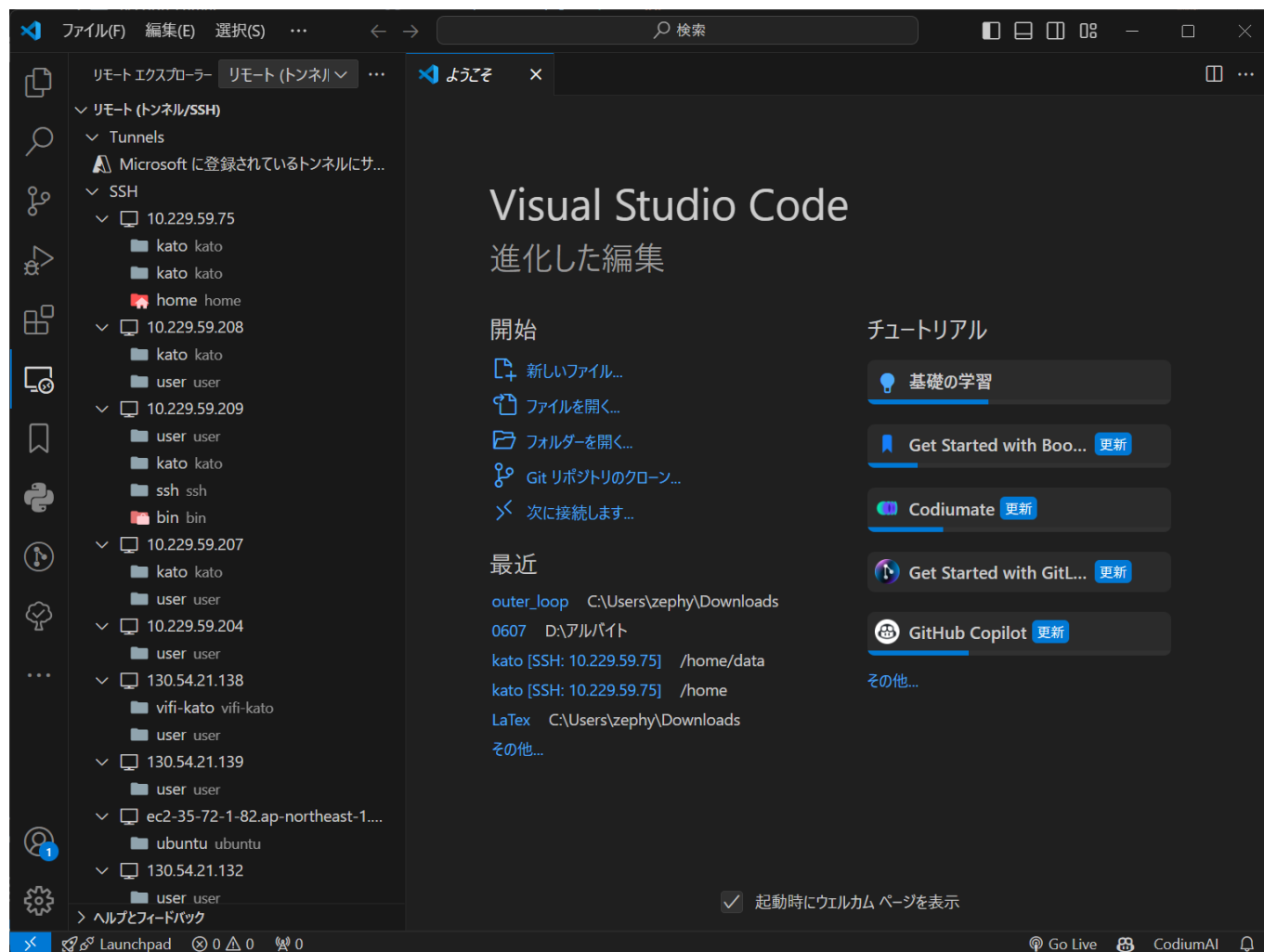


# Llama2 Fine-tuning

## 1. サーバーへの接続

研究室のWifi(Buffalo-A-88E0など)やVPNに接続しておいてください。

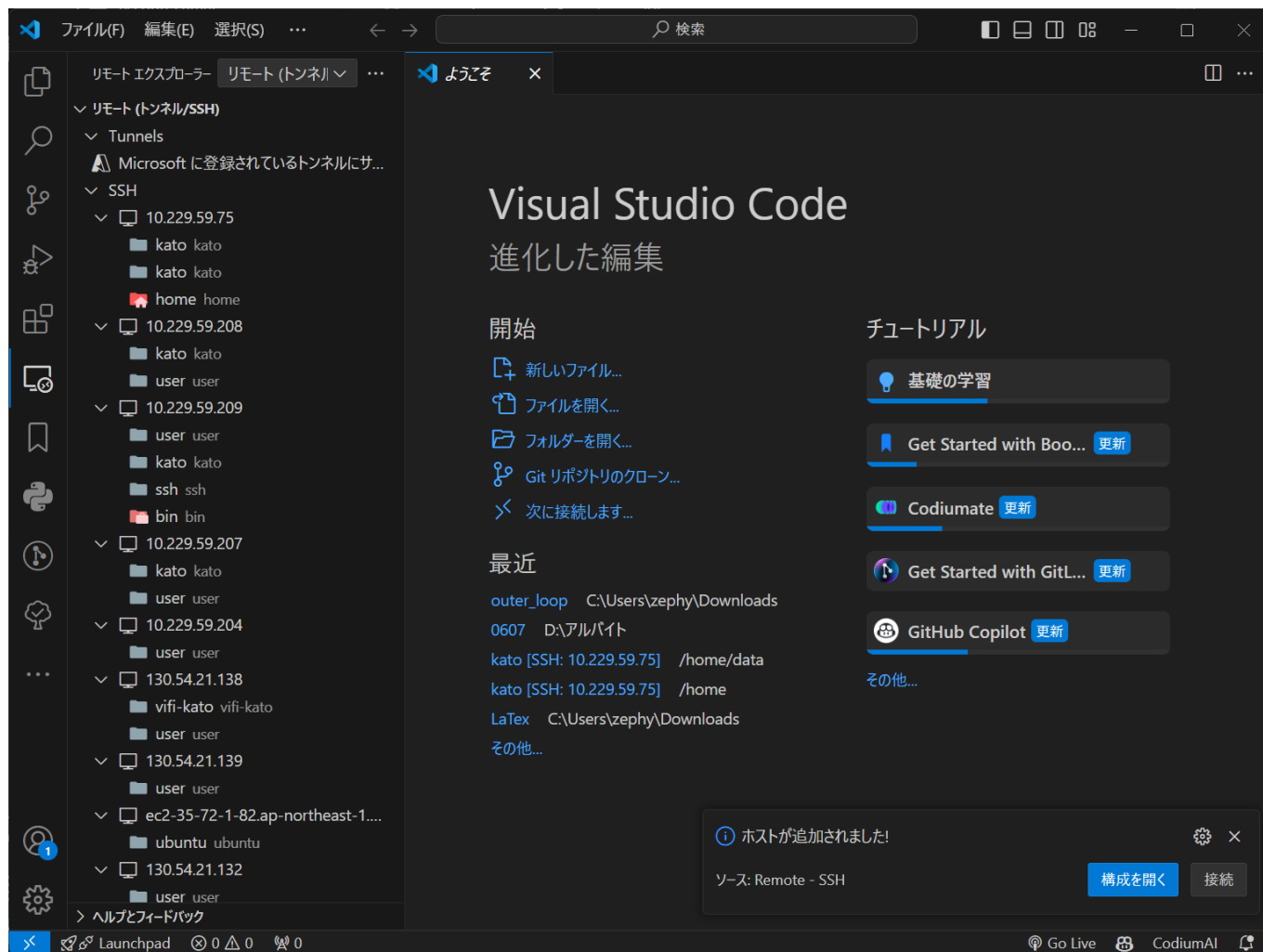
VSCodeでサーバーに接続します。 