



Code Optimization Project

Advanced Verification and Validation A.Y. 2024/25

Giovanni Spaziani (295397)

The following project aims to generate optimized code from (slow) code using Large Language Models.

For this purpose, the PIE Dataset available from the GitHub repository was used (<https://github.com/LearningOpt/pie>). This repository contains source code in slow-fast pairs. The idea is to train a Language Model on these pairs to obtain optimized code from a non-optimized version.

The project was divided into two scenarios:

- **Scenario 1:** In which the rewritten code is generated in the optimized version.
- **Scenario 2:** In which the code is generated in diff-based representation.

For both scenarios, the Dataset was divided into training and test sets. Specifically, for the second scenario, it was necessary to convert the optimized code into a diff-based version of the input code. To do this, the Python **difflib** library was used.

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 #define int long long
6
7 signed main() {
8     int n;
9     cin >> n;
10
11     int count = 0;
12     int k = sqrt(n);
13     int maxi = 1;
14
15     for (int i = 1; i < n; i++) {
16         for (int j = 2; j < n; j++) {
17             if (pow(i, j) <= n) {
18                 count = pow(i, j);
19                 maxi = max(count, maxi);
20                 // cout << i << " " << j << " " << maxi << endl;
21             }
22         }
23     }
24
25     cout << maxi;
26 }
27 }
```

src

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 int main() {
6     int n;
7     cin >> n;
8
9     int mx = 1;
10
11     for (int i = 2; i * i <= n; ++i) {
12         int s = i;
13         while (s * i <= n)
14             s *= i;
15         mx = max(s, mx);
16     }
17
18     cout << mx;
19     return 0;
20 }
```

tgt

The above code (from Scenario 1) is intended to find the largest integer power a^b (with $a \geq 1$, $b \geq 2a$) that is less than or equal to a given number n . The **src** version explores all pairs (i, j) and computes i^j , with many unnecessary calculations. The **tgt** version is optimized: for each base i , it multiplies i until the result remains $\leq n$, avoiding the use of `pow()`, which is slower and less precise.

```

1  #include<bits/stdc++.h>
2
3  using namespace std ;
4
5  int main(){
6
7      long long a;
8
9      cin>>a;
10
11     cout<<(a+1)/2<<endl;
12
13 }

```

src

```

1  --- src // "original" file
2  +++ tgt // "modified" file
3  @@ -1,15 +1,13 @@
4  -#include<bits/stdc++.h>
5  -
6  -using namespace std ;
7  -
8  +#include <stdio.h>
9  +
10 - int main(){
11 -
12 -     long long a;
13 -
14 -     cin>>a;
15 -
16 -     cout<<(a+1)/2<<endl;
17 +
18 +     int a;
19 +     scanf("%d", &a);
20 +     (a%2 == 0)? printf("%d", a/2) : printf("%d", a/2 + 1);
21 +     return 0;
22 }

```

tgt

The above code (from Scenario 2) reads an integer *a* as input. It calculates half of *a*, rounded up. If *a* is even, it returns exactly half of *a*; if *a* is odd, it automatically adds 1. The **src** version uses `cin`, `cout`, and `long long`, which are slower, more cumbersome, and take up more memory (`long long` = 8 bytes). The **tgt** version is lighter and more efficient thanks to the use of `scanf`, `printf` and a more compact type (`int` = 4 bytes).

As LLM, the CodeT5+ model was used, which is publicly available in the GitHub repository (<https://github.com/salesforce/CodeT5/tree/main/CodeT5%2B>). It's an artificial intelligence model that understands and generates source code. It supports tasks such as code completion, translation, optimization, and debugging, and has been trained on a large scale to work across multiple programming languages. Code T5+ models can be used for a wide range of **Code Understanding and Generation** tasks. Due to computational limitations, the 220m model, which uses this number of parameters, was used in the following project.

To use this library in Python, you need to import the libraries as follows:

```
from transformers import AutoModelForSeq2SeqLM, AutoTokenizer, TrainingArguments, Trainer
```

and invoke the model via the code:

```
tokenizer = AutoTokenizer.from_pretrained("Salesforce/codet5p-220m")
```

The next steps to be performed are the tokenization of the source and target codes, which can be obtained through the following lines of code:

```

inputs = tokenizer(
    example["src"],
    max_length=512,
    padding="max_length",
    truncation=True
)
targets = tokenizer(

```

```
example["tgt"],
max_length=512,
padding="max_length",
truncation=True
)
```

Tokenization is required for subsequent training:

```
train_data = datasets.map(
    preprocess_function,
    batched=True,
    remove_columns=datasets.column_names,
    num_proc=4,
    load_from_cache_file=False,
)
```

In this case, **preprocess_function** is the function that generates the tokenization seen previously.

