@IT
atmarkIT

「Python＋PyTorch」と「JoeyNMT」で学ぶニューラル機械翻訳

太田 麻裕美 , 八楽［著］

ニューラル機械翻訳（NMT）の基礎を「JoeyNMT」で学んでみよう（準備編）

精度向上により、近年利用が広まっている「ニューラル機械翻訳」。その仕組みを、自分で動かしながら学んでみましょう。第 1 回は海外の大学で教材として使われている「JoeyNMT」のインストール方法やモデルの訓練方法を紹介します。

（2022 年 06 月 29 日）

ハイデルベルク大学の博士課程に在籍しながら、八楽という会社で「ヤラクゼン」の開発に携わっている太田です。ヤラクゼンは、AI 翻訳から翻訳文の編集、ドキュメントの共有、翻訳会社への発注までを 1 つにする翻訳プラットフォームです。

本連載の目的、前提知識、構成

機械翻訳のフレームワークはよく知られたものが幾つか存在しますが、高機能であるが故にコードベース自体が巨大です。そのため機械翻訳の学習に行き着く前に、フレームワーク特有の仕様などを調べる段階で挫折してしまうことが少なからずあると感じています。

本連載は、機械翻訳モデル開発の経験がほとんどない初心者でも「モデルを実行してみる」ことができるようになるのを目的としています。フレームワークには、小規模で初学者にも扱いやすい「JoeyNMT」を使用します。

なお読み進めてもらうに当たって、以下 3 点の前提知識があることを想定しています。

1. Python の基本的なプログラミングができる
2. 機械学習、特にニューラルネットワークの基礎が分かる（損失関数、バックプロパゲーション、ドロップアウトなどを学んだことがある）
3. 自然言語処理の基礎が分かる（トランスフォーマー、エンコーダー、デコーダー、サブワードトークン、単語埋め込み、ビームサーチなどを聞いたことがある）

機械翻訳の理論的な部分は割愛し「プログラムを実行する」という実践部分に焦点を当てました。実践という意味では 2 点目、3 点目は必須ではないかもしれませんが、解説なしで扱いますので適宜補っていただければと思います。

本連載は、全 3 回を予定しています。第 1 回で、英日翻訳を題材にフレームワークの基本的な使い方を学びます。第 2 回では、フレームワークをカスタマイズする方法を具体例とともに紹介します。第 3 回では、発展編として、音声データを扱えるようにフレームワークを拡張し、自動音声認識、音声翻訳に挑戦してみましょう。

第 1 回の今回は、前半でフレームワークのインストールと学習済みモデルからの翻訳文生成を試します。後半でモデルを訓練する方法、訓練したモデルを評価する方法を概観します。

「JoeyNMT」とは

JoeyNMT は、教育目的に開発されたコンパクトな機械翻訳フレームワークです。海外の大学では入門レベルの授業で採用されている他、論文などで見たアイデアを軽く再現実装してみるといった場面で多く活用されています[※1]。PyTorch で書かれている点も、初心者にはうれしいポイントではないでしょうか。

※ 1：MutNMT、AIMS Senegal、Masakhane など。本記事の FAQ はハイデルベルク大学の機械翻訳ゼミや卒業論文などで JoeyNMT を使用する大学生からの質問に基づいています。

機械翻訳モデル開発の基本的な部分をサクッと学びたい、新しいアイデアを簡単に試したい、という要望に応えてくれるのが JoeyNMT です。大規模な汎用（はんよう）フレームワークを使って一度機械翻訳に手を出してみたものの、そのフレームワークのどの部分がどのように機械翻訳に使われているのか見つけられなかった、コードを読み始めたけれど、どの部分を書き換えれば自分のアイデアを実装できるのか分からなかったという人にも JoeyNMT はぴったりです。逆に言えば、プロダクションに耐え得る機能や性能を持つフレームワークを探している人には物足りないかもしれません。

本連載を執筆するに当たって、一般公開されている JoeyNMT では少しつまずきやすいと思われる、古い依存ライブラリや日本語トークナイズなどの部分をアップデートしたコードを公開しています（※ 2）。アップデートしたバージョンを便宜的に「JoeyNMT2.0」と呼ぶことにします。

※ 2：公開に当たり、JoeyNMT の作者である Julia Kreutzer 氏に JoeyNMT2.0 を監修していただきました。

JoeyNMT の詳細は、以下の論文を参照してください。初心者の学習のたやすさに関するユーザースタディーなど、興味深いデータが載っています。

Kreutzer, J., Bastings, J., & Riezler, S. (2019). Joey NMT: A Minimalist NMT Toolkit for Novices. In Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP): System Demonstrations (pp. 109-114). Association for Computational Linguistics.

論文を読むのはちょっとハードルが高いという方は、JoeyNMT の作者 Julia Kreutzer 氏によるブログ記事も参考にしてみてください。

JoeyNMT2.0 のダウンロードとインストール

JoeyNMT2.0 をインストールします。Python 3.9、PyTorch 1.11.0、CUDA 11.5 の環境で動作を確認しています。

```
$ pip install git+https://github.com/may-/joeynmt.git
```

なお、Google colab では、Python 3.9 に対応するようブランチを指定してインストールする必要があるので注意してください。

```
$ pip install git+https://github.com/may-/joeynmt.git@py3.7
```

学習済みモデルを試す

学習済みモデルをダウンロードする

まずは簡単に試せるよう、事前学習済みモデルを準備しました。以下の URL からダウンロードしてください。JparaCrawl という大規模英日パラレルコーパスで訓練したモデル [※3] のチェックポイント、設定ファイル、語彙（ごい）ファイルが同梱（どうこん）されています。

```
$ wget https://www.cl.uni-heidelberg.de/statnlpgroup/joeynmt/tutorial_enja.
tar.gz
$ tar -xvf tutorial_enja.tar.gz
$ ls tutorial_enja
avg5.ckpt  config.yaml  spm.en.model  spm.ja.model  src_vocab.txt  trg_vocab.
txt
```

※3：このモデルの利用条件は、JparaCrawl のライセンスに基づきます。

インタラクティブ翻訳モード

JoeyNMT には「train」「test」「translate」という 3 つのモードがあります。「translate」モードを選択してインタラクティブに翻訳文を生成してみましょう。

ダウンロードしたフォルダに入っている「config.yaml」ファイルを指定してください。

```
$ python -m joeynmt translate tutorial_enja/config.yaml
2022-05-01 22:01:45,749 - INFO - root - Hello! This is Joey-NMT (version
2.0.0).
2022-05-01 22:01:46,572 - INFO - joeynmt.model - Building an encoder-decoder
model...
2022-05-01 22:01:49,456 - INFO - joeynmt.model - Enc-dec model built.
2022-05-01 22:01:55,974 - INFO - joeynmt.helpers - Load model from tutorial_
enja/avg5.ckpt.
Please enter a source sentence:
I like apples.
2022-05-01 22:08:08,262 - INFO - joeynmt.prediction - Validating on 1 data
points... (Beam search decoding with beam size = 5, alpha = 1.0)
2022-05-01 22:08:10,779 - INFO - joeynmt.prediction - Generation took
2.5159s[sec]. (No references given)
JoeyNMT: Hypotheses ranked by score
JoeyNMT #1: 私はリンゴが好きです。
Please enter a source sentence:
^C
Bye.
```

設定ファイルで指定されたパスからモデル、トークナイザーなどのモジュールが読み込まれます。「Please enter a source sentence:」と表示されたら、翻訳したい文を入力します。今回利用している事前学習済みモデルは英語→日本語のモデルですので、英語を入力してください。日本語訳の候補が表示されます。［Ctrl］＋［C］キーでインタラクティブ翻訳モードを終了できます。

実行に関するよくある質問

Q：マルチ GPU の環境でインタラクティブ翻訳を試したらエラーが出ました。

A：インタラクティブ翻訳モードは Multi-GPU に対応していません。Single GPU または CPU で試してみてください。

Q：複数の候補（n best）を出力させることはできますか？

A：はい、出力できます。「config.yaml」の「testing」セクションにある「n_best」の値を変更してください。

ミニコラム：機械翻訳の難しさ

統計的機械翻訳からニューラル機械翻訳にパラダイムが移り、特にトランスフォーマーアーキテクチャが登場して以降、機械翻訳の精度は劇的に向上し続けています。中でも流ちょう性は、人間の翻訳者と見分けがつかないといわれることもあるほどです。そんな機械翻訳ですが、ディープラーニングの恩恵を受けてもなお、難しいといわれている点が複数あります。

可変長の入力、出力

画像分類などのタスクは、画像のサイズ（ピクセル数）がどのデータポイントも同じという設定がほとんどです。機械翻訳に使われる対訳データでは、短い文もあれば長い文もあり、異なる長さの文対を扱う必要があります。可変長の系列を扱う自然言語処理タスクの中でも、PoS Tagging など出力長が入力長と同じタスクに比べ、機械翻訳では出力長が入力長に必ずしも一致しない（事前に出力長が確定していない）という難しさもあります。

構造予測

入力画像に対して「ネコ」というラベルを予測する、あるいは入力テキストに対してそのジャンル「スポーツ」を予測するといった分類タスクとは異なり、機械翻訳の出力は構造をもった系列です。特定の構造を出力するためには、膨大な候補から適切な構造を効率良く探索する必要があります。

未知語

機械翻訳では、訓練時に見たことがない単語を予測する場面もあり得ます。「Covid-19」といった単語は、数年前には機械学習訓練データにほとんど表れていませんでした。あるいは、新しく設定された絵文字などのように文字そのものが訓練されたモデルの語彙（ごい）に入っていないということもあります。このような未知語をうまく扱う工夫が必要です。

あいまいな正解ラベル

翻訳には、「ただ一つの正解」と呼べるものは基本的には存在しません。ある 1 つの英語の文の意味を表現する日本語の文は、何通りも考えられます。特に日本語には表記ゆれ（「りんご、リンゴ、林檎」「繰越、繰り越し」「半角、全角の違い」など）も多く、表現だけでなく表記さえも一意に定まりません。Web から自動で取得された対訳データには、互いに翻訳になっていない文対などノイズも一定数混じってしまいます。教師あり機械翻訳では、このようなあいまいな教師データからモデルを学習させることになります。

言語間で対応しない言い回し

言語には、複数の意味があったり、あるいは字義通りの意味だけでなく「裏の意味」が込められているといったこともよくあります。そのような多義的な言い回しは、言語間で 1 対 1 対応していないことも多いです。例えば、本質的に異なっていて本来比べられないものを無理やり比べることを、英語で「comparing apples and oranges（リンゴとオレンジを比べる）」と表現することがありますが、ドイツ語では「Äpfel mit Birnen vergleichen（リンゴと洋ナシを比べる）」と言うのが一般的です。このようなフレーズを翻訳するには、字義通りの翻訳「oranges ⇔ Orangen（オレンジ）」「pears ⇔ Birnen（洋ナシ）」を学習しているだけでは足りません。

これは機械翻訳に限った話ではありませんが、どのように大量の対訳データを集めるのか、巨大なモデルを訓練するためのリソースをどのように確保するか、といった実務的な問題もあります。近年の機械翻訳モデルはどんどん巨大化しており、論文などで提案されているものの中には、学生や個人の開発者には手をだせないような規模の非公開データと高額な計算資源を前提にしているものも少なくありません。このような困難を克服するさまざまなアイデアが今、世界中で研究されています。本連載が、機械翻訳研究、開発の世界に飛び込むきっかけの一つになればと願っています。

モデルを訓練する

対訳データの準備

JoeyNMT v1.x は、1 行 1 文のプレーンテキスト形式のファイルを入力として受け付けます。JoeyNMT v2.0 で、Huggingface の datasets ライブラリからの入力もサポートしました。これにより、メモリに乗り切らないような大規模データの扱いも容易になりました。今回は、datasets に入っている The Business Scene Dialogue corpus を使ってみることにします。

id	no	en	ja
190315_E001_13	1	How is it going, Wayne?	ウェイン、調子はどうです？
190315_E001_13	2	I'm not too bad.	まあまあです。
190315_E001_13	3	Thank you very much for coming out today.	今日はご足労ありがとう。

データセットの一例

JoeyNMT はデータの情報（ファイルパスなど）を config.yaml の data セクションで指定します。トークナイズ、ノーマライズなどの事前処理もここで設定できます。事前学習済みモデルのフォルダに同梱されている config.yaml を見てみましょう。

```yaml
name: "tutorial_enja"
joeynmt_version: "2.0.0"
data:
  train: "bsd_ja_en"            # 訓練データのパス（今回の場合はhuggingface datasets.load_dataset() の`path`に渡される値）
  dev: "bsd_ja_en"              # 開発データのパス
  test: "bsd_ja_en"             # テストデータのパス
  dataset_type: "huggingface"   # データセットタイプ {"plain", "tsv", "huggingface"}
  dataset_cfg:                  # `datasets.load_dataset()`に渡される引数
    ignore_verifications: True
  src:
    lang: "en"                  # 入力言語コード
    max_length: 512             # 1文当たりの最大トークン数（これより長い文は、訓練時に無視される）
    min_length: 2               # 1文当たりの最小トークン数（これより短い文は、訓練時に無視される）
    lowercase: False            # 小文字にするかどうか
    normalize: True             # 正規化するかどうか
    level: "bpe"                # トークナイズレベル {"char", "word", "bpe"}
    voc_limit: 5000             # 語彙数
    voc_file: "tutorial_enja/src_vocab.txt" # 語彙ファイルのパス
    tokenizer_type: "sentencepiece" # トークナイザ {"sentencepiece", "subword-nmt"}
    tokenizer_cfg:              # トークナイザ（今回の場合はsentencepieceモデル）に渡される引数
      model_file: "tutorial_enja/spm.en.model"
  trg:
    lang: "ja"                  # 出力言語コード
    max_length: 512             # 1文当たりの最大トークン数（これより長い文は、訓練時に無視される）
    min_length: 2               # 1文当たりの最小トークン数（これより短い文は、訓練時に無視される）
    lowercase: False            # 小文字にするかどうか
    normalize: True             # 正規化するかどうか
    level: "bpe"                # トークナイズレベル {"char", "word", "bpe"}
    voc_limit: 9000             # 語彙数
    voc_file: "tutorial_enja/trg_vocab.txt" # 語彙ファイルのパス
    tokenizer_type: "sentencepiece" # トークナイザ {"sentencepiece", "subword-nmt"}
    tokenizer_cfg:              # トークナイザ（今回の場合はsentencepieceモデル）に渡される引数
      model_file: "tutorial_enja/spm.ja.model"
testing:
  n_best: 5                     # nbestサイズ
  beam_size: 5                  # ビームサイズ（1を指定するとGreedyDecoding）
  beam_alpha: 1.0               # ビームサーチのbrevity penalty
  batch_size: 2048              # ミニバッチのサイズ
  batch_type: "token"           # ミニバッチのサイズを文の数で決めるか、トークンの数で決めるか
  max_output_length: 100        # 最大出力トークン数
```


→目次に戻る

```
  eval_metrics: ["bleu"]          # 評価尺度
  sacrebleu_cfg:                  # sacrebleuに渡される引数
    tokenize: "ja-mecab"
training:
  load_model: "tutorial_enja/avg5.ckpt"
  reset_best_ckpt: False
  reset_scheduler: False
  reset_optimizer: False
  reset_iter_state: False
  random_seed: 42                 # ランダムシード
  optimizer: "adam"               # 最適化アルゴリズム
  adam_betas: [0.9, 0.98]         # Adamのオプション
  loss: "crossentropy"            # 損失関数
  normalization: "tokens"         # バッチ損失を文の数で割るか、トークンの数で割るか
  scheduling: "warmupinversesquareroot" # 学習率スケジューラー
  learning_rate: 0.001            # 学習率
  learning_rate_min: 1.0e-09      # 学習率の最小値（これを下回ると訓練のイテレーションを打ち切る）
  learning_rate_warmup: 4000      # 学習率ウォームアップのステップ数
  clip_grad_norm: 1.0             # 勾配ノルムをこの値でクリップする
  weight_decay: 0.0               # L2正則化のファクター
  label_smoothing: 0.1            # ラベル平滑化のペナルティー
  batch_multiplier: 1             # バックプロパゲートするまでに蓄積する勾配のバッチ数
  batch_size: 4096                # ミニバッチのサイズ
  batch_type: "token"             # ミニバッチのサイズを文の数で決めるか、トークンの数で決めるか
  early_stopping_metric: "bleu" # 訓練を打ち切る基準となる尺度（開発データにおける値）
  epochs: 5                       # 訓練エポック数
  validation_freq: 100            # この数のバッチを処理するごとにバリデーションをする
  logging_freq: 20                # この数のバッチを処理するごとにログを表示する
  model_dir: "tutorial_enja_bsd"  # 訓練したモデルを書き出すフォルダ
  overwrite: False                # 上記フォルダがすでに存在するとき、上書きするかどうか
  shuffle: True                   # 訓練データの順番をシャッフルするかどうか
  use_cuda: True                  # CUDAを使うかどうか
  fp16: False                     # fp16（ハーフプレシジョン）を使うかどうか
  print_valid_sents: [0, 1, 2]  # 開発データの何番目の文をlogに表示するか
  keep_best_ckpts: 5              # モデルを書き出すとき、チェックポイントの数がこの数を超える場合は、開発データの評価スコアが悪いものから削除する（-1を指定すると全てのチェックポイントが保存される）
  num_workers: 0                  # ミニバッチを作る`collate_fn`（トークナイズやパディング）を処理するワーカーの数
model:
  initializer: "xavier"           # 重みの初期化方法
  embed_initializer: "xavier"     # 単語埋め込みの初期化方法
  bias_initializer: "zeros"       # バイアス項の初期化方法
  tied_embeddings: False          # 埋め込み層をエンコーダーとデコーダーで共有するかどうか
  tied_softmax: False             # Softmax層をエンコーダーとデコーダーで共有するかどうか
  encoder:
    type: "transformer"           # エンコーダーのタイプ
    num_layers: 8                 # エンコーダーのレイヤー数
```


