初心者のための Fairseqチュートリアル

NAIST 博士前期課程 やみんちゅ

事前準備

- チュートリアル用のGitHubリポジトリがここから利用できます.

  データとライブラリダウンロードのためにlibrary_installation.shと work_dialog/data_download.sh, work_caption/data_download.shを実行しておくと良い（時間はかなりかかる）

- 想定環境
  - NTT Dialog Transformerを動かす場合 → 24GB以上のGPUメモリ
  - 筆者はLinux Ubuntu18.04で動作を確認しています

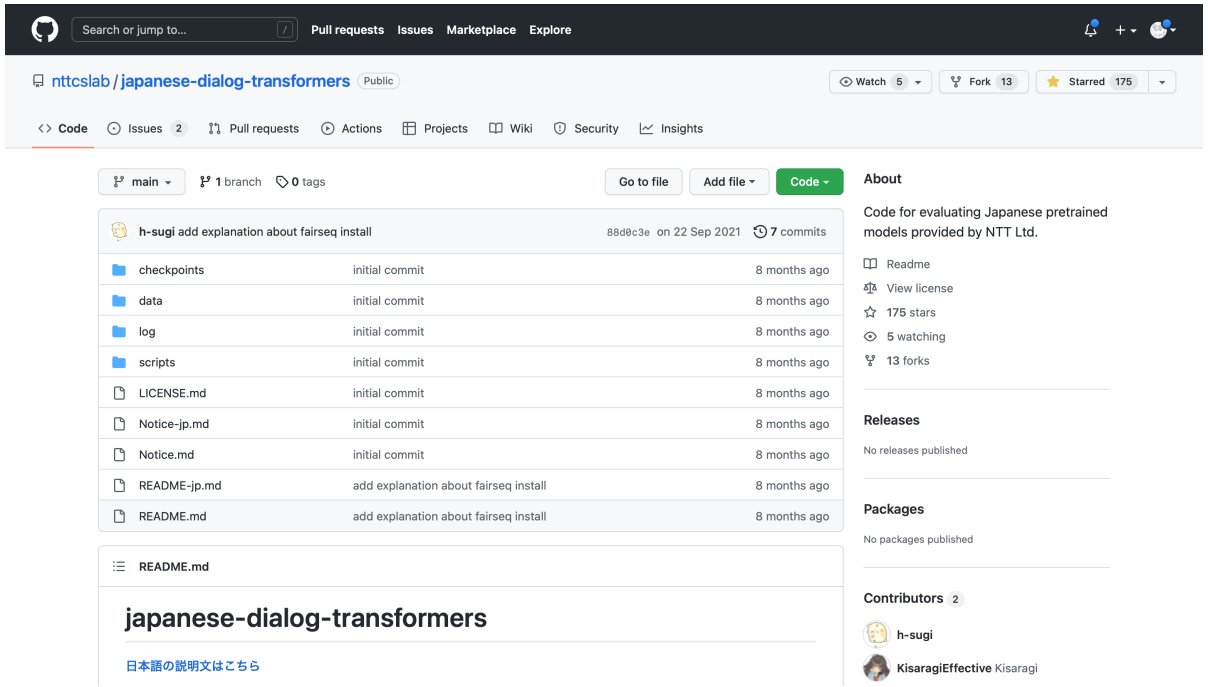
- 想定読者
  - Pytorchでモデルを組める
  - Linuxの基本的なコマンドを使える
  - 深層学習の基本的な知識がある

※このスライドの情報は2022年6月現在のものです

日本語対話モデルがアツい

- 2021年11月にNTT CS研が大規模対話モデルを公開

  ついでに日本語版PersonaChat，EmpatheticDialoguesデータセットも公開．いっぱい遊べる



- **しかし，Meta製言語モデル学習ライブラリのFairseqを使えないと遊べない😢**

このチュートリアルの動機

- そもそもFairseqは自然言語処理で一般的なライブラリなので使えるようにしておきたい
- しかし「落とし穴が多い&日本語資料が少ない」という厄介なライブラリ
- このチュートリアルで，Fairseqを使って対話システムを研究するための道具を獲得してほしい

