Recurrent Neural Networks

Inclass Project 3 - MA4144

This project contains 10 tasks/questions to be completed, some require written answers. Open a markdown cell below the respective question that require written answers and provide (type) your answers. Questions that required written answers are given in blue fonts. Almost all written questions are open ended, they do not have a correct or wrong answer. You are free to give your opinions, but please provide related answers within the context.

After finishing project run the entire notebook once and **save the notebook as a pdf** (File menu -> Save and Export Notebook As -> PDF). You are **required to upload this PDF on moodle**.

Outline of the project

The aim of the project is to build a RNN model to suggest autocompletion of half typed words. You may have seen this in many day today applications; typing an email, a text message etc. For example, suppose you type in the four letter "univ", the application may suggest you to autocomplete it by "university".

We will train a RNN to suggest possible autocompletes given 3 - 4 starting letters. That is if we input a string "univ" hopefully we expect to see an output like "university", "universal" etc.

For this we will use a text file (wordlist.txt) containing 10,000 common English words (you'll find the file on the moodle link). The list of words will be the "**vocabulary**" for our model.

We will use the Python **torch library** to implement our autocomplete model.

Use the below cell to use any include any imports

In [1]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt
import random
from tqdm import tqdm
import time
```

Section 1: Preparing the vocabulary

In [2]:
```
WORD_SIZE = 13
RANDOM_SEED = 123
torch.manual_seed(RANDOM_SEED)
random.seed(RANDOM_SEED)
```

Q1. In the following cell provide code to load the text file (each word is in a newline), then extract the words (in lowercase) into a list.

For practical reasons of training the model we will only use words that are longer that 3 letters and that have a maximum length of WORD_SIZE (this will be a constant we set at the beginning - you can change this and experiment with different WORD_SIZEs). As seen above it is set to 13.

So out of the extracted list of words filter out those words that match our criteria on word length.

To train our model it is convenient to have words/strings of equal length. We will choose to convert every word to length of WORD_SIZE, by adding underscores to the end of the word if it is initially shorter than WORD_SIZE. For example, we will convert the word "university" (word length 10) into "university___" (wordlength 13). In your code include this conversion as well.

Store the processed WORD_SIZE lengthed strings in a list called vocab.

In [3]:
```
#load data each word in a new line, return a list of words
#make sure all are lower case
def load_data(file_name):
    with open(file_name, 'r') as f:
        data = f.readlines()
    return data

#filter out words that are longer than 3 letters and less than word size
def filter_words(data):
    data = [word.strip().lower() for word in data if 3 < len(word.strip()) <= WORD_
    return data

#convert all words to same length of word size by padding with underscores at the e
def pad_words(data):
    data = [word + '_'*(WORD_SIZE - len(word)) for word in data]
```

```
    return data

vocab = pad_words(filter_words(load_data('wordlist.txt')))
```

In the above explanation it was mentioned "for practical reasons of training the model we will only use words that are longer that 3 letters and that have a certain maximum length". In your opinion what could be those practical? Will hit help to build a better model?

Answer

Shorter words carry less amounts of information. They do not contribute to capture meaningful patterns, but only introduce noise to the training data. Also it will help model to generalize better focusing on meaningful data. Model is less prone to overfit when large no of inuseful short data is being filtered out.

Longer words carry more information, but takes more memory and resources, thus making the training more timely and expensive. Filtering out these can yield the faster efficient training specially when training larger models.

Q2 To input words into the model, we will need to convert each letter/character into a number. as we have seen above, the only characters in our list vocab will be the underscore and lowercase english letters. so we will convert these 27 characters into numbers as follows: underscore -> 0, 'a' -> 1, 'b' -> 2, $\cdots$, 'z' -> 26. In the following cell,

(i) Implement a method called char_to_num, that takes in a valid character and outputs its numerical assignment.

(ii) Implement a method called num_to_char, that takes in a valid number from 0 to 26 and outputs the corresponding character.

(iii) Implement a method called word_to_numlist, that takes in a word from our vocabulary and outputs a (torch) tensor of numbers that corresponds to each character in the word in that order. For example: the word "united_____" will be converted to tensor([21, 14, 9, 20, 5, 4, 0, 0, 0, 0, 0, 0, 0]). You are encouraged to use your char_to_num method for this.

(iv) Implement a method called numlist_to_word, that does the opposite of the above described word_to_numlist, given a tensor of numbers from 0 to 26, outputs the corresponding word. You are encouraged to use your num_to_char method for this.

Note: As mentioned since we are using the torch library we will be using tensors instead of the usual python lists or numpy arrays. Tensors are the list equivalent in torch. Torch models only accept tensors as input and they output tensors.

In [4]:
```
def char_to_num(char):
    if(char=="_"):
        return 0
    num=ord(char)-ord("a")+1
    return(num)
```

```
def num_to_char(num):
    if(num==0):
        return "-"
    char=chr(num+ord("a")-1)
    return(char)

def word_to_numlist(word):
    numlist = [char_to_num(char) for char in word]
    numlist = torch.tensor(numlist).long()
    return(numlist)

def numlist_to_word(numlist):
    numlist = numlist.tolist()
    word = ''.join([num_to_char(num) for num in numlist])
    return(word)
```

In [5]:
```
print(char_to_num("a"))
print(num_to_char(1))
print(word_to_numlist("flower"))
print(numlist_to_word(torch.tensor([ 6, 12, 15, 23,  5, 18])))
```

```
1
a
tensor([ 6, 12, 15, 23,  5, 18])
flower
```

We convert letter into just numbers based on their aphabetical order, I claim that it is a very bad way to encode data such as letters to be fed into learning models, please write your explanation to or against my claim. If you are searching for reasons, the keyword 'categorical data' may be useful. Although the letters in our case are not treated as categorical data, the same reasons as for categorical data is applicable. Even if my claim is valid, at the end it won't matter due to something called "embedding layers" that we will use in our model. What is an embedding layer? What is it's purpose? Explain.

Answer

Yes, I agree. Encoding data that way might lead to inefficient training, leading to longer training times and poor generalizations. Model might make misinterpretations regarding the patterns, and thus lead to a poor performance. (Ex: model interprets these numbers as having ordinal meaning, assumes distance between a,b is same as distance between b,c)

Also that way, cant capture semantic similarity between words. (Ex: rose and lily is more related than rose and dog)

For catergorical data which don;t have numerical meaning or order, but only discrete groups of catogeries, we can follow several encoding methods like, onehot encoding, label encoding, frequency encoding etc.

Embedding Layer : A neural network layer that transforms categorical data (like words) into dense, continuous vector spaces where similar items are closer together.

Purpose : Finding similarity between words, where words with similar meanings placed near each other.

Embedding layer helps words learn the semantic relationships. therefore it solves the poor encoding by mapping each character or word to a meaningful vector. At the end the way we encode data won't really matter.

Section 2: Implementing the Autocomplete model

We will implement a RNN model based on LSTM. The video tutorial will be useful. Our model will be only one hidden layer, but feel free to sophisticate with more layers after the project for your own experiments.

Our model will contain all the training and prediction methods as single package in a class (autocompleteModel) we will define and implement below.

In [6]:
```
LEARNING_RATE = 0.005
```

In [7]:
```
class autocompleteModel(nn.Module):

    #Constructor
    def __init__(self, alphabet_size, embed_dim, hidden_size, num_layers):
        super().__init__()

        #Set the input parameters to self parameters
        self.alphabet_size = alphabet_size
        self.embed_dim = embed_dim
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        #Initialize the layers in the model:
        #1 embedding layer, 1 - LSTM cell (hidden layer), 1 fully connected layer w
        self.embedding = nn.Embedding(alphabet_size, embed_dim)
        self.lstm = nn.LSTMCell(embed_dim, hidden_size)
        self.fc = nn.Linear(hidden_size, alphabet_size)


    #Feedforward
    def forward(self, character, hidden_state, cell_state):

        #Perform feedforward in order
        #1. Embed the input (one charcter represented by a number)
        embedded = self.embedding(character)
        #2. Feed the embedded output to the LSTM cell
        (hidden_state, cell_state) = self.lstm(embedded, (hidden_state, cell_state)
        #3. Feed the LSTM output to the fully connected layer to obtain the output
        #4. return the output, and both the hidden state and cell state from the LS
        output = self.fc(hidden_state)

        return output, hidden_state, cell_state
```

```
    #Intialize the first hidden state and cell state (for the start of a word) as z
    def initial_state(self):

        h0 = torch.zeros(1, self.hidden_size)
        c0 = torch.zeros(1, self.hidden_size)

        return (h0, c0)

    #Train the model in epochs given the vocab, the training will be fed in batches
    def trainModel(self, vocab, epochs = 5, batch_size = 100, learning_rate = LEARN
        #Convert the model into train mode
        self.train()
        #Set the optimizer (ADAM), you may need to provide the model parameters  an
        optimizer = torch.optim.Adam(self.parameters(), lr = learning_rate)

