Recurrent Neural Networks

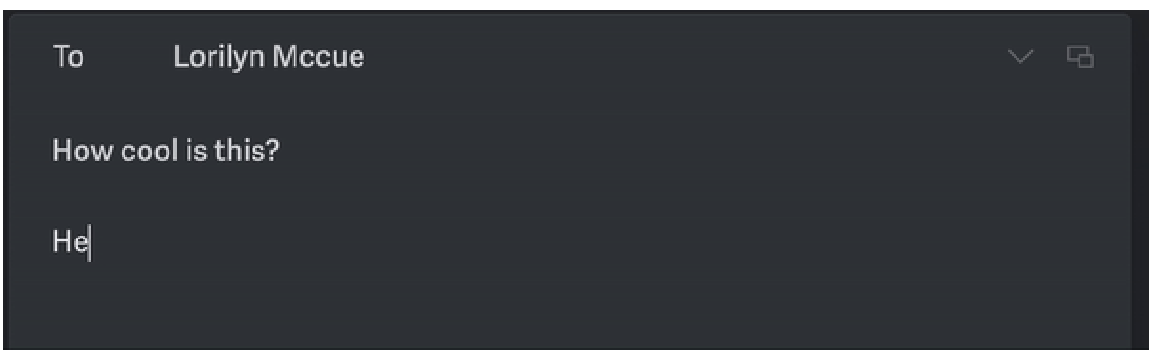
Inclass Project 3 - MA4144

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

torch.manual_seed(42)
np.random.seed(42)
```

Section 1: Preparing the vocabulary

In [2]:
```
WORD_SIZE = 13
```

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]:
```
with open("wordlist.txt",'r') as file:
    words= [line.strip().lower() for line in file]

filter_word = [word for word in words if 3 < len(word) <= WORD_SIZE]