目次

1. Fairseqについての基本的な情報を知ろう！
2. Fairseqで基本的な使い方を学ぼう！
3. Fairseqを自分の研究用にカスタマイズしよう！（応用編）

目次

1. Fairseqについての基本的な情報を知ろう！
2. Fairseqで基本的な使い方を学ぼう！
3. Fairseqを自分の研究用にカスタマイズしよう！

Fairseqとは？

PytorchをベースにMeta (旧Facebook) が作った，sequence2sequenceモデル特化の深層学習ライブラリ

Pytorchベースのライブラリとしては他にHuggingface Transformersなど

- 特徴
  - 充実したCLIツール（後述）
  - マルチGPUでの動作を簡単に実現できる
  - Architectureやcriterionなどについて簡易に拡張可能
  - 多様なconfiguration
- 主な使い方
  - コマンドラインからデータセットや学習済みモデルのパスを指定して学習
  - Interactive機能を使って生成モデルを簡単に試してみる
  - 評価機能を使って簡単に学習したモデルを評価

Fairseqとは？

PytorchをベースにMeta (旧Facebook) が作った，sequence2sequenceモデル特化の深層学習ライブラリ

Pytorchベースのライブラリとしては他にHuggingface Transformersなど

- 特徴
  - 充実したCLIツール（後述）
  - マルチGPUでの動作を簡単に実現できる
  - Architectureやcriterionなどについて簡易に拡張可能
  - 多様なconfiguration
- 主な使い方
  - コマンドラインからデータセットや学習済みモデルのパスを指定して学習
  - Interactive機能を使って生成モデルを簡単に試してみる
  - 評価機能を使って簡単に学習したモデルを評価

今回扱うところ

fairseq-cli

コマンドプロンプトからfairseqを簡易に使うためのCLIツール群（標準で付属）

- fairseq-preprocess

  データセットの前処理

- fairseq-train

  モデルの学習

- fairseq-generate

  与えたテストセットに対して学習済みモデルから出力を生成

- fairseq-interactive

  インタラクティブに学習済みモデルから出力を生成

- fairseq-score

  BLEUスコアを計算

- fairseq-eval-lm

  Perplexityを計算

fairseq-cli

コマンドプロンプトからfairseqを簡易に使うためのCLIツール群

- fairseq-preprocess
  データセットの前処理
- fairseq-train
  モデルの学習

今回扱うところ

- fairseq-generate
  与えたテストセットに対して学習済みモデルから出力を生成
- fairseq-interactive
  インタラクティブに学習済みモデルから出力を生成
- fairseq-score
  BLEUスコアを計算
- fairseq-eval-lm
  Perplexityを計算

目次

1. Fairseqについての基本的な情報を知ろう！
2. Fairseqで基本的な使い方を学ぼう！
3. Fairseqを自分の研究用にカスタマイズしよう！

ここではレポジトリのwork_dialogディレクトリで作業をするよ！

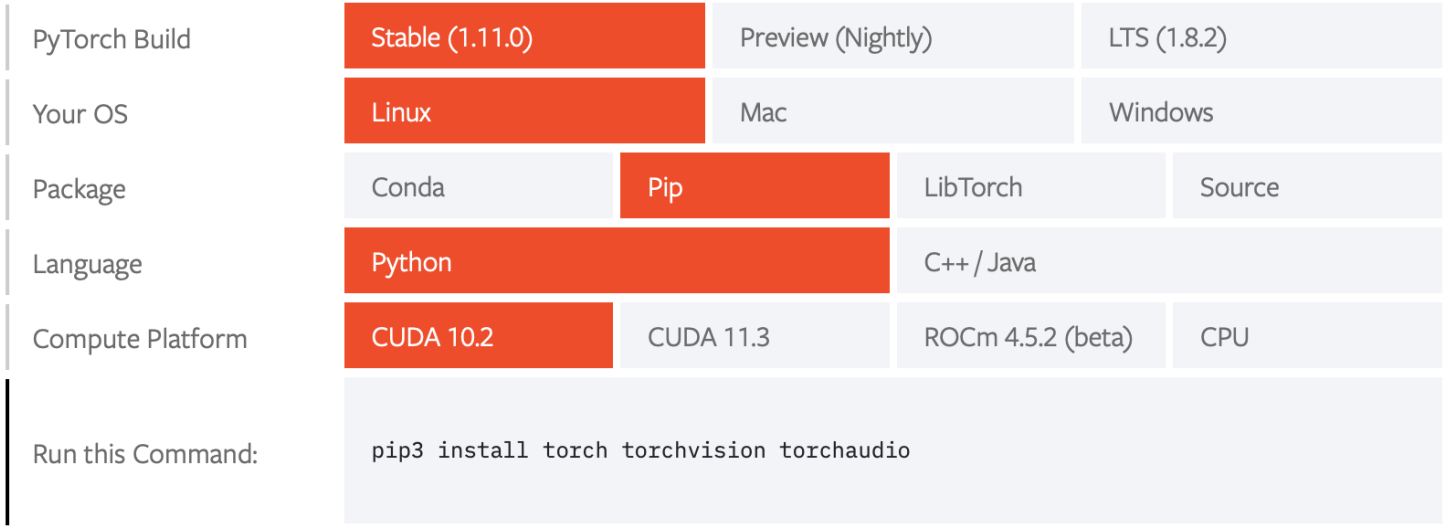
この章でやること

NTT Japanese Dialog Transformers をJPersonaChatデータセットでファインチューニングする．これを通して，Fairseqの基本的な使い方について学習する．

