A REPORT

ON

CS F429 Natural Language Processing: Assignment 2

by

Name of the student	ID number
Pavas Garg	2021A7PS2587H
Atharva Dashora	2021A7PS0127H
Rohit Reddy Daareddy	2021A7PS0372H

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI
(Hyderabad Campus)

Novemeber, 2024

CONTENTS

List of Figures

List of Algorithms

1 Introduction

Machine Translation (MT) systems aim to automatically translate text or speech from one language to another. These systems have evolved significantly over time, moving from rule-based and statistical approaches to modern neural-based systems. The main goal is to preserve the meaning of the original content while adapting it to the syntax, grammar, and semantics of the target language.

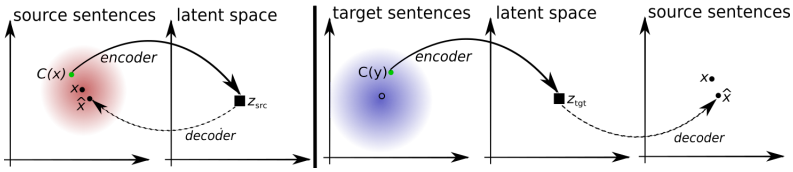

Figure 1: Translation

Types of Machine Translation Systems

1. **Rule-Based Machine Translation (RBMT):**
   - Based on linguistic rules and grammar of source and target languages.
   - Requires extensive manual effort to create dictionaries and grammar rules.
2. **Statistical Machine Translation (SMT):**
   - Relies on probability distributions from bilingual corpora.
   - Models such as phrase-based SMT use large datasets to identify likely translations.
3. **Neural Machine Translation (NMT):**
   - Leverages deep learning to translate text.
   - Employs models like Sequence-to-Sequence (Seq2Seq) with attention mechanisms for high-quality translations.
   - Example: Transformer-based models like Google's BERT and OpenAI's GPT.
4. **Hybrid Approaches:**
   - Combine strengths of rule-based, statistical, and neural methods to enhance translation accuracy.

Challenges in Machine Translation

- **Ambiguity:** Words and phrases can have multiple meanings depending on context.
- **Resource Scarcity:** Many languages lack sufficient parallel corpora.

- **Idiomatic Expressions:** Translating phrases that don't have direct equivalents in the target language.
- **Cultural Nuances:** Contextual and cultural meanings are often hard to capture.

General Unsupervised Approaches in Machine Translation

Unsupervised machine translation addresses the challenges posed by the lack of large parallel corpora. It relies on monolingual corpora and unsupervised learning techniques to train translation models.

Key Approaches

1. **Language Modeling:**
   - Train models to understand and generate text in a single language.
   - Use these models to align two languages in a shared latent space.
2. **Word Embedding Alignment:**
   - Align embeddings of words in different languages using methods like adversarial training or Procrustes analysis.
   - Example: FastText embeddings trained on monolingual data and mapped across languages.
3. **Back-Translation:**
   - Generate synthetic parallel data by translating monolingual data from one language to another and back.
   - Helps iteratively improve the translation quality.
4. **Adversarial Training:**
   - Use a discriminator to ensure the generated translations are indistinguishable from real sentences.
   - Example: Unsupervised NMT frameworks often employ this technique.
5. **Denoising Autoencoders:**
   - Train models to reconstruct sentences from corrupted versions.
   - Help in learning better representations for both source and target languages.
6. **Shared Encoder-Decoder Architectures:**
   - Use a single encoder-decoder model to handle both languages by sharing parameters.
   - Example: Language-independent representations for unsupervised MT.

Advantages of Unsupervised Approaches

- No dependency on large parallel corpora.
- Enables translation for low-resource languages.
- Reduces costs and time for data preparation.

Challenges of Unsupervised Approaches

- Quality is often lower compared to supervised methods with abundant data.
- Difficulties in handling distant language pairs or highly divergent grammar structures.
- High sensitivity to noise in the training data.

2 Concepts

2.1 Denoising Autoencoders (DAEs)

Denoising Autoencoders are a key component in our approach, designed to prevent the trivial copying of input data during training. By corrupting input sentences with noise—such as dropping random words or slightly shuffling their order—the autoencoder is forced to reconstruct the original sentence. This process ensures the encoder learns robust representations that capture the structural and semantic essence of the input. In our model, the encoder uses an LSTM architecture shared across both source and target languages to map noisy inputs into a shared latent space. The decoder, also an LSTM with attention, reconstructs the original input from this latent representation. This approach is crucial in unsupervised machine translation, as it enables the model to generalize from unaligned, noisy data, forming a foundation for effective translation.

2.2 Encoders and Decoders

Encoders and decoders form the backbone of our translation system. The encoder, an LSTM-based model, processes input sentences into a sequence of hidden states, which are then mapped into a shared latent space. By sharing parameters across both source and target languages, the encoder ensures that the latent representations align across languages. The decoder, also an LSTM, generates target sentences one token at a time, using an attention mechanism to focus on the most relevant encoder states during each step of decoding. The use of shared parameters in both the encoder and decoder reduces the model's complexity and promotes consistency between language pairs. This shared architecture enables efficient cross-lingual representation learning, making the model effective for translation tasks in an unsupervised setting.

2.3 Cross-Domain Training

Cross-domain training is a critical step to ensure that sentences can be translated between languages and reconstructed back into their original form. During this process, a noisy translation is generated using the current model, and the encoder-decoder pair is trained to reconstruct the original sentence from this translation. This iterative training procedure aligns the source and target latent spaces, refining the model's ability to map between languages. The cross-domain loss measures the discrepancy between the original sentence and its reconstructed version, driving the encoder-decoder pair to improve with each iteration. This method leverages the power of monolingual data to iteratively enhance translation quality, making it a cornerstone of our unsupervised approach.

2.4 Adversarial Training

Adversarial training introduces a discriminator to align the latent spaces of the source and target languages. The discriminator is tasked with predicting the language of a latent representation, while the encoder is trained to make these representations indistinguishable. This adversarial interaction ensures that the encoder outputs language-invariant features, enabling the decoder to generate accurate and fluent sentences regardless of the input language. The discriminator is a multilayer perceptron with three hidden layers and Leaky ReLU activations, and its predictions guide the encoder to refine its outputs.

Adversarial training plays a crucial role in aligning the source and target domains, complementing the cross-domain and denoising objectives by enhancing the shared latent space's quality.