vocab = [word.ljust(WORD_SIZE,'_') for word in filter_word]
```

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?

By taking words longer than 3 letters, we can remove function words such as "the" and "and," which do not carry significant semantic weight, helping the model avoid overfitting to them. Additionally, by avoiding words longer than the defined WORD_SIZE, we reduce the model's complexity and enforce a fixed word size, which simplifies computation and makes the process more efficient.

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

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

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.

* Yes, this method assigns a misleading relationship between letters by giving them numbers based on their alphabetical order. By doing this, we incorrectly suggest that letters close to each other have a stronger relationship than those that are farther apart, which is not true in the English alphabet. Letters that are far apart may still have a strong relationship, but this method does not capture that.

* In the code, the purpose of the embedding layer is to map the input characters to a vector space that can capture more meaningful relationships and richer information. Even though we initially encode the letters in alphabetical order, the embedding layer learns to represent them in a way that captures the true relationships and patterns between the characters, overcoming the limitations of the alphabetical encoding.

* The purpose of using an embedding layer is to capture the semantic relationships between letters and encode them more meaningfully. Unlike one-hot encoding, which is sparse and high-dimensional, embeddings reduce dimensionality and represent the letters in a dense vector space. These embeddings allow the model to learn how letters are related to each other, which helps the model generalize better and ultimately improves its performance.

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.embed=torch.nn.Embedding(alphabet_size,embed_dim)
        self.lstm=torch.nn.LSTMCell(embed_dim,hidden_size)
        self.fc=torch.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)
        #2. Feed the embedded output to the LSTM cell
        #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

        embedd=self.embed(character)
        (hidden_state,cell_state)=self.lstm(embedd,(hidden_state,cell_state))
        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,l_rate=LEARNING_RATE,p

        #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=l_rate)

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

        for e in range(epochs):

            random.shuffle(vocab) #Shuffle the vocab list the start of each epoch
            num_iter=len(vocab)//batch_size

            for i in range(num_iter):

                #Set the loss to zero, initialize the optimizer with zero_grad at t
                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 eac
                hidden_state,cell_state=self.initial_state()

                #Convert the word into a tensor of number and create input and
                word_to_tensor=word_to_numlist(word)
                #Input will be the first WORD_SIZE - 1 charcters and target is
                inputs=word_to_tensor[:-1]
                targets=word_to_tensor[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 comp
                    output,hidden_state,cell_state=self.forward(inputs[c].unsqu
                    loss+=torch.nn.functional.cross_entropy(output,targets[c].v


            #Compute the average loss per word in the batch and perform backpro
            loss=loss/batch_size
            loss.backward()

            #Update model parameters using the optimizer
            optimizer.step()

            #Update the loss_log
            loss_log.append(loss.item())
            #print(i)
            #print("**************************************************")
        if plot:
            print("Epoch: ", e," loss : ",loss)

    #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.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.

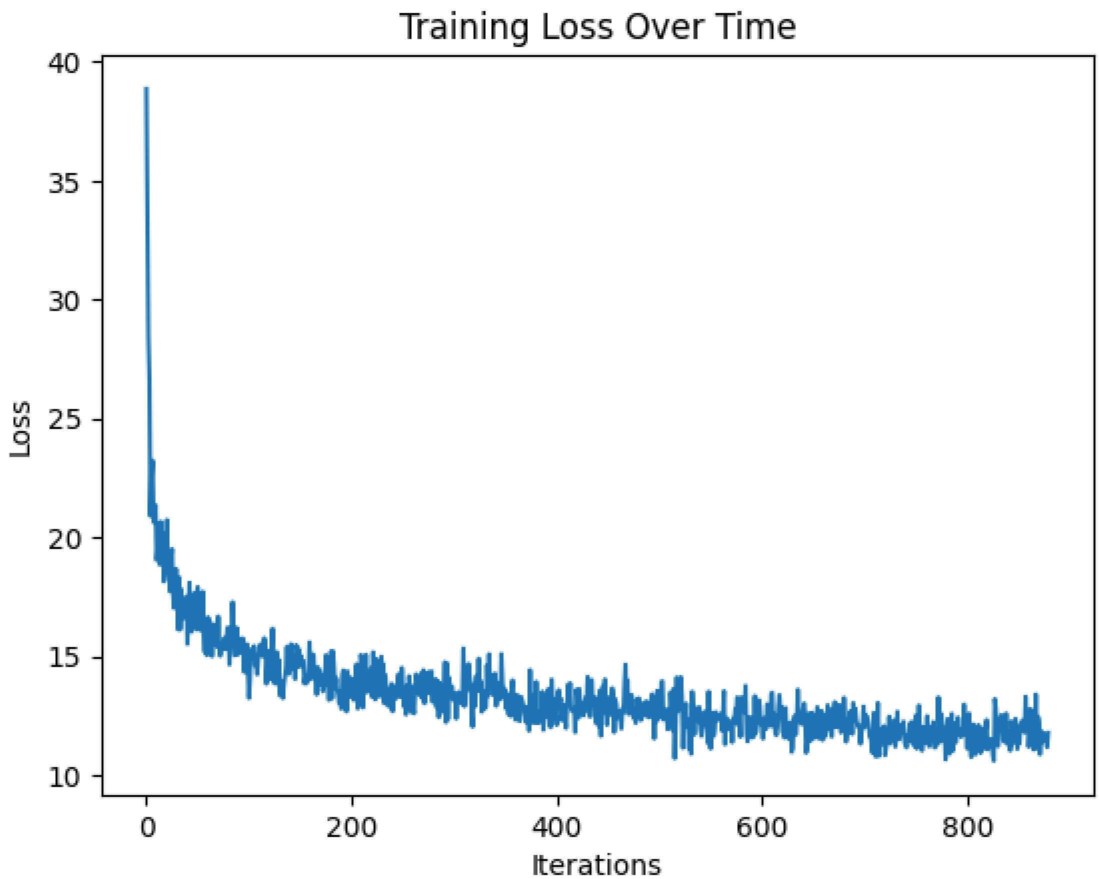
Comment on the results. Is it successful? Do you see familiar substrings in the generated tesxt such as "tion", "ing", "able" etc. What are your suggestions to improve the model?

In the bottom

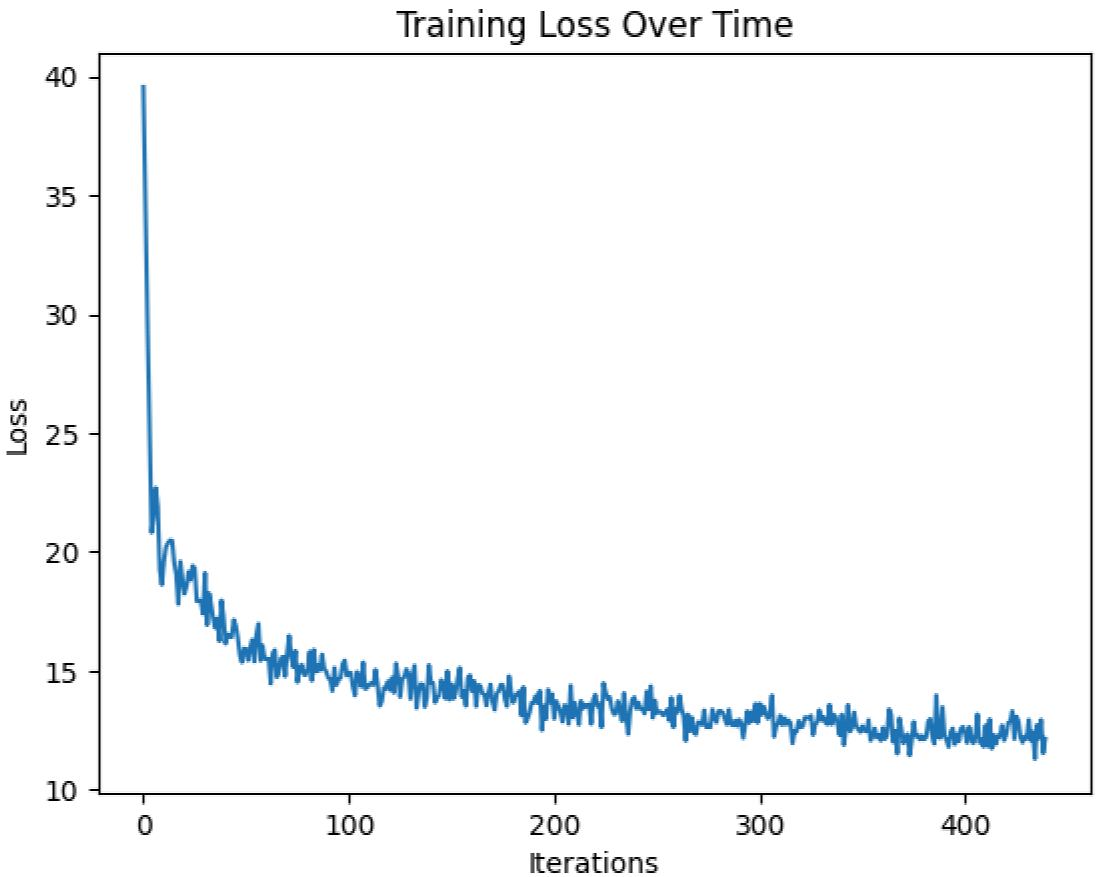
In [16]:
```
from itertools import product
import matplotlib.pyplot as plt

# Define the hyperparameter options
embed_dims = [64,100]
hidden_sizes = [128,256]
learning_rates = [0.005]
batch_sizes = [50,100]
epoch_counts = [5]

# 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},
```

```
    # 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, l_rate


    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,
        '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,
            'loss_log': loss_log
        }