- この章でできるようになること
  1. Fairseqの導入
  2. fairseq-preprocessで前処理
  3. fairseq-trainで学習
  4. fairseq-generateでテストセットから文を生成
  5. fairseq-interactiveで簡単に対話モデルと会話

まずはPytorchを導入する

- FairseqはPytorchをベースにしたライブラリなので導入は必須
- Pytorchはバージョン・OS・GPU・インストール方法・CUDAによってインストールコマンドが変わる
- 公式ページの案内に従ってインストールしよう



親切なことにポチポチすればインストールコマンドがわかるようになっている

Fairseqの導入と最初の罠

罠なんてあるの？
$ pip install fairseq で余裕でしたよ

pip経由からは旧バージョンしか入れられないので注意ぞい！

- ***Fairseq*の最新ビルドは公式の*github*レポジトリからしか入手できない（重要）**
- 正しいインストールコマンドは以下

work/tutorial-install.sh

```
$ git clone https://github.com/pytorch/fairseq
$ cd fairseq
$ pip install --editable ./
```

基本のおさらい～テキストの前処理の流れ～

1. テキストを train/valid/test の3種類に分割する
   - チーティングを防止するため，３つのデータセットは絶対に混ぜてはいけない
   - 翻訳および対話では，ソース(=input)文と教師文が対になるようにする

2. それぞれの文をトークンごとに分割する
   - 分割方法としてはSentencepiece, BertJapaneseTokenizerなど
   - 事前学習済みモデルと同等の方法で実施する

3. トークンをID化する
   - そのままでは扱いづらいので，トークンをトークンリスト（=dictionary）中のインデックスの数字に変換する
   - Dictionaryは事前学習済みモデルで用いられたものと同じものを使う.

JPersonaChatから学習データを作る

- まずはJPersonaChatデータセットをtrain/valid/testに分割しよう
  - 💡作業前にに移動しておこう

```
$ python scripts/extract_persona.py japanese_persona_chat.xlsx data/personal/raw/
```

ポイント

翻訳タスクとしてfairseqで学習するので，**src**（対話履歴）と**dst**（教師文）の２種類のデータが必要

```
data
   -test.dst
   -test.src
   -train.dst
   -train.src
   -valid.dst
   -valid.src
```

フォルダ構造がこうなっていればok

fairseq-preprocessを使った前処理（失敗例）

あとは適当にfairseq-preprocessを使って終わり！

```
#!/bin/bash
fairseq-preprocess ¥
--trainpref data/train --validpref data/valid --testpref data/test ¥
--source-lang src --target-lang dst ¥
--destdir data/perchat/bin ¥
--tokenizer space ¥
--srcdict data/sp/dicts/sp_oall_32k.txt ¥
--tgtdict data/sp/dicts/sp_oall_32k.txt
```

トークン数がやけに少ない上に，半分ぐらい<unk>(辞書中に存在しないトークン)に置き換わってる…

```
2022-05-23 16:29:41 | INFO | fairseq_cli.preprocess | [src] Dictionary: 32002 types
2022-05-23 16:29:41 | INFO | fairseq_cli.preprocess | [src] data/perchat/train.src: 1381 sents, 2762 tokens, 50.0% replaced by <unk>
2022-05-23 16:29:41 | INFO | fairseq_cli.preprocess | [src] Dictionary: 32002 types
2022-05-23 16:29:41 | INFO | fairseq_cli.preprocess | [src] data/perchat/valid.src: 178 sents, 356 tokens, 50.0% replaced by <unk>
2022-05-23 16:29:41 | INFO | fairseq_cli.preprocess | [src] Dictionary: 32002 types
2022-05-23 16:29:41 | INFO | fairseq_cli.preprocess | [src] data/perchat/test.src: 189 sents, 378 tokens, 50.0% replaced by <unk>
2022-05-23 16:29:41 | INFO | fairseq_cli.preprocess | [dst] Dictionary: 32002 types
2022-05-23 16:29:41 | INFO | fairseq_cli.preprocess | [dst] data/perchat/train.dst: 1381 sents, 2762 tokens, 50.0% replaced by <unk>
2022-05-23 16:29:41 | INFO | fairseq_cli.preprocess | [dst] Dictionary: 32002 types
2022-05-23 16:29:41 | INFO | fairseq_cli.preprocess | [dst] data/perchat/valid.dst: 178 sents, 356 tokens, 50.0% replaced by <unk>
2022-05-23 16:29:41 | INFO | fairseq_cli.preprocess | [dst] Dictionary: 32002 types
2022-05-23 16:29:41 | INFO | fairseq_cli.preprocess | [dst] data/perchat/test.dst: 189 sents, 378 tokens, 50.0% replaced by <unk>
2022-05-23 16:29:41 | INFO | fairseq_cli.preprocess | Wrote preprocessed data to data/perchat/bin
```

