Decoding in neural machine translation

Multilingual NLP – Lab 4

Philippos Triantafyllou

January 17, 2026

Table of contents

1 Setup

`JoeyNMT` code, taken from the file.

Load model function.

```
import torch

from tqdm import tqdm
from pathlib import Path

from joeynmt.vocabulary import build_vocab
from joeynmt.model import build_model
from joeynmt.helpers import load_config, get_latest_checkpoint, load_checkpoint


def load_model(cfg_file: str):

    # Load YAML configuration
    with tqdm(total=1, desc="Loading configuration") as pbar:
        cfg = load_config(cfg_file)
        pbar.update(1)

    # Extract directory containing trained model checkpoints
    model_dir = cfg["training"]["model_dir"]

    # Load source and target vocabularies defined in the data config
    with tqdm(total=1, desc="Loading vocabulary") as pbar:
        src_vocab, trg_vocab = build_vocab(cfg["data"])
        pbar.update(1)

    # Load the latest checkpoint and initialize the model
    with tqdm(total=1, desc="Loading model") as pbar:
        # Get the most recent checkpoint
        check = get_latest_checkpoint(Path(model_dir))
        model_checkpoint = load_checkpoint(check, torch.device('cpu'))  # type: ignore

        # Build the model architecture using config and vocabularies
        model = build_model(cfg["model"], src_vocab=src_vocab, trg_vocab=trg_vocab)

        # Restore trained parameters into the model
        model.load_state_dict(model_checkpoint["model_state"])
        pbar.update(1)

    return model
```

Greedy decoding function, used in for evaluating error propagation.

```
from torch import Tensor, IntTensor
from torch.nn import LogSoftmax
from joeynmt.model import Model

log_softmax = LogSoftmax(dim=1)


def greedy_decoding(model: Model, encoder_output: Tensor, max_output_length: int):

    # Build a source mask marking all encoder time steps as valid
    src_mask = torch.tensor([[[True for _ in range(encoder_output.shape[1])]]])

    # Retrieve BOS and EOS token indices
    bos_index = model.bos_index
    eos_index = model.eos_index
```

```
    # Initialize target sequence with only the BOS token and mask
    ys = encoder_output.new_full([1, 1], bos_index, dtype=torch.long)
    trg_mask = src_mask.new_ones([1, 1, 1])

    res = []
    for _ in range(max_output_length):
        model.eval()

        with torch.no_grad():
            logits, _, _, _ = 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
            )

            # Select logits and transform to probabilities
            logits = logits[:, -1]
            _ = log_softmax(logits)

            # Greedily select the token with the highest score
            _, pred_trg_token = torch.max(logits, dim=1)
            pred_trg_token = pred_trg_token.data.unsqueeze(-1)

            # Append predicted token to the target sequence prefix
            ys = torch.cat([ys, IntTensor([[pred_trg_token]])], dim=1)

            # Stop decoding if EOS token is generated
            if pred_trg_token == eos_index:
                break
            # Add predicted token index to result list
            res.append(int(pred_trg_token))

    return res
```

Encode source sentences.

```
from typing import List
from joeynmt.constants import EOS_TOKEN

# Encode a tokenized source sentence into encoder hidden representations
def encode_sentence(sentence: List[str], model):

    # Convert each source token to its vocabulary index and append EOS token
    indexes = [model.src_vocab.lookup(token) for token in sentence + [EOS_TOKEN]]

    # Create a batch of size 1 containing the indexed source sentence
    src = torch.tensor([indexes])

    # Store the true length of the source sentence for the encoder
    lengths = torch.tensor([len(indexes)])

    # Build a source mask marking all source positions as valid
    masks = torch.tensor([[[True for _ in range(len(indexes))]]])

    model.eval()
    with torch.no_grad():
        encoder_output, _, _, _ = model(
            return_type="encode",
            src=src,
            src_length=lengths,
            src_mask=masks
```

```
        )

    return encoder_output
```

Loading configuration and model.

```
# Path to JoeyNMT configuration file
cfg_path = "data/wip_model/config.yaml"

# Load configuration
cfg = load_config(cfg_path) #type: ignore

# Load trained model (CPU)
model = load_model(cfg_path)

# Read maximum output length for decoding
max_output_length = cfg["testing"]["max_output_length"]
```

Building tokenizers.

```
from joeynmt.tokenizers import build_tokenizer

# Build tokenizers defined in the config
tokenizers = build_tokenizer(cfg["data"])

# Select source-language tokenizer
src_lang = cfg["data"]["src"]["lang"]
src_tokenizer = tokenizers[src_lang]
```