2.5 BERT Embeddings

BERT (Bidirectional Encoder Representations from Transformers) embeddings are state-of-the-art contextual word embeddings that capture the meaning of words based on their surrounding context. Unlike traditional static embeddings like Word2Vec or GloVe, which assign a single vector to a word regardless of context, BERT dynamically adjusts word representations depending on the sentence they appear in. This is achieved using a transformer architecture that processes input bidirectionally, meaning it considers both preceding and following words simultaneously. As a result, BERT excels in understanding polysemous words (words with multiple meanings) and complex linguistic relationships, such as syntax and semantics. BERT embeddings are highly versatile and have been successfully applied to various tasks, including text classification, question answering, and sequence labeling. By providing rich, context-aware representations, BERT has significantly advanced the performance of NLP systems.

2.6 spaCy Tokenizers

spaCy tokenizers are efficient and highly customizable tools for splitting text into smaller units called tokens, which include words, punctuation marks, and special characters. The tokenization process is crucial for downstream NLP tasks, as it defines the basic building blocks of text analysis. spaCy's tokenizer uses a deterministic, rule-based approach, which is enhanced by language-specific data to ensure accurate tokenization across diverse languages. For instance, it can handle contractions in English (e.g., "don't" $\rightarrow$ "do" and "n't") and adjust for special cases like hyphenated words or URLs. Additionally, spaCy supports custom tokenization rules, allowing users to tweak the behavior to suit domain-specific requirements. Its integration with other spaCy components, such as part-of-speech tagging and named entity recognition, ensures a seamless pipeline for processing text. With its speed and flexibility, spaCy's tokenizer is widely adopted in both research and production environments.

2.7 Interplay Between Concepts

The denoising autoencoder ensures robust feature learning by reconstructing noisy inputs, while the shared LSTM-based encoder-decoder architecture facilitates translation by mapping source sentences to a common latent space. Cross-domain training aligns source and target domains, enabling the reconstruction of original sentences across languages. Adversarial training refines this alignment by making the latent representations language-invariant. Together, these components form a cohesive system that iteratively improves translation quality without the need for parallel data, demonstrating the power and potential of unsupervised machine translation.

3 Apparatus

Machine	Apple MacBook Pro M1 (2021)
RAM	8 GB LPDDR4X 4266MHz
Processor Name	Apple M1
Processor Specs	8 cores, 8 threads, 2.06-3.22 GHz, L1 Cache: 2MB, L2 Cache: 12MB
Python	Version 3.10

4 Dataset Description

The dataset used for this approach is **monolingual corpora** from both the source and target languages. Unlike traditional supervised translation models that require parallel sentence pairs, this method leverages unaligned data. The monolingual datasets consist of **diverse text samples from various domains**, such as news articles, books, or conversational data, to ensure broad coverage of language structures and vocabulary.

Each dataset should ideally **represent the linguistic and semantic richness of its respective language**, including common idioms, syntax variations, and stylistic differences. Since the approach relies on unsupervised learning, the quality and diversity of the monolingual corpora are critical to achieving robust cross-lingual representations. Preprocessing steps, such as lowercasing, removing special characters, and sentence segmentation, may be applied to standardize the data.

To enhance model training, artificial noise is introduced during preprocessing to corrupt input sentences. This includes random word deletions, shuffling, and replacements, encouraging the model to learn robust patterns for reconstruction and translation. **The availability of large, high-quality monolingual corpora is a key factor in the success of the system.**

4.1 Data Sources

The following datasets were utilized in the implementation:

WMT24 MTData: A collection of machine translation data for the WMT24 workshop, accessible at `https://www2.statmt.org/wmt24/mtdata/`.

Monolingual Data was taken from these resources:

- **English News Commentary:** Dataset containing english news commentary, available at `https://data.statmt.org/news-commentary/v18.1/training-monolingual/`. Text file used was *news-commentary-v18.txt*
- **French Wortschatz Leipzig:** A comprehensive resource for French lexical data, available at `https://wortschatz.uni-leipzig.de/en/download/French`. Text file used was *fra_news_2023_1M-sentences.txt*
- **German Wortschatz Corpus:** A comprehensive resource for German lexical data, available at `https://wortschatz.uni-leipzig.de/en/download/German`. Text file used was *deu-de_web-public_2019_10K-sentences.txt*

Parallel Datasets for Testing were taken from these resources:

- **News-Commentary:** `https://data.statmt.org/news-commentary/v18.1/training-monoli`

5 Methodology for Machine Translation

5.1 Preprocessing Pipeline

The preprocessing pipeline involves multiple steps to clean and prepare input sentences for tokenization and further processing. The steps include removing empty sentences, text preprocessing, tokenization, and filtering long sentences.

Step 1: Removing Empty Sentences Empty sentences or sentences with missing values are removed to ensure data integrity.

Algorithm 1 Removing Empty Sentences

procedure REMOVE-EMPTY-SENTENCES(*df*)
    Filter rows where `Sentence` is not null.
    Filter rows where `Sentence` is not an empty string.
    return Filtered dataset.
end procedure

Step 2: Text Preprocessing Each sentence is converted to lowercase, and punctuation marks are removed to standardize the data.

Algorithm 2 Text Preprocessing

procedure PREPROCESS-TEXT(*sentence*)
    Convert `sentence` to lowercase.
    Remove punctuation from `sentence`.
    return Processed `sentence`.
end procedure

Step 3: Tokenization with spaCy Sentences are tokenized into words using the spaCy library. A custom function, `Remerge-Sent`, recursively merges tokens that are not separated by whitespace to ensure accurate tokenization.

Algorithm 3 Tokenization with spaCy

```
procedure REMERGE-SENT(sent)
    changed ← True
    while changed do
        changed ← False
        for i ← 0 to len(sent) − 1 do
            if sent[i].whitespace_ is False then
                Merge sent[i] and sent[i + 1] using retokenizer.
                changed ← True.
            end if
        end for
    end while
    return Merged sent.
end procedure
procedure TOKENIZE(sentence, nlp)
    doc ← nlp(sentence).
    tokens ← Remerge-Sent(doc).
    return Tokenized tokens.
end procedure
```

Step 4: Filtering Long Sentences Sentences exceeding a predefined maximum length (`MAX_SEQ_LEN`) are removed to maintain computational efficiency.

Algorithm 4 Filtering Long Sentences

```
procedure FILTER-LONG-SENTENCES(df, MAX_SEQ_LEN)
    for all sentence in df do
        Tokenize sentence.
        if len(tokens) > MAX_SEQ_LEN then
            Remove sentence from df.
        end if
    end for
    return Filtered df.
end procedure
```

Complete Preprocessing Pipeline The entire preprocessing pipeline integrates all the above steps. It removes empty sentences, preprocesses text, tokenizes sentences, and filters out overly long sentences.

