Problems L4

March 17, 2025

1 Language Model Problems

Queralt Salvadó (1706789), Anna Blanco (1709582)

1.1 Graded Exercise 1

Implementation of a Spelling Corrector

Step 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.

Note: The word alterations should be at a Levenshtein distance of 1 to make more sense with respect to the testing scenario.

[ ]:
```
import nltk
import random
import numpy as np
nltk.download('gutenberg')
nltk.download('punkt_tab')
from nltk.corpus import gutenberg as corpus
from nltk.metrics.distance import edit_distance
```

```
[nltk_data] Downloading package gutenberg to /root/nltk_data…
[nltk_data]   Unzipping corpora/gutenberg.zip.
[nltk_data] Downloading package punkt_tab to /root/nltk_data…
[nltk_data]   Unzipping tokenizers/punkt_tab.zip.
```

[ ]:
```
sentences = [list(map(str.lower, s)) for s in corpus.sents()]
words = [w.lower() for w in corpus.words()]

random.shuffle(sentences)

# Take 50 sentences as test and leave the rest for training
test_sentences = sentences[:50]
train_sentences = sentences[50:]

# Flatten words and create a vocabulary
```

```python
vocabulary = list(set(words))

# Modify test sentences by replacing one word with a random in-vocabulary word
def modify_sentence(sentence):
    new_sentence = sentence.copy()

    idx = random.randint(0, len(new_sentence)-1)  # generate a random index
    original_word = new_sentence[idx].lower()

    # Create a list of words with an edit distance of 1
    similar_words = [w for w in vocabulary if edit_distance(w, original_word)␣
↪== 1]

    if not similar_words:  # If no similar words are found, skip modification
        return new_sentence, sentence

    typo_word = random.choice(similar_words)

    # Only replace if the typo word is in the vocabulary (safety check)
    if typo_word in vocabulary:
        new_sentence[idx] = typo_word

    return new_sentence, sentence

test_data = [modify_sentence(sent) for sent in test_sentences]
correct_test_sentences = [sentence for _ , sentence in test_data]
modified_test_sentences = [sentence for sentence, _ in test_data]

# Show 5 examples
for i in range(5):
    print('------------------------------------')
    print('Real Sentence:', correct_test_sentences[i])
    print('Modified Sentence:', modified_test_sentences[i])
    print('------------------------------------')
```

```
------------------------------------
Real Sentence: ['they', 'do', 'not', 'sweat', 'and', 'whine', 'about', 'their',
'condition', ',', 'they', 'do', 'not', 'lie', 'awake', 'in', 'the', 'dark',
'and', 'weep', 'for', 'their', 'sins', ',', 'they', 'do', 'not', 'make', 'me',
'sick', 'discussing', 'their', 'duty', 'to', 'god', ',', 'not', 'one', 'is',
'dissatisfied', ',', 'not', 'one', 'is', 'demented', 'with', 'the', 'mania',
'of', 'owning', 'things', ',', 'not', 'one', 'kneels', 'to', 'another', ',',
'nor', 'to', 'his', 'kind', 'that', 'lived', 'thousands', 'of', 'years', 'ago',
