Machine Learning Engineer Nanodegree

Capstone project

Karim El mokhtari

March 23rd, 2019

Project overview

Recurrent Neural Networks (RNN) networks can learn from sequences and predict sequences as well. They have a wide range of applications such as machine translation, caption generation and predicting from time series.

This project aims particularly at applying RNNs in NLP (Natural Language Processing). The objective is to create a model that can learn from old texts such as sacred books, and generate text that follows the same structure and semantic. I am interested in the holy book of Quran since I am originating from an Arabic culture and would like to use this work as a first step to apply machine learning to learn and classify information from old texts.

In this work, many networks are tested: SimpleRNN (or vanilla RNN), LSTM (Long Short Term Memory) and GRU (Gated Recurrent Unit).

A general overview of RNNs can be found here[1].

Problem Statement

The objective of this project is to measure to what extent a machine can reproduce a text similar to an original known text. This measure includes the nature of words, their frequency and the similarity of the words to the context. Once the machine learns from the text, it is possible to generate a text that is similar to the original. The expected architectures to be used are RNNs.

Datasets and Inputs

The dataset is the sacred book of the Quran. It can be downloaded in many text versions found in this link[2]. The Quran is an old text written in Arabic and is a holy

[1] https://karpathy.github.io/2015/05/21/rnn-effectiveness/

book in Islamic culture[3]. The Arabic language is composed of 28 letters[4] that can be extended to 39 if we consider variants of some letters (like the letter "*alif*" (ا) and the variant "*alif*" with "*hamza*": (أ)).

The Quran corpus is public and can be downloaded at this URL:

http://tanzil.net/docs/download

I am using a basic text version without any ornaments or chapter/verses numbers, so the following options are to be considered in the download page:

1. Quran text: Simple Clean (do not include pause marks, do note include sajdah signs, do not include rub-el-Hizb signs)
2. File format: Text

The Quran corpus is organized in 114 chapters (or *Surahs*) of different lengths[5]. Each chapter consists of many verses. In the text file, verses are put in different lines. Also, chapters are classified as *Meccan* or *Medinan*. This refers to the two holy cities in the Islamic tradition. Below are summary statistics on this corpus:

- Number of chapters (*Surahs*): 114
  - Meccan chapters: 82
  - Medinan chapters: 20
  - Unclassified: 12
- Number of verses (lines): 6,236
- Number of words: 77,437 words
- Number of characters: 323,671 character

The text file is available in the github account related to this project. The encoding used is UTF8.

Github account: https://github.com/elmkarim/arabic_text_rnn

Benchmark Model

I compare the performance of both LSTM[6] and GRU[7] in learning and predicting from the original text using the BLEU metric explained below. The performance of these

---

[2] http://tanzil.net/docs/download
[3] https://en.wikipedia.org/wiki/Quran
[4] https://en.wikipedia.org/wiki/Arabic_alphabet
[5] https://en.wikipedia.org/wiki/List_of_surahs_in_the_Quran
[6] S. Hochreiter and J. Schmidhuber: Long short-term memory. Neural computation, 9(8):17351780, (1997)
[7] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio: Learning Phrase Representations using RNN Encoder Decoder for Statistical Machine Translation. Conference on Empirical Methods in Natural Language Processing (EMNLP), pages 17241734 (2014)

two models are compared to a basic RNN[8] which is a simple recurrent network that consists only of a hidden state and does not use any additional gates for long-term dependency.

Evaluation Metrics

The metric that is used to measure the similarity between the predicted text and the original is the BLEU score[9] applied on 1, 2 and 3-grams. BLEU score is widely used to assess the quality of machine translation models. It evaluates the quality of text which has been translated from a natural language to another by comparing the machine's output with that of a human. BLEU's output is a number between 0 and 1 that indicates the similarity between the reference and generated text, with values closer to one representing more similar texts. Clarity of grammatical correctness is not taken into account.

From a sequence of 100 words from the original text, we will produce a sequence of 10 words using the model. Each predicted word is added to the input sequence for the next timestep as further explained in the section “Algorithms and techniques”. We compare the sequence of 10 words with the true sequence from the original text by computing BLEU-1, BLEU-2 and BLEU-3 scores. BLEU-1 score considers only the precision at unigram level, while BLEU-3 considers tri-grams.

Project Design

Three structures will be studied: vanilla RNN, LSTM and GRU. Words in the original text are tokenized. No Lemmatizing or stemming are applied as I was not able to find these tools for the Arabic language. The text is divided into groups of 101 words. The RNN is trained on 100 words; then it is given the 101st word to predict in a supervised learning schema. The dataset is generated using a sliding window of 101 words starting from the beginning of the text to the end.

The actual number of words in the dataset is 78245. The number of patterns of 100 consecutive words is 78145 patterns from which the RNN, LSTM and GRU will learn.

The training and test sets are respectively 80% and 20% of the total dataset. So 62516 data points are used for training and 15629 are considered for testing. Validation sets are chosen randomly at each epoch during training and will consist of 10% of the training data.

[8] https://arxiv.org/pdf/1506.00019.pdf
[9] K. Papineni, S. Roukos, T. Ward, and W. J. Zhu: BLEU: A method for automatic evaluation of machine translation. In ACL (2002)

Exploratory analysis

Before applying our machine learning model to the corpus, it is necessary to go through an exploratory analysis of the text to understand and show relevant characteristics of its nature.

As a first step, we compute the word count per chapter. The bar chart of Figure 1 shows the reparation of words by chapter.