Algorithm 5 Complete Preprocessing Pipeline

procedure PREPROCESS-DATASET(df, nlp, MAX_SEQ_LEN)
    $df \leftarrow$ Remove empty or null sentences.
    for all *sentence* in df **do**
        sentence $\leftarrow$ Preprocess text.
        tokens $\leftarrow$ Tokenize *sentence* using spaCy.
        if $|tokens| >$ MAX_SEQ_LEN **then**
            Remove *sentence* from df.
        end if
    end for
    return Preprocessed df.
end procedure

5.2 Generating BERT Embeddings

In this subsection, we describe the process of generating BERT embeddings for English and French sentences. The steps involved are as follows:

1. **Load Pre-trained BERT Models and Tokenizers:** We load the pre-trained BERT models for English (`'bert-base-uncased'`) and French (`'camembert-base'`), along with their corresponding tokenizers. These tokenizers split the input sentences into subword tokens, which are then passed through the BERT model to obtain token-level embeddings. The embeddings are extracted from the BERT models and are ready for use.

2. **Retrieve BERT Embeddings for Tokens:** For each sentence, the tokenizer splits the sentence into subwords. The sentence is then passed through the BERT model to obtain embeddings. Since some words may be split into multiple subwords, we aggregate the embeddings for those subwords by averaging them back into a single representation for each word.

3. **Generate BERT Embeddings for the Dataset:** After processing the sentences individually, we aggregate the BERT embeddings for all the sentences in the dataset. This process involves tokenizing each sentence, passing it through the BERT model to generate embeddings, and storing them in a list for further use.

4. **Generate Padded BERT Embeddings:** The embeddings are then padded or truncated to ensure that all sequences have the same length. Padding is applied to shorter sequences, while longer sequences are truncated to a predefined maximum sequence length. This ensures uniformity in the embeddings for input to downstream tasks.

Below, we present the algorithms used in this process:

Algorithm 6 Load Pre-trained BERT Models and Tokenizers

Input: None
Output: Tokenizers and BERT models for English and French
Load pre-trained BERT tokenizer for English using `'bert-base-uncased'`
Load pre-trained BERT tokenizer for French using `'camembert-base'`
Load BERT model for English using `'bert-base-uncased'`
Load BERT model for French using `'camembert-base'`
Move models to the appropriate device (e.g., GPU)
Extract word embeddings from BERT models

Algorithm 7 Retrieve BERT Embeddings for Tokens

Input: Tokenized sentence *tokens*, tokenizer, BERT model
Output: Aggregated BERT embeddings for tokens
Tokenize the input sentence using the tokenizer
Pass the tokenized sentence through the BERT model
Extract token embeddings from the BERT model
Retrieve word IDs to align subword tokens with original words
Aggregate subword embeddings into a single token embedding by averaging
Return the aggregated embeddings

Algorithm 8 Generate BERT Embeddings for Dataset

Input: DataFrame *df*, tokenizer, BERT model
Output: List of embeddings for all sentences in *df*
Initialize an empty list to store embeddings
for each sentence in *df* **do**
    Tokenize the sentence into subword tokens
    Generate embeddings for the tokenized sentence using `Get-BERT-Embeddings`
    Append embeddings to the list
end for
Return the list of embeddings

Algorithm 9 Generate Padded BERT Embeddings

Input: DataFrame *df*, List of embeddings *embedding_list*, max sequence length *max_len*
Output: DataFrame with padded embeddings
Pad the sequences in *embedding_list* to the maximum length using `pad_sequence`
if Sequence length exceeds *max_len* **then**
    Truncate the sequences to *max_len*
else
    Pad shorter sequences with zeros to reach *max_len*
end if
Assign padded embeddings to the DataFrame
Print the shape of padded embeddings

Summary:

- **Step 1:** Loads pre-trained models and tokenizers for English and French.
- **Step 2:** Retrieves token embeddings and aggregates them back to the word level.
- **Step 3:** Processes the dataset by tokenizing sentences and generating embeddings.
- **Step 4:** Ensures all sequences are of the same length by padding or truncating them.

This structure allows for efficient generation of BERT embeddings for both English and French text, ensuring that the embeddings are consistent and ready for further processing.

5.3 Noise Injection Techniques for Sequence Data

In this subsection, we describe two noise injection techniques used to augment sequence data: word dropout and sentence shuffling. These techniques are commonly used in data augmentation to improve the robustness of models by introducing controlled noise into the input sentences.

1. **Word Dropout:** This technique randomly drops words from the input sentence based on a probability parameter pwd. For each word in the sentence, if a randomly generated value exceeds the dropout probability, the word is retained; otherwise, it is replaced with a padding token (typically a zero embedding).
2. **Sentence Shuffling:** This technique shuffles the words in the sentence, ensuring that the new positions of the words respect a maximum allowed distance, k, between the original and permuted positions. The shuffle is done in a way that no word is moved too far from its original position, preserving the sentence's structure while introducing noise.
3. **Combined Noise Injection:** The combined noise injection process applies both word dropout and sentence shuffling sequentially to introduce both types of noise into the sequence data.

Below, we present the algorithms used for these techniques:

Algorithm 10 Apply Word Dropout

Input: Sentence *sentence*, Dropout probability *pwd*, Padding embedding *padding_embedding*
Output: Noisy sentence with dropped words
Initialize empty list *noisy_sentence* and $drop\_count = 0$
for each word in *sentence* **do**
    if random value ¿ *pwd* **then**
        Add word to *noisy_sentence*
    else
        Increment *drop_count*
    end if
end for
if *padding_embedding* is None **then**
    Set $padding\_embedding = 0$ (same shape as a word embedding)
end if
Extend *noisy_sentence* by adding *drop_count* number of padding embeddings
Return *noisy_sentence* as a tensor

Algorithm 11 Apply Sentence Shuffling

Input: Sentence *sentence*, Maximum allowed shuffle distance k, Perturbation factor *alpha*
Output: Shuffled sentence
$n \leftarrow$ Length of the sentence
Generate random permutation vector $q = \text{torch.arange}(n) + \text{torch.rand}(n) * \alpha$
Sort q to get permuted indices *permuted_indices*
for each index i in the sentence **do**
    if $|permuted\_indices[i] - i| > k$ **then**
        Set $permuted\_indices[i] = i$
    end if
end for
Shuffle sentence based on *permuted_indices*
Return shuffled sentence

Algorithm 12 Apply Combined Noise (Word Dropout and Sentence Shuffling)

Input: Sequence x, Dropout probability *pwd*, Maximum shuffle distance k, Perturbation factor *alpha*
Output: Noisy sentence after applying both word dropout and sentence shuffling
If x is a list, convert it to tensor
Apply word dropout on x to get noisy sentence x_noisy
Apply sentence shuffling on x_noisy to get final noisy sentence
Return final noisy sentence

Summary of Noise Injection Techniques:

- **Word Dropout:** Randomly drops words in a sentence based on a given probability.

