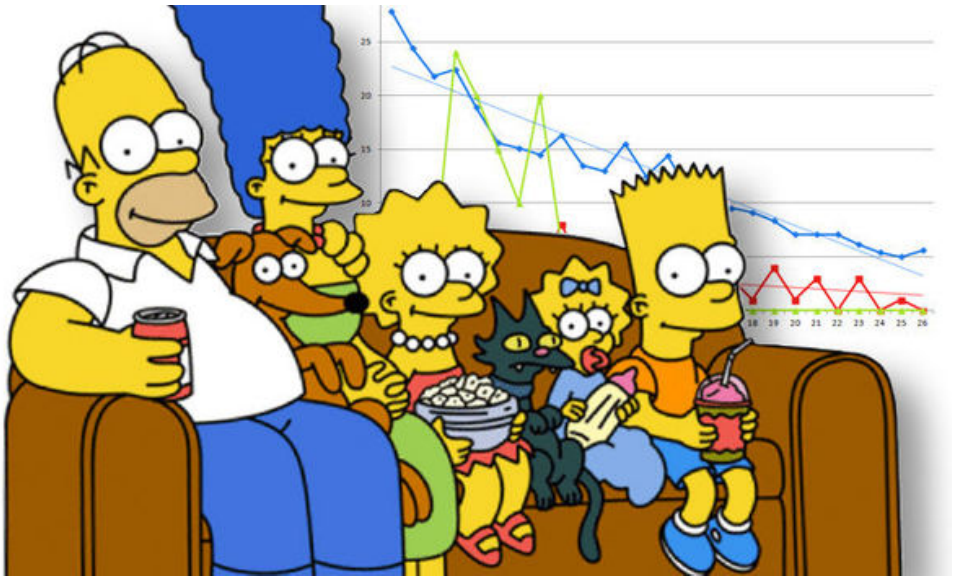
Using AI to write a script for a TV show

So what are we doing?

- Well, we are going to be using AI to write a manuscript for a television show called **The Simpsons** for us!



Why?

- Because it's cool, and because we are **AWESOME**! And that's why.
- Also, it helps us to understand the other applications of **N**atural **L**anguage **P**rocessing other than **Chatbots**.

How are we going to be doing it?

- We are going to be defining a neural network that will do it for us! (Sounds **unbelievable** right? I know)
- Using text generation, we want to see how well **AI** can write more Simpsons scripts based on 600 episodes of text as training data.

So we are going to start with imports down here:

In [1]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation, Dropout
from tensorflow.keras.layers import LSTM
from tensorflow.keras.callbacks import ModelCheckpoint
from keras.utils import np_utils
import numpy as np
import random
import sys
```

```
Using TensorFlow backend.
```

- Now we are done with imports, we can get straight to business! But before that, we need to define what business we have first and what we are going to use for the business.
- So, in order for us to train our model, we need the dataset to be loaded into memory. The dataset we are going to use is the script lines from **The Simpsons** show. (All 600 episodes compressed into one text file! Can you imagine?)
- After loading the dataset, we need to get unique characters.

In [2]:
```
dataset_filename = 'result-3.txt'

# open the file and read it
raw_text = open(dataset_filename).read()

# we have to lower the data to make sure our network recognises all char
acters.
#Also it makes one hot encoding easier
raw_text = raw_text.lower()

unwanted_text = '/:;][{}-=+)(*&^%-_$#@!~1234567890'

# get rid of noise! If you hate noise raise your hand and say 'Python!'
cleaned_text = raw_text.translate(unwanted_text)

# First let's get the unique characters
characters = sorted(list(set(cleaned_text)))
```

- Now that we got this far, it means we now have a clean dataset loaded into memory. **Cool** right?
- Then **what**? Then we convert the characters to integers.
- I hope you are not asking **why**?! Because if you are, the answer is, at the moment it's impossible to model the characters directly (Computers understand numbers more than 'Hello, my name is *whatshisname* '). So they have to go into an intermediary phase, where they are converted into numbers(specifically integers)....
- We omit uncommon symbols (*which is the* `**unwanted_text**` *variable*) because of the noise that they add to the data, and omit spaces in consideration of the size-related problems we might face during training, after all, spaces add a considerable amount of size to the data.
- With these features filtered out of the data, and the data now in a sorted list form, we then create two dictionaries allowing us to translate between word IDs and the words that they represent. The word IDs allow the words to be represented numerically in the model, and the mapping between words and their IDs allows the model's output to be readable by the ~~earth's most abundant plag~~ humans.

In [3]:
```
# Now we start mapping!
character_to_integer = dict((c, i) for i, c in enumerate(characters))
integer_to_character = dict((i, c) for i, c in enumerate(characters))

