Problems L4

Fundamentals of Natural Language

Authors:

```
Sonia Espinilla, 1708919
Agustina Inés Lazzati, 1707964
```

Autonomous University of Barcelona

March 14, 2025

Graded Exercise 1 (6 points)

We will implement a spelling corrector.

1. First, use the Gutenberg corpus from the NLTK package. Using the sents() function of the corpus, take 50 sentences as test and the rest for training. In these 50 test sentences, replace one word for a random in-vocabulary word and save the original sentence for comparison. Show a selection of 5 of these in the final report. You can also try to be a bit deliberate with your choice of random words to assess complex scenarios.

```
!pip install nltk
```

Mostrar salida oculta

```
import nltk
from nltk.corpus import gutenberg as corpus # import the gutenberg corpus and assign it to the variable corpus

nltk.download('gutenberg')
nltk.download('punkt_tab')
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger_eng')
```

Mostrar salida oculta

```
#FUNCTIONS THAT WE WILL USE IN 1.1 EXERCISE PROVIDED IN CAMPUS
import numpy as np
from tqdm.auto import tqdm
from nltk.lm.api import LanguageModel
from nltk.metrics.distance import edit_distance
from typing import List, Dict

def levenshtein(s1: str, s2: str):
    return edit_distance(s1, s2, substitution_cost=1, transpositions=True)

def compute_close_words(vocab: List[str], max_dist: int = 1):
    vocab = np.array(vocab)
    word_lengths = np.array([len(w) for w in vocab])
    dict_lengths = {}
    for l in range(min(word_lengths), max(word_lengths)+1):
        dict_lengths[l] = vocab[word_lengths==l] # needs vocabulary to be a numpy array
    min_length = min(dict_lengths.keys())
    max_length = max(dict_lengths.keys())
    close_words = {}

    for word in tqdm(vocab):
        length = len(word)
        candidate_words = []
        d1 = max(min_length, length - max_dist)
        d2 = min(max_length, length + max_dist)

        for d in range(d1, d2 + 1):
            candidate_words.extend(dict_lengths[d])
            close_words[word] = [w for w in candidate_words if levenshtein(word,w) <= max_dist]
    return close_words


# AGAIN TAKEN FROM CAMPUS
import re
```

```
CLEANUP_REGEXES = [
    (re.compile(r"[0-9]"), ""),  # Remove digits
    (
        re.compile(r"[_*-]?([A-Za-z]+)[_*-]?"),
        lambda match: match.group(1),
    ),  # Remove bold and italics
    (re.compile(r"[\-_!?*&.]"), ""),  # Remove punctuation and extra symbols
]

def preprocess_words(word: str) -> str | None:
    word = word.lower()
    word = word.strip()
    for regex, repl in CLEANUP_REGEXES:
        word = regex.sub(repl, word)
    if len(word) == 0:
        return None
    return word


import random
sentences = corpus.sents()
words = corpus.words()

# This piece of code will convert sentences to lowercase and preprocess words (provided in campus)
text = [
    list(filter(lambda x: x is not None, map(preprocess_words, x)))
    for x in corpus.sents()[:1000]
]

random.shuffle(text) # This is important so that we dont take the first sentences since context may be

# Split into training and testing
test_sentences = text[:50]
train_sentences = text[50:]

# We create vocabulary and compute close words
vocab = list(sorted(set([word for sent in text for word in sent])))
close_words = compute_close_words(vocab, max_dist=1)
```

```
100%                                        2550/2550 [00:47<00:00, 15.10it/s]
```

```
# Now we are going to replace ONE WORD in EACH sentence with a similar word
def replace_random_word(sentences: List[List[str]]):
    sentences_modified = []
    for sentence in sentences:

        s_modified = sentence.copy()
        replace_idx = random.randint(0, len(sentence) - 1)  # Select RANDOM index
        '''
        we have noticed that some words didn't have close words to change for
        so, we created an algorithm that traverse the whole sentence to pick
        a valid word.
        '''
        # Start the loop to replace a word in the sentence
        attempts = 0
        while attempts < len(sentence):
            original_word = sentence[replace_idx]

            # Check if this word has close words
            if original_word in close_words and close_words[original_word]:
                random_word = random.choice(close_words[original_word])  # RANDOM similar word

                # If the word picked is not the same as the original we replace it
                if random_word != original_word:
                    s_modified[replace_idx] = random_word
                    break

            # If no replacement was made we move to the next word
            replace_idx += 1
            if replace_idx >= len(sentence):
                replace_idx = 0  # If we reach the end of the sentence
            attempts += 1

        sentences_modified.append(s_modified)
    return sentences_modified

# Perform replacement
test_sentences_modified = replace_random_word(test_sentences)

# Print 5 examples for evaluation
for i in range(5):
    print(f"Original: {' '.join(test_sentences[i])}")
    print(f"Modified: {' '.join(test_sentences_modified[i])}")
    print()
```

```
Original: you might not give emma such a complete education as your powers would seem to promise ; but you were receiving a very goo
Modified: you might not give emma much a complete education as your powers would seem to promise ; but you were receiving a very goo

Original: " not harriet ' s equal "
Modified: " not harriet ' so equal "

Original: those soft blue eyes , and all those natural graces , should not be wasted on the inferior society of highbury and its cor
Modified: those soft blue eyes , and ill those natural graces , should not be wasted on the inferior society of highbury and its cor

Original: it was impossible to say how much he should be gratified by being employed on such an errand "
Modified: wit was impossible to say how much he should be gratified by being employed on such an errand "

Original: but from one cause or another , i gave it up in disgust
Modified: but from one cause or another , i have it up in disgust
```

