Short-time DNS queries forecasting from the users' and servers' points of view, using deep learning and a pre-trained word embedding

Jorge Merlino and Pablo Rodríguez-Bocca
Instituto de Computación, Facultad de Ingeniería, Universidad de la República,
Montevideo, Uruguay, 11300,
jmerlino,prbocca@fing.edu.uy

Abstract

Word embeddings are used in natural language processing to group semantically similar words. In this paper, we create word embeddings for Internet Domain Names (DNS) from corpora of anonymized DNS queries from an Internet Service Provider. We use each embedding as a layer of a recurrent neural network (RNN) that works as a Language Model for the DNS queries generated by the users. We use these RNNs to predict the next DNS query in two different cases. A first case tries to predict the next domain query from the DNS server's point of view so the corpus is close to the original log data (see [1]). A second case tries to predict the next domain queried by a user from the user's point of view. Here the corpus has larger preprocessing.
We show that this procedure has good accuracy for the DNS server-side problem, but low accuracy for the user-side problem. Moreover, we show that training the same RNN without using the pre-trained embedding takes more time and is substantially less accurate. These results have practical applications for the service's latency reduction, cache optimization in recursive DNS servers, automatic filtering of inappropriate domains, and detecting anomalies.

Keywords: DNS, Word Embeddings, fastText, Recursive Neural Networks, Language Models, Natural Language Processing.

1 Introduction

As the time people spend online has been increasing through the years [2] the understanding of their behaviour has been the focus of academic research. Such results are useful for the media industry [3] and for network applications.

Word embeddings map words to vectors so that words that are close in text are represented by nearby vectors. This is a much better representation than simple one-hot encoding for example. Word embeddings are normally used in natural language processing (NLP) tasks as shown in [4,5]. Recently, word embeddings have been applied to DNS queries in order to extract useful information from this raw data. Examples of this are [6–10]. In our DNS vector embedding we consider the domain names queried after and before some domain name as the context for this domain (i.e. the trace of DNS queries). In the case of the traffic requested by users we group the traffic by IP address so that the context are the domains requested after and before by the same IP address. As a novel application we use our vector embedding to build a neural network that predicts the next domains to be queried to a DNS server by its clients.

This paper presents an application and an extension of our pioneer work on this topic (mainly represented by [6, 7]), where we assess the methodologies for creating embedded spaces, but we do not apply the spaces to a specific problem (as in this work). In particular, in order to create our embedding, we follow the main ideas and results described in [7]. For this work we obtained DNS recursive server logs from a large Internet Service Provider (ISP) with anonymized IP addresses. These logs contain each query resolved by a server. Each line of log indicates the time, the anonymized IP address of the client, the queried domain, and the type of DNS query (A, AAAA, MX, etc.). To create the word embedding for the domains we use the `fastText` [11] model, following the procedure described in [7]. We group the queries by source IP address, build the embedding and use it as an embedding layer in a recurrent neural network to map the input domains into a lower dimension space. Then we train this RNN with the DNS data obtained from the ISP and test the network trying to predict future queries.

At least, there are two points of view in the problem of predicting the next domain name to be queried. From the DNS server's side, it is important to accurately predict the next query that a user will require. This can be used to optimize the performance of the DNS resolver by prefetching domains that are predicted to be queried next and thus reducing the latency time of the future answer and, along the same lines, by not expiring (or refreshing) cached queries that are expected to be queried again in the future. The predictions of the next queries could also be used to detect traffic anomalies when the expected traffic does not coincide with the traffic observed on the server (for example because some users are disconnected from the network or are infected by malware). On the other side, from the user's point of view, it should be interesting to understand the behaviour of each user on the Internet (as we did in [7]). This could be used to provide the user with suggested sites for browsing next and also to help parental control or advertisement blocking algorithms. In this paper, we evaluate the two points of view of the problem, trying two different approaches at the time of processing the DNS data and building the embedding. The first one tries to predict the DNS queries as seen by the server so the logs are not processed besides grouping by IP address. In the second, we try to predict the next queried domain from the point of view of the client, so we drop repeated consecutive domains and preprocess the DNS addresses to keep just the last part as, from the view of the client, `apps.facebook.com` and `pixel.facebook.com` correspond to the same domain (`facebook.com`) even if they actually resolve to different IP addresses. With the first approach, from the server's point of view, we show that the created embedding effectively predicts future queried domains and that gets better results than a baseline algorithm or training a new embedding layer. This is the main contribution of this paper and was presented in [1]. For this journal version, we added the second approach, where we see that the behavioural prediction problem is more difficult, and we obtain low predictability compared with the baseline.

The rest of the paper is organized as follows: Section 2 describes the state of the art of the problem. It introduces previous work in DNS prediction and in the use of embeddings with DNS queries. Section 3 explains the Domain Name System on the Internet, its particularities for our problem, and the data used in this work. Section 4 introduces our methodology to predict the next DNS queries, including the basis of the `fastText` algorithm, and the deep learning Recurrent Neural Network architecture used. Section 5 shows experiments and results of our methodology applied to the problem from the server's side point of view, and Section 6 shows experiments and results from the user's point of view. Finally, Section 7 summarizes conclusions and discusses future work.

2 State of the art: Internet Domain Names Prediction and Use of Embeddings

There are a few papers that try to predict DNS traffic in order to detect anomalies. For example in [12] they use clustering to detect different patterns of traffic for each cluster. They group data by hour and only

recognize increases or decreases of the number of accesses to a domain per hour and assume an anomaly occurred if the rate of accesses does not match the trend. This is a much cruder approach than ours as we can can extract information with source address granularity. In [13] they use neural networks to predict time series of aggregated DNS traffic and notify an anomaly when traffic does not matches the predicted behavior. They again group the traffic by hour and group packets by query type losing source address information. Also in [14] the goal is to use machine learning in order to detect machine generated domain names used by malware command and control (C&C) servers. In this case they do not use real DNS data but train the system with domains extracted from Alexa (which is a service that labels domains with metadata information) and use it to compare with randomly generated domains.

On the other hand, there are a few papers that use embeddings to extract information from DNS queries. The first paper that documents this approach [6] is part of our previous work on this topic; where an embedding is built using the `Word2vec` algorithm and then is used to find semantic similarities between domains (likeness in the meaning of their content). The DNS queries are grouped by source IP address and the domains queried by the same host in a window of time are grouped in a "sentence". These sentences are used to create the embedding (which is an unsupervised process). The assumption is that the distributional hypothesis that applies to text can also apply to domains; so that domains that are close in the list of queried addresses must be semantically similar. The embedding algorithm assures that these domains that occur closely will be represented by vectors that are also near each other in the embedding. Later, in [7] we improve the embedding technique (but we did not extend its practical applications). In those works, we compare different embedding algorithms and concluded that the best result is obtained using the `fastText` algorithm. That is the reason why it is used in this paper, where its methodology is partially recreated.

Other derived works are for example [8] where the embedding is used to detect generated domain names as mentioned before. In this case they label some of the domains using public black and white lists, then preprocess the data removing ccTLDs and grouping by source address as in the previous case and build the embedding using `Word2vec`. Then they train a logistic regression classifier using the previously labeled data as ground truth and the embedding as input. This classifier is used to test if some domain is valid or just randomly generated. The assumption again is that invalid domains will be close on the embedding so that they will be recognized by the trained classifier. They also use an incremental `Word2vec` model that can be updated online as new domains are received. The classifier is also updated by gradient descent.

In [9] the same idea of [6] is repeated with different hyperparameters and data preprocessing. And in [10] the authors analyze the DNS queries generated by clients to infer which of them are IoT (Internet of Things) devices. They define their own embedding algorithm that is based on statistical methods instead of using neural networks. The embedding is constructed by building a matrix of conditional probabilities for pairs of domains and using its columns as vectors. The input DNS data (where the pairs of domains are searched) is also grouped by source address as in the previous cases. This embedding is used as the first layer in a RNN that makes a binary classification between IoT or regular clients. The neural network is trained by existing labeled data of DNS queries made by IoT devices and other clients.