',', 'not', 'one', 'is', 'respectable', 'or', 'unhappy', 'over', 'the', 'whole',
'earth', '.']
Modified Sentence: ['they', 'do', 'not', 'sweat', 'and', 'whine', 'about',
'their', 'condition', ',', 'they', 'do', 'not', 'lie', 'awake', 'in', 'the',
'dark', 'and', 'weep', 'for', 'their', 'sins', ',', 'they', 'do', 'not', 'make',
```

```
'me', 'sick', 'discussing', 'their', 'duty', 'to', 'god', ',', 'not', 'one',
'is', 'dissatisfied', ',', 'mot', 'one', 'is', 'demented', 'with', 'the',
'mania', 'of', 'owning', 'things', ',', 'not', 'one', 'kneels', 'to', 'another',
',', 'nor', 'to', 'his', 'kind', 'that', 'lived', 'thousands', 'of', 'years',
'ago', ',', 'not', 'one', 'is', 'respectable', 'or', 'unhappy', 'over', 'the',
'whole', 'earth', '.']
------------------------------------
------------------------------------
Real Sentence: ['to', 'thee', 'i', 'have', 'transferred', 'all', 'judgement',
',', 'whether', 'in', 'heaven', ',', 'or', 'earth', ',', 'or', 'hell', '.']
Modified Sentence: ['to', 'thee', 'i', 'have', 'transferred', 'ally',
'judgement', ',', 'whether', 'in', 'heaven', ',', 'or', 'earth', ',', 'or',
'hell', '.']
------------------------------------
------------------------------------
Real Sentence: ['into', 'thee', 'such', 'virtue', 'and', 'grace', 'immense',
'i', 'have', 'transfused', ',', 'that', 'all', 'may', 'know', 'in', 'heaven',
'and', 'hell', 'thy', 'power', 'above', 'compare', ';', 'and', ',', 'this',
'perverse', 'commotion', 'governed', 'thus', ',', 'to', 'manifest', 'thee',
'worthiest', 'to', 'be', 'heir', 'of', 'all', 'things', ';', 'to', 'be', 'heir',
',', 'and', 'to', 'be', 'king', 'by', 'sacred', 'unction', ',', 'thy',
'deserved', 'right', '.']
Modified Sentence: ['into', 'thee', 'such', 'virtue', 'and', 'grace', 'immense',
'i', 'have', 'transfused', ',', 'that', 'all', 'may', 'know', 'in', 'heaven',
'and', 'hell', 'thy', 'power', 'above', 'compare', ';', 'and', ',', 'this',
'perverse', 'commotion', 'governed', 'thus', ',', 'to', 'manifest', 'thee',
'worthiest', 'to', 'be', 'heir', 'of', 'all', 'things', ';', 'to', 'bel',
'heir', ',', 'and', 'to', 'be', 'king', 'by', 'sacred', 'unction', ',', 'thy',
'deserved', 'right', '.']
------------------------------------
------------------------------------
Real Sentence: ['3', ':', '11', 'malchijah', 'the', 'son', 'of', 'harim', ',',
'and', 'hashub', 'the', 'son', 'of', 'pahathmoab', ',', 'repaired', 'the',
'other', 'piece', ',', 'and', 'the', 'tower', 'of', 'the', 'furnaces', '.']
Modified Sentence: ['3', ':', '11', 'malchijah', 'the', 'son', 'of', 'harim',
',', 'and', 'hashub', 'the', 'son', 'of', 'pahathmoab', ',', 'repaired', 'the',
'other', 'piece', 'i', 'and', 'the', 'tower', 'of', 'the', 'furnaces', '.']
------------------------------------
------------------------------------
Real Sentence: ['o', 'pioneers', '!']
Modified Sentence: ['(', 'pioneers', '!']
------------------------------------
```

