Text Prediction with LSTM & GRU
(Genearting Abstracts of Arxiv Articles)

- Ranjit Patro (20122)

---

1. Introduction

Abstracts are typically the first section of a scientific paper and provide a brief overview of the research, including the problem statement, methodology, and results. However, abstract writing can be time-consuming, and not all authors are skilled at summarizing their work effectively. Therefore, an automated abstract generator could be a useful tool for researchers, publishers, and even readers who want to quickly understand the gist of a paper.

2. Objective

LSTMs (Long Short-Term Memory networks) are a type of recurrent neural network that is particularly effective at modeling sequential data, such as natural language. By training an LSTM model on a large corpus of arxiv papers and their abstracts, it may be possible to develop a system that can generate high-quality abstracts for new papers.

The ultimate goal of the project could be to improve the efficiency and accessibility of scientific research by automating the tedious task of abstract writing and helping readers quickly identify papers that are most relevant to their interests.

3. Description & Methodology:

This project is implemented at character level, including not only alphanumeric chacters but also punctuation and symbols used for typing mathematical expressions in latex. The architecture includes an embedding layer to learn representations of characters in the text, thus, no pre-learnt embeddings are required. This minimal example makes use of two layers of LSTMs stacked on top of each other, where the state of the cells at each time-step is passed forward (many-to-many). The final layer is a fully connected dense layer to model the probability distribution over the vocabulary (characters). The problem can be regarded as a classification problem where the number of categories is the size of the vocabulary. Thus, the loss function is selected to be categorical cross-entropy, where each example in the training set belongs to a single category. The model was trained on 100k abstracts from papers in the arXiv across all categories. The tained model can autocomplete words and generate syntactically correct sentences, although it lacks semantinc understanding and does displays limited long-term memory. The notebook outlines each step in the process and demonstrates the capabilities and limitations of the model.

Arxiv Dataset:

The arXiv dataset is a large collection of scientific papers in various fields such as physics, mathematics, computer science, and more. It contains over 1.7 million articles, with new ones

being added daily, and is considered one of the most comprehensive open-access repositories of scholarly research. The dataset is freely available for download and use, and has been used extensively by researchers and data scientists for tasks such as text classification, topic modeling, and natural language processing.

Methodology:

The Project done using GPU P 100 in Kaggle. The main steps involved in this work are:

- Import and load dataset
- Pre-process the data
- Model Defining and Training
- Inference by model

Data Preprocessing:

[1]:
```
import json
import re
import string
import itertools
import numpy as np
```

[2]:
```
vocab = [n for n in string.printable]
vocab_size = len(vocab)
char2idx = {v:idx for idx,v in enumerate(vocab)}
idx2char= np.array(vocab)
```

+ Code + Markdown

```
gen_json = (json.loads(line) for line in itertools.islice(open('/kaggle/input/arxiv/arxiv-metadata-oai-snapshot.json','r'),10))
temp = []
for line in gen_json:
  temp.append(line['abstract'])
```

```
temp[0]
```

[4]:
```
'  A fully differential calculation in perturbative quantum chromodynamics is\npresented for the production of massive photon pairs at ha
dron colliders. All\nnext-to-leading order perturbative contributions from quark-antiquark,\ngluon-(anti)quark, and gluon-gluon subproces
ses are included, as well as\nall-orders resummation of initial-state gluon radiation valid at\nnext-to-next-to-leading logarithmic accur
acy. The region of phase space is\nspecified in which the calculation is most reliable. Good agreement is\ndemonstrated with data from th
e Fermilab Tevatron, and predictions are made for\nmore detailed tests with CDF and DO data. Predictions are shown for\ndistributions of
diphoton pairs produced at the energy of the Large Hadron\nCollider (LHC). Distributions of the diphoton pairs from the decay of a Higgs
\nboson are contrasted with those produced from QCD processes at the LHC, showing\nthat enhanced sensitivity to the signal can be obtaine
d with judicious\nselection of events.\n'
```

Going through the examples, we noticed a few things:

1. All abstracts start with 2 blanks and end with a newline character.
2. Newline characters are included within the text for aesthetic purposes.
3. Abstracts use latex to type mathematical expressions, which are enclosed in $$. These expressions also make use of the back-slash '\', underscore '_' and the caret (hat) '^' characters, and may include all types of parenthesis, i.e. '()'. '[]', and '{}'.