2. ***Implement and train an n-gram language model using the NLTK package (check the documentation here). Try using bigrams and trigrams and the variations seen in class.***

```
from nltk.lm.preprocessing import padded_everygram_pipeline
from nltk.lm.models import MLE, StupidBackoff, Laplace
from nltk.lm import Vocabulary
from nltk.tokenize import word_tokenize

# corpus.sents() already tokenize the sentences into word-level
n_bigram = 2
n_trigram = 3

train_data_bigram_gen, vocab_digram = padded_everygram_pipeline(n_bigram, train_sentences)
train_data_trigram_gen, vocab_trigram = padded_everygram_pipeline(n_trigram, train_sentences)

# Convert generators to lists so they can be reused
train_data_bigram = list(train_data_bigram_gen)
train_data_trigram = list(train_data_trigram_gen)

# Convert vocab_digram and vocab_trigram to Vocabulary objects
vocab_bigram = Vocabulary(vocab_digram)
vocab_bigram.update(["<unk>"])
vocab_trigram = Vocabulary(vocab_trigram)
vocab_trigram.update(["<unk>"])

print("Vocabulary Size for Bigram:", len(vocab_bigram))
print("Vocabulary Size for Trigram:", len(vocab_trigram))

# Print out some sample bigrams and trigrams to check its working
sample_bigrams = [list(bigram) for bigram in train_data_bigram[:2]]
print("Sample Bigrams (with padding):", sample_bigrams)
sample_trigrams = [list(trigram) for trigram in train_data_trigram[:2]]
print("Sample Trigrams (with padding):", sample_trigrams)
```

```
Vocabulary Size for Bigram: 2491
Vocabulary Size for Trigram: 2491
Sample Bigrams (with padding): [[('<s>',), ('<s>', 'even'), ('even',), ('even', 'before'), ('before',), ('before', 'miss'), ('miss',
Sample Trigrams (with padding): [[('<s>',), ('<s>', '<s>'), ('<s>', '<s>', 'even'), ('<s>',), ('<s>', 'even'), ('<s>', 'even', 'befo
```

```
# Train the different models in bigram and trigram
# BIGRAMS

train_data_bigram, vocab_bigram = padded_everygram_pipeline(n_bigram, train_sentences)
vocab_bigram = Vocabulary(vocab_bigram)
vocab_bigram.update(["<unk>"])
lm_mle_bigram = MLE(n_bigram)
lm_mle_bigram.fit(train_data_bigram, vocab_bigram)

train_data_bigram, vocab_bigram = padded_everygram_pipeline(n_bigram, train_sentences)
vocab_bigram = Vocabulary(vocab_bigram)
vocab_bigram.update(["<unk>"])
lm_lpc_bigram = Laplace(n_bigram, vocabulary=vocab_bigram)
lm_lpc_bigram.fit(train_data_bigram, vocab_bigram)

train_data_bigram, vocab_bigram = padded_everygram_pipeline(n_bigram, train_sentences)
vocab_bigram = Vocabulary(vocab_bigram)
```

```
vocab_bigram.update(["<unk>"])
lm_sbo_bigram = StupidBackoff(alpha=0.4, order=n_bigram)
lm_sbo_bigram.fit(train_data_bigram, vocab_digram)

# TRIGRAM
train_data_trigram, vocab_trigram = padded_everygram_pipeline(n_trigram, train_sentences)
vocab_trigram = Vocabulary(vocab_trigram)
vocab_trigram.update(["<unk>"])
lm_mle_trigram = MLE(n_trigram)
lm_mle_trigram.fit(train_data_trigram, vocab_trigram)

train_data_trigram, vocab_trigram = padded_everygram_pipeline(n_trigram, train_sentences)
vocab_trigram = Vocabulary(vocab_trigram)
vocab_trigram.update(["<unk>"])
lm_lpc_trigram = Laplace(n_trigram, vocabulary=vocab_trigram)
lm_lpc_trigram.fit(train_data_trigram, vocab_trigram)

train_data_trigram, vocab_trigram = padded_everygram_pipeline(n_trigram, train_sentences)
vocab_trigram = Vocabulary(vocab_trigram)
vocab_trigram.update(["<unk>"])
lm_sbo_trigram = StupidBackoff(alpha=0.4, order=n_trigram)
lm_sbo_trigram.fit(train_data_trigram, vocab_trigram)
```