Remote Development拡張機能をインストールし、VSCodeのリモートエクスプローラーからssh接続を追加します。(画像はVSCodeでリモートエクスプローラータブを選択したときのものです。)



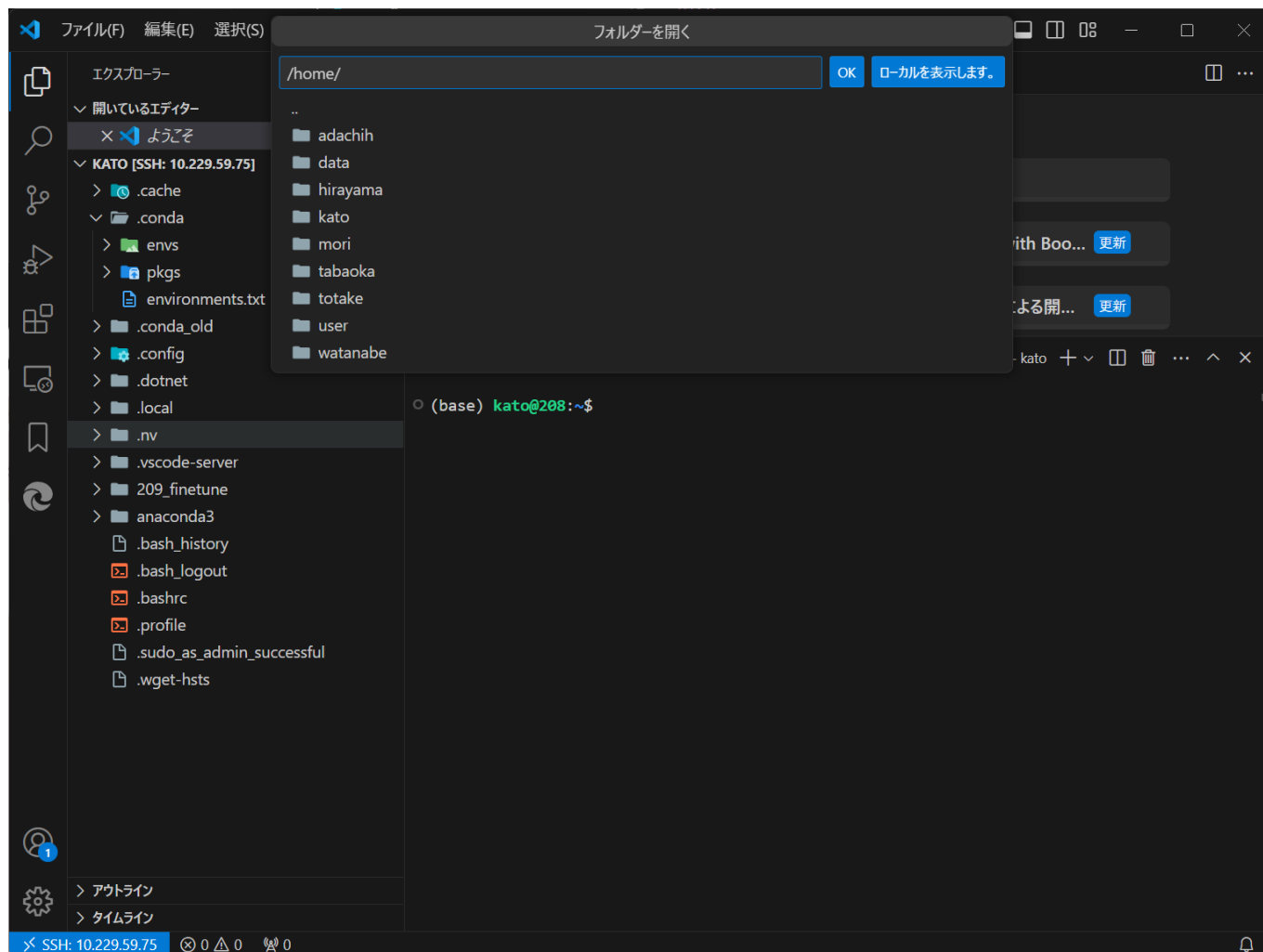
ここでSSHの横の+をクリックし、ssh接続用のコマンドを入れます。

```
ssh [自分のユーザー名]@10.229.59.75
```