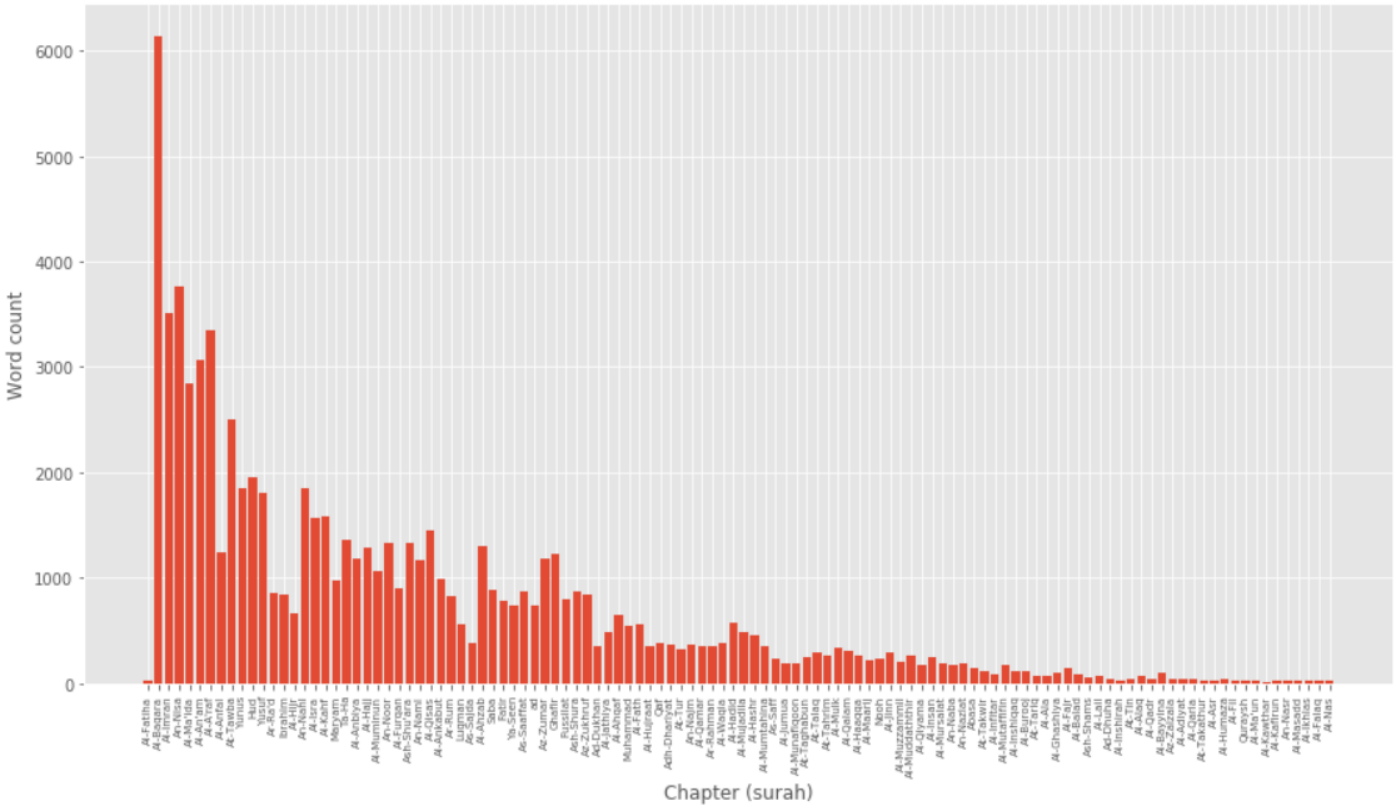

Figure 1: Word count by chapter

We can easily observe a decreasing trend of the word count from the beginning to the end of the corpus. This may suggest that the chapters were arranged according to their length in the Quran. The last thirty chapters are all below 100 words. The second chapter named "Al-Baqara" is by far the lengthiest with 6144 words, followed by the fourth chapter "An-Nissa" with 3767 words. The shortest chapter of the corpus is "Al-Kawthar" that contains only 14 words.

Figure 2 shows a treemap of the top 60 lengthiest chapters of the Quran. The area of the rectangle associated with each chapter is proportional to its word count.

Figure 2: Treemap of the top 60 lengthy chapters

A second analysis shows the words frequency over the whole corpus. A customized list of stopwords was created to remove the frequent words that do not associate with any context.