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

# 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.show()
```

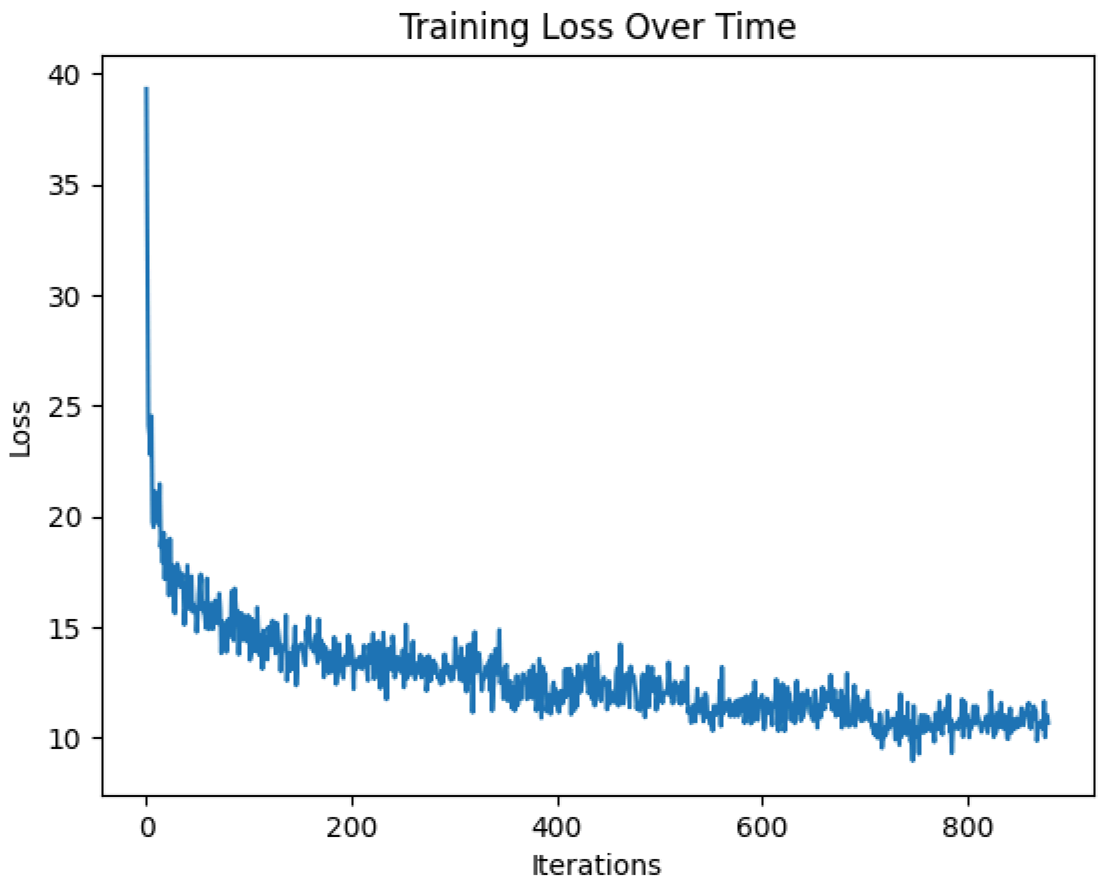
```
Epoch:  0  loss :  tensor(15.0543, grad_fn=<DivBackward0>)
Epoch:  1  loss :  tensor(12.7870, grad_fn=<DivBackward0>)
Epoch:  2  loss :  tensor(12.0648, grad_fn=<DivBackward0>)
Epoch:  3  loss :  tensor(12.4206, grad_fn=<DivBackward0>)
Epoch:  4  loss :  tensor(11.7884, grad_fn=<DivBackward0>)
```



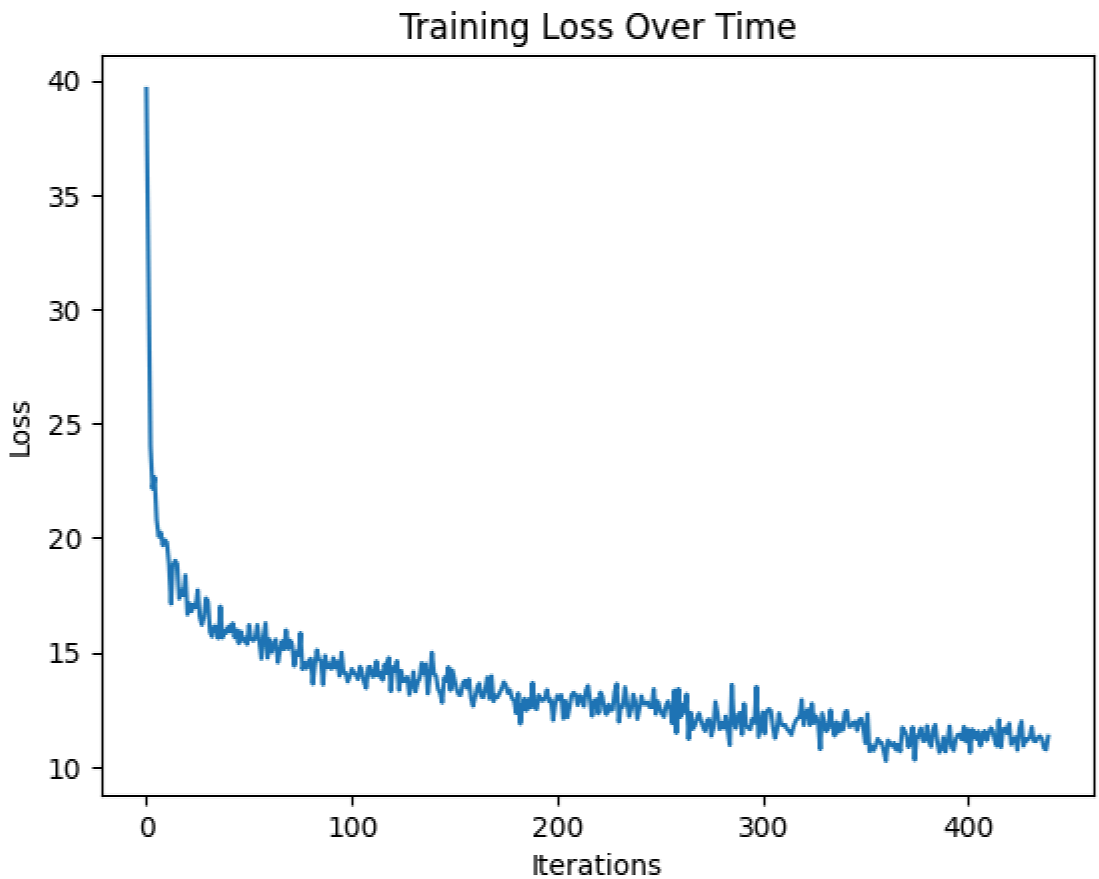