4. In a few examples, the text has been corrupted and non-printable characters appear. For instance, in one exampple, "don't" is rendered as "donâ\x80\x99t".

These characteristics may introduce some noise to the data. While it may be true that a healthy degree of noise can naturally help prevent overfitting, we don't want the model to learn spurious patterns. Thus, some transformations are necessary in order to clean the data.

1. Newline characters within the text are removed. In turn, an additional newline character is appended at the end, such that all abstract start with two blank spaces and end with two newline characters.
2. Characters resulting from corrupted text are transformed back to their original meaning. This is inferred from reading the examples.

It is straightfoward, if somewhat tedious, to find the relevant instances of corrupted text. The following cell performs the necessary transformations on the first 100k records. Note that further corrupted characters may appear beyond this and must be checked manually if more examples are to be included. Finally, the characters are represented as indices using the lookup table previously defined from the vocabulary.

Model Defining and Training:

[35]:
```
import os
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, GRU, Dense
```

+ Code + Markdown

[36]:
```
seq_len = 150
batch_size = 256
dataset = tf.data.Dataset.from_tensor_slices(abs_list)
dataset = dataset.batch(seq_len+1,drop_remainder=True)
dataset = dataset.map(lambda x: (x[:-1],x[1:]))
dataset = dataset.shuffle(1000).batch(batch_size,drop_remainder=True)
```

[37]:
```
dataset
```

[37…
```
<BatchDataset shapes: ((256, 150), (256, 150)), types: (tf.int64, tf.int64)>
```

Next we can initialise the model. This is done using the keras API with tensorflow on the backend. We will use the Embedding, LSTM and Dense layers. We also import 'os' now as this will be used for setting up checkpoints during training.

The strategy for training the model will be dividing the text into non-overlapping fixed-length sequences. The model will be trained to predict the following character within each sequence. Thus, for each sequence there is a training and a target sequence, where the target is just the sequence shifted one character forward.

At this point, we need to choose the length of the sequences. Scientific writting tends to containg longer sentences than other texts, as technical expressions are usually longer and sentences need

to be precise in meaning to avoid ambiguities. Therefore, a sequence of 150 characters can be chosen based on this intuition. You can experiment with different lenghts and verify the results.

The training, which makes use of gradient descent, is performed using mini-batch. This means that the training examples are fed in blocks containing a fixed number of examples. The batch size impacts training speed and accuracy. Too small or too large batch values can be slow to train. The optimum value is not apriori known and this is another hyper-parameter to experiment with. Moreover, it has been noted that large batch sizes may lead to poorer generalisation capability if the learning rate is not adjusted accordingly. In this case, a value of 256 is chosen as initial guess. You can experiment recording training speed and performance using different batch sizes.

The following cell prepares the dataset to use for training:

1. The dataset is intiatlised using the Dataset class from the Tensorflow module. This allows for easy manipulation of the data.
2. The text is then separeted into sequences
3. Training and target sets are created
4. The examples are shuffled and packed into batches.

The model can now be specificied using the Sequential model class from Keras. This minimal example includes four layers.

[38]:
```
def make_model(vocabulary_size,embedding_dimension,rnn_units,batch_size,stateful):
  model = Sequential()
  model.add(Embedding(vocabulary_size,embedding_dimension,
                      batch_input_shape=[batch_size,None]))
  model.add(LSTM(rnn_units,return_sequences=True,stateful=stateful))
  model.add(LSTM(rnn_units,return_sequences=True,stateful=stateful))
  model.add(Dense(vocabulary_size))
  model.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                optimizer='adam',metrics=['accuracy'])
  model.summary()
  return model
```

+ Code + Markdown

```
emb_dim = 256
rnn_units = 1024
model = make_model(vocab_size,emb_dim,rnn_units,batch_size,False)

#The embedding dimension is set to 256 and the number of cells per layer, to 1024
```

The first is an Embedding layer. This layer learns a representation of the characters in a vector space of some dimension. The embedding dimension is the third hyper-parameter so far introduced. Larger dimensions allow for representations that capture more structure, however, they also increase the number of parameters to train. In principle, transfer-learning could be used by loading learnt representations and thus saving training time.