The trained model is then saved for the next testing step:

```
model.save_pretrained(final_checkpoint_dir)
```

Once the model is saved, it can be loaded via:

```
model = AutoModelForSeq2SeqLM.from_pretrained(model_path, trust_remote_code=True)
```

Where **model_path** is the path to the saved model.

To test the model, initialize the pipeline:

```
pipe = pipeline("text2text-generation", model=model, tokenizer=tokenizer, truncation=True)
```

Pipeline is a high-level interface to Hugging Face that simplifies the use of pre-trained models for specific tasks. In this case, the line creates a text-to-text generation pipeline that uses a specific model and tokenizer, automatically handling preprocessing (tokenization and automatic truncation of inputs to meet model length constraints) and output text generation. It is then used on each **input_code** present in the test set data:

```
prediction = pipe(input_code, max_new_tokens=512)[0]["generated_text"]
```

where **input_code** is the **src** field of the PIE data.

Finally, to evaluate the quality of the code optimized by the model, the gem5 simulator, known for its primary use in run-time system optimizations, was used.

The simulator is publicly available at the repository (<https://github.com/gem5/gem5>).

The purpose of this simulation will be to compile the **slow** and **predicted** codes, where **predicted** is the code generated by the model, and evaluate the performance of the executables on the inputs present in the dataset available in the **merged_test_cases** folder of the PIE repository. The merged_test_cases folder contains folders named problem-id, which contain input-output .txt file pairs.

The former are provided as input to the cpp codes, while the latter are used to check the correctness of the predicted code.

To automate the process, a script was used that overwrites the code.cpp file with the input and predicted codes respectively, which is compiled by the following Python command:

```
run_command(  
    "g++ -std=c++17 -O3 -w code.cpp -o code.out 2> results/logs/compile.log",  
    "Compilazione C++"  
)
```

followed by a check of the compile.log file to ensure there are no compilation errors. If any are present, the corresponding code is discarded and considered incorrect.

In Scenario 2 it was necessary to convert the predicted code from the diff-based version to the non-diff-based version using the difflib library.

```
differ = difflib.Differ()  
diff = list(differ.compare(dato["input"], dato["prediction"]))  
reconstructed = list(difflib.restore(diff, 2))
```

If the compilation is successful, the generated executable is invoked on each input related to the problem associated with the compiled code:

```
checkStatus = run_command(f"timeout 120s build/X86/gem5.opt -q --outdir=results --stats-  
file=stats.txt --silent-redirect -r --stdout-file=logs/gem5_stdout.log --stderr-  
file=logs/gem5_stderr.log simulate.py code.out {\"merged_testcases/merged_test_cases/\" +  
dato[\"problem\"] + \"/\" + file}, \"Simulazione gem5\")
```

The Linux shell 2-minute timeout command has been added to prevent infinite loops that could arise from pathological input file configurations.

For the evaluation purposes, the following 3 metrics were used:

1. **Correct (%)**: The percentage of generated programs that successfully execute and produce the expected output for all available test cases.
2. **Speedup**: The absolute improvement in terms of execution time. It is defined as T_I / T_G , where T_I and T_G denote the execution times of the input and generated programs, respectively. The speedup can be set equal to 1 when the generated programs are either incorrect or slower than the input program. The reported measure is summarized by the arithmetic mean.
3. **Percent Optimized (%Opt)**: The percentage of generated code that is both correct and faster across the entire testing set. A generated program is considered faster if it

achieves a speed improvement of at least 10%, corresponding to a Speedup of at least 1.1x.

For both scenarios, three models were generated with different training parameters:

Parameters	Model 1	Model 2	Model 3
Epochs	3	100	200
Max source len	512 token	512 token	512 token
Max target len	512 token	512 token	512 token
Lr (learning rate)	5e-5	5e-5	5e-5
Lr warmup steps	200	200	50
Batch size per replica	4	8	8
Grad acc steps	4	4	4
Local rank	-1	-1	-1
Deepspeed	None	None	None
Fp16	True	True	True

Table 1 – Scenario 1

Parameters	Model 1	Model 2	Model 3
Epochs	3	100	10
Max source len	512 token	512 token	512 token
Max target len	512 token	512 token	512 token
Lr (learning rate)	5e-5	5e-5	3e-4
Lr warmup steps	200	200	50
Batch size per replica	4	8	8
Grad acc steps	4	4	4
Local rank	-1	-1	-1
Deepspeed	None	None	None
Fp16	True	True	True