        #Keep a log of the loss at the end of each training cycle.
        loss_log = []
        avg_loss_log = []

        for e in range(epochs):
            epoch_loss = 0
            #Shuffle the vocab list the start of each epoch

            random.shuffle(vocab)
            num_iter = len(vocab)//batch_size

            for i in range(num_iter):

                    #Set the loss to zero, initialize the optimizer with zero_grad

                    loss = 0
                    optimizer.zero_grad()

                    vocab_batch = vocab[i*batch_size:(i+1)*batch_size]
                    for word in vocab_batch:

                        #Initialize the hidden state and cell state at the start of

                        hidden_state,cell_state= self.initial_state()


                        #Convert the word into a tensor of number and create input
                        #Input will be the first WORD_SIZE - 1 charcters and target

                        word_numlist = word_to_numlist(word)
                        inputs = word_numlist[:-1]
                        targets = word_numlist[1:]



                        #Loop through each character (as a number) in the word
                        for c in range(WORD_SIZE - 1):
                            #Feed the cth character to the model (feedforward) and
                            output, hidden_state, cell_state = self(inputs[c].unsqu
                            loss += torch.nn.functional.cross_entropy(output, targe
```

```
                #Compute the average loss per word in the batch and perform bac

                loss = loss/batch_size
                loss.backward()

                #Update model parameters using the optimizer

                optimizer.step()

                #Update the loss_log
                loss_log.append(loss.item())

                epoch_loss += loss.item()


        avg_epoch_loss = epoch_loss / num_iter
        avg_loss_log.append(avg_epoch_loss)

        print(f"Epoch {e+1}/{epochs} -Avg_Loss: {avg_epoch_loss:.4f} -Final_Los



    #Plot a graph of the variation of the loss.
    if plot:
        plt.plot(loss_log)
        plt.xlabel('Iterations')
        plt.ylabel('Loss')
        plt.title('Training Loss Over Time')
        plt.legend()
        plt.grid()
        plt.show()



    return loss_log


#Perform autocmplete given a sample of strings (typically 3-5 starting letters)
def autocomplete(self, sample):

    #Convert the model into evaluation mode
    self.eval()
    completed_list=[]

    #In the following loop for each sample item initialize hidden and cell stat
    #You will have to convert the output into a softmax (you may use your softm
    for literal in sample:
        hidden_state,cell_state=self.initial_state()
        literal_tensor=word_to_numlist(literal)

        predict=[]

        for i in range(len(literal_tensor)):
            output,hidden_state,cell_state=self.forward(literal_tensor[i].unsqu
```

```
            for _ in range(WORD_SIZE - len(literal)):
                prob_output = torch.nn.functional.softmax(output, dim=-1)
                char_idx = torch.multinomial(prob_output.squeeze(0), 1).item()
                predict.append(num_to_char(char_idx))

                output, hidden_state, cell_state = self.forward(torch.tensor(char_i

            completed_literal = literal + ''.join(predict)
            completed_list.append(completed_literal)

        return(completed_list)
```

Section 3: Using and evaluating the model

(i) Feel free to initialize a autocompleteModel using different embedding dimensions and hidden layer sizes. Use different learning rates, epochs, batch sizes. Train the best model you can. Show the loss curves in you answers.

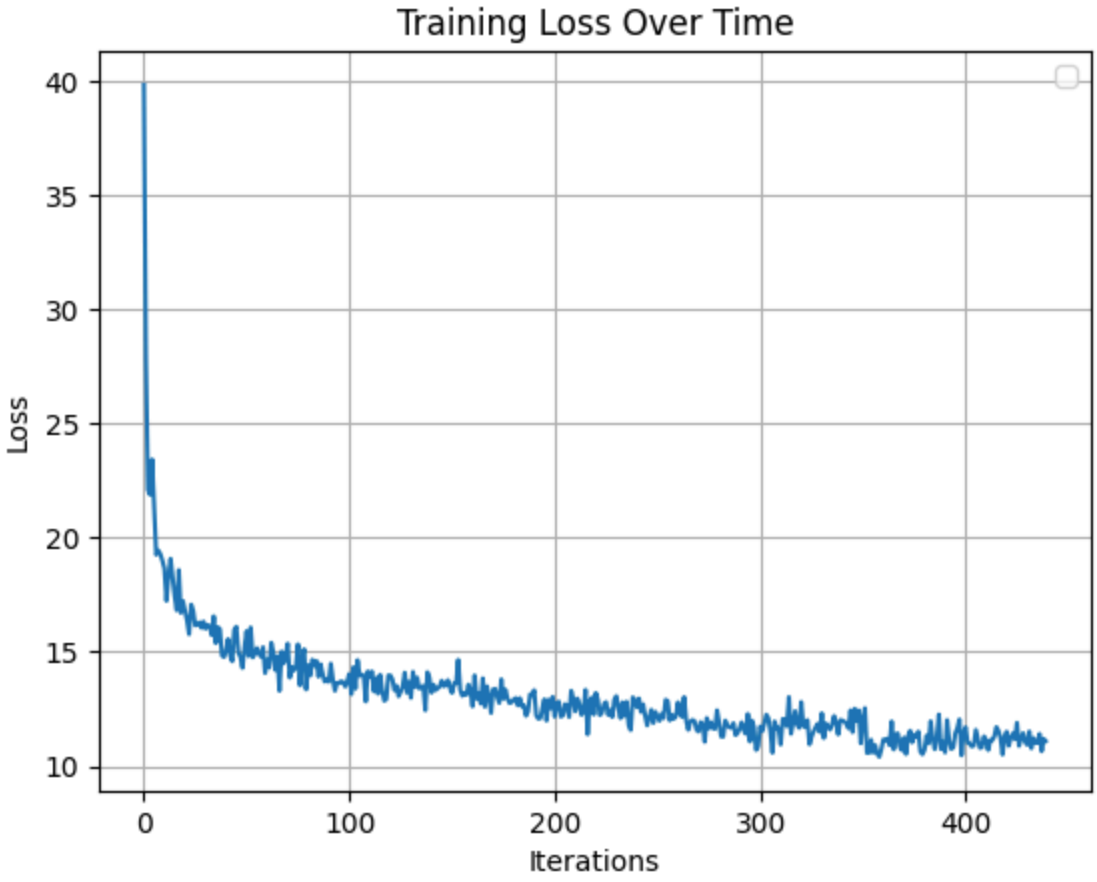
(ii) Evaluate it on different samples of partially filled in words. Eg: ["univ", "math", "neur", "engin"] etc. Please show outputs for different samples.

Comment on the results. Is it successful? Do you see familiar substrings in the generated tesxt such as "tion", "ing", "able" etc. What are your suggestions to improve the model?

Answer (to write answers edit this cell)

In [21]:
```
ALPHABET_SIZE = 27
EMBED_DIM = 128
HIDDEN_SIZE = 256
NUM_LAYERS = 1
LEARNING_RATE = 0.005
EPOCHS = 5
BATCH_SIZE = 100

autocomplete_model = autocompleteModel(ALPHABET_SIZE, EMBED_DIM, HIDDEN_SIZE, NUM_L

autocomplete_model.trainModel(vocab, epochs=EPOCHS, batch_size=BATCH_SIZE,plot=True

test_samples = ["univ", "math", "neur", "engin"]

completed_words = autocomplete_model.autocomplete(test_samples)

for sample, completion in zip(test_samples, completed_words):
    print(f"Input: {sample} -> Completion: {completion}")
```