→目次に戻る

```
    num_heads: 16              # マルチヘッドの数
    embeddings:
      embedding_dim: 1024      # 埋め込み層の次元
      scale: True              # 埋め込みの値を√d倍するかどうか
      dropout: 0.              # 埋め込み層のドロップアウト確率
    hidden_size: 1024          # 隠れ層の次元
    ff_size: 4096              # feed-forward層の次元
    dropout: 0.3               # ドロップアウト確率
    layer_norm: "pre"          # レイヤー正規化を適用する位置
  decoder:
    type: "transformer"        # デコーダーのタイプ
    num_layers: 6              # デコーダーのレイヤー数
    num_heads: 16              # マルチヘッドの数
    embeddings:
      embedding_dim: 1024      # 埋め込み層の次元
      scale: True              # 埋め込みの値を√d倍するかどうか
      dropout: 0.              # 埋め込み層のドロップアウト確率
    hidden_size: 1024          # 隠れ層の次元
    ff_size: 4096              # feed-forward層の次元
    dropout: 0.3               # ドロップアウト確率
    layer_norm: "pre"          # レイヤー正規化を適用する位置
```

joeynmt/configs フォルダに異なる設定のサンプルが幾つかありますので参考にしてください。スクラッチからモデルの訓練を始める場合は、scripts/build_vocab.py で語彙ファイルを生成します。

```
$ python scripts/build_vocab.py tutorial_enja/config.yaml
```

config.yaml で指定されたパスに語彙ファイルがすでに存在する場合は上書きされてしまいますので注意してください。また英語、ドイツ語ペアなど、単語埋め込みレイヤーを入力言語、出力言語で共有したいとき（tied embeddings）は、joint オプション（--joint）を付与してください。入力言語、出力言語を連結したファイルから語彙を学習させることができます。

よくある質問

Q：data セクションの normalize を True に設定すると、どのような正規化が行われますか？

A：Python のビルトイン「unicodedata」の NFKC 正規化が適用されます。詳しくは tokenizer.py の BaseTokenizer クラスを参照してください。

Q：入力言語、出力言語で共通の語彙（Joint Vocabulary）を使用しています。語彙ファイル「src_vocab.txt」「trg_vocab.txt」（またはトークナイザーのモデルファイル）を別々に準備しなければなりませんか？

A：いいえ、語彙ファイルは 1 つで大丈夫です。同じファイルへのパスを、src と trg の両方に設定してください。

Q：スクラッチでモデルを訓練します。語彙サイズはどのように決めたらよいですか？

A：JParaCrawl の論文は日英、英日翻訳に 3 万 2000 種類の語彙を利用しています。幾つか違う値を試してみて、dev データでのスコアが最も良かった値を採用するのが無難かもしれません。VOLT など、適切な語彙サイズを求める方法を提案している研究もあります。

Q：JoeyNMT 1.x で使われていた 1 行 1 文のプレーンテキスト形式のデータを、JoeyNMT2.0 でもそのまま使いたいです。

A：JoeyNMT 1.x の設定ファイルと同じように、train、dev、test に拡張子なしのファイルパスを指定できます。事前にトークナイズされているデータの場合は、level を「"word"」と指定することで、半角スペースで分割されるようになります。ただし、デトークナイズも半角スペースでの join になってしまうので、BPE 分割されたデータの場合は BaseTokenizer クラスの post_process 関数を適宜変更してください。

モデルの訓練

config.yaml の training セクションでパラメーターを指定し、モデルの訓練を制御します。

```
training:
  random_seed: 42                 # ランダムシード
  optimizer: "adam"               # 最適化アルゴリズム
  adam_betas: [0.9, 0.98]         # Adamのオプション
  loss: "crossentropy"            # 損失関数
  normalization: "tokens"         # バッチ損失を文の数で割るか、トークンの数で割るか
  scheduling: "warmupinversesquareroot" # 学習率スケジューラー
  learning_rate: 0.001            # 学習率
  learning_rate_min: 1.0e-09      # 学習率の最小値（これを下回ると訓練のイテレーションを打ち切る）
  learning_rate_warmup: 4000      # 学習率ウォームアップのステップ数
  clip_grad_norm: 1.0             # 勾配ノルムをこの値でクリップする
  weight_decay: 0.0               # L2正則化のファクター
  label_smoothing: 0.1            # ラベル平滑化のペナルティー
  batch_multiplier: 8             # バックプロパゲートするまでに蓄積する勾配のバッチ数
  batch_size: 2048                # ミニバッチのサイズ
  batch_type: "token"             # ミニバッチのサイズを文の数で決めるか、トークンの数で決めるか
  early_stopping_metric: "bleu" # 訓練を打ち切る基準となる尺度（開発データにおける値）
  epochs: 5                       # 訓練エポック数
  validation_freq: 1000           # この数のバッチを処理するごとにバリデーションをする
  logging_freq: 200               # この数のバッチを処理するごとにログを表示する
  model_dir: "tutorial_enja"      # 訓練したモデルを書き出すフォルダ
  overwrite: False                # 上記フォルダがすでに存在するとき、上書きするかどうか
  shuffle: True                   # 訓練データの順番をシャッフルするかどうか
  use_cuda: True                  # CUDAを使うかどうか
  fp16: False                     # fp16（ハーフプレシジョン）を使うかどうか
  print_valid_sents: [0, 1, 2]  # 開発データの何番目の文をlogに表示するか
  keep_best_ckpts: 5              # モデルを書き出すとき、チェックポイントの数がこの数を超える場合は、開発データの評価スコアが悪いものから削除する（-1を指定すると全てのチェックポイントが保存される）
  num_workers: 0                  # ミニバッチを作る`collate_fn`（トークナイズやパディング）を処理するワーカーの数
```

モデルの構成は model セクションで指定します。

```yaml
model:
  initializer: "xavier"          # 重みの初期化方法
  embed_initializer: "xavier"    # 単語埋め込みの初期化方法
  bias_initializer: "zeros"      # バイアス項の初期化方法
  tied_embeddings: False         # 埋め込み層をエンコーダーとデコーダーで共有するかどうか
  tied_softmax: False            # Softmax層をエンコーダーとデコーダーで共有するかどうか
  encoder:
    type: "transformer"          # エンコーダーのタイプ
    num_layers: 8                # エンコーダーのレイヤー数
    num_heads: 16                # マルチヘッドの数
    embeddings:
      embedding_dim: 1024        # 埋め込み層の次元
      scale: True                # 埋め込みの値を√d倍するかどうか
      dropout: 0.                # 埋め込み層のドロップアウト確率
    hidden_size: 1024            # 隠れ層の次元
    ff_size: 4096                # feed-forward層の次元
    dropout: 0.3                 # ドロップアウト確率
    layer_norm: "pre"            # レイヤー正規化を適用する位置
  decoder:
    type: "transformer"          # デコーダーのタイプ
    num_layers: 6                # デコーダーのレイヤー数
    num_heads: 16                # マルチヘッドの数
    embeddings:
      embedding_dim: 1024        # 埋め込み層の次元
      scale: True                # 埋め込みの値を√d倍するかどうか
      dropout: 0.                # 埋め込み層のドロップアウト確率
    hidden_size: 1024            # 隠れ層の次元
    ff_size: 4096                # feed-forward層の次元
    dropout: 0.3                 # ドロップアウト確率
    layer_norm: "pre"            # レイヤー正規化を適用する位置
```

設定ファイルが書けたら、train モードで joeynmt を起動します。

```
$ python -m joeynmt train tutorial_enja/config.yaml
```

データが読み込まれ、モデルが構成されて、訓練のイテレーションが始まります。model_dir には、訓練されたモデルのチェックポイント、logger の出力ファイル、開発データの評価スコア「validation.txt」が記録される他、tensorboard のログフォルダもここに生成されます。訓練開始時にフラグを付けることで、tensorboard でのアテンションの可視化も可能です。

```
$ tensorboard --logdir tutorial_enja/tensorboard
```

設定ファイルの training セクションでリセットパラメーターを False に設定すると、中断したモデルの訓練を続きから再開できます。

```
training:
  load_model: "tutorial_enja/latest.ckpt"
  reset_best_ckpt: False
  reset_scheduler: False
  reset_optimizer: False
  reset_iter_state: False
  model_dir: "tutorial_enja_resume"
  ...
```

model_dir には、中断前の訓練のときとは違う別のパスを指定してください。同じパスを指定すると中断前の訓練で保存されたチェックポイントが上書きされてしまうことがあります。

幾つかのチェックポイントの平均をとることで、より ロバストな予測を得られることが知られています。このチェックポイントの平均を取るためのスクリプト、scripts/checkpoint_averaging.py も準備されています。

```
$ python scripts/average_checkpoints.py --inputs model_dir/*0000.ckpt
--output model_dir/avg.ckpt
```

よくある質問

Q：data セクションの voc_limit と training セクションの batch_size は何が違うんでしょうか？

A：data セクションの voc_limit は、語彙の数、つまり重複を避けたユニークなトークン数です。例えば語彙数が 1000 のとき、モデルが次のトークンを予測するのに、1000 個の候補から確率が高いものを選ぶことになります。training セクションの batch_size は、ミニバッチの大きさです。batch_type が token のとき、batch_size で指定された数のトークンを含むようにミニバッチが作られます。

Q：「RuntimeError: CUDA out of memory.」というエラーが出ます。バッチサイズを小さくして精度が落ちるのは避けたいです。

A：もし GPU の数を増やせるのであれば、GPU の数を増やしてください。GPU の数が限られているのであれば、training セクションの batch_size を小さく（例えば半分に）し、その分、batch_multiplier を（例えば倍に）増やしてみてください。訓練時間は長くなってしまいますが、実質のバッチサイズを維持できます（詳細は「gradient accumulation」というキーワードで検索してみてください）

Q：「実質のバッチサイズ」はどうやって確認できますか？

A：訓練のイテレーションが始まる前に train.log に表示される Train stats の、effective batch size の項目で確認できます。

```
2022-05-02 16:16:43,980 - INFO - joeynmt.training - Train stats:
    device: cuda
    n_gpu: 4
    16-bits training: False
    gradient accumulation: 1
    batch size per device: 2048
    effective batch size (w. parallel & accumulation): 8192
```

Q：モデルサイズ（レイヤー数など）や他のハイパーパラメーターの値はどのように決めたらよいですか？
A：この質問に普遍的な答えはないように思います。手持ちのデータに近いデータで訓練されたモデルを探し、公開されている論文や他のツールキットで採用されているデフォルトの値をまずは試してみるのが安全かもしれません。その値を起点に、幾つか違う値を試してみて、dev データでのスコアが最も良かった値を採用することが多いです。

参考文献ガイド

より深くニューラル機械翻訳を学びたい方のために、参考になる jupyter notebook 形式のチュートリアルを幾つか挙げてみます。平易な英語で書かれており、コードや図も豊富なので、英語が苦手な方でも読み進められるはずです。

- Annotated Transformer：トランスフォーマーアーキテクチャで使われている自己注意機構がどのように計算されるのか解説されています
- Annotated Encoder-Decoder：機械翻訳で使われているエンコーダー・デコーダーモデルの仕組みを、サンプルコードから学ぶことができます
- Translation with a sequence to sequence network and attention：PyTorch で機械翻訳モデルを作ってみるチュートリアルです
- Transformer model for language understanding：Tensorflow で機械翻訳モデルを作ってみるチュートリアルです

モデルを評価する

JoeyNMT では、性能の評価に sacrebleu というライブラリを使用しています。sacrebleu は機械学習モデルの評価として一般的な BLEU スコア、ChrF スコアを利用できます。設定ファイルの testing セクションで評価に関わるパラメーターを指定します。

```
testing:
  n_best: 5                    # nbest
  beam_size: 5                 # ビームサイズ（1を指定するとGreedyDecoding）
  beam_alpha: 1.0              # ビームサーチのbrevity penalty
  batch_size: 2048             # ミニバッチのサイズ
  batch_type: "token"          # ミニバッチのサイズを文の数で決めるか、トークンの数で決めるか
  max_output_length: 100       # 最大出力トークン数
  eval_metrics: "bleu"         # 評価尺度
  sacrebleu:                   # sacrebleuに渡される引数
    tokenize: "ja-mecab"
```

JoeyNMT を test モードで起動します。評価に使いたいチェックポイントが明示的に与えられていない場合は、model_dir の中の best.ckpt が使われます。

```
$ python -m joeynmt test tutorial_enja/config.yaml &amp;#8211;ckpt tutorial_enja/avg.ckpt
2022-05-02 18:22:11,223 - INFO - root - Hello! This is Joey-NMT (version 2.0.0).
2022-05-02 18:22:11,224 - INFO - joeynmt.data - Building tokenizer...
[...]
2022-05-02 18:23:39,560 - INFO - joeynmt.prediction - Decoding on dev set...
2022-05-02 18:25:53,451 - INFO - joeynmt.prediction - Evaluation result (beam search) bleu:  11.01, generation: 131.5355[sec], evaluation: 2.0884[sec]
2022-05-02 18:25:54,992 - INFO - joeynmt.prediction - Decoding on test set...
2022-05-02 18:28:23,234 - INFO - joeynmt.prediction - Evaluation result (beam search) bleu:  11.57, generation: 147.7790[sec], evaluation: 0.2176[sec]
```

よくある質問

Q：思ったような精度がでません。

A:さまざまな原因が考えられます。まずは学習曲線を見てみましょう。train loss は下がっていますか？ validation loss は下がっていますか？　JoeyNMT の FAQ で、デバッグのためのレシピ集としてブログ記事が紹介されています。また、データのアラインメント、トークナイゼーション、フィルタリング、語彙ファイルの中身などを再度確認することを勧めています。

Q：ベースラインのモデルよりも良いスコアが出ました。この差は統計的に有意であるといえるでしょうか。

A：ランダムシードの値だけを変えて同じ設定の実験を繰り返し、どの程度スコアにばらつきが出るか調べます。sacrebleu は v2.0 で bootstrap resampling、approximate randomization という有意差検定（Significance tests）をサポートしました。このようなライブラリを使って差が有意かどうか調べることもできます。

今回は、yaml の設定ファイルを中心に、JoeyNMT を実行する方法を解説しました。第 2 回は JoeyNMT をカスタマイズする方法を学びます。JoeyNMT の強みは、コードを読んだり、書き換えたりするのが容易である点です。次回は JoeyNMT の長所を実感していただけるよう、実践例を交えて紹介していきます。

Discord のチャット bot でニューラル機械翻訳を試そう 「JoeyNMT」のカスタマイズについても解説

精度向上により、近年利用が広まっている「ニューラル機械翻訳」。その仕組みを、自分で動かしながら学んでみましょう。第 2 回はユースケースごとに「JoeyNMT」をカスタマイズする方法や、Discord のチャット bot に組み込む方法を解説します。

（2022 年 07 月 21 日）

ハイデルベルク大学の博士課程に在籍しながら、八楽という会社で「ヤラクゼン」の開発に携わっている太田です。ヤラクゼンは、AI 翻訳から翻訳文の編集、ドキュメントの共有、翻訳会社への発注までを 1 つにする翻訳プラットフォームです。

第 1 回は、機械翻訳フレームワーク「JoeyNMT」の概要、インストール方法、モデルを訓練する方法を紹介しました。今回は、JoeyNMT をカスタマイズする方法を具体的なユースケースを交えながら紹介します。

JoeyNMT は、他のフレームワークに比べてコードの行数で 9 ～ 10 分の 1、ファイル数でも 4 ～ 5 分の 1 [※1] というミニマルな実装が特長で、核となるモジュールはしっかり入っています。機械学習分野における多くのベンチマークで SOTA（State-of-the-Art）に匹敵するベンチマークスコアを出しています。またデバッグ時に stack trace をたどる際、フラットなディレクトリ構造のおかげで迷わずにエラー箇所を探し当てられるのもメリットです。

※ 1：OpenNMT-py、XNMT との比較です。詳細は「Joey NMT: A Minimalist NMT Toolkit for Novices」を参照してください。

それでは、ユースケースごとに JoeyNMT をカスタマイズする方法を見ていきましょう。

JoeyNMT でトークナイザーを変更するには

JoeyNMT はデフォルトで「subword-nmt」「sentencepiece」という 2 つのサブワードトークナイザーに対応しています。では、別のトークナイザーを利用したい場合はどうすればよいでしょうか。

トークナイザーは「joeynmt/tokenizers.py」で定義できます。例として、「fastBPE」を新しく導入してみましょう。

fastBPE は subword-nmt を c++ で実装したライブラリです。「SubwordNMTTokenizer」クラスを継承することにします。

```python
class FaseBPETokenizer(SubwordNMTTokenizer):
  def __init__(self, ...):
    try:
      # fastBPEライブラリをインポート
      import fastBPE
    except ImportError as e:
      logger.error(e)
      raise ImportError from e
    super().__init__(level, lowercase, normalize, [...], **kwargs)
    assert self.level == "bpe"
    # codes_path を取得
    self.codes: Path = Path(kwargs["codes_path"])
    assert self.codes.is_file(), f"codes file {self.codes} not found."
    # fastBPEオブジェクト
    self.bpe = fastBPE.fastBPE(self.codes)
    def __call__(self, raw_input: str, is_train: bool = False) -> List[str]:
      #  fastBPE.apply()
      tokenized = self.bpe.apply([raw_input])
      tokenized = tokenized[0].strip().split()
      # 系列の長さが指定の範囲内におさまっているか確認
      if is_train and self._filter_by_length(len(tokenized)):
        return None
      return tokenized
```

これで fastBPE でのトークナイズができるようになりました。設定ファイルで「tokenizer_type: "fastbpe"」と選択できるようにするため「_build_tokenizer()」で「FaseBPETokenizer」を呼び出せるようにします。

```python
def _build_tokenizer(cfg: Dict) -> BasicTokenizer:
  [...]
  if tokenizer_type == "sentencepiece": [...]
  elif tokenizer_type == "subword-nmt": [...]
  elif tokenizer_type == "fastbpe":
    assert "codes_path" in tokenizer_cfg
    tokenizer = FaseBPETokenizer(
      level=cfg["level"],
      lowercase=cfg.get("lowercase", False),
      normalize=cfg.get("normalize", False),
      max_length=cfg.get("max_length", -1),
      min_length=cfg.get("min_length", -1),
      **tokenizer_cfg,
    )
```

fastBPE には codes ファイルが必要ですので「codes_path」が設定ファイルで指定されていることを確認しましょう。今回導入した「FaseBPETokenizer」オブジェクトを返すようにしています。

補足

トークナイザーの「__call__()」は、データセットからインスタンスを取り出す際に呼び出されます。例えば「PlaintextDataset」では、「get_item()」内で呼び出されています。

```
def get_item(self, idx: int, lang: str, is_train: bool = None):
  [...]
  item = self.tokenizer[lang](line, is_train=is_train)
  return item
```