- **Sentence Shuffling:** Shuffles words in a sentence with a constraint on the maximum allowed distance between word positions.
- **Combined Noise Injection:** Applies both word dropout and sentence shuffling sequentially to introduce controlled noise in the data.

These techniques introduce noise into the input data, which helps in improving the model's robustness and generalization by making the training data more diverse.

5.4 Encoder Architecture and Dimensions

The Encoder module is designed to process sequential data and generate a latent representation using an LSTM layer followed by fully connected layers. Additionally, a language embedding is concatenated with the input features to provide language-specific information. Regularization techniques such as dropout and batch normalization are employed to enhance the model's training stability and prevent overfitting.

The encoder consists of the following key components:

- **LSTM Layer:**

$$\text{LSTM}(x_t) = h_t, c_t$$

where x_t is the concatenation of input features and language embeddings at time step t, and h_t and c_t represent the hidden state and cell state of the LSTM, respectively. The LSTM is bidirectional, so it processes the input sequence in both forward and reverse directions.

The input to the LSTM layer has the shape $(batch\_size, seq\_len, input\_dim+lang\_embed\_dim)$, where:

  - $batch\_size$ is the number of sequences in the batch.
  - seq_len is the length of the sequence.
  - $input\_dim$ is the dimension of each token embedding.
  - $lang\_embed\_dim$ is the size of the language embedding.

The output from the LSTM layer will have shape $(batch\_size, seq\_len, hidden\_dim \times 2)$, since the LSTM is bidirectional.

- **Language Embedding:** The language embedding is generated by the embedding layer:

$$\text{Lang\_Emb} = \text{Embedding}(lang\_idx)$$

where $lang\_idx$ is the index of the language for each input sequence, and Lang_Emb has shape $(batch\_size, lang\_embed\_dim)$. This embedding is then expanded to match the sequence length and concatenated with the input features.

- **Fully Connected Layers:** After the LSTM, the output is passed through three fully connected layers:

$$\text{fc1}(h_t) = W_1 h_t + b_1$$

followed by ReLU activation:

$$\text{ReLU}(\text{fc1}(h_t)) = \max(0, \text{fc1}(h_t))$$

and similar transformations for fc2 and fc3:

$$\text{fc2}(\text{ReLU}(\text{fc1})) = W_2\text{ReLU}(\text{fc1}) + b_2$$

$$\text{fc3}(\text{ReLU}(\text{fc2})) = W_3\text{ReLU}(\text{fc2}) + b_3$$

The dimensionalities of these layers are as follows:

- The first fully connected layer fc1 takes input of shape $(batch\_size, seq\_len, hidden\_dim \times 2)$ and outputs $(batch\_size, seq\_len, hidden\_dim)$.
- The second fully connected layer fc2 takes input of shape $(batch\_size, seq\_len, hidden\_dim)$ and outputs $(batch\_size, seq\_len, hidden\_dim//2)$.
- The third fully connected layer fc3 outputs the final prediction with shape $(batch\_size, seq\_len, output\_dim)$.

- **Regularization:**
  - **Dropout:** Applied after the LSTM output to reduce overfitting. Dropout randomly sets a fraction p of the LSTM output values to zero.
  - **Batch Normalization:** Applied to the LSTM output to stabilize learning by normalizing the activations across the batch.

5.4.1 Dimensions at Each Step

The dimensions of the data at each step in the forward pass are as follows:

1. **Input to LSTM:** The input to the LSTM layer is the concatenation of token embeddings and language embeddings:

   $$\text{Input\_Dim} = (batch\_size, seq\_len, input\_dim + lang\_embed\_dim)$$

   where $input\_dim$ is the dimension of token embeddings, and $lang\_embed\_dim$ is the dimension of language embeddings.

2. **LSTM Output:** The LSTM layer outputs a sequence of hidden states, each of shape $(batch\_size, seq\_len, hidden\_dim \times 2)$, where the factor of 2 accounts for the bidirectional nature of the LSTM.

3. **After Dropout and Batch Normalization:** The dropout and batch normalization are applied to the LSTM output. After batch normalization, the output has the shape:

   $$(batch\_size, seq\_len, hidden\_dim \times 2)$$

4. **Fully Connected Layers:**
   - **fc1 Output:** After the first fully connected layer, the output has shape:

     $$(batch\_size, seq\_len, hidden\_dim)$$

   - **fc2 Output:** After the second fully connected layer, the output has shape:

     $$(batch\_size, seq\_len, hidden\_dim//2)$$

   - **fc3 Output:** The final output from the third fully connected layer has shape:

     $$(batch\_size, seq\_len, output\_dim)$$

5.4.2 Algorithm: Encoder Forward Pass

Below is the algorithm for the forward pass of the encoder:

Algorithm 13 Encoder Forward Pass

Input: Sequence of input embeddings $inputs$, language index $lang\_idx$
Output: Output after applying LSTM and fully connected layers
Get language embedding: lang_emb = lang_embedding($lang\_idx$)
Expand language embedding: lang_emb $\leftarrow$ lang_emb.$expand(-1, inputs.size(1), -1)$
Concatenate input embeddings with language embeddings: inputs_with_lang = torch.cat($inputs$, lang_emb, $dim = -1$)
Apply LSTM: lstm_out, _ = LSTM($inputs\_with\_lang$)
Apply dropout: lstm_out $\leftarrow$ dropout($lstm\_out$)
Apply batch normalization: lstm_out $\leftarrow$ batch_norm(lstm_out.$transpose(1, 2)$).$transpose(1, 2)$
Apply fully connected layers:
fc1_output = ReLU(fc1(lstm_out))
fc2_output = ReLU(fc2(fc1_output))
latent_output = fc3(fc2_output)
Return: fc1_output, latent_output

5.5 Decoder Explanation

The decoder in this architecture is designed as a sequence-to-sequence model with an LSTM and an attention mechanism. It takes the output from the encoder, applies attention to focus on specific parts of the sequence, and generates the predicted output sequence. Here's an overview of the architecture and its dimensions.

5.5.1 Decoder Inputs

The decoder takes several inputs:

- **Latent vectors:** Output from the encoder (dimension: batch_size, seq_len, DECODER_LATENT_
- **Encoder outputs:** Output from the encoder used by the attention mechanism (dimension: batch_size, seq_len, ENCODER_HIDDEN_DIM).
- **Hidden state:** The hidden state from the previous time step, which is updated at each time step (dimension: num_layers, batch_size, DECODER_HIDDEN_DIM).
- **Language embedding:** A language-specific embedding to provide additional context (dimension: batch_size, lang_embed_dim).

5.5.2 Decoder Layer Breakdown

- **LSTM Layer:** The LSTM processes the concatenated input consisting of the context vector, latent vector, and language embedding at each time step.