At last, in [15] the same embedding as [6] is built using `Word2vec` in order to obtain semantic similarity between domains. A multi-class classifier (one-vs-rest linear support vector machine) is trained to detect the category (games, shop, mail, etc) of each domain. Also some examples are shown of similar domains being near in the constructed embedding. The hyperparameters are different from [6, 7].

3 Domain Names on Internet (DNS)

In this section, we will explain the Domain Name System on the Internet, the data used in this work and its particularities to our problem. To do these, we will start with our introduction in previous work:

> "The DNS (Domain Name System) [16, 17] is a decentralized service for naming computers and other resources in a network. Each of these resources is assigned a domain name which is a hierarchical string defining a node in a tree structure. The domain name is formed by the labels of the tree nodes traversed from the node to the root separated by points. For example the domain name *example.com* has *com* as its top level domain, and *www.example.com* as a sub-domain. The DNS system is decentralized as the responsibility for resolving each component of the domain name is delegated to a different name server thus avoiding a single central database and a single point of failure. Also there could be several domain servers to resolve the same domain providing thus a fault tolerant configuration. This system has been in use in the Internet since 1985 and is one of the most essential services in the network. In the Internet the most fundamental service provided by DNS is to translate easily memorized domain names to IP addresses. Each domain can have different sub-domains with different type. The most common types are: A and AAAA that correspond to IPv4 and IPv6 address translation respectively, MX that define SMTP mail

```
21-Mar-2013 06:06:47.216 client <anonymized ip>: query: apps.facebook.com IN A
21-Mar-2013 06:06:48.149 client <anonymized ip>: query: profile.ak.fbcdn.net IN A
21-Mar-2013 06:06:53.215 client <anonymized ip>: query: pixel.facebook.com IN A
21-Mar-2013 06:08:10.728 client <anonymized ip>: query: jacaranda.ceibal.edu.uy IN A
```

Figure 1: Example of some DNS queries from an anonymized IP address one day

exchangers, NS that define other name servers, CNAME that define domain name aliases and PTR that are used for reverse DNS queries (for example to query the domain name for a given IP address).

Each domain has at least one *authoritative* name server that contains the original information about the domain and its sub-domains. An authoritative name server only gives answers to DNS queries from data that has been configured on that server. Potentially, an authoritative name server could delegate a sub-domain to other authoritative servers building a hierarchical tree of authorities. On the top of the hierarchy are the root DNS servers.

On the other hand, there are *recursive* DNS servers that are capable of resolving queries about domain names, by means of recursive queries to possibly several authoritative name servers starting from the root servers. These servers usually cache the results obtained to increase efficiency. The duration of the cached data depends on the TTL (time to live) configuration of each domain at the authoritative servers.

At the user side (in the operation system of the computer or device), is the DNS *resolver*. Resolvers usually query recursive servers to find an answer, and also cache the result during the corresponding domain TTL.

DNS is critical for Internet operation. It is a large and complex system. Here we include a minimum description with the aim of offering a self-contained reading. You can read [18] for a deep explanation." From p. 3 of [7].

3.1 DNS traces: our corpus data

The data used for this research was provided by a large Internet Service Provider (ISP), with millions of Internet users, who use the ISP's recursive name servers. For this study we analyze the queries log in one of these name servers, at March 21$^{\text{th}}$, 2013. Note that the amount of data collected in just one day is extremely large. With more than 148 million of queries, 3.5 million of unique domain names, and 550 thousands of unique IPs. Data across different days are pretty similar, with a little trend with time, where A and AAAA record types correspond to more than 90% of the queries. Please read [7] for a piece of detailed descriptive information about this dataset and extensions.

Figure 1 shows an example of some DNS queries from an anonymized IP address. Each line of the DNS log shows the date and time of the query, the anonymized IP address of the client, the queried domain and the type of DNS query requested by the client. Formally, the log data is a sequence of $< \text{IP}_i, d_j, t_k >$, where the client IP_i queried the domain d_j at time t_k. In the log, there is a set of unique (anonymized) IPs representing each client, $c \in C$, for each client we have a DNS trace, $t_c \in T$, which is an ordered sequence in time $t_c = \{< d_1, t_1 >, < d_2, t_2 >, \ldots\}$ queried by the client.

As we want to predict the domains being queried we want to use an embedding layer that maps domains that are usually queried together into nearby vectors. This problem is very similar to mapping semantically similar words together, where the trace of words are the phrases in texts. Following previous results in natural language processing, word embedding techniques like `fastText` are shown effective for these kind of problems [7].

It is important to note that the DNS traces do not correspond exactly to the sequence of sites visited by the clients. To take into account:

1. It is usual that DNS *resolvers* cache the results of the resolved domains, so that the DNS server only get one query per domain even if it was queried multiple times. Thus we do not have information about the period of use of each domain.

2. It is very common in business and residential connections (but not in mobile services) to have a NAT enabled modem. This NAT hides the activity of many users behind a single public IP address. And therefore, this IP's trace can mix multiple clients (perhaps thousands).

3. Another typical technical limitation is the dynamic IP address assignment in residential and mobile services (as our ISP does). In these services, the client is automatically reconnected after a fixed period (for example 12 hours), where a new IP is assigned. Therefore, an IP identifies a service for a period, and a trace can be the concatenation of session periods from several services. This problem can be avoided when session information (the log about IP assignment to services) is available. Unfortunately, it is not our case.

4. It is usual that Content Providers use several sub-domains to provide their content. Moreover, they normally use external services to provide part of their content (for example they use Content Delivery Networks to provide static and video content). Therefore, when a client accesses a service, it usually adds several domains to his trace (sub-domains of the provider and external sources). These sub-domains are very common and are used to add redundancy and load balance. When they exist, it is relatively easy to predict that the next domain will be one of them, but usually, it is hard to predict which of them will be used next.

5. Also there can be several applications in a client device that generate Internet traffic and thus generate queries to the DNS servers (for example email clients or music players). Because of this the DNS traces are not exactly the list of domains accessed by a user.

From the DNS server's point of view, these limitations are irrelevant. Therefore, in order to generate the corpus for this case, the only preprocessing we do is use only A and AAAA queries (as they are the ones we try to predict), and listing the domains queries by each IP address. But when we try to predict the next domain queried from the user's point of view, these limitations are very important and some mitigation measures have to be taken into account in order to generate the corpus. In this case, the mitigations using in the preprocessing include:

- Use only a subset of IP addresses that belong to residential and mobile users. This way we are limiting the number of different users using the same public address and as such mitigating limitation number 2

- To reduce heterogeneity in the domain names and consider them as seen by the user we truncate the domains names following this algorithm: if the top label is a country code (ccTLD) we take the first three levels of the domain (eg. `google.com.uy`), in other case we take the top two levels (eg. `facebook.com`). With this we try to mitigate limitation number 4.

- We consider that sequences of the same domain in the traces do not provide useful information. They can be originated by clients whose resolvers do not cache queries or as a side effect of the previous mitigation as multiple request of subdomains of the same main domain are all converted to the same base domain. So we convert a sequence $< d1, d1, d2, d2, d2, d1 >$ into $< d1, d2, d1 >$.

These mitigations are based on the ones used in our previous work [7], where our main goal was to identify semantically similar domains.

In both corpora, we also filter the traces to consider only the most popular domains. As we will see in Sections 5 and 6, filtering to the top domains has no large impact on prediction performance because the most popular domains largely concentrate the majority of queries.

It is important then to note that the two approaches tested in this paper use very different corpora and therefore use different embedding as each of them is generated from each corpus in a non-supervised manner.

4 Prediction Using Deep Learning and Word Embedding Techniques

In this section, we present our method to predict DNS traffic, but previously we will introduce the basis of word embedding techniques and recurrent neural networks.