We saw that our generator `padded_everygram_pipeline` got exhausted when training a new model, so we needed to recreate this generator again before training a new model

```
# Generate words using the generate function
n_words = 10
# MLE
mle_bigram = lm_mle_bigram.generate(n_words, text_seed='<s>', random_seed=42)
print("MLE Bigram:",(mle_bigram))
mle_trigram = lm_mle_trigram.generate(n_words, text_seed=['<s>', '<s>'], random_seed=42)
print("MLE Trigram:",(mle_trigram))

# Laplace
laplace_bigram = lm_lpc_bigram.generate(n_words, text_seed='<s>', random_seed=42)
print("Laplace Bigram:", (laplace_bigram))
laplace_trigram = lm_lpc_trigram.generate(n_words, text_seed=['<s>', '<s>'], random_seed=42)
print("Laplace Trigram:", (laplace_trigram))

# StupidBackoff
sbo_bigram = lm_sbo_bigram.generate(n_words, text_seed='<s>', random_seed=42)
print("StupidBackoff Bigram:",(sbo_bigram))
sbo_trigram = lm_sbo_trigram.generate(n_words, text_seed=['<s>', '<s>'], random_seed=42)
print("StupidBackoff Trigram:",(sbo_trigram))
```

```
MLE Bigram: ['never', 'achieved', '</s>', 'an', 'interest', 'in', 'the', 'boy', 'another', '"']
MLE Trigram: ['miss', 'bates', ',', 'let', 'them', 'run', 'about', 'a', 'great', 'comfort']
Laplace Bigram: ['no', ',', 'both', 'a', 'single', 'man', 'thought', 'but', 'in', 'a']
Laplace Trigram: ['mr', 'elton', ',', 'a', 'son', 'and', 'the', 'bride', 'people', 'gone']
StupidBackoff Bigram: ['<UNK>', '<UNK>', '<UNK>', '<UNK>', '<UNK>', '<UNK>', '<UNK>', '<UNK>', '<UNK>', '<UNK>']
StupidBackoff Trigram: ['miss', 'bates', ',', 'let', 'them', 'run', 'about', 'a', 'great', 'comfort']
```

3. ***Try sampling (generating sentences) from the various language models. Which combination seems better? Aid yourself with the test sentences you left earlier that do not contain errors and evaluate the perplexity of the model. Show which model has better perplexity in the report and explain why you think it is the case providing examples.***

```
# BIGRAM
example_sent = test_sentences[1][:2]
print("Test sentence:", example_sent)

# Function to replace the words that arent in the vocabulary with <unk>
def handle_UNKNOW(sentence, vocab):
    return [word if word in vocab else "<unk>" for word in sentence]

example = handle_UNKNOW(example_sent, vocab_bigram)

mle_per_bigram = lm_mle_bigram.perplexity(example)
print(f"Perplexity (MLE Bigram): {mle_per_bigram:.2f}")
mle_lpc_bigram = lm_lpc_bigram.perplexity(example)
print(f"Perplexity (LPC Bigram): {mle_lpc_bigram:.2f}")
mle_sbo_bigram = lm_sbo_bigram.perplexity(example)
print(f"Perplexity (SBO Bigram): {mle_sbo_bigram:.2f}")

example = handle_UNKNOW(example_sent, vocab_trigram)
```

```
# TRIGRAM
mle_per_trigram = lm_mle_trigram.perplexity(example)
print(f"Perplexity (MLE Trigram): {mle_per_trigram:.2f}")
mle_lpc_trigram = lm_lpc_trigram.perplexity(example)
print(f"Perplexity (LPC Trigram): {mle_lpc_trigram:.2f}")
example_sent = test_sentences[1][:2]

example = handle_UNKNOW(example_sent, vocab_bigram)
mle_sbo_trigram = lm_sbo_trigram.perplexity(example)
print(f"Perplexity (SBO Trigram): {mle_sbo_trigram:.2f}")
```

```
Test sentence: ['"', 'not']
Perplexity (MLE Bigram): inf
Perplexity (LPC Bigram): 379.45
Perplexity (SBO Bigram): 9.88
Perplexity (MLE Trigram): inf
Perplexity (LPC Trigram): 393.37
Perplexity (SBO Trigram): 2964.11
```

The issue with infinite perplexity (inf) is that your MLE (Maximum Likelihood Estimation) and Stupid Backoff (SBO) models encounter unknown words or unseen n-grams in the test data, which leads to zero probability for some words. Since perplexity involves division by probability, zero probability causes an infinite perplexity. We tried to prevent this issue by using a function that will change this complex" words by `"UNK"`.