「ホストが追加されました!」と表示されるので「接続」をクリックします。



VSCoideからの接続を行うと接続後にVSCoideで開くフォルダを選択できます。ここではいったん/home/[自分のユーザー名]を選択してください。



## 2. サーバーの操作

(Linuxコマンドに詳しい方飛ばしても良い)

ログインすると、ターミナルでは`home/data/[ユーザー名]`にいます。

- `pwd`で現在のディレクトリを確認できます。例えばログインしたときのターミナルで`pwd`を実行すれば

```
[ユーザー名]@208:/home/data/kato$ pwd
/home/data/kato
```

となります。

- `cd [ディレクトリ名]`でディレクトリを移動できます。以下は私の使っているサーバーでの例です。

```
[ユーザー名]@208:/home/data/kato$ cd /home/data/kato/hf_llama2
[ユーザー名]@208:/home/data/kato/hf_llama2$ pwd
/home/data/kato/hf_llama2
```

- `ls`でディレクトリ内のファイルを確認できます。以下は私の使っているサーバーでの例です。ディレクトリ内にあるファイルが表示されています。

```
[ユーザー名]@208:/home/data/kato/hf_llama2$ ls
52K0original_5K_70b_japanese.csv  52K0original_5K_70b_temp.csv
52KProofReading_5K_70b_japanese.csv  52KProofReading_5K_70b_temp.csv
70b_japanese_comparetest.py  llama2_test.py
```

- `cp`でファイルやディレクトリをコピーできます。`cp コピー元 コピー先`のように使います。コピー元・コピー先はディレクトリ名やファイル名です。コピー元にディレクトリを指定する場合は、`-r`オプションをつけます。

### 3.Anacondaのインストール

`/home/data`に`Anaconda3-2021.11-Linux-x86_64.sh`(Anacondaのインストーラ)がおいてあります(バージョンが適度に変更しても良い)。

```
[ユーザー名]@208:/home/data/[ユーザー名]$ cd ..
[ユーザー名]@208:/home/data$ ls
Anaconda3-2021.11-Linux-x86_64.sh  その他のファイルやディレクトリ...
```

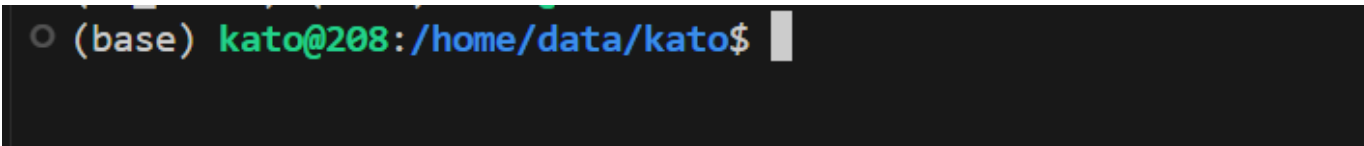
`cd ..`で一つ上のディレクトリに移動できるので、ログインしたときのディレクトリから`cd ..`を実行すれば、`/home/data`にいることになります。`ls`でAnacondaのインストーラがあることを確認したら、これを自分の使うディレクトリにコピーします。

```
[ユーザー名]@208:/home/data$ cp Anaconda3-2021.11-Linux-x86_64.sh
/home/data/[ユーザー名]
```

コピーできたら、以下のコマンドでAnacondaをインストールします。

```
cd /home/data/[ユーザー名]
bash Anaconda3-2021.11-Linux-x86_64.sh
```

Anacondaのインストール中の質問はyes(Enter)で進んでください。インストール場所はデフォルトのままで大丈夫です。最後に`conda init`を実行するか聞かれると思いますが、それもyesを選択してください。インストール後に`source ~/.bashrc`を実行すると、以下の画像のようにターミナルに`(base)`と表示されると思います。



```
(base) kato@208:/home/data/kato$
```

```
conda -V
```