いきなりfairseq-preprocessを使おうとしたのが原因

fairseq-preprocessの役割

fairseq-preprocessではトークンのID化処理とバイナリファイルの作成を行うぞい！

トークン分割自体はあらかじめ自分で行う必要があるぞい！

- トークンのID化処理
  - 指定された区切り文字でトークンを認識する

あらかじめ分割処理をしていないと文全体が1トークンと見做されてうまくいかない

```
2022-05-23 16:29:41 | INFO | fairseq_cli.preprocess | [src] Dictionary: 32002 types
2022-05-23 16:29:41 | INFO | fairseq_cli.preprocess | [src] data/perchat/train.src: 1381 sents, 2762 tokens, 50.0% replaced by <unk>
2022-05-23 16:29:41 | INFO | fairseq_cli.preprocess | [src] Dictionary: 32002 types
2022-05-23 16:29:41 | INFO | fairseq_cli.preprocess | [src] data/perchat/valid.src: 178 sents, 356 tokens, 50.0% replaced by <unk>
2022-05-23 16:29:41 | INFO | fairseq_cli.preprocess | [src] Dictionary: 32002 types
2022-05-23 16:29:41 | INFO | fairseq_cli.preprocess | [src] data/perchat/test.src: 189 sents, 378 tokens, 50.0% replaced by <unk>
2022-05-23 16:29:41 | INFO | fairseq_cli.preprocess | [dst] Dictionary: 32002 types
2022-05-23 16:29:41 | INFO | fairseq_cli.preprocess | [dst] data/perchat/train.dst: 1381 sents, 2762 tokens, 50.0% replaced by <unk>
2022-05-23 16:29:41 | INFO | fairseq_cli.preprocess | [dst] Dictionary: 32002 types
2022-05-23 16:29:41 | INFO | fairseq_cli.preprocess | [dst] data/perchat/valid.dst: 178 sents, 356 tokens, 50.0% replaced by <unk>
2022-05-23 16:29:41 | INFO | fairseq_cli.preprocess | [dst] Dictionary: 32002 types
2022-05-23 16:29:41 | INFO | fairseq_cli.preprocess | [dst] data/perchat/test.dst: 189 sents, 378 tokens, 50.0% replaced by <unk>
2022-05-23 16:29:41 | INFO | fairseq_cli.preprocess | Wrote preprocessed data to data/perchat/bin
```

  - 認識したトークンをdictionaryに従いID化する
- バイナリファイルの作成
  - 実行時に高速でファイルを読み込むために，あらかじめ各ファイルをバイナリ化する

正しい前処理とfairseq-preprocess①

Sentencepieceによるトークン分割

- 生の文からサブワードへ分割するライブラリ（詳細はここ）
- 事前学習で利用したものと必ず同じ分割用モデルを使う

```
$ python scripts/tokenize.py
```

scripts/tokenize.py

```
import sentencepiece as spm

sp = spm.SentencePieceProcessor()
sp.Load("data/sp/sp_oall_32k.model")

def tokenize(raw_text):
    tokenized = sp.EncodeAsPieces(raw_text)
return tokenized
```

はあ。わたしなで肩がコンプレックスでさ。

↓ サブワード分割

はあ。わたしなで肩がコンプレックスでさ。

正しい前処理とfairseq-preprocess②

fairseq-preprocessの実行

```
$ sh preprocess.sh
```

preprocess.sh

```
fairseq-preprocess ¥
--trainpref data/train --validpref data/valid --testpref data/test ¥
--source-lang src --target-lang dst ¥
--destdir data/perchat/bin ¥
--tokenizer space ¥
--srcdict data/sp/dicts/sp_oall_32k.txt ¥
--tgtdict data/sp/dicts/sp_oall_32k.txt
```