```
Embed Dim: 64, Hidden Size: 128, Num Layers: 1, Learning Rate: 0.005, Batch Size: 5
0, Epochs: 5, Final Loss: 11.788394927978516
Epoch:  0  loss :  tensor(15.6893, grad_fn=<DivBackward0>)
Epoch:  1  loss :  tensor(14.3937, grad_fn=<DivBackward0>)
Epoch:  2  loss :  tensor(13.0943, grad_fn=<DivBackward0>)
Epoch:  3  loss :  tensor(12.4881, grad_fn=<DivBackward0>)
Epoch:  4  loss :  tensor(12.1524, grad_fn=<DivBackward0>)
```



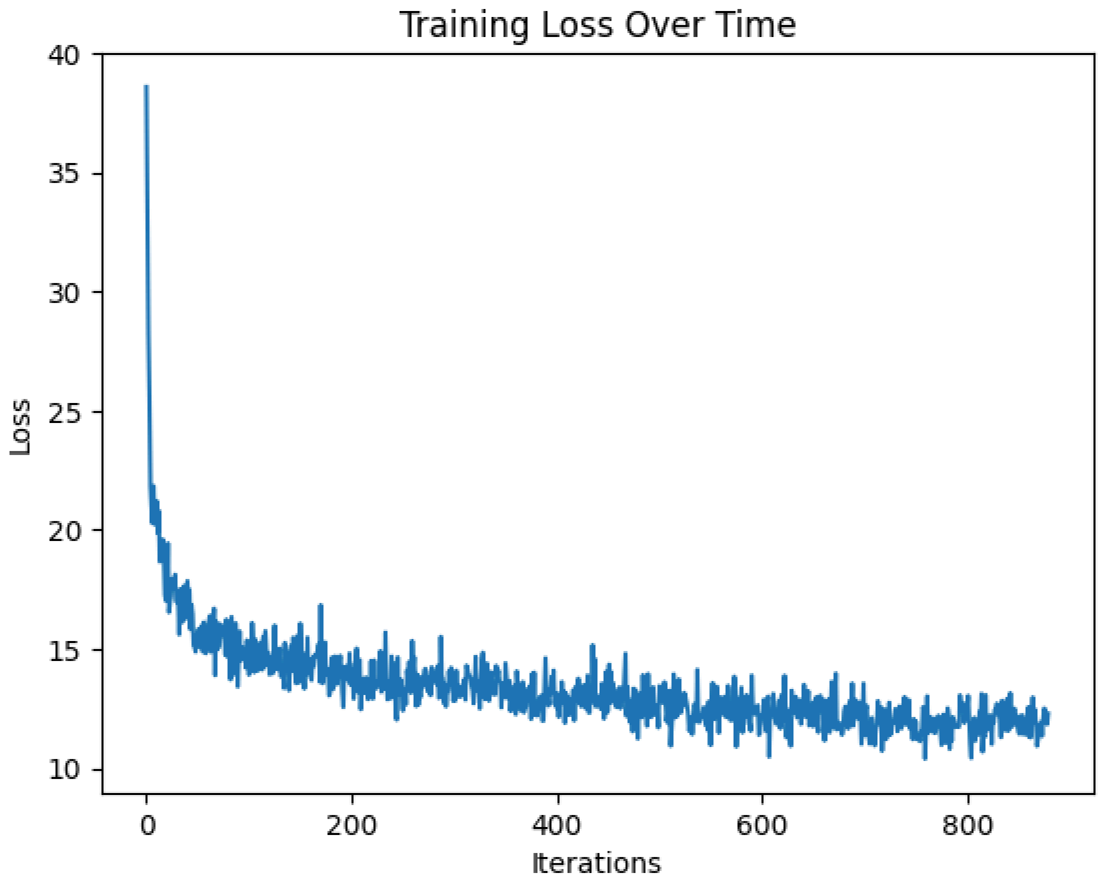

```
Embed Dim: 64, Hidden Size: 128, Num Layers: 1, Learning Rate: 0.005, Batch Size: 10
0, Epochs: 5, Final Loss: 12.152417182922363
Epoch:  0  loss :  tensor(14.0841, grad_fn=<DivBackward0>)
Epoch:  1  loss :  tensor(13.2859, grad_fn=<DivBackward0>)
Epoch:  2  loss :  tensor(13.1986, grad_fn=<DivBackward0>)
Epoch:  3  loss :  tensor(12.0934, grad_fn=<DivBackward0>)
Epoch:  4  loss :  tensor(10.6750, grad_fn=<DivBackward0>)
```



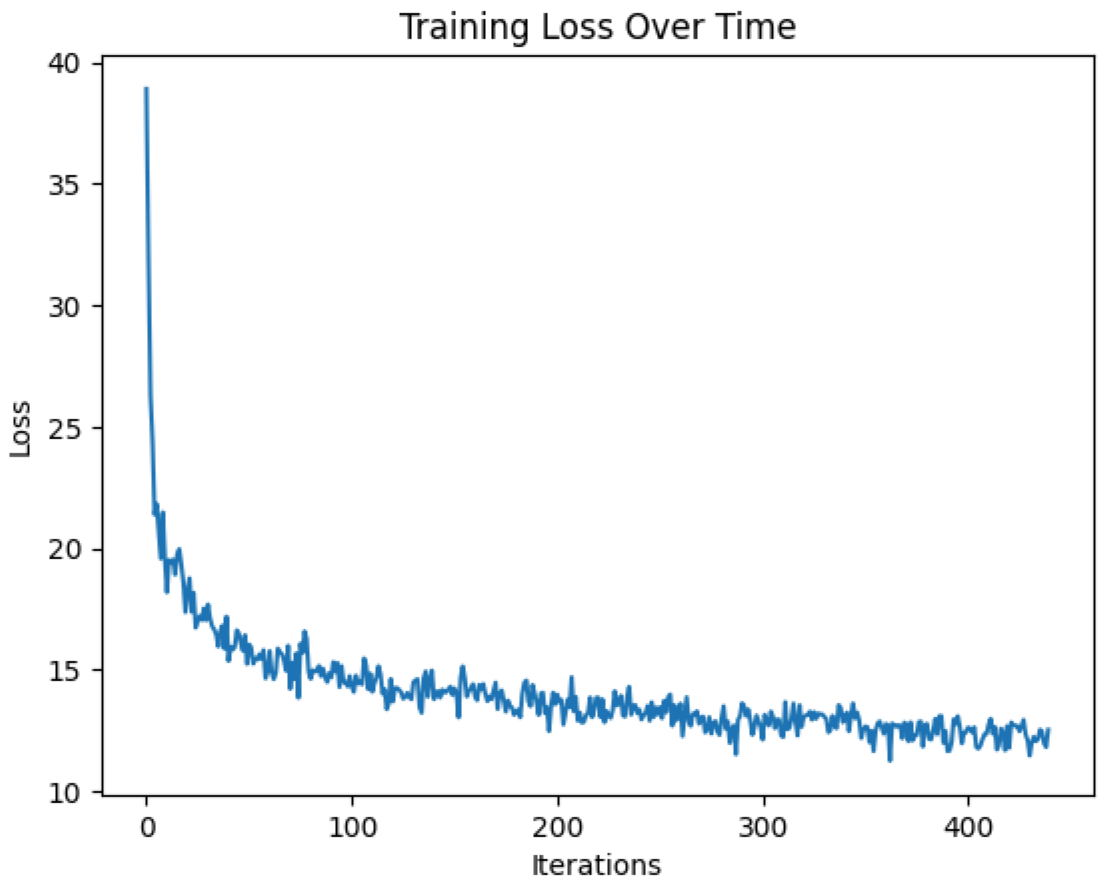