つまり、訓練、予測時の「for batch in data_iterator:」のイテレーションで「__getitem__()」がコールされるたびにトークナイズの関数も呼び出されることになります。これは、BPE dropout を可能にするための実装です。もし、新しく導入するトークナイザーが重い計算を必要としたり、いつも決まった値を返したりするのであれば、データ読み込み時に呼び出される「pre_process()」でトークナイズすることを検討してください（「BaseTokenizer」にある「MosesTokenizer」を利用した事前分割の実装が参考になります）。

JoeyNMT で学習率スケジューラーを変更するには

JoeyNMT は「torch.optim.lr_scheduler」に入っている「ReduceLROnPlateau」「StepLR」「ExponentialLR」の他、transformer でよく使われる「noam スケジューラー」を実装しています。別の学習率スケジューラーを使いたい場合はどうしたらよいでしょうか？

学習率スケジューラーは「joeynmt/builders.py」で定義できます。例として、Inverse Square Root スケジュールを導入してみます。

```
class BaseScheduler:
  def step(self, step):
    """学習率を更新"""
    self._step = step + 1
    rate = self._compute_rate()
    for p in self.optimizer.param_groups:
      p["lr"] = rate
    self._rate = rate
  def _compute_rate(self):
    raise NotImplementedError
```

「BaseScheduler」クラスに、そのステップでの学習率をオプティマイザのパラメーターに渡す部分が実装されています。学習率を計算する「_compute_rate()」関数をオーバーライドします。

Inverse Square Root スケジュールは、ステップ数の二乗根に反比例するように学習率を減衰させます。加えて、warmup の期間は、学習率が線形に増加するようにし、warmup の終わりで与えられた学習率に到達するよう係数（decay_rate）を調節します。

```
class WarmupInverseSquareRootScheduler(BaseScheduler):
  def __init__(
    self,
    optimizer: torch.optim.Optimizer,
    peak_rate: float = 1.0e-3,
    warmup: int = 10000,
    min_rate: float = 1.0e-5,
  ):
    super().__init__(optimizer)
    self.warmup = warmup
    self.min_rate = min_rate
    self.peak_rate = peak_rate
    self.decay_rate = peak_rate * (warmup ** 0.5)
  def _compute_rate(self):
    if step < self.warmup:
      # 線形に増加
      rate = self._step * self.peak_rate / self.warmup
    else:
      # 2乗のルートに反比例
      rate = self.decay_rate * (self._step ** -0.5)
    return max(rate, self.min_rate)
```

今回導入した Inverse Square Root スケジューラーを設定ファイルから選択できるように「build_scheduler()」を変更します。

```
def build_scheduler():
  [...]
  if scheduler_name == "plateau": [...]
  elif scheduler_name == "decaying": [...]
  elif scheduler_name == "exponential": [...]
  elif scheduler_name == "noam": [...]
  elif scheduler_name == "warmupinversesquareroot":
    scheduler = WarmupInverseSquareRootScheduler(
      optimizer=optimizer,
      peak_rate=config.get("learning_rate", 1.0e-3),
      min_rate=config.get("learning_rate_min", 1.0e-5),
      warmup=config.get("learning_rate_warmup", 10000),
    )
    scheduler_step_at = "step"
```

補足

訓練を途中で中断した際、その中断したところから再開できるよう、学習率の変数をチェックポイントに保存しています。スケジューラーで保存すべき変数が異なるため、スケジューラーごとに、どの変数を保存するのかを指定する必要があります。

Inverse Square Root スケジューラーの場合、デフォルトで保存されるステップ数とそのステップ時の学習率に加えて「warmup」「decay_rate」「peak_rate」「min_rate」を保存します。

```
1.  class WarmupInverseSquareRootScheduler(BaseScheduler):
2.    [...]
3.    def state_dict(self):
4.      super().state_dict()
5.      self._state_dict["warmup"] = self.warmup
6.      self._state_dict["peak_rate"] = self.peak_rate
7.      self._state_dict["decay_rate"] = self.decay_rate
8.      self._state_dict["min_rate"] = self.min_rate
9.      return self._state_dict
10.   def load_state_dict(self, state_dict):
11.     super().load_state_dict(state_dict)
12.     self.warmup = state_dict["warmup"]
13.     self.decay_rate = state_dict["decay_rate"]
14.     self.peak_rate = state_dict["peak_rate"]
15.     self.min_rate = state_dict["min_rate"]
```

損失関数のカスタマイズ

機械翻訳では多くの場合、交差エントロピーが損失関数として使われており、JoeyNMT でもデフォルトになっています。損失関数をカスタマイズしたい場合、どうすればよいでしょうか？

損失関数は「jorynmt/loss.py」で定義できます。第 3 回で予定している音声翻訳で必要となる「CTC Loss」と呼ばれる損失関数を、少し先取りしてここで導入してみましょう。既存の「XentLoss」クラスを継承して新しいクラス「XentCTCLoss」を作り、PyTorch で実装されている CTC Loss を呼び出します。

CTC Loss を計算するには、blank を特殊なトークンとして扱う必要があり、その blank のためのトークン ID を指定しなければなりません。新しく blank トークンを定義してもよいのですが、今回は BOS トークン「<s>」で代用することにします。

```python
class XentCTCLoss(XentLoss):
  def __init__(self,
    pad_index: int,
    bos_index: int,
    smoothing: float = 0.0,
    zero_infinity: bool = True,
    ctc_weight: float = 0.3
  ):
    super().__init__(pad_index=pad_index, smoothing=smoothing)
    self.bos_index = bos_index
    self.ctc_weight = ctc_weight
    self.ctc = nn.CTCLoss(blank=bos_index, reduction='sum')
```

「XentCTCLoss」では、すでにある交差エントロピーと CTC の重み付き和を返すようにします。

```python
class XentCTCLoss(XentLoss):
  def forward(self, log_probs, **kwargs) -> Tuple[Tensor, Tensor, Tensor]:
    # CTC Loss の計算に必要な情報がkwargsに入っていることを確認
    assert "trg" in kwargs
    assert "trg_length" in kwargs
    assert "src_mask" in kwargs
    assert "ctc_log_probs" in kwargs
    # 交差エントロピーを計算できるように変形
    log_probs_flat, targets_flat = self._reshape(log_probs, kwargs["trg"])
    # 交差エントロピーを計算
    xent_loss = self.criterion(log_probs_flat, targets_flat)
    # CTC損失を計算
    ctc_loss = self.ctc(
      kwargs["ctc_log_probs"].transpose(0, 1).contiguous(),
      targets=kwargs["trg"], # (seq_length, batch_size)
      input_lengths=kwargs["src_mask"].squeeze(1).sum(dim=1),
      target_lengths=kwargs["trg_length"]
    )
    # 交差エントロピーとCTCの重み付き和を計算
    total_loss = (1.0 - self.ctc_weight) * xent_loss + self.ctc_weight *
ctc_loss
    assert total_loss.item() >= 0.0, "loss has to be non-negative."
    return total_loss, xent_loss, ctc_loss
```

損失関数は、モデルの「forward()」で呼ばれます。「joeynmt/model.py」の該当部分を変更し「XentCTCLoss」を呼び出せるようにします。

```
class Model(nn.Module):
  def forward(self, return_type: str = None, **kwargs):
    [...]
    # 通常のデコーダー出力の他、CTCのためのレイヤーからのデコーダー出力も取得
    out, ctc_out = self._encode_decode(**kwargs)
    # デコーダー出力に対し、log_softmax（各トークンの確率）を計算
    log_probs = F.log_softmax(out, dim=-1)
    # バッチごとに損失を計算
    if isinstance(self.loss_function, XentCTCLoss):
      # CTCレイヤーからの出力についても、log_softmaxを計算
      kwargs["ctc_log_probs"] = F.log_softmax(ctc_out, dim=-1)
      # XentCTCLossのforward()を呼び出す
      total_loss, nll_loss, ctc_loss = self.loss_function(log_probs, **kwargs)
    [...]
```

バックプロパゲーションに使われるのは重み付き和である「total_loss」だけですが、それぞれの損失関数の学習曲線をプロットするため、「nll_loss」「ctc_loss」も返すようにしています。

補足

デコーダー（joeynmt/decoders.py）に、CTCLoss の計算のためのレイヤーを追加しました。

```
class TransformerDecoder(Decoder):
  def __init__(self, ...):
    [...]
    self.ctc_output_layer = nn.Linear(encoder_output_size,
vocab_size, bias=False)
  def forward(self, ...):
    [...]
    out = self.output_layer(x)
    ctc_output = self.ctc_output_layer(encoder_output)
    return out, x, att, None, ctc_output
class  Model(nn.Module):
  def _encode_decode(self, ...):
    [...]
    out, x, att, _, ctc_out = self._decode(...)
    return out, ctc_out
```

トークンペナルティで「翻訳結果の繰り返し」を防ぐ

機械翻訳の出力結果でよくあるのが、繰り返しです。例えば、配布している英日モデルを用いた wmt20 テストセットで、以下のような出力を確認しました。

> 入　力："He begged me, "grandma, let me stay, don't do this to me, don't send me back,"" Hernandez said.
> 出力：「おばあちゃん、おばあちゃん、おばあちゃん、おばあちゃん、おばあちゃん、おばあちゃん、おばあちゃん、おばあちゃん、おばあちゃん、おばあちゃん、おばあちゃん、おばあちゃん、おばあちゃん、おばあちゃん、おばあちゃん、おばあちゃん、おばあちゃん、おばあちゃん、おばあちゃん、おばあちゃん」

根本的には、なぜモデルがこのような繰り返しに高い確率を与えてしまうのかを考える必要があります。ここではその原因には踏み込まず、モデルがこのような繰り返しに高い確率を割り振ったとき、その確率を人為的に低くすることで生成させないという対症療法的な方法を考えます。

JoeyNMT は、貪欲サーチとビームサーチの 2 種類の探索を実装しています。どちらも 1 ステップずつ前から順に生成する auto-regressive、つまりそのステップまでに生成された系列 prefix を使って次のトークンを予測します。そこで、そのステップまでに生成された系列 prefix を調べ、そこにすでに出現したトークンは、次のトークンを予測する際に確率を下げることにします。

この例文でいえば「おばあちゃん、おばあちゃん」まで生成したところで次のトークンを予測する際、モデルの予測をそのまま真に受けると「おばあちゃん」が最も確率の高いトークンになってしまいます。そこで、すでにこの系列 prefix に出現している「おばあちゃん」のトークンの確率を人為的に下げ、生成されないようにブロックしようというわけです。

「search.py」の「transformer_greedy()」を見てみましょう。

```
for step in range(max_output_length):
  with torch.no_grad():
    out, _, _, _ = model(
      return_type="decode",
      trg_input=ys, # すでに生成されたprefixを渡す
      encoder_output=encoder_output,
      encoder_hidden=None,
      src_mask=src_mask,
      unroll_steps=None,
      decoder_hidden=None,
      trg_mask=trg_mask,
      return_attention=return_attention,
    )
  out = out[:, -1]  # logits
  # TODO: repetition penalty / ngram blockerをここで適用
  # もっとも確率が高いトークンを採用
  prob, next_word = torch.max(out, dim=1)
```

各ステップで最も確率が高いトークンを採用する前に、モデルの出力（out）を操作してそれまでのステップで生成されたトークンの確率を下げる repetition penalty を導入します。

```
def penalize_repetition(tokens, scores, penalty):
  scores = torch.gather(scores, 1, tokens)
  scores = torch.where(scores < 0, scores * penalty, scores / penalty)
  scores.scatter_(1, tokens, scores)
  return scores
```

ここで「penalty」には 1 より大きい正の値が入ります。例えば「penalty=2」の場合、すでに出現したトークンの確率を 2 分の 1 にせよ、という意味です。

repetition penalty は、すでに出現した全てのトークンの確率を一律に下げるように働きます。しかし、例えば日本語の助詞「は」などは複数出現する可能性があり、大きなペナルティーを課したくないときもあるでしょう。そこで、すでに出現した系列 prefix の Ngram を計算し、次に生成するトークンがその Ngram に一致する場合は確率を 0 にするという方法もあります。

仮に「[' おばあちゃん ', '、', ' おばあちゃん ', '、']」という系列 prefix があったとします。3gram の繰り返しをブロックする場合「[' おばあちゃん ', '、', ' おばあちゃん ']」と「['、', ' おばあちゃん ', '、']」の 2 つの 3gram がすでに出現していることになります。この系列 prefix の次に来るトークンが仮に「' おばあちゃん '」だった場合、直前の 2 トークンと合わせて「[' おばあちゃん ', '、', ' おばあちゃん ']」となってしまい、すでに出現した 3gram のうちの 1 つと一致してしまいます。すでに出現した 3gram と一致するようなトークン「' おばあちゃん '」を禁止トークン（banned_batch_tokens）として扱い、生成されないようにその確率を「float("-inf")」で上書きします。

```
def block_repeat_ngrams(tokens, scores, no_repeat_ngram_size, step, **kwargs):
  hyp_size = tokens.size(0)
  banned_batch_tokens = [set([]) for _ in range(hyp_size)]
  trg_tokens = tokens.cpu().tolist()
  check_end_pos = step + 2 - no_repeat_ngram_size
  offset = no_repeat_ngram_size - 1
  # 禁止トークンがあるか探します
  for hyp_idx in range(hyp_size):
    if len(trg_tokens[hyp_idx]) > no_repeat_ngram_size:
      ngram_to_check = trg_tokens[hyp_idx][-offset:]
    for i in range(1, check_end_pos):  # ignore BOS
      if ngram_to_check == trg_tokens[hyp_idx][i:i + offset]:
        banned_batch_tokens[hyp_idx].add(trg_tokens[hyp_idx][i + offset])
  # 見つかった禁止トークンのスコアに対し、-infを代入します。
  for i, banned_tokens in enumerate(banned_batch_tokens):
    scores[i, list(banned_tokens)] = float("-inf")
  return scores
```

オブジェクトをいったん CPU に移し、各シークエンス、各トークンを 1 つずつループしながら禁止トークンを探していることからも明らかなように、ngram blocker を使うと探索にかかる時間が著しく増大します。GPU の並列化によるアドバンテージを損ないたくない場合は、repetition penalty を使うことを検討してください。

ここで「repetition_penaly: 2」と設定して、もう一度同じ例文を全く同じ英日モデルからデコードしてみます。

入　力："He begged me, "grandma, let me stay, don't do this to me, don't send me back,"" Hernandez said.
出力：「おばあちゃん、泊まらせてもらって、これもしないで、送ってくれないか」とハーナンデスは言いました。

意図した通り、一度出現したトークンは生成されにくくなっています。

「no_repeat_ngram_size: 4」と設定してみます。

入　力："He begged me, "grandma, let me stay, don't do this to me, don't send me back,"" Hernandez said.
出力：「おばあちゃん、おばあちゃん、これやらない、送ってくれない」と、ヘルナンデスは言いました。

4gram より長いフレーズの繰り返しをブロックできています。

補足

上記の説明では「'おばあちゃん'」が 1 つのトークンであると仮定していました。配布しているモデルではサブワードトークンを使っており、実際はトークンレベルでの生成は以下のようになっています。

```
['「', 'お', 'ば', 'あ', 'ちゃん', '、', 'お', 'ば', 'あ', 'ちゃ', 'ん', '、', 'これ', 'や', 'ら', 'な', 'い', '、', '送', 'って', 'くれ', 'ない', '」', 'と', '、', 'ヘル', 'ナン', 'デ', 'スは', '言', 'いました', '。', '</s>']
```

2 回目の「['お', 'ば', 'あ', 'ちゃ', 'ん']」が生成されるところでは、1 回目の「['お', 'ば', 'あ', 'ちゃん']」の 4gram を避けるために「'ちゃん'」のトークンは選ばれませんでしたが、その代わりに「'ちゃ'」という別のトークンが選ばれ、その次のステップで「'ん'」というトークンが最も高い確率を得ました。その結果、各トークンを結合した出力レベルで見ると、あたかも繰り返しているように見えます。BPE によるトークナイゼーションは何通りもあり得るので、Ngram Blocker を使ってもこのような表層レベルでの繰り返しは起こりえます。

JoeyNMT でアテンション（注意度）を可視化するには

JoeyNMT には、RNN（Recurrent Neural Network）アーキテクチャからエンコーダーとデコーダー間のアテンションをプロットするオプションが用意されています。Transformer アーキテクチャでアテンションをプロットするにはどうすればよいでしょうか？

マルチヘッドトランスフォーマーでは、アテンションは 1 つではありません。全てのレイヤー、全てのヘッドが注意機構で構成されています。エンコーダー層は自己アテンションを、デコーダー層は自己アテンションとクロスアテンションを持っています。今回は、最終レイヤーのクロスアテンションを取り出し、全てのヘッドの平均を取ったものをプロットすることにします。

マルチヘッドアテンションは「joeynmt/transformer_layers.py」で定義されています。「softmax」を取った後の値を、全てのヘッドで平均して返すようにします。

```
class MultiHeadedAttention(nn.Module):
  def forward(self, ..., return_weights=False):
    [...]
    attention_weights = self.softmax(scores)
    [...]
    if return_weights:
        # すべてのヘッドの平均を取る [batch_size, query_len, key_len]
      attention_output_weights = attention_weights.view(
        batch_size, self.num_heads, query_len, key_len
      )
      avg_att = attention_output_weights.sum(dim=1) / self.num_heads
      return output, avg_att
    return output, None
```

「TransformerDecoderLayer」でクロスアテンションを計算する際に「return_weights」フラグを使えるようにします。

```
class TransformerDecoderLayer(nn.Module):
  def forward(self, ..., return_attention=False):
    [...]
    h2, att = self.src_trg_att(
      memory, memory, h1, mask=src_mask,
      return_weights=return_attention
    )
    [...]
    out = self.feed_forward(h2)
    if return_attention:
      return out, att
    return out, None
```

「joeynmt/decoders.py」のトランスフォーマーデコーダーで、最終層のときに「return_attention」フラグをTrue にしてアテンションの重みを取得するようにします。

```
class TransformerDecoder(nn.Module):
  def forward(self, ...):
    [...]
    last_layer = len(self.layers) - 1
    for i, layer in enumerate(self.layers):
      x, att = layer(
        x=x,
        memory=encoder_output,
        src_mask=src_mask,
        trg_mask=trg_mask,
        return_attention=(i == last_layer)
      )
    [...]
    return out, x, att, None
```