Testing.

```
# Raw source sentence
s = "My name is James."

# Preprocess and tokenize
s_tokenized = src_tokenizer(src_tokenizer.pre_process(s))

# Inspect tokenized output
print(s_tokenized)

# Encode the tokenized sentence using the model encoder
encoder_output = encode_sentence(s_tokenized, model)

# Generate target token IDs using greedy decoding
res = greedy_decoding(model, encoder_output, max_output_length)

# Convert token IDs back to readable target tokens
decoded_sentence = model.trg_vocab.array_to_sentence(res) # type: ignore

# Print decoded output
print(decoded_sentence)
```

```
['my', 'name', 'is', 'james.']
['my', 'name', 'is', 'james.']
```

Download data.

```
import pandas as pd

df = pd.read_csv("data/english_portuguese.tsv", sep="\t", header=None, on_bad_lines='skip', names=["id1",
↪  "en", "id2", "pt"])
```

Sort values based on longest sentences. We only use the first 50 sentences for faster execution.

```
df.sort_values(by="en", key=lambda x: x.str.len(), ascending=False, inplace=True)
df = df.head(50).copy()

# Test that all is ok
assert len(df) == 50
assert df["en"].notna().all()
assert df["pt"].notna().all()
```

Export test files.

```
SRC = "data/test_sets/test_en_50.txt"
REF = "data/test_sets/test_pt_50.txt"

df["en"].to_csv( SRC, index=False, header=False )
df["pt"].to_csv( REF, index=False, header=False )
```

Specify test files in the config.

```
cfg["data"]["test"] = {"src": SRC, "trg": REF}
cfg["testing"]["n_best"] = 1
```

Ready to run the beam search.

2 Impact of the beam size

For the beam search, we modify the config with the desired beam size and run the following:

```
python3 -m joeynmt translate config.yaml -o output.txt --ckpt checkpoint.ckpt
```

We also specify the test set in the config file.

The time taken for each beam size is printed below.

```
   beam_size  time_seconds                 output_path
0          1     29.231966   data/outputs/hyp_beam1.txt
1          5     73.003528   data/outputs/hyp_beam5.txt
2         10    117.547759  data/outputs/hyp_beam10.txt
3         15    167.287912  data/outputs/hyp_beam15.txt
4         20    215.639901  data/outputs/hyp_beam20.txt
```

For the evaluation:

```
SRC = "data/test_sets/test_en_50.txt"
REF = "data/test_sets/test_pt_50.txt"
HYP_DIR = "data/outputs"
BEAMS = [1, 5, 10, 15, 20]
```

```
src = open(SRC, encoding="utf-8").read().splitlines()
ref = open(REF, encoding="utf-8").read().splitlines()
refs = [ref]

greedy = open(f"{HYP_DIR}/hyp_beam1.txt", encoding="utf-8").read().splitlines()

assert len(src) == len(ref) == len(greedy)
```

Load `cometkiwi-da`.

```
from comet import download_model, load_from_checkpoint

model_path = download_model("Unbabel/wmt22-cometkiwi-da")
comet_model = load_from_checkpoint(model_path)
```

For each beam size, load the hypotheses and compute BLEU, COMET, and ChrF scores against the references. To compare whether the greedy hypotheses are the same as the beam search ones, we simply compare the two strings. To actually look at "how different" they are, we compute the character-level Levenshtein distance.

We do this in one big for loop:

```
import sacrebleu
import torch
import Levenshtein

rows = []
most_different_examples = {}
most_similar_examples = {}

for beam in BEAMS:
    hyp_path = f"{HYP_DIR}/hyp_beam{beam}.txt"
    hyp = open(hyp_path, encoding="utf-8").read().splitlines()
    assert len(hyp) == len(src)

    # BLEU / chrF
    bleu = sacrebleu.corpus_bleu(hyp, refs).score
    chrf = sacrebleu.corpus_chrf(hyp, refs).score

    # COMET
    comet_data = [{"src": s, "mt": h, "ref": r} for s, h, r in zip(src, hyp, ref)]
    comet_out = comet_model.predict(
        comet_data,
        batch_size=32,
        gpus=1 if torch.cuda.is_available() else 0,
        num_workers=1,
    )
    comet = comet_out.system_score # type: ignore

    # Number of hypotheses identical to greedy
    identical_to_greedy = sum(g == h for g, h in zip(greedy, hyp))

    # Character-level edit distances
    distances = [Levenshtein.distance(g, h) for g, h in zip(greedy, hyp)]

    # Top 5 most different from greedy
    top5_diff_idx = sorted(range(len(distances)), key=lambda i: distances[i], reverse=True)[:5]

    # Add most different examples
    most_different_examples[beam] = [
        {
```

```
            "index": i,
            "src": src[i],
            "greedy": greedy[i],
            "beam": hyp[i],
            "edit_distance": distances[i],
        }
        for i in top5_diff_idx
    ]

    # Top 5 most similar to greedy (but not identical)
    non_zero_diffs = [(i, d) for i, d in enumerate(distances) if d > 0]
    non_zero_diffs.sort(key=lambda x: x[1])

    # Add most similar examples
    most_similar_examples[beam] = [
        {
            "index": i,
            "src": src[i],
            "greedy": greedy[i],
            "beam": hyp[i],
            "edit_distance": d,
        }
        for i, d in non_zero_diffs[:5]
    ]


    # Store aggregate results
    rows.append({
        "beam": beam,
        "BLEU": bleu,
        "chrF": chrf,
        "CometKiwi": comet,
        "identical_to_greedy": identical_to_greedy,
    })
```

```
import pandas as pd

df = pd.DataFrame(rows).sort_values("beam")
display(df)
```

