataset because it is just necessary to have the same length of input strings per batch. The next step is to delete unnecessary columns in the dataset and rename it so it fits the model. In the end, the only columns needed for the training are labels, input_ids, and attention_mask. Then the DataLoader will do the final preparation of the dataset to simplify loading the dataset into the model, batching it to 12 elements per batch, and shuffling the elements of the dataset.

The most important element of the code is the implementation of the training. First, we load the pre-trained model, configure the learning rate in our optimizer, and make sure that our model is trained on the GPU if available. We set the number of epochs to 2 and calculate the number of training steps respectively. Then we define the parameters of our learning rate scheduler. For each batch in each epoch, we get the respective batch, run the batch through the model, calculate the loss, and do the backpropagation of the training. Afterwards, we prepare the optimizer and the learning rate scheduler for the next step, as seen in the following code snippet:

```
model.train()
for epoch in range(num_epochs):
    for batch in train_dataloader:
        batch = {k: v.to(device) for k, v in batch.items()}
        outputs = model(**batch)
        loss = outputs.loss
        loss.backward()

        optimizer.step()
        lr_scheduler.step()
        optimizer.zero_grad()
        progress_bar.update(1)
```

After the training is finished we take the evaluation set of the dataset and calculate the accuracy of the trained model. For the metric we load the respective metric for our dataset:

```
metric = load_metric("super_glue", "boolq")
```

With each batch's output from the model we then compare the prediction with the actual label of each element and calculate the accuracy of the model at the end.

After trying out different hyperparameters, we got the best result with a learning rate of 3e-5 and 100 warmup steps at the beginning. With that, in the best case, our model's accuracy became 78%.

We saved the model and loaded it again in the following step to have it available for our test method. As previously explained the test method gets a question and a context as input. This input first needs to be encoded, so that we can input it in our pre-trained model. This is done with the tokenizer and its encode_plus() method.

In previous training and evaluation sessions, we had problems with executing steps when not all steps are running on the same device. To also assure that this problem will not occur in our method, we define our device again as previously explained. To make sure the model and the encoded_input run on the same device we each configure them to run on this device with the to(device) function:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
input_ids = encoded_input['input_ids'].to(device)
attention_mask = encoded_input['attention_mask'].to(device)
```

Then we run the encoded_input through our model and post-process the output so that the method can simply return a True or False depending on the result. As last step, we use the test on the examples from the task statement

2.1 Problems

While implementing this task, we ran into a few problems. The first problem was that our training crashed because of RAM problems depending on our set hyperparameters. Also, we figured out later that we evaluated the data with the wrong metric. It was important to use the right metric for the dataset chosen. We had to read the superglue evaluate metric documentation to solve this bug [3]. One last bug was the problem mentioned before, if the training and evaluation do not run on the same device (GPU or CPU), it resolves into problems. Due to that we always had to make sure to run the whole process on the same device.

3 Contributions

For our project work, we decided to do the implementation via pair programming. In that way, we programmed most of the code. For bug fixing, we also split up to solve multiple problems at the same time. In general, we both spend around 7 hours in pair programming, 8 hours programming on our own, and 4 hours writing the report.

References

[1] Hugging Face. A full Training. https://huggingface.co/learn/nlp-course/chapter3/4?fw=pt.

[2] Hugging Face. Evaluate. https://huggingface.co/docs/evaluate/index.

[3] Hugging Face. Metric: Super Glue. https://huggingface.co/spaces/evaluate-metric/super_glue.

[4] Hugging Face. Roberta Base. https://huggingface.co/roberta-base.

[5] Hugging Face. Super glue BoolQ Data. https://huggingface.co/datasets/super_glue/viewer/boolq.