Projet d'Apprentissage

Sujet : complétion de code elm

Composition du groupe
Ninon AUTEFAGE-PINIES
Théo BESSEL
~~Théophane CHOLLET~~ ?
Nell TRUONG

Le projet est disponible sur Github.

Table des matières

Objectif du projet

L'objectif de ce projet est de créer un un générateur de code capable de faire de la complétion de code Elm.

> Elm est un langage fonctionnel pour la création d'interface graphique web, compilé en JavaScript.

Récupération de données

Pour récupérer les données, nous avons récupéré des fragments de code directement dans Github. Nous avons créé un script python pour vérifier que les fichiers récupérés de Github sont bien des fichiers elm correct pour éviter d'entraîner le modèle sont des données erronées. Les fichiers .elm sont stockés dans un dossier. Au total, nous avons récupérés *6 927 fichiers*, pour un total de *1207791 lignes de codes*.

Description fonctionnelle du script de récupération des données

Le script se connecte à l'API REST de GitHub, il récupère les résultats de recherche avec les paramètres `q = code` et `langage = Elm`.

Il recherche en bref les fichiers de code ayant pour extension un .elm. À partir de la réponse de l'API on peut récupérer les URLs de téléchargement des fichiers et télécharger ces derniers. Lors de l'étape de

téléchargement nous avons rajouté une petite étape qui vérifie que le contenu du fichier correspond bien à la syntaxe du langage Elm.

Cela permet d'éliminer certains faux positifs au test consistant à regarder l'extension du fichier. Parmis ceux que nous avons rencontrés lors de l'utilisation du script il y avait : Des fichiers mails .eml dont l'extension a été mal orthographiée Des fichiers de configuration (remplis de valeurs numériques) pour un projet donc l'abréviation donnait sûrement "eml" Éliminer ces fichiers indésirables permet d'obtenir une base de données de meilleure qualité et permet d'éviter de générer du code utilisant des tokens incorrects par la suite.

Cette étape d'élimination utilise un petit parseur minimal n'étant pas aussi strict que la syntaxe Elm (ce dernier accepte des codes non valides en Elm mais qui sont tout de même relativement proches de ce langage). Nous aurions pu écrire un parseur complet en utilisant des outils comme ANTLR ou encore directement en OCaml mais cela n'était pas trop dans le cadre du projet et aurait pris beaucoup de temps pour une utilité limitée car le parseur minimal a déjà permis d'éliminer tous les fichiers indésirables utilisant l'extension .eml pour des fichiers n'ayant vraiment rien à voir avec le langage Elm.

L'absence d'un parseur plus évolué a été un peu plus problématique lors de l'évaluation des performances de notre modèle mais ce problème est détaillé dans la section qui lui est dédiée.

Afin de pouvoir récupérer plus de fichiers Elm que ce que ne limite l'API de Github (1000 résultats par requête), nous avons opté pour une variante de la méthode présentée ci-dessus. Cette approche consiste à rechercher tous les dépôts gits contenant des fichiers Elm (donc 1000 dépôts comme résultat de la requête) et à récupérer les fichiers Elm contenus dans ces dépôt. Cela permet d'augmenter drastiquement le nombre de fichiers récupérés avec une moyenne de 10 fichiers Elm par dépôt. Nous avons ainsi pu récolter environ 2 000 000 lignes (code + commentaires) en utilisant environ 40 dépôts (le parcours de ces derniers prenant un temps considérable même en ignorant le parcours de certains dossiers dans les dépôts comme `assets/`, `static/`, `config/`, ... dans lesquels il est peu probable de trouver du code Elm).

Important: dans un soucis de respect de la propriété intellectuelle des scripts récupérés pour la constitution de notre dataset, nous avons pris la décision de ne pas publier notre dataset sur Github/Google Drive/etc.. c'est pourquoi le fine-tuning a été réalisé localement sur une de nos machines et non sur un outil tel que Google Colab. Ce choix est important car il impacte certains choix techniques lors du fine-tuning.

Création du dataset

Une fois notre ensemble de scripts récupéré et filtré, nous avons pu créer un dataset pour le fine-tuning de notre modèle. Pour se faire, un script python se charge de séparer les scripts en "*unités de code*" qui sont les fonctions, types et opérateurs définis dans les scripts.

Afin de rendre l'entrainement meilleur pour la génération de code, nous excluerons les commentaires et tenterons d'exclure les imports et modules. Une fois les *unités de code* récupérées, elles sont compilées en un seul dataset. Le dataset comprends au total 91528 *unités de code*.