  Input: (batch_size, 1, DECODER_LATENT_DIM+ENCODER_HIDDEN_DIM+lang_embed_dim)

  Output: (batch_size, 1, DECODER_HIDDEN_DIM) = (batch_size, 1, 300)

- **Attention Mechanism:** The attention mechanism computes the context vector based on the hidden state and encoder outputs. The context vector has the shape:

$$\text{Context vector: } (\text{batch\_size}, \text{ENCODER\_HIDDEN\_DIM}) = (\text{batch\_size}, 300)$$

- **Fully Connected Layer:** After the LSTM processes the concatenated input, the output is passed through the fully connected layer to produce the final prediction.

  $$\text{Output prediction: } (\text{batch\_size}, \text{seq\_len}, \text{DECODER\_OUTPUT\_DIM}) = (\text{batch\_size}, \text{seq\_len}, 768)$$

5.5.3 Decoder Forward Pass

1. Input Processing: The latent vectors (latent_vectors) have the shape:

$$(\text{batch\_size}, \text{seq\_len}, \text{DECODER\_LATENT\_DIM}) = (\text{batch\_size}, \text{seq\_len}, 100)$$

The encoder outputs (encoder_outputs) have the shape:

$$(\text{batch\_size}, \text{seq\_len}, \text{ENCODER\_HIDDEN\_DIM}) = (\text{batch\_size}, \text{seq\_len}, 300)$$

The language embedding index (lang_idx) produces an embedding of shape:

$$(\text{batch\_size}, \text{lang\_embed\_dim}) = (\text{batch\_size}, 32)$$

2. Attention Mechanism: The hidden state from the previous time step, which is updated at each time step, is passed to the attention mechanism along with the encoder outputs to compute attention scores. The attention mechanism returns the context vector.

3. Concatenation of Inputs: The context vector, the current latent vector, and the language embedding are concatenated and passed as input to the LSTM at each time step.

4. LSTM Processing: The LSTM processes the concatenated input and updates the hidden state at each time step. The output of the LSTM is used to predict the next token in the sequence.

5. Final Prediction: After processing the sequence, the fully connected layer produces the final prediction.

5.5.4 Decoder Dimensions

- **ENCODER_INPUT_DIM:** 768
- **ENCODER_HIDDEN_DIM:** 300
- **ENCODER_OUTPUT_DIM:** 100
- **DECODER_OUTPUT_DIM:** 768
- **DECODER_LATENT_DIM:** 100
- **DECODER_HIDDEN_DIM:** 300

Training Denoising Autoencoder (DAE) Model

Cosine Similarity Loss Function

The cosine similarity loss is used to measure the difference between the predicted outputs and the target outputs in the training process. This loss function is effective for capturing the angular distance between two vectors, which is useful when training models to generate embeddings or reconstruct input sequences.

The `cosine_similarity_loss` function is defined as follows:

- **Predictions and Targets:** The predictions and targets are first flattened to 2D tensors with dimensions (batch_size $\times$ seq_len, embedding_dim). This allows us to treat each token in the sequence as a separate element for comparison.
- **Masking Padding Tokens:** A mask is created to ignore padding tokens during loss computation. Padding tokens are identified by checking if the sum of their embeddings is equal to zero (or a specific padding index, given by `pad_idx`).
- **Cosine Similarity:** The cosine similarity is computed between the flattened predictions and targets using `F.cosine_similarity`. This gives a value between -1 and 1, where 1 indicates perfect similarity.
- **Loss Computation:** The loss is computed as the inverse of the cosine similarity ($1 - \text{cos\_sim}$), and the mask is applied to exclude the padded positions from the loss. The final loss is averaged over all non-padding tokens.

The loss function can be written as:

$$\text{loss} = \frac{1}{\sum \text{mask}} \sum (1 - \text{cos\_sim}) \cdot \text{mask}$$

Training the Denoising Autoencoder (DAE)

The training process for the DAE model consists of multiple epochs where the model learns to map noisy inputs back to their original form. The architecture consists of an encoder, a decoder, and a cosine similarity loss function.

Inputs: - `encoder_model`: The encoder neural network model. - `decoder_model`: The decoder neural network model. - `optimizer`: The optimization algorithm (e.g., Adam). - `data_loader`: The data loader containing the training data. - `clip`: The gradient clipping threshold. - `pad_idx`: The padding token index.

Outputs: - `all_losses`: A list containing the loss value for each epoch.

The training algorithm is described below:

Algorithm 14 Training the Denoising Autoencoder (DAE) Model

Initialize the encoder and decoder models
Initialize the optimizer
Set `NUM_EPOCHS` to the number of epochs
Set `clip` to the gradient clipping threshold
Set `pad_idx` to the padding token index
Initialize `all_losses` as an empty list
for epoch = 1 to `NUM_EPOCHS` **do**
    Set `epoch_loss` to 0
    for batch in `data_loader` **do**
        Extract `inputs`, `targets`, and `lang_idx` from the batch
        Add noise to `inputs` using `apply_noise` and create `noisy_inputs`
        Zero the gradients of the optimizer
        Pass `noisy_inputs` through the encoder to get `encoder_outputs` and `latent_vectors`
        Initialize the decoder's hidden state using `decoder_model.init_hidden()`
        Pass `latent_vectors`, `encoder_outputs`, and `hidden` through the decoder to get `outputs` and updated `hidden`
        Compute the loss using the `cosine_similarity_loss` function
        Backpropagate the loss
        Clip gradients using `torch.nn.utils.clip_grad_norm_`
        Step the optimizer
        Add the loss to `epoch_loss`
    end for
    Compute the average loss for the epoch: avg_epoch_loss = $\frac{\text{epoch\_loss}}{\text{len(data\_loader)}}$
    Append `avg_epoch_loss` to `all_losses`
    Print `avg_epoch_loss`
end for
Return `all_losses`

Training Steps Explanation

- **Adding Noise to Inputs:** During training, noise is added to the inputs to simulate the denoising task. This helps the model learn how to recover the original clean input from the noisy version. The noise is applied using the function `apply_noise`.
- **Forward Pass through Encoder:** The noisy inputs are passed through the encoder to obtain the encoder outputs and latent vectors. These outputs serve as the basis for the decoder's generation of the target sequence.
- **Decoder Forward Pass:** The decoder takes the latent vectors and encoder outputs, along with the previous hidden state, and generates predictions for the target sequence. The attention mechanism in the decoder allows it to focus on relevant parts of the input sequence during this generation process.
- **Loss Calculation:** The cosine similarity loss is computed between the decoder's output and the target sequence. This loss is backpropagated to update the model's parameters.