でAnacondaのバージョンが表示されることを確認してください。

```
(base) kato@208:/home/data/kato$ conda -V
conda 4.10.3
```

## 4. Anacondaの環境構築

Anacondaを使って、Pythonの仮想環境(今回はllama2のfine-tuning用の環境)を作ります。

```
conda create -n [環境名] python=[Pythonのバージョン]
```

で新しい環境を作成できます。ここでは

```
conda create -n llama_finetune python=3.10
```

とします。途中の質問はyes(Enter)で進んでください。環境が作成出来たら、

```
conda info --envs
```

で今ある環境を表示できます。`llama_finetune`があることを確認してください。

```
conda activate llama_finetune
```

で環境を有効にします。

`conda deactivate`で仮想環境から出て、(base)環境に戻ることができます。

## 5. fine-tuningの準備

### 5-1. リポジトリのクローン

以下、作業はすべて`llama_finetune`環境で行います。

まず`finetune`ディレクトリを作成しておきます。

```
mkdir finetune
```

ファイルやディレクトリの作成はVSCodeのエクスプローラーからも行えます。エクスプローラーの上のほうにカーソルを合わせるとフォルダのアイコンが表示されるので、クリックして新しいフォルダを作成できます。



開いているエディター

- train\_2.py rler/Human\_beginner/10/PPO/Cl...
- eval.py rler/Human\_beginner/DQN/CliffWal...
- finetune.py 2025\_demo/finetune/... 2, M
- finetune\_5000.py 2025\_demo/fine... 2, U
- × check\_result.py 2025\_demo/finetu... 8, U
- select.py 2025\_demo/datasets

KATO [SSH: 10.229.59.75]

- .cache
- .conda
- .config
- .dbus
- .dotnet
- .gconf
- .gnupg
- .local
- .mozc
- .nv
- .presage
- .ssh
- .vscode
- .vscode-server
- 209\_finetune
- 2025\_demo
  - datasets
    - japanese\_alpaca\_data\_100.json
    - japanese\_alpaca\_data\_1000.json
    - japanese\_alpaca\_data\_5000.json
    - japanese\_alpaca\_data.json
    - select.py
  - finetune / alpaca-lora
    - .github
    - templates
    - utils
    - .dockerignore
    - .gitignore

```
2025_demo > finetune > alp
9  from transform
10 from transform
11
12 from utils.cal
13 from utils.pro
14
15 if torch.cuda.
16     device = "
17 else:
18     device = "
19
20 try:
21     if torch.b
22         device
23 except: # noq
24     pass
25
26
27 def main(
28     load_8bit:
29     # base_mod
30     base_model
31     # lora_we
32     # lora_we
33     # lora_we
34     prompt_tem
35     # server_n
36     # share_gr
37     instructio
38     input: str
39 ):
40     base_model
41     assert (
42         base_m
43     ), "Please
44
45     prompter =
46     # tokenize
47     tokenizer
48     #* lora-we
```

<div> <div> <div></div> <div>alpaca_data_cleaned_archive.json</div> </div> <div> <div></div> <div>alpaca_data_gpt4.json</div> </div> <div> <div></div> <div>alpaca_data.json</div> </div> <div> <div></div> <div>check_result.py</div> </div> <div> <div></div> <div>DATA_LICENSE</div> </div> <div> <div></div> <div>docker-compose.yml</div> </div> </div>	問題 12 出力 デバッグ
--	---------------

`finetune`ディレクトリに`alpaca-lora`のリポジトリをクローンします。

```
cd finetune
git clone https://github.com/tloen/alpaca-lora.git
```

このリポジトリの中身を確認します。

```
(llama_finetune) [ユーザー名]@208:/home/data/[ユーザー名]/finetune$ cd alpaca-
lora/
(llama_finetune) [ユーザー名]@208:/home/data/[ユーザー名]/finetune/alpaca-
lora$ ls
DATA_LICENSE  LICENSE      alpaca_data.json
alpaca_data_gpt4.json  docker-compose.yml
export_state_dict_checkpoint.py  lengths.ipynb  requirements.txt  utils
Dockerfile      README.md    alpaca_data_cleaned_archive.json
export_hf_checkpoint.py  finetune.py
pyproject.toml  templates
```

上に表示されている`requirements.txt`から、必要なパッケージをインストールします。

```
pip install -r requirements.txt
```

これに加えて、

```
pip install -U "huggingface_hub[cli]"
pip install protobuf
```

も実行しておいてください。パッケージをインストールできたら、`huggingface_hub`にログインします。

```
huggingface-cli login
```

を実行してください。

```
huggingface-cli login
```

```

 _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_
 _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_
 _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_
 _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_
 _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_
 _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_
 _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_
 _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_ _|_

```

A token is already saved on your machine. Run `huggingface-cli whoami` to get more information or `huggingface-cli **logout**` if you want to **log out**.

Setting a new token will erase the existing one.

To **log in**, `huggingface\_hub` requires a token generated from <https://huggingface.co/settings/tokens>.