「joeynmt/search.py」の「transformer_greedy()」で、デコーダーから返ってきたアテンションの値を各ステップでリストに格納し、整形して出力します。

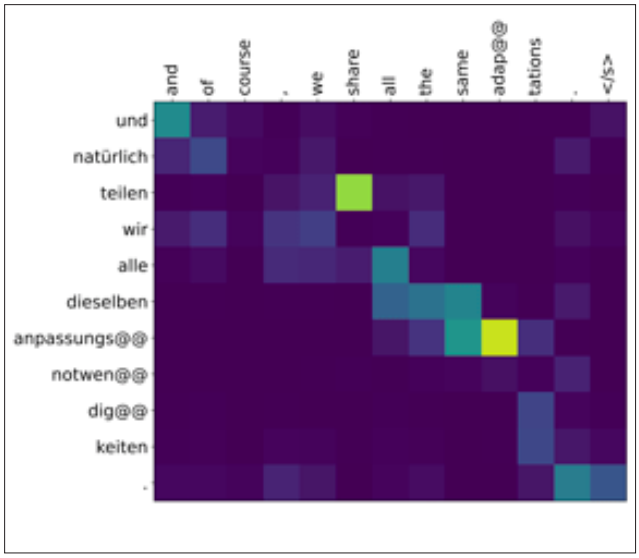
```
def transformer_greedy(...):
  [...]
  # アテンションの値を取得するかどうか
  return_attention: bool = kwargs.get("return_attention", False)
  [...]
  # アテンションの値を格納するためのプレースホルダー
  yt = ys.new_zeros((batch_size, 1, src_len), dtype=torch.float)
  [...]
  for step in range(max_output_length):
    # モデルに次のトークンの確率分布を予測させる
    with torch.no_grad():
      out, _, att, _ = model(
        return_type="decode",
        trg_input=ys,
        encoder_output=encoder_output,
        encoder_hidden=None,
        src_mask=src_mask,
        unroll_steps=None,
        decoder_hidden=None,
        trg_mask=trg_mask,
        return_attention=return_attention,
      )
    [...]
    if return_attention:
      # このステップでデコードした系列prefixの最後のトークンのアテンションの値を格納
      att = att.data[:, -1, :].unsqueeze(1)
      yt = torch.cat([yt, att], dim=1)  # (batch_size, trg_len, src_len)
  [...]
  # BOS-symbol をカット
  output = ys[:, 1:].detach().cpu().numpy()
  attention = yt[:, 1:, :].detach().cpu().numpy() if return_attention else
None
  return output, attention
```

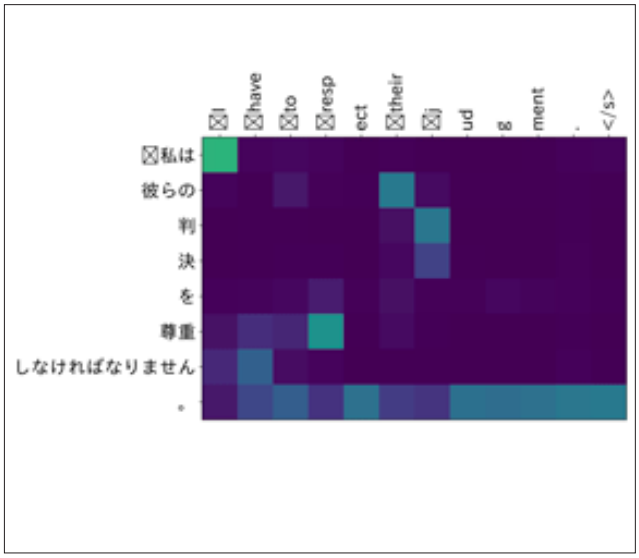
これで、トランスフォーマーでもアテンションをプロットできるようになりました。JoeyNMT をテストモードで起動してみましょう。この時に、「--save_attention」オプションを付けると上記の「transformer_greedy()」の「kwargs」に「save_attention=True」が渡されます。

```
$ python -m joeynmt test config.yaml --save_attention
```

テストセットに入っている全ての文のアテンションをプロットしますので、テストセットにはプロットしたい文だけを入れておくようにしてください。

残念ながら、配布されている日英、英日のモデルではあまりきれいな単語間アラインメントは見られませんでした。アテンションから意味のある単語アラインメントを取り出したい場合は、アラインメントのためのレイヤーを入れるなどの工夫が必要かもしれません。ある程度成功している例として、参考までに独英モデルからプロットしたアテンションもお見せしたいと思います（こちらで配布されています）





→目次に戻る

補足

matplotlib の環境によっては、日本語のフォントが文字化けしてしまうかもしれません。その場合は、日本語に対応したフォントを設定する必要があります。上記のプロットには IPAexGothic を使用しています。「joeynmt/plotting.py」を次のように書き換えてください。

```
[...]
matplotlib.use("Agg")
matplotlib.font_manager.fontManager.addfont("/path/to/ipaexg.ttf")
def plot_heatmap(...):
    [...]
    # font config
    rcParams["xtick.labelsize"] = labelsize
    rcParams["ytick.labelsize"] = labelsize
    rcParams['font.family'] = "IPAexGothic"  # support CJK
    [...]
```

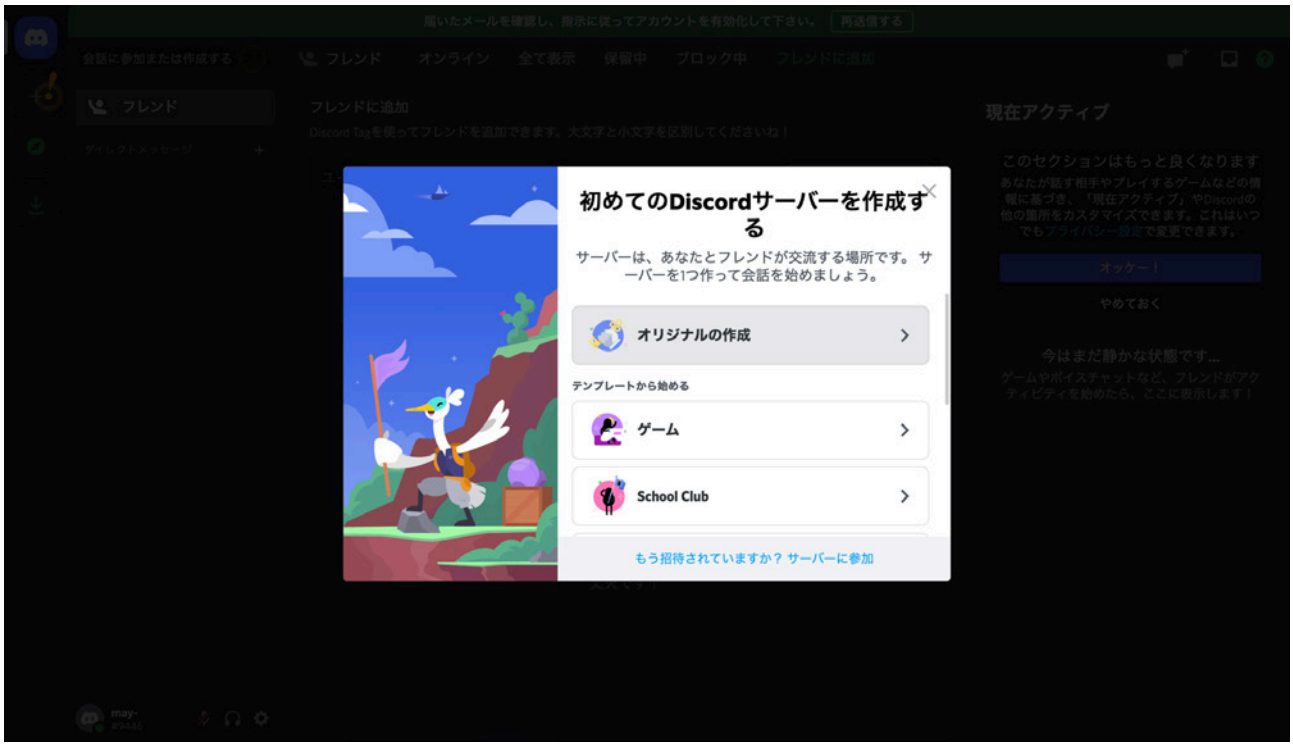
コミュニケーションツール「Discord」用のチャット bot を作ってみよう

ここからは JoeyNMT で訓練したモデルを、コミュニケーションツール「Discord」上のチャット bot として動かす方法を紹介します。

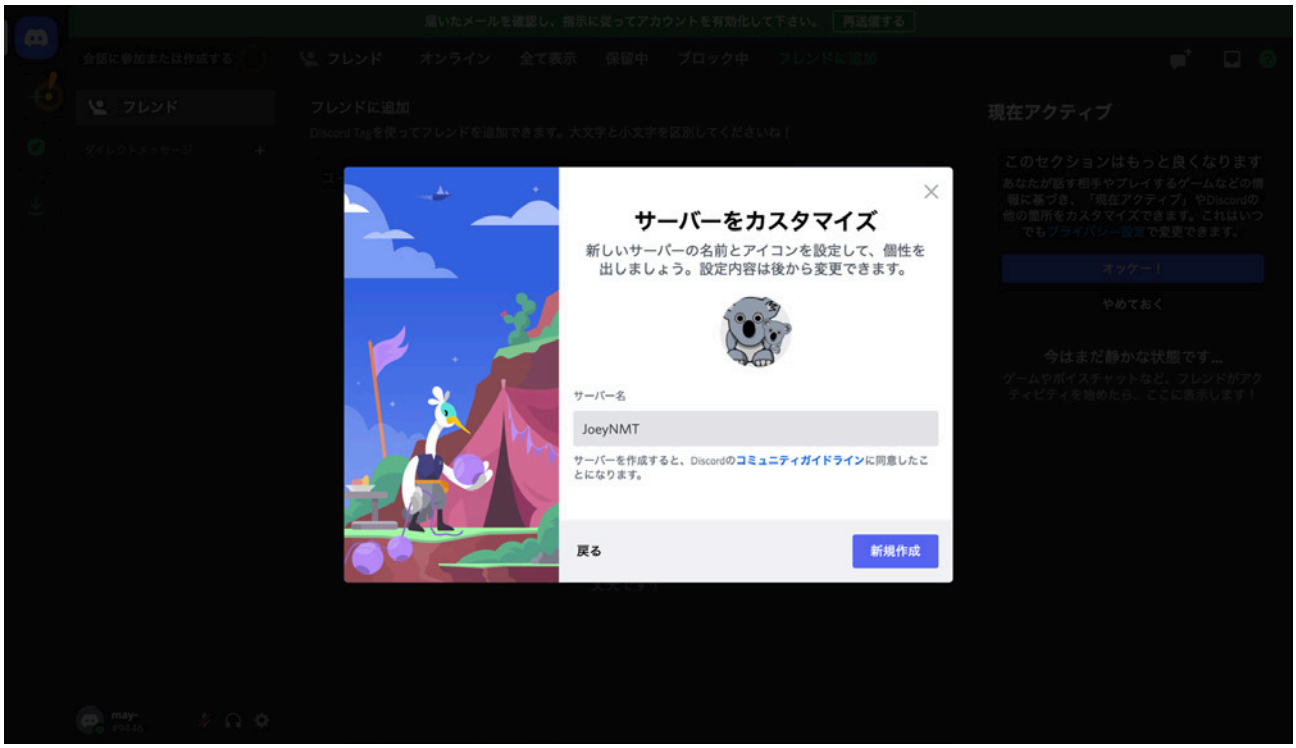
Discord アカウントとサーバの準備

Discord のアカウントがない場合は登録ページでアカウントを作成します。

続いてサーバを作成するポップアップが開きますので「オリジナルの作成」に進みます。



「JoeyNMT」という名前のサーバを作ることにします。

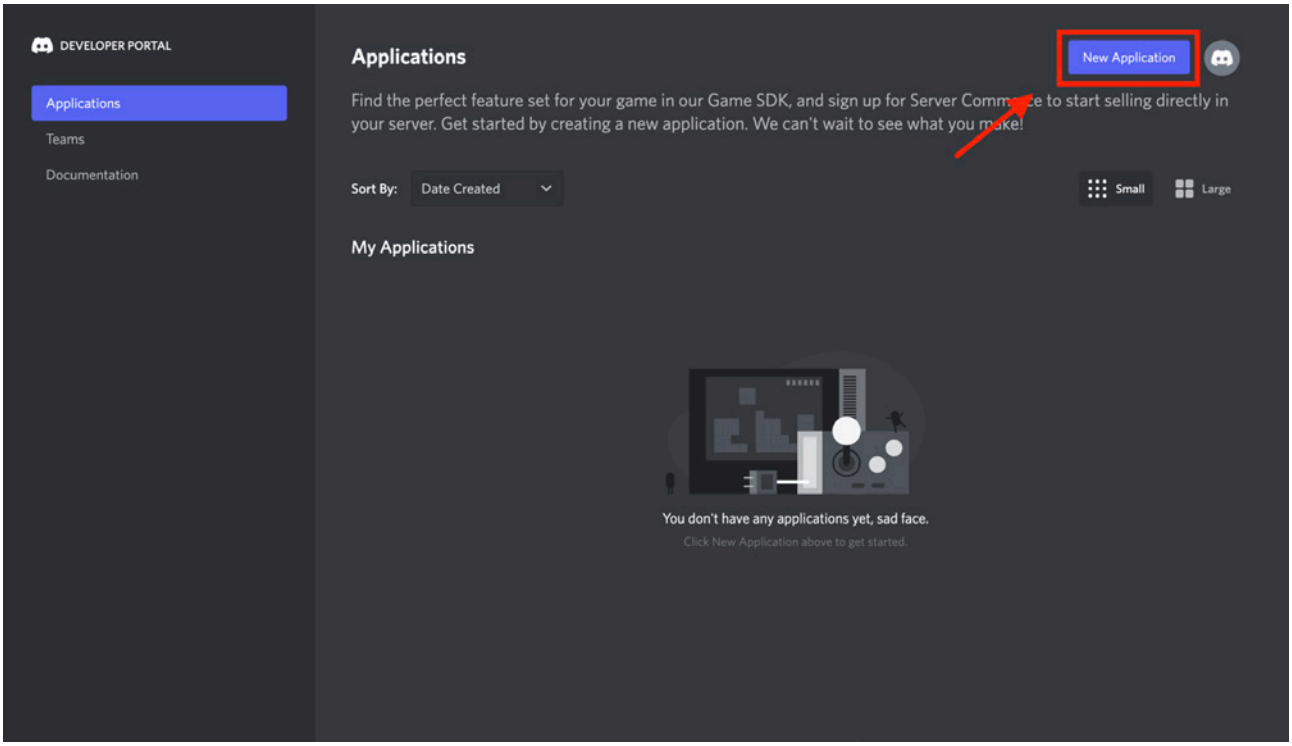


→目次に戻る

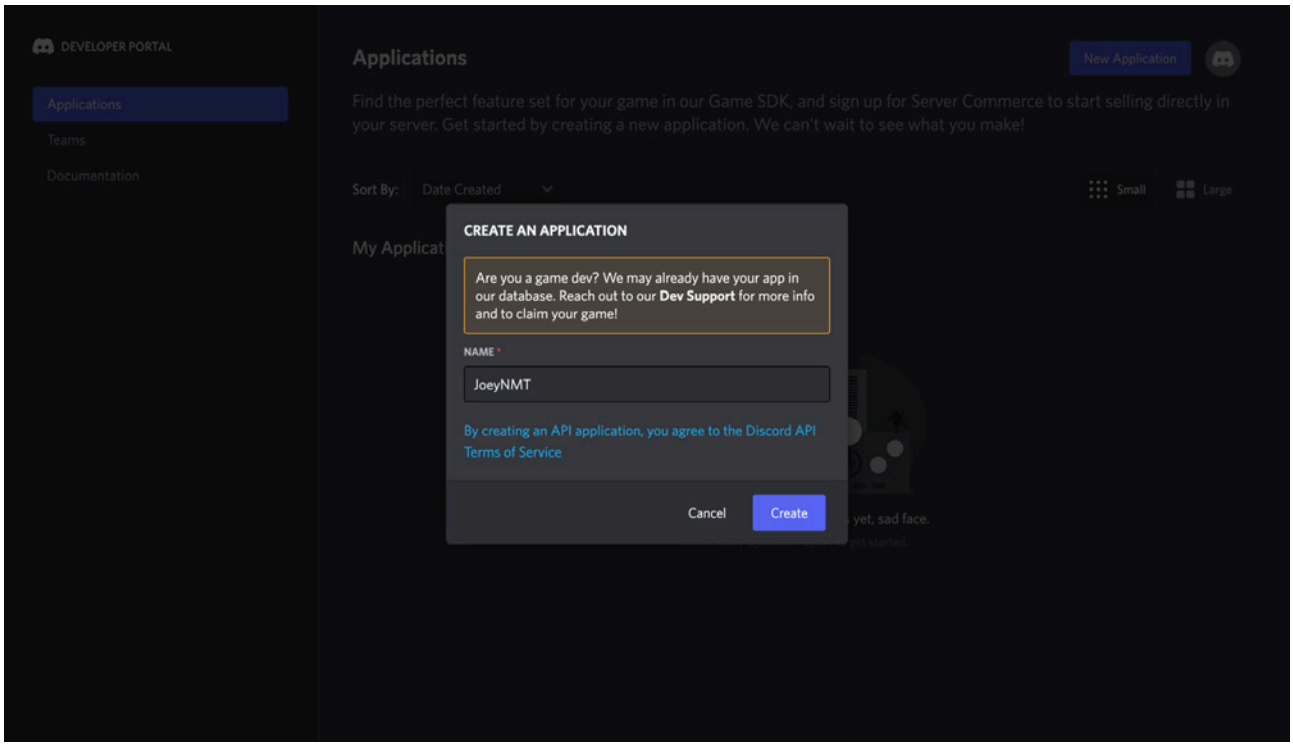
サーバを作成できました。

Bot Application の作成

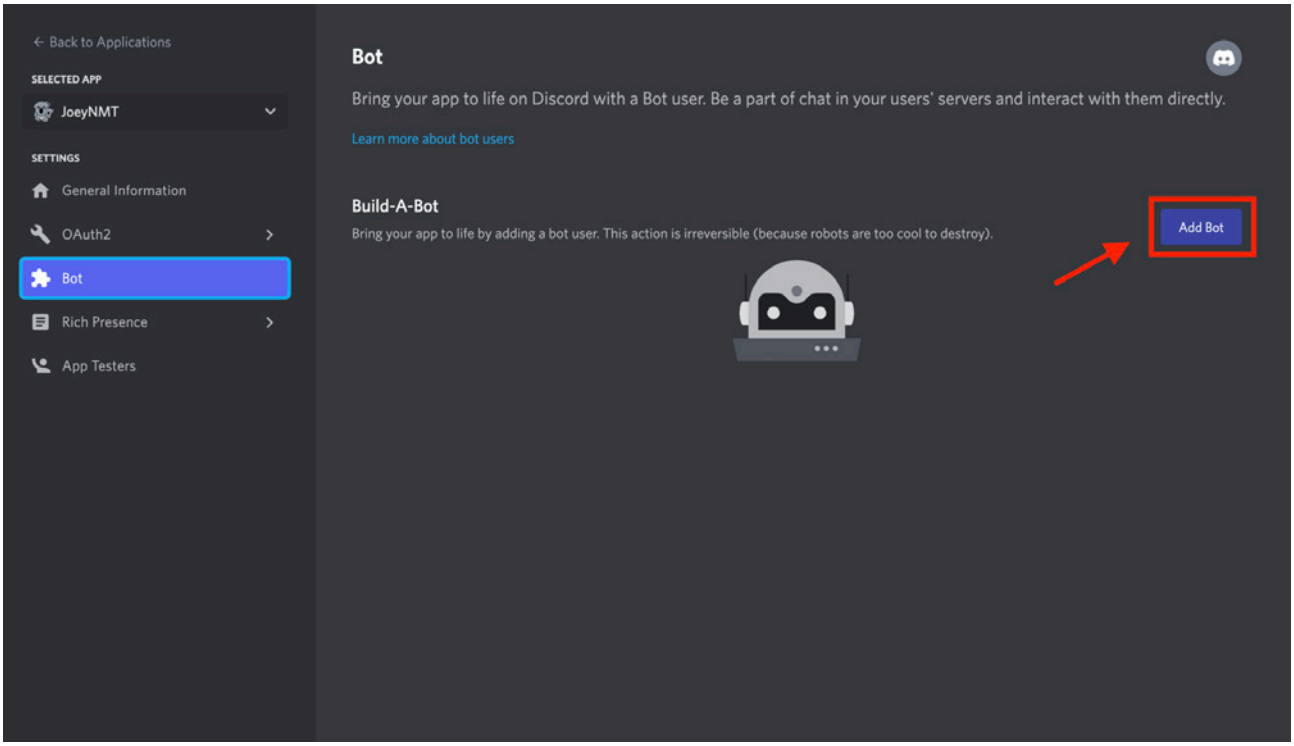
次に、Discord の開発者ポータルにアクセスし、アプリケーションを新規作成します。