	beam	BLEU	chrF	CometKiwi	identical_to_greedy
0	1	2.464129	21.133642	0.374681	50
1	5	1.021093	16.161680	0.379522	3
2	10	0.448007	12.478834	0.354528	3
3	15	0.466036	12.161700	0.346429	1
4	20	0.500691	12.354588	0.347824	1

It appears that increasing the beam size achieves worse results. This is a known phenomenon in the literature where increasing the beam size over 5 decreases the quality, known as the "beam search curse". We can see that even with a beam size of 5, BLEU and chrF scores drastically drop compared to greedy decoding.

Beam search optimizes high probability sequences, but these do not necessarily correspond to better translations. Increasing the beam search size will inevitably amplify this effect. We can slightly see this effect when comparing the number of identical outputs to greedy decoding, which decreases with larger beam sizes.

While BLEU and chrF scores drop, CometKiwi scores remain stable across different beam

sizes (there is a slight drop still). This is likely to the fact that CometKiwi does not rely on a references, and mostly mesures fluency and adequacy of the output. A candidate translation can be fluent but have nothing to do with the reference.

We can look at the most different and most similar examples between greedy and beam search outputs to have a better idea of what is happening. We do not print th output as it will span many pages.

```
for beam in sorted(most_different_examples.keys()):
    if beam == 1:
        continue

    print("=" * 80)
    print(f"Beam size = {beam}")
    print("=" * 80)

    for ex in most_different_examples[beam]:
        print("SRC   :", ex["src"])
        print("GREEDY:", ex["greedy"])
        print("BEAM  :", ex["beam"])
        print("Edit distance:", ex["edit_distance"])
        print()
```

The examples that have the highest and lowest Levenshtein distance between greedy and beam search outputs are almost the same across different beam sizes. The worst candidate translations are not only in the source language (english), but are actually the exact source sentence copied over. Even at beam size 5, the same examples get selected, indicating that above beam size 5, the search space does not change much.

3 Evaluating error propagation

We basically modify the decoding function to maintain two separate prefix tensors:

- `oracle_prefix` which always appends the reference token `ref_ids[t]`
- `greedy_prefix` which appends the predicted token `pred_greedy_id`.