Step 2: Implement and train an n-gram language model using the NLTK package. Try using bigrams and trigrams and the variatinos seen in class.

[ ]:
```python
from nltk.lm.preprocessing import padded_everygram_pipeline
from nltk.lm.models import MLE, StupidBackoff, Laplace
```

```
def train_ngram_model(n, train_data):
    # Preprocess data into n-grams and vocabulary
    train_data, vocab = padded_everygram_pipeline(n, train_data)

    train_data = list(train_data)
    vocab = list(vocab)

    # Initialize n-gram models
    lm_mle = MLE(n)
    lm_lpc = Laplace(n)
    lm_sbo = StupidBackoff(alpha=0.4, order=n)

    # Fit the models to the data
    lm_mle.fit(train_data, vocab)
    lm_lpc.fit(train_data, vocab)
    lm_sbo.fit(train_data, vocab)

    return lm_mle, lm_lpc, lm_sbo

# Train bigram and trigram models
bigram_mle, bigram_lpc, bigram_sbo = train_ngram_model(2, train_sentences)
trigram_mle, trigram_lpc, trigram_sbo = train_ngram_model(3, train_sentences)
```

Step 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.

[ ]:
```
n_words = 10

print('Generate words with a bigram model:', bigram_mle.generate(n_words,␣
↪text_seed='a', random_seed=42))
print('Generate words with a trigram model:', trigram_mle.generate(n_words,␣
↪text_seed='a', random_seed=42))
```

```
Generate words with a bigram model: ['positive', '!', '</s>', '<s>', 'mrs', '.',
'</s>', '-', 'hen', '!"']
Generate words with a trigram model: ['positive', 'assertion', 'that', 'every',
'slayer', 'may', 'flee', 'and', 'escape', '.']
```

Given the results obtained after generating words using different models, we can conclude that trigrams are more effective at capturing the complexity of natural language due to its ability to consider more context. This leads to better, more coherent word generation compared to bigrams, which struggle with continuity and relevance in the output.

As we can see the sentence generated by the trigram (MLE) model is much more coherent (and actually makes sense gramatically) than the one generated by the bigram (MLE) model.

Now we will compute the perplexity of the trained models:

```
[ ]: from nltk.lm.preprocessing import padded_everygram_pipeline
from nltk.lm import MLE, StupidBackoff, Laplace
from nltk import bigrams, trigrams

def compute_perplexity(model, n, test_data):
    test_ngrams, _ = padded_everygram_pipeline(n, test_data)

    flattened_ngrams = [ngram for sentence in test_ngrams for ngram in sentence]

    perplexity = model.perplexity(flattened_ngrams)
    return perplexity

bigram_perplexity_mle = compute_perplexity(bigram_mle, 2, test_sentences)
bigram_perplexity_lpc = compute_perplexity(bigram_lpc, 2, test_sentences)
bigram_perplexity_sbo = compute_perplexity(bigram_sbo, 2, test_sentences)

print('BIGRAMS:')
print('Perplexity Bigram MLE:', bigram_perplexity_mle)
print('Perplexity Bigram Laplace:', bigram_perplexity_lpc)
print('Perplexity Bigram SBO:', bigram_perplexity_sbo)

trigram_perplexity_mle = compute_perplexity(trigram_mle, 3, test_sentences)
trigram_perplexity_lpc = compute_perplexity(trigram_lpc, 3, test_sentences)
trigram_perplexity_sbo = compute_perplexity(trigram_sbo, 3, test_sentences)

print('\n')
print('TRIGRAMS')
print('Perplexity Trigram MLE:', trigram_perplexity_mle)
print('Perplexity Trigram Laplace:', trigram_perplexity_lpc)
print('Perplexity Trigram SBO:', trigram_perplexity_sbo)
```

```
BIGRAMS:
Perplexity Bigram MLE: inf
Perplexity Bigram Laplace: 42308.000000006265
Perplexity Bigram SBO: inf



TRIGRAMS
Perplexity Trigram MLE: inf
Perplexity Trigram Laplace: 42308.000000006374
Perplexity Trigram SBO: inf
```

The perplexity results, particularly the infinite perplexity for the MLE and SBO models, suggest that either the models are facing an excessive number of unseen n-grams (therefore the assigned probability is 0 resulting in infinite perplexity) or that there are errors in the data pipeline or model training. The extremely high perplexity for the Laplace models points to a potential issue with the training data's quality or its mismatch with the test data. *In order to further investigate this; we will try to calculate the perplexity by using the same training data:*