アプリケーション名を指定します。

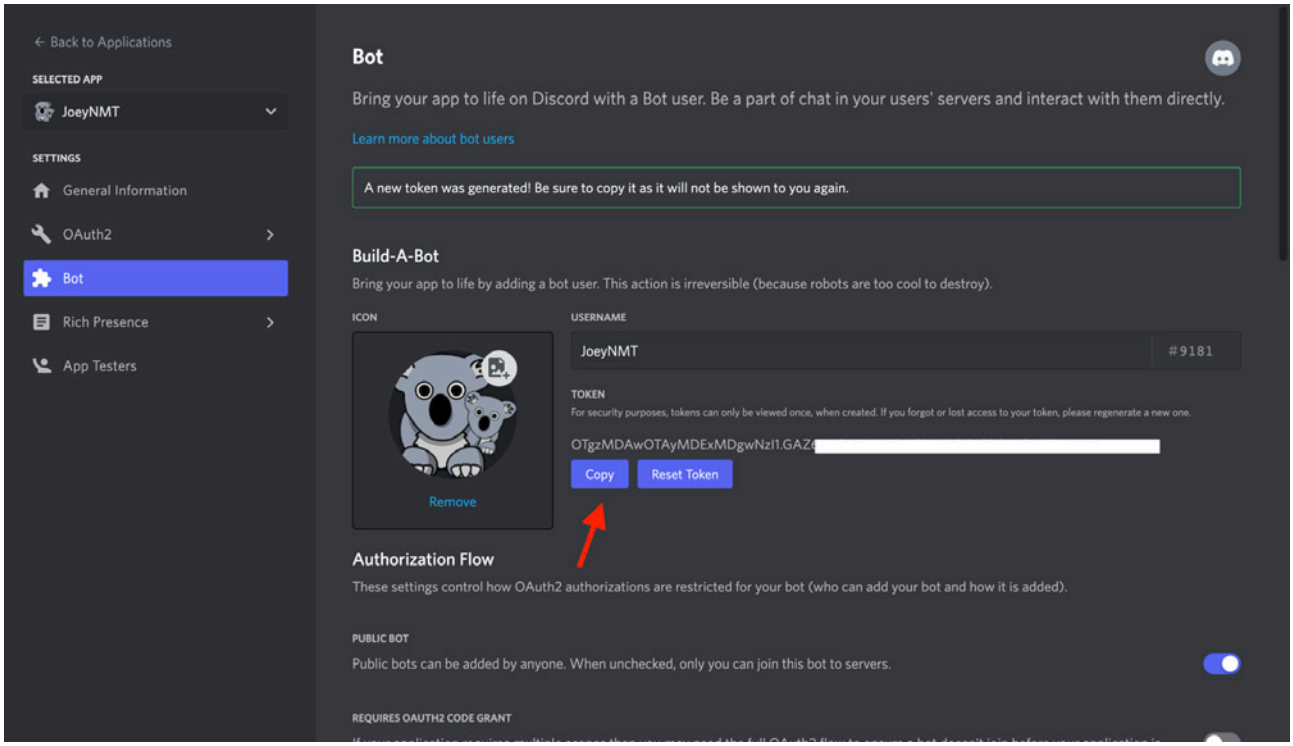


チャット bot 追加のボタンをクリックすると確認のポップアップが開くので許可します。

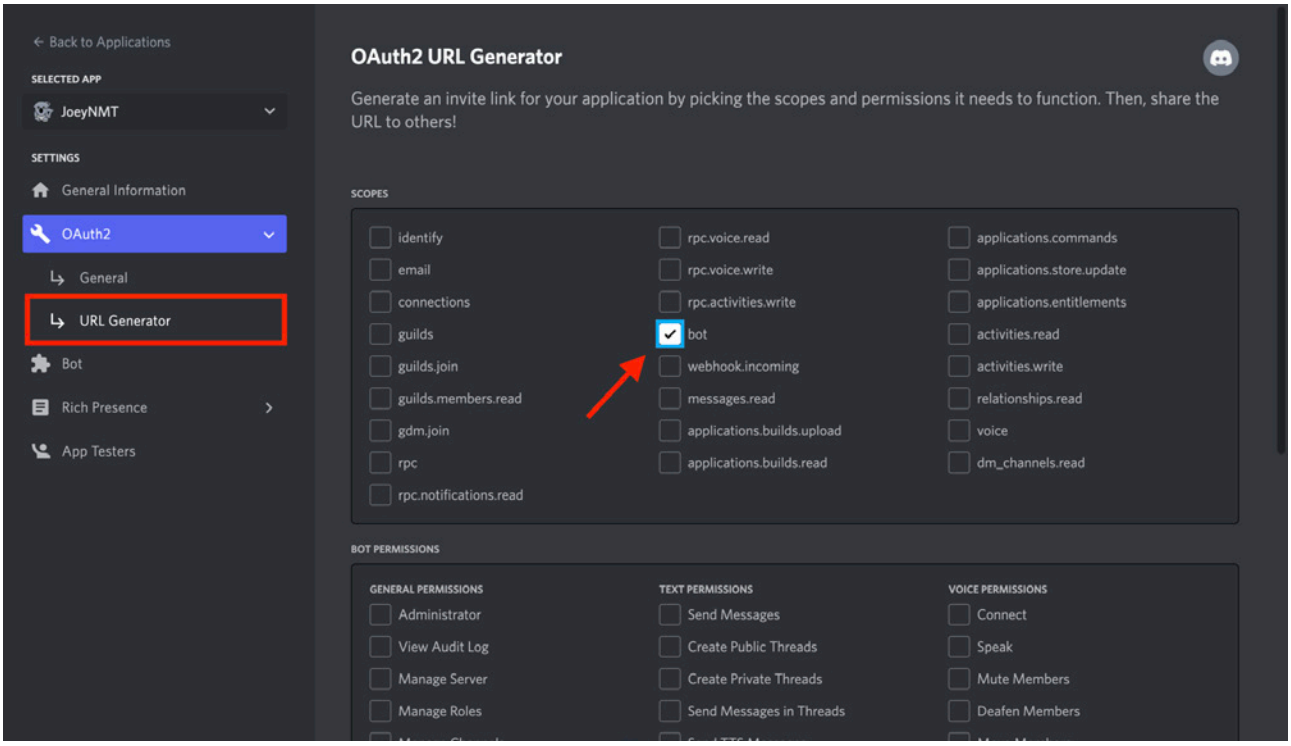


→目次に戻る

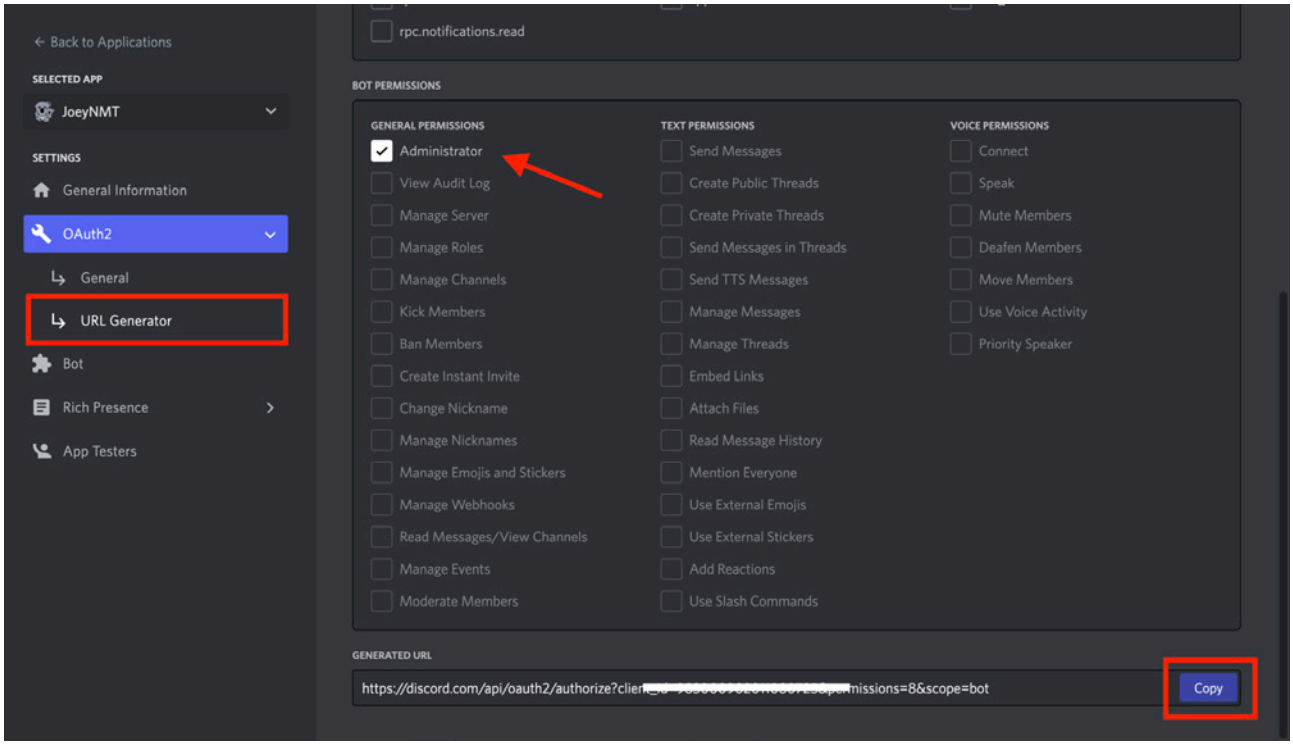
ここでチャット bot 用のアクセストークンが生成されます。後で必要になりますので控えておきます。



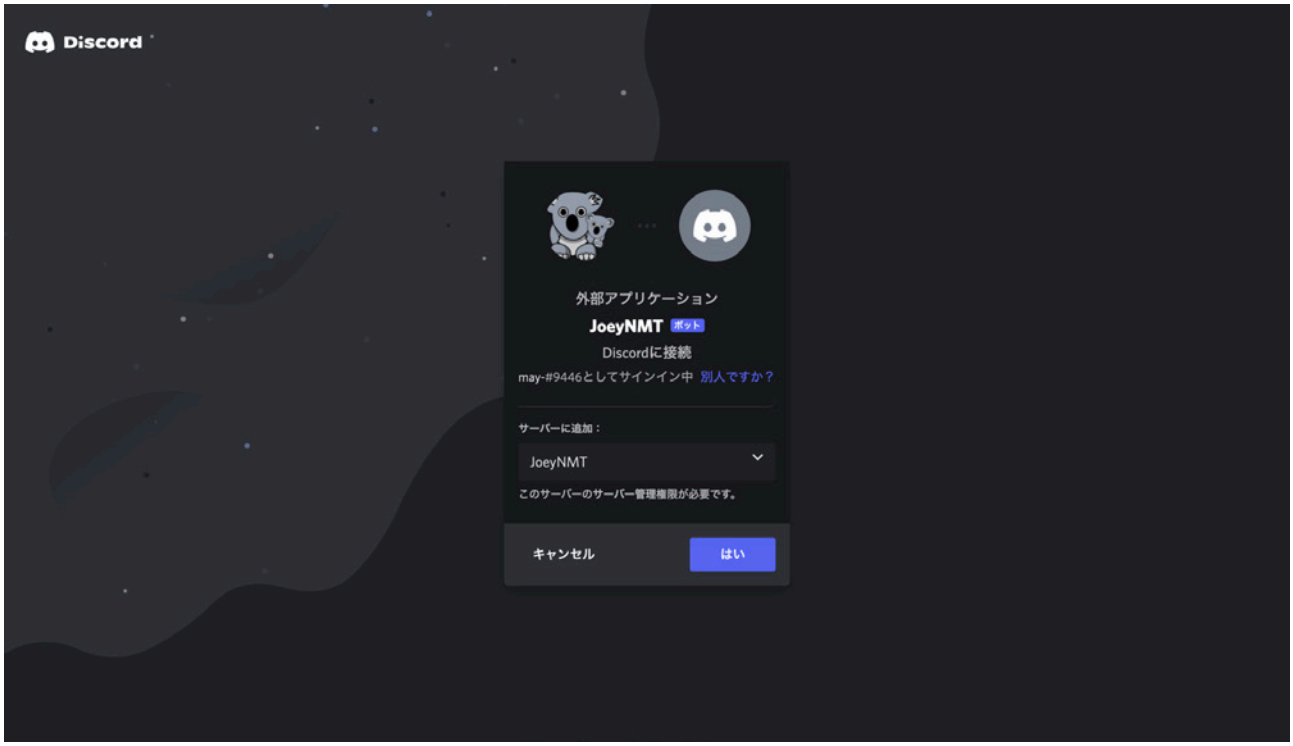
認証に必要なURLを生成します。スコープのセクションでBotを、パーミッションのセクションでAdministratorを選択します。生成された URL にブラウザからアクセスします。