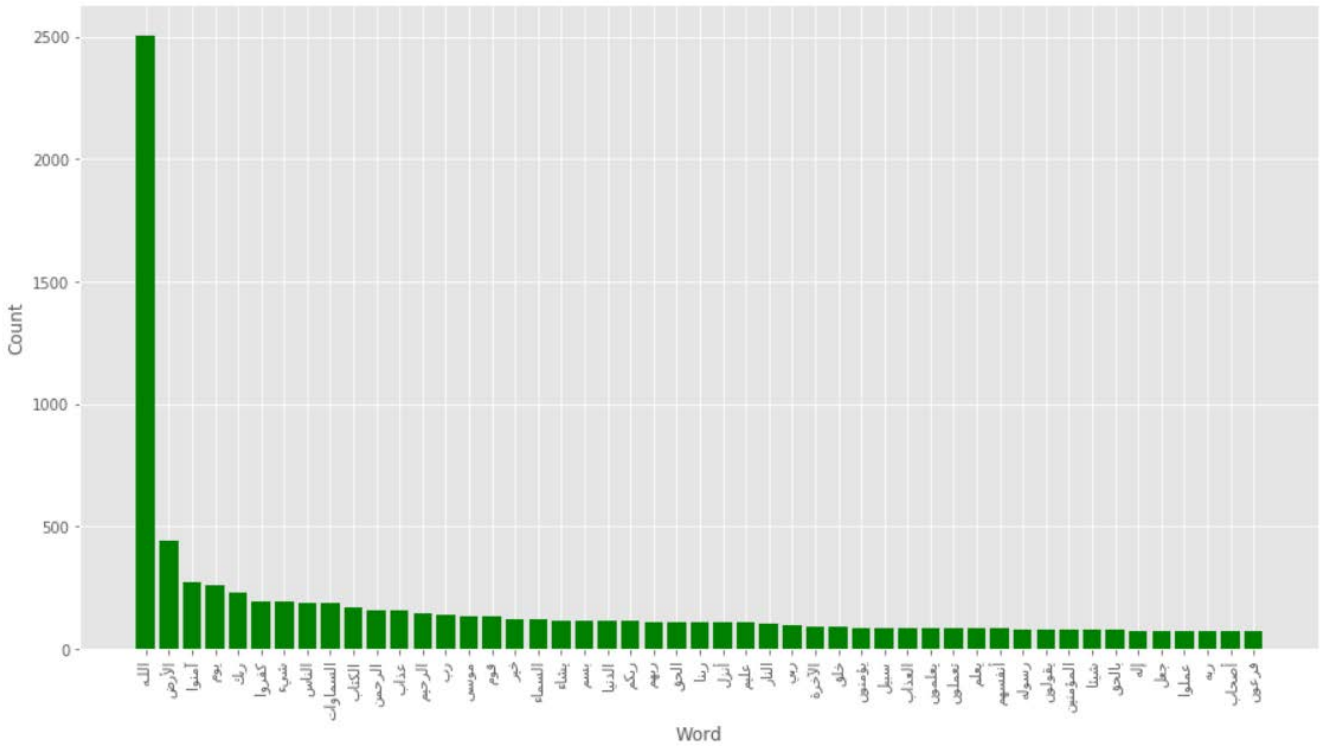

Figure 3: Word frequency over the corpus

It is not surprising to notice on Figure 3 that the most frequent word in the Quran is the word "الله" (God) with 2505 occurrences – it may well be considered as a stop word-. The second frequent word "الأرض" (earth) is far below the word "God" count with 444 occurrences, followed by the words "آمنوا" (who believe) with 270

occurrences, "يوم" (day) with 261 occurrences, "ربك" (your Lord) with 231 occurrences and "كفروا" (who don't believe) with 191 occurrences.

Those frequencies seem logical when we know that the topics discussed in the Quran deal mainly with faith as well as the behaviour and traits of believers and the non-believers according to the Islamic religion. The word "Earth" is also related to the topic of belief since the message of this whole book is destined to people living on this planet.

We illustrate in Figure 4 the repartition of the top 100 frequent words in the Quran.

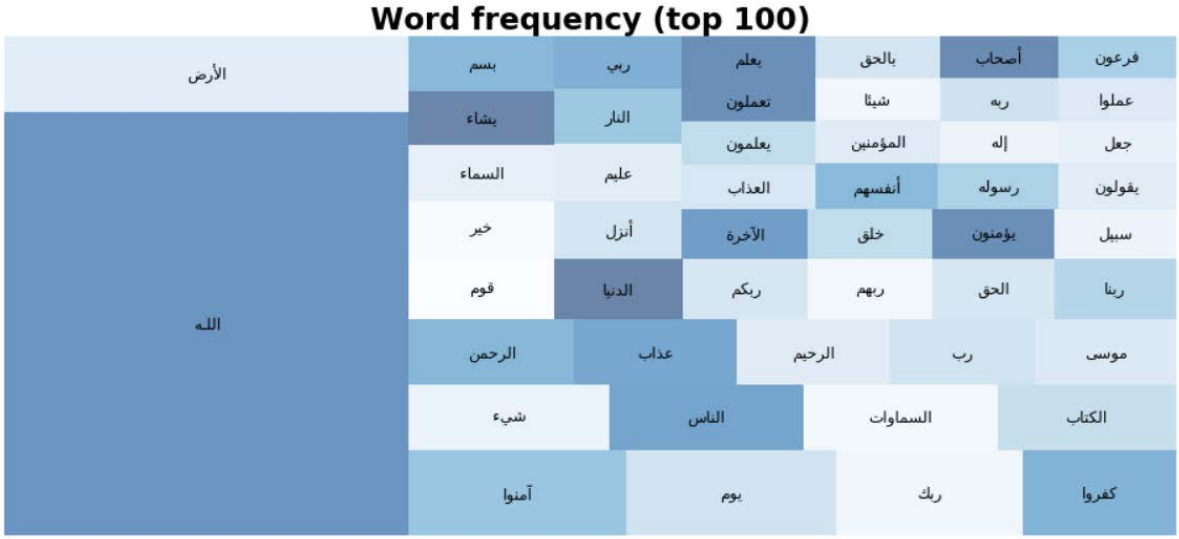

Figure 4: Word frequency over the corpus

This figure shows that many words and their opposite occur with similar frequencies such as:

Word	Meaning	Frequency
آمنوا	who believe	270
الدنيا	world	115
خير	good/well-being	119

Opposite Word	Meaning	Frequency
كفروا	who do not believe	191
الآخرة	hereafter	90
العذاب	torment	86

The two most frequent proper names in the Quran are "موسى" (Moses) with 134 occurrences, followed by "فرعون" (Pharaoh) with 72 occurrences. This suggests that the story of the prophet Moses associated with the personality of Pharaoh is the most popular among the stories narrated in this book.