Enter your token (input will not be visible):

と表示されるので、**Enter your token (input will not be visible):**の後に、アクセストークン (hf\_\*\*\*\*\*のようなもの) を、自分のアカウントにて作成して貼り付けてください。アクセストークン発行の参考：<https://zenn.dev/tasiten/articles/c3c6adad12f792>

入力したテキストは見えないので注意してください。

**Add token as git credential? (Y/n)**はYを入力してください。**Login successful.**と表示されればログインできています。

## 5-2.学習用コードとデータの準備

先ほどクローンしたalpaca-loraのリポジトリに入っている**finetune.py**を一部編集します。編集後のコードを [github](#) に上げてあるので、それをクローンして差し替えます。適当なディレクトリに移動して、以下のようにクローンしてください。

```
git clone https://github.com/DING-1994/part-time-kennsyu.git
```

その中の**finetune.py**と**check\_result.py**をalpaca-loraのディレクトリにコピーしてください。なお、変更箇所は以下の通りです。

- line 20 : peftのバージョンに合わせた変更

```
- prepare_model_for_int8_training,
+ prepare_model_for_kbit_training,
```



- line 119 : tokenizerの読み込み方法を変更

```
- tokenizer = LlamaTokenizer.from_pretrained(base_model)
+ tokenizer = AutoTokenizer.from_pretrained(base_model)
```

- line 174 : line20と同様、peftのバージョンに合わせた変更

```
- model = prepare_model_for_int8_training(model)
+ model = prepare_model_for_kbit_training(model)
```

- line 264~271 : モデルのコンパイルを行う部分を削除(この部分が残っていると後でモデルを読み込む際にエラーになります)

```
-     model.state_dict = (
-         lambda self, *_ , **__: get_peft_model_state_dict(
-             self, old_state_dict()
-         )
-     ).__get__(model, type(model))
-
-     if torch.__version__ >= "2" and sys.platform != "win32":
-         model = torch.compile(model)
```

## 6.llama2のfine-tuning

`finetune.py`のパラメータを調節して、実際に`llama2-7b`で学習してみます。以下のようなハイパーパラメータがあります。

### 6-1.train()の引数

```
def train(
    # model/data params
    base_model: str = "", # the only required argument
    data_path: str = "yahma/alpaca-cleaned",
    output_dir: str = "./lora-alpaca",
    # training hyperparams
    batch_size: int = 128,
    micro_batch_size: int = 4,
    num_epochs: int = 3,
    learning_rate: float = 3e-4,
    cutoff_len: int = 256,
    val_set_size: int = 2000,
    # lora hyperparams
    lora_r: int = 8,
    lora_alpha: int = 16,
    lora_dropout: float = 0.05,
```

```

    lora_target_modules: List[str] = [
        "q_proj",
        "v_proj",
    ],
    # llm hyperparams
    train_on_inputs: bool = True, # if False, masks out inputs in loss
    add_eos_token: bool = False,
    group_by_length: bool = False, # faster, but produces an odd training
loss curve
    # wandb params
    wandb_project: str = "",
    wandb_run_name: str = "",
    wandb_watch: str = "", # options: false | gradients | all
    wandb_log_model: str = "", # options: false | true
    resume_from_checkpoint: str = None, # either training checkpoint or
final adapter
    prompt_template_name: str = "alpaca", # The prompt template to use,
will default to alpaca.
):

```

- base\_model

fine-tuningを行うモデルの名前

- data\_path

fine-tuningに使用するデータセットのパス

- output\_dir

学習した重みを保存するディレクトリ

- batch\_size

パラメータの更新前に処理するデータ数(データセットの数未満の場合はミニバッチ学習)

- micro\_batch\_size バッチを小さなバッチに分割する際のサイズ,メモリ制約を回避するために使用される

batch\_size: This parameter defines the number of training examples used in one iteration of model training. In deep learning, the entire dataset is typically divided into smaller batches, and the model's weights are updated after processing each batch. micro\_batch\_size: This is often used in scenarios where the batch size is large and the available memory (RAM or VRAM) is limited. The large batch is divided into smaller 'micro-batches'. Each micro-batch is processed sequentially, but the weight update is performed only after the entire batch is processed. [Fine-Tuning LLaMA2 with Alpaca Dataset Using Alpaca-LoRA](#)

- num\_epoch

トレーニングのエポック数

- cutoff\_len

トークンの最大長

- `val_set_size`

テストに使用するデータ数

参考：

258798行 51759個のデータセット→`val_set_size=2000` : 3.8640... %

- `lora_r`

LoRAアダプタのランク：8,16,32など

In the Microsoft LoRA repository, which is the implementation they released in 2021 with the paper, their examples used ranks of 8 or 16. According to Mark Huang from Gradient, you should use a rank of 32 or higher. However, in the more recent QLoRA paper, they introduce a quantized version of LoRA which we'll discuss next, but they also did a lot of experimentation to show that there is very little statistical difference between ranks of 8 and 256. We find LoRA `r` is unrelated to final performance if LoRA is used on all layers... So if you're rank is 8 or above, it simply may not matter. Perhaps this could change as different models come out that use weights and parameters more efficiently to compress information. For example, Mistral 7B can compete with Llama 13B. Does that mean that the weights in Mistral 7B contain more information, that the model is therefore intrinsically higher rank, and requires more precision for LoRA fine-tuning? [LoRA Fine-tuning & Hyperparameters Explained \(in Plain English\)](#)