At each position t, we run the model twice with each prefix and compare both predictions to the same reference token. We stop the loop when `pred_greedy_id` equals `eos_index`.

```
def decoding_with_01_loss(model: Model, encoder_output: Tensor, ref_ids: list, max_output_length: int):
    device = encoder_output.device

    # Build a source mask marking all encoder time steps as valid
    src_mask = torch.ones(1, 1, encoder_output.shape[1], dtype=torch.bool, device=device)

    # Retrieve BOS and EOS token indices
    bos_index = model.bos_index
    eos_index = model.eos_index

    # Initialize both prefixes with BOS
    oracle_prefix = torch.full([1, 1], bos_index, dtype=torch.long, device=device)
    greedy_prefix = torch.full([1, 1], bos_index, dtype=torch.long, device=device)

    oracle_loss = 0
    greedy_loss = 0
    count = 0
```

```
    # Determine maximum decoding length
    decode_length = min(len(ref_ids), max_output_length)

    for t in range(decode_length):
        model.eval()

        with torch.no_grad():
            # 1. Predict with ORACLE prefix
            trg_mask_oracle = torch.ones(1, 1, oracle_prefix.shape[1], dtype=torch.bool, device=device)

            # Get logits for oracle prefix
            logits_oracle, _, _, _ = model(
                return_type="decode",
                # Here we provide the oracle prefix
                trg_input=oracle_prefix,
                encoder_output=encoder_output,
                encoder_hidden=None,
                src_mask=src_mask,
                unroll_steps=None,
                decoder_hidden=None,
                # Here we provide the oracle mask
                trg_mask=trg_mask_oracle
            )
            # Get predicted token from oracle
            _, pred_oracle = torch.max(logits_oracle[:, -1], dim=1)
            pred_oracle_id = int(pred_oracle.item())

            # 2. Predict with GREEDY prefix
            trg_mask_greedy = torch.ones(1, 1, greedy_prefix.shape[1], dtype=torch.bool, device=device)

            # Get logits for greedy prefix
            logits_greedy, _, _, _ = model(
                return_type="decode",
                # Here we provide the greedy prefix
                trg_input=greedy_prefix,
                encoder_output=encoder_output,
                encoder_hidden=None,
                src_mask=src_mask,
                unroll_steps=None,
                decoder_hidden=None,
                # Here we provide the greedy mask
                trg_mask=trg_mask_greedy
            )
            _, pred_greedy = torch.max(logits_greedy[:, -1], dim=1)
            pred_greedy_id = int(pred_greedy.item())

            # Count errors
            if pred_oracle_id != ref_ids[t]:
                oracle_loss += 1
            if pred_greedy_id != ref_ids[t]:
                greedy_loss += 1
            count += 1

            # Stop if greedy predicts EOS
            if pred_greedy_id == eos_index:
                break

            # Update prefixes
            oracle_prefix = torch.cat([oracle_prefix, torch.tensor([[ref_ids[t]]], device=device)], dim=1)
            greedy_prefix = torch.cat([greedy_prefix, torch.tensor([[pred_greedy_id]], device=device)],
    ↪  dim=1)

        return oracle_loss, greedy_loss, count
```

Preparing the data for the experiment. We use the same test set as before.

```
# Paths to your test set
SRC_PATH = "data/test_sets/test_en_50.txt"
REF_PATH = "data/test_sets/test_pt_50.txt"

# Load raw sentences
src_sentences = open(SRC_PATH, encoding="utf-8").read().splitlines()
ref_sentences = open(REF_PATH, encoding="utf-8").read().splitlines()
assert len(src_sentences) == len(ref_sentences)

# Build  tokenizers
tokenizers = build_tokenizer(cfg["data"])

# Select source and target language tokenizers
src_lang = cfg["data"]["src"]["lang"]
trg_lang = cfg["data"]["trg"]["lang"]

# Get source and target tokenizers
src_tokenizer = tokenizers[src_lang]
trg_tokenizer = tokenizers[trg_lang]

# Tokenize all sentences
src_tokenized_sentences = [src_tokenizer(src_tokenizer.pre_process(s)) for s in src_sentences]
ref_tokenized_sentences = [trg_tokenizer(trg_tokenizer.pre_process(s)) for s in ref_sentences]
```

We loop over the sentences and compute the 0/1 loss for both oracle and greedy modes. Finally, we compute the average losses across all sentences.

```
total_oracle_loss = 0
total_greedy_loss = 0
total_count = 0

for src_tokens, ref_tokens in zip(src_tokenized_sentences, ref_tokenized_sentences):
    # Encode source sentence
    encoder_output = encode_sentence(src_tokens, model)

    # Convert reference tokens to IDs
    ref_ids = [model.trg_vocab.lookup(tok) for tok in ref_tokens]
    ref_ids.append(model.eos_index)

    # Compute 0/1 losses and accumulate
    o_loss, g_loss, n = decoding_with_01_loss(model, encoder_output, ref_ids, max_output_length)
    total_oracle_loss += o_loss
    total_greedy_loss += g_loss
    total_count += n

print("Average 0/1 loss (oracle):", round(total_oracle_loss / total_count, 3))
print("Average 0/1 loss (greedy):", round(total_greedy_loss / total_count, 3))
```

```
Average 0/1 loss (oracle): 0.493
Average 0/1 loss (greedy): 0.921
```

The results confirm the presence of exposure bias in the NMT model. The oracle mode achieves a lower loss because the model was trained with teacher forcing, and is conditioned to always receive correct previous tokens. On the other hand we clearly see that the greedy mode decoder achieves much higher loss, because of the errors in early predictions corrupting the history and being propagated downstream.

Exposure bias is a real problem and a strong limitation to maximum likelihood training with teacher forcing. To mitigate such an issue, we need to expose the model to its own predictions during training, for example using scheduled sampling or reinforcement learning techniques.