- --trainpref, --validpref, --testpref: 各データのパスのプレフィックス
- --source-lang, --target-lang: ソース・ターゲットのデータの拡張子を指定
- --destdir: バイナリ化したデータの保存先ディレクトリ（指定先が存在しない場合は作成も行う）
- --tokenize: トークンの分割文字を指定
- --srcdict, --tgtdict: ソース・ターゲットそれぞれのID<->単語辞書を指定（ここではsentencepiece modelに同梱されているものを使えばok）

fairseq-trainで学習する

```
$ sh train.sh
```

train.sh

```
fairseq-preprocess ¥
fairseq-train data/perchat/bin/ ¥
--arch transformer ¥
--finetune-from-model data/pretrained/japanese-dialog-transformer-1.6B.pt ¥
--task translation ¥
--save-dir result/perchat/ ¥
--criterion cross_entropy ¥
--batch-size 4 ¥
…（中略・アークテクチャの設定など）
--save-interval 5 ¥
--lr 0.000001 ¥
--max-epoch 20 ¥
--optimizer adafactor ¥
```

ここで行うのは

- 学習のための各種設定

  batch-sizeやcriterion，optimizerなど

- モデルの形状定義

  architectureなど（ややこしくなるのでここは中略）

大事なのは

- 第1引数にバイナリ化したデータセットのパス（data/perchat/bin）を与えること
- Taskにtranslationを設定すること

学習についての補足

学習設定について

- 学習率を低く(E-5以下)，バッチサイズを大きく(64以上)して学習すると吉

  低学習率・大バッチサイズは色んな大規模言語モデルの学習に有効な設定（？）

  いい感じのところを自分で探してみてください

トラブルシューティング

- 学習に異様に時間が掛かる
  - 学習のボトルネック → 大規模言語モデルはモデル保存に時間が掛かる（環境によるが下手すると1回30分ぐらいかかる）
  - --save-intervalオプションで何epoch毎に保存するか指定できるので，これを長く設定しておこう（おすすめは5epoch程度）

- GPUにバッチが載らない！
  - Batch Accumulationを使おう

    「小さなミニバッチ毎の勾配の和を保存しておき、その勾配の和を使ってパラメータ更新をすることにより、擬似的に大きなミニバッチサイズの学習をさせる」手法 *
  - --batch-size × --update-freq (パラメータ更新を止める間隔) が64以上になるように設定しよう

*https://akichan-f.medium.com/大きなミニバッチの効力と-tensorflowを使って大きなミニバッチを学習させる方法-gradient-accumulation-3147d89eb43f

トラブルシューティング

- 外部的な問題で学習が止まってしまった！
  - 特にクラスタマシン環境だと落雷で電源が落ちた，とかセッションが認証切れになった，とかで止まることが稀によくある．
  - その時は落ち着いて，設定を何も変えずにfairseq-trainを回しなおそう．
  - Fairseqが自動で--save-dirに保存されたモデルを読み込み，途中のエポックから学習を再開してくれる．

fairseq-interactiveでシステムと会話する

- サンプリングに関する設定を変更して，インタラクティブな会話をすることができる.
- このコマンドではモデルは直前の発話のみを参照して発話を生成する.

  過去の文脈を保持するやり方は，Japanese dialog transformerのリポジトリを参照
  （というかこのやり方を参照しないと対話モデルとしてはまともに動きません）

$ sh interactive.sh

interactive.sh

```
fairseq-interactive data/perchat/bin/ ¥
--path data/pretrained/japanese-dialog-transformer-1.6B.pt¥
--beam 10 ¥
 --bpe sentencepiece ¥
 --sentencepiece-model data/sp/sp_oall_32k.model ¥
…
--no-repeat-ngram-size 3 ¥
--nbest 10 ¥
--sampling ¥
--sampling-topp 0.9 ¥
--temperature 1.0
```

- pathには先ほど学習させたモデルを指定する
- bpeにsentecepiece，sentencepieceモデルにトークン化処理で使用したモデルと同じものを指定する.
- サンプリングについてのオプション説明は省略（基本的には左に記載した通りの値で良いはず）

fairseq-generateと生成ログの見方

- オプションはfairseq-interactiveと同様なので省略

  というか，やっていること自体はinteractiveと同じ．同じ処理をtestデータに適用して一気に生成例を出しているだけ．

```
$ sh generate.sh
```

generate.sh

```
#!/bin/bash
fairseq-generate ¥
…
```

- 生成例の見方は少し特殊なので注意

```
S-0     こんにちは
W-0     0.366   seconds
H-0     -1.4847989082336426     こんにちは
D-0     -1.4847989082336426     こんにちは
P-0     -1.4516 -1.5180
```

