Next Word Prediction Using Stacked LSTM Networks in Deep Learning

The text dataset is a children stories which is taken from kaggle. Using this dataset, a next word predicting model has been trained using stacked LSTM.

In [3]:
```
!kaggle datasets download -d edenbd/children-stories-text-corpus
```

```
Dataset URL: https://www.kaggle.com/datasets/edenbd/children-stories-text-corpus
License(s): CC0-1.0
children-stories-text-corpus.zip: Skipping, found more recently modified local copy (use --force to force download)
```

In [4]:
```
import zipfile
zip_ref=zipfile.ZipFile("children-stories-text-corpus.zip","r")
zip_ref.extractall()
zip_ref.close()
```

In [5]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import re
```

In [6]:
```
with open('cleaned_merged_fairy_tales_without_eos.txt', 'r', encoding='UTF-8') as file:
    content = file.read()
print(content[:500])
```

```
The Happy Prince.
HIGH above the city, on a tall column, stood the statue of the Happy Prince.  He was gilded all over with thin leaves of f
ine gold, for eyes he had two bright sapphires, and a large red ruby glowed on his sword-hilt.
He was very much admired indeed.  “He is as beautiful as a weathercock,” remarked one of the Town Councillors who wished t
o gain a reputation for having artistic tastes; “only not quite so useful,” he added, fearing lest people should think him
unpractical, which h
```

In [7]:
```
#removing some punctuations using regex
def reg(text):
    pattern=r"[^\w\s\.\,\;]"
    processed=re.sub(pattern,'',text)
    return processed

reg_text=reg(content)
```

This is a huge datasets for a next word predicting model. First 1000000 strings have taken from it so that it can be handled by the system and also to train it quickly.

In [9]:
```
reg_text=reg_text.replace("\n"," ")
txt=reg_text[:1000000]
txt[:1000]
```

Out[9]:
```
'The Happy Prince. HIGH above the city, on a tall column, stood the statue of the Happy Prince.  He was gilded all over
with thin leaves of fine gold, for eyes he had two bright sapphires, and a large red ruby glowed on his swordhilt. He wa
s very much admired indeed.  He is as beautiful as a weathercock, remarked one of the Town Councillors who wished to gai
n a reputation for having artistic tastes; only not quite so useful, he added, fearing lest people should think him unpr
actical, which he really was not. Why cant you be like the Happy Prince asked a sensible mother of her little boy who wa
s crying for the moon.  The Happy Prince never dreams of crying for anything. I am glad there is some one in the world w
ho is quite happy, muttered a disappointed man as he gazed at the wonderful statue. He looks just like an angel, said th
e Charity Children as they came out of the cathedral in their bright scarlet cloaks and their clean white pinafores. How
do you know said the Mathematical Master'
```

In [10]:
```
len(txt)
```

Out[10]:
```
1000000
```

In [11]:
```
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.utils import to_categorical
```

For vectorization, "Tokenizer" object has been used. "fit_on_texts" function allocated a number for each word in that text.

In [13]:
```
token=Tokenizer()
token.fit_on_texts([txt])
```

In [80]:
```python
# Showing index of word
token_word=token.word_index
```

"texts_to_sequences" function converted the text into a list of tokens corresponding the word.

In [16]:
```python
token_line=token.texts_to_sequences([txt])[0]
token_len=len(token_line)
token_len
```

Out[16]:
```
191738
```

In [82]:
```python
token_line[:15]
```

Out[82]:
```
[1, 209, 72, 302, 444, 1, 583, 29, 4, 1030, 3394, 182, 1, 1283, 5]
```

The whole token sequences have been made a array having (n,4) dimension. So that for every row first three elements can be used as input and fourth element as output.

In [86]:
```python
input=[]
for i in range(4*(token_len//4)-3):
    input.append(token_line[i:i+4])
input[:10]
```

Out[86]:
```
[[1, 209, 72, 302],
 [209, 72, 302, 444],
 [72, 302, 444, 1],
 [302, 444, 1, 583],
 [444, 1, 583, 29],
 [1, 583, 29, 4],
 [583, 29, 4, 1030],
 [29, 4, 1030, 3394],
 [4, 1030, 3394, 182],
 [1030, 3394, 182, 1]]
```

Here the splitting has been done which was stated above. "x" is the input and "y_" is the output.

In [21]:
```python
x=[]
y_=[]

for i in input:
    x.append(i[:-1])
    y_.append(i[-1])
x=np.array(x)
y_=np.array(y_)
print(f"shape of x: {x.shape}")
print(f"shape of y: {y_.shape}")
```