Table 2 – Scenario 2

- **Epochs:** an epoch corresponds to a complete pass over the entire training dataset.
- **Max source len and Max target len:** they define, in tokens, the length of the model's input and output. They are used to truncate excessively long texts.
- **Lr:** the speed at which the model updates its weights. The higher it is, the faster it learns; the lower it is, the slower it learns. In the first case, it may diverge; in the second, it may converge better.
- **Lr warmup steps:** number of initial steps during which the learning rate increases linearly from 0 to the value set in Lr, helping the model minimize error (loss). Useful for stabilizing training in the early stages.
- **Batch size per replica:** how much data is processed simultaneously by each GPU. "replica" because each replica of the model (on each GPU) processes the data.
- **Grad acc steps:** number of steps during which the model accumulates the gradient (which indicates how to correct itself to make fewer errors) before performing a weight update. Useful for simulating a larger batch without consuming more GPU memory.
- **Local rank:** used when training on multiple GPUs. The default value of -1 indicates that training is performed on a single machine/GPU.

- **Deepspeed:** allows to enable optimized training via the DeepSpeed library by specifying a .json configuration file. If set to None (default), it is disabled.
- **Fp16:** if specified, enables half-precision training, reducing GPU memory usage and speeding up training (depends on the HW used).

The following image represents an example of a model training run.

```
C:\Users\Giovanni Spaziani\Desktop\Università\MAGISTRALE\2 anno\Vittorio Cortellesa - Advanced Verification and Validation\Progetto\CodeT5\CodeT5>python .\tune_codet5p_seq2seq.py --save-dir .\saved_models\epochs_200 --lr-warmup-steps 50
{'batch_size_per_replica': 8,
 'cache_data': 'cache_data/summarize_python',
 'data_num': -1,
 'deepspeed': None,
 'epochs': 200,
 'fp16': True,
 'grad_acc_steps': 4,
 'load': 'Salesforce/codet5p-220m',
 'local_rank': -1,
 'log_freq': 10,
 'lr': 5e-05,
 'lr_warmup_steps': 50,
 'max_source_len': 512,
 'max_target_len': 512,
 'save_dir': '.\saved_models\epochs_200',
 'save_freq': 500}
==> Loaded 42295 samples
==> Loaded model from Salesforce/codet5p-220m, model size 222882048
Starting main loop
0%|          | 0/100 [00:00<?, ?it/s]P
assing a tuple of 'past_key_values' is deprecated and will be removed in Transformers v4.48.0. You should pass an instance of 'EncoderDecoderCache' instead,
e.g. 'past_key_values=EncoderDecoderCache.from_legacy_cache(past_key_values)'.
{'loss': 8.0653, 'grad_norm': 21.721532821655273, 'learning_rate': 0.0, 'epoch': 0.0}
{'loss': 6.9204, 'grad_norm': 25.381534576416016, 'learning_rate': 9e-06, 'epoch': 0.01}
{'loss': 4.9503, 'grad_norm': 5.408700942993164, 'learning_rate': 1.9e-05, 'epoch': 0.02}
{'loss': 4.1191, 'grad_norm': 4.986625671386719, 'learning_rate': 2.9e-05, 'epoch': 0.02}
{'loss': 4.0311, 'grad_norm': 5.211935043334961, 'learning_rate': 3.9000000000000006e-05, 'epoch': 0.03}
{'loss': 3.4121, 'grad_norm': 4.479816913604736, 'learning_rate': 4.9e-05, 'epoch': 0.04}
{'loss': 3.0898, 'grad_norm': 4.345370292663574, 'learning_rate': 4.1e-05, 'epoch': 0.05}
{'loss': 3.0154, 'grad_norm': 3.740619421005249, 'learning_rate': 3.1e-05, 'epoch': 0.05}
{'loss': 2.9498, 'grad_norm': 4.580867672720492, 'learning_rate': 2.1e-05, 'epoch': 0.06}
{'loss': 2.8376, 'grad_norm': 3.787401752471924, 'learning_rate': 1.1000000000000001e-05, 'epoch': 0.07}
{'loss': 2.9039, 'grad_norm': 4.149991035461426, 'learning_rate': 1.0000000000000002e-06, 'epoch': 0.08}
{'train_runtime': 23833.2677, 'train_samples_per_second': 0.134, 'train_steps_per_second': 0.004, 'train_loss': 3.834406976699829, 'epoch': 0.08}
100%|          | 100/100 [6:37:13<00:00, 238.33s/it]
==> Finish training and save to .\saved_models\epochs_200\Final_checkpoint
```

Figure 1 – Training example for Scenario 1

Due to computational limitations, as mentioned above, the runs to calculate the correctness of the codes predicted by the various models were stopped around problem number 449/628. All runs required a large number of hours, and in some cases the computer on which the code was run exhibited a “Blue Screen of Death” (a fatal system error that causes the computer to suddenly shut down and display a blue screen), like the following image.

