Utility Functions

In [ ]:
```
from __future__ import unicode_literals, print_function, division
from io import open
import unicodedata
import string
import re
import random

import torch
import torch.nn as nn
from torch import optim
import torch.nn.functional as F

import pandas as pd

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

In [ ]:
```
from collections import Counter, defaultdict
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F


cuda_available = torch.cuda.is_available()
```

In [ ]:
```
torch.cuda.is_available()
```

Out[ ]:
```
True
```

In [ ]:
```
torch.cuda.device_count()
```

Out[ ]:
```
1
```

In [ ]:
```
torch.cuda.current_device()
```

Out[ ]:
```
0
```

In [ ]:
```
MAX_LENGTH = 150
```

In [ ]:
```
import time
import math

def as_minutes(s: float) -> str:
    """Converts seconds to a formatted minutes and seconds string."""
    m = math.floor(s / 60)
    s -= m * 60
    return f'{m}m {int(s)}s'

def time_since(start: float, percent: float) -> str:
    """Calculates elapsed time since start and estimated remaining time based on progress percentage."""
    now = time.time()
    elapsed_seconds = now - start
    total_estimated_seconds = elapsed_seconds / percent
    remaining_seconds = total_estimated_seconds - elapsed_seconds
    return f'{as_minutes(elapsed_seconds)} (- {as_minutes(remaining_seconds)})'
```

In [ ]:
```
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
import numpy as np

def showPlot(all_train_losses, all_val_losses, colors, model_labels, title=None):
    plt.figure(figsize=(12, 6))

    # Ensure we have four models' data and four colors
    if len(all_train_losses) != 4 or len(all_val_losses) != 4 or len(colors) != 4 or len(model_labels) != 4:
        raise ValueError("Ensure there are exactly four models' losses, colors, and labels provided.")
    # Plot each model's training and validation loss
    for train_losses, val_losses, color, label in zip(all_train_losses, all_val_losses, colors, model_labels):
        plt.plot(train_losses, color=color, label=f'{label} Training Loss', linestyle='solid')
        plt.plot(val_losses, color=color, label=f'{label} Validation Loss', linestyle='dotted')

    plt.xlabel('Iterations')
    plt.ylabel('Loss')
    plt.title(title if title else 'Training and Validation Losses')
    plt.legend()
    plt.show()
```

In [ ]:
```
def indexesFromSentence(lang, sentence):
    return [lang.word2index[word] for word in sentence.split(' ')]

def tensorFromSentence(lang, sentence):
    indexes = [lang.word2index.get(word, UNK_token) for word in sentence.split(' ')] + [EOS_token]
    tensor = torch.tensor(indexes, dtype=torch.long, device=device).view(-1, 1)
    return tensor

def tensorsFromPair(pair):
    input_tensor = tensorFromSentence(input_lang, pair[0])
    target_tensor = tensorFromSentence(output_lang, pair[1])
    return (input_tensor, target_tensor)
```

Data Processing

In [ ]:
```
# from google.colab import drive
# drive.mount('/content/drive')
file_path = "Cooking_Dataset/"
```

In [ ]:
```
import pandas as pd

train_file_path = file_path + "train.csv"
dev_file_path = file_path + "dev.csv"
test_file_path = file_path + "test.csv"
# train_file_path = "D:/Monash/Master Degree/FIT5217/Ass2/Coding/Cooking_Dataset/Cooking_Dataset/train.csv"
# dev_file_path = "D:/Monash/Master Degree/FIT5217/Ass2/Coding/Cooking_Dataset/Cooking_Dataset/test.csv"

# train_data = pd.read_csv(file_path)
# dev_data = pd.read_csv(dev_file_path)
```

In [ ]:
```
import unicodedata
import re

# Turn a Unicode string to plain ASCII, thanks to
# https://stackoverflow.com/a/518232/2809427
def unicodeToAscii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
    )

# Lowercase, trim, and remove non-letter characters


def normalizeString(s):
    s = unicodeToAscii(s.lower().strip())
    s = re.sub(r"([.!?])", r" \1", s)
    s = re.sub(r"[^a-zA-Z0-9.!?]+", r" ", s)
    return s
```

In [ ]:
```
MAX_LENGTH = 150

def filterPair(p):
    return p is not None and len(p[0].split(' ')) < MAX_LENGTH and len(p[1].split(' ')) < MAX_LENGTH


def filterPairs(pairs):
    return [pair for pair in pairs if filterPair(pair)]
```

In [ ]:
```
SOS_token = 0
EOS_token = 1
UNK_token = 2
```

```
class Lang:
    def __init__(self, name):
        self.name = name
        self.word2index = {}
        self.word2count = {}
        self.index2word = {0: "SOS", 1: "EOS", 2: "UNK"}
        self.n_words = 3  # Count SOS, EOS, and UNK

    def addSentence(self, sentence):
        for word in sentence.split(' '):
            self.addWord(word)

    def addWord(self, word):
        if word not in self.word2index:
            self.word2index[word] = self.n_words
            self.word2count[word] = 1
            self.index2word[self.n_words] = word
            self.n_words += 1
        else:
            self.word2count[word] += 1
```

In [ ]:
```
def readLangs(lang1, lang2, file_path):
    print("Reading lines...")

    # Read the CSV file
    data = pd.read_csv(file_path)

    # Extract the last two columns
    # title_col = data['Title']
    ingredients_col = data.columns[-2]
    recipe_col = data.columns[-1]
    lines = data[[ingredients_col, recipe_col]].dropna().values.tolist()

    # Split every line into pairs and normalize
    pairs = [[normalizeString(s) for s in l] for l in lines]

    input_lang = Lang(lang1)
    output_lang = Lang(lang2)

    return input_lang, output_lang, pairs
```

In [ ]:
```
def prepareData(lang1, lang2, file_path):
    print("Reading lines...")

    # Read the CSV file and extract the required columns
    data = pd.read_csv(file_path)
    ingredients_col = data.columns[-2]
    recipe_col = data.columns[-1]
    lines = data[[ingredients_col, recipe_col]].dropna().values.tolist()

    # Normalize and create pairs
    pairs = [[normalizeString(s) for s in l] for l in lines]

    # Initialize language processing
    input_lang = Lang(lang1)
    output_lang = Lang(lang2)

    # Initialize lists to store lengths for analysis
    ingredient_word_counts = []
    recipe_word_counts = []

    print("Read %s sentence pairs" % len(pairs))
    pairs = [pair for pair in pairs if filterPair(pair)]
    print("Trimmed to %s sentence pairs" % len(pairs))
    print("Counting words...")

    for pair in pairs:
        input_lang.addSentence(pair[0])
        output_lang.addSentence(pair[1])
        ingredient_word_counts.append(len(pair[0].split()))
        recipe_word_counts.append(len(pair[1].split()))

    print("Counted words:")
    print(input_lang.name, input_lang.n_words)
    print(output_lang.name, output_lang.n_words)
    shortest_ingredient_length = min(ingredient_word_counts)
    shortest_recipe_length = min(recipe_word_counts)

    longest_ingredient_length = max(ingredient_word_counts)
    longest_recipe_length = max(recipe_word_counts)

    average_ingredient_length = sum(ingredient_word_counts) / len(ingredient_word_counts)
    average_recipe_length = sum(recipe_word_counts) / len(recipe_word_counts)

    # Displaying the collected statistics
    print(f"Ingredients - Longest: {longest_ingredient_length}, Shortest: {shortest_ingredient_length}, Average: {average_ing
    print(f"Recipes - Longest: {longest_recipe_length}, Shortest: {shortest_recipe_length}, Average: {average_recipe_length}"

    return input_lang, output_lang, pairs
```

In [ ]:
```
input_lang, output_lang, train_pairs = prepareData('Ingredients', 'Recipes', train_file_path)
```

```
Reading lines...
Read 101338 sentence pairs
Trimmed to 82435 sentence pairs
Counting words...
Counted words:
Ingredients 15076
Recipes 28342
Ingredients - Longest: 148, Shortest: 1, Average: 42.12213258931279
Recipes - Longest: 149, Shortest: 1, Average: 74.61791714684297
```

In [ ]:
```
input_lang_dev, output_lang_dev, val_pairs = prepareData('Ingredients_dev', 'Recipes_dev', dev_file_path)
```

```
Reading lines...
Read 797 sentence pairs
Trimmed to 658 sentence pairs
Counting words...
Counted words:
Ingredients_dev 1816
Recipes_dev 3155
Ingredients - Longest: 144, Shortest: 2, Average: 40.838905775075986
Recipes - Longest: 148, Shortest: 3, Average: 74.2127659574468
```

In [ ]:
```
input_lang_test, output_lang_test, test_pairs = prepareData('Ingredients_dev', 'Recipes_dev', test_file_path)
```

```
Reading lines...
Read 778 sentence pairs
Trimmed to 650 sentence pairs
Counting words...
Counted words:
Ingredients_dev 1853
Recipes_dev 2982
Ingredients - Longest: 140, Shortest: 1, Average: 40.94307692307692
Recipes - Longest: 149, Shortest: 3, Average: 74.11846153846155
```

In [ ]:
```
print(train_pairs[0])
```

```
['6 tb butter or margarine softened 3 4 c c and h powdered sugar 1 c all purpose flour 1 tb milk 2 eggs 1 c c and h granulated
sugar 1 2 c cocoa 2 tb flour 1 2 ts baking powder 1 2 ts salt 1 ts vanilla 1 4 ts almond extract optional 1 c chopped almonds
or pecans', 'cream together butter and powdered sugar . blend in 1 cup flour and milk . spread evenly in bottom of ungreased 9
inch square pan . bake in 350 degree oven 10 to 12 minutes . beat eggs slightly combine dry ingredients and add to eggs . blen
d in vanilla and almond extract fold in almonds . spread over hot baked layer return to oven and bake 20 minutes longer . cool
while warm cut into 24 bars .']
```

Baseline 1: Sequence-to-Sequence model without attention

In [ ]:
```
import torch
import torch.nn as nn

class EncoderRNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers=1, dropout=0.0):
        super(EncoderRNN, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.embedding = nn.Embedding(input_size, hidden_size)
        self.lstm = nn.LSTM(hidden_size, hidden_size, num_layers=num_layers, dropout=dropout)

    def forward(self, input, hidden):
        embedded = self.embedding(input).view(1, 1, -1)
        # print(embedded.size(), hidden[0].size(), hidden[1].size())
        output, hidden = self.lstm(embedded, hidden)
        return output, hidden

    def initHidden(self):
        return (torch.zeros(self.num_layers, 1, self.hidden_size, device=device),
                torch.zeros(self.num_layers, 1, self.hidden_size, device=device))
```

In [ ]:
```
import torch.nn.functional as F

class DecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size, num_layers=1, dropout=0.0):
        super(DecoderRNN, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.embedding = nn.Embedding(output_size, hidden_size)
        self.lstm = nn.LSTM(hidden_size, hidden_size, num_layers=num_layers, dropout=dropout if num_layers > 1 else 0)
        self.out = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        output = self.embedding(input).view(1, 1, -1)
        output = torch.relu(output)
        output, hidden = self.lstm(output, hidden)
        output = self.softmax(self.out(output[0]))
        return output, hidden

    def initHidden(self):
        return (torch.zeros(self.num_layers, 1, self.hidden_size, device=device),
                torch.zeros(self.num_layers, 1, self.hidden_size, device=device))
```

In [ ]:
```
def indexesFromSentence_m1(lang, sentence):
    return [lang.word2index[word] for word in sentence.split(' ')]

def tensorFromSentence_m1(lang, sentence):
    indexes = [lang.word2index.get(word, UNK_token) for word in sentence.split(' ')] + [EOS_token]
    tensor = torch.tensor(indexes, dtype=torch.long, device=device).view(-1, 1)
    return tensor

def tensorsFromPair_m1(pair):
    input_tensor = tensorFromSentence_m1(input_lang, pair[0])
    target_tensor = tensorFromSentence_m1(output_lang, pair[1])
    return (input_tensor, target_tensor)
```

In [ ]:
```
teacher_forcing_ratio = 1

def train_m1(input_tensor, target_tensor, encoder, decoder, encoder_optimizer, decoder_optimizer, criterion, max_length=MAX_L
    encoder_hidden = encoder.initHidden()

    encoder_optimizer.zero_grad()
    decoder_optimizer.zero_grad()

    input_length = input_tensor.size(0)
    target_length = target_tensor.size(0)

    encoder_outputs = torch.zeros(max_length, encoder.hidden_size, device=device)

    loss = 0

    # Pass the input through the encoder
    for ei in range(input_length):
        encoder_output, encoder_hidden = encoder(
            input_tensor[ei], encoder_hidden)
        if ei < max_length:
            encoder_outputs[ei] = encoder_output[0, 0]

    # Prepare decoder's input and hidden state
    decoder_input = torch.tensor([[SOS_token]], device=device)
    decoder_hidden = encoder_hidden  # Transfer encoder's final hidden state to decoder

    use_teacher_forcing = True if random.random() < teacher_forcing_ratio else False

    if use_teacher_forcing:
        # Teacher forcing: Feed the target as the next input
        for di in range(target_length):
            decoder_output, decoder_hidden = decoder(
                decoder_input, decoder_hidden)
            loss += criterion(decoder_output, target_tensor[di])
            decoder_input = target_tensor[di]  # Teacher forcing

    else:
        # Without teacher forcing: use its own predictions as the next input
        for di in range(target_length):
            decoder_output, decoder_hidden = decoder(
                decoder_input, decoder_hidden)
            topv, topi = decoder_output.topk(1)
            decoder_input = topi.squeeze().detach()  # detach from history as input

            loss += criterion(decoder_output, target_tensor[di])
            if decoder_input.item() == EOS_token:
                break

    loss.backward()

    encoder_optimizer.step()
    decoder_optimizer.step()

    return loss.item() / target_length
```

In [ ]:
```
def evaluate_m1(input_tensor, target_tensor, encoder, decoder, criterion, max_length=MAX_LENGTH):
    encoder_hidden = encoder.initHidden()
    input_length = input_tensor.size(0)
    target_length = target_tensor.size(0)
    encoder_outputs = torch.zeros(max_length, encoder.hidden_size, device=device)
    loss = 0

    # Encoder
    for ei in range(input_length):
        encoder_output, encoder_hidden = encoder(input_tensor[ei], encoder_hidden)
        if ei < max_length:
            encoder_outputs[ei] = encoder_output[0, 0]

    # Decoder
    decoder_input = torch.tensor([[SOS_token]], device=device)
    decoder_hidden = encoder_hidden

    for di in range(target_length):
        decoder_output, decoder_hidden = decoder(decoder_input, decoder_hidden)
        topv, topi = decoder_output.topk(1)
        decoder_input = topi.squeeze().detach()  # Use its own predictions as the next input
        loss += criterion(decoder_output, target_tensor[di])
        if decoder_input.item() == EOS_token:
            break

    return loss.item() / target_length
```

In [ ]:
```
def validate_model_m1(encoder, decoder, val_pairs, criterion, max_length=MAX_LENGTH, num_samples=300):
    # Randomly select a subset of validation pairs
    if len(val_pairs) > num_samples:
        sampled_pairs = random.sample(val_pairs, num_samples)
    else:
        sampled_pairs = val_pairs

    total_loss = 0
    with torch.no_grad():
        for input_tensor, target_tensor in sampled_pairs:
            loss = evaluate_m1(input_tensor, target_tensor, encoder, decoder, criterion, max_length)
            total_loss += loss

    average_loss = total_loss / len(sampled_pairs)
    return average_loss
```

In [ ]:
```
import torch.optim as optim
import random
import torch.optim as optim
import random
import time

def train_iters_m1(encoder, decoder, n_iters, print_every=1000, plot_every=100, learning_rate=0.01):
    start_time = time.time()
    plot_losses = []
    val_losses = []
    print_loss_total = 0  # Reset every print_every
    plot_loss_total = 0  # Reset every plot_every

    # Use Adam optimizer as specified in your model requirements
    encoder_optimizer = optim.Adam(encoder.parameters(), lr=learning_rate)
    decoder_optimizer = optim.Adam(decoder.parameters(), lr=learning_rate)

    # Prepare training data
    training_pairs = [tensorsFromPair_m1(random.choice(train_pairs))
                      for i in range(n_iters)]
    validation_pairs = [tensorsFromPair_m1(random.choice(val_pairs))
                      for i in range(n_iters)]
    criterion = nn.NLLLoss()

    for iter in range(1, n_iters + 1):
        training_pair = training_pairs[iter - 1]
        input_tensor = training_pair[0]
        target_tensor = training_pair[1]

        # Train with the selected pair
        loss = train_m1(input_tensor, target_tensor, encoder,
                     decoder, encoder_optimizer, decoder_optimizer, criterion)
        print_loss_total += loss
        plot_loss_total += loss

        # Print progress and average loss
        if iter % print_every == 0:
            print_loss_avg = print_loss_total / print_every
            print_loss_total = 0
            print(f'{time_since(start_time, iter / n_iters)} ({iter} {iter / n_iters:.2%}) {print_loss_avg:.4f}')

        # Store average loss for plotting
        if iter % plot_every == 0:
            plot_loss_avg = plot_loss_total / plot_every
            plot_losses.append(plot_loss_avg)
            plot_loss_total = 0

            # Validation evaluation
            val_loss = validate_model_m1(encoder, decoder, validation_pairs, criterion)
            val_losses.append(val_loss)
            print(f'Validation Loss: {val_loss:.4f} at iteration {iter}')

    # showPlot(plot_losses, val_losses)
    return plot_losses, val_losses
```

In [ ]:
```
hidden_size = 256
encoder_m1 = EncoderRNN(input_lang.n_words, hidden_size).to(device)
decoder_m1 = DecoderRNN(hidden_size, output_lang.n_words).to(device)
train_losses, val_losses = train_iters_m1(encoder_m1, decoder_m1, 40000, print_every=500, plot_every=500)
# showPlot(train_losses, val_losses, red)
```