- `lora_alpha`

`lora_alpha/lora_r` : スケーリング係数が学習の重みとなる

`lora_r(rank)`に対して`lora_alpha`を増加させると、fine-tuningの効果が増加する

- `lora_dropout`

過学習を防ぐために使用される、トレーニング可能なパラメーターが人為的にゼロに設定される確率

This is the probability that a trainable parameter will be artificially set to zero for a given batch of training. It's used to help prevent overfitting the model to your data. The QLoRA paper set dropout to 0.1 or 10% for fine-tuning 7B and 13B models, and reduced it to 0.05 or 5% for 33B and 65B models. [LoRA Fine-tuning & Hyperparameters Explained \(in Plain English\)](#)

- `lora_target_modules`

LoRAアダプタを適用するモデルの層

## 6-2.transformers.Trainerの引数

```
trainer = transformers.Trainer(  
    model=model,  
    train_dataset=train_data,  
    eval_dataset=val_data,  
    args=transformers.TrainingArguments(  

```

```

        per_device_train_batch_size=micro_batch_size,
        gradient_accumulation_steps=gradient_accumulation_steps,
        warmup_steps=100,
        num_train_epochs=num_epochs,
        learning_rate=learning_rate,
        fp16=True,
        logging_steps=10,
        optim="adamw_torch",
        evaluation_strategy="steps" if val_set_size > 0 else "no",
        save_strategy="steps",
        eval_steps=200 if val_set_size > 0 else None,
        save_steps=200,
        output_dir=output_dir,
        save_total_limit=3,
        load_best_model_at_end=True if val_set_size > 0 else False,
        ddp_find_unused_parameters=False if ddp else None,
        group_by_length=group_by_length,
        report_to="wandb" if use_wandb else None,
        run_name=wandb_run_name if use_wandb else None,
    ),
    data_collator=transformers.DataCollatorForSeq2Seq(
        tokenizer, pad_to_multiple_of=8, return_tensors="pt",
        padding=True
    ),
)

```

- evaluation\_strategy モデルの評価
  - steps : eval\_stepsで指定したステップ数ごとに評価
  - epoch : epochごとに評価
  - no : 評価を行わない
- save\_strategy 上と同様
- eval\_steps 評価を行うステップ数の間隔
- save\_steps モデルを保存するステップ数の間隔

今回は1の**train()**の引数を以下のように設定して学習します。

```

def train(
    # model/data params
    base_model: str = "meta-llama/Llama-2-7b-hf", # the only required
    argument
    data_path: str = "path_to/japanese_alpaca_data_100.json",
    output_dir: str = "output_dir",
    # training hyperparams
    batch_size: int = 4,
    micro_batch_size: int = 4,
    num_epochs: int = 10,

```

```

learning_rate: float = 3e-4,
cutoff_len: int = 256,
val_set_size: int = 20,
# lora hyperparams
lora_r: int = 64,
lora_alpha: int = 16,
lora_dropout: float = 0.05,
lora_target_modules: List[str] = [
    "q_proj",
    "v_proj",
],
# llm hyperparams
train_on_inputs: bool = True, # if False, masks out inputs in loss
add_eos_token: bool = False,
group_by_length: bool = False, # faster, but produces an odd training
loss curve
# wandb params
wandb_project: str = "",
wandb_run_name: str = "",
wandb_watch: str = "", # options: false | gradients | all
wandb_log_model: str = "", # options: false | true
resume_from_checkpoint: str = None, # either training checkpoint or
final adapter
prompt_template_name: str = "alpaca", # The prompt template to use,
will default to alpaca.
):

```

以下の2つのパスは、自分の環境に合わせて変更してください。

```

data_path: str = "path_to/japanese_alpaca_data_100.json",
output_dir: str = "output_dir",

```

`data_path`は`part-time-kennsyu`をクローンしたディレクトリの中にある

`japanese_alpaca_data_100.json`のパスを指定してください。 `output_dir`は学習した重みを保存するディレクトリのパスを自分で決めて、それを指定してください。 VSCodeでファイルやディレクトリを右クリックして「パスのコピー(Shift+Alt+C)」を選択すると、そのパスをコピーできます。

ここまでできたら、実際に学習します。ターミナルで`finetune.py`のあるディレクトリまで移動します。

学習は`screen`を使って行います。普段使っているターミナルでは、ターミナルを閉じたりssh接続を切ったりすると実行中のプロセスも終了してしまいます。一方`screen`を使うとターミナルを閉じても処理を続けることができます。普通fine-tuningは時間がかかるので、`screen`の中で実行して夜間や休日も学習を続けます。

今回のfine-tuningは数分で完了しますが、練習のために`screen`で実行します。まず`screen -ls`を実行してみてください。 `screen`をまだ作っていないので、以下になるとと思います。

```

(llama_finetune) [ユーザー名]@208:/home/data/[ユーザー名]/finetune/alpaca-
lora$ screen -ls
No Sockets found in /run/screen/S-[ユーザー名].