We ended up with the following results:

- MLE (Maximum Likelihood Estimation) fails for both bigrams and trigrams. We believe that this happens because the test sentence contains unseen words or n-grams, and MLE assigns them zero probability.
- Laplace smoothing (LPC) helps by assigning small probabilities to unseen n-grams, reducing perplexity to reasonable values.
- Stupid Backoff (SBO) works well for bigrams (9.88 perplexity), meaning it generalizes well for shorter contexts. However, SBO struggles with trigrams (11813.12 perplexity), suggesting that longer contexts cause extreme probability shifts.

---

4. ***The spelling corrector we want to implement follows the Noisy Channel Model. The spelled sentence $X = x_1 \ldots x_n$ is an altered version of the intended sentence $W = w_1 \ldots w_n$ passing through a noisy channel.***

   Given a set of candidate sentences $C(X)$ that can correct sentence X, our goal is to find the one that maximises the probability $P(W|X)$. Unfortunately, we do not have a corpus that allows us to estimate the probability of X, but we can apply Bayes to obtain this from the reverse conditioning:

   $W^* = \arg\max_{W \in C(X)} P(W|X) = \arg\max_{W \in C(X)} P(X|W)P(W)/P(X)$

   Since we are optimising for W, $P(X)$ will be constant for all computations, so we can disregard it from the equation, obtaining

   $W^* = \arg\max_{W \in C(X)} P(X|W)P(W)$

   In order to obtain $P(W)$, we can use a language model. Using the chain rule and the Markov assumption in a bigram, for instance, we can compute this by

   $P(w_1 \ldots w_k) = \prod_{t=2}^{k} P(w_t|w_{t-1})$

   In order to obtain $P(X|W)$ a simplifying assumption can be made. We can set a fixed $0 \leq \alpha \leq 1$ that signifies the probability of writing x_i instead of w_i. Given that we assume the lengths of X and W are the same and that each word is spelled independently, we obtain:

   Which is the same as saying that we assume the text word is the most likely and all possible substitutions of this word get a slice of the remaining probability. The likelihood for the full sequence is then obtained by

   $P(X|W) = \prod_{i=1}^{k} P(x_i|w_i)$.

   For the sake of simplicity, we will assume only one word changes per sentence and we will assume substitutions are performed only on words that are at a limited levenshteindistance of 1.

```
import numpy as np
from tqdm.auto import tqdm

def candidates(sentence: List[str], close_words: Dict[str, List[str]], vocab: List[str]):
    sent_x = sentence
    candidates = [sent_x]
    #first close_words
    for ii, word_x in enumerate(sent_x):
        if word_x in close_words:
            for cand_w in close_words[word_x]:
                if cand_w != word_x:
                    cand_sent = sent_x.copy()
                    cand_sent[ii] = cand_w
                    candidates.append(cand_sent)
    #if not work,back to vocab
    if len(candidates) == 1:  #original sentences exists?
        for ii, word_x in enumerate(sent_x):
            for cand_w in vocab:
                if cand_w != word_x:
                    cand_sent = sent_x.copy()
                    cand_sent[ii] = cand_w
                    candidates.append(cand_sent)

    return candidates

# This function computes the log prior for a sentence
def log_prior(sentence_w: List[str], lm: LanguageModel):
    sentence_padded = ['<s>', '<s>'] + sentence_w + ['</s>']
    num_words = len(sentence_padded)
    log_prior_w = 0.0

    for i in range(2, num_words - 1):  # omit </s> because likelihoods don't have it
        # Uses trigrams, adapt it if you change the n
        score = lm.logscore(sentence_padded[i], [sentence_padded[i - 2], sentence_padded[i - 1]])
        #without this doesn't work

        if score == float('-inf'):  # Handle zero probability
            score = np.log(1e-10)  # Assign a small probability

        log_prior_w += score
    return log_prior_w
```