```
0m 39s (- 52m 3s) (500 1.25%) 5.7404
Validation Loss: 3.1464 at iteration 500
1m 15s (- 49m 13s) (1000 2.50%) 5.0871
Validation Loss: 4.5419 at iteration 1000
1m 54s (- 48m 50s) (1500 3.75%) 4.9808
Validation Loss: 5.4051 at iteration 1500
2m 32s (- 48m 14s) (2000 5.00%) 4.9294
Validation Loss: 6.3542 at iteration 2000
3m 11s (- 47m 59s) (2500 6.25%) 4.9165
Validation Loss: 4.4583 at iteration 2500
3m 49s (- 47m 11s) (3000 7.50%) 4.8167
Validation Loss: 3.7760 at iteration 3000
4m 25s (- 46m 10s) (3500 8.75%) 4.9312
Validation Loss: 4.2331 at iteration 3500
5m 3s (- 45m 27s) (4000 10.00%) 4.8652
Validation Loss: 5.2609 at iteration 4000
5m 42s (- 44m 59s) (4500 11.25%) 4.8574
Validation Loss: 6.1817 at iteration 4500
6m 20s (- 44m 25s) (5000 12.50%) 4.7463
Validation Loss: 5.3430 at iteration 5000
6m 59s (- 43m 51s) (5500 13.75%) 4.7113
Validation Loss: 5.3141 at iteration 5500
7m 38s (- 43m 16s) (6000 15.00%) 4.5969
Validation Loss: 7.0327 at iteration 6000
8m 17s (- 42m 44s) (6500 16.25%) 4.5925
Validation Loss: 4.3965 at iteration 6500
8m 54s (- 42m 0s) (7000 17.50%) 4.6825
Validation Loss: 3.8457 at iteration 7000
9m 31s (- 41m 16s) (7500 18.75%) 4.6399
Validation Loss: 4.9504 at iteration 7500
10m 7s (- 40m 31s) (8000 20.00%) 4.6825
Validation Loss: 4.2275 at iteration 8000
10m 45s (- 39m 51s) (8500 21.25%) 4.5597
Validation Loss: 3.4895 at iteration 8500
11m 22s (- 39m 11s) (9000 22.50%) 4.5805
Validation Loss: 5.8752 at iteration 9000
12m 2s (- 38m 39s) (9500 23.75%) 4.6448
Validation Loss: 2.8273 at iteration 9500
12m 39s (- 37m 59s) (10000 25.00%) 4.6736
Validation Loss: 5.9556 at iteration 10000
13m 18s (- 37m 24s) (10500 26.25%) 4.6834
Validation Loss: 1.7144 at iteration 10500
13m 54s (- 36m 40s) (11000 27.50%) 4.5048
Validation Loss: 9.1039 at iteration 11000
14m 49s (- 36m 44s) (11500 28.75%) 4.6298
Validation Loss: 2.3660 at iteration 11500
15m 48s (- 36m 52s) (12000 30.00%) 4.6881
Validation Loss: 1.9253 at iteration 12000
16m 44s (- 36m 49s) (12500 31.25%) 4.6044
Validation Loss: 1.8871 at iteration 12500
17m 41s (- 36m 44s) (13000 32.50%) 4.6360
Validation Loss: 5.2728 at iteration 13000
18m 42s (- 36m 43s) (13500 33.75%) 4.5821
Validation Loss: 1.5446 at iteration 13500
19m 37s (- 36m 27s) (14000 35.00%) 4.5913
Validation Loss: 1.9007 at iteration 14000
20m 35s (- 36m 12s) (14500 36.25%) 4.5578
Validation Loss: 3.6497 at iteration 14500
21m 34s (- 35m 57s) (15000 37.50%) 4.6176
Validation Loss: 2.8972 at iteration 15000
22m 34s (- 35m 41s) (15500 38.75%) 4.6254
Validation Loss: 3.0278 at iteration 15500
23m 32s (- 35m 19s) (16000 40.00%) 4.5948
Validation Loss: 3.3687 at iteration 16000
24m 32s (- 34m 57s) (16500 41.25%) 4.7448
Validation Loss: 1.8933 at iteration 16500
25m 28s (- 34m 27s) (17000 42.50%) 4.7051
Validation Loss: 2.1642 at iteration 17000
26m 24s (- 33m 57s) (17500 43.75%) 4.5939
Validation Loss: 5.5429 at iteration 17500
27m 27s (- 33m 33s) (18000 45.00%) 4.5569
Validation Loss: 6.3712 at iteration 18000
28m 28s (- 33m 5s) (18500 46.25%) 4.6792
Validation Loss: 2.9571 at iteration 18500
29m 27s (- 32m 33s) (19000 47.50%) 4.6182
Validation Loss: 2.0924 at iteration 19000
30m 23s (- 31m 57s) (19500 48.75%) 4.5908
```

```
Validation Loss: 11.0523 at iteration 19500
31m 31s (- 31m 31s) (20000 50.00%) 4.4982
Validation Loss: 8.1792 at iteration 20000
32m 37s (- 31m 1s) (20500 51.25%) 4.6002
Validation Loss: 1.8402 at iteration 20500
33m 33s (- 30m 22s) (21000 52.50%) 4.5714
Validation Loss: 4.4725 at iteration 21000
34m 34s (- 29m 45s) (21500 53.75%) 4.5549
Validation Loss: 5.3819 at iteration 21500
35m 36s (- 29m 8s) (22000 55.00%) 4.5299
Validation Loss: 3.8305 at iteration 22000
36m 36s (- 28m 28s) (22500 56.25%) 4.5026
Validation Loss: 7.0824 at iteration 22500
37m 39s (- 27m 50s) (23000 57.50%) 4.6093
Validation Loss: 1.2505 at iteration 23000
38m 38s (- 27m 7s) (23500 58.75%) 4.4177
Validation Loss: 11.0871 at iteration 23500
39m 48s (- 26m 32s) (24000 60.00%) 4.5560
Validation Loss: 11.2957 at iteration 24000
40m 59s (- 25m 55s) (24500 61.25%) 4.4583
Validation Loss: 8.4185 at iteration 24500
42m 4s (- 25m 14s) (25000 62.50%) 4.5484
Validation Loss: 10.6660 at iteration 25000
43m 14s (- 24m 35s) (25500 63.75%) 4.6721
Validation Loss: 3.7042 at iteration 25500
44m 11s (- 23m 47s) (26000 65.00%) 4.6582
Validation Loss: 11.0120 at iteration 26000
45m 21s (- 23m 6s) (26500 66.25%) 4.5932
Validation Loss: 8.9205 at iteration 26500
46m 26s (- 22m 21s) (27000 67.50%) 4.5681
Validation Loss: 11.8032 at iteration 27000
47m 37s (- 21m 38s) (27500 68.75%) 4.6910
Validation Loss: 10.9378 at iteration 27500
48m 47s (- 20m 54s) (28000 70.00%) 4.6626
Validation Loss: 6.3156 at iteration 28000
49m 51s (- 20m 7s) (28500 71.25%) 4.5634
Validation Loss: 8.0881 at iteration 28500
50m 54s (- 19m 18s) (29000 72.50%) 4.4592
Validation Loss: 1.3703 at iteration 29000
51m 50s (- 18m 27s) (29500 73.75%) 4.6426
Validation Loss: 11.5387 at iteration 29500
53m 2s (- 17m 40s) (30000 75.00%) 4.5499
Validation Loss: 11.3821 at iteration 30000
54m 9s (- 16m 52s) (30500 76.25%) 4.6004
Validation Loss: 11.9320 at iteration 30500
55m 12s (- 16m 1s) (31000 77.50%) 4.5628
Validation Loss: 11.1271 at iteration 31000
56m 17s (- 15m 11s) (31500 78.75%) 4.5213
Validation Loss: 11.6993 at iteration 31500
57m 25s (- 14m 21s) (32000 80.00%) 4.5722
Validation Loss: 4.3269 at iteration 32000
58m 22s (- 13m 28s) (32500 81.25%) 4.6106
Validation Loss: 1.3170 at iteration 32500
59m 17s (- 12m 34s) (33000 82.50%) 4.5041
Validation Loss: 12.4726 at iteration 33000
60m 23s (- 11m 42s) (33500 83.75%) 4.5826
Validation Loss: 1.5965 at iteration 33500
61m 17s (- 10m 48s) (34000 85.00%) 4.5584
Validation Loss: 11.3257 at iteration 34000
62m 23s (- 9m 56s) (34500 86.25%) 4.4347
Validation Loss: 6.1703 at iteration 34500
63m 22s (- 9m 3s) (35000 87.50%) 4.5457
Validation Loss: 1.9455 at iteration 35000
64m 17s (- 8m 9s) (35500 88.75%) 4.5138
Validation Loss: 3.0190 at iteration 35500
65m 14s (- 7m 14s) (36000 90.00%) 4.6437
Validation Loss: 3.3580 at iteration 36000
66m 10s (- 6m 20s) (36500 91.25%) 4.5711
Validation Loss: 3.9862 at iteration 36500
67m 4s (- 5m 26s) (37000 92.50%) 4.5414
Validation Loss: 3.4321 at iteration 37000
67m 59s (- 4m 31s) (37500 93.75%) 4.5820
Validation Loss: 1.9324 at iteration 37500
68m 54s (- 3m 37s) (38000 95.00%) 4.5854
Validation Loss: 9.0068 at iteration 38000
69m 58s (- 2m 43s) (38500 96.25%) 4.6134
Validation Loss: 3.1443 at iteration 38500
```

```
70m 55s (- 1m 49s) (39000 97.50%) 4.6350
Validation Loss: 8.3306 at iteration 39000
71m 57s (- 0m 54s) (39500 98.75%) 4.5555
Validation Loss: 2.6765 at iteration 39500
72m 54s (- 0m 0s) (40000 100.00%) 4.7042
Validation Loss: 3.5261 at iteration 40000
```

Baseline 2: Sequence-to-Sequence model with attention

In [ ]:
```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class AttnDecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size, num_layers=1, dropout_p=0.1, max_length=150):
        super(AttnDecoderRNN, self).__init__()
        self.num_layers = num_layers
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.dropout_p = dropout_p
        self.max_length = max_length

        self.embedding = nn.Embedding(self.output_size, self.hidden_size)
        self.dropout = nn.Dropout(self.dropout_p)
        self.lstm = nn.LSTM(self.hidden_size, self.hidden_size, num_layers=num_layers, dropout=dropout_p if num_layers > 1 el
        self.out = nn.Linear(self.hidden_size * 2, self.output_size)

    def forward(self, input, hidden, cell, encoder_outputs):
        embedded = self.embedding(input).view(1, 1, -1)
        embedded = self.dropout(embedded)

        # print(hidden.size())
        # print(embedded.size())
        lstm_output, (hidden, cell) = self.lstm(embedded, (hidden, cell))

        attn_weights = F.softmax(torch.bmm(lstm_output, encoder_outputs.unsqueeze(0).permute(0, 2, 1)), dim=-1)
        attn_output = torch.bmm(attn_weights, encoder_outputs.unsqueeze(0))

        concat_output = torch.cat((attn_output[0], lstm_output[0]), 1)
        output = F.log_softmax(self.out(concat_output), dim=1)

        return output, hidden, cell, attn_weights

    def initHidden(self):
        return (torch.zeros(self.num_layers, 1, self.hidden_size, device=device),
                torch.zeros(self.num_layers, 1, self.hidden_size, device=device))
```

In [ ]:
```python
import torch
import random

teacher_forcing_ratio = 1

def train_attn_m2(input_tensor, target_tensor, encoder, decoder, encoder_optimizer, decoder_optimizer, criterion, max_length=
    encoder_hidden, encoder_cell = encoder.initHidden()

    encoder_optimizer.zero_grad()
    decoder_optimizer.zero_grad()

    input_length = input_tensor.size(0)
    target_length = target_tensor.size(0)
    # print("Encoder hidden state size:", encoder_hidden.size())
    # print("Encoder cell state size:", encoder_cell.size())

    encoder_outputs = torch.zeros(max_length, encoder.hidden_size, device=device)

    loss = 0

    for ei in range(input_length):
        encoder_output, (encoder_hidden, encoder_cell) = encoder(
            input_tensor[ei], (encoder_hidden, encoder_cell))
        encoder_outputs[ei] = encoder_output[0, 0]

    decoder_input = torch.tensor([[SOS_token]], device=device)

    decoder_hidden = encoder_hidden
    decoder_cell = encoder_cell

    # print("Decoder hidden state size:", decoder_hidden.size())
    # print("Decoder cell state size:", decoder_cell.size())

    use_teacher_forcing = True if random.random() < teacher_forcing_ratio else False

    if use_teacher_forcing:
        # Teacher forcing: Feed the target as the next input
        for di in range(target_length):
            decoder_output, decoder_hidden, decoder_cell, decoder_attention = decoder(
                decoder_input, decoder_hidden, decoder_cell, encoder_outputs)
            loss += criterion(decoder_output, target_tensor[di])
            decoder_input = target_tensor[di]  # Teacher forcing

    else:
        # Without teacher forcing: use its own predictions as the next input
        for di in range(target_length):
            decoder_output, decoder_hidden, decoder_cell, decoder_attention = decoder(
                decoder_input, decoder_hidden, decoder_cell, encoder_outputs)
            topv, topi = decoder_output.topk(1)
            decoder_input = topi.squeeze().detach()  # detach from history as input

            loss += criterion(decoder_output, target_tensor[di])
            if decoder_input.item() == EOS_token:
                break

    loss.backward()

    torch.nn.utils.clip_grad_norm_(encoder.parameters(), 1)
    torch.nn.utils.clip_grad_norm_(decoder.parameters(), 1)

    encoder_optimizer.step()
    decoder_optimizer.step()

    return loss.item() / target_length
```

In [ ]:

```
def evaluate_m2(input_tensor, target_tensor, encoder, decoder, criterion, max_length=150):
    encoder_hidden, encoder_cell = encoder.initHidden()
    input_length = input_tensor.size(0)
    target_length = target_tensor.size(0)
    encoder_outputs = torch.zeros(max_length, encoder.hidden_size, device=device)

    # Encoder
    for ei in range(input_length):
        # print(input_tensor[ei], encoder_hidden.size())
        encoder_output, (encoder_hidden, encoder_cell) = encoder(
            input_tensor[ei], (encoder_hidden, encoder_cell))
        if ei < max_length:
            encoder_outputs[ei] = encoder_output[0, 0]


    loss = 0

    # Decoder
    decoder_input = torch.tensor([[SOS_token]], device=device)
    decoder_hidden = encoder_hidden
    decoder_cell = encoder_cell

    for di in range(target_length):
        decoder_output, decoder_hidden, decoder_cell, decoder_attention = decoder(
            decoder_input, decoder_hidden, decoder_cell, encoder_outputs)
        topv, topi = decoder_output.topk(1)
        decoder_input = topi.squeeze().detach()  # Use its own predictions as the next input
        loss += criterion(decoder_output, target_tensor[di])
        if decoder_input.item() == EOS_token:
            break

    return loss.item() / target_length
```

In [ ]:

```
def validate_model_m2(encoder, decoder, val_pairs, criterion, max_length=MAX_LENGTH, num_samples=300):
    # Randomly select a subset of validation pairs
    if len(val_pairs) > num_samples:
        sampled_pairs = random.sample(val_pairs, num_samples)
    else:
        sampled_pairs = val_pairs
    total_loss = 0
    with torch.no_grad():
        for input_tensor, target_tensor in sampled_pairs:
            loss = evaluate_m2(input_tensor, target_tensor, encoder, decoder, criterion, max_length)
            total_loss += loss

    average_loss = total_loss / len(sampled_pairs)
    return average_loss
```

In [ ]:
```
from torch import optim

def trainIters_attn_m2(encoder, decoder, n_iters, print_every=1000, plot_every=100, learning_rate=0.01):
    start = time.time()
    plot_losses = []
    val_losses = []
    print_loss_total = 0  # Reset every print_every
    plot_loss_total = 0  # Reset every plot_every

    encoder_optimizer = optim.Adam(encoder.parameters(), lr=learning_rate)
    decoder_optimizer = optim.Adam(decoder.parameters(), lr=learning_rate)

    training_pairs = [tensorsFromPair(random.choice(train_pairs)) for _ in range(n_iters)]
    validation_pairs = [tensorsFromPair(random.choice(val_pairs))
                  for i in range(n_iters)]
    criterion = nn.NLLLoss()

    for iter in range(1, n_iters + 1):
        training_pair = training_pairs[iter - 1]
        input_tensor = training_pair[0]
        target_tensor = training_pair[1]

        loss = train_attn_m2(input_tensor, target_tensor, encoder, decoder, encoder_optimizer, decoder_optimizer, criterion)
        print_loss_total += loss
        plot_loss_total += loss

        if iter % print_every == 0:
            print_loss_avg = print_loss_total / print_every
            print_loss_total = 0
            print('%s (%d %d%%) %.4f' % (time_since(start, iter / n_iters),
                                         iter, iter / n_iters * 100, print_loss_avg))

        if iter % plot_every == 0:
            plot_loss_avg = plot_loss_total / plot_every
            plot_losses.append(plot_loss_avg)
            plot_loss_total = 0

            val_loss = validate_model_m2(encoder, decoder, validation_pairs, criterion)
            val_losses.append(val_loss)
            print(f'Validation Loss: {val_loss:.4f} at iteration {iter}')

    # showPlot(plot_losses)
    return plot_losses, val_losses
```

In [ ]:
```
hidden_size = 256
encoder_m2 = EncoderRNN(input_lang.n_words, hidden_size).to(device)
decoder_m2 = AttnDecoderRNN(hidden_size, output_lang.n_words, dropout_p=0.1).to(device)

train_losses_m2, val_losses_m2 = trainIters_attn_m2(encoder_m2, decoder_m2, 40000, print_every=500, plot_every=500)
```

```
1m 22s (- 108m 14s) (500 1%) 5.4518
Validation Loss: 1.0074 at iteration 500
2m 34s (- 100m 26s) (1000 2%) 4.8721
Validation Loss: 1.2710 at iteration 1000
3m 50s (- 98m 36s) (1500 3%) 4.6758
Validation Loss: 2.0906 at iteration 1500
5m 8s (- 97m 34s) (2000 5%) 4.6500
Validation Loss: 4.2636 at iteration 2000
6m 31s (- 97m 51s) (2500 6%) 4.5815
Validation Loss: 1.9344 at iteration 2500
7m 48s (- 96m 18s) (3000 7%) 4.5087
Validation Loss: 3.2414 at iteration 3000
9m 8s (- 95m 20s) (3500 8%) 4.4826
Validation Loss: 2.8676 at iteration 3500
10m 26s (- 93m 54s) (4000 10%) 4.5239
Validation Loss: 3.2182 at iteration 4000
11m 45s (- 92m 46s) (4500 11%) 4.4888
Validation Loss: 2.7802 at iteration 4500
13m 5s (- 91m 39s) (5000 12%) 4.3827
Validation Loss: 2.3352 at iteration 5000
14m 21s (- 90m 1s) (5500 13%) 4.5347
Validation Loss: 4.8827 at iteration 5500
15m 44s (- 89m 14s) (6000 15%) 4.3834
Validation Loss: 2.9620 at iteration 6000
17m 4s (- 87m 58s) (6500 16%) 4.2951
Validation Loss: 5.0535 at iteration 6500
18m 25s (- 86m 53s) (7000 17%) 4.4187
Validation Loss: 4.8413 at iteration 7000
19m 48s (- 85m 49s) (7500 18%) 4.5096
Validation Loss: 5.5891 at iteration 7500
21m 15s (- 85m 2s) (8000 20%) 4.2934
Validation Loss: 4.8001 at iteration 8000
22m 38s (- 83m 54s) (8500 21%) 4.2602
Validation Loss: 5.6117 at iteration 8500
24m 4s (- 82m 54s) (9000 22%) 4.3878
Validation Loss: 5.0995 at iteration 9000
25m 25s (- 81m 36s) (9500 23%) 4.3033
Validation Loss: 5.6512 at iteration 9500
26m 48s (- 80m 25s) (10000 25%) 4.2056
Validation Loss: 3.7456 at iteration 10000
28m 11s (- 79m 10s) (10500 26%) 4.2800
Validation Loss: 3.9120 at iteration 10500
29m 33s (- 77m 54s) (11000 27%) 4.2740
Validation Loss: 4.2586 at iteration 11000
30m 57s (- 76m 43s) (11500 28%) 4.3329
Validation Loss: 3.9715 at iteration 11500
32m 19s (- 75m 25s) (12000 30%) 4.3392
Validation Loss: 2.7210 at iteration 12000
33m 36s (- 73m 55s) (12500 31%) 4.2613
Validation Loss: 3.5416 at iteration 12500
34m 56s (- 72m 33s) (13000 32%) 4.2600
Validation Loss: 4.4769 at iteration 13000
36m 15s (- 71m 10s) (13500 33%) 4.2453
Validation Loss: 5.7029 at iteration 13500
37m 36s (- 69m 50s) (14000 35%) 4.3174
Validation Loss: 3.8637 at iteration 14000
38m 57s (- 68m 30s) (14500 36%) 4.2950
Validation Loss: 4.4481 at iteration 14500
40m 21s (- 67m 15s) (15000 37%) 4.2012
Validation Loss: 3.6802 at iteration 15000
41m 45s (- 66m 0s) (15500 38%) 4.2267
Validation Loss: 3.4939 at iteration 15500
43m 8s (- 64m 42s) (16000 40%) 4.3040
Validation Loss: 3.0134 at iteration 16000
44m 32s (- 63m 25s) (16500 41%) 4.2064
Validation Loss: 4.8644 at iteration 16500
45m 54s (- 62m 7s) (17000 42%) 4.1687
Validation Loss: 2.3992 at iteration 17000
47m 18s (- 60m 48s) (17500 43%) 4.1755
Validation Loss: 4.5401 at iteration 17500
48m 41s (- 59m 30s) (18000 45%) 4.1260
Validation Loss: 4.3365 at iteration 18000
50m 3s (- 58m 10s) (18500 46%) 4.2292
Validation Loss: 4.6901 at iteration 18500
51m 28s (- 56m 54s) (19000 47%) 4.2679
Validation Loss: 4.8011 at iteration 19000
52m 53s (- 55m 35s) (19500 48%) 4.2759
```