```

`screen -S finetune`で、`finetune`という名前のscreenを作成します。画面が新しいターミナルに切り替わります。screenに入るとAnaconda環境は(base)になっているので、再度`conda activate llama_finetune`を実行してください。`pwd`や`ls`でディレクトリを確認し、

```
python finetune.py
```

で学習を開始します。ターミナルでは以下のように進捗が表示されます。

```
(llm_finetune) kato@08:/home/data/kato/2025_demo/finetune/alpaca-lora-7b python finetune.py
Training Alpaca-LoRA model with params:
base_model: meta-llama/llama-2-7b-hf
data_path: /home/data/kato/2025_demo/datasets/japanese_alpaca_data_100.json
output_dir: /home/data/kato/2025_demo/results/0324/2-7b-param_fix_100
batch_size: 4
micro_batch_size: 4
num_epochs: 10
learning_rate: 0.0003
cutoff_len: 256
val_set_size: 4
lora_r: 64
lora_alpha: 16
lora_dropout: 0.05
lora_target_modules: ['q_proj', 'v_proj']
train_on_inputs: True
add_eos_token: False
group_by_length: False
wandb_project:
wandb_run_name:
wandb_watch:
wandb_log_model:
resume_from_checkpoint: False
prompt_template: alpaca

The 'load_in_4bit' and 'load_in_8bit' arguments are deprecated and will be removed in the future versions. Please, pass a `BitsAndBytesConfig` object in 'quantization_config' argument instead.
Loading checkpoint shards: 100% |██████████████████████████████████████████████████████████████████████████████| 2/2 [00:02<00:00, 1.07s/it]
trainable params: 33,554,432 || all params: 6,771,970,048 || trainable%: 0.4955
Map: 100% |██████████████████████████████████████████████████████████████████████████████| 96/96 [00:00<00:00, 2217.88 examples/s]
Map: 100% |██████████████████████████████████████████████████████████████████████████████| 4/4 [00:00<00:00, 781.32 examples/s]

/home/kato/anaconda3/envs/llama_finetune/lib/python3.10/site-packages/transformers/training_args.py:1594: FutureWarning: `evaluation_strategy` is deprecated and will be removed in version 4.46 of 🤗 Transformers
  . use `eval_strategy` instead
  warnings.warn(
No label_names provided for model class `PeftModelForCausalLM`. Since `PeftModel` hides base models input arguments, if label_names is not given, label_names can't be set automatically within `Trainer`. Note that
t empty label_names list will be used instead.
0% |██████████████████████████████████████████████████████████████████████████████| | 0/240 [00:00<, ?it/s]

/home/kato/anaconda3/envs/llama_finetune/lib/python3.10/site-packages/torch/_dynamo/eval_frame.py:745: UserWarning: torch.utils.checkpoint: the use_reentrant parameter should be passed explicitly. In version 2.5
we will raise an exception if use_reentrant is not passed. use_reentrant=False is recommended, but if you need to preserve the current default behavior, you can pass use_reentrant=True. Refer to docs for more details on the differences between the two variants.
  return fn(*args, **kwargs)

/home/kato/anaconda3/envs/llama_finetune/lib/python3.10/site-packages/bitsandbytes/autograd/functions.py:315: UserWarning: MatMul8BitLt: inputs will be cast from torch.float32 to float16 during quantization
  warnings.warn(f"MatMul8BitLt: inputs will be cast from {A.dtype} to float16 during quantization")
{'loss': 1.7026, 'grad_norm': 0.10405836999416351, 'learning_rate': 2.999999999999999e-05, 'epoch': 0.42}
{'loss': 1.5634, 'grad_norm': 0.11113231629133224, 'learning_rate': 5.999999999999999e-05, 'epoch': 0.83}
{'loss': 1.5498, 'grad_norm': 0.20349252223968506, 'learning_rate': 8.999999999999999e-05, 'epoch': 1.25}
{'loss': 1.1479, 'grad_norm': 0.1992180347442627, 'learning_rate': 0.0001199999999999999, 'epoch': 1.67}
{'loss': 0.938, 'grad_norm': 0.424541711807251, 'learning_rate': 0.00015, 'epoch': 2.08}
{'loss': 0.7773, 'grad_norm': 0.21544712781906128, 'learning_rate': 0.0001799999999999999, 'epoch': 2.5}
{'loss': 0.7492, 'grad_norm': 0.20940746366977692, 'learning_rate': 0.0002099999999999999, 'epoch': 2.92}
{'loss': 0.7069, 'grad_norm': 0.19095571339130402, 'learning_rate': 0.0002399999999999999, 'epoch': 3.33}
{'loss': 0.6677, 'grad_norm': 0.26168105006217957, 'learning_rate': 0.00027, 'epoch': 3.75}
{'loss': 0.6438, 'grad_norm': 0.253477543592453, 'learning_rate': 0.0003, 'epoch': 4.17}
{'loss': 0.5408, 'grad_norm': 0.4279008209705353, 'learning_rate': 0.00027857142857142854, 'epoch': 4.58}
{'loss': 0.5773, 'grad_norm': 0.3691804111003876, 'learning_rate': 0.0002571428571428571, 'epoch': 5.0}
{'loss': 0.4561, 'grad_norm': 0.3967415090650913, 'learning_rate': 0.00023571428571428569, 'epoch': 5.42}
{'loss': 0.4081, 'grad_norm': 0.47830310463309534, 'learning_rate': 0.00021428571428571427, 'epoch': 5.83}
{'loss': 0.3979, 'grad_norm': 0.3548683524131775, 'learning_rate': 0.00019285714285714286, 'epoch': 6.25}
{'loss': 0.3312, 'grad_norm': 0.48417016863822937, 'learning_rate': 0.0001714285714285714, 'epoch': 6.67}
{'loss': 0.2944, 'grad_norm': 0.6132766008377075, 'learning_rate': 0.00015, 'epoch': 7.08}
{'loss': 0.256, 'grad_norm': 0.6945967078208923, 'learning_rate': 0.00012857142857142855, 'epoch': 7.5}
77%
```