4.1 Word Embedding Basis

The first computationally-efficient predictive model for learning word embeddings was `Word2vec` [19–21]. `Word2vec` has two variants: Continuous Bag-of-Words (CBOW) and Skip-Gram. In CBOW the model is trained to predict target words from source context words, while the Skip-Gram model does the inverse and is trained to predict source context words from the target words. Both have shown to be useful for extracting similarity of words in a text but in general Skip-Gram outperforms CBOW.

The `fastText` algorithm [11] is an extension of the Skip-Gram model that represents each word as a sum of n-grams that consider subword information. Word boundaries are marked by special symbols that are

inserted in order to distinguish subwords from complete shorter words. For example in [11] it is shown that for $n = 3$ the n-grams for the word $<where>$ would be $<wh$, whe, her, ere and $re>$. This way it is possible to distinguish the word $<her>$ from the subword her. Usually the algorithm receives two parameters that limit the length of the subwords considered. For example if they are configured as $(3, 6)$ only the n-grams of length between 3 and 6 will be considered. This representation provides support for words out of the vocabulary as some of the n-grams for the word might appear even if the word itself does not. As we can not possibly work with all possible domains this allows us to make predictions following domains that are similar but not appear exactly as such in our vocabulary like `example.com` and `example.org`.

Now, we will explain the Skip-Gram model in detail, and later we will extend it to explain the `fastText` model. As mentioned, the objective of the Skip-Gram model is to create an embedding useful to predict context words from source words. Thus, given a sequence of T training words as $w_1, w_2, \ldots, w_T$, the Skip-Gram model objective is to maximize the average log-likelihood:

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0}\log p(w_{t+j}|w_t) \tag{1}$$

where c is the size of the context training window. The number of possible training samples increases with c. The $p(w_{t+j}|w_t)$ probability function is defined as the softmax function:

$$p(w_i|w_j) = \frac{\exp\left[\text{score}(w_j, w_i)\right]}{\sum_{w=1}^{W}\exp\left[\text{score}(w_j, w)\right]} \tag{2}$$

where W is the size of the vocabulary. The $score(w_i, w_j)$ is a measure of the compatibility of w_i and w_j. Usually the inner product of their vector representation is used, i.e. $score(w_i, w_j) = v_{w_i}^{\top} v_{w_j}$. But, the computation of (2) is usually impractical as the cost of calculating $\nabla \log p(w_i|w_j)$ is proportional to W which is usually large (order 10^4 in our case). To overcome this problem, we instead use the Noise Contrastive Estimation (NCE) which approximately maximizes the log of (2). The objective of NCE is:

$$\log \sigma({v'_{w_i}}^{\top} v_{w_j}) + \sum_{w=1}^{k}\mathbb{E}_{w\sim P_n(w)}\left(\log \sigma({v'_w}^{\top} v_{w_j})\right)$$

where v_w and v'_w are the embedding vectors of w after and before the maximization step, ${v'_{w_i}}^{\top} v_{w_j}$ is the inner product and σ is the binary logistic regression function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

The idea is to differentiate the word w_i from draws from the noise distribution $P_n(w)$ using logistic regression, where there are k noise (negative) samples for each data sample.

Several variants of this simple algorithm have been developed in the last years. Please read [19] for a deep explanation of the algorithm and [22] as an example of extension to a paragraph vectorization method (instead of word vectorization). `FastText` uses the same equations as the Skip-Gram model except for the score function in (2) that only predicts one context word. Instead let G be the set of n-grams for word w (including w itself) and v_g the vector representation for each $g \in G$. Thus, in [23] the vector for each word w is represented by the sum of the vector for the complete word and the n-gram vectors as:

$$\sum_{g\in G} v_g$$

and the scoring function is:

$$\text{score}(w, w_j) = \sum_{g\in G} v_g^{\top} v_{w_j} \tag{3}$$

where v_{w_j} is the vector representation of word w_j.

As mentioned before, several n-grams of different size are considered for each word in addition to the word itself including the boundary symbols. The score includes the sum of all these n-grams.

4.2 Recurrent Neural Networks (RNN)

In a simple feedforward neural network, the information flows through a series of operations performed on the nodes without a feedback loop or any consideration of the order of the data. Therefore, output at time $t+1$ only depends on the input at time t so it is hard to model dependencies of the value on $t+1$ on earlier

input values. However, in our case we have sequential data in the form of the DNS traces and the order of the domains in the data is important as the next value does not depend only on the last value but in all previous context.

The Recurrent Neural Networks (RNN) [24, 25], are a form of neural network designed specifically for sequential data. They take into account the sequence order; that means that the state of the network is impacted, not only by the current input but also by their recent past. The reason that RNNs are good at sequence-related problems is that they are good at retaining important information from the previous inputs and using this past context information to modify the current output. In order to achieve this, the RNN architecture allows time-delayed connections among hidden units, enabling it to discover temporal correlations on the sequential data.

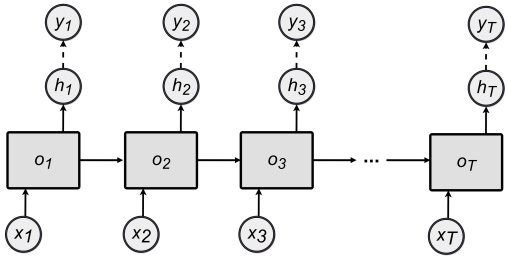


Figure 2: Fully Recurrent Neural Networks (FRNN) architecture unrolled in time by creating a copy of the model for each time step. Where x_t is the input of the RNN at time t, o_t is the hidden state at the same time, and y_t is the output at the same time

Following the description in [26], Figure 2 displays the Fully RNN architecture unrolled in time (for simplicity) by creating a copy of the model for each time step. Fully RNN architecture is the most general RNN architecture, where other RNN architectures limit the connections between neurons in some way. Equations (4) provide the mathematical representation of a single FRNN network cell, for a given step t.

$$\begin{aligned} o_t &= W_o \times [x_t, h_{t-1}] + b_o, \\ h_t &= \tanh(o_t), \\ y_t &= \text{softmax}(W_y \times h_t + b_y), \end{aligned} \tag{4}$$

where x_t is the input of the network at timestep t, o_t is the hidden state of the network at the same timestep, and y_t is the output at the same timestep. Usually, the output at the final timestep h_T is processed by the softmax activation function in order to produce a probabilistic output, y_T.

RNNs are harder to train than regular networks due to the vanishing gradient problem [27], especially when long term patterns in the data are needed. The first RNN implementation which obtained good results was the Long Short-Term Memory (LSTM) [28] cells. LSTM addresses the problem of training over long sequences and retaining memory by introducing a few more gates that control access to the cell state. The new cell structure helps to maintain a more constant error so that it allows recurrent nets to continue to learn over many time steps (which sometimes can be over 1000). The RNN cell will only output the hidden value, while the LSTM cell will output the hidden value with new cell states. Figure 3 describes the LSTM architecture of a cell, while Eq. (5) provides its mathematical representation where $\circ$ represents the element-wise product.

$$\begin{aligned} i_t &= \sigma(W_i \times [x_t, h_{t-1}] + b_i), \\ f_t &= \sigma(W_f \times [x_t, h_{t-1}] + b_f), \\ a_t &= \tanh(W_a \times [x_t, h_{t-1}] + b_a), \\ c_t &= f_t \circ c_{t-1} + i_t \circ a_t, \\ o_t &= \sigma(W_o \times [x_t, h_{t-1}] + b_o), \\ h_t &= o_t \circ \tanh(c_t). \end{aligned} \tag{5}$$

In addition to incorporating the previous output and the current input to generate the output, LSTM differs from a typical RNN by keeping the hidden state information, incorporating it to generate the output, and updating the new cell states. The key new component is the cell state c_t, which is capable of storing information over long periods of time. At each timestep, the update of the cell state is a complex process that involves: the forget gate f_t that controls the flow of information from the previous cell state c_{t-1}, the cell activation a_t that introduces new information, and the input gate i_t that controls the flow of new

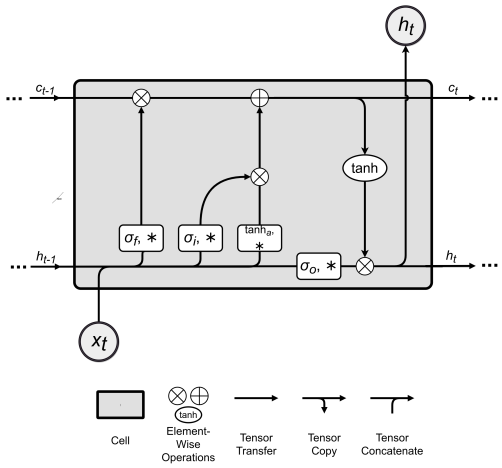


Figure 3: Long Short-Term Memory (LSTM) network architecture. Where x_t is the input of the LSTM at time t, o_t is the hidden state, and c_t is the cell state. This cell architecture can be unrolled in time in a similar way than Figure 2

information. The hidden state h_t is obtained by combining the output signal o_t and the cell state c_t. And again, the output at the final timestep h_T is processed by the softmax activation function in order to produce a probabilistic output, y_T.