```
Validation Loss: 5.6371 at iteration 19500
54m 16s (- 54m 16s) (20000 50%) 4.1995
Validation Loss: 3.4412 at iteration 20000
55m 37s (- 52m 54s) (20500 51%) 4.1949
Validation Loss: 5.8580 at iteration 20500
57m 0s (- 51m 35s) (21000 52%) 4.2479
Validation Loss: 3.6210 at iteration 21000
58m 20s (- 50m 12s) (21500 53%) 4.2333
Validation Loss: 4.8035 at iteration 21500
59m 42s (- 48m 50s) (22000 55%) 4.2589
Validation Loss: 4.4879 at iteration 22000
61m 2s (- 47m 28s) (22500 56%) 4.1869
Validation Loss: 5.1587 at iteration 22500
62m 24s (- 46m 7s) (23000 57%) 4.2942
Validation Loss: 6.0860 at iteration 23000
63m 45s (- 44m 46s) (23500 58%) 4.1461
Validation Loss: 6.7740 at iteration 23500
64m 44s (- 43m 9s) (24000 60%) 4.2502
Validation Loss: 5.0771 at iteration 24000
65m 38s (- 41m 31s) (24500 61%) 4.2413
Validation Loss: 5.2742 at iteration 24500
66m 34s (- 39m 56s) (25000 62%) 4.1911
Validation Loss: 3.9061 at iteration 25000
67m 27s (- 38m 21s) (25500 63%) 4.2688
Validation Loss: 7.0292 at iteration 25500
68m 25s (- 36m 50s) (26000 65%) 4.2676
Validation Loss: 4.1542 at iteration 26000
69m 19s (- 35m 18s) (26500 66%) 4.2714
Validation Loss: 3.8104 at iteration 26500
70m 12s (- 33m 48s) (27000 67%) 4.2747
Validation Loss: 4.0334 at iteration 27000
71m 6s (- 32m 19s) (27500 68%) 4.1570
Validation Loss: 6.8998 at iteration 27500
72m 2s (- 30m 52s) (28000 70%) 4.1824
Validation Loss: 4.2719 at iteration 28000
72m 59s (- 29m 27s) (28500 71%) 4.2903
Validation Loss: 4.7108 at iteration 28500
73m 54s (- 28m 2s) (29000 72%) 4.2516
Validation Loss: 4.0133 at iteration 29000
74m 48s (- 26m 37s) (29500 73%) 4.3031
Validation Loss: 4.5598 at iteration 29500
75m 42s (- 25m 14s) (30000 75%) 4.2001
Validation Loss: 3.5316 at iteration 30000
76m 37s (- 23m 51s) (30500 76%) 4.2854
Validation Loss: 5.6733 at iteration 30500
77m 32s (- 22m 30s) (31000 77%) 4.2515
Validation Loss: 3.2811 at iteration 31000
78m 26s (- 21m 10s) (31500 78%) 4.2401
Validation Loss: 4.4686 at iteration 31500
79m 21s (- 19m 50s) (32000 80%) 4.2070
Validation Loss: 5.7917 at iteration 32000
80m 16s (- 18m 31s) (32500 81%) 4.3883
Validation Loss: 6.5113 at iteration 32500
81m 10s (- 17m 13s) (33000 82%) 4.2775
Validation Loss: 4.2302 at iteration 33000
82m 3s (- 15m 55s) (33500 83%) 4.3352
Validation Loss: 4.3510 at iteration 33500
82m 55s (- 14m 37s) (34000 85%) 4.2732
Validation Loss: 4.9674 at iteration 34000
83m 49s (- 13m 21s) (34500 86%) 4.2239
Validation Loss: 4.7708 at iteration 34500
84m 43s (- 12m 6s) (35000 87%) 4.2863
Validation Loss: 2.7811 at iteration 35000
85m 34s (- 10m 50s) (35500 88%) 4.1844
Validation Loss: 3.7298 at iteration 35500
86m 25s (- 9m 36s) (36000 90%) 4.2314
Validation Loss: 3.3363 at iteration 36000
87m 16s (- 8m 22s) (36500 91%) 4.2262
Validation Loss: 4.5114 at iteration 36500
88m 8s (- 7m 8s) (37000 92%) 4.2611
Validation Loss: 3.6438 at iteration 37000
88m 59s (- 5m 55s) (37500 93%) 4.3872
Validation Loss: 3.7393 at iteration 37500
89m 50s (- 4m 43s) (38000 95%) 4.1994
Validation Loss: 4.7592 at iteration 38000
90m 43s (- 3m 32s) (38500 96%) 4.2916
Validation Loss: 5.5433 at iteration 38500
```

```
91m 37s (- 2m 20s) (39000 97%) 4.4168
Validation Loss: 4.8480 at iteration 39000
92m 31s (- 1m 10s) (39500 98%) 4.3388
Validation Loss: 5.7028 at iteration 39500
93m 26s (- 0m 0s) (40000 100%) 4.2571
Validation Loss: 6.1473 at iteration 40000
```

Extend Model 1: Using pretrained embeddings (GLoVe)

In [ ]:
```python
class GloveDataset:
    def __init__(self, input_lang, output_lang, pairs, window_size=2, embedding_dim=50):
      self.input_lang = input_lang
      self.output_lang = output_lang
      self.pairs = pairs
      self._window_size = window_size

      #self._tokens = word_tokenize(text)
      input_text = ' '.join([' '.join(pair[0].split()) for pair in pairs])
      output_text = ' '.join([' '.join(pair[1].split()) for pair in pairs])
      text = input_text + ' ' + output_text

      self._tokens = text.split(" ")

      word_counter = Counter()
      word_counter.update(self._tokens)
      self._word2id = {w:i for i, (w,_) in enumerate(word_counter.most_common())}
      self._id2word = {i:w for w, i in self._word2id.items()}
      self._vocab_len = len(self._word2id)

      self._id_tokens = [self._word2id[w] for w in self._tokens]

      self._create_coocurrence_matrix()

      print("# of words: {}".format(len(self._tokens)))
      print("Vocabulary length: {}".format(self._vocab_len))

      # self.embeddings = self.load_glove_embeddings(glove_path, embedding_dim)

    def _create_coocurrence_matrix(self):
        # initialize a default dictionary where each key maps to a 'counter'
        cooc_mat = defaultdict(Counter)
        # iterate over each token and its index
        for i, w in enumerate(self._id_tokens):
            # define the context window around the current token
            start_i = max(i - self._window_size, 0)
            end_i = min(i + self._window_size + 1, len(self._id_tokens))

            # iterate over the context window
            for j in range(start_i, end_i):
                # ensure the current token is not counted as its own context
                if i != j:
                    # get the context token
                    c = self._id_tokens[j]
                    # update the co-occurrence count, weighted by the incerse of the distance between tokens
                    cooc_mat[w][c] += 1 / abs(j-i)

        # initialize lists to store the indices and co-occurrence values
        self._i_idx = list()
        self._j_idx = list()
        self._xij = list()

        #Create indexes and x values tensors
        # loop through co-occurrence matrix
        for w, cnt in cooc_mat.items():
            for c, v in cnt.items():
                # append the word index, context word index, co-occurrence count
                self._i_idx.append(w)
                self._j_idx.append(c)
                self._xij.append(v)

        self._i_idx = torch.LongTensor(self._i_idx).to(device)
        self._j_idx = torch.LongTensor(self._j_idx).to(device)
        self._xij = torch.FloatTensor(self._xij).to(device)
    def get_batches(self, batch_size):
        "generate batches of data"
        #Generate random idx
        rand_ids = torch.LongTensor(np.random.choice(len(self._xij), len(self._xij), replace=False))

        # iterate over the shuffled indices in steps of 'batch_size'
        for p in range(0, len(rand_ids), batch_size):
            # get the current batch of indices
            batch_ids = rand_ids[p:p+batch_size]
            # yield the co-occurrence values and corresponding indices for the current batch
            yield self._xij[batch_ids], self._i_idx[batch_ids], self._j_idx[batch_ids]
```

In [ ]:
```
class GloveModel(nn.Module):
    def __init__(self, num_embeddings, embedding_dim):
        # call the constructor of the parent class (nn.module)
        super(GloveModel, self).__init__()

        # create an embedding layer
        # num_embeddings: the size of the vocabulary
        # embedding_dim: the dimension of the embedding vectors
        # wi: target word
        # wj: context word
        # bi: bias terms of target words
        # bj: bias terms of context words
        self.wi = nn.Embedding(num_embeddings, embedding_dim)
        self.wj = nn.Embedding(num_embeddings, embedding_dim)
        self.bi = nn.Embedding(num_embeddings, 1)
        self.bj = nn.Embedding(num_embeddings, 1)

        # initialize the weight
        self.wi.weight.data.uniform_(-1, 1)
        self.wj.weight.data.uniform_(-1, 1)
        self.bi.weight.data.zero_()
        self.bj.weight.data.zero_()

    def forward(self, i_indices, j_indices):
        # retrives the embedding vectors
        w_i = self.wi(i_indices)
        w_j = self.wj(j_indices)
        b_i = self.bi(i_indices).squeeze()
        b_j = self.bj(j_indices).squeeze()

        # compute the dot product of the target and context embedding, then add bias terms
        x = torch.sum(w_i * w_j, dim=1) + b_i + b_j

        # predicted log of the co-occurrence count for each word pair
        return x
```

In [ ]:
```
def weight_func(x, x_max, alpha):
    # wx: tensor of weights corresponding to the co-occurrence counts
    wx = (x/x_max)**alpha
    # weights in the range [0, 1]
    wx = torch.min(wx, torch.ones_like(wx))
    if cuda_available:
        return wx.cuda()
    else:
        return wx


def wmse_loss(weights, inputs, targets):
    # element-wise MSE loss between input and output, multiply weight
    loss = weights * F.mse_loss(inputs, targets, reduction='none')
    if cuda_available:
        return torch.mean(loss).cuda()
    else:
        return torch.mean(loss)
```

In [ ]:
```
# initialize GloveDataset
# loads the text, tokenizes it, build the vocabulary, create co-occurrence matrix
dataset = GloveDataset(input_lang, output_lang, train_pairs)
# set embedding dimension to 50
EMBED_DIM = 50

# initialize the glove model with vocabulary size and embedding dimension
glove = GloveModel(dataset._vocab_len, EMBED_DIM).to(device)
```

```
# initializes the Adagrad optimizer with the model parameters, learning rate = 0.05
optimizer = optim.Adagrad(glove.parameters(), lr=0.05)

#training
# number of training epochs
N_EPOCHS = 10
# batvh size
BATCH_SIZE = 500
# max value for weighting function normalization
X_MAX = 100
# exponent used in the weighting function
ALPHA = 0.75
# calculate the number of batches per epoch
n_batches = int(len(dataset._xij) / BATCH_SIZE)
# initializes an empty list to store the loss values
loss_values = list()

# iterates over number of epochs
for e in range(1, N_EPOCHS+1):
    batch_i = 0

    for x_ij, i_idx, j_idx in dataset.get_batches(BATCH_SIZE):

        batch_i += 1

        optimizer.zero_grad()

        outputs = glove(i_idx, j_idx)
        weights_x = weight_func(x_ij, X_MAX, ALPHA)
        loss = wmse_loss(weights_x, outputs, torch.log(x_ij))

        loss.backward()

        optimizer.step()

        loss_values.append(loss.item())

        if batch_i % 100 == 0:
            print("Epoch: {}/{} \t Batch: {}/{} \t Loss: {}".format(e, N_EPOCHS, batch_i, n_batches, np.mean(loss_values[-20:

plt.plot(loss_values)
print("Saving model...")
torch.save(glove.state_dict(), "GloVe.pt")
```

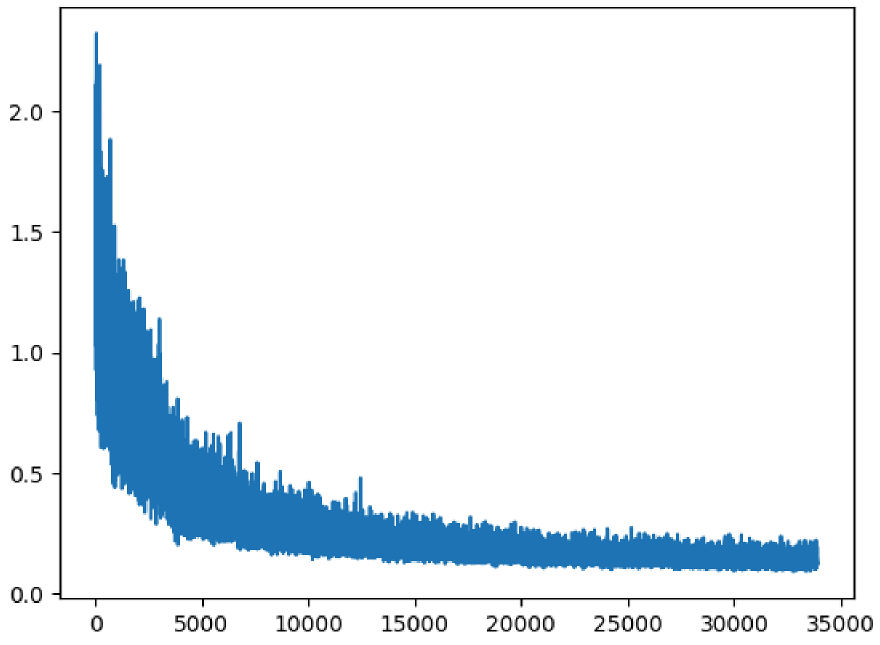
```
# of words: 9623466
Vocabulary length: 34455
Epoch: 1/10      Batch: 100/3389         Loss: 1.3114466488361358
Epoch: 1/10      Batch: 200/3389         Loss: 1.1533688008785248
Epoch: 1/10      Batch: 300/3389         Loss: 1.2073701798915863
Epoch: 1/10      Batch: 400/3389         Loss: 1.1062311053276062
Epoch: 1/10      Batch: 500/3389         Loss: 1.0919590711593627
Epoch: 1/10      Batch: 600/3389         Loss: 1.0618666648864745
Epoch: 1/10      Batch: 700/3389         Loss: 1.0188938587903977
Epoch: 1/10      Batch: 800/3389         Loss: 0.8842014670372009
Epoch: 1/10      Batch: 900/3389         Loss: 0.8945489093661309
Epoch: 1/10      Batch: 1000/3389        Loss: 0.8445174753665924
Epoch: 1/10      Batch: 1100/3389        Loss: 0.9426991760730743
Epoch: 1/10      Batch: 1200/3389        Loss: 0.8252364248037338
Epoch: 1/10      Batch: 1300/3389        Loss: 0.8086512446403503
Epoch: 1/10      Batch: 1400/3389        Loss: 0.9087293475866318
Epoch: 1/10      Batch: 1500/3389        Loss: 0.7914608299732209
Epoch: 1/10      Batch: 1600/3389        Loss: 0.7813426837325096
Epoch: 1/10      Batch: 1700/3389        Loss: 0.7162507086992264
Epoch: 1/10      Batch: 1800/3389        Loss: 0.7600308746099472
Epoch: 1/10      Batch: 1900/3389        Loss: 0.7146928951144218
Epoch: 1/10      Batch: 2000/3389        Loss: 0.7561000794172287
Epoch: 1/10      Batch: 2100/3389        Loss: 0.7022433832287789
Epoch: 1/10      Batch: 2200/3389        Loss: 0.6759865343570709
Epoch: 1/10      Batch: 2300/3389        Loss: 0.6434744283556938
Epoch: 1/10      Batch: 2400/3389        Loss: 0.6271590381860733
Epoch: 1/10      Batch: 2500/3389        Loss: 0.6452850729227066
Epoch: 1/10      Batch: 2600/3389        Loss: 0.6051141962409019
Epoch: 1/10      Batch: 2700/3389        Loss: 0.5939026519656181
Epoch: 1/10      Batch: 2800/3389        Loss: 0.6050391614437103
Epoch: 1/10      Batch: 2900/3389        Loss: 0.6004761800169944
Epoch: 1/10      Batch: 3000/3389        Loss: 0.5790632471442223
Epoch: 1/10      Batch: 3100/3389        Loss: 0.5931592538952828
Epoch: 1/10      Batch: 3200/3389        Loss: 0.5871605455875397
Epoch: 1/10      Batch: 3300/3389        Loss: 0.5402861818671226
Epoch: 2/10      Batch: 100/3389         Loss: 0.48704167157411576
Epoch: 2/10      Batch: 200/3389         Loss: 0.470212011039257
Epoch: 2/10      Batch: 300/3389         Loss: 0.4138426333665848
Epoch: 2/10      Batch: 400/3389         Loss: 0.40430945456027984
Epoch: 2/10      Batch: 500/3389         Loss: 0.38814754858613015
Epoch: 2/10      Batch: 600/3389         Loss: 0.4147121965885162
Epoch: 2/10      Batch: 700/3389         Loss: 0.43174156844615935
Epoch: 2/10      Batch: 800/3389         Loss: 0.4170937806367874
Epoch: 2/10      Batch: 900/3389         Loss: 0.42264667302370074
Epoch: 2/10      Batch: 1000/3389        Loss: 0.41748942583799364
Epoch: 2/10      Batch: 1100/3389        Loss: 0.3777651578187943
Epoch: 2/10      Batch: 1200/3389        Loss: 0.41662627160549165
Epoch: 2/10      Batch: 1300/3389        Loss: 0.3817656487226486
Epoch: 2/10      Batch: 1400/3389        Loss: 0.37622717171907427
Epoch: 2/10      Batch: 1500/3389        Loss: 0.3696301504969597
Epoch: 2/10      Batch: 1600/3389        Loss: 0.3815418154001236
Epoch: 2/10      Batch: 1700/3389        Loss: 0.3958295792341232
Epoch: 2/10      Batch: 1800/3389        Loss: 0.3836063235998154
Epoch: 2/10      Batch: 1900/3389        Loss: 0.3866722106933594
Epoch: 2/10      Batch: 2000/3389        Loss: 0.3522880956530571
Epoch: 2/10      Batch: 2100/3389        Loss: 0.3897036015987396
Epoch: 2/10      Batch: 2200/3389        Loss: 0.3739802084863186
Epoch: 2/10      Batch: 2300/3389        Loss: 0.3713572219014168
Epoch: 2/10      Batch: 2400/3389        Loss: 0.4067389518022537
Epoch: 2/10      Batch: 2500/3389        Loss: 0.3505144461989403
Epoch: 2/10      Batch: 2600/3389        Loss: 0.32881829887628555
Epoch: 2/10      Batch: 2700/3389        Loss: 0.3442339576780796
Epoch: 2/10      Batch: 2800/3389        Loss: 0.3474142298102379
Epoch: 2/10      Batch: 2900/3389        Loss: 0.34648687541484835
Epoch: 2/10      Batch: 3000/3389        Loss: 0.3391000680625439
Epoch: 2/10      Batch: 3100/3389        Loss: 0.33608212471008303
Epoch: 2/10      Batch: 3200/3389        Loss: 0.3266866967082024
Epoch: 2/10      Batch: 3300/3389        Loss: 0.33258806318044665
Epoch: 3/10      Batch: 100/3389         Loss: 0.30140961036086084
Epoch: 3/10      Batch: 200/3389         Loss: 0.3270683281123638
Epoch: 3/10      Batch: 300/3389         Loss: 0.2687843918800354
Epoch: 3/10      Batch: 400/3389         Loss: 0.287013154476881
Epoch: 3/10      Batch: 500/3389         Loss: 0.2897579610347748
Epoch: 3/10      Batch: 600/3389         Loss: 0.29354787319898606
Epoch: 3/10      Batch: 700/3389         Loss: 0.28070811927318573
Epoch: 3/10      Batch: 800/3389         Loss: 0.2885786212980747
Epoch: 3/10      Batch: 900/3389         Loss: 0.2729181870818138
```