```
Epoch 1/5 -Avg_Loss: 16.3675 -Final_Loss: 14.041800498962402
Epoch 2/5 -Avg_Loss: 13.5113 -Final_Loss: 12.949760437011719
Epoch 3/5 -Avg_Loss: 12.5434 -Final_Loss: 13.024914741516113
Epoch 4/5 -Avg_Loss: 11.7644 -Final_Loss: 12.537410736083984
Epoch 5/5 -Avg_Loss: 11.1237 -Final_Loss: 11.090359687805176
```

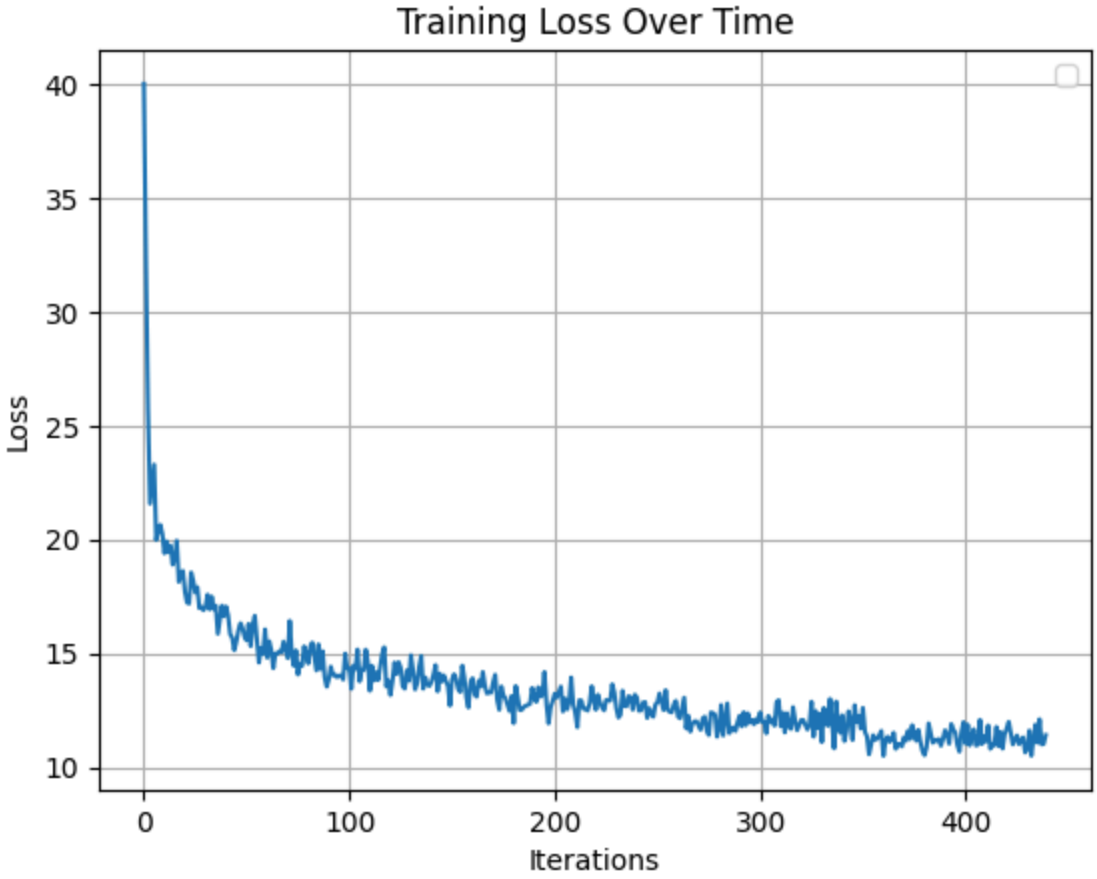



```
Input: univ -> Completion: univoigizer--
Input: math -> Completion: mathadjapen--
Input: neur -> Completion: neurobitl----
Input: engin -> Completion: engina-------
```

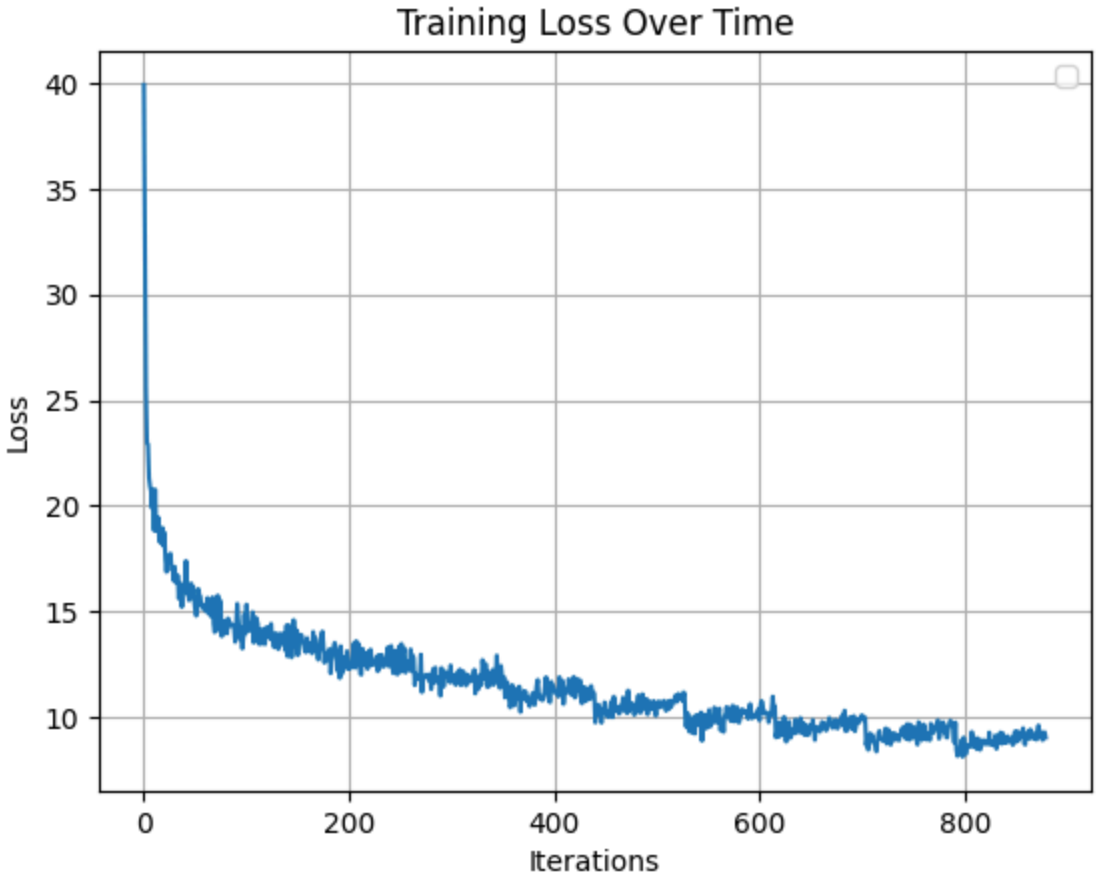
In [23]:
```python
from itertools import product
import matplotlib.pyplot as plt

# Define the hyperparameter options
embed_dims = [64,32]
hidden_sizes = [256]
learning_rates = [0.005,0.001]
batch_sizes = [100]
epoch_counts = [5,10]

# Initialize an empty list to store results
results = []
min_error = float('inf')  # Set initial minimum error to infinity
best_model_params = None   # To store the best model parameters

# Create all combinations of hyperparameters
param_combinations = product(embed_dims, hidden_sizes, learning_rates, batch_sizes,

# Iterate over each combination of hyperparameters
for embed_dim, hidden_size, lr, batch_size, epochs in param_combinations:
    print(f"Training model with embed_dim={embed_dim}, hidden_size={hidden_size}, l

    # Initialize the model
    model = autocompleteModel(27, embed_dim, hidden_size, 1)
```

```
    # Train the model and get the loss log
    loss_log = model.trainModel(vocab, epochs=epochs, batch_size=batch_size, learni

    # Get the final loss from the loss log
    final_loss = loss_log[-1]  # Final loss after the last epoch

    # Store the results
    results.append({
        'embed_dim': embed_dim,
        'hidden_size': hidden_size,
        'num_layers': 1,  # Assuming num_layers is fixed at 1, adjust if necessary
        'learning_rate': lr,
        'batch_size': batch_size,
        'epochs': epochs,
        'final_loss': final_loss,
        'loss_log': loss_log
    })

    # Print the final loss for the current hyperparameter combination
    print(f"Embed Dim: {embed_dim}, Hidden Size: {hidden_size}, "
          f"Num Layers: 1, Learning Rate: {lr}, "
          f"Batch Size: {batch_size}, Epochs: {epochs}, Final Loss: {final_loss}")


    # Check if the final loss is less than the minimum error
    if final_loss < min_error:
        min_error = final_loss
        best_model_params = {
            'embed_dim': embed_dim,
            'hidden_size': hidden_size,
            'learning_rate': lr,
            'batch_size': batch_size,
            'epochs': epochs
        }


# After the loop, display the best model parameters and its final loss
print(f"Best Model Parameters: {best_model_params}")
print(f"Minimum Final Loss: {min_error}")

# Optionally, plot the loss curves for the best model
plt.plot(best_model_params['loss_log'])
plt.title('Loss Curve of Best Model')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.ylim(bottom=0)