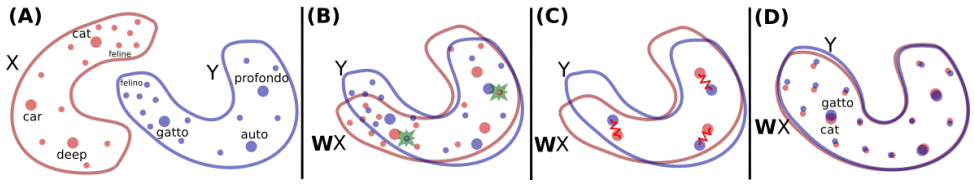

Figure 2: Word to Word Translation

- **Gradient Clipping:** To avoid exploding gradients, the gradients are clipped using `torch.nn.utils.clip_grad_norm_`.
- **Optimizer Step:** The optimizer updates the model's parameters based on the computed gradients and the selected learning rate.
- **Epoch Loss Tracking:** The loss for each batch is accumulated over the epoch, and the average loss for the epoch is computed.

Plotting the Loss

After training, the `plot_loss` function is called to visualize the loss curve over epochs. This helps in diagnosing how well the model is learning and whether any adjustments are needed in the training process.

5.6 Word-by-Word Translation Model with Adversarial Training and Procrustes Refinement

The algorithm for the word-by-word translation model consists of several steps, including adversarial training, Procrustes refinement, and computing the similarity between source and target word embeddings. Below is a step-by-step breakdown of the algorithm:

1. **Embedding Normalization:** Normalize both the source and target word embeddings to have unit length. This ensures that the cosine similarity calculation is not biased by the magnitude of the vectors.

$$X_i = \frac{X_i}{\|X_i\|}, \quad Y_j = \frac{Y_j}{\|Y_j\|}$$

   where X_i and Y_j represent the source and target embeddings, respectively.

2. **Tensor Conversion:** Convert the normalized embeddings into tensors that can be processed by the neural network during the adversarial training process.

3. **Linear Transformation Initialization:** Initialize the linear transformation matrix W as an identity matrix, which will map the source embedding space to the target embedding space.

$$W = I$$

   where I is the identity matrix.

4. **Discriminator Model:** A discriminator model is trained to distinguish between the source and target embeddings. The discriminator is a neural network with one hidden layer:

$$D(z) = \sigma(W_2 \cdot \text{ReLU}(W_1 \cdot z + b_1) + b_2)$$

where W_1, W_2 are weight matrices, and b_1, b_2 are biases. The output $D(z)$ is a probability that z comes from the target distribution.

5. **Adversarial Training Loop:** The adversarial training loop alternates between two steps:

   (a) Train the discriminator D to correctly classify the source and target embeddings.

   (b) Update the transformation matrix W to minimize the discriminator's ability to distinguish between the transformed source embeddings and the target embeddings.

   The loss function for the discriminator L_{disc} is the binary cross-entropy between the predictions and true labels:

$$L_{\text{disc}} = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log D(x_i) + (1 - y_i)\log(1 - D(x_i))\right]$$

   where y_i are the true labels (1 for target, 0 for source).

   The loss function for the transformation matrix W, used to update W, is:

$$L_{\text{map}} = -\frac{1}{N}\sum_{i=1}^{N}\log D(W(X_i))$$

6. **Procrustes Refinement:** After the adversarial training, apply Procrustes refinement to the transformation matrix W. This is done by performing Singular Value Decomposition (SVD) on the matrix $Y^T WX$ and updating W to enforce orthogonality:

$$U, S, V^T = \text{SVD}(Y^T WX)$$
$$W = UV^T$$

7. **Cosine Similarity Calculation:** After training, compute the cosine similarity between the transformed source embeddings and the target embeddings. This results in a similarity matrix:

$$S(i, j) = \frac{W(X_i) \cdot Y_j}{\|W(X_i)\|\|Y_j\|}$$

   where $S(i, j)$ is the cosine similarity between the transformed source word X_i and the target word Y_j.

8. **Word Mapping:** Finally, the word mappings between the source and target languages are obtained by selecting the most similar target word for each source word:

$$\text{source\_to\_target}(X_i) = \arg\max_j S(i, j)$$

   and vice versa for target-to-source mapping:

$$\text{target\_to\_source}(Y_j) = \arg\max_i S(i, j)$$

Summary of the Algorithm:

1. Normalize source and target embeddings.
2. Convert embeddings into tensors.
3. Initialize a transformation matrix W.
4. Train the discriminator to distinguish between source and target embeddings.
5. Perform adversarial training to update W and confuse the discriminator.
6. Refine W using Procrustes analysis.
7. Compute cosine similarity between source and target embeddings.
8. Map source words to target words and vice versa based on cosine similarity.

5.7 Cross-Domain Non-Parallel Training Algorithm

The following algorithm describes the cross-domain non-parallel training process. It uses auto-encoding, cross-domain reconstruction, and adversarial losses to align latent spaces between source and target domains.

Algorithm 15 Cross-Domain Non-Parallel Training Algorithm

Require: Encoder Encoder, Decoder Decoder, Discriminator Disc, optimizers enc_dec_optimizer, disc_optimizer, data loaders source_loader, target_loader, vocabulary mappings source_to_target, target_to_source, hyperparameters $\lambda_{\text{auto}}, \lambda_{\text{cd}}, \lambda_{\text{adv}}$, clip, number of epochs NUM_EPOCHS.

Ensure: Trained Encoder, Decoder, and Discriminator.

for epoch = 1 to NUM_EPOCHS **do**
  Initialize epoch_loss ← 0
  for all (source_batch, target_batch) in zip(source_loader, target_loader) **do**
    Step 1: Auto-Encoding Loss for Source Domain
    Add noise: noisy_source ← apply_noise(source_batch[0])
    Encode: source_enc_outputs, source_latents ← Encoder(noisy_source)
    Decode: source_decoded ← Decoder(source_latents, source_enc_outputs)
    Compute loss: loss_auto_source ← cosine_similarity_loss(source_decoded, source_batch[1])
    Step 2: Auto-Encoding Loss for Target Domain
    Add noise: noisy_target ← apply_noise(target_batch[0])
    Encode: target_enc_outputs, target_latents ← Encoder(noisy_target)
    Decode: target_decoded ← Decoder(target_latents, target_enc_outputs)
    Compute loss: loss_auto_target ← cosine_similarity_loss(target_decoded, target_batch[1])
    Step 3: Cross-Domain Reconstruction (Source → Target → Source)
    Translate: translated_to_target ← apply_word_by_word(source_batch[0], source_to_target)
    Add noise: noisy_translation ← apply_noise(translated_to_target)
    Encode: target_enc_outputs, target_latents ← Encoder(noisy_translation)
    Decode back: translated_back_to_source ← Decoder(target_latents, target_enc_outputs)
    Compute loss: loss_source_target_cd ← cosine_similarity_loss(translated_back_to_source, sourc
    Step 4: Cross-Domain Reconstruction (Target → Source → Target)
    Translate: translated_to_source ← apply_word_by_word(target_batch[0], target_to_source)
    Add noise: noisy_translation ← apply_noise(translated_to_source)
    Encode: source_enc_outputs, source_latents ← Encoder(noisy_translation)
    Decode back: translated_back_to_target ← Decoder(source_latents, source_enc_outputs)
    Compute loss: loss_target_source_cd ← cosine_similarity_loss(translated_back_to_target, targe
    Step 5: Adversarial Training of Discriminator
    Compute disc_loss_src and disc_loss_tgt for source and target latents
    Backpropagate and update: disc_optimizer.step()
    Step 6: Adversarial Loss for Encoder
    Compute adv_loss_gen to fool the discriminator
    Step 7: Total Loss and Update Encoder-Decoder
    Compute total loss:

$$\text{total\_loss} \leftarrow \lambda_{\text{auto}}(\text{loss\_auto\_source}+\text{loss\_auto\_target})+\lambda_{\text{cd}}(\text{loss\_source\_target\_cd}+\text{loss\_target\_source}$$

    Backpropagate and update: enc_dec_optimizer.step()
    Accumulate epoch loss: epoch_loss ← epoch_loss + total_loss
  end for
  Update word mappings: source_to_target, target_to_source ← update_word_by_word()
end for

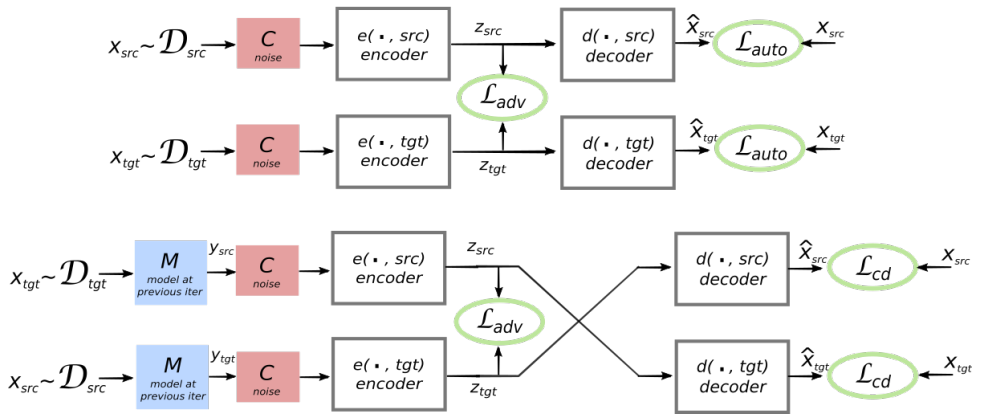

Figure 3: Training Methodology

6 Results

Using a **word-to-word parallel translation dictionary that was trained using unsupervised methods** alongside the encoder-decoder architecture improved results by explicitly modeling the relationships between words in the source and target languages.

While the traditional encoder-decoder approach focuses on encoding the input sequence into a fixed-length context vector and decoding it into the output sequence, a word-to-word translation matrix introduces an additional layer of structured knowledge. This dictionary captures direct word-level alignments between source and target languages, providing the model with finer-grained information about how words in one language map to words in another. This can help reduce ambiguities, especially in cases of words with multiple meanings or complex syntactic structures. By combining the encoder-decoder architecture with a word-to-word translation matrix, the model can leverage both the contextual understanding from the encoder and the explicit word-level correspondence from the translation matrix, leading to more accurate and fluent translations.

6.1 Translation Metrics and Samples

These metrics were tested on first **100** samples from the parallel datasets, and cross models were trained **only on 200 sentences** from different monolingual data sources with **20 epochs** due to system memory limitations.

6.2 English-French Model

English to French Translation Metric:

- **BLEU Score:** [6.0642144388591694e-232]
- **TER Score:** [2656.28]

Sample Translation:

- **Input Sentence (English):** [SAN FRANCISCO – It has never been easy to have a rational conversation about the value of gold.]

- **Translated Sentence (French):** [Lord Lord Lordigival Lordincarnear Lord Ronde Lordushi lear leincarnecollabo LordushiLMvalval LordushiMS Lordval Lordushi roux LordAgnaissanceushi Lordministre Lord Lordushi leushiar Lordval Lordushi LordagueUD Lord]
- **Actual Reference Sentence (French):** [SAN FRANCISCO – Il n'a jamais été facile d'avoir une discussion rationnelle sur la valeur du métal jaune.]

French to English Translation Metric:

- **BLEU Score:** [2.877875078311163e-232]
- **TER Score:** [3863.56]

Sample Translation:

- **Input Sentence (French):** [Et aujourd'hui, alors que le cours de l'or a augmenté de 300 pour cent au cours de la dernière décennie, c'est plus difficile que jamais.]
- **Translated Sentence (English):** [presidency dozenneasdilly rosalie presidency winneas¤ rosalieneasdilly rosalie winneas walk insideneas rosalie understandneas-neasneas rosalie inside rosaliegold win neighbourhood subsided rosaliencies within¤neasnciesneas neighbourhood rosalie within rosalieneasneas neighbourhood walkneas presidency when¤ explain]
- **Actual Reference Sentence (English):** [Lately, with gold prices up more than 300% over the last decade, it is harder than ever.]

6.3 English-German Model

English to German Translation Metric:

- **BLEU Score:** [0]
- **TER Score:** [4287.20]

Sample Translation:

- **Input Sentence (English):** [SAN FRANCISCO – It has never been easy to have a rational conversation about the value of gold.]
- **Translated Sentence (German):** [Jan Sees Seeseiche schwäch schwächbachs könnten Sees Seesaun Seesverantwortantwort nächste Leistantwort Sees Jankönnenaun Sees LeistWirtschaft schwäch Seesantwortaufgabe schwächaun schwäch schwächaun schwäch schwäch Sees Bescheinigung Seesantwort Sees Sees JanRegaun schwäch Leist Creek schwäch Sees Sees]
- **Actual Reference Sentence (German):** [SAN FRANCISCO – Es war noch nie leicht, ein rationales Gespräch über den Wert von Gold zu führen.]

German to English Translation Metric:

- **BLEU Score:** [4.003310835127927e-232]
- **TER Score:** [5049.64]

Sample Translation:

- **Input Sentence (German):** [SAN FRANCISCO – Es war noch nie leicht, ein rationales Gespräch über den Wert von Gold zu führen.]
- **Translated Sentence (English):** [since operation peace hold towards mama government shaking operation shaking used lady mama contact wan there mama sample mama mamawash mama mama mama mamamb fingerprints peters lakemb sincemb mama peters peters holdmb hold there mama mama mama collection theremb mama since collection wan]
- **Actual Reference Sentence (English):** [SAN FRANCISCO – It has never been easy to have a rational conversation about the value of gold.]