```
Epoch: 3/10      Batch: 1000/3389      Loss: 0.28691752105951307
Epoch: 3/10      Batch: 1100/3389      Loss: 0.2703740008175373
Epoch: 3/10      Batch: 1200/3389      Loss: 0.2787871040403843
Epoch: 3/10      Batch: 1300/3389      Loss: 0.2744743123650551
Epoch: 3/10      Batch: 1400/3389      Loss: 0.25876803398132325
Epoch: 3/10      Batch: 1500/3389      Loss: 0.26206022277474406
Epoch: 3/10      Batch: 1600/3389      Loss: 0.27264114916324617
Epoch: 3/10      Batch: 1700/3389      Loss: 0.29274541363120077
Epoch: 3/10      Batch: 1800/3389      Loss: 0.2628495037555695
Epoch: 3/10      Batch: 1900/3389      Loss: 0.27566052600741386
Epoch: 3/10      Batch: 2000/3389      Loss: 0.27108135297894476
Epoch: 3/10      Batch: 2100/3389      Loss: 0.27568008899688723
Epoch: 3/10      Batch: 2200/3389      Loss: 0.281137989461422
Epoch: 3/10      Batch: 2300/3389      Loss: 0.2539467096328735
Epoch: 3/10      Batch: 2400/3389      Loss: 0.2782115273177624
Epoch: 3/10      Batch: 2500/3389      Loss: 0.25925354063510897
Epoch: 3/10      Batch: 2600/3389      Loss: 0.2745399706065655
Epoch: 3/10      Batch: 2700/3389      Loss: 0.2670555576682091
Epoch: 3/10      Batch: 2800/3389      Loss: 0.2649217896163464
Epoch: 3/10      Batch: 2900/3389      Loss: 0.267499178647995
Epoch: 3/10      Batch: 3000/3389      Loss: 0.2482533760368824
Epoch: 3/10      Batch: 3100/3389      Loss: 0.25764927789568903
Epoch: 3/10      Batch: 3200/3389      Loss: 0.2462947830557823
Epoch: 3/10      Batch: 3300/3389      Loss: 0.27346240505576136
Epoch: 4/10      Batch: 100/3389       Loss: 0.2358957588672638
Epoch: 4/10      Batch: 200/3389       Loss: 0.22507194802165031
Epoch: 4/10      Batch: 300/3389       Loss: 0.22285090610384942
Epoch: 4/10      Batch: 400/3389       Loss: 0.24191173836588858
Epoch: 4/10      Batch: 500/3389       Loss: 0.232894966006279
Epoch: 4/10      Batch: 600/3389       Loss: 0.23276684507727624
Epoch: 4/10      Batch: 700/3389       Loss: 0.2378115952014923
Epoch: 4/10      Batch: 800/3389       Loss: 0.24500413089990616
Epoch: 4/10      Batch: 900/3389       Loss: 0.23032892048358916
Epoch: 4/10      Batch: 1000/3389      Loss: 0.22571853026747704
Epoch: 4/10      Batch: 1100/3389      Loss: 0.22723521888256074
Epoch: 4/10      Batch: 1200/3389      Loss: 0.23761889487504959
Epoch: 4/10      Batch: 1300/3389      Loss: 0.2355004370212555
Epoch: 4/10      Batch: 1400/3389      Loss: 0.2272703155875206
Epoch: 4/10      Batch: 1500/3389      Loss: 0.21395558267831802
Epoch: 4/10      Batch: 1600/3389      Loss: 0.2345266245305538
Epoch: 4/10      Batch: 1700/3389      Loss: 0.22524700984358786
Epoch: 4/10      Batch: 1800/3389      Loss: 0.22113099247217177
Epoch: 4/10      Batch: 1900/3389      Loss: 0.2219391167163849
Epoch: 4/10      Batch: 2000/3389      Loss: 0.21772745177149772
Epoch: 4/10      Batch: 2100/3389      Loss: 0.23250899836421013
Epoch: 4/10      Batch: 2200/3389      Loss: 0.22424327135086058
Epoch: 4/10      Batch: 2300/3389      Loss: 0.22690935656428338
Epoch: 4/10      Batch: 2400/3389      Loss: 0.22379335165023803
Epoch: 4/10      Batch: 2500/3389      Loss: 0.221271163970232
Epoch: 4/10      Batch: 2600/3389      Loss: 0.22530099302530288
Epoch: 4/10      Batch: 2700/3389      Loss: 0.2149711109697819
Epoch: 4/10      Batch: 2800/3389      Loss: 0.22414958402514457
Epoch: 4/10      Batch: 2900/3389      Loss: 0.2220744803547859
Epoch: 4/10      Batch: 3000/3389      Loss: 0.211798794567585
Epoch: 4/10      Batch: 3100/3389      Loss: 0.21400730013847352
Epoch: 4/10      Batch: 3200/3389      Loss: 0.22558965384960175
Epoch: 4/10      Batch: 3300/3389      Loss: 0.21749341636896133
Epoch: 5/10      Batch: 100/3389       Loss: 0.2113370805978775
Epoch: 5/10      Batch: 200/3389       Loss: 0.2070684477686882
Epoch: 5/10      Batch: 300/3389       Loss: 0.21979182288050653
Epoch: 5/10      Batch: 400/3389       Loss: 0.21503524333238602
Epoch: 5/10      Batch: 500/3389       Loss: 0.2168586529791355
Epoch: 5/10      Batch: 600/3389       Loss: 0.18466252088546753
Epoch: 5/10      Batch: 700/3389       Loss: 0.19001130610704423
Epoch: 5/10      Batch: 800/3389       Loss: 0.19976432174444197
Epoch: 5/10      Batch: 900/3389       Loss: 0.18808198496699333
Epoch: 5/10      Batch: 1000/3389      Loss: 0.20305225253105164
Epoch: 5/10      Batch: 1100/3389      Loss: 0.19423335716128348
Epoch: 5/10      Batch: 1200/3389      Loss: 0.1969269596040249
Epoch: 5/10      Batch: 1300/3389      Loss: 0.19089270234107972
Epoch: 5/10      Batch: 1400/3389      Loss: 0.21013243719935418
Epoch: 5/10      Batch: 1500/3389      Loss: 0.19398750960826874
Epoch: 5/10      Batch: 1600/3389      Loss: 0.18977291285991668
Epoch: 5/10      Batch: 1700/3389      Loss: 0.18508658185601234
Epoch: 5/10      Batch: 1800/3389      Loss: 0.1927406132221222
Epoch: 5/10      Batch: 1900/3389      Loss: 0.18804785013198852
Epoch: 5/10      Batch: 2000/3389      Loss: 0.19682723060250282
```

```
Epoch: 5/10      Batch: 2100/3389        Loss: 0.2018749162554741
Epoch: 5/10      Batch: 2200/3389        Loss: 0.18933161199092866
Epoch: 5/10      Batch: 2300/3389        Loss: 0.19420884177088737
Epoch: 5/10      Batch: 2400/3389        Loss: 0.21281322091817856
Epoch: 5/10      Batch: 2500/3389        Loss: 0.2080029897391796
Epoch: 5/10      Batch: 2600/3389        Loss: 0.18765875250101088
Epoch: 5/10      Batch: 2700/3389        Loss: 0.17723194807767867
Epoch: 5/10      Batch: 2800/3389        Loss: 0.190533534437418
Epoch: 5/10      Batch: 2900/3389        Loss: 0.1918628640472889
Epoch: 5/10      Batch: 3000/3389        Loss: 0.1908055104315281
Epoch: 5/10      Batch: 3100/3389        Loss: 0.19673240110278128
Epoch: 5/10      Batch: 3200/3389        Loss: 0.19827969819307328
Epoch: 5/10      Batch: 3300/3389        Loss: 0.18678686022758484
Epoch: 6/10      Batch: 100/3389         Loss: 0.1846492074429989
Epoch: 6/10      Batch: 200/3389         Loss: 0.18179413452744483
Epoch: 6/10      Batch: 300/3389         Loss: 0.18266999423503877
Epoch: 6/10      Batch: 400/3389         Loss: 0.1853642113506794
Epoch: 6/10      Batch: 500/3389         Loss: 0.18249445110559465
Epoch: 6/10      Batch: 600/3389         Loss: 0.1826523594558239
Epoch: 6/10      Batch: 700/3389         Loss: 0.18304111436009407
Epoch: 6/10      Batch: 800/3389         Loss: 0.17248538583517076
Epoch: 6/10      Batch: 900/3389         Loss: 0.1788921982049942
Epoch: 6/10      Batch: 1000/3389        Loss: 0.18008704259991645
Epoch: 6/10      Batch: 1100/3389        Loss: 0.16737425774335862
Epoch: 6/10      Batch: 1200/3389        Loss: 0.17751829624176024
Epoch: 6/10      Batch: 1300/3389        Loss: 0.18210765421390535
Epoch: 6/10      Batch: 1400/3389        Loss: 0.1932343177497387
Epoch: 6/10      Batch: 1500/3389        Loss: 0.18469193056225777
Epoch: 6/10      Batch: 1600/3389        Loss: 0.176373852789402
Epoch: 6/10      Batch: 1700/3389        Loss: 0.16972544565796852
Epoch: 6/10      Batch: 1800/3389        Loss: 0.18004323169589043
Epoch: 6/10      Batch: 1900/3389        Loss: 0.18792436718940736
Epoch: 6/10      Batch: 2000/3389        Loss: 0.17709023952484132
Epoch: 6/10      Batch: 2100/3389        Loss: 0.16952313706278802
Epoch: 6/10      Batch: 2200/3389        Loss: 0.1823703147470951
Epoch: 6/10      Batch: 2300/3389        Loss: 0.1715390458703041
Epoch: 6/10      Batch: 2400/3389        Loss: 0.175603049993515
Epoch: 6/10      Batch: 2500/3389        Loss: 0.17952572032809258
Epoch: 6/10      Batch: 2600/3389        Loss: 0.17131135873496534
Epoch: 6/10      Batch: 2700/3389        Loss: 0.1722335435450077
Epoch: 6/10      Batch: 2800/3389        Loss: 0.17029365301132202
Epoch: 6/10      Batch: 2900/3389        Loss: 0.16951109543442727
Epoch: 6/10      Batch: 3000/3389        Loss: 0.16817160621285437
Epoch: 6/10      Batch: 3100/3389        Loss: 0.1727713890373707
Epoch: 6/10      Batch: 3200/3389        Loss: 0.18013615906238556
Epoch: 6/10      Batch: 3300/3389        Loss: 0.17727765813469887
Epoch: 7/10      Batch: 100/3389         Loss: 0.17156531438231468
Epoch: 7/10      Batch: 200/3389         Loss: 0.17617973387241365
Epoch: 7/10      Batch: 300/3389         Loss: 0.16110249385237693
Epoch: 7/10      Batch: 400/3389         Loss: 0.17500016689300538
Epoch: 7/10      Batch: 500/3389         Loss: 0.16990993395447732
Epoch: 7/10      Batch: 600/3389         Loss: 0.16779637336730957
Epoch: 7/10      Batch: 700/3389         Loss: 0.16092430464923382
Epoch: 7/10      Batch: 800/3389         Loss: 0.15994762629270554
Epoch: 7/10      Batch: 900/3389         Loss: 0.16742865070700647
Epoch: 7/10      Batch: 1000/3389        Loss: 0.1689802050590515
Epoch: 7/10      Batch: 1100/3389        Loss: 0.17162931114435195
Epoch: 7/10      Batch: 1200/3389        Loss: 0.1756255753338337
Epoch: 7/10      Batch: 1300/3389        Loss: 0.17654299549758434
Epoch: 7/10      Batch: 1400/3389        Loss: 0.1638149507343769
Epoch: 7/10      Batch: 1500/3389        Loss: 0.1620418518781662
Epoch: 7/10      Batch: 1600/3389        Loss: 0.17176548838615419
Epoch: 7/10      Batch: 1700/3389        Loss: 0.1661794688552618
Epoch: 7/10      Batch: 1800/3389        Loss: 0.15709717459976674
Epoch: 7/10      Batch: 1900/3389        Loss: 0.16089645773172379
Epoch: 7/10      Batch: 2000/3389        Loss: 0.16189536303281785
Epoch: 7/10      Batch: 2100/3389        Loss: 0.17578255273401738
Epoch: 7/10      Batch: 2200/3389        Loss: 0.1582667574286461
Epoch: 7/10      Batch: 2300/3389        Loss: 0.16776821240782738
Epoch: 7/10      Batch: 2400/3389        Loss: 0.16707131192088126
Epoch: 7/10      Batch: 2500/3389        Loss: 0.15813305079936982
Epoch: 7/10      Batch: 2600/3389        Loss: 0.1585428636521101
Epoch: 7/10      Batch: 2700/3389        Loss: 0.15970255732536315
Epoch: 7/10      Batch: 2800/3389        Loss: 0.16335244625806808
Epoch: 7/10      Batch: 2900/3389        Loss: 0.15230275318026543
Epoch: 7/10      Batch: 3000/3389        Loss: 0.1638004034757614
Epoch: 7/10      Batch: 3100/3389        Loss: 0.17323277443647384
```