plt.grid()
plt.show()
```

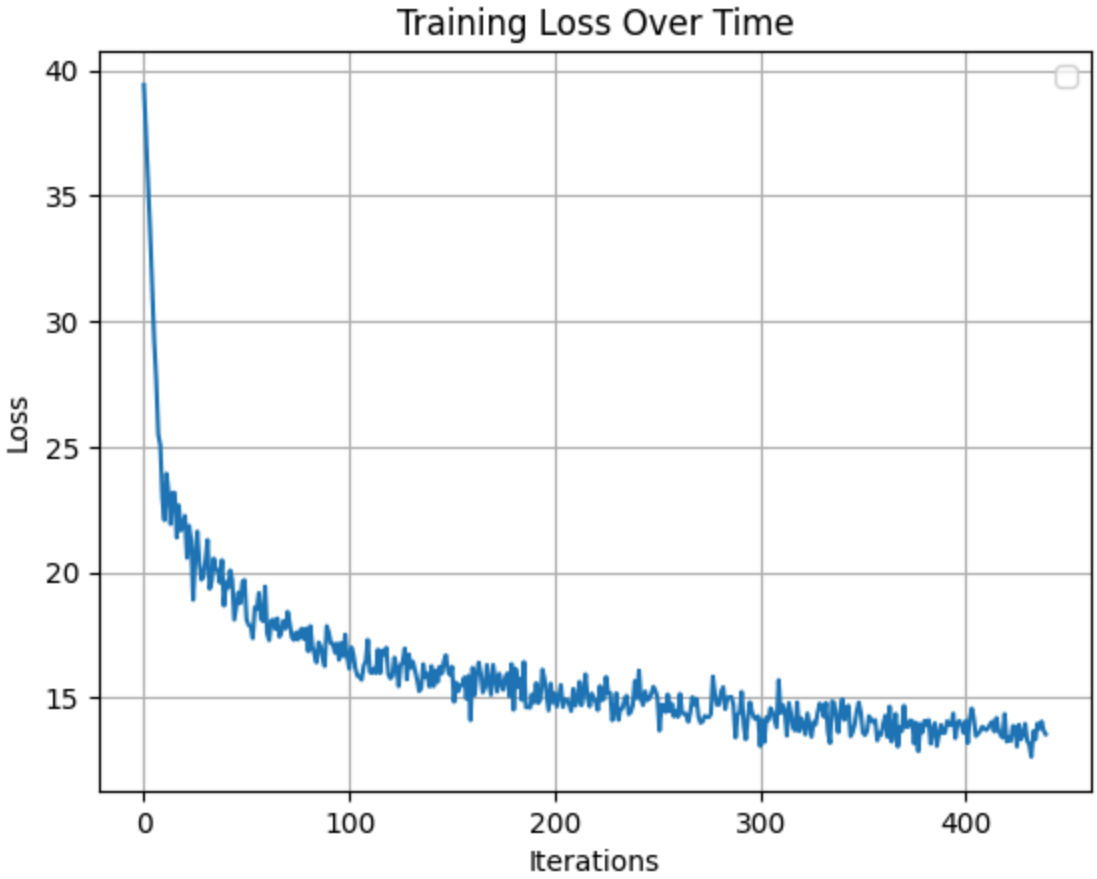
```
Training model with embed_dim=64, hidden_size=256, lr=0.005, batch_size=100, epochs=
5
Epoch 1/5 -Avg_Loss: 17.2576 -Final_Loss: 15.095148086547852
Epoch 2/5 -Avg_Loss: 13.9016 -Final_Loss: 13.154792785644531
Epoch 3/5 -Avg_Loss: 12.8180 -Final_Loss: 13.05328369140625
Epoch 4/5 -Avg_Loss: 11.9959 -Final_Loss: 11.561933517456055
Epoch 5/5 -Avg_Loss: 11.2605 -Final_Loss: 11.414704322814941
```



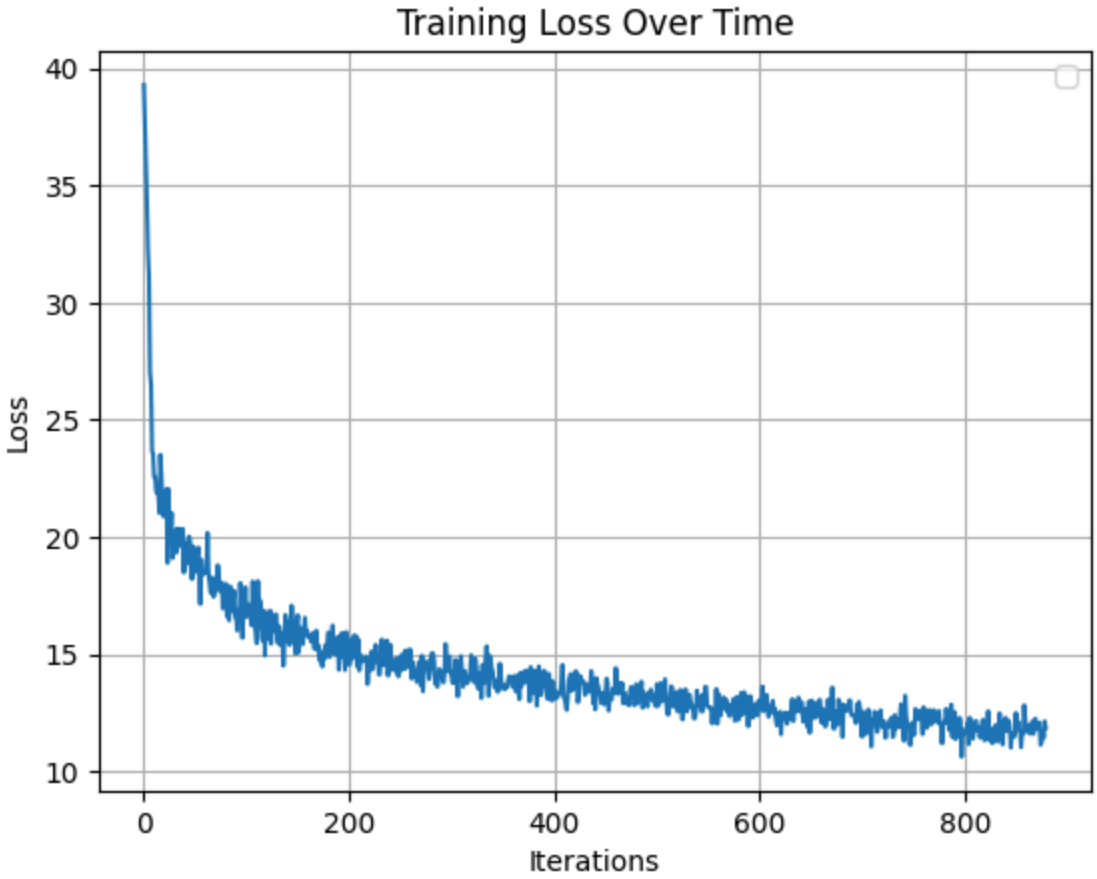

```
Embed Dim: 64, Hidden Size: 256, Num Layers: 1, Learning Rate: 0.005, Batch Size: 10
0, Epochs: 5, Final Loss: 11.414704322814941
Training model with embed_dim=64, hidden_size=256, lr=0.005, batch_size=100, epochs=
10
Epoch 1/10 -Avg_Loss: 17.0144 -Final_Loss: 14.333520889282227
Epoch 2/10 -Avg_Loss: 13.8013 -Final_Loss: 13.38319206237793
Epoch 3/10 -Avg_Loss: 12.7302 -Final_Loss: 12.764041900634766
Epoch 4/10 -Avg_Loss: 11.8821 -Final_Loss: 11.89345932006836
Epoch 5/10 -Avg_Loss: 11.1415 -Final_Loss: 11.087586402893066
Epoch 6/10 -Avg_Loss: 10.5249 -Final_Loss: 11.094442367553711
Epoch 7/10 -Avg_Loss: 10.0241 -Final_Loss: 10.513531684875488
Epoch 8/10 -Avg_Loss: 9.5701 -Final_Loss: 9.969864845275879
Epoch 9/10 -Avg_Loss: 9.1995 -Final_Loss: 9.739927291870117
Epoch 10/10 -Avg_Loss: 8.8995 -Final_Loss: 9.020211219787598
```



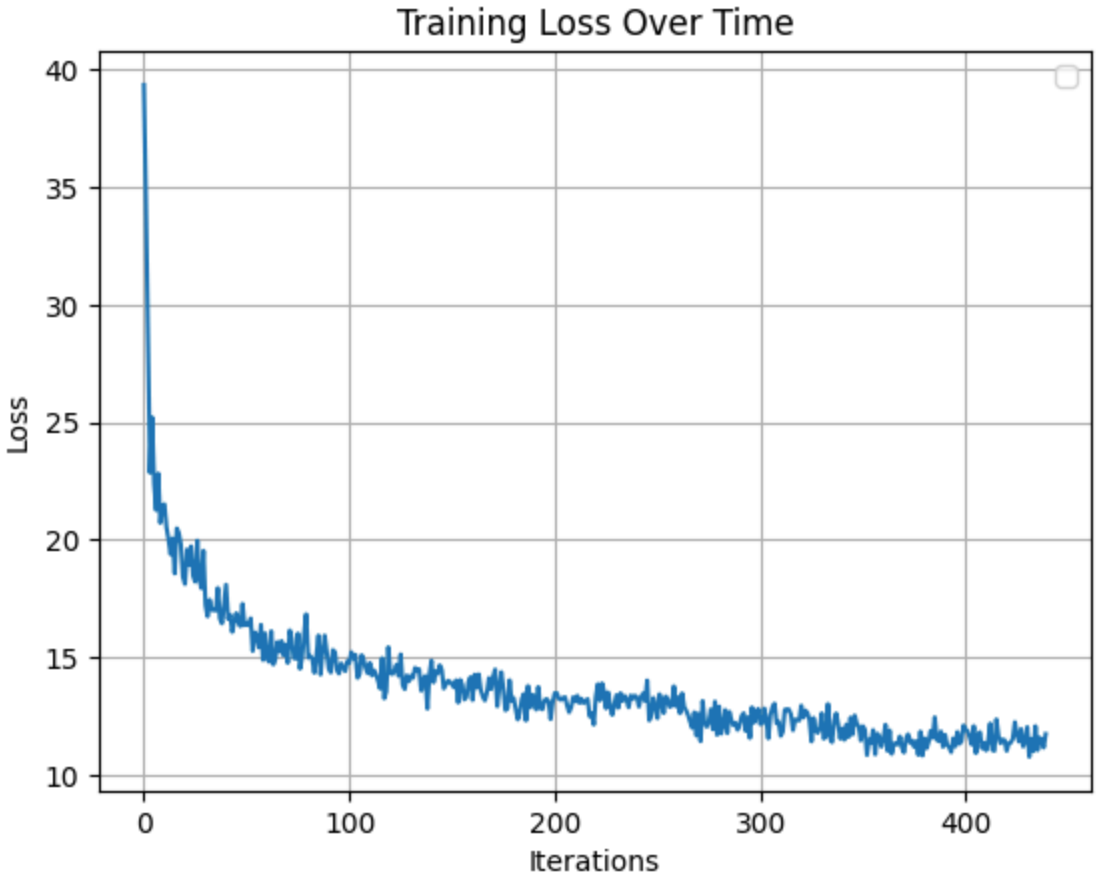