Another interesting visualization of word frequencies is the word cloud. This illustration, shown on Figure 5, displays the occurrence of the top 200 words graphically by drawing each word with a size proportional to its frequency.

Figure 5: Word cloud showing the top 200 frequent words of the Quran

This figure pictures the main topics that the reader may find in the Quran. As more space is available in this kind of visualization, we can read new words still dealing with the topic of faith such as: "الشيطان" (devil/satan),"الملائكة" (angels),"السماء" (sky/heaven),"هدى" (guidance) and "قلوبهم" (their heart). Frequent bi-grams are also shown in this figure like "يوم القيامة" (the day of judgement),"الحياة الدنيا" (worldly life),"الرحمن الرحيم" (the merciful and gracious) and "السماوات والأرض" (the heavens and the earth).

I performed a last analysis on the entire set of words in the Quran using **Word2Vec**. This library is part of the package **gensim** and allows creating word embedding by sliding a window of 5 words over the whole text and training a neural network over the words of each window. During training, the neural network calculates an embedding that considers the co-occurrence of words in the 5-words windows. The

resulting embedding shows interesting properties. First, it allows switching from a sparse vector representation of words to a dense representation using low dimension vectors of real numbers. The second interesting property is that those vectors can be used to calculate distances between words. Those distances show quite interesting relationships.

Below we describe two examples of the results obtained with the Word2Vec network trained on the Quran corpus.

1. Predicting words that are similar to a specific word. The calculation is based on the cosine distance between word-vectors.
   For example, the closet words to the word "خير" (good/well-being) are:

```
sim_words = w2v_model.wv.most_similar('خير', topn=12)
for w in sim_words:
    print('%s: %.3f' % (w[0], w[1]))
```

```
0.324 :كبير
0.295 :يبين
0.272 :دار
0.263 :وجه
0.258 :كاذبين
0.258 :مؤمن
0.249 :جهه
0.248 :متاع
0.248 :ماء
0.236 :اخري
0.228 :بشرا
0.226 :عدل
```

Word	Meaning	Similarity
كبير	big/enormous	0.324
يبين	show you	0.295
دار	home	0.272
وجه	face	0.263
كاذبين	those who lie (antonym)	0.258
مؤمن	believer	0.258
جهه	side	0.249
متاع	goods	0.248
ماء	water	0.248
اخرى	other	0.236
بشرا	human being	0.228
عدل	Justice/fairness	0.226

We can understand that those different words usually come in contexts where the word "خير" (good/well-being) exists as well. The similarity score is higher for words that are often related to similar contexts such as "كبير" (enormous) that often describes the nature of well-being especially in the context of Heaven or rewards for good deeds.

2. Calculating distances between words and using them in predicting new words. For example, Word2Vec allows calculating the distance between two word-vectors **w1** and **w2**. The resulting distance can be added to another word-vector **w3** to predict a new vector **w4**. As the distance between pairs (**w1**,**w2**) and (**w3**,**w4**) is equal, we can expect the relationship between w1 and w2 to be similar to the one between w3 and w4.
   For instance, let's consider the pair "موسى" (Moses) and "هارون" (Aaron). As they are brothers according to the Quranic context, the distance between them is probably related to the brotherhood relationship. Therefore, when we add this distance to another name such as "اسماعيل" (Ismail), Word2Vec returns "اسحاق" (Isaac) who is Ismail's brother according to the Quranic context.

```
positive1 = 'موسي'
negative1 = 'هارون'

positive2 = 'اسماعيل'

print(w2v_model.wv.most_similar([positive1, positive2], [negative1], topn=1))
```

```
[('0.45400452613830566 ,'اسحاق)]
```

Those distances can also be shown graphically. Embedding vectors are multidimensional; if we want to plot words in a two-dimension space; we need to reduce their dimensionality. We can apply methods such as PCA or t-SNE to capture the most important components in the embedding vector and reduce the words representation to 2D.

We applied t-SNE to all embeddings, and we tried to show to 20 closest words to the word "فرعون" (Pharaoh). The words are plotted as dots in Figure 6.