```
Epoch: 7/10      Batch: 3200/3389        Loss: 0.16788602173328399
Epoch: 7/10      Batch: 3300/3389        Loss: 0.17096176072955133
Epoch: 8/10      Batch: 100/3389         Loss: 0.15621763467788696
Epoch: 8/10      Batch: 200/3389         Loss: 0.15594348609447478
Epoch: 8/10      Batch: 300/3389         Loss: 0.1606444850564003
Epoch: 8/10      Batch: 400/3389         Loss: 0.15560464411973954
Epoch: 8/10      Batch: 500/3389         Loss: 0.1523997776210308
Epoch: 8/10      Batch: 600/3389         Loss: 0.1444482583552599
Epoch: 8/10      Batch: 700/3389         Loss: 0.16134921833872795
Epoch: 8/10      Batch: 800/3389         Loss: 0.1547511238604784
Epoch: 8/10      Batch: 900/3389         Loss: 0.14585658721625805
Epoch: 8/10      Batch: 1000/3389        Loss: 0.15183352902531624
Epoch: 8/10      Batch: 1100/3389        Loss: 0.15499440655112268
Epoch: 8/10      Batch: 1200/3389        Loss: 0.16562280654907227
Epoch: 8/10      Batch: 1300/3389        Loss: 0.15501788891851903
Epoch: 8/10      Batch: 1400/3389        Loss: 0.14706580117344856
Epoch: 8/10      Batch: 1500/3389        Loss: 0.15375151112675667
Epoch: 8/10      Batch: 1600/3389        Loss: 0.15620943903923035
Epoch: 8/10      Batch: 1700/3389        Loss: 0.15993837863206864
Epoch: 8/10      Batch: 1800/3389        Loss: 0.16513758786022664
Epoch: 8/10      Batch: 1900/3389        Loss: 0.16268682032823562
Epoch: 8/10      Batch: 2000/3389        Loss: 0.15323004983365535
Epoch: 8/10      Batch: 2100/3389        Loss: 0.1535791978240013
Epoch: 8/10      Batch: 2200/3389        Loss: 0.1526669505983591
Epoch: 8/10      Batch: 2300/3389        Loss: 0.15019438937306404
Epoch: 8/10      Batch: 2400/3389        Loss: 0.15908039063215257
Epoch: 8/10      Batch: 2500/3389        Loss: 0.16202878206968307
Epoch: 8/10      Batch: 2600/3389        Loss: 0.15807702839374543
Epoch: 8/10      Batch: 2700/3389        Loss: 0.1557417158037424
Epoch: 8/10      Batch: 2800/3389        Loss: 0.15121881365776063
Epoch: 8/10      Batch: 2900/3389        Loss: 0.158225104957819
Epoch: 8/10      Batch: 3000/3389        Loss: 0.1674963690340519
Epoch: 8/10      Batch: 3100/3389        Loss: 0.15641362294554711
Epoch: 8/10      Batch: 3200/3389        Loss: 0.15583414621651173
Epoch: 8/10      Batch: 3300/3389        Loss: 0.15929254405200483
Epoch: 9/10      Batch: 100/3389         Loss: 0.14814992472529412
Epoch: 9/10      Batch: 200/3389         Loss: 0.14864846877753735
Epoch: 9/10      Batch: 300/3389         Loss: 0.1451479233801365
Epoch: 9/10      Batch: 400/3389         Loss: 0.14432041794061662
Epoch: 9/10      Batch: 500/3389         Loss: 0.14764039479196073
Epoch: 9/10      Batch: 600/3389         Loss: 0.14448228850960732
Epoch: 9/10      Batch: 700/3389         Loss: 0.15065944492816924
Epoch: 9/10      Batch: 800/3389         Loss: 0.14289154075086116
Epoch: 9/10      Batch: 900/3389         Loss: 0.14591547548770906
Epoch: 9/10      Batch: 1000/3389        Loss: 0.14580020159482956
Epoch: 9/10      Batch: 1100/3389        Loss: 0.14265390075743198
Epoch: 9/10      Batch: 1200/3389        Loss: 0.1395305074751377
Epoch: 9/10      Batch: 1300/3389        Loss: 0.15394266918301583
Epoch: 9/10      Batch: 1400/3389        Loss: 0.1518335159868002
Epoch: 9/10      Batch: 1500/3389        Loss: 0.1476492777466774
Epoch: 9/10      Batch: 1600/3389        Loss: 0.14756182655692102
Epoch: 9/10      Batch: 1700/3389        Loss: 0.14394672363996505
Epoch: 9/10      Batch: 1800/3389        Loss: 0.1469017591327429
Epoch: 9/10      Batch: 1900/3389        Loss: 0.1488670241087675
Epoch: 9/10      Batch: 2000/3389        Loss: 0.14813337549567224
Epoch: 9/10      Batch: 2100/3389        Loss: 0.14275326915085315
Epoch: 9/10      Batch: 2200/3389        Loss: 0.14741081073880197
Epoch: 9/10      Batch: 2300/3389        Loss: 0.1388707034289837
Epoch: 9/10      Batch: 2400/3389        Loss: 0.1502829946577549
Epoch: 9/10      Batch: 2500/3389        Loss: 0.1434073518961668
Epoch: 9/10      Batch: 2600/3389        Loss: 0.1432145431637764
Epoch: 9/10      Batch: 2700/3389        Loss: 0.13906763270497322
Epoch: 9/10      Batch: 2800/3389        Loss: 0.14730155989527702
Epoch: 9/10      Batch: 2900/3389        Loss: 0.1471054721623659
Epoch: 9/10      Batch: 3000/3389        Loss: 0.14755289033055305
Epoch: 9/10      Batch: 3100/3389        Loss: 0.1500639509409666
Epoch: 9/10      Batch: 3200/3389        Loss: 0.1458226628601551
Epoch: 9/10      Batch: 3300/3389        Loss: 0.1392060834914446
Epoch: 10/10     Batch: 100/3389         Loss: 0.14133161902427674
Epoch: 10/10     Batch: 200/3389         Loss: 0.14426651373505592
Epoch: 10/10     Batch: 300/3389         Loss: 0.13952943719923497
Epoch: 10/10     Batch: 400/3389         Loss: 0.13387492932379247
Epoch: 10/10     Batch: 500/3389         Loss: 0.14194779358804227
Epoch: 10/10     Batch: 600/3389         Loss: 0.14770587235689164
Epoch: 10/10     Batch: 700/3389         Loss: 0.14216193109750747
Epoch: 10/10     Batch: 800/3389         Loss: 0.13969655111432075
Epoch: 10/10     Batch: 900/3389         Loss: 0.13379767686128616
```

```
Epoch: 10/10      Batch: 1000/3389        Loss: 0.136509146168828
Epoch: 10/10      Batch: 1100/3389        Loss: 0.1401516292244196
Epoch: 10/10      Batch: 1200/3389        Loss: 0.133111234009266
Epoch: 10/10      Batch: 1300/3389        Loss: 0.13362447693943977
Epoch: 10/10      Batch: 1400/3389        Loss: 0.14413965605199336
Epoch: 10/10      Batch: 1500/3389        Loss: 0.13982123024761678
Epoch: 10/10      Batch: 1600/3389        Loss: 0.1453992161899805
Epoch: 10/10      Batch: 1700/3389        Loss: 0.1466843318194151
Epoch: 10/10      Batch: 1800/3389        Loss: 0.13724636249244213
Epoch: 10/10      Batch: 1900/3389        Loss: 0.1411814358085394
Epoch: 10/10      Batch: 2000/3389        Loss: 0.1425471629947424
Epoch: 10/10      Batch: 2100/3389        Loss: 0.14011404514312745
Epoch: 10/10      Batch: 2200/3389        Loss: 0.1345388475805521
Epoch: 10/10      Batch: 2300/3389        Loss: 0.13750953711569308
Epoch: 10/10      Batch: 2400/3389        Loss: 0.13636833392083644
Epoch: 10/10      Batch: 2500/3389        Loss: 0.14050417877733706
Epoch: 10/10      Batch: 2600/3389        Loss: 0.13415048383176326
Epoch: 10/10      Batch: 2700/3389        Loss: 0.1399781584739685
Epoch: 10/10      Batch: 2800/3389        Loss: 0.13742921762168409
Epoch: 10/10      Batch: 2900/3389        Loss: 0.13993247002363204
Epoch: 10/10      Batch: 3000/3389        Loss: 0.13769720904529095
Epoch: 10/10      Batch: 3100/3389        Loss: 0.1376831453293562
Epoch: 10/10      Batch: 3200/3389        Loss: 0.14523460119962692
Epoch: 10/10      Batch: 3300/3389        Loss: 0.13650414310395717
Saving model...
```



In [ ]:

```python
pretrained_embeddings = glove.wi.weight.data
```

In [ ]:

```python
class EncoderRNN_m3(nn.Module):
    def __init__(self, input_size, embedding_size, hidden_size, pretrained_embeddings=None):
        super(EncoderRNN_m3, self).__init__()
        self.hidden_size = hidden_size
        self.embedding_size = embedding_size

        if pretrained_embeddings is not None:
            self.embedding = nn.Embedding.from_pretrained(pretrained_embeddings)
        else:
            self.embedding = nn.Embedding(input_size, embedding_size)

        # Replace GRU with LSTM
        self.lstm = nn.LSTM(embedding_size, hidden_size)

    def forward(self, input, hidden):
        embedded = self.embedding(input).view(1, 1, -1)
        output, hidden = self.lstm(embedded, hidden)
        return output, hidden

    def initHidden(self):
        # Return both hidden state and cell state
        return (torch.zeros(1, 1, self.hidden_size, device=device),
                torch.zeros(1, 1, self.hidden_size, device=device))
```

In [ ]:
```
class AttnDecoderRNN_m3(nn.Module):
    def __init__(self, embedding_size, hidden_size, output_size, dropout_p=0.1, max_length=MAX_LENGTH, pretrained_embeddings=
        super(AttnDecoderRNN_m3, self).__init__()
        self.hidden_size = hidden_size
        self.embedding_size = embedding_size
        self.output_size = output_size
        self.dropout_p = dropout_p
        self.max_length = max_length

        if pretrained_embeddings is not None:
            self.embedding = nn.Embedding.from_pretrained(pretrained_embeddings)
        else:
            self.embedding = nn.Embedding(output_size, embedding_size)

        self.attn = nn.Linear(self.hidden_size + self.embedding_size, max_length)
        self.attn_combine = nn.Linear(self.hidden_size + self.embedding_size, hidden_size)
        self.dropout = nn.Dropout(dropout_p)

        # Replace GRU with LSTM
        self.lstm = nn.LSTM(hidden_size, hidden_size)

        self.out = nn.Linear(hidden_size, output_size)

    def forward(self, input, hidden, encoder_outputs):
        embedded = self.embedding(input).view(1, 1, -1)
        embedded = self.dropout(embedded)

        attn_weights = F.softmax(
            self.attn(torch.cat((embedded[0], hidden[0][0]), 1)), dim=1)
        attn_applied = torch.bmm(attn_weights.unsqueeze(0),
                                 encoder_outputs.unsqueeze(0))

        output = torch.cat((embedded[0], attn_applied[0]), 1)
        output = self.attn_combine(output).unsqueeze(0)
        output = F.relu(output)
        output, hidden = self.lstm(output, hidden)

        output = F.log_softmax(self.out(output[0]), dim=1)
        return output, hidden, attn_weights

    def initHidden(self):
        # Return both hidden state and cell state
        return (torch.zeros(1, 1, self.hidden_size, device=device),
                torch.zeros(1, 1, self.hidden_size, device=device))
```

In [ ]:
```
teacher_forcing_ratio = 1


def train_m3(input_tensor, target_tensor, encoder, decoder, encoder_optimizer, decoder_optimizer, criterion, max_length=MAX_L
    encoder_hidden = encoder.initHidden()

    encoder_optimizer.zero_grad()
    decoder_optimizer.zero_grad()

    input_length = input_tensor.size(0)
    target_length = target_tensor.size(0)

    encoder_outputs = torch.zeros(max_length, encoder.hidden_size, device=device)

    loss = 0

    for ei in range(input_length):
        encoder_output, encoder_hidden = encoder(
            input_tensor[ei], encoder_hidden)
        encoder_outputs[ei] = encoder_output[0, 0]

    decoder_input = torch.tensor([[SOS_token]], device=device)

    decoder_hidden = encoder_hidden

    use_teacher_forcing = True if random.random() < teacher_forcing_ratio else False

    if use_teacher_forcing:
        # Teacher forcing: Feed the target as the next input
        for di in range(target_length):
            decoder_output, decoder_hidden, decoder_attention = decoder(
                decoder_input, decoder_hidden, encoder_outputs)
            loss += criterion(decoder_output, target_tensor[di])
            decoder_input = target_tensor[di]  # Teacher forcing

    else:
        # Without teacher forcing: use its own predictions as the next input
        for di in range(target_length):
            decoder_output, decoder_hidden, decoder_attention = decoder(
                decoder_input, decoder_hidden, encoder_outputs)
            topv, topi = decoder_output.topk(1)
            decoder_input = topi.squeeze().detach()  # detach from history as input

            loss += criterion(decoder_output, target_tensor[di])
            if decoder_input.item() == EOS_token:
                break

    loss.backward()

    encoder_optimizer.step()
    decoder_optimizer.step()

    return loss.item() / target_length
```

In [ ]:
```
def evaluate_m3(input_tensor, target_tensor, encoder, decoder, criterion, max_length=150):
    # Initialize hidden states
    encoder_hidden = encoder.initHidden()
    input_length = input_tensor.size(0)
    target_length = target_tensor.size(0)
    encoder_outputs = torch.zeros(max_length, encoder.hidden_size, device=device)

    # Encoder processing
    for ei in range(min(input_length, max_length)):
        encoder_output, encoder_hidden = encoder(
            input_tensor[ei], encoder_hidden)
        encoder_outputs[ei] = encoder_output[0, 0]

    # Prepare for decoding
    decoder_input = torch.tensor([[SOS_token]], device=device)
    decoder_hidden = encoder_hidden

    loss = 0

    # Decoder processing
    with torch.no_grad():
        for di in range(target_length):
            decoder_output, decoder_hidden, decoder_attention = decoder(
                decoder_input, decoder_hidden, encoder_outputs)
            topv, topi = decoder_output.topk(1)
            decoder_input = topi.squeeze().detach()

            loss += criterion(decoder_output, target_tensor[di])
            if decoder_input.item() == EOS_token:
                break

    return loss.item() / target_length
```

In [ ]:
```
def validate_model_m3(encoder, decoder, val_pairs, criterion, max_length=MAX_LENGTH, num_samples=300):
    # Randomly select a subset of validation pairs
    sampled_pairs = random.sample(val_pairs, num_samples) if len(val_pairs) > num_samples else val_pairs

    total_loss = 0
    with torch.no_grad():
        for input_tensor, target_tensor in sampled_pairs:
            loss = evaluate_m3(input_tensor, target_tensor, encoder, decoder, criterion, max_length)
            total_loss += loss

    average_loss = total_loss / len(sampled_pairs)
    return average_loss
```

In [ ]:
```
def trainIters_m3(encoder, decoder, n_iters, print_every=1000, plot_every=100, learning_rate=0.01):
    start = time.time()
    plot_losses = []
    val_losses = []
    print_loss_total = 0  # Reset every print_every
    plot_loss_total = 0  # Reset every plot_every
    encoder_optimizer = optim.SGD(encoder.parameters(), lr=learning_rate)
    decoder_optimizer = optim.SGD(decoder.parameters(), lr=learning_rate)
    training_pairs = [tensorsFromPair(random.choice(train_pairs))
                      for i in range(n_iters)]
    validation_pairs = [tensorsFromPair(random.choice(val_pairs))
                      for i in range(n_iters)]
    criterion = nn.NLLLoss()

    for iter in range(1, n_iters + 1):
        training_pair = training_pairs[iter - 1]
        input_tensor = training_pair[0]
        target_tensor = training_pair[1]

        loss = train_m3(input_tensor, target_tensor, encoder,
                     decoder, encoder_optimizer, decoder_optimizer, criterion)
        print_loss_total += loss
        plot_loss_total += loss

        if iter % print_every == 0:
            print_loss_avg = print_loss_total / print_every
            print_loss_total = 0
            print('%s (%d %d%%) %.4f' % (time_since(start, iter / n_iters),
                                         iter, iter / n_iters * 100, print_loss_avg))

        if iter % plot_every == 0:
            plot_loss_avg = plot_loss_total / plot_every
            plot_losses.append(plot_loss_avg)
            plot_loss_total = 0

            val_loss = validate_model_m3(encoder, decoder, validation_pairs, criterion)
            val_losses.append(val_loss)
            print(f'Validation Loss: {val_loss:.4f} at iteration {iter}')

    # showPlot(plot_losses)
    return plot_losses, val_losses
```

In [ ]:
```
hidden_size = 256
encoder_m3 = EncoderRNN_m3(input_lang.n_words, EMBED_DIM, hidden_size, pretrained_embeddings=pretrained_embeddings).to(device
decoder_m3 = AttnDecoderRNN_m3(EMBED_DIM, hidden_size, output_lang.n_words, pretrained_embeddings=pretrained_embeddings).to(d
train_losses_m3, val_losses_m3 = trainIters_m3(encoder_m3, decoder_m3, 40000, print_every=500, plot_every=500)
```

```
0m 51s (- 67m 53s) (500 1%) 6.8338
Validation Loss: 7.0869 at iteration 500
1m 52s (- 72m 58s) (1000 2%) 5.9809
Validation Loss: 5.9415 at iteration 1000
2m 51s (- 73m 29s) (1500 3%) 5.6504
Validation Loss: 3.5177 at iteration 1500
3m 43s (- 70m 48s) (2000 5%) 5.3893
Validation Loss: 7.0276 at iteration 2000
4m 44s (- 71m 12s) (2500 6%) 5.2305
Validation Loss: 5.4754 at iteration 2500
5m 40s (- 70m 3s) (3000 7%) 4.9774
Validation Loss: 7.5884 at iteration 3000
6m 42s (- 69m 52s) (3500 8%) 4.8530
Validation Loss: 8.1912 at iteration 3500
7m 44s (- 69m 42s) (4000 10%) 4.7112
Validation Loss: 7.5652 at iteration 4000
8m 47s (- 69m 18s) (4500 11%) 4.6644
Validation Loss: 8.1024 at iteration 4500
9m 47s (- 68m 33s) (5000 12%) 4.5722
Validation Loss: 7.0250 at iteration 5000
10m 45s (- 67m 28s) (5500 13%) 4.5093
Validation Loss: 8.2394 at iteration 5500
11m 51s (- 67m 9s) (6000 15%) 4.4428
Validation Loss: 6.6786 at iteration 6000
12m 48s (- 66m 2s) (6500 16%) 4.3600
Validation Loss: 4.1338 at iteration 6500
13m 44s (- 64m 45s) (7000 17%) 4.2763
Validation Loss: 5.5175 at iteration 7000
14m 38s (- 63m 26s) (7500 18%) 4.2089
Validation Loss: 5.5589 at iteration 7500
15m 31s (- 62m 4s) (8000 20%) 4.2661
Validation Loss: 3.4343 at iteration 8000
16m 23s (- 60m 43s) (8500 21%) 4.1480
Validation Loss: 4.3775 at iteration 8500
17m 13s (- 59m 19s) (9000 22%) 4.2042
Validation Loss: 5.7774 at iteration 9000
18m 7s (- 58m 12s) (9500 23%) 4.1036
Validation Loss: 7.5474 at iteration 9500
19m 4s (- 57m 14s) (10000 25%) 4.1220
Validation Loss: 8.4220 at iteration 10000
20m 4s (- 56m 23s) (10500 26%) 4.0211
Validation Loss: 7.6242 at iteration 10500
21m 1s (- 55m 24s) (11000 27%) 4.0371
Validation Loss: 7.1003 at iteration 11000
21m 59s (- 54m 30s) (11500 28%) 3.9150
Validation Loss: 6.8491 at iteration 11500
22m 53s (- 53m 23s) (12000 30%) 3.9760
Validation Loss: 6.2431 at iteration 12000
23m 51s (- 52m 29s) (12500 31%) 3.9923
Validation Loss: 7.2537 at iteration 12500
24m 47s (- 51m 28s) (13000 32%) 3.9209
Validation Loss: 5.1867 at iteration 13000
25m 40s (- 50m 23s) (13500 33%) 3.9066
Validation Loss: 4.6665 at iteration 13500
26m 33s (- 49m 18s) (14000 35%) 3.9649
Validation Loss: 6.2562 at iteration 14000
27m 30s (- 48m 21s) (14500 36%) 3.8863
Validation Loss: 7.3843 at iteration 14500
28m 28s (- 47m 27s) (15000 37%) 3.8840
Validation Loss: 5.7838 at iteration 15000
29m 20s (- 46m 22s) (15500 38%) 3.8790
Validation Loss: 8.1210 at iteration 15500
30m 18s (- 45m 28s) (16000 40%) 3.8027
Validation Loss: 8.2221 at iteration 16000
31m 16s (- 44m 32s) (16500 41%) 3.8584
Validation Loss: 7.0178 at iteration 16500
32m 12s (- 43m 34s) (17000 42%) 3.8490
Validation Loss: 8.0814 at iteration 17000
33m 11s (- 42m 40s) (17500 43%) 3.8139
Validation Loss: 8.4393 at iteration 17500
34m 11s (- 41m 47s) (18000 45%) 3.7440
Validation Loss: 6.6245 at iteration 18000
35m 6s (- 40m 48s) (18500 46%) 3.8704
Validation Loss: 7.6932 at iteration 18500
36m 4s (- 39m 52s) (19000 47%) 3.8272
Validation Loss: 6.5816 at iteration 19000
36m 59s (- 38m 53s) (19500 48%) 3.7743
```