```
Embed Dim: 64, Hidden Size: 256, Num Layers: 1, Learning Rate: 0.005, Batch Size: 10
0, Epochs: 10, Final Loss: 9.020211219787598
Training model with embed_dim=64, hidden_size=256, lr=0.001, batch_size=100, epochs=
5
Epoch 1/5 -Avg_Loss: 20.5303 -Final_Loss: 16.58734893798828
Epoch 2/5 -Avg_Loss: 16.1123 -Final_Loss: 15.309237480163574
Epoch 3/5 -Avg_Loss: 14.9987 -Final_Loss: 14.34235954284668
Epoch 4/5 -Avg_Loss: 14.3136 -Final_Loss: 13.513270378112793
Epoch 5/5 -Avg_Loss: 13.7536 -Final_Loss: 13.543316841125488
```



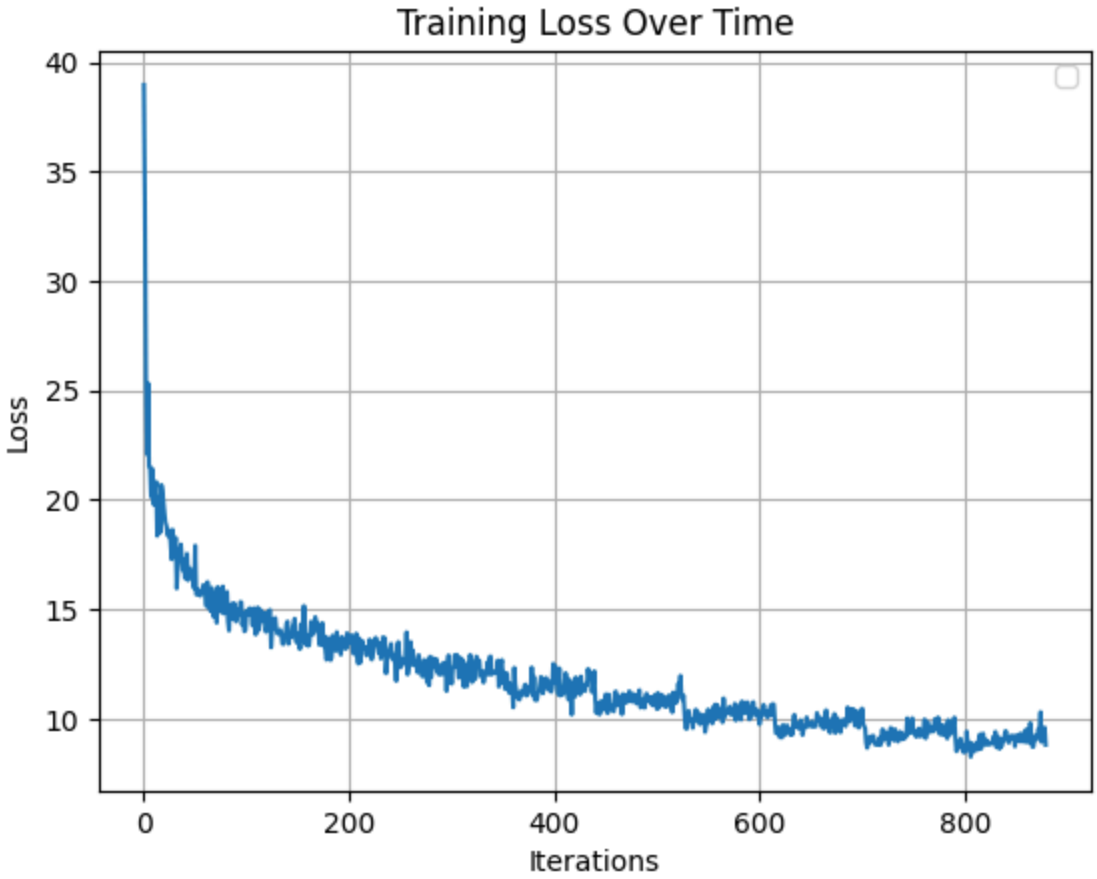

```
Embed Dim: 64, Hidden Size: 256, Num Layers: 1, Learning Rate: 0.001, Batch Size: 10
0, Epochs: 5, Final Loss: 13.543316841125488
Training model with embed_dim=64, hidden_size=256, lr=0.001, batch_size=100, epochs=
10
Epoch 1/10 -Avg_Loss: 20.6232 -Final_Loss: 17.385190963745117
Epoch 2/10 -Avg_Loss: 16.0896 -Final_Loss: 15.141366958618164
Epoch 3/10 -Avg_Loss: 14.8934 -Final_Loss: 14.403339385986328
Epoch 4/10 -Avg_Loss: 14.1893 -Final_Loss: 13.659252166748047
Epoch 5/10 -Avg_Loss: 13.6627 -Final_Loss: 13.839895248413086
Epoch 6/10 -Avg_Loss: 13.2096 -Final_Loss: 12.994089126586914
Epoch 7/10 -Avg_Loss: 12.7997 -Final_Loss: 12.663787841796875
Epoch 8/10 -Avg_Loss: 12.4492 -Final_Loss: 11.676470756530762
Epoch 9/10 -Avg_Loss: 12.1021 -Final_Loss: 11.363189697265625
Epoch 10/10 -Avg_Loss: 11.7750 -Final_Loss: 11.850336074829102
```



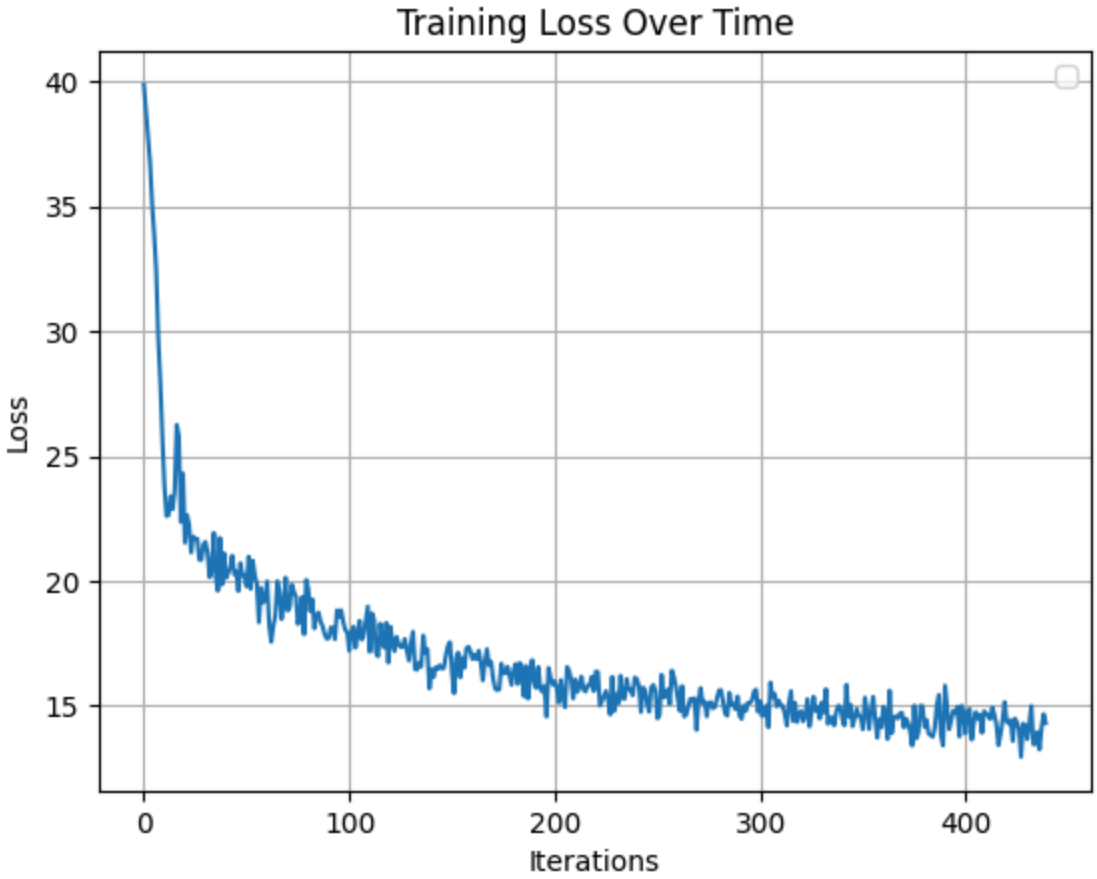