In summary, the current output of RNN is decided by two terms: the current input and the previous output. While the current output of LSTM is decided by three terms: the current input, the previous output, and the previous state.

We use a variaton of LSTM cells called Gated Recurrent Units [29] or GRUs that are simpler than LSTMs. Figure 4 describes the LSTM architecture of a cell, while Eq. (6) provides its mathematical representation.

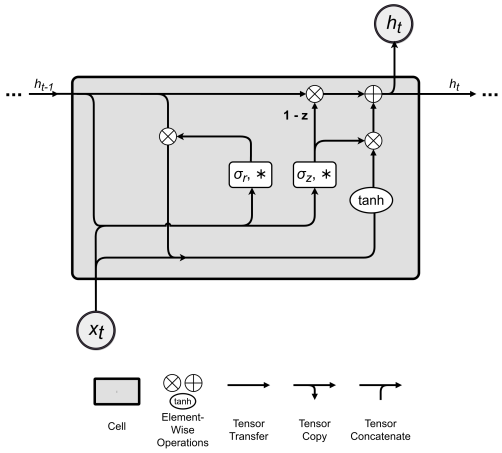


Figure 4: Fully Gated Recurrent Unit (GRU) network architecture. Where x_t is the input at time t, z_t is the update gate, r_t is the reset gate, and h_t is the output vector. This cell architecture can be unrolled in time in a similar way than Figure 2

$$
\begin{aligned}
z_t &= \sigma(W_z \times x_t + U_z \times h_{t-1} + b_z), \\
r_t &= \sigma(W_r \times x_t + U_r \times h_{t-1} + b_r), \\
\hat{h}_t &= \tanh(W_h \times x_t + U_h \times (r_t \circ h_{t-1}) + b_h), \\
h_t &= (1 - z_t) \circ h_{t-1} + z_t \circ \hat{h}_t,
\end{aligned}
\tag{6}
$$

4.3 Artificial Network architecture to Predict the next DNS queries

In order to predict the next DNS queries, we will use an RNN architecture with three layers: an embedding layer as the input, a hidden layer of neurons (LSTM or GRU), and an output dense layer. See Figure 5.

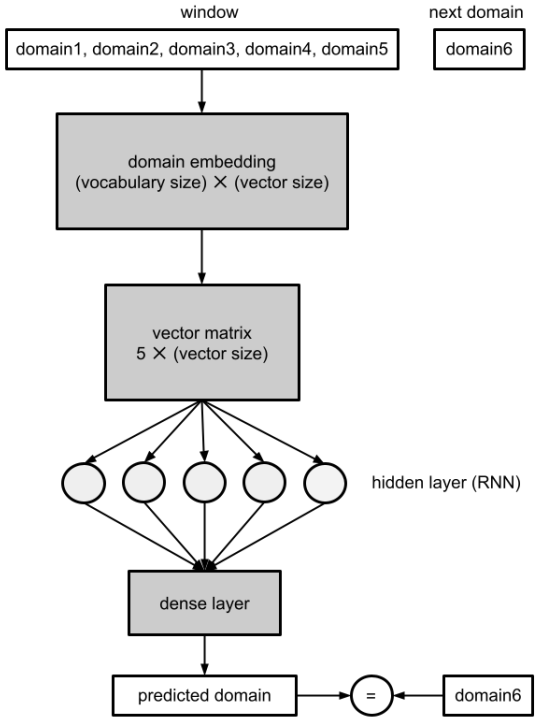


Figure 5: Artificial Network architecture to Predict the next DNS queries. It has three layers: an embedding layer as the input, a hidden layer of neurons (LSTM or GRU), and an output dense layer

Each training sample fed to the system is composed initially by a set of consecutive domains and the next domain is used to evaluate the loss. The first embedding layer is used to map these domains into vector representations. Early approaches used the One-Hot encoder which is based on a bag-of-words representation. Thus, each word is represented as a vector with dimensionality equal to the number of unique terms in the vocabulary. Only one of these dimensions could take on the value 1 and the others are 0. The vectors that represent the domains with the One-Hot encoder will be mostly empty. This representation is too sparse to provide the intrinsic relationship between domains. In order to solve this problem we can use an embedding layer that transforms the input into vectors on any desired size keeping related domains closer.

This embedding can be trained as part of the RNN training or can be left with fixed vectors. Also it can be initialized with a precomputed embedding (which can also be trained further) or can be initialized with random values and trained from scratch. In Sections 5 and 6, we test these three approaches using the `fastText` embedding of domains to initialize the embedding layer when appropriate:

- `fastText` embedding: the embedding layer is initialized with the pre-trained `fastText` embedding, and it is not modified during RNN training;
- `fastText` re-trained embedding: the embedding layer is initialized with the pre-trained `fastText` embedding, and it is re-trained with the RNN architecture; and
- new embedding: the embedding layer is initialized with random values, and it is trained with the RNN architecture.

The output of the embedding layer is a matrix composed by one vector per domain in the window. We also convert the next domain to a vector which is later compared with the network prediction in order to train it by back propagation.

The next layer is the hidden layer of the RNN itself. We test different number of neurons and also different cell gates (LSTM and GRU) and the use of dropout to avoid overfitting. Each neuron receives the matrix of domain vectors and generates a new vector as prediction.

The last layer is a dense layer that mixes the outputs of each neuron (of previous layer) into a single vector output that is compared with the next domain vector computed before. The comparision is computed using a loss function (we test mean squared error and mean absolute error). We also test a few different activation functions on this layer (linear and relu).

During the testing phase, we get a vector predicted by the RNN that does not necessarily match exactly with any of the domain vectors in the embedding. What we do is to find the nearest vector in the embedding to the predicted vector. For this, we use the cosine distance in the vector space.

It is important to note, that we use the described architecture to predict the next DNS queries in both scenarios: from server's and user's points of view, where the word embedding changes because the different corpora used, and many other hyperparameters, as explained next, in result sections.

5 Experiments and Results from the server's point of view

In this work, we use the DNS logs of a day (March 21th, 2013) as input of the training process, with almost 148 million of queries. From these queries, we get the DNS traces, grouping by IP address and removing the less popular domains. As expected, there is a large variation of popularity between domains. Figure 6 shows

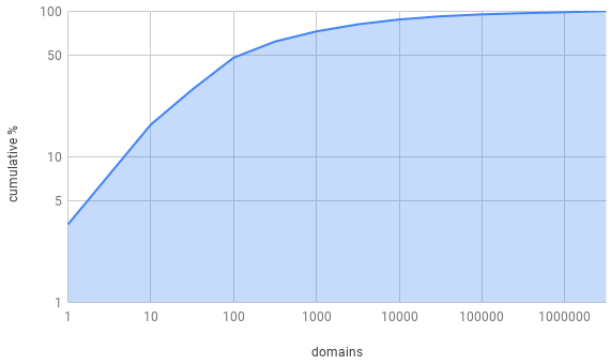


Figure 6: Popularity of domains, i.e. 100% cumulative percentage of requests per domain (log axes)

the cumulative percentages of requests per domain, where domains are represented by popularity position (from left to right) on the x-axis. We can see that top 3 thousand domains accumulate 80% of the total traffic, top 100 domains accumulate the 50% and top 10 domains accumulate almost 17%. Considering this large asymmetry in popularity between domains, we decide to filter the traces and take only the 40000 more popular domains (92,6% of the queries). The impact of this in prediction is low, but the impact on training performance is very high. As result, our corpus has 33479 DNS traces (different IP's addresses), with more than 136 million queries, of 40000 unique domains. This corpus will be used to train our word embedding technique.