Implement the missing parts of the spell corrector. Evaluate it on the sentences from the test set and reflect on the situations on which it works well and those it does not.

```
# This function computes the likelihood of a sentence given the noisy channel assumption
def log_likelihood(sentence_x: List[str], sentence_w: List[str], alpha: float = 0.95):
    log_likelihood_w = 0.0
    for x_i, w_i in zip(sentence_x, sentence_w):
        if x_i == w_i:
            log_likelihood_w += np.log(alpha)
        else:
            edit_dist = edit_distance(x_i, w_i)
            prob = (1 - alpha) * np.exp(-edit_dist)
            log_likelihood_w += np.log(prob)
    return log_likelihood_w

# This function finds the best candidate sentence using the Noisy Channel Model
def best_candidate(sentence: List[str], lm: LanguageModel, close_words: Dict[str, List[str]], vocab: List[str], alpha: float):
    candidate_sentences = candidates(sentence, close_words, vocab)
    best_sentence = None
    best_score = float('-inf')

    for sentence_w in candidate_sentences:
        prior = log_prior(sentence_w, lm)  # Compute P(W) using the language model
        likelihood = log_likelihood(sentence, sentence_w, alpha)  # Compute P(X|W)
        score = prior + likelihood  # Compute P(W|X) ~ P(X|W) P(W)

        if score > best_score:
            best_sentence = sentence_w
            best_score = score

    return best_sentence  # Return the best candidate sentence



test_sentences_corrected = []

for sentence in test_sentences_modified:
  sentence
  corrected = best_candidate(sentence, lm_mle_trigram, close_words, vocab, alpha= 0.1)
  test_sentences_corrected.append(corrected)

# Evaluate spell corrector
for i in range(5):
  print("Original: ", " ".join(test_sentences[i]))
  print("Modified: ", " ".join(test_sentences_modified[i]))
  print("Corrected:", " ".join(test_sentences_corrected[i]))
  print()
```

```
Original:  you might not give emma such a complete education as your powers would seem to promise ; but you were receiving a very gc
Modified:  you might not give emma much a complete education as your powers would seem to promise ; but you were receiving a very gc
Corrected: you might not give emma much a complete education as your powers would seem to promise ; but you were receiving a very gc
```

```
Original:  " not harriet ' s equal "
Modified:  " not harriet ' so equal "
Corrected: " not harriet ' s equal "

Original:  those soft blue eyes , and all those natural graces , should not be wasted on the inferior society of highbury and its co
Modified:  those soft blue eyes , and ill those natural graces , should not be wasted on the inferior society of highbury and its co
Corrected: those soft blue eyes , and ill those natural graces i should not be wasted on the inferior society of highbury and its co

Original:  it was impossible to say how much he should be gratified by being employed on such an errand "
Modified:  wit was impossible to say how much he should be gratified by being employed on such an errand "
Corrected: it was impossible to say how much he should be gratified by being employed on such an errand "

Original:  but from one cause or another , i gave it up in disgust
Modified:  but from one cause or another , i have it up in disgust
Corrected: but from one cause or another ; i have it up in disgust
```

```python
from sklearn.metrics import accuracy_score

def accuracy_corrector(test_sentences, test_sentences_corrected):
    corrected_sentences = [" ".join(test_sentences_corrected) for test_sentences_corrected in test_sentences_corrected]
    original_sentences = [" ".join(sentence) for sentence in test_sentences]

    # Word-level accuracy
    correct_words = sum(
        orig_word == corr_word
        for orig_sent, corr_sent in zip(original_sentences, corrected_sentences)
        for orig_word, corr_word in zip(orig_sent.split(), corr_sent.split())
    )
    total_words = sum(len(sentence.split()) for sentence in original_sentences)

    sentence_accuracy = accuracy_score(original_sentences, corrected_sentences) * 100
    word_accuracy = (correct_words / total_words) * 100 if total_words else 0

    print(f"Sentence-Level Accuracy: {sentence_accuracy:.2f}%")
    print(f"Word-Level Accuracy: {word_accuracy:.2f}%")

    return sentence_accuracy, word_accuracy

accuracy_log = accuracy_corrector(test_sentences, test_sentences_corrected)
```

```
Sentence-Level Accuracy: 36.00%
Word-Level Accuracy: 94.06%
```

As we are calculating the accuracy of the spell corrector in correctly correcting the modified word, when calculating the word-level accuracy, the value will always be high since, at most, only two words will be changed, remaining the rest equal.

Graded Exercise 2 (3 points)

In the following code, you can see the implementation of the neural-based LM from Bengio et al.

```python
import torch
import torch.nn as nn

class NNLM(nn.Module):
    def __init__(self, num_classes, dim_input, dim_hidden, dim_embedding):
        super().__init__()
        self.num_classes = num_classes
        self.dim_input = dim_input
        self.dim_hidden = dim_hidden
        self.dim_embedding = dim_embedding
        self.embeddings = nn.Embedding(
            self.num_classes, self.dim_embedding
        )  # embedding layer or look up table
        self.hidden1 = nn.Linear(
            self.dim_input * self.dim_embedding, self.dim_hidden, bias=False
        )
        self.ones = nn.Parameter(torch.ones(self.dim_hidden))
        self.hidden2 = nn.Linear(self.dim_hidden, self.num_classes, bias=False)
        self.hidden3 = nn.Linear(
            self.dim_input * self.dim_embedding, self.num_classes, bias=False
        )  # final layer
        self.bias = nn.Parameter(torch.ones(self.num_classes))

    def forward(self, X):
        word_embeds = self.embeddings(X)
        X = word_embeds.view(-1, self.dim_input * self.dim_embedding)  # first layer
        tanh = torch.tanh(self.ones + self.hidden1(X))  # tanh layer
        output = (
            self.bias + self.hidden3(X) + self.hidden2(tanh)
        )  # summing up all the layers with bias
        return output
```