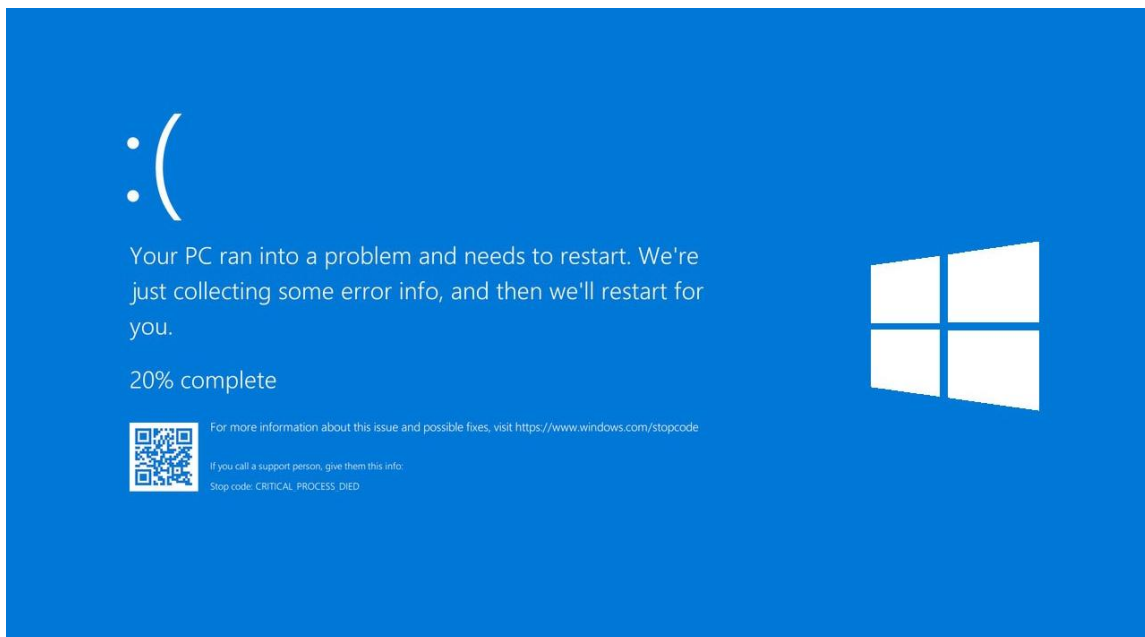


Figure 2 – Blue screen of death error in Windows

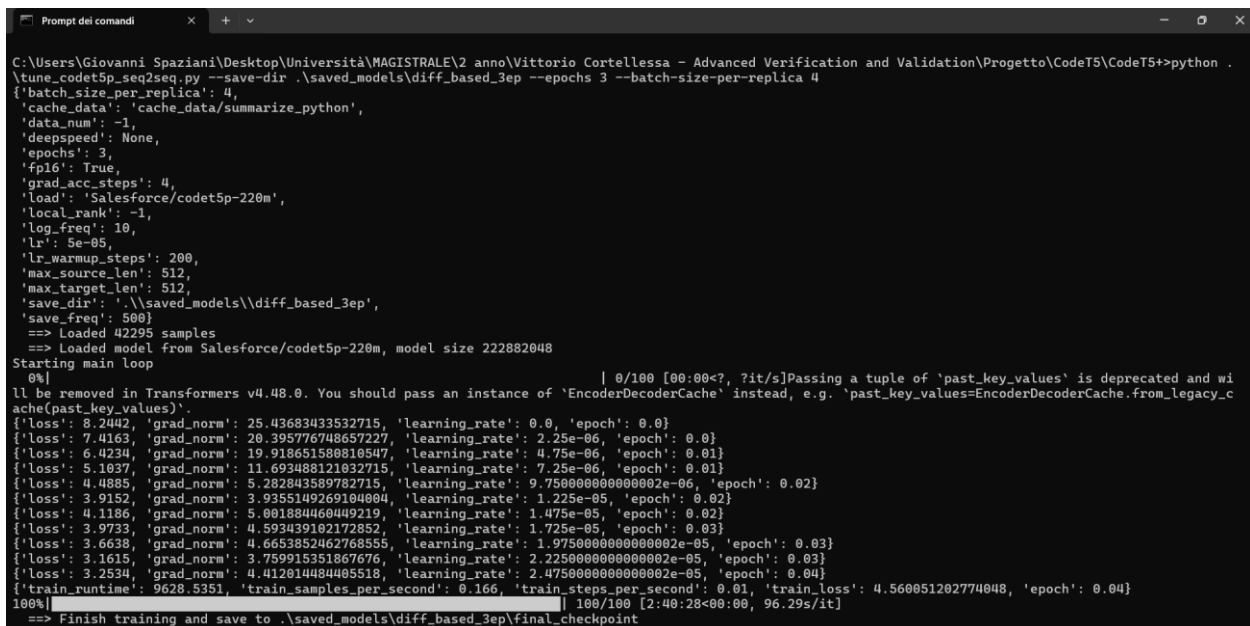
The metrics calculated on the three different models are:

Evaluation	Model 1	Model 2	Model 3
Training time	03:42:15	06:14:00	06:37:13
Testing time	03:11:00	03:55:41	04:15:22
Simulation time on gem5	11:25:17	13:23:54	15:56:49
Average speedup	1.0148335982057637	2.2732303648422025	2.294143576599569
Percentage correct	0.22%	7.35%	26.28%
Optimized percentage	0.00%	4.90%	10.24%
Total number of programs	450 (including header lines in csv files.)	450 (including header lines in csv files.)	450 (including header lines in csv files.)

Table 3 - Scenario 1

From the results in *Table 3*, it is shown that a clear performance progression occurs as the number of epochs increases. *Model 1* performs poorly, with a negligible percentage of correct predictions, and no actual optimizations. *Model 2* improves significantly in both correctness and average speedup, indicating that extended training may leads to better results (obviously, it doesn't only depend on the number of epochs, it also depends on other parameters). However, *Model 3* achieves the best overall performance: although its average speedup is like *Model 2*, it produces a much higher percentage of correct predictions and optimized programs. This suggests that the model not only generates more accurate predictions but also does so across a larger portion of programs within the test set, making the improvements more consistent and robust.

Moving on to the simulation results for *Scenario 2*, a problem immediately emerged regarding the training of the CodeT5+ model. The latter was initially trained with 3 epochs, and subsequently with 100 epochs. Below are screenshots of the terminal with the training parameter configurations:



```

C:\Users\Giovanni Spaziani\Desktop\Università\MAGISTRALE\2 anno\Vittorio Cortellessa - Advanced Verification and Validation\Progetto\CodeT5\CodeT5+>python .
\tune_codet5p_seq2seq.py --save-dir .\saved_models\diff_based_3ep --epochs 3 --batch-size-per-replica 4
{'batch_size_per_replica': 4,
 'cache_data': 'cache_data/summarize_python',
 'data_num': -1,
 'deepspeed': None,
 'epochs': 3,
 'fp16': True,
 'grad_acc_steps': 4,
 'load': 'Salesforce/codet5p-220m',
 'local_rank': -1,
 'log_freq': 10,
 'lr': 5e-05,
 'lr_warmup_steps': 200,
 'max_source_len': 512,
 'max_target_len': 512,
 'save_dir': '.\saved_models\diff_based_3ep',
 'save_freq': 500}
==> Loaded 42295 samples
==> Loaded model from Salesforce/codet5p-220m, model size 222882048
Starting main loop
0%
| 0/100 [00:00<?, ?it/s] Passing a tuple of 'past_key_values' is deprecated and wi
ache(past_key_values)'.
{'loss': 8.2402, 'grad_norm': 25.43683433532715, 'learning_rate': 0.0, 'epoch': 0.0}
{'loss': 7.4163, 'grad_norm': 20.305776748657227, 'learning_rate': 2.25e-06, 'epoch': 0.0}
{'loss': 6.4234, 'grad_norm': 19.918651580810547, 'learning_rate': 4.75e-06, 'epoch': 0.01}
{'loss': 5.1037, 'grad_norm': 11.693488121832715, 'learning_rate': 7.25e-06, 'epoch': 0.01}
{'loss': 4.4885, 'grad_norm': 5.282843589782715, 'learning_rate': 9.750000000000002e-06, 'epoch': 0.02}
{'loss': 3.9152, 'grad_norm': 3.9355149269104804, 'learning_rate': 1.225e-05, 'epoch': 0.02}
{'loss': 4.1186, 'grad_norm': 5.001884460449219, 'learning_rate': 1.475e-05, 'epoch': 0.02}
{'loss': 3.9733, 'grad_norm': 4.593439102172852, 'learning_rate': 1.725e-05, 'epoch': 0.03}
{'loss': 3.6638, 'grad_norm': 4.6653852462768555, 'learning_rate': 1.9750000000000002e-05, 'epoch': 0.03}
{'loss': 3.1615, 'grad_norm': 3.759915351867676, 'learning_rate': 2.2250000000000002e-05, 'epoch': 0.03}
{'loss': 3.2534, 'grad_norm': 4.412014484405518, 'learning_rate': 2.4750000000000002e-05, 'epoch': 0.04}
{'train_runtime': 9628.5351, 'train_samples_per_second': 0.166, 'train_steps_per_second': 0.01, 'train_loss': 4.560051202774048, 'epoch': 0.04}
100%
==> Finish training and save to .\saved_models\diff_based_3ep\final_checkpoint