Le dataset prend la forme d'un fichier JsonLines (.jsonl), un format de fichier adapté pour le traitement de larges quantités de données et facile de lecture, ce qui est particulièrement adapté pour notre problème.

Choix du modèle

Si notre choix intial de modèle s'était porté sur llama2 pour sa compatibilité avec la librairie Unsloth (pour un entraînement plus rapide), nous avons au final choisi Starcoder2-3b, un modèle ayant un tokenizer entraîné

spécifiquement sur du code et fine-tunable. Nous avons choisi la version 3b car il s'agit de sa version la plus légère (3 milliards de paramètres), et nous avions besoin que le modèle soit suffisament léger pour être fine-tuné sur une de nos machines.

Bonus: voir la section bonus pour l'architecture du réseau.

Fine-tuning d'un modèle

D'un point de vue technique, nous avons utilisé les outils de HuggingFace: datasets, transformers (Pour la récupération du modèle et le tokenizer), peft (Parameter-Efficien Fine-Tuning, pour le fine Tuning via LoRA) ainsi que torch et bitsandbytes (wrapper au dessus de CUDA) pour faire tourner l'entraînement sur GPU et profiter de la précision à 4 bits pour réduire l'utilisation mémoire lors du fine-tuning.

Une fois notre dataset chargé, nous avons pu procéder au fine-tuning du modèle que nous avons choisi. Nous avons réalisé le fine-tuning deux fois, une fois en limitant le nombre de steps à 3000 et une fois sans le limiter (68646 steps ! Cet entraînement a duré près de 37 heures non-stop).

Nous avions tenté d'utiliser Unsloth pour optimiser le fine-tuning mais nous n'avions pas réussi pour llama. Au final cette piste fût abandonnée lorsque nous sommes passés sur starcoder2, faute de compatibilité.

Processus du fine tuning

Cette partie explique pas-à-pas le processus de fine-tuning d'un model avec notre dataset. Ce processus est pratiquement identique à celui du TP sur la génération de texte.

Import du modèle et du tokenizer

```
from transformers import AutoTokenizer, AutoModelForCausalLM, Trainer,
TrainingArguments, BitsAndBytesConfig
from datasets import load_dataset
import torch

model_name = "bigcode/starcoder2-3b"
device = "cuda" # for GPU usage or "cpu" for CPU usage

tokenizer = AutoTokenizer.from_pretrained(model_name)
tokenizer.pad_token = tokenizer.eos_token

bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_compute_dtype=torch.float16,
    bnb_4bit_use_double_quant=True,
    bnb_4bit_quant_type="nf4"
)

model = AutoModelForCausalLM.from_pretrained(
    model_name,
    quantization_config=bnb_config,
    trust_remote_code=True,).to(device)

model.config.pad_token_id = tokenizer.pad_token_id # for LoRA
```

Dans cette partie, on charge le modèle que l'on cherche à fine-tuner. De plus, on ajoutera par la même occasion une configuration de quantisation, une méthode utilisée pour la réduction des coûts de calcul et des coûts en mémoire du fine-tuning, par exemple en utilisant une précision à 4 bits seulement.

Nous configurons par la même occasion l'utilisation de CUDA avec `.to("cuda")` afin de réaliser le processus de fine-tuning sur GPU, ce qui devrait théoriquement nous offrir un gain de temps considérable.

Les étapes suivantes sont très similaires au TP et consistent au chargement du dataset, à la configuration des paramètres du fine-tuning avec LoRA et des paramètres d'entraînement du modèle.

Chargement du dataset

```
# Load your dataset, assuming jsonl format with "text" key
dataset = load_dataset("json", data_files={"train": "elm_code_units.jsonl"})

# Tokenize function for causal LM
def tokenize_function(examples):
    encoding = tokenizer(
        examples["text"],
        truncation=True,
        padding="max_length",
        max_length=512,
    )
    encoding["labels"] = encoding["input_ids"].copy()
    return encoding

tokenized_dataset = dataset.map(tokenize_function, batched=True, remove_columns=
["text"])
```

Configuration de LoRA pour le fine-tuning

```
from peft import LoraConfig, get_peft_model

per_device_train_batch_size = 1
gradient_accumulation_steps = 8  # Increased to keep the same effective batch size

peft_config = LoraConfig(
    r=8,
    lora_alpha=16,
    target_modules=["q_proj", "v_proj"],  # Adjust for StarCoder2
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM"
)

model = get_peft_model(model, peft_config)
```