5.1 Pre-trained Word Embeddings

For the embedding, we train the `fastText` algorithm with the corpus. We use the official C++ implementation from `https://fasttext.cc`. We create several embeddings of size 64 with different subword sizes as shown in Table 1. This is a really fast process taking only about 14 minutes for each embedding in a Intel(R) Core(TM) i5-7400 CPU @ 3.00GHz. The process uses all four cores of the processor and requires about 1.6 GiB of resident memory.

Table 1: Pre-trained Word Embeddings using the `fastText` algorithm with 64 dimensions, and several n-grams options

Name	Method	Dimension	n-grams
`ft-64-3/6`	`fastText`	64	3/6
`ft-64-7/13`	`fastText`	64	7/13
`ft-64-10/13`	`fastText`	64	10/13
`ft-64-11/17`	`fastText`	64	11/17

We next show in Table 2 some examples of the nearest vectors to some common DNS names. It can be seen that they are related domains that usually are queried next to the principal domain. These examples are taken from the `ft-64-7/13` embedding, similar results are obtained from the other options.

Table 2: Nearest vectors to some common domains in the `ft-64-7/13` embedding

www.facebook.com	
Nearest	pixel.facebook.com
Second nearest	profile.ak.fbcdn.net
Third nearest	developers.facebook.com
www.mercadolibre.com.uy	
Nearest	listado.mercadolibre.com.uy
Second nearest	articulo.mercadolibre.com.uy
Third nearest	home.mercadolibre.com.uy
www.gmail.com	
Nearest	mailattachment.googleusercontent.com
Second nearest	accounts.youtube.com
Third nearest	accounts.google.com.uy
www.youtube.com	
Nearst	i2.ytimg.com
Second nearest	i3.ytimg.com
Third nearest	s.ytimg.com

5.2 Best architecture and Hyperparameter tuning

Now, we can build the RNN network as explained in Section 4.3. The network has three layers (an embedding layer as the input, a hidden layer, and an output layer), where several hyperparameters can be tuned. In summary, we chose as hyperparameters: one of the four embeddings of Table 1, the number of neurons in the hidden layer, the activation function of the neurons, the dropout probability (turning off some neurons on the hidden layer during training to avoid overfitting), the loss function, the optimizer method, the window size and the number of epochs. We use this same hyperparameters in combination with LTSM and GRU neurons. The different options tested are shown in Table 3. Each possible combination of hyperparameters is implemented using the Keras API [30] in the TensorFlow backend [31], and is trained on a multi-core supercomputer housed at Uruguay's National Center for Supercomputing (Xeon Gold 6138 processor with NVIDIA P100 Tesla GPU)[1].

Table 3: Hyperparameters options tested

Hyperparameter	Options
type of neurons on the hidden layer	GRU and LSTM
embedding name	ft-64-3/6, ft-64-7/13 ft-64-10/13 and ft-64-11/17
number of neurons on the hidden layer	150, 100 and 50
activation function	linear, relu
dropout	0 and 0.2
loss function	mean squared error, mean absolute error
optimizer	ADAM, RMSPROP
training window size	10 and 5
number of epochs	100, 50 and 25

The hyperparameter tuning takes more than 200 hours in different runs. We use 70% of data for training and 30% for testing. And the best result (the option that reaches the lower loss value) is obtained using the `ft-64-7/13` embedding, 100 GRU neurons, no dropout, linear activation, mean absolute error loss function, RMSPROP optimizer, a 10 domain window, and 50 epochs.

The different hyperparameters are compared using the accuracy obtained considering the nearest vector in the embedding to the vector predicted by the RNN, and also the accuracy if the second or third nearest vectors are considered. In Tables 4(a) and 4(b) we show the results of some combinations of parameters for the two arquitectures under study (GRU and LSTM) and show the difference with respect to the best result obtained globally. We can see that in general LSTM obtains lower results than GRU for most hyperparameter combinations.

[1] Centro Nacional de Supercomputación (ClusterUY), https://www.cluster.uy/, 2021.

Table 4: Comparison of hyperparameters

(a) GRU neurons

Hyperparameters	Difference
ft-64-10/13, 100, GRU, linear, 0 mean_absolute_error, RMSPROP, 10, 50	98.8 %
ft-64-3/6, 100, GRU, linear, 0 mean_absolute_error, RMSPROP, 10, 50	98.6 %
ft-64-11/17, 100, GRU, linear, 0 mean_absolute_error, RMSPROP, 10, 50	97.8 %
ft-64-7/13, 50, GRU, linear, 0 mean_absolute_error, RMSPROP, 10, 50	96.7 %
ft-64-7/13, 100, GRU, linear, 0.2 mean_absolute_error, RMSPROP, 10, 50	94.7 %
ft-64-7/13, 100, GRU, linear, 0 mean_absolute_error, ADAM, 10, 50	94.8 %
ft-64-7/13, 100, GRU, linear, 0 mean_absolute_error, RMSPROP, 5, 50	94.9 %
ft-64-7/13, 100, GRU, linear, 0 mean_absolute_error, RMSPROP, 10, 25	99.3 %

(b) LSTM neurons

Hyperparameters	Difference
ft-64-7/13, 100, LSTM, linear, 0 mean_absolute_error, RMSPROP, 10, 50	99.7 %
ft-64-7/13, 100, LSTM, relu, 0 mean_absolute_error, RMSPROP, 10, 50	61.8 %
ft-64-7/13, 100, LSTM, linear, 0 mean_absolute_error, RMSPROP, 5, 50	94.5 %
ft-64-7/13, 50, LSTM, linear, 0 mean_absolute_error, RMSPROP, 10, 50	90.2 %
ft-64-3/6, 100, LSTM, linear, 0 mean_absolute_error, RMSPROP, 10, 50	97.4 %
ft-64-3/6, 100, LSTM, linear, 0 mean_absolute_error, RMSPROP, 5, 50	91.6 %
ft-64-3/6, 50, LSTM, linear, 0 mean_absolute_error, RMSPROP, 5, 50	84.2 %
ft-64-10/13, 100, LSTM, linear, 0 mean_absolute_error, RMSPROP, 10, 50	98.7 %

In Figure 7 we show the loss and validation loss during the 50 epochs of training with the best hyperparameters. We use 0.5% of samples for validation loss for every epoch. The curves do not show overfitting.

Figure 7: Loss and validation loss per epoch in training with the best hyperparameters

In Table 5 we show some examples of contexts and the model predictions (three nearest domains) compared with the true next domain that occurs in the traces. In the first example, we see that the model correctly predicts a weather related domain but not the correct one. In the second case, the predicted domain (*mercadolibre.com.uy*) is correct but the particular true subdomain appears in the second nearest position (*syi.mercadolibre.com.uy*). In the third case, the domain is also correct and the subdomain queried appears as the first nearest predicted option.

5.3 Comparison with baseline and other embeddings

In this section, we will compare the previously described architecture with a baseline, and with two variants of embedding layers. For this, we use a smaller dataset, with 10 million samples of length 5, where 7 million are used for training and the rest (3 million) for testing.

Our best `fastText` architecture (with the hyperparameters tuned, and the pre-trained `ft-64-7/13` embedding), is trained over this dataset and gets about 26% of correct predictions with the nearest vector and about 38% of accuracy considering the first three nearest vectors.