```
Validation Loss: 3.3315 at iteration 19500
37m 50s (- 37m 50s) (20000 50%) 3.7548
Validation Loss: 6.5113 at iteration 20000
38m 42s (- 36m 49s) (20500 51%) 3.7512
Validation Loss: 6.0085 at iteration 20500
39m 38s (- 35m 52s) (21000 52%) 3.6718
Validation Loss: 5.9697 at iteration 21000
40m 32s (- 34m 52s) (21500 53%) 3.6595
Validation Loss: 6.4411 at iteration 21500
41m 25s (- 33m 53s) (22000 55%) 3.7298
Validation Loss: 5.5496 at iteration 22000
42m 21s (- 32m 56s) (22500 56%) 3.7187
Validation Loss: 6.3122 at iteration 22500
43m 16s (- 31m 59s) (23000 57%) 3.6433
Validation Loss: 8.0893 at iteration 23000
44m 17s (- 31m 5s) (23500 58%) 3.6085
Validation Loss: 5.4103 at iteration 23500
45m 10s (- 30m 6s) (24000 60%) 3.6901
Validation Loss: 5.0248 at iteration 24000
46m 6s (- 29m 10s) (24500 61%) 3.6430
Validation Loss: 6.9707 at iteration 24500
47m 0s (- 28m 12s) (25000 62%) 3.6736
Validation Loss: 7.1808 at iteration 25000
48m 1s (- 27m 18s) (25500 63%) 3.6811
Validation Loss: 7.7729 at iteration 25500
48m 59s (- 26m 22s) (26000 65%) 3.6526
Validation Loss: 8.3038 at iteration 26000
49m 58s (- 25m 27s) (26500 66%) 3.6332
Validation Loss: 4.4013 at iteration 26500
50m 52s (- 24m 29s) (27000 67%) 3.6988
Validation Loss: 5.7896 at iteration 27000
51m 48s (- 23m 32s) (27500 68%) 3.6336
Validation Loss: 6.3974 at iteration 27500
52m 43s (- 22m 35s) (28000 70%) 3.5980
Validation Loss: 6.9159 at iteration 28000
53m 38s (- 21m 38s) (28500 71%) 3.7153
Validation Loss: 4.5469 at iteration 28500
54m 35s (- 20m 42s) (29000 72%) 3.6658
Validation Loss: 4.6024 at iteration 29000
55m 31s (- 19m 45s) (29500 73%) 3.5899
Validation Loss: 6.4438 at iteration 29500
56m 28s (- 18m 49s) (30000 75%) 3.5683
Validation Loss: 3.1772 at iteration 30000
57m 19s (- 17m 51s) (30500 76%) 3.5720
Validation Loss: 4.7285 at iteration 30500
58m 14s (- 16m 54s) (31000 77%) 3.6386
Validation Loss: 7.6278 at iteration 31000
59m 9s (- 15m 57s) (31500 78%) 3.6019
Validation Loss: 4.9729 at iteration 31500
60m 2s (- 15m 0s) (32000 80%) 3.6384
Validation Loss: 3.8696 at iteration 32000
60m 54s (- 14m 3s) (32500 81%) 3.5930
Validation Loss: 5.1198 at iteration 32500
61m 49s (- 13m 6s) (33000 82%) 3.6543
Validation Loss: 6.6876 at iteration 33000
62m 44s (- 12m 10s) (33500 83%) 3.5297
Validation Loss: 7.6035 at iteration 33500
63m 44s (- 11m 14s) (34000 85%) 3.6349
Validation Loss: 4.9975 at iteration 34000
64m 36s (- 10m 18s) (34500 86%) 3.6049
Validation Loss: 6.8932 at iteration 34500
65m 35s (- 9m 22s) (35000 87%) 3.5281
Validation Loss: 6.8804 at iteration 35000
66m 32s (- 8m 26s) (35500 88%) 3.6100
Validation Loss: 9.1252 at iteration 35500
67m 30s (- 7m 30s) (36000 90%) 3.5372
Validation Loss: 7.0960 at iteration 36000
68m 27s (- 6m 33s) (36500 91%) 3.4734
Validation Loss: 7.2615 at iteration 36500
69m 23s (- 5m 37s) (37000 92%) 3.5056
Validation Loss: 7.0008 at iteration 37000
70m 21s (- 4m 41s) (37500 93%) 3.5770
Validation Loss: 3.1584 at iteration 37500
71m 10s (- 3m 44s) (38000 95%) 3.5486
Validation Loss: 7.6434 at iteration 38000
72m 9s (- 2m 48s) (38500 96%) 3.5134
Validation Loss: 4.5911 at iteration 38500
```

```
73m 2s (- 1m 52s) (39000 97%) 3.5541
Validation Loss: 7.2139 at iteration 39000
73m 58s (- 0m 56s) (39500 98%) 3.4586
Validation Loss: 5.7428 at iteration 39500
74m 51s (- 0m 0s) (40000 100%) 3.4840
Validation Loss: 8.3670 at iteration 40000
```

Extend Model 2: Plug-and-Play Recipe Generation with Content Planning

In [ ]:
```python
import torch
import torch.nn as nn

class EncoderRNN_m4(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers=2, dropout=0.1):
        # call the constructor of the parent class 'nn.model'
        super(EncoderRNN_m4, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        # initialize embedding layer
        # this layer used to convert input into dense vectors of hidden size
        self.embedding = nn.Embedding(input_size, hidden_size)
        # initialize th LSTM
        # num_layers: number of lstm layer in the network
        # droupour: input and recurrent connections within each lstm layer
        self.lstm = nn.LSTM(hidden_size, hidden_size, num_layers=num_layers, dropout=dropout)

    def forward(self, input, hidden):
        # embeds the input tensor using the embedding layer
        embedded = self.embedding(input).view(1, 1, -1)
        output = embedded
        # pass embedded input and initial hidden stage through lstm layer
        output, (hidden, cell) = self.lstm(output, hidden)
        # return the output tensor and final hidden cell states from lstm layer
        return output, (hidden, cell)

    def initHidden(self):
        return (torch.zeros(self.num_layers, 1, self.hidden_size, device=device),
                torch.zeros(self.num_layers, 1, self.hidden_size, device=device))
```

In [ ]:
```python
import torch.nn.functional as F
import torch

class AttnDecoderRNN_m4(nn.Module):
    def __init__(self, hidden_size, output_size, stage_size, num_layers=2, dropout_p=0.1, max_length=150):
        super(AttnDecoderRNN_m4, self).__init__()
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.dropout_p = dropout_p
        self.max_length = max_length

        self.embedding = nn.Embedding(self.output_size, self.hidden_size)
        # stage
        self.stage_embedding = nn.Embedding(stage_size, self.hidden_size)
        self.dropout = nn.Dropout(self.dropout_p)
        self.lstm = nn.LSTM(self.hidden_size, self.hidden_size, num_layers=num_layers, dropout=dropout_p)
        self.out = nn.Linear(self.hidden_size * 2, self.output_size)

    def forward(self, input, hidden, cell, encoder_outputs, current_stage):
        embedded = self.embedding(input).view(1, 1, -1)
        # stage
        stage_embedded = self.stage_embedding(current_stage).view(1, 1, -1)
        combined_embedded = torch.cat((embedded, stage_embedded), 2)
        combined_embedded = self.dropout(combined_embedded)
        # embedded = self.dropout(embedded)

        lstm_output, (hidden, cell) = self.lstm(embedded, (hidden, cell))

        attn_weights = F.softmax(torch.bmm(lstm_output, encoder_outputs.unsqueeze(0).permute(0, 2, 1)), dim=-1)
        attn_output = torch.bmm(attn_weights, encoder_outputs.unsqueeze(0))

        concat_output = torch.cat((attn_output[0], lstm_output[0]), 1)
        output = F.log_softmax(self.out(concat_output), dim=1)
        return output, hidden, cell, attn_weights

    def initHidden(self):
        return (torch.zeros(self.num_layers, 1, self.hidden_size, device=device),
                torch.zeros(self.num_layers, 1, self.hidden_size, device=device))
```

In [ ]:
```
def criteria_for_next_stage(token):
    return token in ['.', ';']
```

In [ ]:
```
from transformers import BartForConditionalGeneration, BartTokenizer, GPT2LMHeadModel, GPT2Tokenizer

# Load the content planner model and tokenizer
planner = BartForConditionalGeneration.from_pretrained('yinhongliu/recipe_with_plan_bart_planner').to(device)
planner_tokenizer = BartTokenizer.from_pretrained('yinhongliu/recipe_with_plan_bart_planner')
```

```
c:\Users\mirik\anaconda3\envs\NLP\Lib\site-packages\transformers\adapters\__init__.py:27: FutureWarning: The `adapter-transfor
mers` package is deprecated and replaced by the `adapters` package. See https://docs.adapterhub.ml/transitioning.html.
  warnings.warn(
```

In [ ]:
```
import torch
import random

teacher_forcing_ratio = 1

def train_m4(input_tensor, target_tensor, encoder, decoder, encoder_optimizer, decoder_optimizer, criterion, planner, planner
    # Initialize the hidden state and cell state of the encoder
    encoder_hidden, encoder_cell = encoder.initHidden()

    # Zero gradients for encoder and decoder
    encoder_optimizer.zero_grad()
    decoder_optimizer.zero_grad()

    # Determine the length of the input and target sequence
    input_length = input_tensor.size(0)
    target_length = target_tensor.size(0)

    # Initialize a tensor to hold encoder output
    encoder_outputs = torch.zeros(max_length, encoder.hidden_size, device=device)

    # Generate the content plan
    ingredients_text = " ".join([input_lang.index2word[token.item()] for token in input_tensor])
    inputs = planner_tokenizer(ingredients_text, return_tensors='pt', max_length=512, truncation=True).to(device)
    content_plan_output = planner.generate(**inputs)
    content_plan = planner_tokenizer.decode(content_plan_output[0], skip_special_tokens=True).split()

    # Debugging line to check the content plan output
    # print("Content Plan:", content_plan)

    content_plan_indices = [stage_vocab.get(stage, stage_vocab['<unk>']) for stage in content_plan]
    content_plan_tensor = torch.tensor(content_plan_indices, device=device)  # Convert to tensor

    loss = 0

    # Iterate over input sequence
    for ei in range(input_length):
        encoder_output, (encoder_hidden, encoder_cell) = encoder(
            input_tensor[ei], (encoder_hidden, encoder_cell))
        encoder_outputs[ei] = encoder_output[0, 0]

    # Initialize the decoder input with the start-of-sequence token
    decoder_input = torch.tensor([[SOS_token]], device=device)

    # Initialize hidden and cell for decoder to the final hidden and cells states of the encoder
    decoder_hidden = encoder_hidden
    decoder_cell = encoder_cell

    current_stage_index = 0
    current_stage = content_plan_tensor[current_stage_index].unsqueeze(0).unsqueeze(0)  # Correctly shaped tensor

    use_teacher_forcing = True if random.random() < teacher_forcing_ratio else False

    if use_teacher_forcing:
        # Teacher forcing: Feed the target as the next input
        for di in range(target_length):
            decoder_output, decoder_hidden, decoder_cell, decoder_attention = decoder(
                decoder_input, decoder_hidden, decoder_cell, encoder_outputs, current_stage)
            loss += criterion(decoder_output, target_tensor[di])
            decoder_input = target_tensor[di]  # Teacher forcing

            if criteria_for_next_stage(decoder_input):  # Define this function
                current_stage_index += 1
                if current_stage_index < len(content_plan_tensor):
                    current_stage = content_plan_tensor[current_stage_index].unsqueeze(0).unsqueeze(0)

    else:
        # Without teacher forcing: use its own predictions as the next input
        for di in range(target_length):
            decoder_output, decoder_hidden, decoder_cell, decoder_attention = decoder(
                decoder_input, decoder_hidden, decoder_cell, encoder_outputs, current_stage)
            topv, topi = decoder_output.topk(1)
            decoder_input = topi.squeeze().detach()  # detach from history as input

            loss += criterion(decoder_output, target_tensor[di])
            if decoder_input.item() == EOS_token:
                break

            if criteria_for_next_stage(decoder_input):  # Define this function
                current_stage_index += 1
                if current_stage_index < len(content_plan_tensor):
                    current_stage = content_plan_tensor[current_stage_index].unsqueeze(0).unsqueeze(0)

    # Backpropagate the loss
    loss.backward()

    # Clip the gradients to prevent exploding gradients
    torch.nn.utils.clip_grad_norm_(encoder.parameters(), 1)
    torch.nn.utils.clip_grad_norm_(decoder.parameters(), 1)

    # Update the parameters of the encoder and decoder using their respective optimizers
    encoder_optimizer.step()
    decoder_optimizer.step()

    return loss.item() / target_length
```

In [ ]:

```
def evaluate_m4(input_tensor, target_tensor, encoder, decoder, criterion, max_length=150):
    encoder_hidden, encoder_cell = encoder.initHidden()
    input_length = input_tensor.size(0)
    target_length = target_tensor.size(0)
    encoder_outputs = torch.zeros(max_length, encoder.hidden_size, device=device)

    ingredients_text = " ".join([input_lang.index2word[token.item()] for token in input_tensor])
    inputs = planner_tokenizer(ingredients_text, return_tensors='pt', max_length=512, truncation=True).to(device)
    content_plan_output = planner.generate(**inputs)
    content_plan = planner_tokenizer.decode(content_plan_output[0], skip_special_tokens=True).split()

    content_plan_indices = [stage_vocab.get(stage, stage_vocab['<unk>']) for stage in content_plan]
    content_plan_tensor = torch.tensor(content_plan_indices, device=device)  # Convert to tensor

    # Encoder
    for ei in range(input_length):
        # print(input_tensor[ei], encoder_hidden.size())
        # ==
        encoder_output, (encoder_hidden, encoder_cell) = encoder(
            input_tensor[ei], (encoder_hidden, encoder_cell))
        # ==
        if ei < max_length:
            encoder_outputs[ei] = encoder_output[0, 0]

    current_stage_index = 0
    current_stage = content_plan_tensor[current_stage_index].unsqueeze(0).unsqueeze(0)  # Correctly shaped tensor

    loss = 0

    # Decoder
    decoder_input = torch.tensor([[SOS_token]], device=device)
    decoder_hidden = encoder_hidden
    decoder_cell = encoder_cell

    for di in range(target_length):
        # ====
        decoder_output, decoder_hidden, decoder_cell, decoder_attention = decoder(
                decoder_input, decoder_hidden, decoder_cell, encoder_outputs, current_stage)
        # ====
        topv, topi = decoder_output.topk(1)
        decoder_input = topi.squeeze().detach()  # Use its own predictions as the next input
        loss += criterion(decoder_output, target_tensor[di])
        if decoder_input.item() == EOS_token:
            break

    return loss.item() / target_length
```

In [ ]:
```
def validate_model_m4(encoder, decoder, val_pairs, criterion, max_length=MAX_LENGTH, num_samples=300):
    # Randomly select a subset of validation pairs
    if len(val_pairs) > num_samples:
        sampled_pairs = random.sample(val_pairs, num_samples)
    else:
        sampled_pairs = val_pairs

    total_loss = 0
    with torch.no_grad():
        for input_tensor, target_tensor in sampled_pairs:
            loss = evaluate_m4(input_tensor, target_tensor, encoder, decoder, criterion, max_length)
            total_loss += loss

    average_loss = total_loss / len(sampled_pairs)
    return average_loss
```

In [ ]:
```
from transformers import BartForConditionalGeneration, BartTokenizer, GPT2LMHeadModel, GPT2Tokenizer

def trainIters_m4(encoder, decoder, n_iters, print_every=1000, plot_every=100, learning_rate=0.01):
    start = time.time()
    plot_losses = []
    val_losses = []
    print_loss_total = 0  # Reset every print_every
    plot_loss_total = 0  # Reset every plot_every

    # initialize adam optimizers for encoder and decoder
    encoder_optimizer = optim.Adam(encoder.parameters(), lr=learning_rate)
    decoder_optimizer = optim.Adam(decoder.parameters(), lr=learning_rate)

    # # Load the content planner model and tokenizer
    # planner = BartForConditionalGeneration.from_pretrained('yinhongliu/recipe_with_plan_bart_planner').to(device)
    # planner_tokenizer = BartTokenizer.from_pretrained('yinhongliu/recipe_with_plan_bart_planner')

    # create training pairs by randomly selecting pairs from dataset and convert them into tensors
    training_pairs = [tensorsFromPair(random.choice(train_pairs)) for _ in range(n_iters)]
    validation_pairs = [tensorsFromPair(random.choice(val_pairs))
                  for i in range(n_iters)]

    # Negative log likelihood loss criterion for training
    criterion = nn.NLLLoss()

    for iter in range(1, n_iters + 1):
        training_pair = training_pairs[iter - 1]
        input_tensor = training_pair[0]
        target_tensor = training_pair[1]

        input_tensor = input_tensor.to(device)
        target_tensor = target_tensor.to(device)

        # call train_attn to compute loss for current iteration
        # loss = train_attn(input_tensor, target_tensor, encoder, decoder, encoder_optimizer, decoder_optimizer, criterion)
        loss = train_m4(input_tensor, target_tensor, encoder, decoder, encoder_optimizer, decoder_optimizer, criterion, plann
        print_loss_total += loss
        plot_loss_total += loss

        if iter % print_every == 0:
            print_loss_avg = print_loss_total / print_every
            print_loss_total = 0
            print('%s (%d %d%%) %.4f' % (time_since(start, iter / n_iters),
                                         iter, iter / n_iters * 100, print_loss_avg))
        if iter % plot_every == 0:
            plot_loss_avg = plot_loss_total / plot_every
            plot_losses.append(plot_loss_avg)
            plot_loss_total = 0

            val_loss = validate_model_m4(encoder, decoder, validation_pairs, criterion)
            val_losses.append(val_loss)
            print(f'Validation Loss: {val_loss:.4f} at iteration {iter}')
    # showPlot(plot_losses)
    return plot_losses, val_losses
```

In [ ]:
```
hidden_size = 256

stage_vocab = {
    'pre-processing': 0,
    'mixing': 1,
    'cooking': 2,
    'post-processing': 3,
    'final': 4,
    '<unk>': 5  # Unknown token to handle any unexpected stages
}
stage_size = len(stage_vocab)