Paramètres d'entraînement

```
from transformers import DataCollatorForLanguageModeling

training_args = TrainingArguments(
    output_dir="./starcoder_elm_finetuned",
    per_device_train_batch_size=4,
    num_train_epochs=3,
    save_steps=500,
    save_total_limit=2,
    logging_steps=100,
    learning_rate=3e-4,
    fp16=True,
    report_to="none"
)

data_collator = DataCollatorForLanguageModeling(
    tokenizer=tokenizer,
    mlm=False,  # because you're doing causal LM
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_dataset["train"],
    data_collator=data_collator,
)
```

Les paraètres ont été choisis pour la machine sur laquelle le fine-tuning a été réalisé. C'est la raison pour laquelle le nombre de batch par appareil est configuré à 1, afin de ne pas surcharger le GPU sur lequel serait entraîné le modèle.

Entraînement

```
import os
import torch
os.environ["PYTORCH_CUDA_ALLOC_CONF"] = "expandable_segments:True"

# Empty CUDA's cache to prevent OOM errors (OutOfMemory)
torch.cuda.empty_cache()

trainer.train()
model.save_pretrained("./starcoder_elm_finetuned")
tokenizer.save_pretrained("./starcoder_elm_finetuned")
```

Dans cette dernière étape, on réalise l'étape d'entraînement (fine-tuning) et on enregistre localement le modèle ré-entraîné afin de pouvoir le charger à l'avenir.

Avant d'entraîner notre modèle, nous avons à nouveau dû prendre en considération la machine sur laquelle nous allions entraîner notre modèle et ses limitations. Afin d'éviter que le programme ne se trouve à court de mémoire en VRAM, nous autorisons les segments alloués à être de tailles variables et procédons à un

nettoyage du cache de CUDA pour éviter tout résidu de précédentes opérations qui pourraient compromettre l'execution de l'entraînement.

Vérification des résultats

Le fichier `verification.ipynb` sert à la vérification du (des) modèle(s) que nous avons fine-tunés.

Les dépendances nécessaires à son éxécution sont les mêmes que pour le fine-tuning.

Le script prend en entrée un fichier de test, ici nommé `test.json` dans lequel seront contenus tous nos tests. Un test prend la forme d'un début de code à compléter.

Voici un exemple de fichier de test:

```
[
  {"prompt" : "main ="},
  {"prompt" : "main =\n picture"},
  {"prompt" : "update : Msg -> Model -> (Model, Cmd Msg)\nupdate msg model ="},
  {"prompt" : "fact : Int -> Int\nfact="},
  {"prompt" : "simpleSum : (Int, Int) -> Int\nsimpleSum ="},
  {"prompt" : "listLength ="}
]
```

Le script génère un nombre `gen_per_prompt` de générations par prompt dans le fichier de test en utilisant le modèle choisi (paramétré par `model_path`). Nous vérifions ensuite que chaque code généré par notre modèle est syntaxiquement correct en utilisant le même parseur que nous avions utilisé pour la constitution du dataset.

Le script produit ensuite un fichier contenant toutes les générations de code (paramétré par `generation_output`), un fichier rapport sur les générations et le pourcentage de réussite est produit (paramétré par `summary_output`).

Résultats

Quelques exemples

Exemple 1

> Prompt :
>
> ```
> "view : Model -> Html Msg
>  view model = "
> ```
>
> Output :
>
> ```
> view : Model -> Html Msg
> view model =
>    let
> ```

```
        input =
            inputModel model
                |> Html.form []
                |> Html.map Input
    in
    div []
        [ input
       , div [] [ button [ onClick Clear ] [
```

Exemple 2

Prompt :

```
"simpleSum : (Int, Int) -> Int
 simpleSum ="
```

Output :

```
simpleSum : (Int, Int) -> Int
simpleSum =
  (++) << first << second << first << second << first << second << first <<
second << first << second << first << second << first << second << first <<
second <<
```

Exemple 3

Prompt :

```
"viewCard : Card -> String
 viewCard card ="
```

Output :

```
viewCard : Card -> String
viewCard card =
    """{
        "card": {
            "name": """ ++ String.quote card.name ++ """,
            "value": """ ++ String.quote card.value ++ """,
```

Exemple 4

Prompt :

```
"module BlorkemonCards exposing
   ( Card
   , compareShinyPower
   , expectedWinner
   , isMorePowerful
   , maxPower
   , sortByCoolness
   , sortByMonsterName
   )"
```

Output :

```
module BlorkemonCards exposing
    ( Card
    , compareShinyPower
    , expectedWinner
    , isMorePowerful
    , maxPower
    , sortByCoolness
    , sortByMonsterName
    )

import Array exposing (Array)
```

Annalyse quantitative des résultats

Pour analyser le code généré par notre modèle, nous avons choisi d'utiliser le parser créé pour la vérification des fichiers lors de la récupération des données.