```

Figure 3 - Training of the model with 3 epochs for Scenario 2


```

Prompt del comandi
C:\Users\Giovanni Spaziani\Desktop\Università\MAGISTRALE\2 anno\Vittorio Cortellesa - Advanced Verification and Validation\Progetto\CodeT5\CodeT5>python .
\tune_codet5p_seq2seq.py --save-dir .\saved_models\diff_based_100ep --epochs 100 --b
atch-size-per-replica 8
{'batch_size_per_replica': 8,
 'cache_data': 'cache_data/summarize_python',
 'data_num': -1,
 'deepspeed': None,
 'epochs': 100,
 'fp16': True,
 'grad_acc_steps': 4,
 'load': 'Salesforce/codet5p-220m',
 'local_rank': -1,
 'log_freq': 10,
 'lr': 5e-05,
 'lr_warmup_steps': 200,
 'max_source_len': 512,
 'max_target_len': 512,
 'save_dir': '.\saved_models\diff_based_100ep',
 'save_freq': 500}
==> Loaded 42295 samples
==> Loaded model from Salesforce/codet5p-220m, model size 222882048
Starting main loop
0%
is deprecated and will be removed in Transformers v4.48.0. You should pass an instance of 'EncoderDecoderCache' instead, e.g. 'past_key_values=EncoderDecode
rCache.from_legacy_cache(past_key_values)'
{'loss': 8.0653, 'grad_norm': 21.721532821655273, 'learning_rate': 0.0, 'epoch': 0.0}
{'loss': 7.3385, 'grad_norm': 21.0124568939209, 'learning_rate': 2.25e-06, 'epoch': 0.01}
{'loss': 6.5378, 'grad_norm': 25.52418327331543, 'learning_rate': 4.75e-06, 'epoch': 0.02}
{'loss': 5.1013, 'grad_norm': 13.828131675720215, 'learning_rate': 7.25e-06, 'epoch': 0.02}
{'loss': 4.5783, 'grad_norm': 5.590925693511963, 'learning_rate': 9.750000000000002e-06, 'epoch': 0.03}
{'loss': 4.0659, 'grad_norm': 4.517269611358643, 'learning_rate': 1.225e-05, 'epoch': 0.04}
{'loss': 3.8657, 'grad_norm': 4.557953357696533, 'learning_rate': 1.475e-05, 'epoch': 0.05}
{'loss': 3.7464, 'grad_norm': 3.886207103729248, 'learning_rate': 1.725e-05, 'epoch': 0.05}
{'loss': 3.5219, 'grad_norm': 4.485307693481445, 'learning_rate': 1.9750000000000002e-05, 'epoch': 0.06}
{'loss': 3.2425, 'grad_norm': 3.6275062561035156, 'learning_rate': 2.2250000000000002e-05, 'epoch': 0.07}
{'loss': 3.2145, 'grad_norm': 4.109954357147217, 'learning_rate': 2.4750000000000002e-05, 'epoch': 0.08}
{'train_runtime': 23984.2583, 'train_samples_per_second': 0.133, 'train_steps_per_second': 0.004, 'train_loss': 4.528546094894409, 'epoch': 0.08}
100%
==> Finish training and save to .\saved_models\diff_based_100ep\final_checkpoint