# input_lang.n_words: size of the vocabulary of the input language
encoder_m4 = EncoderRNN_m4(input_lang.n_words, hidden_size).to(device)
# dropout_p: dropoup probability, which help in regularization by randomly dropping connections during training
decoder_m4 = AttnDecoderRNN_m4(hidden_size, output_lang.n_words, stage_size=len(stage_vocab), dropout_p=0.1).to(device)
train_losses_m4, val_losses_m4 = trainIters_m4(encoder_m4, decoder_m4, 40000, print_every=500, plot_every=500)
```

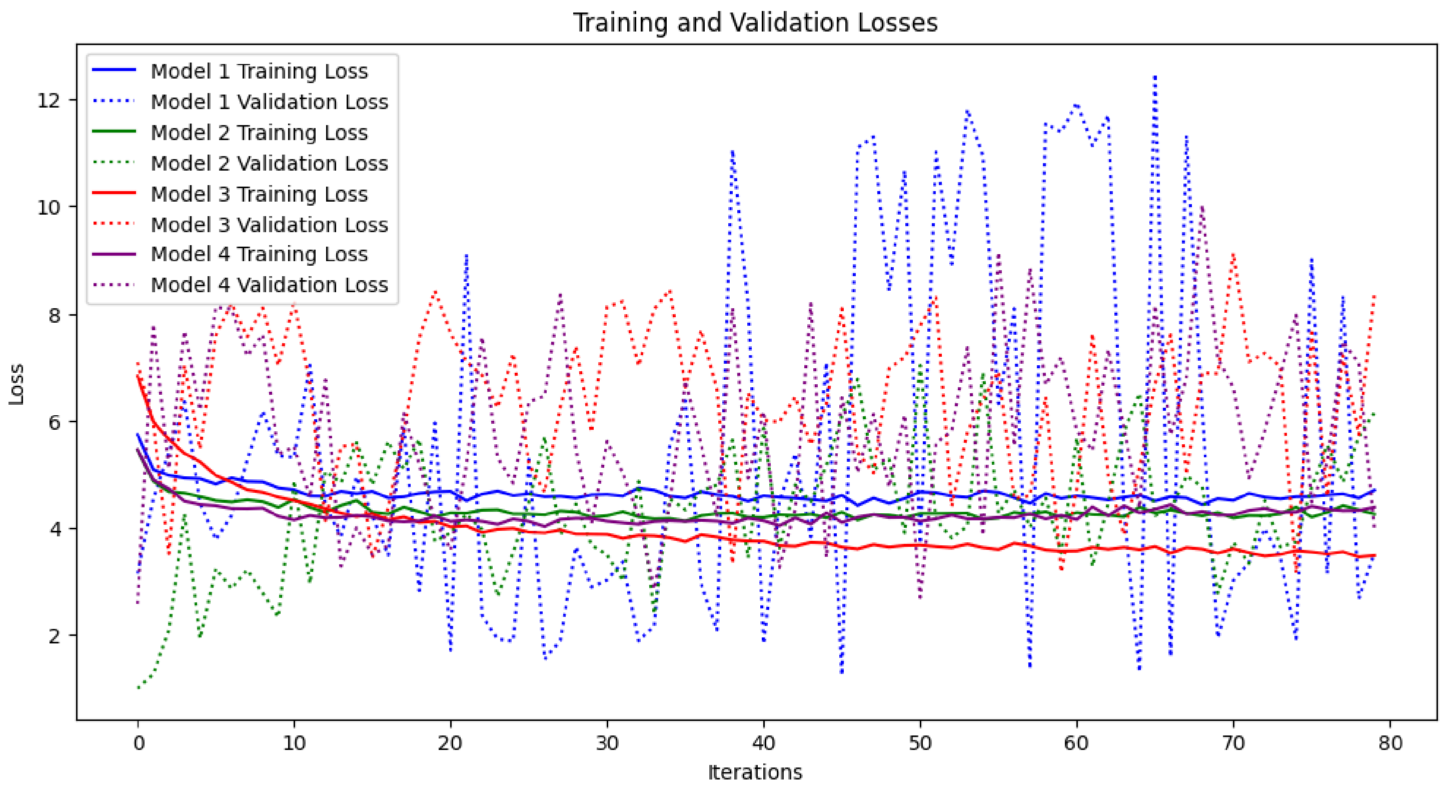
```
c:\Users\mirik\anaconda3\envs\NLP\Lib\site-packages\transformers\generation\utils.py:1288: UserWarning: Neither `max_length` n
or `max_new_tokens` has been set, `max_length` will default to 20 (`generation_config.max_length`). Controlling `max_length` v
ia the config is deprecated and `max_length` will be removed from the config in v5 of Transformers -- we recommend using `max_
new_tokens` to control the maximum length of the generation.
  warnings.warn(
1m 46s (- 140m 34s) (500 1%) 5.4597
Validation Loss: 2.5831 at iteration 500
3m 52s (- 151m 16s) (1000 2%) 4.8968
Validation Loss: 7.7976 at iteration 1000
6m 7s (- 157m 11s) (1500 3%) 4.7299
Validation Loss: 5.1080 at iteration 1500
8m 18s (- 157m 58s) (2000 5%) 4.4976
Validation Loss: 7.6728 at iteration 2000
10m 36s (- 159m 9s) (2500 6%) 4.4361
Validation Loss: 6.1969 at iteration 2500
12m 48s (- 157m 59s) (3000 7%) 4.4136
Validation Loss: 8.1146 at iteration 3000
15m 4s (- 157m 9s) (3500 8%) 4.3612
Validation Loss: 8.0804 at iteration 3500
17m 17s (- 155m 40s) (4000 10%) 4.3583
Validation Loss: 7.2097 at iteration 4000
19m 31s (- 154m 3s) (4500 11%) 4.3685
Validation Loss: 7.6046 at iteration 4500
21m 45s (- 152m 21s) (5000 12%) 4.2155
Validation Loss: 5.3755 at iteration 5000
23m 56s (- 150m 13s) (5500 13%) 4.1474
Validation Loss: 5.5150 at iteration 5500
26m 6s (- 147m 54s) (6000 15%) 4.2309
Validation Loss: 4.5708 at iteration 6000
28m 15s (- 145m 36s) (6500 16%) 4.1867
Validation Loss: 6.7533 at iteration 6500
30m 26s (- 143m 31s) (7000 17%) 4.1962
Validation Loss: 3.2869 at iteration 7000
32m 31s (- 140m 56s) (7500 18%) 4.2341
Validation Loss: 4.0071 at iteration 7500
34m 43s (- 138m 55s) (8000 20%) 4.2033
Validation Loss: 3.4966 at iteration 8000
36m 51s (- 136m 37s) (8500 21%) 4.1300
Validation Loss: 3.6743 at iteration 8500
38m 57s (- 134m 10s) (9000 22%) 4.1145
Validation Loss: 6.1385 at iteration 9000
41m 9s (- 132m 7s) (9500 23%) 4.1288
Validation Loss: 5.1825 at iteration 9500
43m 19s (- 129m 58s) (10000 25%) 4.2149
Validation Loss: 4.1547 at iteration 10000
45m 27s (- 127m 43s) (10500 26%) 4.1213
Validation Loss: 3.5897 at iteration 10500
47m 34s (- 125m 24s) (11000 27%) 4.1486
Validation Loss: 5.1719 at iteration 11000
49m 45s (- 123m 19s) (11500 28%) 4.1219
Validation Loss: 7.5577 at iteration 11500
51m 59s (- 121m 18s) (12000 30%) 4.0682
Validation Loss: 5.3036 at iteration 12000
54m 10s (- 119m 11s) (12500 31%) 4.1647
Validation Loss: 4.8335 at iteration 12500
56m 19s (- 116m 59s) (13000 32%) 4.1217
Validation Loss: 6.3470 at iteration 13000
58m 32s (- 114m 55s) (13500 33%) 4.0268
Validation Loss: 6.4637 at iteration 13500
60m 41s (- 112m 42s) (14000 35%) 4.1590
Validation Loss: 8.3478 at iteration 14000
62m 55s (- 110m 39s) (14500 36%) 4.1788
Validation Loss: 5.4530 at iteration 14500
65m 6s (- 108m 30s) (15000 37%) 4.1757
Validation Loss: 4.0774 at iteration 15000
67m 13s (- 106m 15s) (15500 38%) 4.1248
Validation Loss: 5.6138 at iteration 15500
69m 23s (- 104m 5s) (16000 40%) 4.0944
Validation Loss: 5.0138 at iteration 16000
71m 33s (- 101m 55s) (16500 41%) 4.0689
Validation Loss: 4.2322 at iteration 16500
73m 42s (- 99m 43s) (17000 42%) 4.1198
Validation Loss: 2.8569 at iteration 17000
75m 46s (- 97m 25s) (17500 43%) 4.1365
Validation Loss: 4.4701 at iteration 17500
77m 55s (- 95m 13s) (18000 45%) 4.1217
Validation Loss: 6.7389 at iteration 18000
80m 7s (- 93m 6s) (18500 46%) 4.1415
Validation Loss: 5.5596 at iteration 18500
82m 17s (- 90m 57s) (19000 47%) 4.1260
Validation Loss: 4.1999 at iteration 19000
84m 25s (- 88m 45s) (19500 48%) 4.0870
```

```
Validation Loss: 8.1042 at iteration 19500
86m 39s (- 86m 39s) (20000 50%) 4.1812
Validation Loss: 4.9249 at iteration 20000
88m 48s (- 84m 28s) (20500 51%) 4.1331
Validation Loss: 6.1700 at iteration 20500
90m 57s (- 82m 17s) (21000 52%) 4.0403
Validation Loss: 3.2584 at iteration 21000
93m 3s (- 80m 4s) (21500 53%) 4.1852
Validation Loss: 4.3434 at iteration 21500
95m 8s (- 77m 50s) (22000 55%) 4.0715
Validation Loss: 8.1802 at iteration 22000
97m 19s (- 75m 41s) (22500 56%) 4.2823
Validation Loss: 3.4505 at iteration 22500
99m 28s (- 73m 31s) (23000 57%) 4.1024
Validation Loss: 6.4310 at iteration 23000
101m 39s (- 71m 22s) (23500 58%) 4.1983
Validation Loss: 5.0301 at iteration 23500
103m 47s (- 69m 11s) (24000 60%) 4.2509
Validation Loss: 6.1223 at iteration 24000
105m 56s (- 67m 1s) (24500 61%) 4.1923
Validation Loss: 4.7652 at iteration 24500
108m 2s (- 64m 49s) (25000 62%) 4.1954
Validation Loss: 6.1042 at iteration 25000
110m 8s (- 62m 37s) (25500 63%) 4.1246
Validation Loss: 2.7028 at iteration 25500
112m 13s (- 60m 25s) (26000 65%) 4.1682
Validation Loss: 5.5737 at iteration 26000
114m 25s (- 58m 17s) (26500 66%) 4.2386
Validation Loss: 5.8151 at iteration 26500
116m 34s (- 56m 7s) (27000 67%) 4.1688
Validation Loss: 7.3605 at iteration 27000
118m 46s (- 53m 59s) (27500 68%) 4.1691
Validation Loss: 3.8622 at iteration 27500
120m 52s (- 51m 48s) (28000 70%) 4.1950
Validation Loss: 9.1417 at iteration 28000
123m 2s (- 49m 39s) (28500 71%) 4.1885
Validation Loss: 5.5298 at iteration 28500
125m 16s (- 47m 30s) (29000 72%) 4.2641
Validation Loss: 8.8496 at iteration 29000
127m 31s (- 45m 23s) (29500 73%) 4.1675
Validation Loss: 6.6669 at iteration 29500
129m 42s (- 43m 14s) (30000 75%) 4.2355
Validation Loss: 7.1816 at iteration 30000
131m 53s (- 41m 4s) (30500 76%) 4.1558
Validation Loss: 5.8115 at iteration 30500
134m 0s (- 38m 54s) (31000 77%) 4.3932
Validation Loss: 5.4428 at iteration 31000
136m 7s (- 36m 43s) (31500 78%) 4.2256
Validation Loss: 7.3297 at iteration 31500
138m 15s (- 34m 33s) (32000 80%) 4.4128
Validation Loss: 5.8621 at iteration 32000
140m 21s (- 32m 23s) (32500 81%) 4.2824
Validation Loss: 4.3738 at iteration 32500
142m 25s (- 30m 12s) (33000 82%) 4.3486
Validation Loss: 8.1474 at iteration 33000
144m 35s (- 28m 3s) (33500 83%) 4.4344
Validation Loss: 5.7367 at iteration 33500
146m 43s (- 25m 53s) (34000 85%) 4.2551
Validation Loss: 6.8148 at iteration 34000
148m 51s (- 23m 43s) (34500 86%) 4.3060
Validation Loss: 10.0178 at iteration 34500
151m 3s (- 21m 34s) (35000 87%) 4.2373
Validation Loss: 7.1524 at iteration 35000
153m 12s (- 19m 25s) (35500 88%) 4.2433
Validation Loss: 6.6170 at iteration 35500
155m 20s (- 17m 15s) (36000 90%) 4.3258
Validation Loss: 4.9282 at iteration 36000
157m 24s (- 15m 5s) (36500 91%) 4.3630
Validation Loss: 5.8104 at iteration 36500
159m 31s (- 12m 56s) (37000 92%) 4.2897
Validation Loss: 6.8600 at iteration 37000
161m 41s (- 10m 46s) (37500 93%) 4.2957
Validation Loss: 7.9708 at iteration 37500
163m 50s (- 8m 37s) (38000 95%) 4.4032
Validation Loss: 4.4801 at iteration 38000
165m 55s (- 6m 27s) (38500 96%) 4.3460
Validation Loss: 5.2145 at iteration 38500
```

```
168m 1s (- 4m 18s) (39000 97%) 4.3209
Validation Loss: 7.4748 at iteration 39000
170m 8s (- 2m 9s) (39500 98%) 4.3252
Validation Loss: 7.0693 at iteration 39500
172m 17s (- 0m 0s) (40000 100%) 4.3860
Validation Loss: 3.9853 at iteration 40000
```

Plot

In [ ]:
```python
import matplotlib.pyplot as plt

def showPlot(all_train_losses, all_val_losses, colors, model_labels, title=None):
    plt.figure(figsize=(12, 6))

    # Check if the correct amount of data is provided
    if len(all_train_losses) != 4 or len(all_val_losses) != 4 or len(colors) != 4 or len(model_labels) != 4:
        raise ValueError("Ensure there are exactly four models' losses, colors, and labels provided.")

    # Plot each model's training and validation loss
    for train_losses, val_losses, color, label in zip(all_train_losses, all_val_losses, colors, model_labels):
        plt.plot(train_losses, color=color, label=f'{label} Training Loss', linestyle='solid')
        plt.plot(val_losses, color=color, label=f'{label} Validation Loss', linestyle='dotted')

    plt.xlabel('Iterations')
    plt.ylabel('Loss')
    plt.title(title if title else 'Training and Validation Losses')
    plt.legend()
    plt.show()

# Example data for testing the function
train_losses_model1 = train_losses
val_losses_model1 = val_losses
train_losses_model2 = train_losses_m2
val_losses_model2 = val_losses_m2
train_losses_model3 = train_losses_m3
val_losses_model3 = val_losses_m3
train_losses_model4 = train_losses_m4
val_losses_model4 = val_losses_m4

colors = ['blue', 'green', 'red', 'purple']  # Different colors for each model
labels = ['Model 1', 'Model 2', 'Model 3', 'Model 4']  # Labels for each model

# Calling the function with the test data
showPlot([train_losses_model1, train_losses_model2, train_losses_model3, train_losses_model4],
         [val_losses_model1, val_losses_model2, val_losses_model3, val_losses_model4],
         colors, labels)
```



Predict Functions

In [ ]:
```python
def predict_m1(encoder, decoder, sentence, max_length=MAX_LENGTH):
    with torch.no_grad():
        input_tensor = tensorFromSentence(input_lang, sentence)
        input_length = input_tensor.size(0)
        encoder_hidden = encoder.initHidden()

        encoder_outputs = torch.zeros(max_length, encoder.hidden_size, device=device)

        # Encode each input tensor and update hidden states
        for ei in range(input_length):
            encoder_output, encoder_hidden = encoder(input_tensor[ei].unsqueeze(0), encoder_hidden)
            if ei < max_length:
                encoder_outputs[ei] = encoder_output[0, 0]

        decoder_input = torch.tensor([[SOS_token]], device=device)  # Start with SOS
        decoder_hidden = encoder_hidden  # Pass the encoder's final hidden state to the decoder

        decoded_words = []
        # decoder_attentions = torch.zeros(max_length, max_length)  # If attention is not used, this can be omitted

        for di in range(max_length):
            decoder_output, decoder_hidden = decoder(
                decoder_input, decoder_hidden)
            topv, topi = decoder_output.data.topk(1)
            if topi.item() == EOS_token:
                decoded_words.append('<EOS>')
                break
            else:
                decoded_words.append(output_lang.index2word[topi.item()])

            decoder_input = topi.squeeze().detach()

        return decoded_words
```

In [ ]:
```python
def predict_m2(encoder, decoder, sentence, max_length=150):
    with torch.no_grad():
        input_tensor = tensorFromSentence(input_lang, sentence)
        input_length = input_tensor.size()[0]
        encoder_hidden, encoder_cell = encoder.initHidden()

        encoder_outputs = torch.zeros(max_length, encoder.hidden_size, device=device)
        for ei in range(input_length):
            encoder_output, (encoder_hidden, encoder_cell) = encoder(input_tensor[ei], (encoder_hidden, encoder_cell))
            encoder_outputs[ei] += encoder_output[0, 0]

        decoder_input = torch.tensor([[SOS_token]], device=device)  # SOS

        decoder_hidden = encoder_hidden
        decoder_cell = encoder_cell

        decoded_words = []
        decoder_attentions = torch.zeros(max_length, max_length)

        for di in range(max_length):
            decoder_output, decoder_hidden, decoder_cell, decoder_attention = decoder(
                decoder_input, decoder_hidden, decoder_cell, encoder_outputs)
            decoder_attentions[di] = decoder_attention.data
            topv, topi = decoder_output.data.topk(1)
            if topi.item() == EOS_token:
                decoded_words.append('<EOS>')
                break
            else:
                decoded_words.append(output_lang.index2word[topi.item()])

            decoder_input = topi.squeeze().detach()

        return decoded_words, decoder_attentions[:di + 1]
```

In [ ]:
```
def predict_m3(encoder, decoder, sentence, max_length=MAX_LENGTH):
    with torch.no_grad():
        input_tensor = tensorFromSentence(input_lang, sentence)
        input_length = input_tensor.size()[0]
        encoder_hidden = encoder.initHidden()

        encoder_outputs = torch.zeros(max_length, encoder.hidden_size, device=device)

        for ei in range(input_length):
            encoder_output, encoder_hidden = encoder(input_tensor[ei],
                                                     encoder_hidden)
            encoder_outputs[ei] += encoder_output[0, 0]

        decoder_input = torch.tensor([[SOS_token]], device=device)  # SOS

        decoder_hidden = encoder_hidden

        decoded_words = []
        decoder_attentions = torch.zeros(max_length, max_length)

        for di in range(max_length):
            decoder_output, decoder_hidden, decoder_attention = decoder(
                decoder_input, decoder_hidden, encoder_outputs)
            decoder_attentions[di] = decoder_attention.data
            topv, topi = decoder_output.data.topk(1)
            if topi.item() == EOS_token:
                decoded_words.append('<EOS>')
                break
            else:
                decoded_words.append(output_lang.index2word[topi.item()])

            decoder_input = topi.squeeze().detach()

        return decoded_words, decoder_attentions[:di + 1]
```

In [ ]:
```
def predict_m4(encoder, decoder, sentence, planner, planner_tokenizer, stage_vocab, max_length=150):
    with torch.no_grad():
        input_tensor = tensorFromSentence(input_lang, sentence)
        input_length = input_tensor.size()[0]
        encoder_hidden, encoder_cell = encoder.initHidden()

        encoder_outputs = torch.zeros(max_length, encoder.hidden_size, device=device)

        for ei in range(input_length):
            encoder_output, (encoder_hidden, encoder_cell) = encoder(input_tensor[ei], (encoder_hidden, encoder_cell))
            encoder_outputs[ei] += encoder_output[0, 0]