S: ソース文，T: ターゲット文，W: 推論時間，H: トークン化された推論結果，

D: デトークナイズされた推論結果

P: トークン毎の確率(logit)？

ここから先は沼なので

一旦休憩

目次

1. Fairseqについての基本的な情報を知ろう！
2. Fairseqで基本的な使い方を学ぼう！
3. Fairseqを自分の研究用にカスタマイズしよう！

ここではレポジトリのwork_captionディレクトリで作業をするよ！

この章でやること

Legacyな画像キャプション生成モデルの学習を通して，自作のモデル・criterion・画像データの利用の仕方を学ぶ

- この章でできるようになること
  1. Fairseqの構成と仕組みの理解
  2. Fairseqに自作のモデルとcriterionの組み込み
  3. Fairseqで画像データセットの読み込み

ここまで使い方を学びましたが…

仕組みは無視して機能を使えることを目標に説明したが,

そもそもFairseqってどういう仕組みで動いてる？🤔

- モデルはどこでインスタンスしてるの？
- Criterionはどこで呼び出してる？
- fairseq-trainやfairseq-generateは結局何をしていたの？
- テキストデータしか読み込めないの？

カスタマイズするにはどうしたらいい？🤔

- 自作モデル・自作criterionを使うには？
- 画像データや強化学習の報酬を読み込むには？

- 解き明かすには”task”について理解することが必要

“task” is all you need.

- task とは？
  - 扱いたいタスクの種類に応じて学習に必要なモデルのインスタンス・バッチ処理・パラメータ更新・サンプリングなどの一連の機能を集約したクラス（いわゆるpipelineに近い）
  - CLIツールはtaskのインスタンスとconfigurationの受け渡しを担うのが主な仕事（極端に言えば）
  - taskが全ての司令塔
  - taskは種類によって読み込めるデータセット・タスク設定が異なる（次スライド）

fairseq-train
fairseq-generate
fairseq-interactive

インスタンス

taskクラス

モデル・optimizerのインスタンス
データの読み込みとバッチ処理
Criterionの呼び出し
…etc

Taskの種類

主な系統

- Language Modeling 系

  事前学習を行う

- Speech 系

  音声翻訳等扱う．音声データを扱える

- Translation 系

  テキスト翻訳を扱う．対話もこのtaskを用いる

詳細は右図（もしくは公式HPのCLIツールのページ）を参照

Possible choices: multilingual_language_modeling, speech_unit_modeling, hubert_pretraining, translation, multilingual_translation, semisupervised_translation, translation_from_pretrained_xlm, speech_to_text, text_to_speech, frm_text_to_speech, legacy_masked_lm, audio_pretraining, audio_finetuning, sentence_ranking, online_backtranslation, simul_speech_to_text, simul_text_to_text, cross_lingual_lm, denoising, multilingual_denoising, multilingual_masked_lm, language_modeling, masked_lm, speech_to_speech, sentence_prediction, translation_from_pretrained_bart, sentence_prediction_adapters, translation_multi_simple_epoch, translation_lev, dummy_lm, dummy_masked_lm, dummy_mt

task

Default: “translation”

ここまで使い方を学びましたが…

仕組みは無視して機能を使えることを目標に説明したが,

そもそもFairseqってどういう仕組みで動いてる？🤔

- モデルはどこでインスタンスしてるの？ → **taskクラスでインスタンスしている**
- Criterionはどこで呼び出してる？ → **taskクラスで呼び出してる**
- fairseq-trainやfairseq-generateは結局何をしていたの？ → **taskクラスのインスタンス**
- テキストデータしか読み込めないの？ → **taskが対応していれば音声も読み込める**

カスタマイズするには

- 自作モデル・自作

- 画像や強化学習の報酬を読み込むには？

taskが全てを解決する

- 解き明かすには ”task” について理解することが必要

Fairseqをカスタマイズするには

ここまでの話で考えると既存のtaskを改変してあげれば良さそうだが…

モデルとcriterionのために他のコードとの整合性を保ちながらtaskの改変するのしんどくないですか?

モデルとcriterionの改変なら，register機能で簡単に実現できるぞい

既存taskで対応していない画像データもtaskの改変なしで…?