```
shape of x: (191733, 3)
shape of y: (191733,)
```

In [22]:
```python
x_col=x.shape[1]
print(f"total column in x: {x_col}")
```

```
total column in x: 3
```

We can notice that token started from 1. So to include 0, "+1" business has been done here.

In [24]:
```python
total_word=len(token.word_index)
total_word=total_word+1
total_word
```

Out[24]:
```
9234
```

This is a multiclass classification. So the output can't be just a number. So, the categorization has been done using "to_categorical" function. Also notice that, the output can be any word from our word vocabulary. so, the "total_word" has been taken as length of one output.

In [26]:
```python
y=to_categorical(y_,num_classes=total_word)
y.shape
```

Out[26]:
```
(191733, 9234)
```

In [27]:
```python
from tensorflow import keras
from keras import Sequential
from keras.layers import Dense, LSTM, Embedding,Dropout
```

1. Word embedding has been done to create a array corresponding every unique word/token.
2. Then LSTM layers have been used for finding the pattern.
3. To reduce overfitting droupout has been used.
4. As activation function, Softmax has been used because it is a multiclass classification problem.

In [29]:
```
model=Sequential()

model.add(keras.Input(shape=(x_col,)))
model.add(Embedding(total_word,100))
model.add(LSTM(500, return_sequences=True))
model.add(LSTM(500))
model.add(Dropout(0.5))
model.add(Dense(total_word,activation="softmax"))

model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 3, 100)	923,400
lstm (LSTM)	(None, 3, 500)	1,202,000
lstm_1 (LSTM)	(None, 500)	2,002,000
dropout (Dropout)	(None, 500)	0
dense (Dense)	(None, 9234)	4,626,234

Total params: 8,753,634 (33.39 MB)

Trainable params: 8,753,634 (33.39 MB)

Non-trainable params: 0 (0.00 B)

In [30]:
```
model.compile(optimizer="adam",loss="categorical_crossentropy",metrics=["accuracy"])
```

In [31]:
```
# Using earlystopping to save from overfitting

callback=keras.callbacks.EarlyStopping(
    monitor="loss",
    min_delta=0,
    patience=0,
    verbose=1,
    mode="auto",
    baseline=None,
)
```