In order to evaluate the quality of these results, we implement a simple baseline predictor that answers with the three most popular domains of the corpus. They are `www.facebook.com`, `a.rootservers.net` and `zxv10032h1081` (this last domain is traffic automatically generated by the ZTE modems, very popular in our traces). The most popular domain baseline gets about 4.8% correct predictions, the second 2.7% and the third 1.7%; an accuracy of 9.2% over the dataset (considering the three predictions).

As said, we also compare our architecture with two variants of the embedding layer:

Table 5: Context and model prediction examples

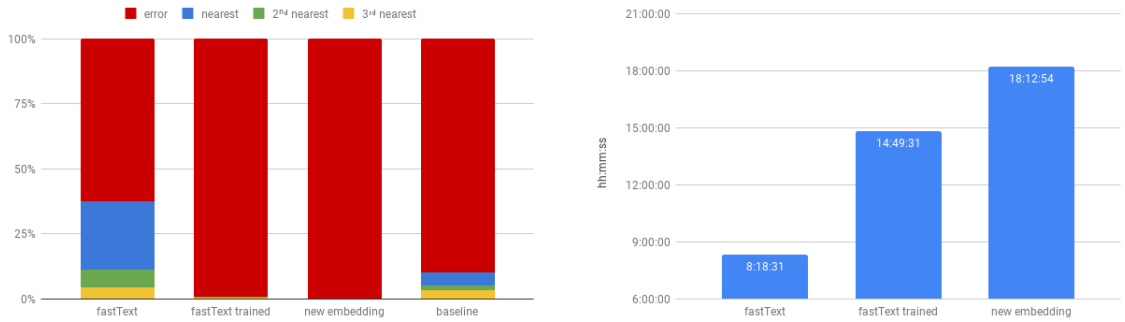
Context	video.google.com.uy, espanol.weather.com, webcache.googleusercontent.com www.tutiempo.net, meteorologiauruguay.blogspot.com
Correct	*uy.pronosticodeltiempo.info*
Predicted	meteorologiauruguay.blogspot.com, meteorologiauruguay.blogspot.com.ar bcpsalto.blogspot.com
Context	static.ak.facebook.com, registration.mercadolibre.com.uy, syi.mercadolibre.com.uy vehiculoshome.mercadolibre.com.uy, registration.mercadolibre.com.uy
Correct	*syi.mercadolibre.com.uy*
Predicted	registration.mercadolibre.com.uy, *syi.mercadolibre.com.uy*, articulo.mercadolibre.com.uy
Context	plus.google.com, maps.google.com.uy, translate.google.com.uy, plus.google.com translate.google.com.uy
Correct	*video.google.com.uy*
Predicted	*video.google.com.uy*, translate.google.com.uy, maps.google.com.uy

- a `fastText` re-trained embedding: where the embedding layer is initialized with the pre-trained `ft-64-7/13` embedding, and it is re-trained with the RNN architecture; and
- a new embedding: where the embedding layer is initialized with random values, and it is trained from scratch with the RNN architecture.

Figure 8(a) summarizes the results. For each variant, we show the accuracy obtained considering the nearest vector in the embedding to the vector predicted by the RNN, and also the accuracy if the second or third nearest vectors are considered. The best results are obtained by using the original `fastText` embedding and leaving it constant during the whole experiment, with 38% of accuracy considering the first three nearest vectors. These results are significantly better that the ones obtained by training the embedding, starting it from scratch, or using the baseline. Comparing our nearest vector prediction with the most popular domain in the baseline we get about 5 times better results. The same happens if we considering the second and third nearest domains also, slightly improving our results (about 10% better). This is positive as we prefer to give a single correct prediction.

Moreover, the training time presents clear differences between variants, and it is shown in Figure 8(b). Please note that no training is needed for the baseline. The time taken to run the test with the original embedding is about 50% of the time required in case the embedding is trained during the training of the neural network.

As summary, this section shows that the pre-trained embedding already encodes good relationships between domains and, as was shown before, creating this embedding is relatively inexpensive (14 minutes in general purpose office hardware).



(a) Accuracy results.

(b) Training time.

Figure 8: Accuracy and training time for four predictors: 1) `fastText` embedding: the embedding layer is initialized with the pre-trained `fastText` embedding, and it is not modified during RNN training; 2) `fastText` re-trained embedding: the embedding layer is initialized with the pre-trained `fastText` embedding, and it is re-trained with the RNN architecture; 3) new embedding: the embedding layer is initialized with random values, and it is trained with the RNN architecture; and 4) a simple baseline that uses the three most popular domains

6 Experiments and Results from the user's point of view

In this section, we partially repeat the experiments of previous Section 5, using a different corpus coming form the preprocessing that mitigates the DNS trace limitations (described in Section 3.1). We use the same DNS logs (March 21th, 2013, with almost 148 million of queries) as input of preprocessing to generate the corpus. The preprocessing reduces the number of different domains from 3.5 million to 2 million and also changes the distribution of domains as can be seen in Figure 9. Now the most popular domain has 17% of the total traffic and the top three represent almost 30%. In general we see that the traffic is accumulated in a much smaller set of domains. Again, we filter the traces and take only the 40000 more popular domains that in this case accounts for almost 97% of all data.

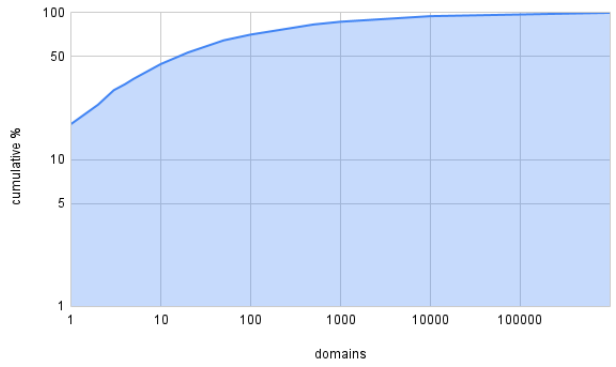


Figure 9: Popularity of domains, i.e. 100% cumulative percentage of requests per domain (log axes)

6.1 Pre-trained Word Embeddings

With this new corpus, we train new word embeddings, using the same `fastText` algorithm implementation and the same hardware as in the previous section. We create embeddings with the same subword sizes as the previous case as shown in Table 6.

Table 6: Pre-trained Word Embeddings for the user's view corpus using the `fastText` algorithm with 64 dimensions

Name	Method	Dimension	n-grams
`ft-user-64-3/6`	`fastText`	64	3/6
`ft-user-64-7/13`	`fastText`	64	7/13
`ft-user-64-10/13`	`fastText`	64	10/13
`ft-user-64-11/17`	`fastText`	64	11/17

Using the `ft-user-64-7/13` embedding, in Table 7 we show the nearest vectors to the same DNS names presented in Table 2. Note that in this case we remove the first parts of the domain name as we simplified the domains during preprocessing. It can be seen that there is a relation between near domain vectors. Some of them probably are close because they are queried together automatically during the load of the page. Some of them are semantically related for example `gmail.com` and `icloud.com` as both are email providers.

6.2 Best architecture and Hyperparameter tuning

The RNN described in Section 4.3 is built choosing as hyperparameters: one of the four embeddings of Table 6, the number of neurons in the hidden layer, the window size and number of epochs. The type of neurons were fixed at GRU, the activation function at linear, the dropout at 0, the loss function at mean absolute error and the optimizer at RMSPROP that were the best parameters from the previous experiment. We also use the same values for training, testing and validation as in the previous case. The different options tested are shown in Table 8. They were tested with the same software and hardware at Uruguay's National Center for Supercomputing.

The hyperparameter tuning took about 100 hours in total. And the best result (the option that reaches the lower loss value) is obtained using the `ft-user-64-3/6` embedding, 150 neurons, a 5 domain window, and 25 epochs which is different than the previous case.