```
Embed Dim: 64, Hidden Size: 256, Num Layers: 1, Learning Rate: 0.005, Batch Size: 5
0, Epochs: 5, Final Loss: 10.675048828125
Epoch:  0  loss :  tensor(14.8483, grad_fn=<DivBackward0>)
Epoch:  1  loss :  tensor(13.5572, grad_fn=<DivBackward0>)
Epoch:  2  loss :  tensor(13.1660, grad_fn=<DivBackward0>)
Epoch:  3  loss :  tensor(12.3078, grad_fn=<DivBackward0>)
Epoch:  4  loss :  tensor(11.3172, grad_fn=<DivBackward0>)
```



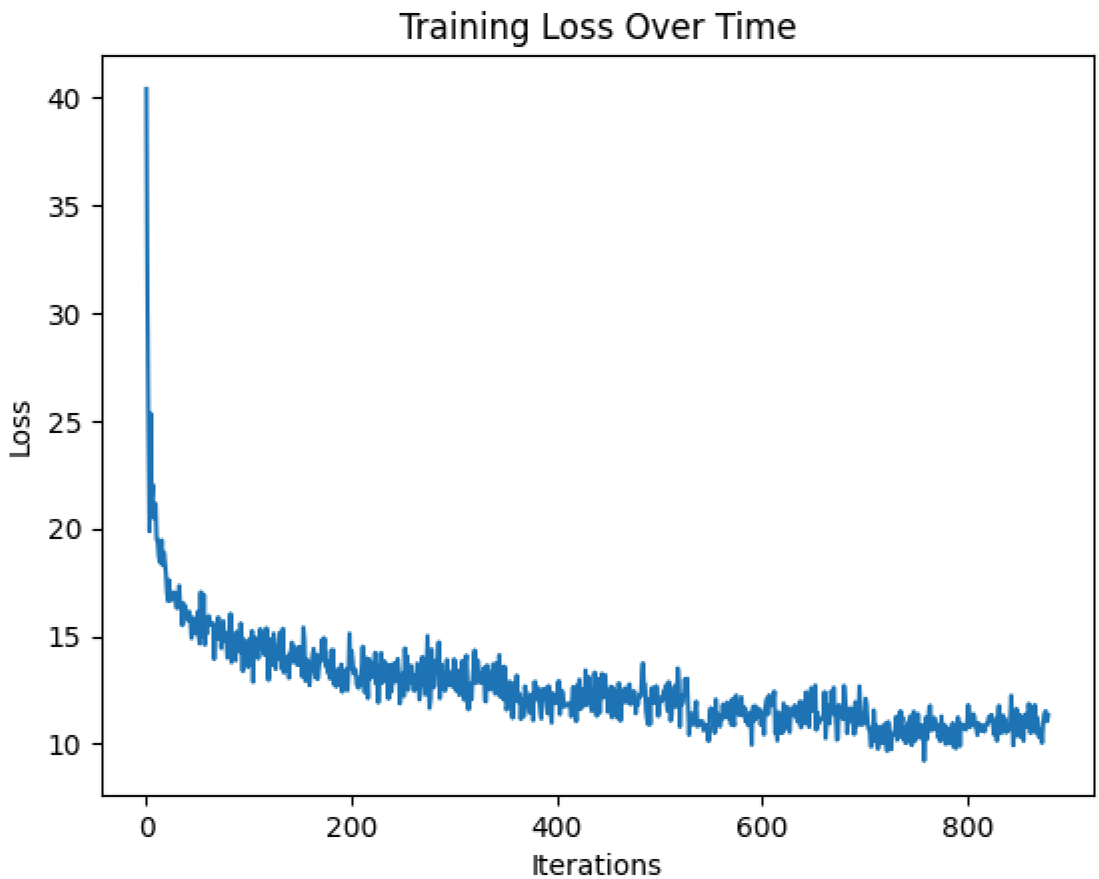

```
Embed Dim: 64, Hidden Size: 256, Num Layers: 1, Learning Rate: 0.005, Batch Size: 10
0, Epochs: 5, Final Loss: 11.317233085632324
Epoch:  0  loss :  tensor(14.3415, grad_fn=<DivBackward0>)
Epoch:  1  loss :  tensor(13.8530, grad_fn=<DivBackward0>)
Epoch:  2  loss :  tensor(12.6375, grad_fn=<DivBackward0>)
Epoch:  3  loss :  tensor(11.7564, grad_fn=<DivBackward0>)
Epoch:  4  loss :  tensor(12.2873, grad_fn=<DivBackward0>)
```



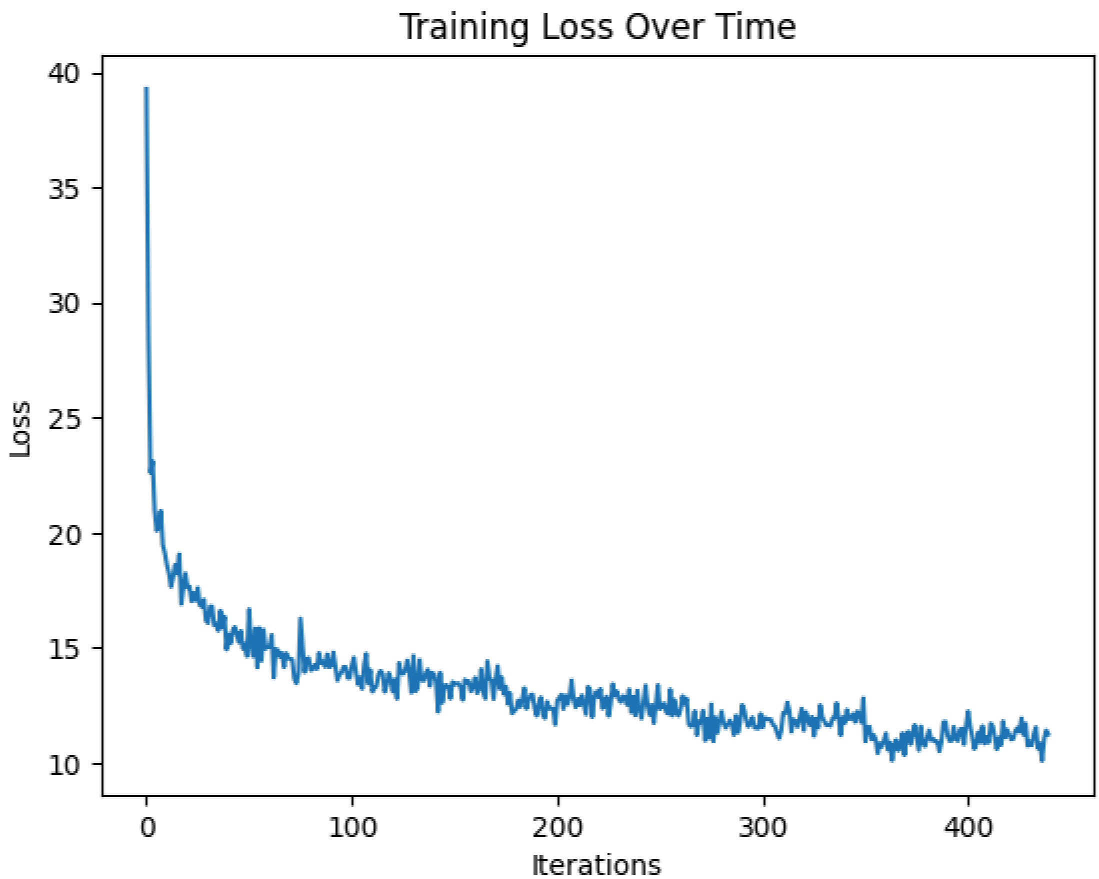