[ ]:
```python
def compute_perplexity(model, n, test_data):
    test_ngrams, _ = padded_everygram_pipeline(n, test_data)

    flattened_ngrams = [ngram for sentence in test_ngrams for ngram in sentence]

    perplexity = model.perplexity(flattened_ngrams)
    return perplexity

bigram_perplexity_mle = compute_perplexity(bigram_mle, 2, train_sentences)
bigram_perplexity_lpc = compute_perplexity(bigram_lpc, 2, train_sentences)
bigram_perplexity_sbo = compute_perplexity(bigram_sbo, 2, train_sentences)

print('BIGRAMS:')
print('Perplexity Bigram MLE:', bigram_perplexity_mle)
print('Perplexity Bigram Laplace:', bigram_perplexity_lpc)
print('Perplexity Bigram SBO:', bigram_perplexity_sbo)

trigram_perplexity_mle = compute_perplexity(trigram_mle, 3, train_sentences)
trigram_perplexity_lpc = compute_perplexity(trigram_lpc, 3, train_sentences)
trigram_perplexity_sbo = compute_perplexity(trigram_sbo, 3, train_sentences)

print('\n')
print('TRIGRAMS')
print('Perplexity Trigram MLE:', trigram_perplexity_mle)
print('Perplexity Trigram Laplace:', trigram_perplexity_lpc)
print('Perplexity Trigram SBO:', trigram_perplexity_sbo)
```

```
BIGRAMS:
Perplexity Bigram MLE: 202.8611367430209
Perplexity Bigram Laplace: 42307.99998423167
Perplexity Bigram SBO: inf


TRIGRAMS
Perplexity Trigram MLE: 68.2349731465614
Perplexity Trigram Laplace: 42307.999983443035
Perplexity Trigram SBO: inf
```

When evaluating the models on the same data used for training, the perplexity notably decreases, especially for the MLE models. Which suggests that the infinite perplexity might be due to the fact that the model is encountering unknown words when computing the perplexity with the test data.

More specifically, we observe that the Trigram MLE model results in the lowest perplexity, which makes sense since it has access to more context, allowing it to make better predictions. This coincides with the results we obtained while generating words with both bigrams and trigrams, in which we concluded that trigrams (specifically the MLE model) provided more coherent sentences.

Additionally, it's worth noting that the perplexities for both the Laplace and Stupid BackOff models have remained unchanged, which strongly suggests that there may be an issue in the earlier steps of the code, as mentioned above.

By trying different subcases and investigating it, we deduced that the error behind these results should be in the preprocessing of the data either when passing it to the model, or when passing it to the perplexity function.

Step 4: 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.

[ ]:
```
from nltk.lm.api import LanguageModel
from nltk.metrics.distance import edit_distance
from typing import List, Dict
import numpy as np
from tqdm.auto import tqdm

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

def candidates(close_words: Dict[str, List[str]], vocab: List[str], sentence:␣
↪List[str]):
    sent_x = sentence
    for word_x in sent_x:
        assert word_x in vocab
    candidates = [sent_x]
```

```
    for ii, word_x in enumerate(sent_x):
        for cand_w in close_words[word_x]:
            #if cand_w != word_x:
            #    cand_sent = sent_x.copy()
            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]])
        log_prior_w += score
    return log_prior_w
```

[ ]:
```
import re

# We have a word-level output, so this will be slow regardless of how
# we set it up. In the re package, we can use functions as replacements.
# These functions take a match object as parameter.

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 nltk
from nltk.corpus import gutenberg as corpus

nltk.download("punkt")
nltk.download("punkt_tab")
nltk.download("gutenberg")

text = [
    list(filter(lambda x: x is not None, map(preprocess_words, x)))
    for x in corpus.sents()[:1000]
]

vocab = list(sorted(set([x for sent in text for x in sent])))
close_words = compute_close_words(vocab, 1)
```

```
[nltk_data] Downloading package punkt to /root/nltk_data…
[nltk_data]   Unzipping tokenizers/punkt.zip.
[nltk_data] Downloading package punkt_tab to /root/nltk_data…
[nltk_data]   Package punkt_tab is already up-to-date!
[nltk_data] Downloading package gutenberg to /root/nltk_data…
[nltk_data]   Package gutenberg is already up-to-date!
```

```
  0%|          | 0/2550 [00:00<?, ?it/s]
```