Nous avons générés du code à partir de 21 prompts *différents*, puis nous avons vérifié ces codes à l'aide du parser. Voici les résultats obtenus :

Prompt N°	Successes	Fails	Success rate
0	0	3	0.0
1	1	2	33.333333333333336
2	0	3	0.0
3	0	3	0.0
4	0	3	0.0
5	0	3	0.0
6	0	3	0.0
7	0	3	0.0

Prompt N°	Successes	Fails	Success rate
8	2	1	66.66666666666667
9	0	3	0.0
10	0	3	0.0
11	0	3	0.0
12	0	3	0.0
13	0	3	0.0
14	1	2	33.333333333333336
15	1	2	33.333333333333336
16	0	3	0.0
17	0	3	0.0
18	1	2	33.333333333333336
19	1	2	33.333333333333336
20	3	0	100.0

Number of tries: 63 Number of successes: 10 Number of fails: 53 Succes rate: 15.873015873015873%

On remarque que le code généré n'est que rarement syntaxiquement correct. Cela est en partie dû au fait que le code généré s'interrompt souvent au milieu d'une expression et ne respecte donc pas l'arbre syntaxique :
Exemple 5

> Prompt :
>
> ```
> "listLength ="
> ```
>
> Output :
>
> ```
> listLength = List.length list
> |> Maybe.withDefault 0
> |> Int.toFloat listLengthFloat
> listLengthFloat : Float -> Int -> List a -> Int
> listLengthFloat listLength _ list =
>     if listLength ==
> ```

Exemple 6

> Prompt :

```
"view : Model -> Html Msg
 view model ="
```

Output :

```
view : Model -> Html Msg
view model =
  let
    (title, description) =
      if model.page.isSearch then
        ("Search", "Search for your favorite food, drinks, or restaurant.")
      else
        ("Recipe
```

Ces deux exemples sont syntaxiquement incorrects car le code s'achève au milieu d'une expression, cependant, le début de ce qui a été généré respecte la syntaxe.

Parois, le code généré peut être syntaxiquement correct mais être absurde, comme dans l'exemple ci-dessous : Exemple 8 :

Prompt :

```
"main ="
```

Output :

```
main =
animation view model modelMsg modelMsg modelMsg modelMsg modelMsg modelMsg
modelMsg modelMsg modelMsg modelMsg modelMsg modelMsg modelMsg modelMsg
modelMsg modelMsg modelMsg modelMsg modelMsg modelMsg modelMsg modelMsg
```

Annalyse qualitative des résultats

Pour ce qui est de la qualité du code généré, celui-ci ressemble globalement à du code elm, mais n'est que rarement correct que ce soit syntaxiquement ou sémantiquement.

On remarque néanmoins que le code généré est meilleur lorsque le prompt est un nom de fonction *connue*. Exemple 9 :

Prompt :

```
"fact : Int -> Int
 fact="
```

Output :

```
fact : Int -> Int
fact=
  Native.Basics.fact

fib : Int -> Int
fib =
  Native.Basics.fib

gcd : Int -> Int -> Int
gcd =
  Native.Basics.gcd
```

Exemple 10

Prompt :

```
"simpleSum : (Int, Int) -> Int
 simpleSum ="
```

Output :

```
simpleSum : (Int, Int) -> Int
simpleSum =
  +)

sum : (Int, Int) -> Int
sum (a,b) =
  a + b

sum3 : (Int, Int,
```

Exemple 11

Prompt :

```
"listLength ="
```

Output :

```
listLength =
    List.length << List.filter identity list |> List.length


listLength2 : List a -> List a -> Int
listLength2 xs ys =
    List.foldl (\x y -> x + y)
```

Dans les exemples ci-dessus, le code n'est pas syntaxiquement correct, il échoue donc lors du test par le parse, pourtant ces exemples sont plus intéressants que certains qui ont réussi le test du parsing.