The second and third layers are made of LSTMs, recurrent units whose current state and output depend not only on the inputs but on the previous state. The number of cells can be tuned to increase the complexity of the model. Note that return_sequences is set to True, indicating that the cells pass forward their state at each time step (many-to-many mode), instead of just returning the state at the last time step (many-to-one mode). The 'stateful' parameter will be set to False during training but can be turned on for generating text later on. In this mode, the internal state of the cells is randomly initialised at the start of each batch. Note that since we have shuffled the training

examples and we are assuming that 150 characters is enough context to predict the next one, it would not make sense to train with 'stateful' in this case.

The fourth layer is a fully connected dense layer with the number of cells given by the vocabulary size. The activation values from this layer will be passed through a softmax function so they can be interpreted as the probability distribution over the space of characters. Note that the activation function is not included in here as it is more efficient to apply it when the loss is computed.

```
emb_dim = 256
rnn_units = 1024
model = make_model(vocab_size,emb_dim,rnn_units,batch_size,False)

#The embedding dimension is set to 256 and the number of cells per layer, to 1024
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (256, None, 256)          25600
_________________________________________________________________
lstm_2 (LSTM)                (256, None, 1024)         5246976
_________________________________________________________________
lstm_3 (LSTM)                (256, None, 1024)         8392704
_________________________________________________________________
dense_1 (Dense)              (256, None, 100)          102500
=================================================================
Total params: 13,767,780
Trainable params: 13,767,780
Non-trainable params: 0
_________________________________________________________________
```

The loss is the categorical cross-entropy loss. The function 'SparseCategoricalCrossentropy', since the targets are sparse vectors. This saves resources on memory as it is not necessary to load all the elements, just the index where the vectors are non-zero. Note the option 'from_logits' is set to True, indicating that the softmax function is to be applied to the result of the last layer. Here, we have used 10 epochs for training.

```
model.fit(dataset,epochs=10,callbacks=[checkpoint_callback])
```

```
Epoch 1/10
2071/2071 [==============================] - 1111s 537ms/step - loss: 1.3233 - accuracy: 0.6181
Epoch 2/10
2071/2071 [==============================] - 1115s 539ms/step - loss: 0.9732 - accuracy: 0.7083
Epoch 3/10
2071/2071 [==============================] - 1118s 540ms/step - loss: 0.9262 - accuracy: 0.7211
Epoch 4/10
2071/2071 [==============================] - 1121s 541ms/step - loss: 0.8997 - accuracy: 0.7284
Epoch 5/10
2071/2071 [==============================] - 1121s 541ms/step - loss: 0.8818 - accuracy: 0.7333
Epoch 6/10
2071/2071 [==============================] - 1120s 541ms/step - loss: 0.8691 - accuracy: 0.7367
Epoch 7/10
2071/2071 [==============================] - 1121s 541ms/step - loss: 0.8597 - accuracy: 0.7392
Epoch 8/10
2071/2071 [==============================] - 1121s 541ms/step - loss: 0.8527 - accuracy: 0.7410
Epoch 9/10
2071/2071 [==============================] - 1119s 541ms/step - loss: 0.8473 - accuracy: 0.7424
Epoch 10/10
2071/2071 [==============================] - 1119s 540ms/step - loss: 0.8432 - accuracy: 0.7435
```

Alternatively, LSTM can be replaced with GRU which is faster to train. Below is the GRU implementation:

```
def make_model(vocabulary_size,embedding_dimension,rnn_units,batch_size,stateful):
  model = Sequential()
  model.add(Embedding(vocabulary_size,embedding_dimension,
                      batch_input_shape=[batch_size,None]))
  model.add(GRU(rnn_units,return_sequences=True,stateful=stateful))
  model.add(GRU(rnn_units,return_sequences=True,stateful=stateful))
  model.add(Dense(vocabulary_size))
  model.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                optimizer='adam',metrics=['accuracy'])
  model.summary()
  return model
```

```
emb_dim = 256
rnn_units = 1024
model = make_model(vocab_size,emb_dim,rnn_units,batch_size,False)
```

```
checkpoint_dir = '/kaggle/temp/'
checkpoint_prefix = os.path.join(checkpoint_dir,'chkpt_{epoch}')
checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(checkpoint_prefix,
                                                         save_weights_only=True)
```

```
model.fit(dataset,epochs=10,callbacks=[checkpoint_callback])
```