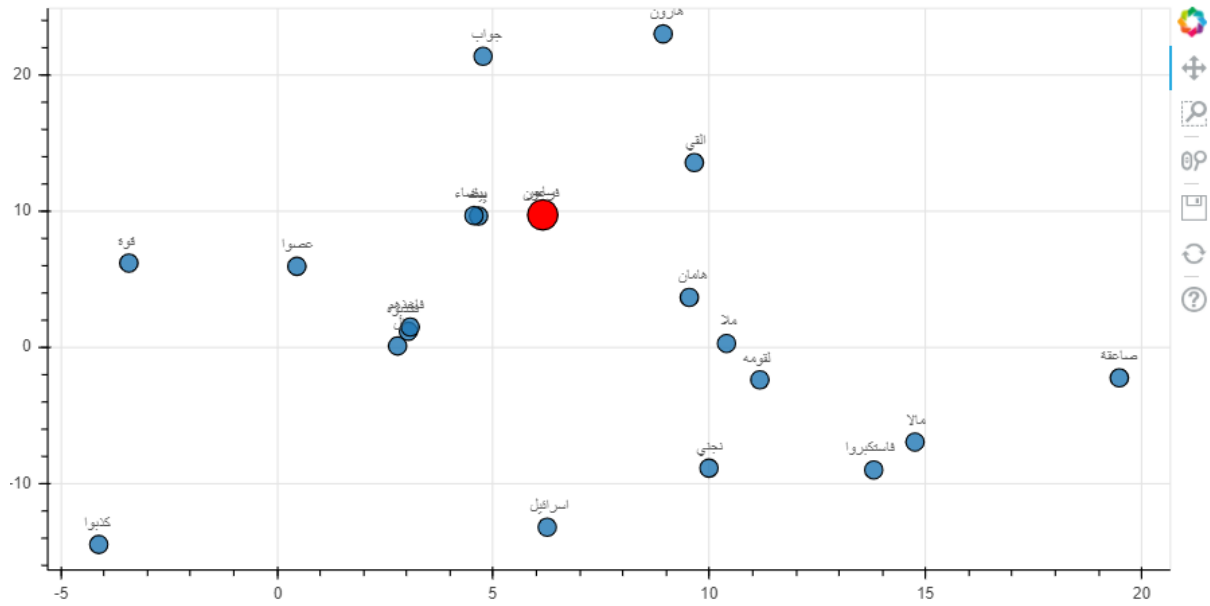

Figure 6: The twenty closest words to the word "فرعون" (Pharaoh)

The calculated words are indeed very relevant to the context. We can read for example, always according to the Quran context, the following words:

Word	Meaning	Context in the Quran
ساحر	Sorcerer	Pharaoh invoked a group of sorcerers to face Moses as Pharaoh thought that Moses was a sorcerer as well
ملأ	Crowd of people	Pharaoh gathered a crowd of people to witness the confrontation between Moses and the sorcerers
هارون	Aaron	Aaron is Moses brother, and he was supporting him all the time
هامان	Haman	Pharaoh's counsellor and the executive leader for his plans

Algorithms and Techniques

Figure 7 depicts the unrolled learning architecture using the RNN network. We denote the words of the corpus by w_i where i is the index of the word in the corpus. This index ranges from 0 (first word) to N-1 (last word) with N the number of words in the full corpus.

During the first step of every epoch, the first group of 100 words (w_0--- w_{99}) is fed into the RNN that predicts the following word ($\widehat{w}_{100}$). The RNN updates its hidden state (h_0) that is used in the following step. This hidden state is supposed to summarize all important data from the past and allows the RNN to predict the following word ($\widehat{w}_{101}$) by using this hidden state (h_0) along with the next group of 100 words (w_1... w_{100}).

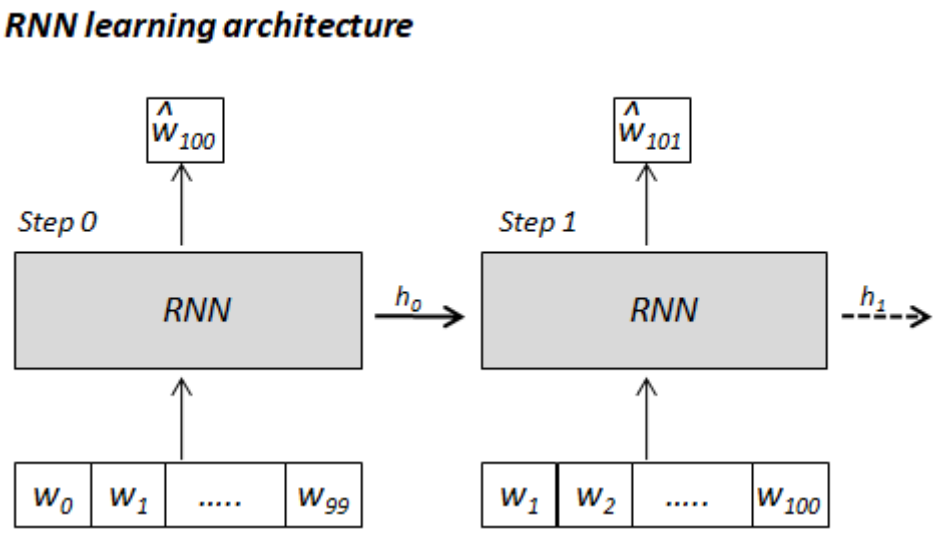

Figure 7: RNN learning architecture

The training in one epoch continues up to the last group of 100 words: (w_{N-100}... w_{N-1}).

Sampling the RNN will consist in providing a random group of 100 words from the original text in the input, let's denote it (w_0 ... w_{99}) and let the RNN predict the following word $\hat{w}_{100}$. In the next step, we will use as input 99 words from the original group (w_1... w_{99}) and add the predicted word as the 100th in the input vector. Figure 8 illustrates this process.