```
Embed Dim: 64, Hidden Size: 256, Num Layers: 1, Learning Rate: 0.001, Batch Size: 10
0, Epochs: 10, Final Loss: 11.850336074829102
Training model with embed_dim=32, hidden_size=256, lr=0.005, batch_size=100, epochs=
5
Epoch 1/5 -Avg_Loss: 17.8584 -Final_Loss: 15.213212966918945
Epoch 2/5 -Avg_Loss: 14.2044 -Final_Loss: 13.649389266967773
Epoch 3/5 -Avg_Loss: 13.0997 -Final_Loss: 12.873824119567871
Epoch 4/5 -Avg_Loss: 12.2302 -Final_Loss: 11.910443305969238
Epoch 5/5 -Avg_Loss: 11.4626 -Final_Loss: 11.729236602783203
```



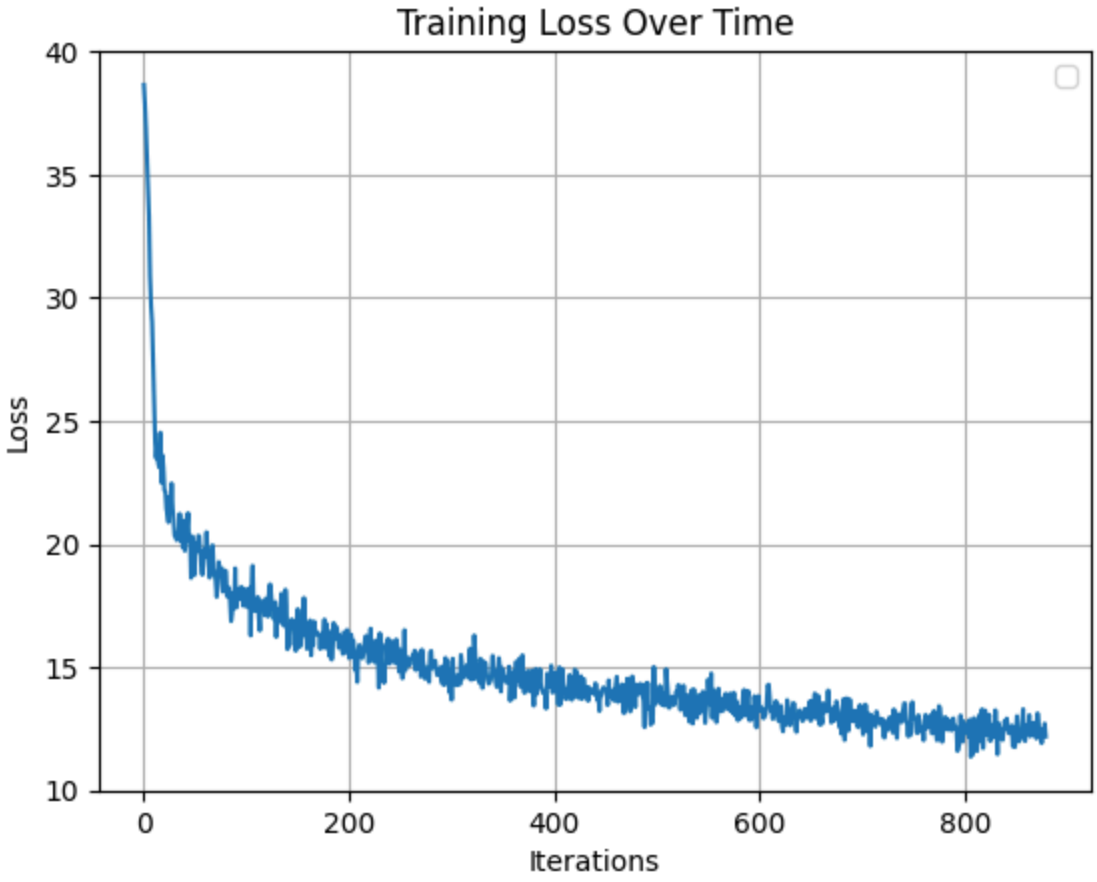

```
Embed Dim: 32, Hidden Size: 256, Num Layers: 1, Learning Rate: 0.005, Batch Size: 10
0, Epochs: 5, Final Loss: 11.729236602783203
Training model with embed_dim=32, hidden_size=256, lr=0.005, batch_size=100, epochs=
10
Epoch 1/10 -Avg_Loss: 17.7164 -Final_Loss: 14.996870040893555
Epoch 2/10 -Avg_Loss: 14.2290 -Final_Loss: 13.339072227478027
Epoch 3/10 -Avg_Loss: 13.1432 -Final_Loss: 12.832131385803223
Epoch 4/10 -Avg_Loss: 12.2766 -Final_Loss: 12.11475944519043
Epoch 5/10 -Avg_Loss: 11.4909 -Final_Loss: 12.18234634399414
Epoch 6/10 -Avg_Loss: 10.8372 -Final_Loss: 10.211494445800781
Epoch 7/10 -Avg_Loss: 10.2282 -Final_Loss: 9.912074089050293
Epoch 8/10 -Avg_Loss: 9.7571 -Final_Loss: 9.577287673950195
Epoch 9/10 -Avg_Loss: 9.3503 -Final_Loss: 9.120260238647461
Epoch 10/10 -Avg_Loss: 9.0125 -Final_Loss: 8.813283920288086
```



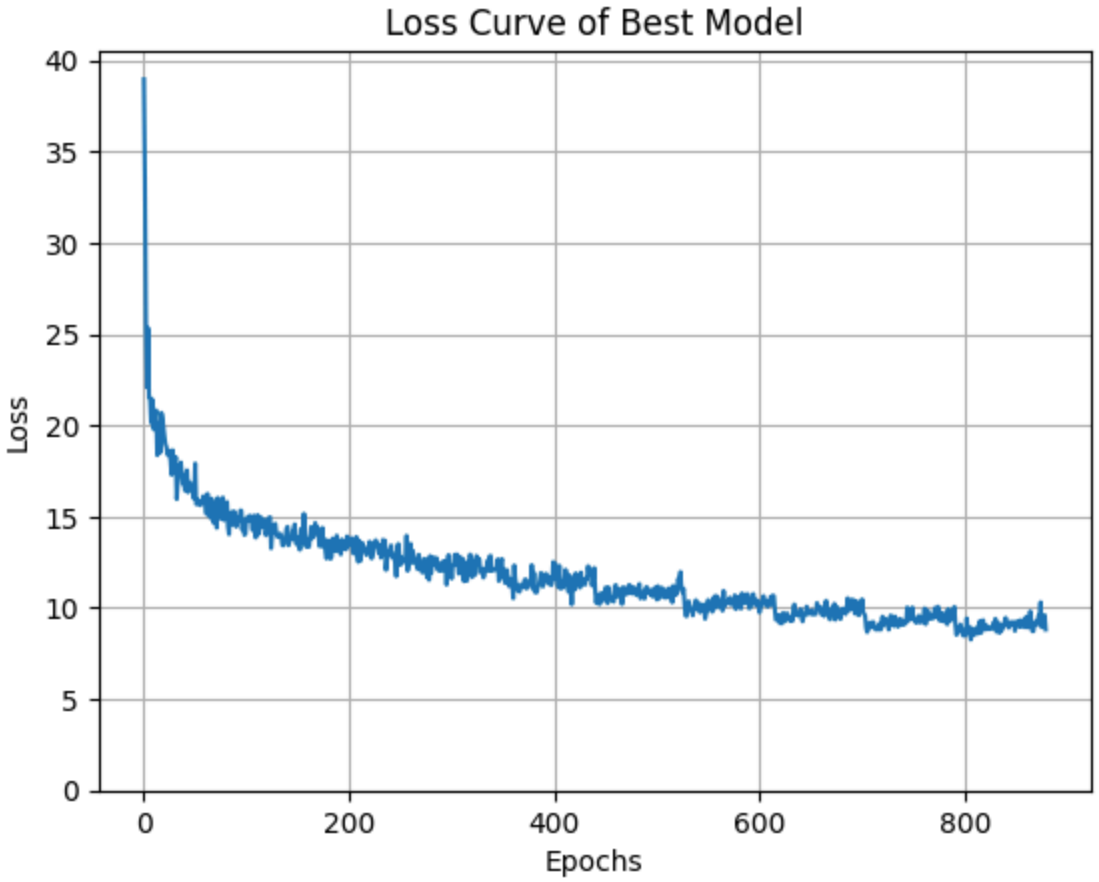