[ ]:
```
# update function to only allow modifications from within the vocabulary
def modify_sentence(sentence, vocabulary):
    new_sentence = [word for word in sentence if word in vocabulary]  # Remove␣
↪OOV words
    if not new_sentence:  # If the sentence becomes empty, return
        return None

    idx = random.randint(0, len(new_sentence) - 1)  # Generate a random index
    original_word = new_sentence[idx].lower()

    # Create a list of words with an edit distance of 1, only selecting␣
↪in-vocabulary words
    similar_words = [w for w in vocabulary if edit_distance(w, original_word)␣
↪== 1 and w in vocabulary]

    if not similar_words:  # If no similar words are found, return
        return None

    typo_word = random.choice(similar_words)

    # Replace only if the typo word is in the vocabulary
    new_sentence[idx] = typo_word
```

```
    return new_sentence, sentence
```

[ ]:
```
def spell_corrector(lm: LanguageModel, close_words: Dict[str, List[str]], vocab:
 ↪ List[str], sentence: List[str]):
    candidate_sentences = candidates(close_words, vocab, sentence)
    best_sentence = None
    best_score = -float('inf')

    for candidate in candidate_sentences:
        score = log_prior(candidate, lm)
        if score > best_score:
            best_score = score
            best_sentence = candidate
    return (best_sentence if best_sentence else sentence), best_score # Return␣
 ↪original if no correction

# Evaluate the spelling corrector on the modified train sentences

# Modify sentences while filtering out None values
sentences = [modify_sentence(sentence, vocab) for sentence in train_sentences[:
 ↪100]]
sentences = [s for s in sentences if s is not None]  # Remove None values

# Separate modified and correct sentences
correct_sentences = [sentence for _, sentence in sentences]
modified_sentences = [sentence for sentence, _ in sentences]

#for sentence in modified_test_sentences:
corrected_sentences = []
for i in range(len(modified_sentences)):
    corrected_sentence, best_score = spell_corrector(trigram_mle, close_words,␣
 ↪vocab, modified_sentences[i])
    corrected_sentences.append(corrected_sentence)
    if best_score > -float('inf'):
      print(f"Original: {' '.join(correct_sentences[i])}")
      print(f"Modified: {' '.join(modified_sentences[i])}")
      print(f"Corrected: {' '.join(corrected_sentences[i])}")
      print()
```

```
Original: " i have not given mine , and i give it to leonora ."
Modified: " i gave not given mine , and i give it to
Corrected: " i have not given mine , and i give it to

Original: " what !
Modified: " that
Corrected: " what
```

Original: i felt for my sister most severely .
Modified: i fell for my sister most
Corrected: i felt for my sister most

Original: but ah !
Modified: put ah
Corrected: but ah

Original: thy kingdom come .
Modified: cope
Corrected: come

Original: cried the proprietor hastily .
Modified: cried they
Corrected: cried the

Original: " he chose to say he was employed "--
Modified: " he chose to lay he was employed
Corrected: " he chose to say he was employed

Original: i do believe in liberty .
Modified: i do believe it liberty
Corrected: i do believe in liberty

Original: he is slaine
Modified: the is
Corrected: he is

Original: let her name her own supper , and go to bed .
Modified: let her name her town supper , and go to bed
Corrected: let her name her own supper , and go to bed

Original: at first he refused to tell them from whom he got them , because he had bought them , he said , under a promise of secrecy .
Modified: sat first he refused to tell them from whom he got them , because he had bought them , he said , under a promise of
Corrected: at first he refused to tell them from whom he got them , because he had bought them , he said , under a promise of

Original: there was really nothing against the man at all .
Modified: there was really nothing against the mean at all
Corrected: there was really nothing against the man at all