→目次に戻る

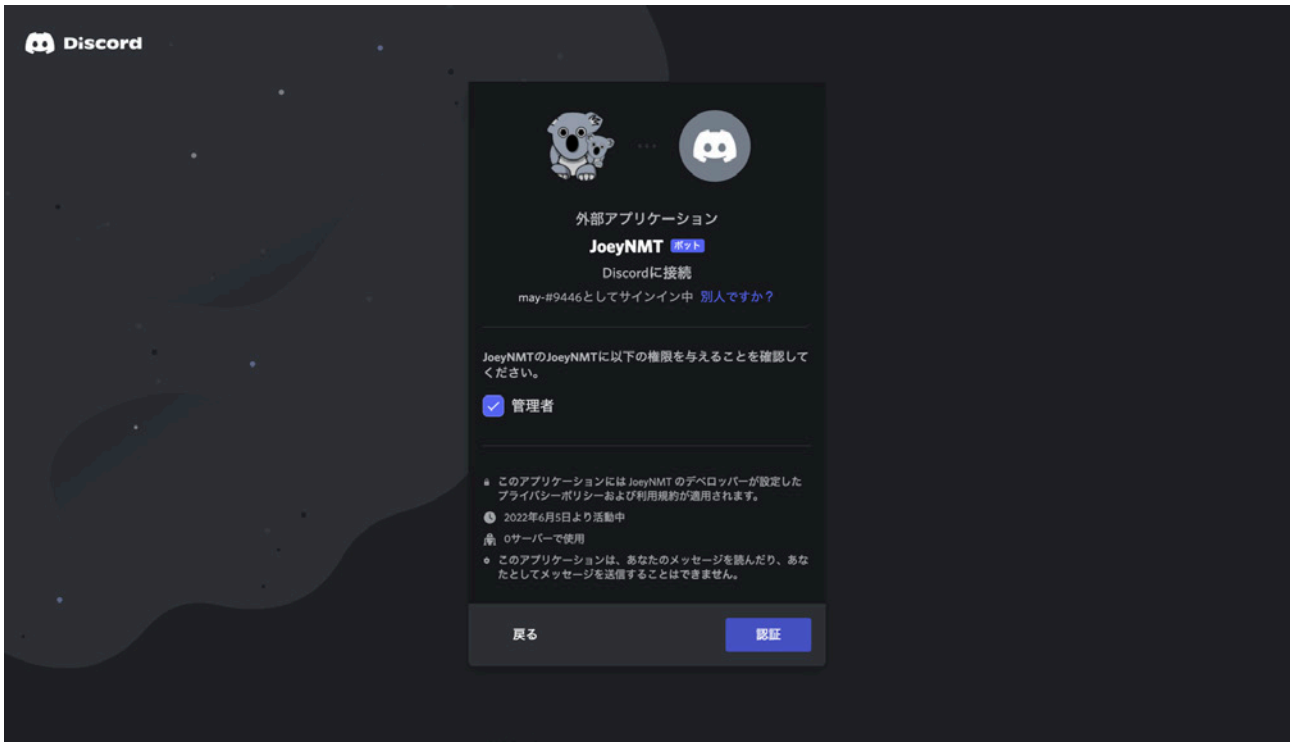


認証のポップアップが開きます。ここで、初めに作成したサーバをドロップダウンから選択します。

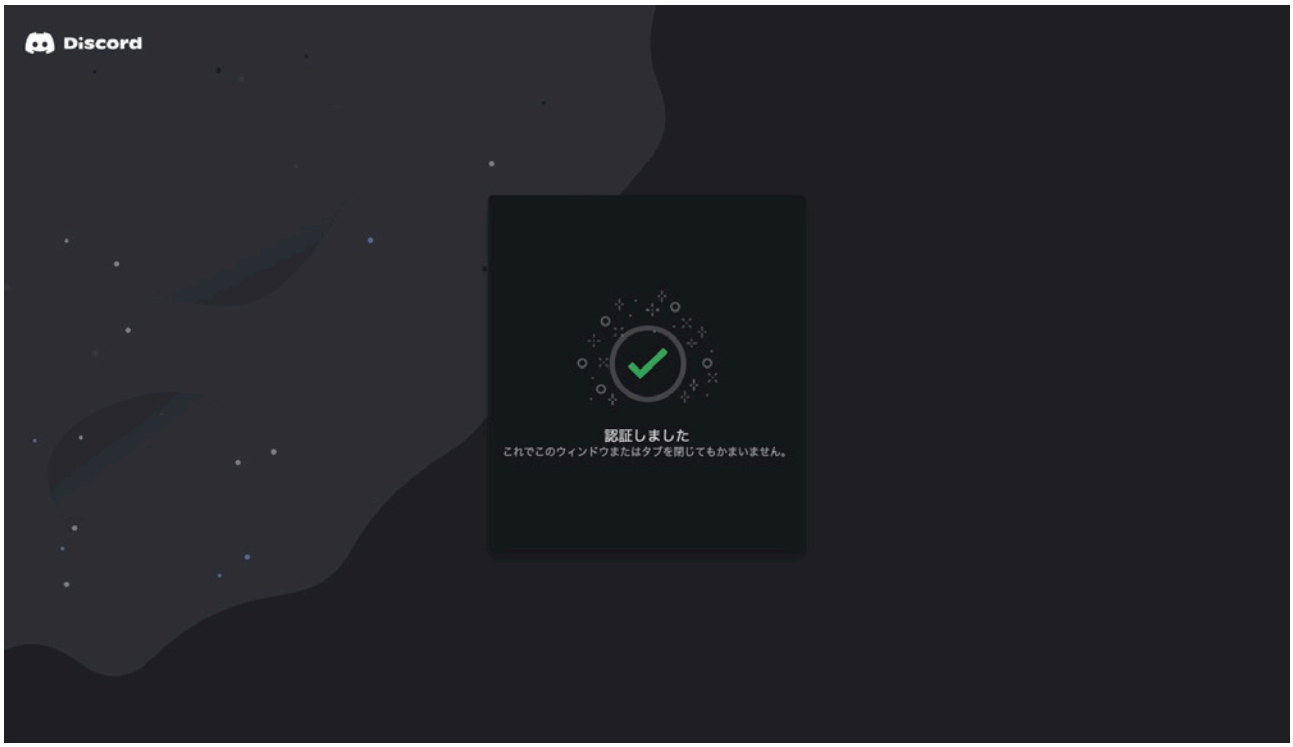


→目次に戻る

管理者権限を与えることを確認して認証します。



これで設定は一通り終わりました。



→目次に戻る

サーバに戻ると、チャット bot が追加されています。

チャット bot 用スクリプトの作成

チャット bot 用のスクリプトとして、discord.py ライブラリを使います。discord.py は pip コマンドでインストールできます。

```
$ pip install discord.py
```

ではスクリプト（discord_joey.py）を書いていきましょう。

必要なライブラリをインポートします。設定のパートで作成したチャット bot のアクセストークンをスクリプトにコピーします。

```
1. [...]
2. import discord
3. [...]
4. # access token
5. TOKEN = 'your-access-token-here'
```

チャット bot には英日、日英のモデルを利用します（学習済みモデルを配布していますのでご利用ください）。JoeyNMT のインタラクティブモードは single GPU もしくは CPU で動作します。

```
CFG_FILES = {
  'en-ja': './models/jparacrawl_enja/config.yaml',
  'ja-en': './models/jparacrawl_jaen/config.yaml'
}
DEVICE = torch.device("cuda")  # DEVICE = torch.device("cpu")
N_GPU = 1  # N_GPU = 0
```

イベントを定義します。`on_ready()` で JoeyNMT の学習済みモデルを読み込み、`on_message()` で翻訳を返すようにします。

```
client = discord.Client()
@client.event
async def on_ready():
  [...] # モデルの読み込み
@client.event
async def on_message(message):
  [...]  # メッセージが来たら翻訳して返す
```

モデルの読み込みは「joeynmt/prediction.py」の「translate()」とほぼ同じ手順で行います。

```
def load_joeynmt_model(cfg_file):
  [...]
  # ボキャブラリを取得
  src_vocab, trg_vocab = build_vocab(cfg["data"], model_dir=model_dir)
  # モデルを構成
  model = build_model(cfg["model"], src_vocab=src_vocab, trg_vocab=trg_vocab)
  # 保存されたチェックポイントからパラメータを読み込む
  ckpt = resolve_ckpt_path(None, load_model, model_dir)
  model_checkpoint = load_checkpoint(ckpt, device=device)
  model.load_state_dict(model_checkpoint["model_state"])
  if device.type == "cuda":
    model.to(device)
```

トークナイザー、入力を string から id に変換するエンコーダーを構成し、インタラクティブモードのための stream dataset を作ります。stream dataset は初めは空で、入力が来るとその都度キャッシュを更新します。

```
def load_joeynmt_model(cfg_file):
  [...]
  src_lang = cfg["data"]["src"]["lang"]
  trg_lang = cfg["data"]["trg"]["lang"]
  # トークナイザー
  tokenizer = build_tokenizer(cfg["data"])
  # エンコーダー
  sequence_encoder = {
    src_lang: partial(src_vocab.sentences_to_ids, bos=False, eos=True),
    trg_lang: None,
  }
  # インタラクティブモードのためのデータセットオブジェクト
  test_data = build_dataset(
    dataset_type="stream",
    path=None,
    src_lang=src_lang,
    trg_lang=trg_lang,
    split="test",
    tokenizer=tokenizer,
    sequence_encoder=sequence_encoder,
  )
```

幾つかのデコーディングオプションを、インタラクティブモードに対応するように書き換えます。

```
def load_joeynmt_model(cfg_file):
  [...]
  test_cfg = cfg["testing"]
  test_cfg["batch_type"] = "sentence"
  test_cfg["batch_size"] = 1
  test_cfg["n_best"] = 1
  test_cfg["return_prob"] = "none"
  test_cfg["return_attention"] = False
```

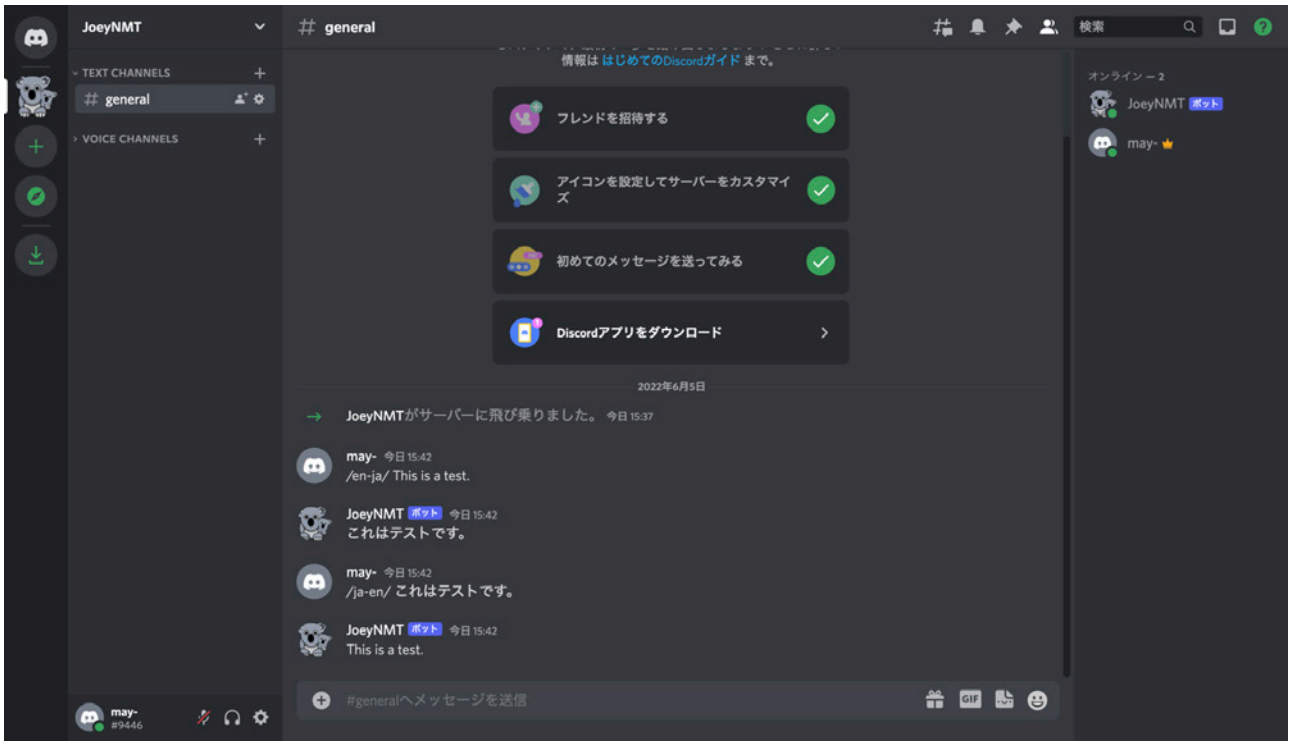
メッセージを翻訳する「translate()」では「joeynmt/prediction.py」の「predict()」を呼び出しています。メッセージには、翻訳方向を示す言語タグ「/ja-en/」または「/en-ja/」がついているものとし「get_language_tag()」でこの言語タグと本文を分けています。言語タグの設定に合わせて、翻訳結果を取得します。

```python
@client.event
async def on_message(message):
  # メッセージを言語タグと本文に分ける
  src_input = message.content.strip()
  lang_tag, src_input = get_language_tag(src_input)
  if lang_tag in CFG_FILES:
    # 翻訳を取得
    translation = translate(
      src_input,
      model_dict[lang_tag],
      data_dict[lang_tag],
      cfg_dict[lang_tag],
    )
    # 翻訳結果を返す
    await message.channel.send(translation)
```

GitHub のリポジトリに「discord_joey.py」をアップロードしてありますので参考にしてください。では、実行してみます。

```
$ python discord_joey.py
logged in.
Joey NMT: en-ja model loaded successfully.
Joey NMT: ja-en model loaded successfully.
```

モデルがロードされたことを確認したら、Discord 上でチャット bot に話し掛けてみます。言語タグを付けるのを忘れずに。



翻訳結果を返してくれています！　実行されていることが確認できました。

さいごに

今回はユースケースに合わせて JoeyNMT をカスタマイズする方法を解説しました。同様のシナリオを別のツールキットで実現しようとすると、この何倍ものコードを書き換える必要があります。JoeyNMT の場合、実行スピードを上げるための最適化などはほとんどされておらず、あまり高度なことはできないと感じられた方もいらっしゃるかもしれません。しかし、頭の中で思い描いている変更を愚直に実装できるのはとても大きなアドバンテージだと感じています。

機械翻訳を良くするアイデアはあっても、既存のフレームワークでは実装が難し過ぎると感じる方、python プログラミングや自然言語処理に取り組み始めて日が浅い初心者の方が、JoeyNMT を使って学ぶきっかけになれば幸いです。

次回は、音声入力からテキスト（文字起こし、翻訳）を生成できるように、JoeyNMT を変更する手順を解説します。

「JoeyNMT」で音声データを使った自動音声認識、音声翻訳モデルを作る

精度向上により、近年利用が広まっている「ニューラル機械翻訳」。その仕組みを、自分で動かしながら学んでみましょう。第 3 回は「JoeyNMT」を音声に対応させて、音声認識や音声翻訳のタスクをエンドツーエンドで解くモデルを構築してみましょう。

（2022 年 08 月 17 日）

ハイデルベルク大学の博士課程に在籍しながら、八楽という会社で「ヤラクゼン」の開発に携わっている太田です。ヤラクゼンは、AI 翻訳から翻訳文の編集、ドキュメントの共有、翻訳会社への発注までを 1 つにする翻訳プラットフォームです。

第 2 回は、Discord のチャット bot でニューラル機械翻訳を試す方法と「JoeyNMT」のカスタマイズ方法を紹介しました。第 3 回は「JoeyNMT」を音声に対応させて、音声認識や音声翻訳のタスクをエンドツーエンド（E2E）で解くモデルを構築する方法を紹介します。

「自動音声認識」タスクとは

自動音声認識（Automatic Speech Recognition: ASR）といえば、ある言語での音声の入力を受け付け、音声を書き起こしたテキストを返すタスクです。音声翻訳（Speech Translation: ST）は、ある言語で音声の入力を受け付ける部分は同じですが、別の言語に翻訳されたテキストを出力するタスクです。今回紹介する E2E は、入力音声の言語で書き起こしせず、ダイレクトに別の言語のテキストを生成するモデルになります。

例えば、日本語音声を入力してその日本語を書き起こすのは自動音声認識タスク、日本語音声を入力して日本語が書き起こされることなくダイレクトに英語のテキストに翻訳されるのは E2E 音声翻訳タスクに分類されます。

本記事では、音声認識と音声翻訳を合わせて、Speech-to-Text（S2T）タスクと呼ぶことにします。

インストール

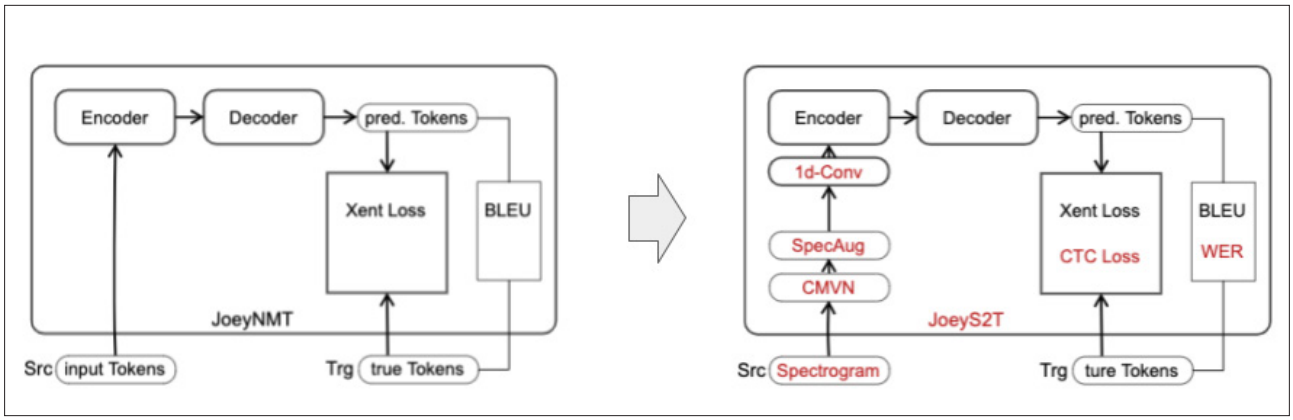
S2T のためのコードを GitHub にアップロードしました。まずは下記リポジトリからインストールしてください。

```
$ pip install git+https://github.com/may-/joeys2t.git
```

モデルの構成

テキストの機械翻訳のモデルをベースに、S2T に対応させるため、以下のモジュールを実装していきます。

1. 入出力フォーマット
2. データ拡張（CMVN、SpecAugment）
3. 畳み込みレイヤー
4. エンコーダー／デコーダー
5. CTC 損失
6. 評価（WER）



それでは一つ一つ詳しく見ていきましょう。

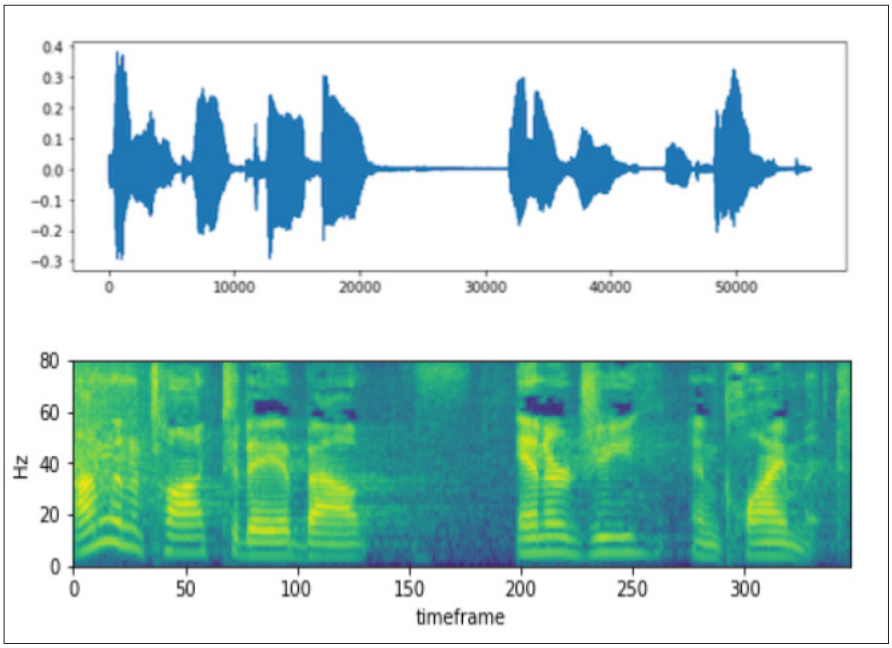
→目次に戻る

入出力フォーマット

入力：スペクトログラム

モデルに音声を入力する際、生の音声波形（waveform）は S2T タスクにおいてあまり良い特徴量とはいえません。ニューラルネットへの移行が起こる以前から、音声スペクトログラムと呼ばれる、横軸にフレーム数、縦軸に周波数をとって各フレーム、周波数におけるエネルギーの強さを 2 次元配列で表現する特徴量が音声認識タスクで広く使われてきました。