Table 7: Nearest vectors to some common domains in the `ft-user-64-7/13` embedding

facebook.com	
Nearest	fbcdn.net
Second nearest	akamaihd.net
Third nearest	google.com
mercadolibre.com.uy	
Nearest	mercadolibre.com
Second nearest	mlapps.com
Third nearest	mercadoclics.com.uy
gmail.com	
Nearest	icloud.com
Second nearest	apple.com
Third nearest	google.com
youtube.com	
Nearest	ytimg.com
Second nearest	youtubenocookie.com
Third nearest	google.es

Table 8: Hyperparameters options tested

Hyperparameter	Options
embedding name	ft-user-64-3/6, ft-user-64-7/13 ft-user-64-10/13 and ft-user-64-11/17
number of neurons on the hidden layer	150, 100 and 50
training window size	10 and 5
number of epochs	100, 50 and 25

The different hyperparameters are as before compared using the accuracy obtained comparing the three nearest vectors in the embedding to the vector predicted by the RNN. In Table 9 we show the results of some combinations of parameters tested compared with the best combination.

In Figure 10 we show the loss and validation loss during the 25 epochs of training with the best hyperparameters.

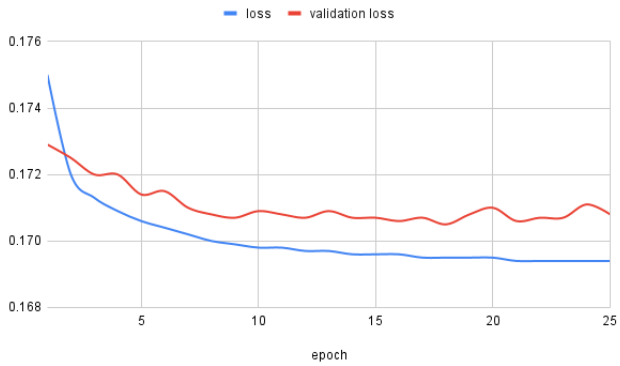


Figure 10: Loss and validation loss per epoch in training with the best hyperparameters

In Table 10 we show some examples of contexts and the model predictions (three nearest domains) compared with the true next domain that occurs in the traces. In the first case we see a context with mostly journalistic sites (from Uruguay and Argentina) and the predictions are also related to a newspaper from Uruguay. The real domain is in fact another newspaper but in this case from Ecuador. In the second case, the last three domains are related to *mercadolibre* so the predictions are also domains related with this site. The correct one is in the second position. In the third case, the context is dominated by clothing retail sites and also are the nearest three domains. Here the correct domain is the first option. Some of these sites no longer exist but most of them were owned by Amazon so that explains the relation between them as they were probably cross-linked in each of their pages.

Table 9: Comparison of hyperparameters

Hyperparameters	Difference
ft-user-64-10/13, 50, 5, 100	89.7 %
ft-user-64-3/6, 100, 5, 50	99.3 %
ft-user-64-7/13, 50, 10, 50	87.9 %
ft-user-64-3/6, 150, 10, 25	98.8 %
ft-user-64-10/13, 150, 5, 100	95.5 %
ft-user-64-7/13, 150, 5, 25	97.0 %

Table 10: Context and model prediction examples

Context	montevideo.com.uy, akamai.net, google.com, continental.com.ar, elpais.com.uy
Correct	*elcomercio.com*
Predicted	elpais.com.uy, elgallito.com.uy, diarioelpais.com
Context	googleanalytics.com, facebook.com, mercadolibre.com.ar, mlstatic.com, mercadolibre.com.ar
Correct	*mlapps.com*
Predicted	mlstatic.com, *mlapps.com*, mercadolibre.com
Context	fabric.com, imdb.com, junglee.com, myhabit.com, shopbop.com
Correct	*soap.com*
Predicted	*soap.com*, myhabit.com shopbop.com

6.3 Comparison with baseline and other embeddings

As before, we will compare the previously described architecture with a baseline, and with two variants of embedding layers. Again, we will use a smaller dataset, with 10 million samples, where 70% are used for training and the rest for testing.

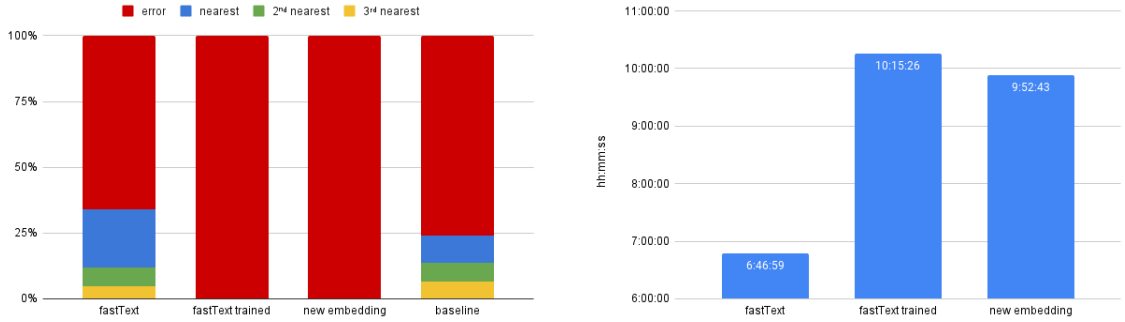
Our best `fastText` architecture (with the hyperparameters tuned, and the pre-trained `ft-user-64-3/6` embedding), is trained over this dataset and gets about 22% of correct predictions with the nearest vector and about 34% of accuracy considering the first three nearest vectors. We again evaluate the quality of these results with a baseline predictor that tests the three most popular domains. As we merge some domains form the previous corpus into the same one and remove duplicates the most popular domains are different than the ones we had before. In this case they are `facebook.com`, `google.com` and `fbcdn.net`. The most popular domain baseline gets about 10% correct predictions, the second 7% and the third 6.5% and an accuracy of 24.1% over the dataset (considering the three predictions).

As said, we compare our architecture with two variants of the embedding layer: a `fastText` re-trained embedding, and a newly trained embedding from scratch. Figure 11(a) summarizes the results. For each variant, we show the accuracy obtained considering the nearest vector in the embedding to the vector predicted by the RNN, and also the accuracy if the second or third nearest vectors are considered. We see that our predictions are better than the baseline in the sense that they predict the next domain queried by the user on the first try with more than double the accuracy (22% vs 10%). However, the cumulative difference considering the three domains is not much better than the baseline (34% vs 24%). The comparison between the original embedding and the options of training the embedding or starting it from scratch remain the same, with the option of using the precalculated embedding and leaving it constant being much better than the others both in terms of accuracy and training time (Figure 11(b)).

6.4 Comparison between results of the two corpora cases

With the first corpus, from the server's side point of view, we got good results more than four times better than the baseline. However, the results were not nearly as good with the second corpus (from the user's point of view) as they are only about 40% better than the baseline. The main difference from the first case is that our architecture is slightly worse than before but, most importantly, the baseline is much better. This is probably influenced by the preprocessing as it joins different domains into the same and thus the three most popular domains accumulate almost 30% of all traffic.

As summary, we can say that in this case, as we try to predict the domains queried as seen by the user, the baseline of the most popular domains is a much better predictor of the user behaviour and, even if we are able to learn from the input data, what we get is not much better that just returning the most popular domains.



(a) Accuracy results.

(b) Training time.

Figure 11: Accuracy and training time for four predictors: 1) `fastText` embedding: the embedding layer is initialized with the pre-trained `fastText` embedding, and it is not modified during RNN training; 2) `fastText` re-trained embedding: the embedding layer is initialized with the pre-trained `fastText` embedding, and it is re-trained with the RNN architecture; 3) new embedding: the embedding layer is initialized with random values, and it is trained with the RNN architecture; and 4) a simple baseline that uses the three most popular domains

7 Conclusions

In this work, we show the use of pre-trained vector embeddings of domains to predict the next domain to be queried in the Internet Domain Names System (DNS). We evaluate two variants of the prediction problem: the first case tries to predict the next domain queried by a user from the recursive DNS server's point of view, i.e. it tries to predict the effective new query received by the server. The second case tries to predict the next domain queried by a user from the user's point of view, i.e. it tries to predict the navigation behaviour of the users. The first variant is useful to design and optimize the service performance (cache policies and latency times at the DNS recursive server). While the second one is useful to understand the user's behaviour.