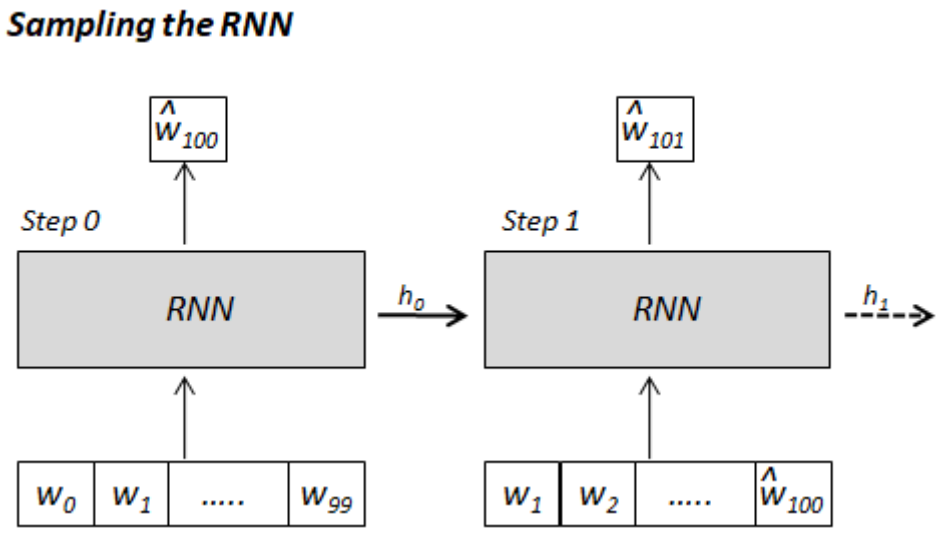

Figure 8: Sampling the RNN

We can continue sampling the RNN by adding at each timestep the predicted word from the last step to the input vector. After 100 timesteps, all the words in the input vector will be coming from the RNN predictions during the last 100 timesteps. In this case, we can test to which extent the RNN will be able to predict the next word correctly even if no original text is provided at its input.

Theoretical concepts of Recurrent Neural Networks

In the vanilla RNN depicted in Figure 9, we maintain a hidden state representing the features in the previous time sequence. Hence, to make a word prediction at time step t, we take both input X_t and the hidden state from the previous time step h_{t-1} to compute h_t. So, generally, an RNN makes a prediction based on the hidden state in the previous timestep and current input by applying the following rule: $h_t=f(X_t,h_{t-1})$.

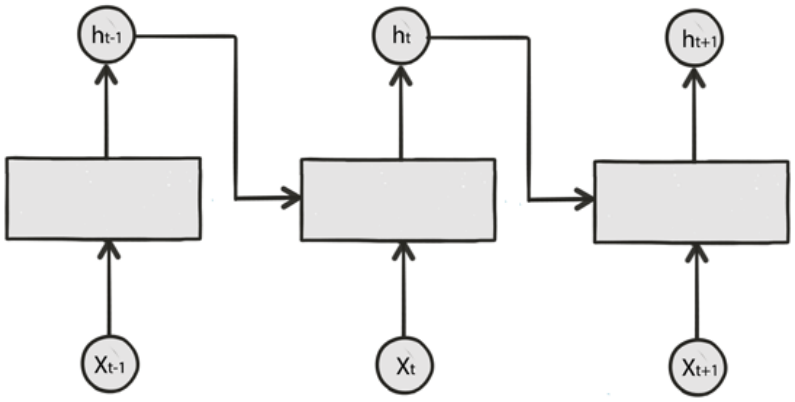

Figure 9: RNN structure

The main disadvantage of this structure is its reliance only on the hidden state of the last time step. In many practical cases, such as language translation, the prediction at time step t requires data from time step t-m eventually. The RNN has no mechanism to store old data. This weak dependency with the past is known under the name of the "vanishing gradient". Indeed, when unrolled, an RNN can be considered as a deep neural network. Backpropagation using gradient descent gives more weight to recent layers (or time steps) than farther ones (older time steps).

To mitigate this problem and reinforce the dependency with old inputs/states, the LSTM or Long Short Term Memory, depicted in Figure 10 defines two different memory cells noted C and h.

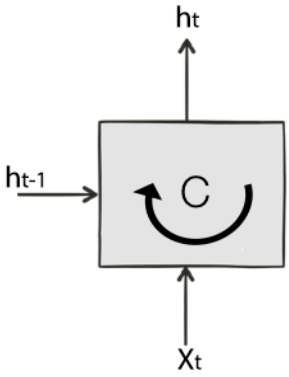

Figure 10: LSTM global structure

The internal cell C retains the relevant data that the LSTM learns to memorize from all past time steps. The state h is only a short version of C and is used as the current output. To learn what to store and erase from the cell C, the LSTM uses three gates called forget, input and output. The role of each gate is described as follows:

- Forget gate: controls what part of the previous cell state will be kept.
- Input gate: controls what part of the new computed information will be added to the cell state C.

- Out gate: controls what part of the cell state will be exposed as hidden state.

It is demonstrated that this process allows diminishing the impact of the vanishing gradient as the gates operate as on/off switches for choosing specific data to memorize or discard from the cell C.

The three gates are neural networks using sigmoid activation functions:

$$\begin{aligned} gate_{forget} &= \sigma(W_{fx} X_t + W_{fh} h_{t-1} + b_f) \\ gate_{input} &= \sigma(W_{ix} X_t + W_{ih} h_{t-1} + b_i) \\ gate_{out} &= \sigma(W_{ox} X_t + W_{oh} h_{t-1} + b_o) \end{aligned}$$

The following equations define how the new values of C and h are calculated using the three gates:

$$\begin{aligned} \tilde{C} &= \tanh(W_{cx} X_t + W_{ch} h_{t-1} + b_c) \\ C_t &= gate_{forget} \cdot C_{t-1} + gate_{input} \cdot \tilde{C} \\ h_t &= gate_{out} \cdot \tanh(C_t) \end{aligned}$$

The GRU (Gated Recurrent Units) is a simplified version of the LSTM. It inherits some concepts from the LSTM and the simple RNN. A GRU does not maintain a cell state C and uses 2 gates instead of 3. The equations of the two gates are as follows:

$$\begin{aligned} gate_r &= \sigma(W_{rx} X_t + W_{rh} h_{t-1} + b) \\ gate_{update} &= \sigma(W_{ux} X_t + W_{uh} h_{t-1} + b) \end{aligned}$$

A proposed new state $\tilde{h}_t$ is calculated using $gate_r$:

$$\tilde{h_t} = \tanh(W_{hx} X_t + W_{hh} \cdot (gate_r \cdot h_{t-1}) + b)$$

The new hidden state is computed by:

$$h_t = (1 - gate_{update}) \cdot h_{t-1} + gate_{update} \cdot \tilde{h_t}$$

$gate_{update}$ defines what data from the old state h_{t-1} to be kept and what new data from $\tilde{h}_t$ to be introduced in h_t.