Change the code of the spell corrector to use this model. Show the final code in your deliverable and use the same test scenarios as before to assess whether it works.

In order to implement this, you will need to write a PyTorch DataLoader.

```
from torch.utils.data import Dataset, DataLoader # Import the Dataset class from torch.utils.data
class FixedWindow(Dataset):
    def __init__(self, words, length_window):
        super().__init__()
        self.length_window = length_window
        self.words = [word for sentence in test_sentences for word in sentence]
        self.length_window = length_window
        self.word2id = {'<pad>': 0, '<unk>': 1}  # Add '<pad>' token with ID 0 and '<unk>' with ID 1

        for idx, word in enumerate(set(self.words), 2):  # Start from index 2
            self.word2id[word] = idx
        self.id2word = {idx: word for word, idx in self.word2id.items()}
        self.vocabulary_size = len(self.word2id)

        self.ids = [self.word2id[word] for word in self.words if word in self.word2id]

    def __len__(self):
        return len(self.ids) - self.length_window

    def __getitem__(self, idx):
        context = self.words[idx:idx + self.length_window - 1]
        target = self.words[idx + self.length_window - 1]
        context_ids = [self.word2id[word] for word in context if word in self.word2id]
        target_id = self.word2id.get(target, self.word2id['<unk>'])  # Use '<unk>' id if the target word is not in vocabulary # Use '<ur
        return torch.tensor(context_ids, dtype=torch.long), torch.tensor(target_id, dtype=torch.long)


import torch.optim as optim # Import the optim module
length_window = 5 # Limit total words
dataset = FixedWindow(test_sentences, length_window)
print(f"Total training samples: {len(dataset)}")

batch_size = 1000  # 5 to debug
dataloader = DataLoader(
    dataset, batch_size=batch_size, shuffle=False
)  # shuffle=False to debug


if True:
    for nbatch, (X, y) in enumerate(dataloader):
        print("batch {}".format(nbatch))
        print("X = {}".format(X))
        print("y = {}".format(y))
        for x, z in zip(X.numpy(), y.numpy()):
            print([dataset.id2word[w] for w in x], end=" ")
            print(dataset.id2word[z])
        if nbatch == 3:
            break


num_classes = dataset.vocabulary_size
dim_input = length_window - 1
dim_hidden = 50
dim_embedding = 32
learning_rate = 1e-3
num_epochs = 60

model = NNLM(num_classes, dim_input, dim_hidden, dim_embedding)

loss_fn = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

path = "NNLM.pt"
do_train = True
do_test = True

# Use Colab for this
```

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
model = model.to(device)

if do_train:
    size = len(dataloader.dataset)
    for epoch in range(num_epochs):
        for batch, (X, y) in enumerate(dataloader):
            X, y = X.to(device), y.to(device)
            pred = model(X)
            loss = loss_fn(pred, y)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            if batch % 100 == 0:
                loss, current = loss.item(), batch * batch_size
                print(
                    "Epoch {} loss: {:>7f}  [{:>5d}/{:>5d}]".format(
                        epoch + 1, loss, current, size
                    )
                )

    torch.save({"model_state_dict": model.state_dict()}, path)
else:
    checkpoint = torch.load(path)
    model.load_state_dict(checkpoint["model_state_dict"])
```

Mostrar salida oculta

We use the padding token pad. We use them because is a easy form to work with neural network that need a fixed number of length of the inputs. So we add a special token (pad) to the shorter sequences to make them the same length as the longest sequence. So for example:

"I love cats (pad) (pad)"

"Dogs are great pets"

To have the same length that is 5.

We have added the pad on the definition of Fixed Window to everytime that we used, we make it start a again from 0 in each sentence.

We also use another token, unk, that is used in predicted word ID (predicted_id) back into the actual word using a dictionary called id2word. If the predicted_id doesn't exist in the dictionary, it returns the token (which stands for "unknown").