Original: ' well !'
Modified: ' tell
Corrected: ' well

```
Original: " it was at gibraltar , mother , i know .
Modified: " it was t , mother , i know
Corrected: " it was i , mother , i know

Original: she waited quietly in the passage .
Modified: see waited in the
Corrected: she waited in the

Original: then would i remove abimelech .
Modified: then could i
Corrected: then would i

Original: " what did you hear ?"
Modified: " what bid you hear
Corrected: " what did you hear

Original: where could you expect a more gentlemanlike , agreeable man ?
Modified: where would you expect a more gentlemanlike , agreeable man
Corrected: where could you expect a more gentlemanlike , agreeable man

Original: " and what do you suppose we are ?"
Modified: " and what do you supposed we are
Corrected: " and what do you suppose we are

Original: i threw myself into the major .
Modified: v threw myself into the
Corrected: i threw myself into the

Original: they had heard of such a thing , but they had only heard of it .
Modified: they had heard of such ah thing , but they had only heard of it
Corrected: they had heard of such a thing , but they had only heard of it

Original: " but you have spent a whole week in making this thing for her ."
Modified: " but your have spent a whole week in making this thing for her
Corrected: " but you have spent a whole week in making this thing for her

Original: " rubbish !"
Modified: d
Corrected: "

Original: fly my lord , flye
Modified: my t
Corrected: my ,
```

First, it's important to note that the trigram and bigram MLE models failed to handle unseen data from the test set, resulting in **infinite perplexity**. Because of this, when we attempted to use the spelling corrector on test sentences, no corrections were made. This happened because the `log_prior` function always returned `-inf`, preventing the model from identifying a more probable sentence than the given one.

To address this issue (something we anticipated) we tested our spelling corrector on **training sentences** instead. While we knew this wasn't ideal, it was the only way to evaluate the model. We found that the corrector worked on some sentences but had a **low accuracy** (9/48   19%).

Another challenge was that all words in the input sentences had to be in the vocabulary, which was quite limited. To work around this, we modified the function from *Step 1* to remove out-of-vocabulary words, ensure the replacement word was in the vocabulary, and return None if this wasn't possible. In order to avoid raising an `AssertionError` everytime that an out-of-vocabulary word was found.

Although this approach has its limitations, it allowed us to test the model and analyze its performance. As expected, the corrector successfully fixed some sentences. In other cases, it produced valid but unexpected corrections. This happens because the model selects the sentence with the **highest probability**, which doesn't always match the expected one. Since our approach is probabilistic, such deviations are unavoidable.

1.2 Graded Exercise 2

Change the code of the spell corrector to use the model proposed. 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 tqdm import tqdm
import numpy as np

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader

import nltk

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 -1
        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)
        #Change in this line
        X = word_embeds.view(-1, self.dim_input * self.dim_embedding)  # first␣
↪layer #Change dim_input to length_window
        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
```

[ ]:
```
class FixedWindow(Dataset):
    def __init__(self, words, length_window):
        """
        Processes the corpus into fixed-size word windows.
        Creates (context, target) pairs for training.
        """
        super().__init__()
        self.length_window = length_window

        # Create vocabulary mapping
        self.vocab = list(set(words))
        self.vocabulary_size = len(self.vocab)

        self.word2id = {word: idx for idx, word in enumerate(self.vocab)}
        self.id2word = {idx: word for word, idx in self.word2id.items()}

        # Convert words to IDs
        self.ids = [self.word2id[word] for word in words]

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

    def __getitem__(self, idx):
        context = self.ids[idx:idx + self.length_window - 1]
        target = self.ids[idx + self.length_window - 1 ]
        return torch.tensor(context, dtype=torch.long).to(device), torch.
↪tensor(target, dtype=torch.long).to(device)
```

[ ]:
```
length_window = 5
dataset = FixedWindow(words, length_window)

batch_size = 64
dataloader = DataLoader(
    dataset, batch_size=batch_size, shuffle=True
)  # 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
"""
# In comments as if not the output is too large
num_classes = dataset.vocabulary_size
dim_input = length_window
dim_hidden = 50
dim_embedding = 32
learning_rate = 1e-3
num_epochs = 20

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)
                #In comments as if not the output was too large