```
Embed Dim: 100, Hidden Size: 128, Num Layers: 1, Learning Rate: 0.005, Batch Size: 5
0, Epochs: 5, Final Loss: 12.28734016418457
Epoch:  0  loss :  tensor(14.6696, grad_fn=<DivBackward0>)
Epoch:  1  loss :  tensor(13.2523, grad_fn=<DivBackward0>)
Epoch:  2  loss :  tensor(13.8560, grad_fn=<DivBackward0>)
Epoch:  3  loss :  tensor(12.6181, grad_fn=<DivBackward0>)
Epoch:  4  loss :  tensor(12.5115, grad_fn=<DivBackward0>)
```



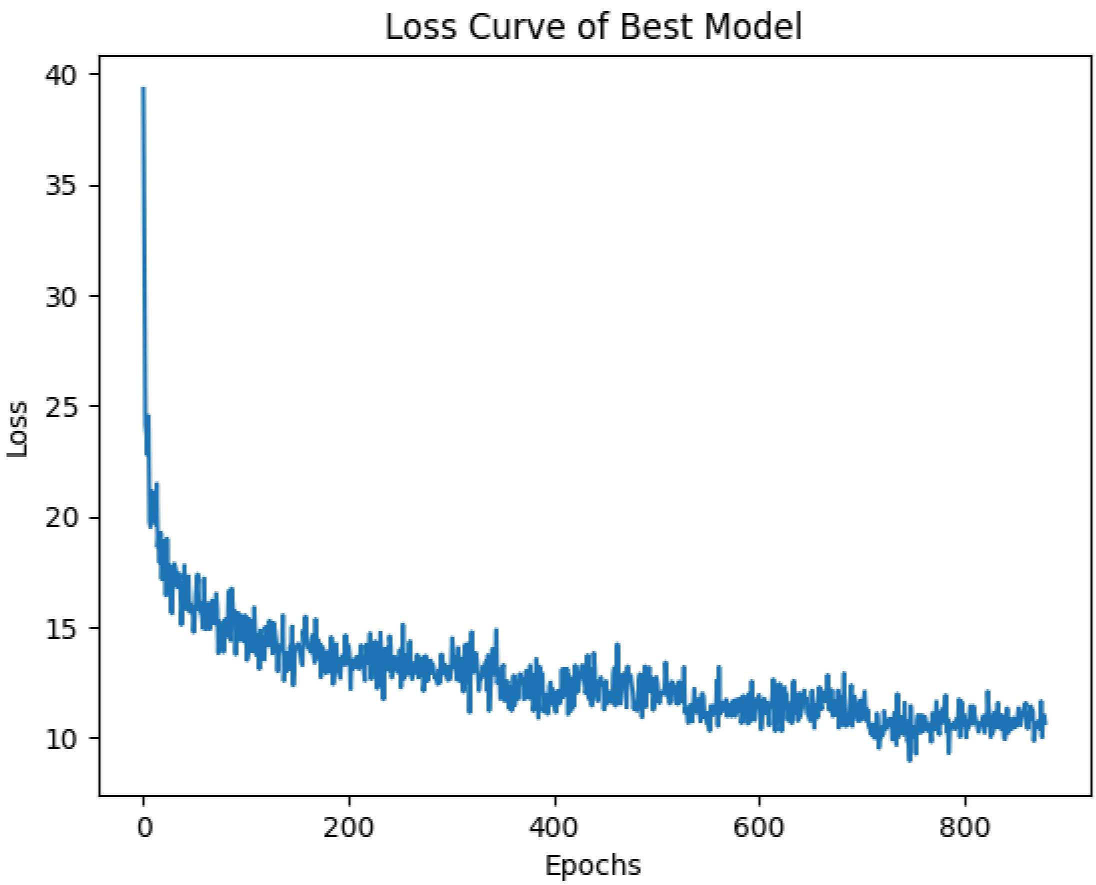

```
Embed Dim: 100, Hidden Size: 128, Num Layers: 1, Learning Rate: 0.005, Batch Size: 1
00, Epochs: 5, Final Loss: 12.511545181274414
Epoch:  0  loss :  tensor(13.5959, grad_fn=<DivBackward0>)
Epoch:  1  loss :  tensor(11.5773, grad_fn=<DivBackward0>)
Epoch:  2  loss :  tensor(13.0380, grad_fn=<DivBackward0>)
Epoch:  3  loss :  tensor(10.9580, grad_fn=<DivBackward0>)
Epoch:  4  loss :  tensor(11.3345, grad_fn=<DivBackward0>)
```