        # Generate the content plan
        ingredients_text = sentence
        inputs = planner_tokenizer(ingredients_text, return_tensors='pt', max_length=512, truncation=True).to(device)
        content_plan_output = planner.generate(**inputs)
        content_plan = planner_tokenizer.decode(content_plan_output[0], skip_special_tokens=True).split()
        # print("Generated Content Plan:", content_plan)
        content_plan_indices = [stage_vocab.get(stage, stage_vocab['<unk>']) for stage in content_plan]
        content_plan_tensor = torch.tensor(content_plan_indices, device=device)

        decoder_input = torch.tensor([[SOS_token]], device=device)  # SOS

        decoder_hidden = encoder_hidden
        decoder_cell = encoder_cell

        decoded_words = []
        decoder_attentions = torch.zeros(max_length, max_length)

        current_stage_index = 0
        current_stage = content_plan_tensor[current_stage_index].unsqueeze(0).unsqueeze(0)  # Correctly shaped tensor

        for di in range(max_length):
            decoder_output, decoder_hidden, decoder_cell, decoder_attention = decoder(
                decoder_input, decoder_hidden, decoder_cell, encoder_outputs, current_stage)
            decoder_attentions[di] = decoder_attention.data
            topv, topi = decoder_output.data.topk(1)
            if topi.item() == EOS_token:
                decoded_words.append('<EOS>')
                break
            else:
                decoded_words.append(output_lang.index2word[topi.item()])

            decoder_input = topi.squeeze().detach()

            if criteria_for_next_stage(decoder_input):  # Define this function
                current_stage_index += 1
                if current_stage_index < len(content_plan_tensor):
                    current_stage = content_plan_tensor[current_stage_index].unsqueeze(0).unsqueeze(0)

        return decoded_words, decoder_attentions[:di + 1]
```

Quantitative Evaluation

In [ ]:
```
pip install pandas nltk
```

```
Requirement already satisfied: pandas in c:\users\mirik\anaconda3\envs\nlp\lib\site-packages (2.2.2)
Requirement already satisfied: nltk in c:\users\mirik\anaconda3\envs\nlp\lib\site-packages (3.8.1)
Requirement already satisfied: numpy>=1.23.2 in c:\users\mirik\anaconda3\envs\nlp\lib\site-packages (from pandas) (1.26.4)
Requirement already satisfied: python-dateutil>=2.8.2 in c:\users\mirik\anaconda3\envs\nlp\lib\site-packages (from pandas) (2.
8.2)
Requirement already satisfied: pytz>=2020.1 in c:\users\mirik\anaconda3\envs\nlp\lib\site-packages (from pandas) (2024.1)
Requirement already satisfied: tzdata>=2022.7 in c:\users\mirik\anaconda3\envs\nlp\lib\site-packages (from pandas) (2024.1)
Requirement already satisfied: click in c:\users\mirik\anaconda3\envs\nlp\lib\site-packages (from nltk) (8.1.7)
Requirement already satisfied: joblib in c:\users\mirik\anaconda3\envs\nlp\lib\site-packages (from nltk) (1.4.2)
Requirement already satisfied: regex>=2021.8.3 in c:\users\mirik\anaconda3\envs\nlp\lib\site-packages (from nltk) (2024.5.15)
Requirement already satisfied: tqdm in c:\users\mirik\anaconda3\envs\nlp\lib\site-packages (from nltk) (4.66.4)
Requirement already satisfied: six>=1.5 in c:\users\mirik\anaconda3\envs\nlp\lib\site-packages (from python-dateutil>=2.8.2->p
andas) (1.16.0)
Requirement already satisfied: colorama in c:\users\mirik\anaconda3\envs\nlp\lib\site-packages (from click->nltk) (0.4.6)
Note: you may need to restart the kernel to use updated packages.
```

In [ ]:
```
import pandas as pd
import nltk
from nltk.translate.bleu_score import sentence_bleu, SmoothingFunction
from nltk.translate.meteor_score import meteor_score
from nltk.translate import bleu_score, meteor_score
nltk.download('punkt')
nltk.download('wordnet')
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\mirik\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package wordnet to
[nltk_data]     C:\Users\mirik\AppData\Roaming\nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
```

Out[ ]: True

In [ ]:
```
def quantitative_analysis(test_pairs, models, model_functions):
    results = {
        'Model 1': {'BLEU-4': [], 'METEOR': [], 'Avg. % Given Items': [], 'Avg. Extra Items': []},
        'Model 2': {'BLEU-4': [], 'METEOR': [], 'Avg. % Given Items': [], 'Avg. Extra Items': []},
        'Model 3': {'BLEU-4': [], 'METEOR': [], 'Avg. % Given Items': [], 'Avg. Extra Items': []},
        'Model 4': {'BLEU-4': [], 'METEOR': [], 'Avg. % Given Items': [], 'Avg. Extra Items': []}
    }

    # Iterate over all the test pairs
    for idx, (ingredient, reference) in enumerate(test_pairs):
        reference_tokens = nltk.word_tokenize(reference.lower())

        for model_index, model in enumerate(models):
            # Select the appropriate prediction function and arguments
            if model_index == 0:  # Model 1
                output_words = model_functions[model_index](model[0], model[1], ingredient)
            elif model_index == 1:  # Model 2
                output_words, attention = model_functions[model_index](model[0], model[1], ingredient)
            elif model_index == 2:  # Model 3
                output_words, attention = model_functions[model_index](model[0], model[1], ingredient)
            elif model_index == 3:  # Model 4
                output_words, attention = model_functions[model_index](model[0], model[1], ingredient, planner, planner_token

            output_sentence = ' '.join(output_words)
            pred_tokens = nltk.word_tokenize(output_sentence.lower())

            # Calculate BLEU-4 and METEOR
            bleu = bleu_score.sentence_bleu([reference_tokens], pred_tokens, smoothing_function=bleu_score.SmoothingFunction(
            meteor = meteor_score.single_meteor_score(reference_tokens, pred_tokens)  # Use tokenized reference and predictio

            # Calculate Avg. % Given Items and Avg. Extra Items
            ref_set = set(reference_tokens)
            pred_set = set(pred_tokens)
            avg_given_items = 100 * len(ref_set & pred_set) / len(ref_set) if ref_set else 0
            avg_extra_items = len(pred_set - ref_set)

            # Append results
            model_name = f'Model {model_index + 1}'
            results[model_name]['BLEU-4'].append(bleu)
            results[model_name]['METEOR'].append(meteor)
            results[model_name]['Avg. % Given Items'].append(avg_given_items)
            results[model_name]['Avg. Extra Items'].append(avg_extra_items)

    # Average the results for each metric
    for model_result in results.values():
        for key in model_result:
            model_result[key] = sum(model_result[key]) / len(model_result[key]) if model_result[key] else 0

    return results
```

In [ ]:
```
models = [(encoder_m1, decoder_m1), (encoder_m2, decoder_m2), (encoder_m3, decoder_m3), (encoder_m4, decoder_m4)]
model_functions = [predict_m1, predict_m2, predict_m3, predict_m4]
results = quantitative_analysis(test_pairs, models, model_functions)
# Convert results to a DataFrame for easier viewing
results_df = pd.DataFrame(results)
print(results_df)
```

```
                      Model 1    Model 2    Model 3   Model 4
BLEU-4               0.008282   0.007393   0.011448  0.005452
METEOR               0.080962   0.083673   0.135333  0.061210
Avg. % Given Items  12.096125  11.994506  18.125431  9.076380
Avg. Extra Items     9.489231   6.398462  17.435385  7.850769
```

In [ ]:
```
def read_recipes_from_file(file_path):
    with open(file_path, 'r', encoding='utf-8') as file:
        lines = file.readlines()

    # Assuming structure: even lines after titles are ingredients and recipes
    gold_recipe = lines[4].strip()  # Adjust line numbers based on actual structure
    generated_recipe = lines[7].strip()

    return gold_recipe, generated_recipe
```

In [ ]:
```
from nltk.tokenize import word_tokenize

def extract_ingredients(tokens):
    ingredients = []
    capture = False
    current_ingredient = []
    for token in tokens:
        if token == '<':
            capture = True
            current_ingredient = []
        elif token == '>':
            if current_ingredient:
                ingredients.append(''.join(current_ingredient))
            capture = False
        elif capture:
            current_ingredient.append(token)

    return set(ingredients)

def calculate_metrics(gold_recipe, generated_recipe):
    # Tokenize the recipes
    gold_tokens = word_tokenize(gold_recipe.lower())
    generated_tokens = word_tokenize(generated_recipe.lower())
    # print(gold_tokens)
    # print(generated_tokens)

    # Extract ingredients using the new method
    gold_ingredients = extract_ingredients(gold_tokens)
    generated_ingredients = extract_ingredients(generated_tokens)

    # Calculate BLEU-4
    bleu_score_value = bleu_score.sentence_bleu([gold_tokens], generated_tokens, smoothing_function=bleu_score.SmoothingFunct

    # Calculate METEOR
    meteor_score_value = meteor_score.single_meteor_score(gold_tokens, generated_tokens)

    # Calculate Average Percentage of Given Items
    common_ingredients = gold_ingredients.intersection(generated_ingredients)
    avg_given_items = 100 * len(common_ingredients) / len(gold_ingredients) if gold_ingredients else 0

    # Calculate Average Extra Items
    avg_extra_items = len(generated_ingredients - gold_ingredients)

    return bleu_score_value, meteor_score_value, avg_given_items, avg_extra_items
```

```
In [ ]: # File path to the text file
        metric_sample_file_path = file_path + 'metric_sample.txt'

        gold_recipe, generated_recipe = read_recipes_from_file(metric_sample_file_path)

        bleu, meteor, avg_given, avg_extra = calculate_metrics(gold_recipe, generated_recipe)

        print(f"BLEU-4 Score: {bleu:.4f}")
        print(f"METEOR Score: {meteor:.4f}")
        print(f"Avg. % Given Items: {avg_given:.2f}%")
        print(f"Avg. Extra Items: {avg_extra}")
```

```
BLEU-4 Score: 0.2757
METEOR Score: 0.5479
Avg. % Given Items: 100.00%
Avg. Extra Items: 2
```

Qualitative Evaluation

```
In [ ]: # Load the existing CSV file
        generated_file_path = file_path +  "generated_31044891.csv"
        generated_data = pd.read_csv(generated_file_path)
        generated_data['Ingredients'] = generated_data['Ingredients'].apply(normalizeString)
```

```
In [ ]: import pandas as pd

        def generate(encoder, decoder, model, ingredients, generated_data):
            """Evaluates the encoder and decoder models by processing each ingredient and updates the DataFrame accordingly."""
            # Define column names based on the model number
            column_names = {
                1: "Generated Recipe - Baseline 1",
                2: "Generated Recipe - Baseline 2",
                3: "Generated Recipe - Extended 1",
                4: "Generated Recipe - Extended 2"
            }
```

```
    column_name = column_names[model]

    # Ensure the column exists in the DataFrame
    if column_name not in generated_data.columns:
        generated_data[column_name] = pd.Series([None] * len(generated_data), index=generated_data.index)

    # Generate predictions for each ingredient
    predictions = []
    for idx, ingredient in enumerate(ingredients):
        # Determine which model evaluation function to call
        if model == 1:
            output_words = predict_m1(encoder, decoder, ingredient)
        elif model == 2:
            output_words, attention = predict_m2(encoder, decoder, ingredient)
        elif model == 3:
            output_words, attention = predict_m3(encoder, decoder, ingredient)
        elif model == 4:
            output_words, attention = predict_m4(encoder, decoder, ingredient, planner, planner_tokenizer, stage_vocab)

        output_sentence = ' '.join(output_words)
        predictions.append(output_sentence)

        # Assign predictions to the DataFrame
        generated_data.at[idx, column_name] = output_sentence

    return generated_data

# Extract ingredients for prediction
ingredients = generated_data['Ingredients'].tolist()

# Predict recipes for each model and update the CSV file
for model in range(1, 5):
    encoder = globals()[f'encoder_m{model}']
    decoder = globals()[f'decoder_m{model}']
    generated_data = generate(encoder, decoder, model, ingredients, generated_data)

# Save the updated DataFrame to the same CSV file
generated_data.to_csv(generated_file_path, index=False)

print("Recipe prediction and data saving completed.")
```

```
C:\Users\mirik\AppData\Local\Temp\ipykernel_33872\3864532078.py:35: FutureWarning: Setting an item of incompatible dtype is de
precated and will raise an error in a future version of pandas. Value 'combine all ingredients in a large bowl . add the remai
ning ingredients . add the chicken . <EOS>' has dtype incompatible with float64, please explicitly cast to a compatible dtype
first.
  generated_data.at[idx, column_name] = output_sentence
C:\Users\mirik\AppData\Local\Temp\ipykernel_33872\3864532078.py:35: FutureWarning: Setting an item of incompatible dtype is de
precated and will raise an error in a future version of pandas. Value 'beat eggs and salt together . beat until smooth . beat
until smooth . beat until smooth . beat until smooth . beat until smooth . beat until smooth . beat until smooth . beat until
smooth . beat until smooth . beat until smooth . beat egg whites until smooth . beat until smooth . beat until smooth . beat u
ntil smooth . beat until smooth . beat until smooth . beat until smooth . beat until smooth . beat until smooth . beat until s
mooth . beat until smooth . beat until smooth . beat until smooth . beat until smooth . beat until smooth . beat until smooth
. beat until smooth . beat until smooth . beat until smooth . beat until smooth . beat until smooth . beat egg whites until sm
ooth . beat until smooth . beat egg whites until smooth . beat until' has dtype incompatible with float64, please explicitly c
ast to a compatible dtype first.
  generated_data.at[idx, column_name] = output_sentence
C:\Users\mirik\AppData\Local\Temp\ipykernel_33872\3864532078.py:35: FutureWarning: Setting an item of incompatible dtype is de
precated and will raise an error in a future version of pandas. Value 'combine sugar and sugar in a saucepan and bring to a bo
il . add the flour and salt and beat until well blended . add the flour and salt and beat until well blended . add the flour a
nd salt and beat until well blended . add the flour and baking powder . add the dry ingredients and beat well . add the dry in
gredients and beat well . add the dry ingredients and beat well . add the dry ingredients and beat well . add the dry ingredie
nts and beat well . add the dry ingredients and beat until well blended . add the dry ingredients and beat well . add the milk
and vanilla . pour the batter into the prepared pan and bake for 30 minutes . cool and cut into squares . <EOS>' has dtype inc
ompatible with float64, please explicitly cast to a compatible dtype first.
  generated_data.at[idx, column_name] = output_sentence
c:\Users\mirik\anaconda3\envs\NLP\Lib\site-packages\transformers\generation\utils.py:1288: UserWarning: Neither `max_length` n
or `max_new_tokens` has been set, `max_length` will default to 20 (`generation_config.max_length`). Controlling `max_length` v
ia the config is deprecated and `max_length` will be removed from the config in v5 of Transformers -- we recommend using `max_
new_tokens` to control the maximum length of the generation.
  warnings.warn(
C:\Users\mirik\AppData\Local\Temp\ipykernel_33872\3864532078.py:35: FutureWarning: Setting an item of incompatible dtype is de
precated and will raise an error in a future version of pandas. Value 'mix the egg mixture . add the raisins . add the raisins
. add the raisins . add the raisins . add the raisins . add the raisins . add the raisins . add the raisins . pour into a grea
sed muffin tins . bake at 350 degrees for 45 minutes . <EOS>' has dtype incompatible with float64, please explicitly cast to a
compatible dtype first.
  generated_data.at[idx, column_name] = output_sentence
c:\Users\mirik\anaconda3\envs\NLP\Lib\site-packages\transformers\generation\utils.py:1288: UserWarning: Neither `max_length` n
or `max_new_tokens` has been set, `max_length` will default to 20 (`generation_config.max_length`). Controlling `max_length` v
ia the config is deprecated and `max_length` will be removed from the config in v5 of Transformers -- we recommend using `max_
new_tokens` to control the maximum length of the generation.
  warnings.warn(
Recipe prediction and data saving completed.
```

In [ ]:
```python
train_losses_m1 = train_losses
val_losses_m1 = val_losses
```

In [ ]:
```python
def generate(encoder, decoder, model, ingredients):
    """Evaluates the encoder and decoder models by processing given ingredients and prints the generated recipe."""
    # Determine which model evaluation function to call
    if model == 1:
        output_words = predict_m1(encoder, decoder, ingredients)
        model_name = "Baseline 1"
    elif model == 2:
        output_words, attention = predict_m2(encoder, decoder, ingredients)
        model_name = "Baseline 2"
    elif model == 3:
        output_words, attention = predict_m3(encoder, decoder, ingredients)
        model_name = "Extension 1"
    elif model == 4:
        output_words, attention = predict_m4(encoder, decoder, ingredients, planner, planner_tokenizer, stage_vocab)
        model_name = "Extension 2"

    output_sentence = ' '.join(output_words)

    # Print the model and the corresponding generated recipe
    print(f"{model_name} Generated Recipe: {output_sentence}\n")

# Sample ingredient input
ingredients = "2 c sugar, 1/4 c lemon juice, 1 c water, 1/3 c orange juice, 8 c strawberries"

# Predict recipes for each model and print them
for model in range(1, 5):
    encoder = globals()[f'encoder_m{model}']
    decoder = globals()[f'decoder_m{model}']
    generate(encoder, decoder, model, ingredients)
```

```
Baseline 1 Generated Recipe: combine all ingredients in a large bowl . add the onion and garlic . add the onion and garlic . <EOS>

Baseline 2 Generated Recipe: combine all ingredients in a saucepan . <EOS>

Extension 1 Generated Recipe: combine the sugar and salt . add the milk and beat until well blended . add the dry ingredients and beat well . add the dry ingredients and beat well . add the dry ingredients and beat well . add the dry ingredients and beat well . add the dry ingredients and beat well . add the dry ingredients and beat until well blended . add the flour and salt and beat until well blended . add the dry ingredients and beat well . add the milk and beat until well blended . add the milk and beat until well blended . add the milk and beat until well blended . add the milk and beat until well blended . add the milk and beat until well blended . add the milk and beat until well blended . add the milk and beat until well blended . add the dry

Extension 2 Generated Recipe: in a small saucepan combine all ingredients . <EOS>
```

```
c:\Users\mirik\anaconda3\envs\NLP\Lib\site-packages\transformers\generation\utils.py:1288: UserWarning: Neither `max_length` nor `max_new_tokens` has been set, `max_length` will default to 20 (`generation_config.max_length`). Controlling `max_length` via the config is deprecated and `max_length` will be removed from the config in v5 of Transformers -- we recommend using `max_new_tokens` to control the maximum length of the generation.
  warnings.warn(
```