スペクトログラムの抽出方法は幾つか種類がありますが、人間の音声の周波数帯に特化した Mel Filterbank という変換を採用している論文が、E2E モデルでは主流になっています。



加えて、スペクトログラムを抽出する前の音声波形の段階で、背景ノイズの低減やスピード調整などが行われることもあります。また録音の質（背景雑音、非母語話者による録音など）にばらつきが多いときはメタデータによるフィルタリングや、女声男声の数のバランスを取るといったことも行われることがあります。今回はフィルタリングせず、データセットに入っている音声波形をそのまま使って Mel Filterbank スペクトログラムを抽出します。

本記事では割愛しますが、スペクトログラムのような音響工学に基づく特徴量抽出方法以外にも、wav2vec という特徴量ベクトルの値を深層学習で学習させる手法が 2019 年に提案され、盛んに研究されています。言語モデルがさまざまな自然言語処理タスクの事前訓練として定着していったように、今後、wav2vec が音声系のタスクの事前訓練として広く用いられるようになっていくのではと思います [※1]。

※ 1：Fine-Tune Wav2Vec2 for English ASR with Transformers

→目次に戻る

出力：テキスト

英語の音声認識ではアルファベットの文字が出力ラベルとして使われています。しかし、BPE の手法が広まって以降はどの言語の S2T でもサブワードレベルの分割が出力ラベルになるケースが増えてきました(※2)。トークナイズについては基本的にはテキストの機械翻訳のときとほとんど変わりませんが、語彙（ごい）サイズはテキストの機械翻訳よりも小さく作ることが多いようです。

※2：他にもマルチリンガルタスクへの応用で、IPA の発音記号を使ったり、音素（Phoneme）を使ったりする工夫も提案されています（https://doi.org/10.1109/ICASSP.2014.6855086 など）。https://arxiv.org/abs/2009.04707 など、文字レベルの分割とサブワードレベルの分割を比較した研究も参考にしてみてください。

音声の前処理は、テキストの前処理とは異なります。「えーと」といった言いよどみの書き起こしや、音声では発話されない句読点やかぎかっこなどの記号を取り除いたり、音声と表記にずれがある数字などを正規化したりする処理が挙げられます。特に字幕から作られたデータセットでは「(拍手)」など発話されていない描写が書き起こしテキストに入っていることが多々あります。これらを取り除いたり、トークナイズしたりする際に「(拍手)」が分割されないよう 1 つのトークンとして扱うなどの工夫も必要です。

入力データの句読点を外す前処理を施す場合は、モデルによる生成後、出力された書き起こしにも句読点を付け戻す後処理が必要になることもあります。

JoeyS2T 実装

JoeyS2T では、音声ファイルへのパスと書き起こしテキストを各行に入れた tsv ファイルを入力に取るようにしています。「src」の列は、.wav などの音声波形ファイル名、.npy のスペクトログラムファイル名、もしくは .zip ファイル名とバイトオフセットのいずれかの形式で入力音声へのパスを指定します (※3)。音声波形をスペクトログラムに変化するため「torchaudio」の sox warpper を利用しています。この変換は時間がかかるので、訓練を始める前にあらかじめ抽出しておき、numpy の 2 次元配列として保存しておくことにします。

※3：fairseq S2T の入力形式に準拠しています。

id	src	n_frames	trg
1272-128104-0	LibriSpeech/dev-clean/1272/128104/1272-128104-0000.flac	584	mister quilter is the apostle of the middle classes and we are glad to welcome his gospel
1272-128104-1	LibriSpeech/dev-clean/1272/128104/1272-128104-0001.flac	480	nor is mister quilter's manner less interesting than his matter
1272-128104-2	LibriSpeech/dev-clean/1272/128104/1272-128104-0002.flac	1247	he tells us that at this festive season of the year with christmas and roast beef looming before us similes drawn from eating and its results occur most readily to the mind

.flac ファイル名を指定した例

id	src	n_frames	trg
1688-142285-3	fbank80/1688-142285-3.npy	504	i really liked that account of himself better than anything else he said
1688-142285-4	fbank80/1688-142285-4.npy	446	his statement of having been a shop boy was the thing i liked best of all
1688-142285-5	fbank80/1688-142285-5.npy	428	you who were always accusing people of being shoppy at helstone

.npy のスペクトログラムファイル名を指定した例

id	src	n_frames	trg
1089-134686-0	fbank80.zip:56130780594:333568	1042	he hoped there would be stew for dinner turnips and carrots and bruised potatoes and fat mutton pieces to be ladled out in thick peppered flour fattened sauce
1089-134686-1	fbank80.zip:78098607294:104448	326	stuff it into you his belly counselled him
1089-134686-2	fbank80.zip:90906859847:211648	661	after early nightfall the yellow lamps would light up here and there the squalid quarter of the brothels

.zip ファイル名とオフセットを指定した例

このスペクトログラムの抽出と入力 tsv ファイルの生成をするスクリプトを準備しました。

```
$ python scripts/prepare_librispeech.py --data_root $WORK_DIR/LibriSpeech
```

LibriSpeech データセットは、英語の音声認識タスクでよく使われるベンチマークです。960 時間の音声を含む大きなデータセットで、私の環境では上記処理に丸 1 日かかりました。また、ダウンロードした生の音声波形と抽出したスペクトログラムのファイルを合わせると 160GB 程度の大きさになります。ディスク容量に注意してください。

LibriSpeech 以外にも幾つかサンプルのスクリプトが JoeyS2T に入っています。別のデータセットを使いたい場合は、このスクリプトを書き換えるところから始めてみてください。

2. データ拡張（CMVN、SpecAugment）

CMVN

Cepstral Mean Variance Normalization: CMVN は、スペクトログラムの入力値の平均を 0、分散を 1 にすることでスケールの偏りやノイズを軽減する正規化手法です。

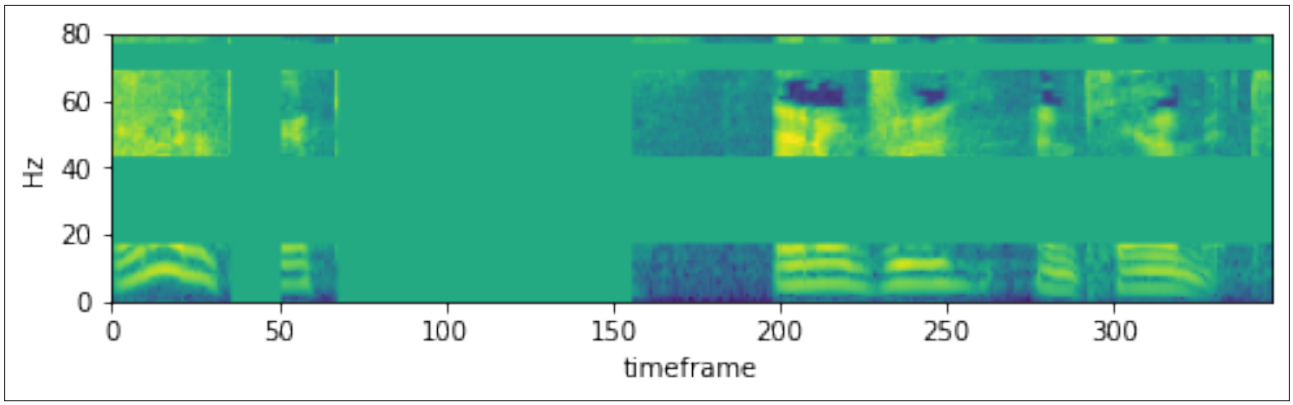
各インスタンスのスペクトログラム 2 次元配列の平均、分散を計算し、インスタンスごとに平均を引いて分散で割る手法（Utterance-CMVN）と、全てのインスタンスから平均、分散を計算し、その 1 つの値を使って正規化する手法（Global-CMVN）の 2 種類があります。JoeyS2T では前者の Utterance-CMVN を実装しています [※4]。

※ 4：Utterance-CMVN と Global-CMVN の比較は、https://arxiv.org/abs/2011.04884 での議論が参考になります。

SpecAugment

過学習を防ぐ工夫として、SpecAugment と呼ばれるマスキングを適用します。SpecAugment は、スペクトログラムの値を、時間軸方向（垂直向き）、周波数方向（水平向き）とランダムに選んでマスクアウトする手法です。マスクした場所は、そのインスタンスのスペクトログラムの平均値で埋めてしまいます。各エポックでそのインスタンスが呼び出されるたびに違うマスクが適用されるので、疑似的にデータ数をかさ増しする効果もあります [※5]。

※ 5：画像処理の分野で、入力イメージを回転したり反転したりするなど、ラベルに対して不変な変換を施してデータ数を増やすのに似ているかもしれません。



イメージとしては、通話などをしていて途中で数カ所接続が途切れる、マイクの設定などのせいで高い声の周波数帯だけ聞こえづらい、低い声の周波数帯だけゆがんでいて声が普段と違って聞こえるといった障害が起きたとしても、文脈から推測すれば話している内容が補完できるというような状況を想像してみてください。SpecAugment のマスキングは、その通信障害のようなものを疑似的に取り入れることで過学習を避け、よりロバストなモデルを作ることに貢献しているといえます。

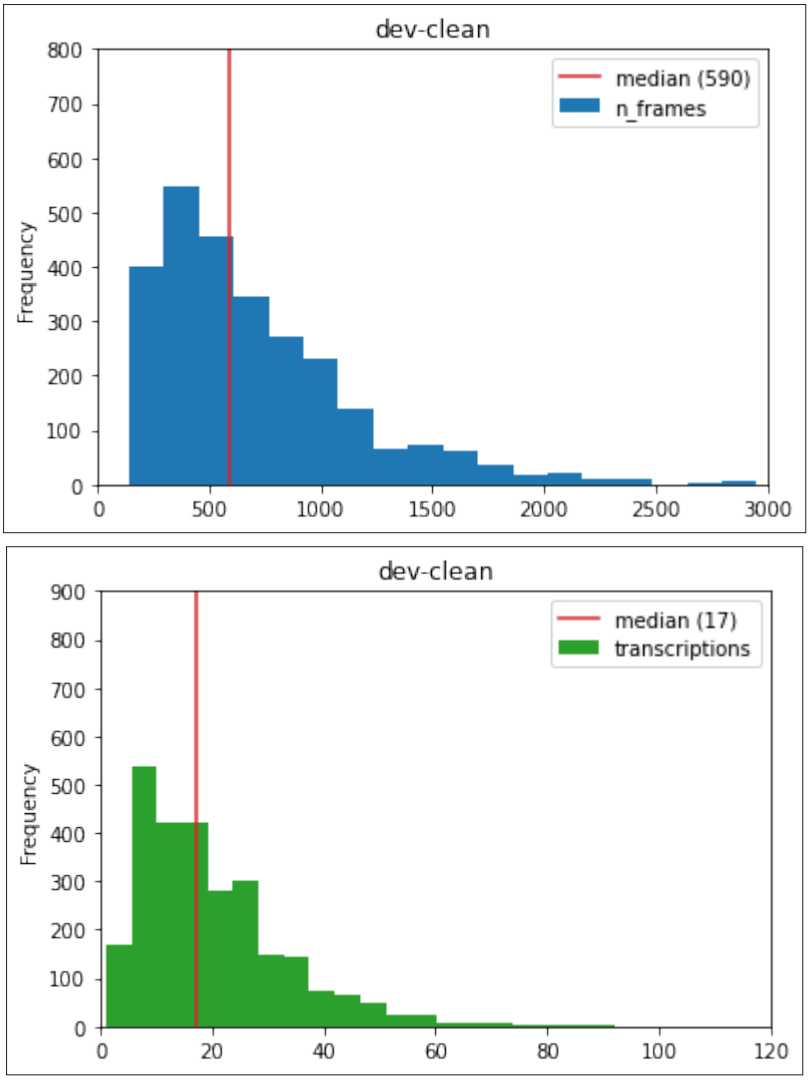
JoeyS2T 実装

CMVN、SpecAugment は「joeynmt/data_augmentation.py」で定義されています。ここで定義されたクラスを、トークナイザーが呼ばれるタイミングで適用します。JoeyNMT v2 では、バッチイテレーションの中でインスタンスを取ってくる（データセットの「__getitem__()」関数を参照）たびにトークナイザーが呼ばれるので、1 つのインスタンスでも毎回違うマスクを適用することができます。

```
# joeynmt/tokenizer.py
class SpeechProcessor:
  def __init__(self, [...], **kwargs):
    [...]
    self.specaugment = SpecAugment(**kwargs["specaugment"])
    self.cmvn = CMVN(**kwargs["cmvn"])
  def __call__(self, line: str, is_train: bool) -> np.ndarray:
    # tsvファイルの`src`列で指定されたパスから音声データを読み込む
    item = get_features(self.root_path, line)
    [...]
    # CMVN(正規化)はすべてのsplitのデータに適用
    item = self.cmvn(item)
    # SpecAugment(マスキング)は訓練データに適用
    if is_train:
      item = self.specaugment(item)
    return item
```

3. 畳み込みレイヤー

音声入力のスペクトログラムは、テキストの入力に比べて系列長が 10 倍程度長くなります。LibriSpeech の dev セットで比較すると、テキストのサブワード系列長の中央値は 17 であるのに対し、スペクトログラムフレーム数の中央値は 590 となっています。



このような長い系列をトランスフォーマーで扱うのは、計算効率でも性能の面でも困難です。そこで、入力系列を畳み込みレイヤーで短くしてからエンコーダー渡すようにします。時間軸方向に畳み込む 1d-conv をストライド 2 で n 回適用すると系列長を 2n だけ減らすことができます。

音声スペクトログラムでは、1 つのフレームが 1 つのターゲットトークンに対応しているということはあまりなく、複数のフレームが 1 つのターゲットトークンを表している場合がほとんどです。音声のこのような冗長性を考えると、畳み込みが役に立つことも感覚的に納得できるのではないでしょうか。

JoeyS2T 実装

```python
# joeynmt/encoders.py
class Conv1dSubsampler(nn.Module):
  """1次元畳み込みによるサブサンプリング"""
  def __init__(self,
    in_channels: int,
    mid_channels: int,
    out_channels: int = None,
    kernel_sizes: List[int] = (3, 3)
  ):
    super().__init__()
    self.kernel_sizes = kernel_sizes
    self.n_layers = len(kernel_sizes)
    # n_layersの数だけ1次元畳み込みレイヤーをスタックする
    self.conv_layers = nn.ModuleList(
      nn.Conv1d(
        in_channels if i == 0 else mid_channels // 2,
        mid_channels if i < self.n_layers - 1 else out_channels * 2,
        kernel_size=k,
        stride=2,
        padding=k // 2,
      ) for i, k in enumerate(kernel_sizes)
    )
    def get_out_seq_lens_tensor(self, in_seq_lens_tensor):
      # 公式ドキュメンテーションの計算式から系列長を求める
      # https://pytorch.org/docs/stable/generated/torch.nn.Conv1d.html
      out = in_seq_lens_tensor.clone()
      for k in self.kernel_sizes:
        out = ((out.float() + 2 * (k // 2) - (k - 1) - 1) / 2 +
1).floor().long()
      return out
    def forward(self, src_tokens, src_lengths):
      # DataParallelによるバッチの割り当て後に最大系列長を計算し直す
      max_len = torch.max(src_lengths).item()
      assert max_len > 0, "empty batch!"
      if src_tokens.size(1) != max_len:
        src_tokens = src_tokens[:, :max_len, :]
      assert src_tokens.size(1) == max_len, (src_tokens.size(), max_len,
src_lengths)
      _, in_seq_len, _ = src_tokens.size()  # -> B x T x (C x D)
      x = src_tokens.transpose(1, 2).contiguous()  # -> B x (C x D) x T
      # 畳み込みレイヤーの後、非線形アクティベーション(glu)を適用
      for conv in self.conv_layers:
        x = conv(x)
        x = nn.functional.glu(x, dim=1)
        _, _, out_seq_len = x.size()
        x = x.transpose(1, 2).contiguous()  # -> B x T x (C x D)
      # 畳み込み後の系列長を計算。
      out_seq_lens = self.get_out_seq_lens_tensor(src_lengths)
      assert x.size(1) == torch.max(out_seq_lens).item(), \
        (x.size(), in_seq_len, out_seq_len, out_seq_lens)
      return x, out_seq_lens
```

畳み込みレイヤーを適用すると系列長が短くなるので、それに合わせてパディングマスクも計算し直す必要があります。PyTorch DataParallel でバッチが複数 GPU に割り当てられると、バッチの元の最大系列長が分からなくなります。そのため、複数 GPU に割り当てられる前に最大長を保持しておき、その最大長に合わせて全ての GPU のバッチのパディングを計算し直しています。

```
# joeynmt/encoders.py
class TransformerEncoder(Encoder):
  [...]
  def forward(self, embed_src, src_length, mask, **kwargs):
    # 1次元畳み込みによるサブサンプリング
    if self.subsample:
      embed_src, src_length = self.subsampler(embed_src, src_length)
    # パディングマスクを計算し直す
    if mask is None:
      mask = lengths_to_padding_mask(src_length).unsqueeze(1)
    [...]
    return x, None, mask
```

損失関数に渡すパディングマスクも、バッチオブジェクトのメンバーである「batch.src_mask」ではなく Encoder 内で計算し直した「src_mask」に置き換える必要があります。

```
# joeynmt/model.py
class Model(nn.Module):
  [...]
  def forward(self, return_type, **kwargs):
    if return_type == "loss":
      # エンコーダー・デコーダーからの出力を受け取る
      out, ctc_out, src_mask = self._encode_decode(**kwargs)
      # 交差エントロピーの計算につかう各トークンの対数確率を計算
      log_probs = F.log_softmax(out, dim=-1)
      # src_maskを計算し直したもので書き換える
      kwargs["src_mask"] = src_mask
      # CTC損失の計算につかうフレームごとの対数確率を計算
      kwargs["ctc_log_probs"] = F.log_softmax(ctc_out, dim=-1)
      # 損失関数からの返り値（バッチ内で和をとったもの）
      batch_loss = self.loss_function(log_probs, **kwargs)
      [...]
```

4. エンコーダー／デコーダー

エンコーダー／デコーダーの構成は、テキストの機械翻訳とほぼ同じです。コードは（畳み込みレイヤーに関する変更を除けば）全く同じものを使いますが、設定で少し気を付けるべきポイントを挙げます。