```
#ALTERNATIVE FOR CALCULATING
def calculate_accuracy(original_sentences, corrected_sentences):
    total_words = 0
    correct_words = 0

    for original, corrected in zip(original_sentences, corrected_sentences):
        # Compare each word
        for orig_word, corr_word in zip(original, corrected):
            total_words += 1
            if orig_word == corr_word:
                correct_words += 1

    # Calculate accuracy
    accuracy = (correct_words / total_words) * 100
    return accuracy


# Training Loop
for epoch in range(num_epochs):
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)
        pred = model(X)
        loss = loss_fn(pred, y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

def replace_modified(test_sentences_modified, model, close_words, dataset, device):
    corrected_sentences = []
    model.eval()  # Set the model to evaluation mode

    for sentence in test_sentences_modified:
        # Calculate the probability of each word in the sentence
        word_probabilities = []
        for i in range(len(sentence)):
            # Get the context words (previous words in the sentence)
            context = sentence[max(0, i - (length_window - 1)):i]
            context_ids = [dataset.word2id.get(word, 0) for word in context]  # Use 0 for unknown words
            # Pad the context if necessary
            if len(context_ids) < length_window - 1:
                context_ids = [0] * (length_window - 1 - len(context_ids)) + context_ids

            # Convert context to tensor and move to the appropriate device
            context_tensor = torch.tensor([context_ids], dtype=torch.long).to(device)

            # Get the model's prediction
            with torch.no_grad():
                output = model(context_tensor)
                probabilities = torch.softmax(output, dim=1)  # Get the probability distribution

            # Get the probability of the current word
            current_word_id = dataset.word2id.get(sentence[i], 0)  # Use 0 for unknown words
            current_word_prob = probabilities[0, current_word_id].item()
            word_probabilities.append((i, current_word_prob))

        # Find the word with the lowest probability
        min_prob_index = min(word_probabilities, key=lambda x: x[1])[0]

        # Replace the word with the lowest probability
        corrected_sentence = sentence.copy()
        if sentence[min_prob_index] in close_words:
            # Replace with the first close word (or you can choose randomly or based on some other criteria)
            corrected_sentence[min_prob_index] = close_words[sentence[min_prob_index]][0]
        else:
            # If no close word is available, use the model's predicted word
            context = sentence[max(0, min_prob_index - (length_window - 1)):min_prob_index]
            context_ids = [dataset.word2id.get(word, 0) for word in context]
            if len(context_ids) < length_window - 1:
                context_ids = [0] * (length_window - 1 - len(context_ids)) + context_ids
            context_tensor = torch.tensor([context_ids], dtype=torch.long).to(device)
            with torch.no_grad():
                output = model(context_tensor)
                predicted_id = torch.argmax(output, dim=1).item()
            predicted_word = dataset.id2word.get(predicted_id, sentence[min_prob_index])  # Fallback to original word
            corrected_sentence[min_prob_index] = predicted_word

        corrected_sentences.append(corrected_sentence)

    return corrected_sentences
# Apply the model on test data
test_sentence_corrected = replace_modified(test_sentences_modified, model, close_words, dataset, device)

# Output Evaluation
for i in range(5):
    print(f"Original: {' '.join(test_sentences[i])}")
    print(f"Modified: {' '.join(test_sentences_modified[i])}")
    print(f"Corrected: {' '.join(test_sentence_corrected[i])}")
    accuracy = calculate_accuracy(test_sentences[i], test_sentence_corrected[i])
    print(f"Accuracy: {accuracy:.2f}%")
    print()
```

```
Original: you might not give emma such a complete education as your powers would seem to promise ; but you were receiving a very goo
Modified: you might not give emma much a complete education as your powers would seem to promise ; but you were receiving a very goo
Corrected: you might not give emma much a complete education as your powers would seem to promise ; but you were receiving a very go
Accuracy: 99.63%

Original: " not harriet ' s equal "
Modified: " not harriet ' so equal "
Corrected: " not harriet ' s equal "
Accuracy: 100.00%

Original: those soft blue eyes , and all those natural graces , should not be wasted on the inferior society of highbury and its con
Modified: those soft blue eyes , and ill those natural graces , should not be wasted on the inferior society of highbury and its con
Corrected: those soft blue eyes , and all those natural graces , should not be wasted on the inferior society of highbury and its co
Accuracy: 100.00%

Original: it was impossible to say how much he should be gratified by being employed on such an errand "
Modified: wit was impossible to say how much he should be gratified by being employed on such an errand "
Corrected: it was impossible to say how much he should be gratified by being employed on such an errand "
Accuracy: 100.00%

Original: but from one cause or another , i gave it up in disgust
Modified: but from one cause or another , i have it up in disgust
Corrected: but from one cause or another , i gave it up in disgust
Accuracy: 100.00%
```