```

Figure 4 – Training of the model with 100 epochs for Scenario 2

The runs apparently succeeded, but the model failed to learn the desired output structure. The generated predictions do not contain the “+” and “-” symbols (to represent lines of code added or removed from the original code, respectively), but tend to produce portions of complete code, almost more closely resembling *Scenario 1* than *Scenario 2*. As a result, it was not possible to perform a comparison between prediction and correctness, nor to apply the reconstruction of the final code through diffs. As already mentioned, this anomaly occurred with various settings for the number of training epochs and tuning of the main parameters (from the previous screenshots, in fact, we can see that between the run with 3 epochs and the one with 100 epochs, there was a change in the passage of the “batch size per replica” parameter, where for the first model it was set to 4, while for the second it was set to 8).

A further test was performed for *Scenario 2* (as we can see in *Table 2*), i.e. the model was trained with 10 epochs, with additional parameters changed compared to the previous trainings (see image below):


```

C:\Users\Giovanni Spaziani\Desktop\Università\MAGISTRALE\2 anno\Vittorio Cortellesa - Advanced Verification and Validation\Progetto\CodeT5\CodeT5+>python .
\tune_codet5p_seq2seq.py --save-dir .\saved_models\diff_based_10eptest --epochs 10 --batch-size-per-replica 4 --lr 3e-4 --lr-warmup-steps 50
{'batch_size_per_replica': 4,
 'cache_data': 'cache_data/summarize_python',
 'data_num': -1,
 'deepspeed': None,
 'epochs': 10,
 'fp16': True,
 'grad_acc_steps': 4,
 'load': 'Salesforce/codet5p-220m',
 'local_rank': -1,
 'log_freq': 10,
 'lr': 0.0003,
 'lr_warmup_steps': 50,
 'max_source_len': 512,
 'max_target_len': 512,
 'save_dir': '.\saved_models\diff_based_10eptest',
 'save_freq': 500}
==> Loaded 42295 samples
==> Loaded model from Salesforce/codet5p-220m, model size 222882048
Starting main loop
0% | 0/100 [00:00<?, ?it/s]P
assing a tuple of 'past_key_values' is deprecated and will be removed in Transformers v4.48.0. You should pass an instance of 'EncoderDecoderCache' instead,
e.g. 'past_key_values=EncoderDecoderCache.from_legacy_cache(past_key_values)'.
{'loss': 8.2442, 'grad_norm': 25.43683433532715, 'learning_rate': 0.0, 'epoch': 0.0}
{'loss': 5.7817, 'grad_norm': 4.916317462921143, 'learning_rate': 5.399999999999999e-05, 'epoch': 0.01}
{'loss': 3.9131, 'grad_norm': 4.03676700592041, 'learning_rate': 0.00011399999999999999, 'epoch': 0.01}
{'loss': 3.2903, 'grad_norm': 4.626638412475586, 'learning_rate': 0.00017399999999999997, 'epoch': 0.01}
{'loss': 2.9357, 'grad_norm': 4.280560493469238, 'learning_rate': 0.000234, 'epoch': 0.02}
{'loss': 2.3179, 'grad_norm': 3.8805463314056396, 'learning_rate': 0.000294, 'epoch': 0.02}
{'loss': 1.9007, 'grad_norm': 4.569063663482666, 'learning_rate': 0.00024599999999999996, 'epoch': 0.02}
{'loss': 1.3693, 'grad_norm': 3.683814764022827, 'learning_rate': 0.000186, 'epoch': 0.03}
{'loss': 0.9587, 'grad_norm': 3.1067888736724854, 'learning_rate': 0.00012599999999999997, 'epoch': 0.03}
{'loss': 0.7128, 'grad_norm': 2.1649911403656006, 'learning_rate': 6.599999999999999e-05, 'epoch': 0.03}
{'loss': 0.641, 'grad_norm': 2.2310473918914795, 'learning_rate': 5.999999999999999e-06, 'epoch': 0.04}
{'train_runtime': 9375.9701, 'train_samples_per_second': 0.171, 'train_steps_per_second': 0.011, 'train_loss': 2.4067536926269533, 'epoch': 0.04}
100% | 100/100 [2:36:16<00:00, 93.76s/it]
==> Finish training and save to .\saved_models\diff_based_10eptest\final_checkpoint

```

Figure 5 – Training of the model with 10 epochs and different parameters for Scenario 2

As we can see in the image above, in addition to the different number of epochs, the “lr” (learning rate) and “lr warmup steps” parameters have been changed, going from:

- “5e-5” to “3e-4” for the first parameter.
- “200” to “50” for the second parameter.