深いエンコーダー

テキストの機械翻訳では、同じレイヤー数のエンコーダーとデコーダーを使うことが多いでしょう。S2T タスクの場合、エンコーダーをデコーダーより深くし、より複雑な学習により多くのパラメーターを割り当てると性能が上がるという報告があります [(※6)]。

※6：https://arxiv.org/abs/1904.13377 など

転移学習

音声翻訳の場合に使われるテクニックとして、エンコーダーのパラメーターを ASR 事前学習モデルで、デコーダーのパラメーターを MT 事前学習モデルで初期化するという方法があります[(※7)]。スクラッチから音声翻訳モデルを訓練するのは不安定になったり、収束させるのに時間がかかりすぎたりすることがあります。事前学習モデルからパラメーターの値を転移させることで訓練を安定させたり、処理時間を短縮させたりすることを狙っています。

※7：https://arxiv.org/abs/1911.08870

JoeyS2T 実装

チェックポイントを読み込む際、レイヤーの名前を確認して、あるチェックポイントからはエンコーダーのパラメーターのみを、別のチェックポイントからはデコーダーのパラメーターのみを読み込むようにします。

```
1.  # joeynmt/training.py
2.  class TrainManager:
3.    [...]
4.    def __init__(self, model: Model, cfg: dict) -> None:
5.      [...]
6.      for layer_name, load_path in [("encoder", load_encoder), ("decoder",
    load_decoder)]:
7.        if load_path is not None:
8.          self.init_layers(path=load_path, layer=layer_name)
9.    def init_layers(self, path: Path, layer: str) -> None:
10.     layer_state_dict = OrderedDict()
11.     logger.info("Loading %s laysers from %s", layer, path)
12.     ckpt = load_checkpoint(path=path, device=self.device)
13.     for k, v in ckpt["model_state"].items():
14.       if k.startswith(layer):
15.         layer_state_dict[k] = v
16.     self.model.load_state_dict(layer_state_dict, strict=False)
```

Source 側の単語埋め込みはスキップしてスペクトログラムを直接エンコーダーに送るため「Model.src_embed」には「torch.nn.Embeding」オブジェクトではなく「torch.nn.Identity」オブジェクトを入れておきます。

```python
# joeynmt/model.py
def build_model(cfg, src_vocab, trg_vocab):
  [...]
  src_embed = Embeddings(
    **enc_cfg["embeddings"],
    vocab_size=len(src_vocab),
    padding_idx=src_vocab.pad_index
  ) if task ==  "MT" else None
  [...]
  model = Model(
    encoder=encoder,
    decoder=decoder,
    src_embed=src_embed if task == "MT" else nn.Identity(),
    trg_embed=trg_embed,
    src_vocab=src_vocab,
    trg_vocab=trg_vocab,
    task=task)
  [...]
  return model
```

5. CTC 損失

多くの機械学習タスク同様、テキストの機械翻訳でも、交差エントロピーを損失関数として採用することがほとんどです。E2E 音声認識でも、交差エントロピーを最小化する目的関数を採用したトランスフォーマー型のモデルがより高い精度を達成してきました。この交差エントロピーに加えて「Connectionist Temporal Classification: CTC」と呼ばれる長い系列をうまく扱う工夫を損失関数に取り込む手法が提案されています[※8]。

※8：https://doi.org/10.1109/ICASSP.2017.7953075

CTC 損失は、手書き文字認識や音声認識のような、入力と出力の長さが大きく異なるようなタスクで使われています。これらのタスクでは、正解のテキスト系列を出力させることが目的であって、各出力ラベルがどの入力フレームに対応するかはあまり重視されていません。

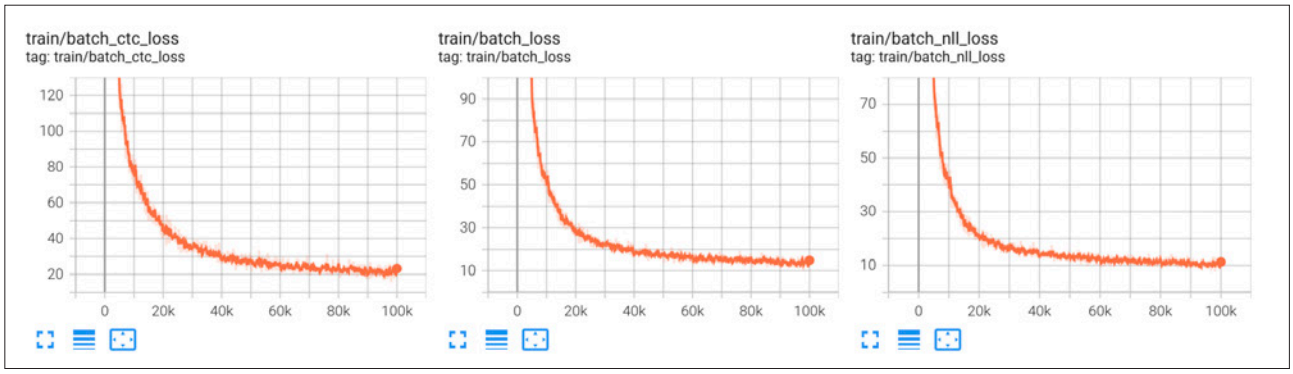
例えば、仮に 100 フレームの「こんにちは」という音声入力があるとします。この音声が、20 フレームずつ均等に「こ」「ん」「に」「ち」「は」というラベルにそれぞれ対応していたとしても、最初の 60 フレームが「こ」に対応していて残りの 40 フレームが「んにちは」に対応していたとしても、どちらの場合でも「こんにちは」という正解ラベルを出力できれば、音声認識の目的は達成できると考えます。

JoeyS2T では、交差エントロピーと CTC の両方の損失を最小化する目的関数を採用しています[※9]。

$$\mathcal{L}_{\text{total}} = (1-\lambda)\mathcal{L}_{\text{Xent}} + \lambda\mathcal{L}_{\text{CTC}}$$

※9:https://doi.org/10.1109/JSTSP.2017.2763455

ここで、λはハイパーパラメーターで、設定ファイルの「ctc_weight」に指定します。一般に、CTC 損失が交差エントロピー損失よりも大きな値になることが多いです。両方の損失がともに全体の損失に貢献するようなλの値を、予備実験を行って決めるのがよいかもしれません。例えば、以下のような学習曲線の場合、10k ステップ時点で交差エントロピーの損失がおよそ 40、CTC 損失がおよそ 80 ですので、λ＝ 0.3 くらいに設定すると（1-0.3）×40 ＝ 28、0.3×80 ＝ 24 となり、ちょうど両方の損失のバランスが取れそうです。



JoeyS2T 実装

損失関数のコードをカスタマイズする方法については第 2 回の記事を参照してください。

6. 評価（WER）

音声認識の評価には「Word Error Rate: WER」という評価尺度がよく使われます。モデルの出力と正解ラベルの間の編集距離（edit distance）に基づく指標で、小さい値ほどモデルの出力と正解ラベルの間に違いが少ない、つまり精度が良いことを示しています。

JoeyS2T 実装

JoeyS2T では、Cython で実装された editdistance パッケージをインポートしています。

モデルが出力した one-hot-encoding のトークンのリストを 1 つの文字列に戻した後、sacrebleu のトークナイザーとともに「wer()」関数に渡しています。設定ファイルの「sacrebleu_cfg」の項目で、トークナイザーの種類を指定することができます。

```python
# joeynmt/metrics.py
import editdistance
def wer(
  hypotheses: List[str],
  references: List[str],
  tokenizer: Callable,
) -> float:
  numerator = 0.0  # 分子
  denominator = 0.0  # 分母
  # コーパスレベルで編集距離のカウントの総数を累積
  for hyp, ref in zip(hypotheses, references):
    # 単語分割
    hyp = tokenizer(hyp)
    ref = tokenizer(ref)
    # utteranceごとにモデル出力と正解ラベル間の編集距離を計算
    numerator += editdistance.eval(hyp, ref)
    denominator += len(ref)
  return (numerator / denominator) * 100 if denominator else 0.0
```

※ sclite での計算結果と一致することを確認しています。

【補足】参考文献ガイド

- Speech Translation Tutorial：音声翻訳で参考文献を何か一つ挙げるとしたら、断然この「EACL 2021」の音声翻訳チュートリアルをおすすめします。講義ビデオ、スライド、文献リスト、データ、ツールなど多岐にわたる情報がまとまっています。カスケード型から最新の End to End まで広く学ぶことができます
- Speech Processing for Machine Learning：スペクトログラムについてより詳しく知りたい方はこの記事を読んでみてください
- Sequence ModelingWith CTC：CTC 損失関数について、分かりやすい図を用いて視覚的に説明してくれている記事です。前向き後ろ向きアルゴリズムについては日本語で読めるこちらの記事をおすすめします
- An Illustrated Tour of Wav2vec 2.0：近年研究が進んでいる「wav2vec2.0」の解説記事です。後続論文の「wav2vec-U」は、音声認識のチュートリアルとしても読むことができると思います
- Speech Translation and the End-to-End Promise: Taking Stock of Where We Are：音声翻訳の概要をつかむには、このような包括的なサーベイ論文が助けになるかもしれません
- Python で学ぶ音声認識 機械学習実践シリーズ：日本語の書籍では、基礎知識から数式の解説、実装まで広く扱っているこちらの本を参考文献として挙げたいと思います

モデルの訓練

「入出力フォーマット」のセクションで言及した LibriSpeech コーパスから、clean100 カテゴリーのデータを使ってモデルを訓練します [※10]。

※ 10 : LibriSpeech はデータサイズが大きいので Google Colab では扱うのが難しいかもしれません。Google Colab で動かしてみたい場合は、小さいデータを扱った notebook を参照してください。

コンフィギュレーションファイルで、Speech-to-Text タスクの設定をします。「src」の「min_length」は、1d-Conv で圧縮するカーネルサイズより長くなるように設定してください。

```
data:
  task: "S2T"             # Speech-to-Textタスク
  train: "path/to/LibriSpeech/joey_train-clean-100"   # 訓練データ
  dev: "path/to/LibriSpeech/joey_dev-clean"           # 開発データ
  test: "path/to/LibriSpeech/joey_test-clean"         # テストデータ
  dataset_type: "speech" # データセットタイプは"speech"に設定
  src:
    lang: "en"            # 言語タグ
    level: "frame"        # 入力レベルは"frame"に設定
    num_freq: 80          # スペクトログラムの周波数
    min_length: 10        # スペクトログラムの最小フレーム数
    max_length: 6000      # スペクトログラムの最大フレーム数
    tokenizer_type: "speech" # トークナイザータイプは"speech"に設定
    tokenizer_cfg:
      specaugment:        # SpecAugmentのパラメーター
        freq_mask_n: 2
        freq_mask_f: 27
        time_mask_n: 2
        time_mask_t: 100
        time_mask_p: 1.0
      cmvn:               # CMVNのパラメーター
        norm_means: True
        norm_vars: True
        before: True
    trg:
      lang: "en"
      level: "bpe"
      lowercase: True
      max_length: 512
      voc_min_freq: 1
      voc_limit: 5000
      voc_file: "path/to/LibriSpeech/spm_train-clean-
100_unigram5000.vocab.txt"
      tokenizer_type: "sentencepiece"
      tokenizer_cfg:
        model_file: "path/to/LibriSpeech/spm_train-clean-
100_unigram5000.model"
        pretokenizer: "none"
testing:
```

```
    n_best: 1
    beam_size: 20
    beam_alpha: 1.0
    batch_size: 10000
    batch_type: "token"
    max_output_length: 100  # 書き起こしテキストの最大出力長
    eval_metrics: ["wer"]    # 評価尺度
    sacrebleu_cfg:
      tokenize: "13a"    # 評価する時に使うトークナイザー
training:
    #load_model: "models/librispeech100h/best.ckpt"
    #load_encoder: "models/ASR/best.ckpt"  # 音声翻訳の際、エンコーダーをASR事前学習モデルのパラメーターで初期化する
    #load_decoder: "models/MT/best.ckpt"    # 音声翻訳の際、デコーダーをMT事前学習モデルのパラメーターで初期化する
    reset_best_ckpt: False
    reset_scheduler: False
    reset_optimizer: False
    reset_iter_state: False
    random_seed: 321
    optimizer: "adam"
    adam_betas: [0.9, 0.98]
    scheduling: "warmupinversesquareroot"
    learning_rate: 2.0e-3
    learning_rate_min: 1.0e-6
    learning_rate_warmup: 10000
    clip_grad_norm: 10.0
    weight_decay: 0.
    batch_size: 20000
    batch_type: "token"
    batch_multiplier: 4
    normalization: "batch"
    epochs: 300
    updates: 100000
    validation_freq: 1000
    logging_freq: 100
    early_stopping_metric: "wer"
    model_dir: "models/librispeech100h"
    overwrite: False
    shuffle: True
    use_cuda: True
    print_valid_sents: [0, 1, 2]
    keep_best_ckpts: 10
    label_smoothing: 0.1
    loss: "crossentropy-ctc" # 交差エントロピーとCTCのジョイントオブジェクティブ
    ctc_weight: 0.3          # CTC損失の係数
model:
    initializer: "xavier"
    init_gain: 1.0
    bias_initializer: "zeros"
    embed_initializer: "xavier"
    embed_init_gain: 1.0
```

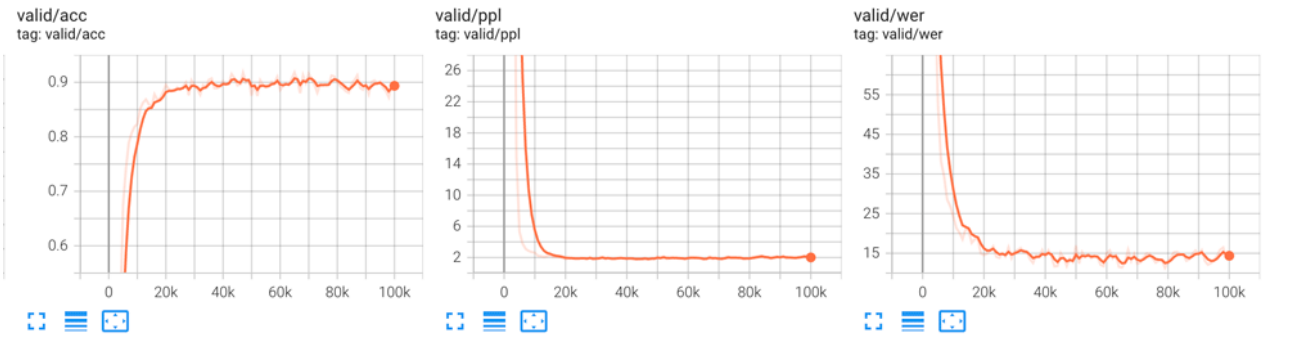

→目次に戻る

```
  tied_embeddings: False
  tied_softmax: False
  encoder:
    type: "transformer"
    num_layers: 16
    num_heads: 4
    embeddings:
      embedding_dim: 80  # 入力スペクトログラムの周波数(エンコーダー側は単語埋め込み層なし。)
    hidden_size: 512
    ff_size: 2048
    dropout: 0.1
    freeze: False
    subsample: True       # 1d-conv を使って入力系列を圧縮するかどうか
    conv_kernel_sizes: [5, 5] # 1d-convのカーネルサイズ
    conv_channels: 512    # 1d-convの隠れ層のサイズ
    in_channels: 80       # 入力スペクトログラムの周波数
    layer_norm: "pre"
  decoder:
    type: "transformer"
    num_layers: 8
    num_heads: 4
    embeddings:
      embedding_dim: 512
      scale: True
      dropout: 0.1
      hidden_size: 512
    ff_size: 2048
    dropout: 0.1
    freeze: False
    layer_norm: "pre"
```

「train」モードで訓練を始めます。

```
$ python -m joeynmt train configs/librispeech_100h.yaml --skip_test
```

およそ 30k ステップで、Accuracy が 0.9、Perplexity が 2、Word Error Rate が 15 くらいの値に落ち着いてきます。



以降はあまり変化が見られなかったので、100k ステップでいったん訓練を打ち切りました。100k ステップを回すのに、NVIDIA RTX A6000 の GPU で約 22 時間かかりました。

モデルの評価

保存されたチェックポイント 10 個の平均を取ります。

```
$ python scripts/average_checkpoints.py --inputs models/librispeech100h/*00.ckpt --output models/librispeech100h/avg10.py
```

「test」モードでモデルの性能を評価します。

```
$ python -m joeynmt test configs/librispeech_100h.yaml --ckpt models/librispeech100h/avg10.py
2022-06-30 01:01:47,581 - INFO - root - Hello! This is Joey-NMT (version 2.0.0).
[...]
2022-06-30 01:02:14,239 - INFO - joeynmt.prediction - Decoding on dev set...
2022-06-30 01:02:14,239 - INFO - joeynmt.prediction - Predicting 2703 example(s)...
2022-06-30 01:14:47,924 - INFO - joeynmt.prediction - Evaluation result wer: 11.04
2022-06-30 01:14:47,928 - INFO - joeynmt.prediction - Decoding on test set...
2022-06-30 01:14:47,928 - INFO - joeynmt.prediction - Predicting 2620 example(s)...
2022-06-30 01:23:56,345 - INFO - joeynmt.prediction - Evaluation result wer: 12.33
```

この学習済みモデルは公開しています。jupyter notebook [※11] では、モデルの書き起こし結果とともに、その入力音声を聞くこともできます。ぜひリポジトリからアクセスしてみてください。

※ 11：この notebook は、AIMS Senegal での「NMT in Practice」の講義で使ったものが基になっています。

最後に

3 回にわたってお届けしてきた本連載も今回で最後となりました。「JoeyNMT」は 2019 年にオープンソース化されて以降、多くのコントリビューターに支えられて少しずつ成長してきました。ミニマリスティックな哲学は保ちつつ、新機能の実装、古い依存ライブラリからくる問題への対処など、ボランティアの手で継続的にアップデートされています。

当初の開発目的であった教育用途はもちろん、翻訳の枠を超え、画像キャプション生成、手話翻訳、強化学習などさまざまなプロジェクトで用いられています。本連載での音声認識により、応用の幅はさらに広がったのではないでしょうか。また日本語で「JoeyNMT」を紹介する機会をいただけたことをとてもうれしく思います。本連載が、ニューラル機械翻訳の世界に飛び込むきっかけになれば幸いです。

編集：@IT 編集部
発行：アイティメディア株式会社