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

for i in range(10):
    print(f"Original Sentence: {' '.join(correct_test_sentences[i])}")
    print(f"Modified Sentence: {' '.join(modified_test_sentences[i])}")
    print(f"Corrected Sentence: {' '.join(corrected_sentences[i])}")
    print()
```

For this exercise, we built a Neural Network Language Model (NNLM) to predict the next word in a sentence using a fixed window of previous words. The model includes an embedding layer that converts words into dense vectors, followed by a simple neural network with a hidden layer that processes these embeddings and makes predictions. The goal is for the model to learn word relationships and generate the most likely next word based on context.

To prepare the training data, we created the `FixedWindow` class, which processes the text by breaking it into overlapping word sequences of a fixed length. Each sequence consists of a context (the first `n-1` words) and a target (the last word). The model is trained to predict this target word given the context. The dataset also includes a vocabulary mapping, where each word is assigned a unique ID to be used in training.

When attempting to train the model, we ran into hardware limitations. The GPU in Colab reached its usage limit, and running it on the CPU was far too slow, taking over an hour per epoch. Since we didn't have enough time to wait for training to complete, it wasn't possible to properly test the model. However, based on the architecture and implementation, we believe it should work as expected.

1.3 Graded Exercise 3

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 w_t conditioned on $w_1 \dots w_{t-1}$.

Explain how you would do so. How different is it from computing the same metric on an n-gram model?

In order to compute the perplexity metric for a Neural Network-based model we would need to do the following:

1.First of all, tokenize the input sequence (break the input text into tokens). For example, for our sentence:

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

We would want to obtain the following:

[1]:
```
sentence =['Joseph','was','an','elderly','nay','an','old','man',
           'very','old','perhaps','though','hale','and','sinewy']
```

2.Then, we would want to feed the tokenized sentence into the model. Since it is an autoregressive model, for each token w_t it computes the probability $ P(w_t | w_1, w_2, ..., w_{t-1}) $. This can be achieved by feeding the tokens into the model one by one, from w_1 to w_T.

3.The next thing to do is to calculate the log probabilities. Which is given by $logP(w_t|w_1, w_2, \dots, w_{t-1})$. The reason to do this is because the probability of a sequence of tokens is generally quite small, since the model assigns a small probability to each token. Multiplying these small probabilities will lead to extremely small numbers (underflow problem), which is why we use logarithms to turn multiplications into additions and solve this problem.

4.Once we have the log probabilities over all tokens we can sum them, which will give us the log-likelihood of the sequence.

$$\text{Log Likelihood} = \sum_{t=1}^{T} \log P(w_t|w_1, w_2, \dots, w_{t-1})$$

5.Finally, with these log-probabilities we can compute the perplexity as:

$$\text{Perplexity} = \exp\left(-\frac{1}{T}\sum_{t=1}^{T} \log P(w_t|w_1, w_2, \dots, w_{t-1})\right)$$

which will be a scalar value representing the perplexity of the model over the sequence of tokens.

Difference from computing perplexity on an n-gram model The main difference between calculating perplexity on a neural network-based model and on an n-gram model lies on how conditional probabilities are computed:

N-gram model: In an n-gram model, the conditional probability $P(w_t|w_1, w_2, \dots, w_{t_1})$ is approximated by a fixed-size of previous tokens (n-1 tokens). For example:

- In a **bigram model** (n=2) we compute $P(w_t|w_{t_1})$, meaning only the previous word is considered.
- In a **trigram model** (n=3) we would use the two previous words.

These probability are estimated based on frequency counts from the training data. For example, in a bigram we would compute $P(w_t|w_{t-1})$ as:

$$P(w_t|w_{t-1}) = \frac{\text{count}(w_{t-1}, w_t)}{\text{count}(w_{t-1})}$$

Neural network-based model: Neural networks, on the other hand, can consider all previous tokens in the sequence, allowing them to capture long-range dependencies and complex patterns. Unlike n-grams, which rely on fixed context sizes, neural networks learn these dependencies during training, making them more flexible and capable of modeling broader relationships in the data.