# Summary of what we now have after that sort of tedious process
num_characters = len(cleaned_text)
vocabulary_length = len(characters)

print('Total number of characters: {}'.format(num_characters))
print('The length of the vocabulary: {}'.format(vocabulary_length))
```

```
Total number of characters: 1369454
The length of the vocabulary: 63
```

- Now for those who might have been wondering, 'Oh no! What was that all about?', allow me to briefly explain in short (*very short*).
- Take this as example, the list of unique sorted lowercase characters in the dataset could be as follows:

```
['\n', '\r', ' ', '!', '"', "'", '(', ')', '*', ',', '-', '.', ':', ';', '?', '[',
']', '_', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '\xbb', '\xbf', '\xef']
```

- You can see that there may be some characters that we could remove to further clean up the dataset that will reduce the vocabulary and may improve the modeling process. (Which is what we did at the start!)
- After cleaning, we created a set of all of the distinct characters in the dataset, then creating a map of each character to a unique integer.
- Then finally, (something that looks serious), we summarized number of unique characters and the length of the vocabulary that we are going to be using for training (in a few minutes).

- We now need to define the training data for the network. There is a lot of flexibility in how you choose to break up the text and expose it to the network during training.
- As for us, we will split the dataset text up into subsequences with a fixed length of 100 characters, an arbitrary length. We could just as easily split the data up by sentences and pad the shorter sequences and truncate the longer ones (but it makes more sense this way, trust me!).
- Each training pattern of the network is comprised of 100 time steps of one character (X) followed by one character output (y). When creating these sequences, we slide this window along the whole dataset one character at a time, allowing each character a chance to be learned from the 100 characters that preceded it (except the first 100 characters of course).

- As we split up the dataset into these sequences, we convert the characters to integers using our lookup table we prepared earlier (remember the dictionaries? If you don't, scroll up!).
- Enough talking! (for a few minutes for now...)

In [4]:
```
# prepare the dataset of input to output pairs encoded as integers
sequence_length = 5
dataX = []
dataY = []
for i in range(0, num_characters - sequence_length, 1):
        input_sequence = cleaned_text[i:i + sequence_length]
        output_sequence = cleaned_text[i + sequence_length]
        dataX.append([character_to_integer[char] for char in input_seque
nce])
        dataY.append(character_to_integer[output_sequence])
num_patterns = len(dataX)
print('Life is awesome right? The total number of patterns is {}'.format
(num_patterns))
```

```
Life is awesome right? The total number of patterns is 1369449
```

- Now that we have prepared our training data we need to transform it so that it is suitable for use with Keras.
- First we must transform the list of input sequences into the form [samples, time steps, features] expected by an LSTM network.

- Next we need to rescale the integers to the range 0-to-1 to make the patterns easier to learn by the LSTM network that uses the sigmoid activation function by default.
- Finally, we need to convert the output patterns (single characters converted to integers) into a one hot encoding (remember Friday's lecture?).
- This is so that we can configure the network to predict the probability of each of the (*vocabulary length goes here*) different characters in the vocabulary (an easier representation) rather than trying to force it to predict precisely the next character.
- Each y value is converted into a sparse vector (don't let this word get to you, if you don't know what it means, ask uncle Google :)), with a length of (*vocabulary length goes here*), full of zeros except with a 1 in the column for the letter (integer) that the pattern represents.
- This is implemented easily, like this:

In [5]:
```
# reshape X to be [samples, time steps, features]
X = np.reshape(dataX, (num_patterns, sequence_length, 1))