```
accur = accuracy_corrector(test_sentences,test_sentence_corrected)
```

```
Sentence-Level Accuracy: 38.00%
Word-Level Accuracy: 96.79%
```

As we can see thanks to the analitics of the Sentence-Level and Word-Level Accuracy, the implementation of the neural-based Learning Model performed a little bit better at correcting the modified version of our test sentences, rather than using probabilities to calculate the best candidate.

Graded Exercise 3 (1 point)

Suppose that you want to compute the Perplexity metric for a given Neural Network-based model. You want to do so on the string

```
Joseph was an elderly, nay, an old man, very old, perhaps, though hale and sinewy.
```

The model is autoregressive, meaning that it is trained to produce a new token wt conditioned on w 1...wt−1. Explain how you would do so. How different is it from computing the same metric on an n-gram model?

To compute the Perplexity of a given string using a neural network-based autoregressive model, we have to follow this steps:

1. Tokenize the string in this case into individual words, puntuation marks or sub-word units depending on the tokenizer.
2. Autoregressive model based on the previous tokens, we need to compute the conditional probability P(wt | w1,w2,..., wt-1). The result will be the probability distribution for each token in a sequence given the previous tokens.
3. Calculate Log-Likelihood, for each token you calculated using predited probabilities and later you summ all the log-likelihoood of all tokens.
4. We can calculate the perplexity that is the exponential of the negative average log-likelihood.

The idea is that the model is better at predicting the next token (lower perplexity) when the log-likelihood is higher. High perplexity means the model is unsure about the next token and has a poorer performance.

Difference between Neural Network Autoregressive models and n-Gram Models:

Autoregressive models:

- Learn complex patterns in the data, so is better at modeling context.
- The model computes conditional probabilities for each token in the sequence given all the previous tokens.
- Neural network models are typically more flexible and can adjust the context window dynamically.

n-Gram Models:

- You calculate the probability of a token based only on the last n-1 tokens. For example in bigram model P(wt | wt-1).
- Are simpler and do not capture long dependencies well because they are limited to a fixed content window.
- Model more complex relationships in text.

```
#n-grams model
import math
from collections import defaultdict

text = "Joseph was an elderly, nay, an old man, very old, perhaps, though hale and sinewy."
tokens = text.lower().split()

def generate_bigrams(tokens):
    bigrams = [(tokens[i], tokens[i+1]) for i in range(len(tokens) - 1)]
    return bigrams

def compute_bigram_probabilities(bigrams):
    bigram_counts = defaultdict(int)
    unigram_counts = defaultdict(int)

    for w1, w2 in bigrams:
        bigram_counts[(w1, w2)] += 1
        unigram_counts[w1] += 1

    bigram_probabilities = defaultdict(float)
    for (w1, w2), count in bigram_counts.items():
        bigram_probabilities[(w1, w2)] = count / unigram_counts[w1]

    return bigram_probabilities, unigram_counts

def calculate_perplexity(text, bigram_probabilities, unigram_counts):
    tokens = text.lower().split()
    bigrams = generate_bigrams(tokens)

    log_likelihood = 0
    for w1, w2 in bigrams:
        if (w1, w2) in bigram_probabilities:
            prob = bigram_probabilities[(w1, w2)]
            log_likelihood += math.log(prob)
        else:
            log_likelihood += math.log(1e-10)

    perplexity = math.exp(-log_likelihood / len(bigrams))
    return perplexity

# Generate bigrams and compute probabilities
bigrams = generate_bigrams(tokens)
bigram_probabilities, unigram_counts = compute_bigram_probabilities(bigrams)

print(text)
perplexity = calculate_perplexity(text, bigram_probabilities, unigram_counts)
print(f"Perplexity (Bigram Model): {perplexity:.2f}")
```

```
Joseph was an elderly, nay, an old man, very old, perhaps, though hale and sinewy.
Perplexity (Bigram Model): 1.10
```

```
#Autoregressive model:
import torch
from transformers import GPT2Tokenizer, GPT2LMHeadModel

def compute_perplexity(text, model_name="gpt2"):
    tokenizer = GPT2Tokenizer.from_pretrained(model_name)
    model = GPT2LMHeadModel.from_pretrained(model_name)
    model.eval()

    inputs = tokenizer(text, return_tensors="pt")
    input_ids = inputs["input_ids"]

    # Compute loss (log likelihood)
    with torch.no_grad():
        outputs = model(input_ids, labels=input_ids)
        loss = outputs.loss

    perplexity = torch.exp(loss)
    return perplexity.item()

text = "Joseph was an elderly, nay, an old man, very old, perhaps, though hale and sinewy."
print(text)
ppl = compute_perplexity(text)
print(f"Perplexity with Autoregressive model: {ppl:.2f}")
```

```
Joseph was an elderly, nay, an old man, very old, perhaps, though hale and sinewy.
Perplexity with Autoregressive model: 44.31
```