Below is the training time, loss and accuracy in each epoch. The Training time drastrictally drops here with respects to training time of each epoch in LSTM.

```
Epoch 1/10
2071/2071 [==============================] - 876s 423ms/step - loss: 1.2426 - accuracy: 0.6382
Epoch 2/10
2071/2071 [==============================] - 876s 423ms/step - loss: 0.9852 - accuracy: 0.7055
Epoch 3/10
2071/2071 [==============================] - 885s 427ms/step - loss: 0.9486 - accuracy: 0.7155
Epoch 4/10
2071/2071 [==============================] - 879s 424ms/step - loss: 0.9310 - accuracy: 0.7203
Epoch 5/10
2071/2071 [==============================] - 881s 425ms/step - loss: 0.9210 - accuracy: 0.7229
Epoch 6/10
2071/2071 [==============================] - 887s 429ms/step - loss: 0.9147 - accuracy: 0.7246
Epoch 7/10
2071/2071 [==============================] - 888s 429ms/step - loss: 0.9107 - accuracy: 0.7257
Epoch 8/10
2071/2071 [==============================] - 883s 426ms/step - loss: 0.9080 - accuracy: 0.7263
Epoch 9/10
2071/2071 [==============================] - 879s 425ms/step - loss: 0.9066 - accuracy: 0.7267
Epoch 10/10
2071/2071 [==============================] - 886s 428ms/step - loss: 0.9057 - accuracy: 0.7269
```

4. Results

The model can now be used to generate text. To do this, we can pass a string to act as a seed for the text we want to generate. Note this should be converted to the index representation using the table previously created. The following function takes care of this by converting a given string to the index representation, adding an artificial batch dimension of one, and running predictions on a loop until the desired number of characters is generated.

The next character is predicted by drawing at random from the probability distribution returned by the model. Note that since the targets were shifted sequences, the model outputs a vector of lenth 150 with probability distributions for each character in the sequence. We are only interested in the last one, i.e., the probability distribution of the unseen character.

The predicted character is used as the input in the next iteration. Note that the state of the LSTM cells is reset at the start of each independent prediction but it is remembered while predicting a Fsequence of characters. In other words, if you call the function more than once, it is assumed that each call is independent of the previous ones. model.reset_states() can be commented out to make sure the model to remember what it saw in previous calls. This is if all calls are part of the same sequence of text.

Here, I have implemented a simple character-RNN model to generate text trained on abstracts from articles in the arXiv. The model contains an Embedding layer, two layers of LSTMs and a dense fully-connected layer. Given enough training time, this simple architecture can autocomplete words, predict the next word and formulate complete sentences that are syntactically correct. The more context it is given, the more specific its predictions are, as evidenced by the appearance of key words associated to particular scientific areas. When faced with non-scientific text, it will tend to go back to scientific topics, as that is what it has been trained to do. It can recognise when sequences are non-English and non-words, however in these cases it cannot give coherent predictions. Although the spelling and grammar of the predicted text is mostly correct, the model lacks understanding of the topic sometime.

5. Analysis

Now, analyzing the builded model with different seed value, gives different results. The analysis results were there in the code file. Here are some of the analysis:

```
# Can the model guess obvious characters/words?
seed = ("This is a short string to test if the model knows what the next character sh")
for k in range(5):
  temp = generate_text(model,seed,100)
  print(temp)
```

```
ould appear in the mock sample observations or discs. We derive the rotational rate and 2

sigma_v af
ould be incoherent and transverse forms. In particular we show that the mentioned $\kappa$-conceustio
ould be discussed, which includes SN Ihagims, 14 counter's 'physical' understanding of protein operat
ould improve from underdiption of the electronic structure and intermittency. In particular, we evalu
ares a before the full distribution of the information control. On the other hand, just an order of m
```

It correctly autocomplets "sh-" to "shares", "should", which could be possible options. Given the context, "should" should be favoured, and it does appear 4/5 times. When a sentence in seed given, it takes probability distribution over all the words and gives the resultant/best possible one. Since, this model is unable to detect wheather the seed given is correct or not, irrespective of that it always try to find the best possibility, which sometime goes completely wrong.