# normalize
X = X / float(vocabulary_length)

# one hot encode the output variable (Remember what Doc said about this
last week?)
y = np_utils.to_categorical(dataY)
```

- Finally, we are done with the *not really boring* stuff, we can safely define our model based on an LSTM.
- You might be asking, "This is the *n* th time you have mentioned this **LSTM** thing! What is it?"
- It is an acronym for **L**ong **S**hort **T**erm **M**emory. It is a type of a neural network that works by repeating a chain of the same network, each passing a message to a successor, but with a bit of memory (*I mean it's in the name isn't it?*).
- Here we define a single hidden LSTM layer with 256 memory units. The network uses dropout with a probability of 20.
- The output layer is a Dense layer using the softmax activation function to output a probability prediction for each of the characters between 0 and 1.
- The problem is really a single character classification problem with (*vocabulary length goes here*) classes and as such is defined as optimizing the log loss (cross entropy), here using the ADAM optimization algorithm for speed.

In [6]:
```
# define the LSTM model
model = Sequential()
model.add(LSTM(256, input_shape=(X.shape[1], X.shape[2]))) # this is the
LSTM layer. Looks too simple right?
model.add(Dropout(0.2))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam')
```

- The network is slow to train (especially on my machine), so we will use model checkpointing to record all of the network weights to file each time an improvement in loss is observed at the end of the epoch. We will use the best set of weights (lowest loss) to instantiate our generative model in the next section

In [7]:
```
# define the checkpoint
weight_filepath="weights/weights-improvement-{epoch:02d}-{loss:.4f}.hdf5
"
checkpoint = ModelCheckpoint(weight_filepath, monitor='loss', verbose=1,
save_best_only=True, mode='min')
callbacks_list = [checkpoint]
```

- We can now fit our model to the data (Finally! *again :)*) But this is the real deal. Here we use 10 epochs (to save time) and a large batch size of 128 patterns to train our baby AI.

In [ ]:
```
model.fit(X, y, epochs=10, batch_size=128, callbacks=callbacks_list)
```

```
Epoch 1/10
1369449/1369449 [==============================] - 785s 573us/step - loss
: 2.7642

Epoch 00001: loss improved from inf to 2.76418, saving model to weights/w
eights-improvement-01-2.7642.hdf5
Epoch 2/10
 175232/1369449 [==>...........................] - ETA: 12:42 - loss: 2.5
574
```

- After running the example, you should have a number of weight checkpoint files in the local "weights" directory (if you have one! Otherwise you should get a red ugly error telling you that you don't!).
- You can delete them all except the one with the smallest loss value.

In [10]:
```
# load the network weights
weight_file = "weights/weights-improvement-09-2.0737.hdf5"
model.load_weights(weight_file)
model.compile(loss='categorical_crossentropy', optimizer='adam')
```

- The simplest way to use the Keras LSTM model to make predictions is to first start off with a seed sequence as input, generate the next character then update the seed sequence to add the generated character on the end and trim off the first character. This process is repeated for as long as we want to predict new characters (e.g. a sequence of 1,000 characters in length).
- We can pick a random input pattern as our seed sequence, then print generated characters as we generate them.

In [11]:
```
# pick a random seed
starting_seed = np.random.randint(0, len(dataX)-1)
pattern = dataX[starting_seed]
print('Our seed: ')
print("\"", ''.join([integer_to_character[value] for value in pattern]),
"\"")
# generate characters
for i in range(1000):
        x = np.reshape(pattern, (1, len(pattern), 1))
        x = x / float(vocabulary_length)
        prediction = model.predict(x, verbose=0)
        index = np.argmax(prediction)
        result = integer_to_character[index]
        input_sequence = [integer_to_character[value] for value in patte
rn]
        sys.stdout.write(result)
        pattern.append(index)
        pattern = pattern[1:len(pattern)]
print("\nCompleted!!!.")
```

```
Our seed:
" d0you "
0do0to0toeey0000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000
Completed!!!.
```

- After training, the results might not be satisfactory (imagine if you ask a baby to write your assignment/homework for you, that won't go well right?), so we can try parameter tuning etc. in order to make it better.
- We could try to increase the size of the network, or we could change the cost function (the `categorical_crossentropy` mentioned above) or we could give ourselves a pat on the back for creating a baby AI that can write a script for some episodes of a t.v. show, I mean anyone could have done it right?

Next then what?

- **Game of Thrones?**
- or maybe **Infinity War 2 or 3?**

THANK YOU VERY MUCH!

We would like to appreciate our sponsor:

visit their website (http://www.deepanalytics.ai)

Like their Facebook page (https://www.facebook.com/DeepAnalyticsAI/)

Join our Facebook group (https://www.facebook.com/groups/hararеschoolofai/)

Download more from our GitHub repository (https://github.com/HarareSchoolOfAI/)