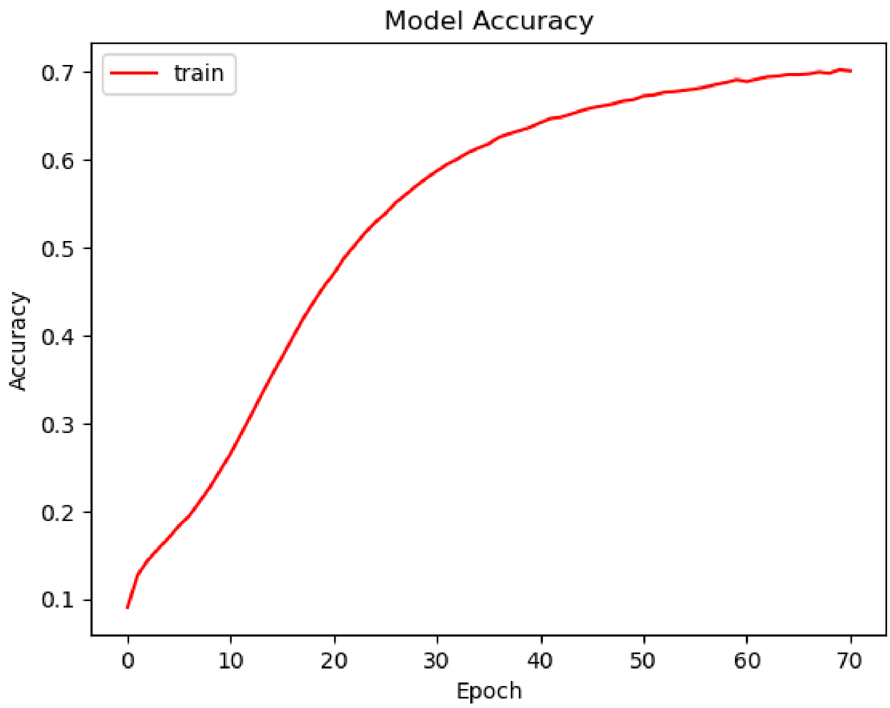
In [32]:
```
history=model.fit(x,y,epochs=100, batch_size=50, callbacks=callback)  #, validation_data=(x_test,y_test)
```

```
Epoch 1/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 91s 23ms/step - accuracy: 0.0769 - loss: 6.3871
Epoch 2/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 84s 22ms/step - accuracy: 0.1233 - loss: 5.5513
Epoch 3/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.1438 - loss: 5.2408
Epoch 4/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.1580 - loss: 5.0369
Epoch 5/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 84s 22ms/step - accuracy: 0.1695 - loss: 4.8613
Epoch 6/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.1847 - loss: 4.6762
Epoch 7/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 84s 22ms/step - accuracy: 0.1975 - loss: 4.5126
Epoch 8/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 84s 22ms/step - accuracy: 0.2139 - loss: 4.3399
Epoch 9/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.2347 - loss: 4.1209
Epoch 10/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.2541 - loss: 3.9301
Epoch 11/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.2750 - loss: 3.7294
Epoch 12/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 84s 22ms/step - accuracy: 0.2982 - loss: 3.5363
Epoch 13/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 84s 22ms/step - accuracy: 0.3224 - loss: 3.3371
Epoch 14/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 84s 22ms/step - accuracy: 0.3485 - loss: 3.1553
Epoch 15/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 84s 22ms/step - accuracy: 0.3698 - loss: 2.9869
Epoch 16/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 84s 22ms/step - accuracy: 0.3923 - loss: 2.8340
Epoch 17/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 84s 22ms/step - accuracy: 0.4157 - loss: 2.6872
Epoch 18/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 84s 22ms/step - accuracy: 0.4401 - loss: 2.5372
Epoch 19/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 84s 22ms/step - accuracy: 0.4586 - loss: 2.4299
Epoch 20/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 84s 22ms/step - accuracy: 0.4770 - loss: 2.3188
Epoch 21/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.4925 - loss: 2.2187
Epoch 22/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.5114 - loss: 2.1239
Epoch 23/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 84s 22ms/step - accuracy: 0.5257 - loss: 2.0466
Epoch 24/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.5382 - loss: 1.9759
Epoch 25/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 84s 22ms/step - accuracy: 0.5516 - loss: 1.9074
Epoch 26/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 84s 22ms/step - accuracy: 0.5618 - loss: 1.8515
Epoch 27/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 84s 22ms/step - accuracy: 0.5751 - loss: 1.7843
Epoch 28/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 84s 22ms/step - accuracy: 0.5834 - loss: 1.7408
Epoch 29/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 84s 22ms/step - accuracy: 0.5942 - loss: 1.6854
Epoch 30/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 84s 22ms/step - accuracy: 0.6022 - loss: 1.6497
Epoch 31/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 84s 22ms/step - accuracy: 0.6122 - loss: 1.6019
Epoch 32/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 84s 22ms/step - accuracy: 0.6188 - loss: 1.5692
Epoch 33/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.6251 - loss: 1.5361
Epoch 34/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.6307 - loss: 1.5145
Epoch 35/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.6376 - loss: 1.4801
Epoch 36/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.6398 - loss: 1.4636
Epoch 37/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.6504 - loss: 1.4291
Epoch 38/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.6529 - loss: 1.4041
Epoch 39/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.6552 - loss: 1.3991
Epoch 40/100
```