```
Embed Dim: 100, Hidden Size: 256, Num Layers: 1, Learning Rate: 0.005, Batch Size: 5
0, Epochs: 5, Final Loss: 11.334527969360352
Epoch:  0  loss :  tensor(14.1726, grad_fn=<DivBackward0>)
Epoch:  1  loss :  tensor(13.3152, grad_fn=<DivBackward0>)
Epoch:  2  loss :  tensor(12.8010, grad_fn=<DivBackward0>)
Epoch:  3  loss :  tensor(11.5129, grad_fn=<DivBackward0>)
Epoch:  4  loss :  tensor(11.2413, grad_fn=<DivBackward0>)
```


Training Loss Over Time
Loss
Iterations
40
35
30
25
20
15
10
0
100
200
300
400

```
Embed Dim: 100, Hidden Size: 256, Num Layers: 1, Learning Rate: 0.005, Batch Size: 1
00, Epochs: 5, Final Loss: 11.241314888000488
Best Model Parameters: {'embed_dim': 64, 'hidden_size': 256, 'learning_rate': 0.005,
'batch_size': 50, 'epochs': 5, 'loss_log': [39.324432373046875, 31.640846252441406,
24.14260482788086, 23.668184280395508, 22.771163940429688, 24.537456512451172, 19.76
0305404663086, 19.516942977905273, 21.157737731933594, 20.423786163330078, 20.318761
825561523, 20.192180633544922, 19.65702247619629, 21.472291946411133, 18.61034965515
1367, 19.094568252563477, 17.904203414916992, 19.263349533081055, 17.16123771667480
5, 18.776988983154297, 18.8835391998291, 18.783390045166016, 16.428409576416016, 18.
999191284179688, 16.981325149536133, 17.814777374267578, 17.309703826904297, 15.7104
43496704102, 15.611804008483887, 17.573457717895508, 17.867019653320312, 17.72677993
774414, 17.48131561279297, 16.81084442138672, 17.134302139282227, 17.41165733337402
3, 16.714357376098633, 15.104217529296875, 15.779736518859863, 16.02642059326172, 1
7.789464950561523, 15.828569412231445, 16.397520065307617, 16.430543899536133, 17.29
7039031982422, 15.687590599060059, 16.030092239379883, 15.727794647216797, 15.892333
030700684, 14.77381706237793, 15.382867813110352, 16.459096908569336, 17.02060890197
754, 17.385530471801758, 15.806462287902832, 16.11975860595703, 15.667062759399414,
15.527347564697266, 14.95126724243164, 17.200986862182617, 15.858564376831055, 14.89
8526191711426, 15.848898887634277, 16.05945587158203, 15.923052787780762, 14.9064941
40625, 15.75634765625, 16.205474853515625, 15.248238563537598, 15.619140625, 15.1600
51345825195, 16.512866973876953, 15.824397087097168, 13.819727897644043, 15.26934528
35083, 14.6136474609375, 14.3247709274292, 15.037196159362793, 13.869865417480469, 1
4.744215965270996, 15.354808807373047, 14.872422218322754, 14.607158660888672, 16.61
3510131835938, 14.554363250732422, 14.396092414855957, 16.745088577270508, 16.059679
03137207, 15.412327766418457, 15.702729225158691, 13.766613960266113, 14.94363784790
039, 15.626014709472656, 15.618633270263672, 14.406280517578125, 15.508336067199707,
14.652176856994629, 15.296704292297363, 14.344243049621582, 15.536332130432129, 15.4
59927558898926, 13.509320259094238, 13.831277847290039, 15.368734359741211, 13.90322
1130371094, 14.413800239562988, 14.086888313293457, 13.905728340148926, 15.888602256
774902, 14.69087028503418, 13.737476348876953, 14.012969970703125, 14.5997285842895
5, 13.104059219360352, 14.849822044372559, 14.086870193481445, 14.520500183105469, 1
3.662188529968262, 13.526830673217773, 15.014015197753906, 14.632126808166504, 14.43
1662559509277, 14.163744926452637, 15.291540145874023, 14.829776763916016, 14.516104
698181152, 13.807560920715332, 15.194940567016602, 14.930672645568848, 14.4991493225
09766, 14.106714248657227, 13.00160026550293, 13.948029518127441, 14.15295982360839
8, 13.986237525939941, 14.062457084655762, 15.536728858947754, 12.575069427490234, 1
3.356731414794922, 13.838735580444336, 13.011798858642578, 13.23217487335205, 13.127
532005310059, 14.121650695800781, 13.928903579711914, 15.029956817626953, 12.3636693
95446777, 13.328701972961426, 14.134869575500488, 14.133472442626953, 14.15508937835
6934, 14.248735427856445, 13.977056503295898, 13.838506698608398, 13.40181064605712
9, 13.273768424987793, 14.859960556030273, 14.372488021850586, 15.478314399719238, 1
4.784196853637695, 14.093706130981445, 14.060896873474121, 13.995675086975098, 13.88
0110740661621, 14.337549209594727, 14.638047218322754, 13.741320610046387, 13.390722
274780273, 15.355609893798828, 14.122230529785156, 14.430866241455078, 13.5663728713
98926, 14.248558044433594, 12.730069160461426, 14.0614652633667, 14.084145545959473,
13.647690773010254, 13.689550399780273, 12.917201042175293, 13.876211166381836, 13.2
85964012145996, 13.875371932983398, 12.970157623291016, 14.548789978027344, 14.06610
39352417, 12.420745849609375, 13.092912673950195, 13.995501518249512, 14.23481464385
9863, 13.2842435836792, 13.026178359985352, 12.84869384765625, 12.69244384765625, 1
3.609149932861328, 13.427776336669922, 13.749995231628418, 13.306025505065918, 14.63
3153915405273, 13.337091445922852, 14.1096830368042, 14.171741485595703, 13.68952751
159668, 12.195439338684082, 13.483713150024414, 13.291114807128906, 13.1383285522460
94, 13.29703140258789, 13.622182846069336, 13.465187072753906, 13.647174835205078, 1
3.107841491699219, 13.571429252624512, 14.175902366638184, 13.369853973388672, 13.47
3799705505371, 12.949009895324707, 12.5828857421875, 14.202300071716309, 13.57485866
5466309, 13.382189750671387, 13.894675254821777, 14.718762397766113, 12.652880668640
```



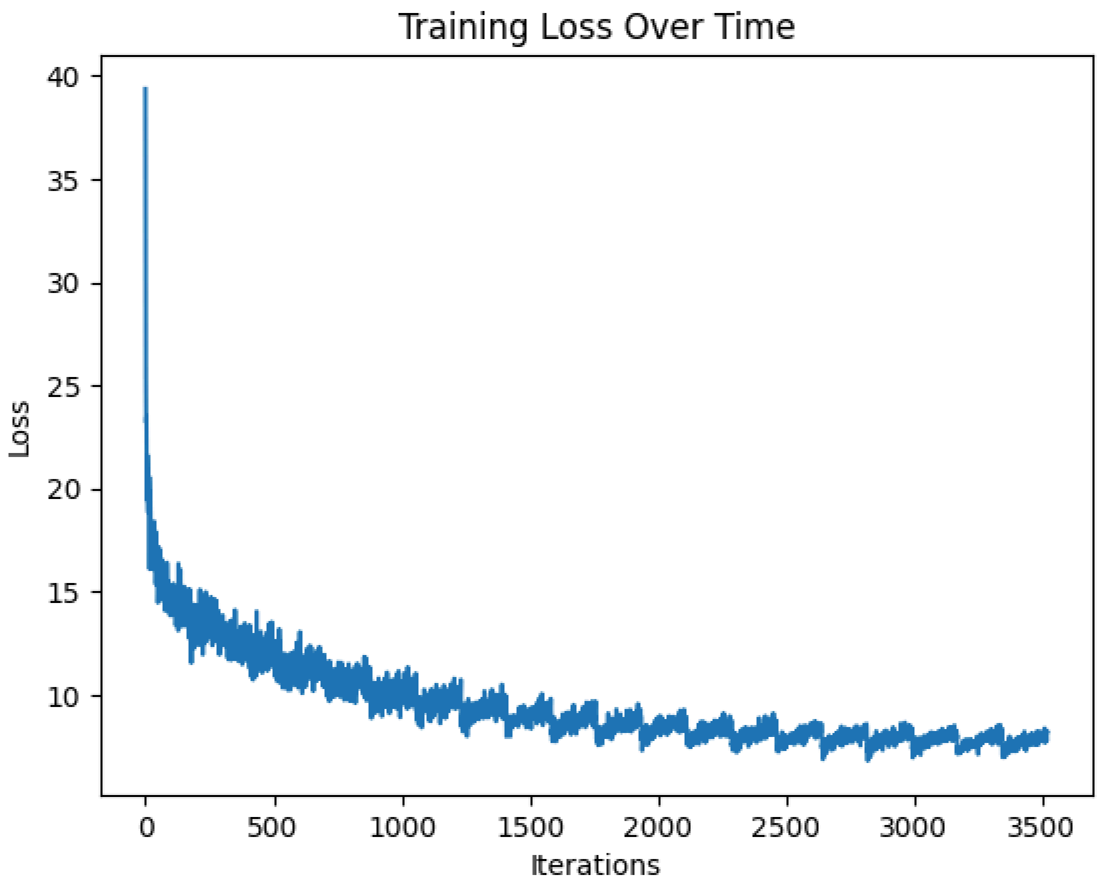
According to the grid search the best results were achieved when embed_dim=64 , hidden_size=256 , learning rate=0.005 , batch size=50

Training the model with the best parameters for 20 epochs

In [17]:
```
from itertools import product
import matplotlib.pyplot as plt