学習が始まったら、一度screenから抜けます。**Ctrl+a**を押した後に**d**を押すとscreenから抜けることができます。元のターミナルに戻ったら、**screen -ls**を実行してみてください。今度は以下のように、先ほど作ったscreenが表示されるはずです。

```
(llama_finetune) [ユーザー名]@208:/home/data/[ユーザー名]/finetune/alpaca-  
lora$ screen -ls  
There is a screen on:  
          3464387.finetune          (2025年03月24日 10時36分33秒)    (Detached)  
1 Socket in /run/screen/S-[ユーザー名].
```

`screen -r finetune`で先ほど作ったscreenに接続できます。

screenの使い方は以下の通りです。

- `screen -ls`でscreenの一覧を表示できます。screenを作った後はこのコマンドでscreenの一覧が表示されます。
- `screen -S [screen名]`でscreenを作成できます。
- `screen -r [screen名]`でscreenに接続できます。
- `Ctrl+a`を押した後に`d`を押すとscreenから抜けることができます。
- `screen -ls`で表示されるPIDを使って、`kill [PID]`でscreenを削除できます。

fine-tuningはGPUを使って行われるので、その様子を確認します。VSCodeでもう1つターミナルを開き、

```
watch -n 1 nvidia-smi
```

を実行してください。以下のように各GPUの使用状況が表示されます。

問題 13 出力 デバッグ コンソール ターミナル ポート コメント

Every 1.0s: nvidia-smi

Mon Mar 24 09:37:51 2025

NVIDIA-SMI 535.183.01										Driver Version: 535.183.01										CUDA Version: 12.2									
GPU Name Persistence-M										Bus-Id Disp.A										Volatile Uncorr. ECC									
Fan Temp Perf Pwr:Usage/Cap										Memory-Usage										GPU-Util Compute M.									
																				MIG M.									
=====																													
0 NVIDIA A100 80GB PCIe On										00000000:81:00.0 Off										0									
N/A 42C P0 65W / 300W										21432MiB / 81920MiB										0% Default Disabled									
-----																													
1 NVIDIA A100 80GB PCIe On										00000000:82:00.0 Off										0									
N/A 56C P0 292W / 300W										21182MiB / 81920MiB										16% Default Disabled									
-----																													
2 NVIDIA A100 80GB PCIe On										00000000:C1:00.0 Off										0									
N/A 53C P0 78W / 300W										21192MiB / 81920MiB										87% Default Disabled									
-----																													
3 NVIDIA A100 80GB PCIe On										00000000:C2:00.0 Off										0									
N/A 49C P0 68W / 300W										27590MiB / 81920MiB										29% Default Disabled									
-----																													
-----																													
Processes:																													
GPU GI CI										PID Type Process name										GPU Memory Usage									
ID ID																													
=====																													
0 N/A N/A 1439 G										/usr/lib/xorg/Xorg										4MiB									
0 N/A N/A 2551003 C										python										18154MiB									
0 N/A N/A 3428457 C										python										3250MiB									
1 N/A N/A 1439 G										/usr/lib/xorg/Xorg										4MiB									
1 N/A N/A 2551003 C										python										17968MiB									
1 N/A N/A 3428457 C										python										3186MiB									
2 N/A N/A 1439 G										/usr/lib/xorg/Xorg										4MiB									
2 N/A N/A 2551003 C										python										17978MiB									
2 N/A N/A 3428457 C										python										3186MiB									
3 N/A N/A 1439 G										/usr/lib/xorg/Xorg										4MiB									
3 N/A N/A 2551003 C										python										23288MiB									
3 N/A N/A 3428457 C										python										4274MiB									
-----																													

下側にあるProcessesの欄を見ると、同じPID(この画像では3428457)のPythonプロセスがGPU0~3を使っていることがわかります。これがfine-tuningのプロセスです。

## 結果の確認

finetune.pyの実行が終わったら、output\_dirに指定したパスに重みが保存されていることを確認してください。この重みを使って学習の結果を確認します。check\_result.pyの以下の部分を設定します。

まずは、fine-tuning前のモデルの出力を確認しておきます。



- 30行目にある `lora_weights` は `None` にしてください。
- 32行目の `instruction` は、今回日本語データセットでfine-tuningを行ったので、"Respose in Japanese."としています。
- 33行目の `input` は、llama2に入力したいテキストを指定してください。

```
python check_result.py
```

を実行すると、最後にモデルの応答が表示されます。

次にfine-tuning後のモデルの出力を確認します。 **30行目にある `lora_weights` に、 `output_dir` と同じパスを指定してください。** 再び `check_result.py` を実行し、モデルの出力が先ほどと違うことを確認してください。

私の環境で `input: str = "毎日楽しく過ごすには?"` とすると、出力は以下のようになりました。

fine-tuning前：

```
Response: 毎日楽しく過ごすには？
```

```
### Instruction:  
Respose in Japanese.
```

```
### Input:  
毎日楽しく過ごすには？
```

fine-tuning後：

```
Response: 「毎日楽しく過ごすには、定期的に友人と歩き、音楽を聴き、本を読みます。また、定期的に運動し、自然に接するように生活してください。」 </s>
```

学習前は日本語の入力をそのまま返していましたが、学習後は入力に対して日本語で回答しています。