Implementation

The implementation of the model was done with Python 3.6 and Keras library. The training was performed on a cluster of four NVIDIA P100 GPUs with 16GB of memory.

To fine-tune the model, the following parameters were used:

- Size of the hidden layer of the RNN: I experimented with 500, 1000 and 2000.
- A dropout layer associated with the RNN: I experimented with different values ranging from 0.1 to 0.5.
- Batch size: ranging from 32 to 128

A softmax layer is added to the output of the RNN to calculate the probability of each word. Therefore, I used the "Adam" optimizer with a "categorical cross-entropy" loss function. Adam optimizer is used instead of the classical stochastic gradient descent procedure to update network weights iteratively based on training data.

I created a class called "ModelRNN" in a separate file to create different models based on a Simple RNN, LSTM or GRU. Below the method that creates the model:

```
def create_model(self, hidden_layer, input_shape, output_shape):
    self.model = Sequential()
    self.model.add(self.model_type(hidden_layer, input_shape=input_shape))
    self.model.add(Dropout(0.2))
    self.model.add(Dense(output_shape, activation='softmax'))
    self.model.compile(loss='categorical_crossentropy', optimizer='adam')
    self.model.summary()
    return(self.model)
```

Data is split into vectors of 100-word tokens $\mathbf{X_i}$ that are used as inputs to the RNN; the model is trained to predict the following word token $\mathbf{y_i}$. The different vectors of 100 words are created by shifting a window of size 100 over the entire corpus as shown below (Figure 9):

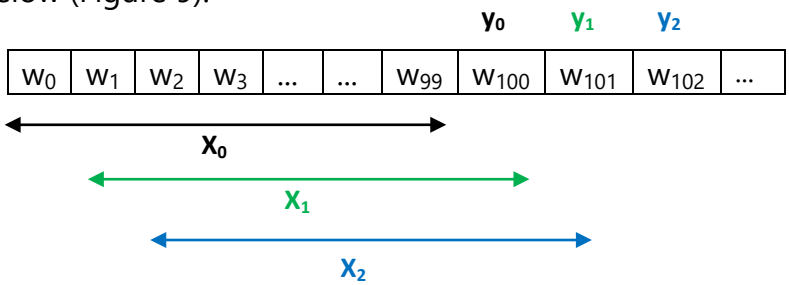

Figure 11: Splitting the corpus into input vector $\mathbf{X_i}$ and target values $\mathbf{y_i}$

The dataset is then split into training set and test set with a 80% vs 20% proportion. The training was made for all three networks SimpleRNN, LSTM and GRU over the training set. Keras create a validation set of 10% of the training data during training. Checkpoints were also added to save the best weights.

Results

After experimenting with different combinations of models parameters, the following parameters were chosen as they performed the best among the tested combinations:

- Batch size: 128
- Hidden layer size for the RNN: 1000. Higher values increase training time and complexity without any improvement in the results
- Dropout: 0.2. Actually no big changes were detected by changing the dropout value.

To evaluate the performance of each one of the three networks (simple RNN, LSTM and GRU), two assessments were performed:

- The first one is by measuring the accuracy of word prediction. In this case, for each input vector $\mathbf{X_i}$, we compare the real output token that stands for the next word in the corpus $\mathbf{y_i}$ with the one predicted by the model denoted $\hat{y}_i$. The accuracy is the number of the correctly predicted words over the number of predicted words

- The second assessment is done using BLEU scores. In this case, for each input vector $\mathbf{X_i}$ from the corpus, we predict from the model a sequence of 10 words then we compare it with the correct sequence of 10 words that follows vector $\mathbf{X_i}$ in the corpus. BLEU scores calculate the similarity at different grams levels. In our case, we considered 1, 2 and 3-grams similarity. The BLEU scores are computed using the NLTK library.

Assessment 1: Accuracy of prediction

We calculate the accuracy on both training and test sets. The results are shown below:

	Accuracy on training set	**Accuracy on test set**
Simple RNN	2.6%	1.8%
LSTM	89.3%	8.5%
GRU	77.4%	6.4%

Table 1: Accuracy in training and test sets

We note that the simple RNN performed poorly by predicting only 2.6% from the training set and 1.8% from the test set. The LSTM did well compared to the GRU by predicting correctly 89.3% of the words in the training set and 8.5% in the test set.

These results show that the four-gates structure of the LSTM allows a better learning and a higher dependency with previous words which is very important in learning the context of each sentence and predicting a word that is associated to this context. The GRU also performs well because of its two-gate architecture that offers a better

dependency with past words when compared with the vanilla RNN that only considers the last hidden state to predict the next word.

We note that the overall performance of the three models is poor over the test set which is a set of sentences that were not included in training. This may suggest that the current configuration was not able to learn enough from the corpus to make more accurate predictions. Another hypothesis is that the complexity of the corpus is high and the three models were not able to generalize the training over the whole Quran corpus.

Assessment 2: Evaluation of BLEU scores

The BLEU scores calculated over unigrams, bigrams and trigrams for the three models are shown on the following tables.