# Define the hyperparameter options
embed_dims = [64]
hidden_sizes = [256]
learning_rates = [0.005]
batch_sizes = [50]
epoch_counts = [20]

# 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},
```

```
    # 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, l_rate

    # 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}")
```

```
Epoch:  0  loss :  tensor(15.0824, grad_fn=<DivBackward0>)
Epoch:  1  loss :  tensor(13.0303, grad_fn=<DivBackward0>)
Epoch:  2  loss :  tensor(12.5582, grad_fn=<DivBackward0>)
Epoch:  3  loss :  tensor(11.5894, grad_fn=<DivBackward0>)
Epoch:  4  loss :  tensor(10.4705, grad_fn=<DivBackward0>)
Epoch:  5  loss :  tensor(11.0322, grad_fn=<DivBackward0>)
Epoch:  6  loss :  tensor(10.7255, grad_fn=<DivBackward0>)
Epoch:  7  loss :  tensor(9.9407, grad_fn=<DivBackward0>)
Epoch:  8  loss :  tensor(9.1783, grad_fn=<DivBackward0>)
Epoch:  9  loss :  tensor(8.6643, grad_fn=<DivBackward0>)
Epoch:  10  loss :  tensor(8.8889, grad_fn=<DivBackward0>)
Epoch:  11  loss :  tensor(8.9221, grad_fn=<DivBackward0>)
Epoch:  12  loss :  tensor(8.6386, grad_fn=<DivBackward0>)
Epoch:  13  loss :  tensor(7.9135, grad_fn=<DivBackward0>)
Epoch:  14  loss :  tensor(8.3070, grad_fn=<DivBackward0>)
Epoch:  15  loss :  tensor(8.4028, grad_fn=<DivBackward0>)
Epoch:  16  loss :  tensor(8.3472, grad_fn=<DivBackward0>)
Epoch:  17  loss :  tensor(8.0470, grad_fn=<DivBackward0>)
Epoch:  18  loss :  tensor(8.3436, grad_fn=<DivBackward0>)
Epoch:  19  loss :  tensor(8.2010, grad_fn=<DivBackward0>)
```




```
Embed Dim: 64, Hidden Size: 256, Num Layers: 1, Learning Rate: 0.005, Batch Size: 5
0, Epochs: 20, Final Loss: 8.201017379760742
```

In [18]:
```
model.autocomplete(["univ", "math", "neur", "engin"])
```

Out[18]:
```
['university---', 'mathematee---', 'neural-------', 'engineer-----']
```

In [19]:
```
model.autocomplete(["calc","sci","gard","tele"])
```

Out[19]:
```
['calcium------', 'scientific---', 'gardening----', 'television---']
```

In [20]:
```
model.autocomplete(["gener","diffi","challe","acade"])
```

Out[20]:
```
['general------', 'difficulties-', 'challenged---', 'academics----']
```

In [21]:
```
model.autocomplete(["gend","acci","accomo","dict"])
```

Out[21]:
```
['gender-------', 'accing-------', 'accomonities-', 'dictionary---']
```

As shown in the above example, the model has correctly predicted most of the words. However, in some cases, it did not complete the words accurately. By training the model for more epochs, conducting a larger grid search, and increasing the number of LSTM layers or the complexity of the model, we can further improve its performance and achieve better results.

As mentioned in the question, we can observe familiar substrings such as "ties" in "difficulties" and "ing" in "gardening." The model has learned these substrings and has attempted to apply them to other words, which resulted in incorrect outputs such as "accomonities" and "accing."