無理だから大人しくtaskを改変するのじゃ!
Taskのregister機能を使えば既存のコードを破壊せずに実装できるぞい！

register機能

- 自作のモデル・criterion・taskをFairseqの外部から登録する機能
- @register_hogehogeという形式の修飾子として使用

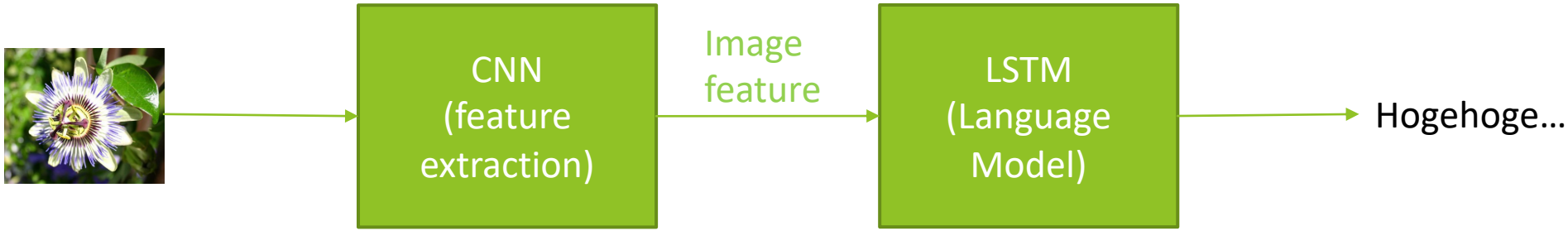
model	criterion	task
@register_model() @register_model_architecture()	@register_criterion()	@register_task()

使うregister機能の対照表

- 自作モデルを使う際には一つのフォルダにコードをまとめておき，そのフォルダのパスを実行時にオプションで指定する.

今回の目標

CNN + LSTMのレガシーな感じで画像キャプションをする



- CNN (Encoder)は画像特徴量抽出器として働く

  今回はCNNの学習はしない（その方が性能が出るはず&学習時間の軽量化のため）
- 画像特徴量はLSTMの台書のトークンとして入力される.
- LSTM (Decoder) は画像特徴量を元にして単語系列を予測する.

自作モデルを作成する①

まずはEncoder，Decoderのモデルを定義しよう

```
class CNNEncoder(FairseqEncoder):
    def __init__(self, embed_size=256):
        # Load the pretrained ResNet-152 and replace top fc layer.
        super(CNNEncoder, self).__init__(dictionary=None)
        …

    def forward(self, images, src_lengths):
        ….

class LSTMDecoder(FairseqDecoder):
    def __init__(self, dictionary, encoder_hidden_dim=256,
embed_dim=256,  hidden_dim=256):
        super().__init__(dictionary)
        …

    def forward(self, prev_output_tokens, encoder_out):
        ….
```

- 基本的には既存のモデルのコードと見比べながら作るのがよい
- ここで注意すべき点は
  1. 整合性を楽に維持するために FairseqEncoder, FairseqDecoderクラスを継承して書く
  2. forward関数の引数も，既存の実装から変更しないように書く

     引数を変更すると学習用のコードとの整合性が取れなくなってしまう

自作モデルを作成する②

次にEncoderDecoderModelを定義しよう

```
@register_model('image_caption')
class ImageCaptionModel(FairseqEncoderDecoderModel):
    @staticmethod
    def add_args(parser):
        parser.add_argument('--encoder-embed-dim', type=int,
        metavar='N',help='dimensionality of the encoder embeddings’,)
        ….

    @classmethod
    def build_model(cls, args, task):
        encoder = CNNEncoder(embed_size=args.encoder_embed_dim)
        decoder = LSTMDecoder(
        dictionary=task.target_dictionary,
        encoder_hidden_dim=args.encoder_hidden_dim,
        embed_dim=args.decoder_embed_dim,
        hidden_dim=args.decoder_hidden_dim,
        )
        model = ImageCaptionModel(encoder, decoder)
        return model
```

- @register_model(model_name)をクラス定義の直前に書き，fairseqにモデルを認識させる
- CLIツールで追加オプションを指定したい場合，add_args関数内で定義する．
- モデルをインスタンスするためのbuild_model関数をオーバーライドする．
  1. クラス自体をFairseqEncoderDecoderModelを継承して作成する．
  2. 次に，build_model関数内で，自己参照する形でmodelをインスタンスして返す．

自作モデルを作成する②

最後にarchitectureを定義しよう

```
@register_model_architecture('image_caption', 'image_caption')
def tutorial_simple_caption(args):
    args.encoder_embed_dim = getattr(args, 'encoder_embed_dim', 256)
    args.encoder_hidden_dim = getattr(args, 'encoder_hidden_dim', 256)
    args.decoder_embed_dim = getattr(args, 'decoder_embed_dim', 256)
    args.decoder_hidden_dim = getattr(args, 'decoder_hidden_dim', 256)
```