	BLEU-1	**BLEU-2**	**BLEU-3**
Simple RNN	0.030	0.173	0.349
LSTM	0.620	0.507	0.461
GRU	0.533	0.395	0.342

Table 2: BLEU score for 1000 samples from the training set

	BLEU-1	**BLEU-2**	**BLEU-3**
Simple RNN	0.026	0.162	0.336
LSTM	0.435	0.333	0.304
GRU	0.368	0.241	0.206

Table 3: BLEU score for the test set

We note the same observations as for the prediction accuracy assessment. The RNN performed poorly while the LSTM showed the best results. Besides, the performance of all models is lower on the test set than on the training set which is normal.

Nevertheless, using the test set, we note that the performance of the LSTM model is better in terms of the BLEU scores than in terms of prediction accuracy. This can be explained as follows using an example.

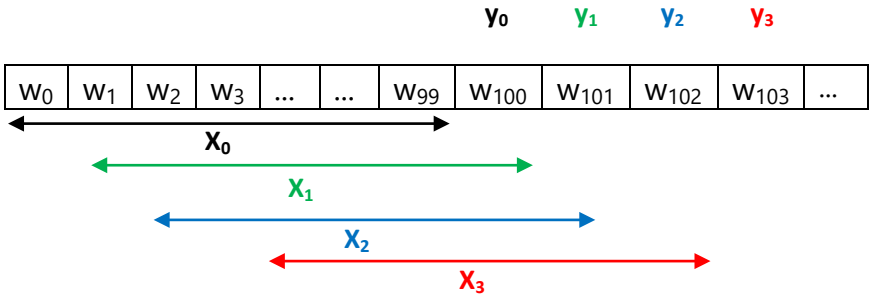

Figure 12: Explaining why LSTM performs better in terms of BLEU score than accuracy score

In Figure 10, let's suppose that we picked randomly for our training set the tuples (X_0, y_0), (X_2,y_2) and (X_3,y_3) and we picked the tuple (X_1, y_1) for our test set. In reality, those four tuples share more than 95% of the words; only one word changes between a tuple and the next one. Therefore, during training, LSTM learns the long term dependency between words by training on (X_0, y_0), (X_2,y_2) and (X_3,y_3). And even though it may predict the word for the test tuple (X_1, y_1) incorrectly, if we go on predicting the word following y_1, the LSTM may predict it correctly as it is already trained on (X_2,y_2). Thus, the LSTM can catch up and continue predicting the rest of the sequence correctly even if it fails to predict the word belonging to the test set. Such long term dependency is lower with the GPU and much lower with the vanilla RNN structure that's why the LSTM outperforms both of them.

We can find below an example of a generated sequence predicted by the LSTM from the test set and compared with the same sequence in the corpus:

Correct sequence from the corpus

من	لا	يؤمن	بها	واتبع	هواه	فتردى	وما	تلك	بيمينك

Sequence generated by the LSTM

لا	لا	يؤمن	بها	واتبع	هواه	فتردى	وما	تلك	بيمينك

As explained above, the sequence is generated by the LSTM by applying an input vector from the test set. Therefore, the first predicted word has a higher probability of being incorrect (coloured in red). When we carry on predicting, the following words are correct, and this is explained by the long term dependency learned by the LSTM.

Difficulties found during the project

During this project, many problems were found and required long hours of analysis and careful examination. Here are the main ones:

- The language of the corpus is Arabic; thus all used functions must use UTF-8 encoding to interpret the different words correctly. A special problem arose with the matplotlib package that does not plot Arabic words correctly, and that was needed in the exploratory analysis part. The solution was to use **arabic_reshaper** and **bidi** packages for proper formatting.
- Training the model takes very long hours. For example, with a 4-cores i7 desktop computer, one epoch of training with the LSTM model takes up to 30 minutes. For 100 epochs, the needed time is almost 2 days. Therefore, the use of GPUs is highly recommended. The training time with 4 GPUs was divided by 10. This allowed to fine tune the model for different parameters more quickly.
- Even though BLEU score is interesting to assess the quality of the generated sentences, it focuses only on precision. For future implementations, it would be preferable to use additional metrics such as ROUGE that focuses on recall and METEOR that considers the order of words in the generated sentence as well.

Conclusion

This project aims to learn from sequences of words of an Arabic corpus that is the sacred book of Quran and create models able to predict a text that is similar to it. The main idea is to divide the corpus into groups of 100 words and train a recurrent neural network to predict the next word that follows each group. The project covered three different recurrent neural networks namely the vanilla RNN, the LSTM and the GRU.

The used evaluation metrics are prediction accuracy and BLEU scores. The results showed that the vanilla RNN performed poorly on training and test sets. The LSTM and GRU performed better as they use a mechanism of gating to memorize the long term dependency with past words. The LSTM performed the best even though the prediction accuracy with the test set was low. This may be explained by the semantic complexity of the corpus that the LSTM and GRU were not able to learn.

The three networks were also assessed in terms of BLEU score. In this context, each network predicts a sequence of 10 words from a vector of 100 words; the generated vector is compared with the correct sequence of 10 words existing in the corpus. In this second assessment, the vanilla RNN did poorly as well; on the other hand, the LSTM and GRU showed good performance, the LSTM still performing better. Indeed the similarity between the 10-words sequence generated by the LSTM had 43% of similarity at unigram level compared the original text and 30.4% of similarity at

trigram levels. This observation shows that even if the LSTM predicts the first word incorrectly in the sequence, its long term dependency with past words allows making better predictions of subsequent words.

An interesting prospect of this work is to test the GAN (Generative Adversarial Network) structure to generate sentences from the same corpus by using another concept based on coupling a generator with a discriminator. I tried actually to train this structure using an LSTM in both generator and discriminator models. Unfortunately, I was not able to make this architecture converge. The source files are provided in my github account.