```
Embed Dim: 32, Hidden Size: 256, Num Layers: 1, Learning Rate: 0.005, Batch Size: 10
0, Epochs: 10, Final Loss: 8.813283920288086
Training model with embed_dim=32, hidden_size=256, lr=0.001, batch_size=100, epochs=
5
Epoch 1/5 -Avg_Loss: 21.9407 -Final_Loss: 18.17572593688965
Epoch 2/5 -Avg_Loss: 17.2397 -Final_Loss: 16.292091369628906
Epoch 3/5 -Avg_Loss: 15.7077 -Final_Loss: 14.568434715270996
Epoch 4/5 -Avg_Loss: 14.9164 -Final_Loss: 15.327245712280273
Epoch 5/5 -Avg_Loss: 14.3565 -Final_Loss: 14.30786418914795
```




```
Embed Dim: 32, Hidden Size: 256, Num Layers: 1, Learning Rate: 0.001, Batch Size: 10
0, Epochs: 5, Final Loss: 14.30786418914795
Training model with embed_dim=32, hidden_size=256, lr=0.001, batch_size=100, epochs=
10
Epoch 1/10 -Avg_Loss: 21.7271 -Final_Loss: 18.030946731567383
Epoch 2/10 -Avg_Loss: 17.1174 -Final_Loss: 16.376615524291992
Epoch 3/10 -Avg_Loss: 15.6574 -Final_Loss: 15.398537635803223
Epoch 4/10 -Avg_Loss: 14.8693 -Final_Loss: 14.65153694152832
Epoch 5/10 -Avg_Loss: 14.3065 -Final_Loss: 13.8923978805542
Epoch 6/10 -Avg_Loss: 13.8481 -Final_Loss: 13.231057167053223
Epoch 7/10 -Avg_Loss: 13.4557 -Final_Loss: 13.239089012145996
Epoch 8/10 -Avg_Loss: 13.0854 -Final_Loss: 13.486189842224121
Epoch 9/10 -Avg_Loss: 12.7448 -Final_Loss: 12.601700782775879
Epoch 10/10 -Avg_Loss: 12.4184 -Final_Loss: 12.184906005859375
```


Training Loss Over Time
Loss
Iterations
40
35
30
25
20
15
10
0
200
400
600
800

```
Embed Dim: 32, Hidden Size: 256, Num Layers: 1, Learning Rate: 0.001, Batch Size: 10
0, Epochs: 10, Final Loss: 12.184906005859375
Best Model Parameters: {'embed_dim': 32, 'hidden_size': 256, 'learning_rate': 0.005,
'batch_size': 100, 'epochs': 10, 'loss_log': [38.96080017089844, 33.57205581665039,
25.64749526977539, 22.089468002319336, 25.327634811401367, 21.54067611694336, 21.516
456604003906, 20.145774841308594, 21.425676345825195, 20.320098876953125, 19.7700443
26782227, 20.573049545288086, 20.82459259033203, 18.36408233642578, 18.6677837371826
17, 20.157468795776367, 18.52289581298828, 20.689424514770508, 20.434839248657227, 1
9.884920120239258, 19.410058975219727, 18.961767196655273, 18.813098907470703, 18.40
7121658325195, 18.355182647705078, 18.609987258911133, 18.058595657348633, 17.289039
611816406, 18.638582229614258, 17.920764923095703, 17.885221481323242, 18.2698383331
29883, 15.957801818847656, 16.77924156188965, 17.768808364868164, 17.32585906982422,
17.967592239379883, 17.356626510620117, 16.792030334472656, 17.30277442932129, 16.44
5337295532227, 16.405309677124023, 17.54009246826172, 16.334491729736328, 16.7501544
95239258, 16.87734603881836, 16.65876007080078, 16.54758644104004, 16.40148735046386
7, 15.971707344055176, 17.912710189819336, 15.68368911743164, 15.68800163269043, 15.
910550117492676, 15.744750022888184, 15.623478889465332, 15.735108375549316, 15.7909
46006774902, 16.123821258544922, 15.93862533569336, 15.653947830200195, 15.151145935
058594, 16.248096466064453, 15.114269256591797, 15.269270896911621, 14.9251594543457
03, 15.9848051071167, 14.961406707763672, 14.66502571105957, 15.724079132080078, 15.
798657417297363, 14.386923789978027, 16.029144287109375, 15.388466835021973, 15.4868
59321594238, 14.892138481140137, 14.922403335571289, 16.046354293823242, 14.80985260
0097656, 15.363201141357422, 15.510577201843262, 15.80196762084961, 14.4605941772460
94, 14.033873558044434, 15.216604232788086, 14.559120178222656, 15.251089096069336,
14.996870040893555, 15.27906608581543, 14.634573936462402, 14.443238258361816, 14.98
4552383422852, 14.694208145141602, 14.912610054016113, 15.340182304382324, 15.323337
55493164, 14.276324272155762, 14.785735130310059, 13.997274398803711, 14.05891132354
7363, 14.932300567626953, 14.91183090209961, 14.768056869506836, 15.053153038024902,
14.753393173217773, 14.772442817687988, 14.258552551269531, 15.052077293395996, 15.0
02053260803223, 13.852149963378906, 14.567296981811523, 15.088171005249023, 14.06256
0081481934, 14.358037948608398, 14.962102890014648, 14.722204208374023, 14.444437026
977539, 14.540526390075684, 14.850129127502441, 13.975345611572266, 14.3219957351684
57, 14.361157417297363, 14.938384056091309, 14.97163200378418, 13.267016410827637, 1
3.999703407287598, 14.525378227233887, 14.161166191101074, 14.608867645263672, 14.16
5888786315918, 13.9176025390625, 13.873390197753906, 13.997117042541504, 14.06486892
7001953, 13.908703804016113, 13.438309669494629, 13.413891792297363, 13.746844291687
012, 13.825963973999023, 14.339426040649414, 14.449289321899414, 13.427759170532227,
13.787405014038086, 13.765523910522461, 13.740716934204102, 14.222870826721191, 14.2
24373817443848, 14.576729774475098, 13.583089828491211, 13.481244087219238, 13.49838
7336730957, 13.890881538391113, 13.185657501220703, 14.322646141052246, 14.120777130
126953, 13.598623275756836, 15.156869888305664, 14.020835876464844, 13.3757362365722
66, 13.799018859863281, 13.337872505187988, 13.359932899475098, 13.876360893249512,
14.441414833068848, 13.826704978942871, 13.945865631103516, 13.945101737976074, 14.6
52493476867676, 14.139543533325195, 14.410453796386719, 14.235525131225586, 13.38496
3035583496, 13.393681526184082, 13.752086639404297, 14.381810188293457, 13.339072227
478027, 13.778410911560059, 13.091684341430664, 12.702081680297852, 12.9314889907836
91, 13.690044403076172, 13.170494079589844, 12.68366813659668, 12.814057350158691, 1
3.802599906921387, 13.214944839477539, 13.385293006896973, 13.110993385314941, 13.96
284294128418, 13.524731636047363, 13.165549278259277, 13.089363098144531, 12.9284706
11572266, 13.746770858764648, 13.346332550048828, 13.246369361877441, 13.61645412445
0684, 13.20334243774414, 13.917118072509766, 13.543160438537598, 13.459023475646973,
13.735814094543457, 13.827946662902832, 13.361363410949707, 12.99682331085205, 12.96
9054222106934, 12.99560260772705, 13.894050598144531, 12.976117134094238, 12.5234336
85302734, 13.677385330200195, 12.570935249328613, 13.073312759399414, 13.55701065063
4766, 13.20581340789795, 13.194360733032227, 13.29081916809082, 12.891809463500977,
13.251023292541504, 13.107234954833984, 13.625127792358398, 13.227755546569824, 12.7
```



From the above parameter tuning, we get the following configuration in the best model with least training loss:

embed_dim = 32

hidden_size = 256

learning_rate = 0.005

batch_size = 100

epochs = 5

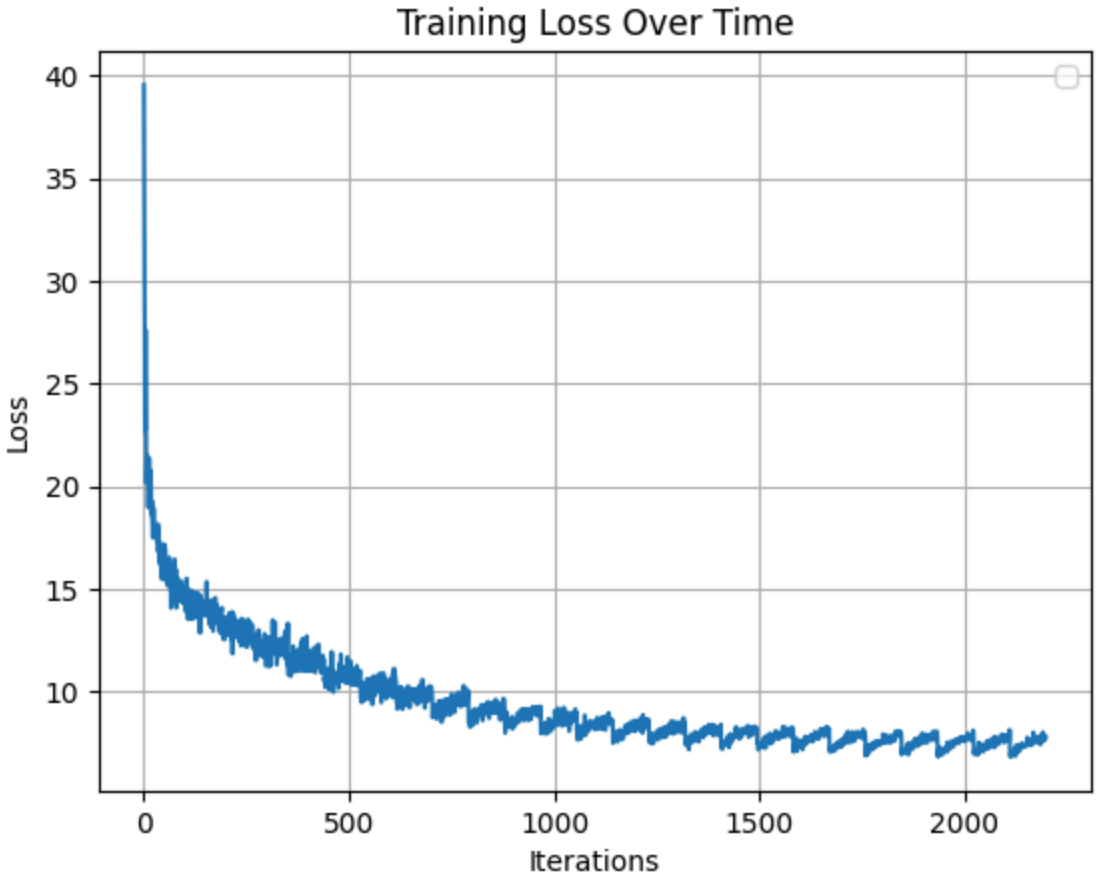
minimal loss = 8.81

Therefore to improve results, we now train the model with same parameters, but for 25 epochs.

In [12]:
```
ALPHABET_SIZE = 27
EMBED_DIM = 32
HIDDEN_SIZE = 256
NUM_LAYERS = 1
LEARNING_RATE = 0.005
EPOCHS = 25
BATCH_SIZE = 100
```