Our methodology works as follows. Unsupervised, we create a vector embedding from a corpus of real anonymized DNS log queries from a large Internet Service Provider (ISP). We use the embedding as a layer of a recurrent neural network (RNN) that is trained to predict the next DNS query generated by a user. We apply this methodology for the two problem variants, creating two different corpora from the same log data, and evaluating several vector embeddings and other hyperparameters for both cases. In all the cases, we use the `fastText` algorithm to create the embeddings, which was shown high performance for this task in our previous work.

For the first problem variant (the server's side point of view), we obtain 38% of accuracy considering the first three nearest vectors. It is a very good result considering the size of the problem with 40000 domains. Even if the results are not near 100% they can be useful for applications like anomaly detection and cache optimization where they can give useful results even if they are not the best possible. Also, we show that our results are significantly better than the ones obtained by re-training the embedding, starting it from scratch, or using a baseline (with an accuracy of 9.2%). However, the results were not nearly as good with the second variant (from the user's point of view). Where we obtain 34% of accuracy considering the first three nearest vectors, and it is only about 40% better than the baseline (with an accuracy of 24.1%). The main difference from the first case is that our solution is slightly worse than before but, most importantly, the baseline is much better.

It is important to note, that in the two problem variants, the generic pre-trained embedding gets better results than embedding specifically trained for this particular application. The use of pre-trained embedding has several advantages as the embedding has to be created only once and can be used in many applications. This allows to dedicate more resources once to create the embedding and later save time by reusing the trained embedding in various applications.

As future work, other applications for this pre-trained embedding can be tested to see if it proves useful for other uses. Also, the possibility of updating the embedding and the network online as new queries are seen can be exploited in order to implement a system that can predict domains with real-time data. Moreover, it is possible to test this same methodology using a different deep learning architecture based on transformers or encoders/decoders.

8 Acknowledgments

This research was partially supported by the "Programa de Desarrollo de las Ciencias Básicas (PEDECIBA)" of Uruguay. Some of the calculations reported in this paper were performed in ClusterUY, a newly installed platform for high-performance scientific computing at the National Supercomputing Center, Uruguay.

References

[1] J. Merlino and P. Rodríguez-Bocca, "Short-time prediction of dns queries using deep learning and pre-trained word embedding," in *2021 XLVII Latin American Computing Conference (CLEI)*, 2021, pp. 1–10.

[2] O. (communications regulator in the UK), "Adults' media use and attitudes. report 2016," 2016, report available from www.ofcom.org.uk. [Online]. Available: https://www.ofcom.org.uk/research-and-data/media-literacy-research/adults-media-use-and-attitudes

[3] J. Yan, N. Liu, G. Wang, W. Zhang, Y. Jiang, and Z. Chen, "How much can behavioral targeting help online advertising?" in *Proceedings of the 18th International Conference on World Wide Web*, ser. WWW '09. New York, NY, USA: ACM, 2009, pp. 261–270. [Online]. Available: http://doi.acm.org/10.1145/1526709.1526745

[4] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. P. Kuksa, "Natural language processing (almost) from scratch," *CoRR*, vol. abs/1103.0398, 2011. [Online]. Available: http://arxiv.org/abs/1103.0398

[5] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th International Conference on Machine Learning*, ser. ICML '08. New York, NY, USA: ACM, 2008, pp. 160–167. [Online]. Available: http://doi.acm.org/10.1145/1390156.1390177

[6] W. Lopez, J. Merlino, and P. Rodriguez-Bocca, "Vector representation of internet domain names using a word embedding technique," in *2017 XLIII Latin American Computer Conference (CLEI)*, 2017, pp. 1–8.

[7] W. López, J. Merlino, and P. Rodríguez-Bocca, "Learning semantic information from internet domain names using word embeddings," *Eng. Appl. Artif. Intell.*, vol. 94, p. 103823, 2020. [Online]. Available: https://doi.org/10.1016/j.engappai.2020.103823

[8] X. Fang, X. Sun, J. Yang, and X. Liu, "Domain-embeddings based dga detection with incremental training method," 2020.

[9] A. Arora, "Dns2vec: Exploring internet domain names through deep learning." Santa Clara, CA: USENIX Association, Aug. 2019.

[10] F. Le, M. Srivatsa, and D. Verma, "Unearthing and exploiting latent semantics behind dns domains for deep network traffic analysis," 2019.

[11] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *arXiv preprint arXiv:1607.04606*, 2016.

[12] W. Ruan, Y. Liu, and R. Zhao, "Pattern discovery in dns query traffic," *Procedia Computer Science*, vol. 17, pp. 80 – 87, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S18770509101452

[13] D. Madariaga, M. Panza, and J. Bustos-Jiménez, "Dns traffic forecasting using deep neural networks," in *Machine Learning for Networking*, É. Renault, P. Mühlethaler, and S. Boumerdassi, Eds. Cham: Springer International Publishing, 2019, pp. 181–192.

[14] R. Begleiter, Y. Elovici, Y. Hollander, O. Mendelson, L. Rokach, and R. Saltzman, "A fast and scalable method for threat detection in large-scale dns logs." *2013 IEEE International Conference on Big Data, Big Data, 2013 IEEE International Conference on*, pp. 738 – 741, 2013. [Online]. Available: http://proxy.timbo.org.uy/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=edseee&AN=edseee.6691646&lang=es&site=eds-live

[15] W. Jiang, K. Yu, X. Wu, F. Wei, and B. Wang, "Domain2vec: Vector representation of mobile server's domain based on mobile user visiting sequences," 03 2018, pp. 232–236.

[16] IETF Network Working Group, "Internet Domain Name System Standard: Domain names - concepts and facilities (RFC 1034)," 1987.

[17] ——, "Internet Domain Name System Standard: Domain names - implementation and specification (RFC 1035)," 1987.

[18] A. P. and L. C., *DNS and BIND, 4th Edition.* New York: O'Reilly, Apr. 2001.

[19] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *CoRR*, vol. abs/1301.3781, 2013. [Online]. Available: http://arxiv.org/abs/1301.3781

[20] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *CoRR*, vol. abs/1310.4546, 2013. [Online]. Available: http://arxiv.org/abs/1310.4546

[21] Y. Goldberg and O. Levy, "word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method," *CoRR*, vol. abs/1402.3722, 2014. [Online]. Available: http://arxiv.org/abs/1402.3722

[22] Q. V. Le and T. Mikolov, "Distributed representations of sentences and documents," *CoRR*, vol. abs/1405.4053, 2014. [Online]. Available: http://arxiv.org/abs/1405.4053

[23] T. Mikolov, E. Grave, P. Bojanowski, C. Puhrsch, and A. Joulin, "Advances in pre-training distributed word representations," *CoRR*, vol. abs/1712.09405, 2017. [Online]. Available: http://arxiv.org/abs/1712.09405

[24] J. L. Elman, "Finding structure in time," *Cognitive Science*, vol. 14, no. 2, pp. 179–211, 1990. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1207/s15516709cog1402_1

[25] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, March 1994.

[26] R. Ahmad, B. Yang, G. Ettlin, A. Berger, and P. Rodríguez-Bocca, "A machine-learning based convlstm architecture for ndvi forecasting," *International Transactions in Operational Research*, vol. n/a, no. n/a. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/itor.12887

[27] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training Recurrent Neural Networks," *CoRR*, vol. abs/1211.5063, p. arXiv:1211.5063, Nov 2012. [Online]. Available: https://arxiv.org/abs/1211.5063v2

[28] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[29] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," 2014.

[30] F. Chollet *et al.*, "Keras," https://github.com/fchollet/keras, 2015.

[31] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/