- Tutorial_simple_caption関数を使って，CLIのオプションから読み込みたいものを追加する．（関数名は何でも良い）
- @register_model_architecture(model_name, architecture_name)で先ほどの関数を修飾すると，architectureがfairseqに登録され，CLIツールから使用可能になる．
- 修飾された側の関数には引数としてargs (CLIツールから読み込んだオプション)が渡される

自作criterionを作成する

```
@register_criterion("focal", dataclass=CrossEntropyCriterionConfig)
class ForcalLoss(FairseqCriterion):
    …
    def compute_loss(self, model, net_output, sample, reduce=True):
        lprobs = model.get_normalized_probs(net_output, log_probs=True)
        lprobs = lprobs.view(-1, lprobs.size(-1))
        target = model.get_targets(sample, net_output).view(-1)

        loss = torch.tensor([])
        for lprob, tgt in zip(lprobs, target):
        loss = F.nll_loss(lprob, tgt, ignore_index=self.padding_idx, reduction="sum" if reduce else "none",)
        loss = torch.stack((loss, loss))
        loss = loss.mean()

        return loss, loss
```

- Criterionのクラス自体はFairseqCriterionクラスを継承しておくのが無難
- @register_criterionで自作クラスを修飾し，fairseq側に認識させる
- 基本的にはcompute_loss関数をオーバーライドしておけば良い

自作taskを作成する①

```
@register_task('captioning')
class CaptioningTask(FairseqTask):
    @staticmethod
    def add_args(parser):
        …
    @classmethod
    def setup_task(cls, args, **kwargs):
        captions_dict = Dictionary()
        return CaptioningTask(args, captions_dict)
    …
```

- add_args関数の働きはmodelのものと同様
- setup_task関数はtask自体をインスタンスするための関数で，build_model関数とやっていることは同じ

自作taskを作成する②

```
@register_task('captioning')
class CaptioningTask(FairseqTask):
    ....
    def load_dataset(self, split, **kwargs):
        ....
        self.datasets[split] = ImageCaptionDataset(img_ds=image_ds,
        cap_ds=captions_ds,
        cap_dict=self.captions_dict,shuffle=True)


    ....
```

- load_dataset関数をオーバーライドし，自作関数をここで読み込ませることが重要
- 辞書型のself.datasetsに，train, test, valid (splitで指定している) ごとにデータセットを格納するように処理を書く

自作taskを作成する③

念の為，データセットクラスについても説明

```
class ImageCaptionDataset(FairseqDataset):
    …
    def collater(self, samples):
        …
        return {
            'id': indices,
                'net_input': {
                    'src_tokens': torch.stack(source_image_batch, dim=0),
                    'src_lengths': source_lengths,
                    'prev_output_tokens': rotate_batch,
                },
                'target': target_batch,
                'ntokens': target_ntokens,
                'nsentences': num_sentences,
         }
```

- 作成するDatasetはFairseqDatasetを継承する
- Fairseq独特の関数としてcollaterがある

  これはミニバッチを返す関数になっている.

  辞書形式のValueに対応するミニバッチ化されたデータが入っているという認識でok

CLIツールから自作したものを呼び出そう

- まず，scripts/customフォルダの中に，fairseqから各モジュールが見えるように__init__.pyを作成する．

scripts/custom/__init__.py
```
from . import task
from . import models
from . import criterion
```

- 次に，CLIツールのオプションで--user-dirオプションを指定する．

train.sh
```
#!/bin/bash

fairseq-train ¥
--user-dir scripts/customs ¥
--arch image_caption ¥
--task captioning ¥
--criterion focal ¥
…
```

--arch, --task, --criterionで指定するのは，
各registerの引数として入れた名前

sh train.sh で学習が回り始めれば，自作コードの作成が成功！

ポイント

- model, task, criterionそれぞれにお約束的な書き方があるので，自作する前に一度既存実装を観察しておくと良い

  基本的にはこのチュートリアルのリポジトリを参考にすれば良いと思う…

- 各関数の入出力は下手にいじらない方が良い

  入出力を変更すると，関数の呼び出し先のコードとの整合性で失敗してしまう可能性大

チュートリアルはここまで！
お疲れ様でした！

おわりに ～参考ページ集～

- 公式HP https://fairseq.readthedocs.io/en/latest/
- 公式レポジトリ https://github.com/facebookresearch/fairseq
- fairseq for image captioning https://github.com/krasserm/fairseq-image-captioning

  Transformerベースのimage captioningと強化学習関連の実装がされている