```
autocomplete_model = autocompleteModel(ALPHABET_SIZE, EMBED_DIM, HIDDEN_SIZE, NUM_L

autocomplete_model.trainModel(vocab, epochs=EPOCHS, batch_size=BATCH_SIZE,plot=True
```

```
Epoch 1/25 -Avg_Loss: 17.8170 -Final_Loss: 14.847609519958496
Epoch 2/25 -Avg_Loss: 14.2006 -Final_Loss: 13.508740425109863
Epoch 3/25 -Avg_Loss: 13.1213 -Final_Loss: 13.284475326538086
Epoch 4/25 -Avg_Loss: 12.2476 -Final_Loss: 13.330739974975586
Epoch 5/25 -Avg_Loss: 11.5008 -Final_Loss: 11.41234016418457
Epoch 6/25 -Avg_Loss: 10.8103 -Final_Loss: 11.033064842224121
Epoch 7/25 -Avg_Loss: 10.2616 -Final_Loss: 10.442239761352539
Epoch 8/25 -Avg_Loss: 9.7882 -Final_Loss: 9.723847389221191
Epoch 9/25 -Avg_Loss: 9.3792 -Final_Loss: 9.989803314208984
Epoch 10/25 -Avg_Loss: 9.0412 -Final_Loss: 9.19263744354248
Epoch 11/25 -Avg_Loss: 8.7752 -Final_Loss: 9.283089637756348
Epoch 12/25 -Avg_Loss: 8.5489 -Final_Loss: 8.667895317077637
Epoch 13/25 -Avg_Loss: 8.3618 -Final_Loss: 8.612689018249512
Epoch 14/25 -Avg_Loss: 8.2018 -Final_Loss: 8.392465591430664
Epoch 15/25 -Avg_Loss: 8.0703 -Final_Loss: 8.460643768310547
Epoch 16/25 -Avg_Loss: 7.9489 -Final_Loss: 8.409553527832031
Epoch 17/25 -Avg_Loss: 7.8784 -Final_Loss: 8.170496940612793
Epoch 18/25 -Avg_Loss: 7.7883 -Final_Loss: 8.222044944763184
Epoch 19/25 -Avg_Loss: 7.7259 -Final_Loss: 8.191271781921387
Epoch 20/25 -Avg_Loss: 7.6517 -Final_Loss: 7.796881198883057
Epoch 21/25 -Avg_Loss: 7.5921 -Final_Loss: 7.889554023742676
Epoch 22/25 -Avg_Loss: 7.5567 -Final_Loss: 7.755916118621826
Epoch 23/25 -Avg_Loss: 7.5081 -Final_Loss: 8.037590980529785
Epoch 24/25 -Avg_Loss: 7.4891 -Final_Loss: 8.145620346069336
Epoch 25/25 -Avg_Loss: 7.4476 -Final_Loss: 7.769320011138916
```



```
 9.017511367797852,
 9.216476440429688,
 9.255054473876953,
 9.084470748901367,
 9.22561264038086,
 9.159872055053711,
 9.057975769042969,
 8.83911418914795,
 9.280338287353516,
 9.156360626220703,
 9.089595794677734,
 9.149633407592773,
 8.85910701751709,
 9.062335968017578,
 8.785390853881836,
 9.283089637756348,
 8.013225555419922,
 7.9594621658325195,
 8.203685760498047,
 8.220438957214355,
 8.061783790588379,
 8.18976879119873,
 8.225933074951172,
 8.123786926269531,
 8.034914016723633,
 8.127964973449707,
 8.492942810058594,
 8.228314399719238,
 8.350793838500977,
 7.946028232574463,
 8.11330795288086,
 8.301894187927246,
 8.07046890258789,
 8.242198944091797,
 8.32048225402832,
 8.690378189086914,
 8.443327903747559,
 8.508345603942871,
 8.048816680908203,
 8.560279846191406,
 8.507519721984863,
 8.499460220336914,
 8.506181716918945,
 8.540411949157715,
 8.459811210632324,
 8.602640151977539,
 8.221206665039062,
 8.275226593017578,
 ...]
```

In [20]:
```
test_samples = ["univ", "math", "neur", "engin"]
outputs = autocomplete_model.autocomplete(test_samples)

# Print the results
for sample, output in zip(test_samples, outputs):
    print(f"Input: '{sample}' -> Output: '{output}'")
```

```
Input: 'univ' -> Output: 'universe-----'
Input: 'math' -> Output: 'mathematical-'
Input: 'neur' -> Output: 'neural-------'
Input: 'engin' -> Output: 'engineering--'
```

In [22]:
```
test_words = ['dict', 'stat', 'fic', 'writi']
outputs = autocomplete_model.autocomplete(test_words)
for sample, output in zip(test_words, outputs):
    print(f"Input: '{sample}' -> Output: '{output}'")
```

```
Input: 'dict' -> Output: 'dictionary---'
Input: 'stat' -> Output: 'station------'
Input: 'fic' -> Output: 'ficst--------'
Input: 'writi' -> Output: 'writings-----'
```

In [23]:
```
test_words = ['pl', 'diz', 'netw', 'min']
outputs = autocomplete_model.autocomplete(test_words)
for sample, output in zip(test_words, outputs):
    print(f"Input: '{sample}' -> Output: '{output}'")
```

```
Input: 'pl' -> Output: 'plates-------'
Input: 'diz' -> Output: 'dizos--------'
Input: 'netw' -> Output: 'network------'
Input: 'min' -> Output: 'mineral------'
```

In [24]:
```
test_words = ['hig', 'capa', 'bott', 'wat']
outputs = autocomplete_model.autocomplete(test_words)
for sample, output in zip(test_words, outputs):
    print(f"Input: '{sample}' -> Output: '{output}'")
```

```
Input: 'hig' -> Output: 'highlights---'
Input: 'capa' -> Output: 'capabilities-'
Input: 'bott' -> Output: 'bottles------'
Input: 'wat' -> Output: 'watched------'
```

In [29]:
```
test_words = ['anim', 'monk', 'tre', 'fores']
outputs = autocomplete_model.autocomplete(test_words)
for sample, output in zip(test_words, outputs):
    print(f"Input: '{sample}' -> Output: '{output}'")
```

```
Input: 'anim' -> Output: 'animals------'
Input: 'monk' -> Output: 'monkey-------'
Input: 'tre' -> Output: 'trench-------'
Input: 'fores' -> Output: 'forest-------'
```

Above word autocompletions depicts the model has trained well with time. Befor well trained, model predicted somewhat bad. univoigizer--,mathadjapen--,neurobitl---,engina- but our best model gave out 'universe-----','mathematical-','neural-------','engineering--' which seems to have kearnt semantic relations well. These words are much related to each other than previous results.

Next examples also proves it. animals, monkey, trench, forest are well related to each other. We have trained to receive commoun suffixes like watched -ed, capabilities-ies, writings-ings as well. Overall the model has trained quite OK, but there's still chance for improvement.