However, the results obtained were identical to those observed with 3 and 100 epochs: the predictions still do not respect the diff-based format, completely missing the addition and removal symbols of lines of code, thus resulting in complete code fragments rather than “patches”.

A proof that proves the above is demonstrated in the next section.

After training the model on the diff-based Dataset, a testing process was carried out using the following Python script.

```

test_model.py > ...
1  from transformers import AutoTokenizer, AutoModelForSeq2SeqLM, pipeline
2  import json
3  from evaluate import load
4  from tqdm import tqdm
5
6  model_path = "saved_models/diff_based_100ep/final_checkpoint"
7
8  model = AutoModelForSeq2SeqLM.from_pretrained(model_path, trust_remote_code=True)
9  tokenizer = AutoTokenizer.from_pretrained("Salesforce/codet5p-220m")
10
11 # === Loading test set ===
12 def load_jsonl(path):
13     with open(path, 'r', encoding='utf-8') as f:
14         return [json.loads(line) for line in f]
15
16 test_data = load_jsonl("final_test_diff_based.jsonl")
17
18 # === Setup pipeline ===
19 pipe = pipeline("text2text-generation", model=model, tokenizer=tokenizer, truncation=True)
20
21 # === Predictions ===
22 results = []
23 for idx,item in tqdm(enumerate(test_data)):
24     input_code = item["src_code"]
25     reference_code = item["target"]
26
27     prediction = pipe(input_code, max_new_tokens=512)[0]["generated_text"]
28
29     results.append({
30         "input": input_code,
31         "reference": reference_code,
32         "prediction": prediction,
33         "problem": item["problem_id"]
34     })
35
36 # === Valutation: Exact Match ===
37 def exact_match(pred, ref):
38     return pred.strip() == ref.strip()
39
40 accuracy = sum(exact_match(r["prediction"], r["reference"]) for r in results) / len(results)
41 print(f"\nExact Match Accuracy: {accuracy:.2%}")
42
43 # === Save the results ===
44 with open("comp_code_opt_100ep_diff_based.json", "w", encoding="utf-8") as f:
45     json.dump(results, f, indent=2, ensure_ascii=False)

```

Figure 6 – Testing

This script loads the saved model (in this case, “diff_based_100ep/final checkpoint”) and applies it to each example in the test set (final_test_diff_based.jsonl). Specifically:

- The `src_code` field is provided as input to the model pipeline.
- The model, using *HuggingFace’s text2text-generation*, generates a prediction in the form of a string.
- For each example, both the generated prediction and the target field, which contains the correct diff-based part, are saved.

The result after running this script is a .json file (in this example the resulting .json file is “comp_code_opt_100ep_diff_based.json”) and can be viewed via the following image.

Figure 7 – Test phase result with predictions

As we can see, this file contains:

- **input:** the original (unoptimized) source code, equivalent to the src_code field in the test set file “final_test_diff_based.json”.
- **reference:** the expected correct code portion, in diff-based format, with lines marked with “+” and “-”. This field corresponds to the target of the test set file.
- **prediction:** the output generated by the model, which should have represented a diff-based portion (but, as highlighted by the red rectangles in the image, does not respect this structure due to a structural problem of the used model).
- **problem:** ID of the problem associated with each example.

This file, therefore, represents the result of the testing phase in Scenario 2, and was used to verify whether the model was able to generate the expected outputs.

This experiment confirms that the problem is not due to underfitting related to the number of epochs or other parameters, but seems to be structural, due to a difficulty of the CodeT5+ model in learning the specific semantics of the diff format.

Anyway, for *Scenario 2*, the times indicated for training and testing the various models are reported in the following table, excluding the calculation of the metrics, as the model does not provide an output consistent with the expected format.

Evaluation	Model 1	Model 2	Model 3 (10 epochs)
Training time	03:05:19	06:39:44	02:36:16
Testing time	03:54:09	03:15:33	02:55:28

Table 4 – Scenario 2

Obviously, an increase in epochs corresponds to a significant increase in both time and performance, encountering the computational limitations of the machine.

In conclusion, the behavior observed in *Scenario 2* suggests that the CodeT5+ model struggles to correctly internalize the structured semantics of diff-based, giving “priority” to sequential outputs that are more attributable to a complete code generation, rather than transformational ones. The systematic absence of the “+” and “-” symbols therefore compromises the entire post-processing pipeline, making the reconstruction of the optimized code via diff parsing inapplicable.