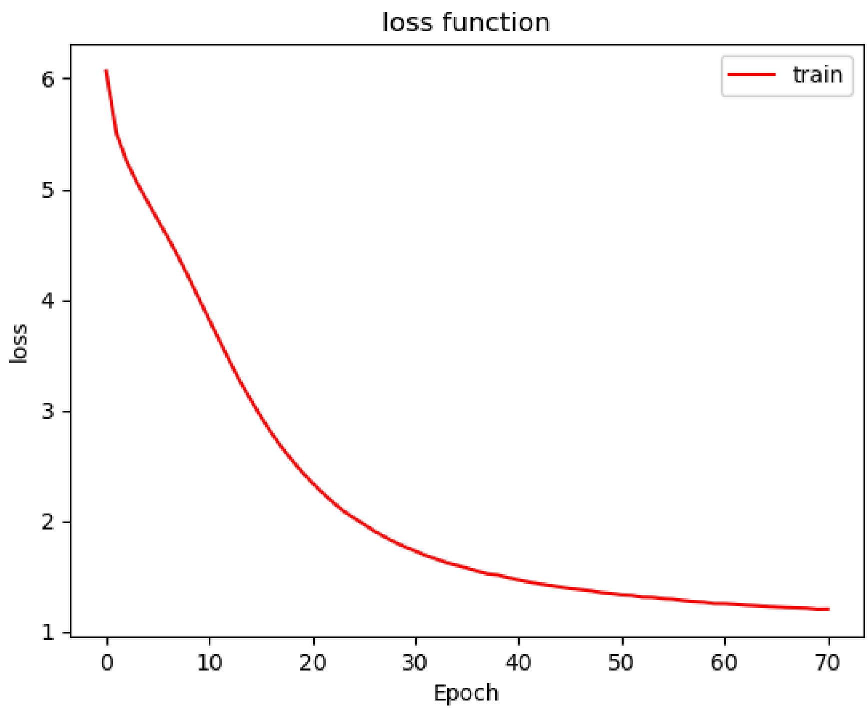
```
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.6603 - loss: 1.3692
Epoch 41/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.6647 - loss: 1.3548
Epoch 42/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.6697 - loss: 1.3377
Epoch 43/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.6710 - loss: 1.3197
Epoch 44/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.6739 - loss: 1.3091
Epoch 45/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.6807 - loss: 1.2867
Epoch 46/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.6815 - loss: 1.2789
Epoch 47/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 87s 23ms/step - accuracy: 0.6843 - loss: 1.2661
Epoch 48/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.6824 - loss: 1.2662
Epoch 49/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.6876 - loss: 1.2453
Epoch 50/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.6886 - loss: 1.2447
Epoch 51/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.6930 - loss: 1.2282
Epoch 52/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.6937 - loss: 1.2273
Epoch 53/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 84s 22ms/step - accuracy: 0.6964 - loss: 1.2117
Epoch 54/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 84s 22ms/step - accuracy: 0.6995 - loss: 1.2020
Epoch 55/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.7003 - loss: 1.1905
Epoch 56/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.7018 - loss: 1.1909
Epoch 57/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.7035 - loss: 1.1760
Epoch 58/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.7062 - loss: 1.1693
Epoch 59/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.7079 - loss: 1.1629
Epoch 60/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.7101 - loss: 1.1594
Epoch 61/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.7091 - loss: 1.1481
Epoch 62/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.7126 - loss: 1.1439
Epoch 63/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.7160 - loss: 1.1315
Epoch 64/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.7142 - loss: 1.1342
Epoch 65/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.7177 - loss: 1.1269
Epoch 66/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.7161 - loss: 1.1251
Epoch 67/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.7180 - loss: 1.1172
Epoch 68/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.7198 - loss: 1.1156
Epoch 69/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.7179 - loss: 1.1121
Epoch 70/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.7214 - loss: 1.1072
Epoch 71/100
3835/3835 ━━━━━━━━━━━━━━━━━━━━ 85s 22ms/step - accuracy: 0.7196 - loss: 1.1095
Epoch 71: early stopping
```

In [33]:

```python
plt.plot(history.history["accuracy"],color="red",label="train")
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend()
plt.show()
```



In [34]:
```
plt.plot(history.history["loss"],color="red",label="train")
plt.title('loss function')
plt.ylabel('loss')
plt.xlabel('Epoch')
plt.legend()
plt.show()
```



Predicting next word using this trained model.

In [36]:
```
def predict_next(inp_text):
    token_line=token.texts_to_sequences([inp_text])[0]
    token_len=len(token_line)
    if token_len<x_col:
        input=pad_sequences([token_line],maxlen=x_col, padding="pre")
    elif token_len>x_col:
        input=token_line[-x_col:]
    else:
        input=token_line
    pred=model.predict(np.array([input]))
    predict_index=np.argmax(pred)
    for word,index in token.word_index.items():
        if index==predict_index:
```

```
            print(word)
    return
```

In [37]:
```
predict_next("stood the statue of")
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 282ms/step
the
```

The original line was "...stood the statue of the Happy Prince."

In [39]:
```
predict_next("when they finally plodded")
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step
came
```

The original line was "...when they finally plodded home."

In [41]:
```
predict_next("He is as beautiful")
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step
as
```

The original line was "...He is as beautiful as a weathercock..."

So, this model is not predicting very well. The training accuracy is 71.96% which is not so good. However, we can use larger dataset and also more LSTM layers to get good accuracy and prediction if the system's capacity permits.