French-German Model

French to German Translation Metric:

- **BLEU Score:** [0]
- **TER Score:** [5313.48]

Sample Translation:

- **Input Sentence (French):** [SAN FRANCISCO – Il n'a jamais été facile d'avoir une discussion rationnelle sur la valeur du métal jaune.]
- **Translated Sentence (German):** [Attentat Kriegs Kriegs Kriegs Kriegs Attentat widersp Kriegses samt mußte Kriegs vernichtet Denkmal Denkmalhelm Einladung einges Einladung Kriegs entstamm entsprach Zelt Kriegsfallenden mußte Kriegs nämlich Denkmal nämlich Kriegs Kriegs Kriegs Kriegs Kriegs Schiffs Einladung neues Kriegs Kriegs Denkmal entsprachleib samt etwa samt Einladung Kriegs etwa entstamm]
- **Actual Reference Sentence (German):** [SAN FRANCISCO – Es war noch nie leicht, ein rationales Gespräch über den Wert von Gold zu führen.]

German to French Translation Metric:

- **BLEU Score:** [2.6270262679228365e-232]
- **TER Score:** [5477.43]

Sample Translation:

- **Input Sentence (German):** [SAN FRANCISCO – Es war noch nie leicht, ein rationales Gespräch über den Wert von Gold zu führen.]

- **Translated Sentence (French):** [ailleurs ailleurs vêtu Pareil inversement falloir Thu“ look MO Pareil ailleursvaitoût ailleurs ailleurs ailleurs ailleurs automatiquement historique ailleursoût ailleurs ailleurs historique ailleursoût semblant Autrefois 2.0 Autrefois“oût ailleurs Pareil Schw MO ailleurs inversement MO Pareil 2.0 Pareil dénommé inversement ailleurs inversement Pareil“ ailleurs]
- **Actual Reference Sentence (French):** [SAN FRANCISCO – Il n'a jamais été facile d'avoir une discussion rationnelle sur la valeur du métal jaune.]

Results observed were not good, most likely because of training limiatations of the system. But these scores could be improved further by training these cross domain models on a larger dataset.

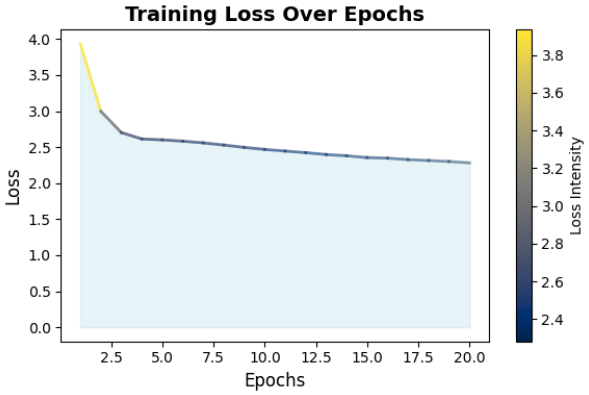

Figure 4: Cross Train Loss

As could be observed from train loss graph, loss is not decreasing at a faster rate because of less number of train samples.

7 Command Line Interface (CLI)

The CLI tool has three main features described below:

- The CLI tool can train a new translation model using two independent monolingual corpora and save the trained encoder and decoder models.
- It can translate text or a single sentence between the source and target languages using the trained encoder and decoder models as input.
- The tool can evaluate the trained model using parallel datasets and display metrics such as BLEU and TER scores.

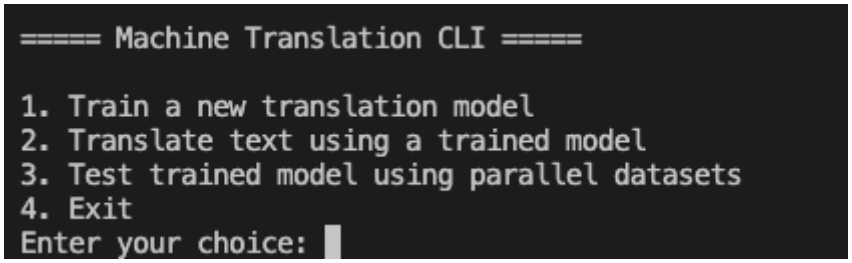

Figure 5: CLI Menu

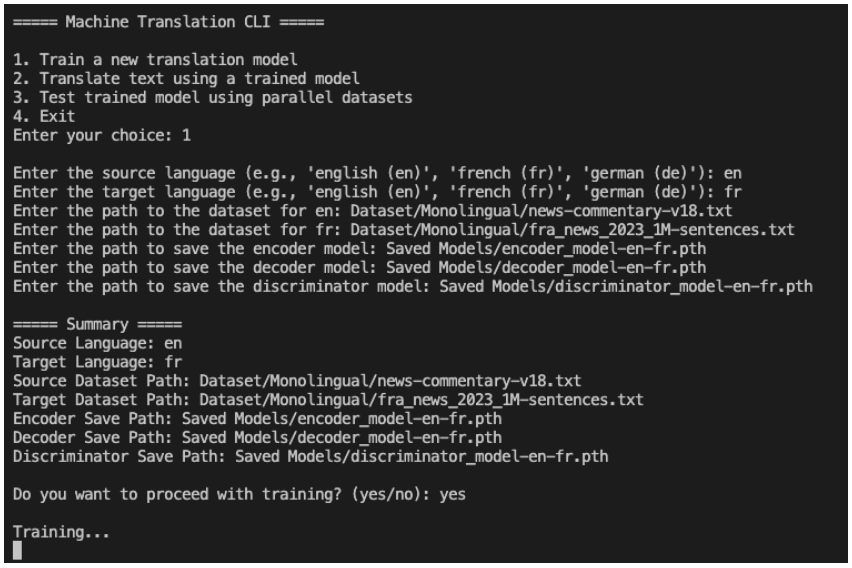

Figure 6: Train Model

8 References

[Con+18] [Lam+18] [BCB16]

References

[BCB16] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2016. arXiv: 1409.0473 [cs.CL]. URL: https://arxiv.org/abs/1409.0473.

[Con+18] Alexis Conneau et al. *Word Translation Without Parallel Data*. 2018. arXiv: 1710.04087 [cs.CL]. URL: https://arxiv.org/abs/1710.04087.

[Lam+18] Guillaume Lample et al. *Unsupervised Machine Translation Using Monolingual Corpora Only*. 2018. arXiv: 1711.00043 [cs.CL]. URL: https://arxiv.org/abs/1711.00043.