When the context of the seed is short the model is not detecting the theme properly, it is giving some scientific details, but when the context in the seed is large enough it is able to detect the proper theme and try to produce more accurate results.

```
# Unrelated string
seed = ("This string does not have to do with science at all, it's a text "
        "about baby shark. Did you know baby shark is the most watched video "
        "on Youtube as of November 2020? That is insane, this is because ")
for k in range(5):
  temp = generate_text(model,seed,100)
  print(temp)
```

```
binding on the temperature and (2) non-zero temperatures will enable only one ferroelectric coupling
variations with random complex networks is generated by orders of rod set of sequences in leaf-less s
if the system is chosen for the IP channel, one are showing that at least some orthosemplands are rel
it can be used to identify and compare their results and obtain results about Portage's lower bounds
super stars tend to faster Statistics in merger shocks when the kinematic motion of ~1.4 loo signals
```

```
# Text in another language. The characters have structure (word and sentences)
# but the model has never seen these combinations before
seed = ("Este texto esta escrito en otro idioma que el modelo no ha aprendido, "
        "veamos que pasa si recibe estas palabras raras que no existen en "
        "ingles. Lo que el modelo predice es que ")
for k in range(5):
  temp = generate_text(model,seed,100)
  print(temp)
```

```
to their inner small transfer data is proposed. The measurement method was established for measured e
to the fact that peculiar volumes can be avoided by that there is no zero knot over a

simple isometr
adjoint and isotropic dissipation scheduling in stage-only control spreadsheet-space structure involv
accurately, within the submm enverse entangled discretization is easy to use the planetary eciment wi
and, no measurement is postulated on the related original generalisations that provide a

threefold,
```

When the model gets completely different from the actual learnt things, it is giving some scientific context related to that, which is not much clear and relatable.

```
# Gibberish
seed = ("m0n923h lxnaefpw;'[kdawpe_dlen;a[ak[k] [';jd0389hufw")
for k in range(5):
  temp = generate_text(model,seed,100)
  print(temp)
```

```
eighz, Reggeiem(195)Ga(sed) Both Hamiltonian approximation calculations we compiled with Bayesian tec
ame, the Reesti Eu), e.g., Sommerfect hinge of inequalities which are propled that (involving vacanci
s]),i] R^3, there is a CAT-295 manomagnetic analogue of Alfvenic, Coupled Decrean liquid semiclassica
}] of the moduli space of a

thirdinal set $M$. We then show that all 'classical results are generali
-.ergin)] have the most relevant scale invariant unitary statistics. Simulators and momentum des and
```

6. Conclusion

In conclusion, this project explored the use of LSTM and GRU neural networks for text generation. The results showed that both models can effectively generate coherent and meaningful text. The project justifies that the GRU takes lesser train time over LSTM and the produced sentences were more or less similar in both the cases. The project also highlighted the importance of hyperparameter tuning and training on a large corpus of text to achieve optimal results.

Benefits of GRU over LSTM:

Both GRU (Gated Recurrent Unit) and LSTM (Long Short-Term Memory) are types of recurrent neural networks that are commonly used in natural language processing tasks such as language modeling, speech recognition, and text generation. While both models have similar capabilities, there are some benefits to using GRU over LSTM:

1. Fewer Parameters: GRUs have fewer parameters than LSTMs, which can make them faster to train and less prone to overfitting. This can be especially beneficial when working with smaller datasets or when computational resources are limited.
2. Simpler Architecture: GRUs have a simpler architecture than LSTMs, which can make them easier to understand and implement. This can be beneficial for researchers or developers who are new to deep learning and want to experiment with different types of models.
3. Better Handling of Short-Term Dependencies: GRUs are designed to handle short-term dependencies more effectively than LSTMs. This means that they may perform better on tasks that require modeling sequences with short-term dependencies, such as speech recognition or handwriting recognition.

That being said, LSTMs are still a popular choice for many natural language processing tasks and may perform better than GRUs on certain types of data or tasks that require modeling longer-term dependencies. Ultimately, the choice between using GRU or LSTM will depend on the specific requirements of the task at hand and the available resources for training and implementation.

Overall, the findings suggest that LSTM and GRU models can be useful for generating text in a variety of applications, including language modeling, dialogue generation, and content creation. As with any machine learning research, there is always room for further improvement and exploration, such as experimenting with different model architectures or incorporating additional features like attention mechanisms. Nonetheless, the results of this project demonstrate the potential of LSTM and GRU models for text generation and provide a solid foundation for future research in this area.