Influence des paramètres de génération Nous avons aussi de jouer sur certains paramètres de générations tels que la "température" (allant de 0 à 2.0, c'est la créativité de la réponse, capacité à produire de l'aléatoire) et l'échantillonnage par noyau (allant jusqu'à, ce paramètre joue sur le choix parmis les tokens les plus probables, le pplus bas le plus déterministe).

Ces expériences n'ont néanmoins rien révélé de particulièrement impactant sur la quantité de code valide généré.

Comparaison de la quantité d'entraînement

Si nos résultats numériques en terme de quantité de code valide n'est pas tout à fait exacte du fait que le parser n'est pas complètement correct, causant la non détection de certains codes corrects, nous avons noté que les statistiques restent gloablement constamment entre 10 et 15 pourcents indépendamment de l'utilisation du modèle fine-tuné sur 68000+ étapes et sur 3000. Ainsi, nous avons remarqué qu'un très long entraînement n'est pas nécessaire pour avoir un générateur de code théoriquement fonctionnel.

Conclusion

En conclusion, nous avons réussi à fine-tuner un modèle pour la génération de code Elm, bien que celui-ci ne soit pas parfait. Le code ressemble généraleemnt à ce qu'on attendrait d'une fonction écrite en elm, au détail près que le modèle a tendance à halluciner plus il a à écrire.

Difficultés rencontrées

Nous avons rencontré des difficultés avec le parser, qui n'acceptait finalement pas certains codes pourtant valides. En effet ce dernier utilisait uniquement des expressions régulières pour vérifier si le code répondait à la syntaxe du Elm ce qui ne falicitait pas une description rigoureuse et complète de la syntaxe de ce langage. Un outil comme ANTLR ou OCamlLex aurait été plus adapté mais aurais demandé de prendre un peu de temps pour spécifier la syntaxe de Elm, sur laquelle nous n'avons pas vraiment trouvé de spécification, le seul document officiel étant : https://elm-lang.org/docs/syntax

Nous avons eu quelques difficultés quant à la compatibilité de certaines dépendances du projet sur certains ordinateurs, en particulier dans le cas de CUDA pour l'utilisation GPU, qui n'est pas compatible avec toutes les cartes et puces graphiques ainsi que BitsAndBytes, que nous avons utilisé pour "alléger" l'entraînement et qui s'avère être une wrapper autour de CUDA.

Une autre difficulté a été notre première tentative de fine-tuner un modèle llama avec la librairie Unsloth (supposer alléger le fine-tuning) qui s'était soldée d'un échec.

Pertinence de l'évaluation des résultats Comme on a pu le voir lors de l'évaluation de nos résultats, vérifier si le code généré est syntaxiquement correct ne suffit pas pour avoir une évaluation pertinente de la qualité de notre modèle. En effet, un code syntaxiquement correct peut n'avoir aucun intêret car ne correspondant à rien (*Exemple 4*) tandis que un code non abouti peut être parfois très pertinent sans pour autant respecter la syntaxe (*Exemple 3*).

Pistes d'améliorations Pour améliorer les performance de notre modèle, il pourrait être judicieux de s'intéresser plus au tokenizer et de ne s'intéresser qu'aux tokens les plus présents en elm, à l'image de ce qui avait été fait lors du TP sur la génération de Tweets.

Une piste d'amélioration évidente est notre parser, qui n'était finalement pas complètement fonctionnel. Celui-ci ne laissait pas passer certains codes fonctionnels, dégradant la qualité de notre vérification.

Bonus

Statistiques

- Nombre de BSOD (Blue Screen Of Death - Windows) rencontrés lors du projet : 4
- Nombre d'heures de fine-tuning : 37 + 2,5 = 39,5 heures !
- Nombre de nuits que Nell a passé sur son canapé pour laisser le modèle s'entraîner : 2
- Nombre de timeouts de la part de l'API Github lors du scrapping : trop
- Nombre d'heures de travail de Théophane : 20min

Mais il est fou Exemple de résultat amusant obtenu avec un abaissement des paramètres de température et d'échantillonage par noyau:

```
--- Prompt 21 ---
listLength =

--- Génération 1 ---
listLength = 256 * 1024 * 1024 / 8 / 8 / 8 * 8
listLength = 1024 * 1024 * 10
```

Architecture de Starcoder

Une fois notre réseau entraîné et sauvegardé, il nous a été possible de tracer une représentation de la structure du réseau: