Deep Learning for Enterprise API traffic

Mamoon Yunus
Forum Systems
mamoon@stanford.com

Abstract

From IoT to cloud services, modern communication almost exclusively relies on HTTP-based APIs. Typically, corporations deploy an API gateway as a proxy to manage inbound and outbound API calls. We focus on inspecting API gateway traffic log files and building DNN-based models for inbound traffic. With a predictive model trained and enabled on an API gateway, intelligent traffic shaping and Quality of Service (QoS) decisions can be made on-board the API gateway automatically, even before invoking remote enterprise systems. We model remote latency based on inbound HTTP-based API messages. We compare predictive metrics between sparse one-hot-encoded and dense Word2Vec-embedded representations of inbound API messages. Our research shows that dense vector representation are a viable and efficient alternative to encoded representations for HTTP-based enterprise API traffic.

1 Introduction

APIs are the backbone of our modern digital economy. As shown in Figure [1], typical API request-response pair involves a client making an HTTP(S) call with a URI, query parameters, and additional HTTP header information, to an API gateway deployed in the DMZ. The API gateway processes the request and invokes an enterprise system such as database, application server or a mainframe (remote system). The response is usually an XML or JSON message. To protect communication over APIs, corporations deploy an API Security gateway, such as Forum Sentry [1] that acts like deep-content firewalls at the corporate edge.

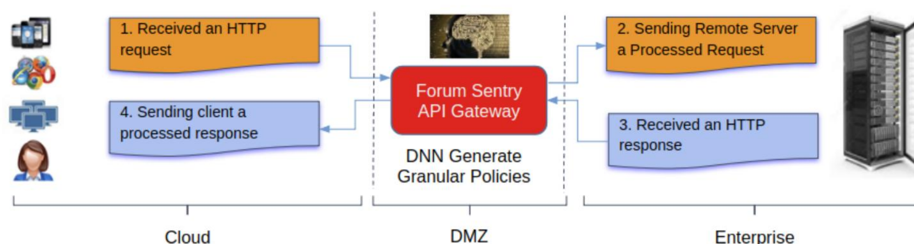

Figure 1: Deep Neural Network adds granular policies to API gateway architecture

Centralizing API security is a core enterprise business practice [2] especially for companies that are on-boarding hundreds of API services as they integrate with the customers and internal systems. Whenever a new API service in on-boarded, a new policy has to be manually authored. Most companies keep coarse-grain policies that apply to a broad array of API services. Although this strategy reduces the number of polices, it prevents companies from setting fine-grain policies specific for a given API. Such fine-grain policies provide better Quality of Service (QoS) but require labor-intensive policy authoring.

The objective of this project is to use deep neural networks to learn API traffic flow characteristics such as HTTP remote latency. The trained DNN models can then be used to dynamically build fine-grained policies aligned with customer specific API traffic patterns.

2 Related work

HTTP anomaly detection has been investigated using Naive Bayes on URLs [4] and a variety of clustering algorithms [5]. Commercial API security products claim machine learning and AI capabilities [3]. In this project we explore novel techniques such as sparse and dense representations of deep HTTP header content, beyond URLs. We also combine word embedding unsupervised techniques with supervised neural networks to build end-to-end DNNs. This end-to-end framework can scale with dynamic extraction of features that extend beyond HTTP headers to deeper HTTP payloads (JSON, XML) in API-based communication.

CS230: Deep Learning, Winter 2018, Stanford University, CA.

3 Data and Features

For this project, Forum Sentry log files are the primary source for API data, although this may be extended any sources of API data. For pre-processing these log files and extracting features, three python classes (LogData, BaseFeatures, DerivedFeatures) are developed.

3.1 Input Features

From $\sim 5M$ lines of log files, $\sim 20,000$ round trip request-response pairs are extracted that represent 45 minutes of business transaction before the log files roll over. These transactions serve as the data set for training and validation with a 75%-25% training-validation split. As shown in Figure [2], categorical input features such as URI, Method, Protocol and Client IP are extracted.

	Scheme	Method	short_URL	Client	remote_latency
0	https	POST	/CareCredit/getAccount	10.78.36.12	359
1	https	POST	/MWEncryptD/services/TokenService	10.78.36.78	236
2	https	POST	/MWEncryptD/services/TokenService	10.78.36.78	235
3	https	POST	/CareCredit/iOvationRegister	10.78.36.12	641

Figure 2: Examples from data set.

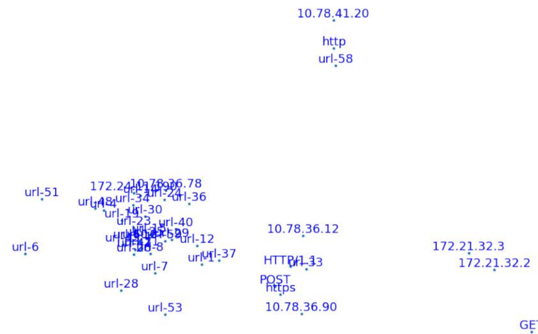

Figure 3: 2-D PCA for embedded vectors

We used sparse one-hot encoding and dense word embedding to explore feature representation. A total of 41 features are extracted from HTTP headers. Using Word2Vec, we concluded that 5 features had non-repeating values. For example, HTTP cookies are randomly generated for HTTP request and therefore have no predictive value since each new message input to a trained DNN will have a unique random value for the cookie. Such features are removed from the feature list.

2-D PCA visualization of word embedding vectors, Figure [3], shows that dense representation of HTTP header features have a structure with certain features showing affinity. For example, in the top right corner, we see *url-58*, *http*, and a client IP *10.78.41.20* clustered together. We inspected the log files manually to verify that indeed these features are adjacent and in the same HTTP request. This, along with other, verified feature relationships provide us confidence in the efficacy of dense vector representations as an alternative to sparse one-hot encoding.

As seen in Table [1], we considered a variety of feature counts and their encoded and embedded representations. Our results will show that even for a dense vector of size $D = 5$, word embedding performs remarkably well. Using this vector size, we obtain a high ratio of encoded feature values to embedded values. This shows a clear information compression advantage for HTTP content when we choose to use dense embedding over sparse encoding representation.

Features	Encoding	Embedding	Ratio	Vocab
37	19085	185	103:1	5125
17	10317	85	121:1	1881
8	322	40	8:1	204

Table 1: Encoding and Embedding Size and Ratio

The variation in number of features selected displayed in Table [1] is based on the structure of the HTTP headers. The first 8 fields are required standard HTTP fields [9]. Beyond these fields, non-standard headers that are custom,

dynamic, and user-defined may exist. We construct a super-set of all such features and set them to *None* by default, only to be populated when the features are present in a message.

3.2 Target

We target predicting remote latency based on HTTP request header features. The remote latency is captured in milliseconds as shown in Figure [4]. For network traffic, it is advised to look at log of latency [3] values instead of raw values given the high variance and the log-skew nature of the latency distribution. Figure [5] shows log_{10} remote latency distribution. On initiating an HTTP request to an API gateway, predicting remote response latency

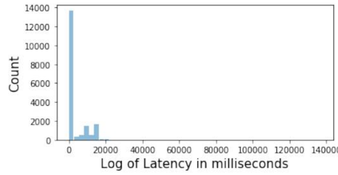

Figure 4: Raw latency

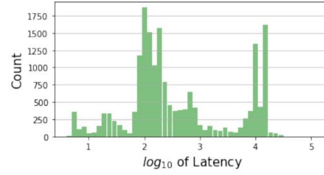

Figure 5: log_{10} latency

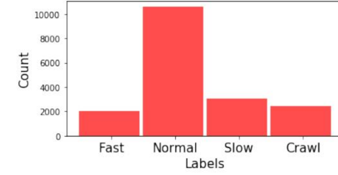

Figure 6: Label latency bins=4

enables efficient traffic shaping, Quality of Services (QoS) and resource allocation. We re-frame our target from a regression to a classification problem. Instead of predicting a numerical value for remote latency, it is more useful for API gateway resource allocation to predict a label based on an configurable histogram. The number of histogram bins can represent traffic latency categories such as *fast, normal, slow, crawl* for a bin size of 4 as shown in Figure [6].

4 Methods

Figure [7] shows two input representation methods, encoding and embedding that were the focus of this project. In addition to Softmax classification output, we built a similar regression model for predicting log_{10} of remote latency. Standard DNN techniques including Adam, L2 and dropout regularization were also explored. ReLU was used for hidden layers. For Word2Vec embedding, context window size, word vector length also served as hyper-parameters.

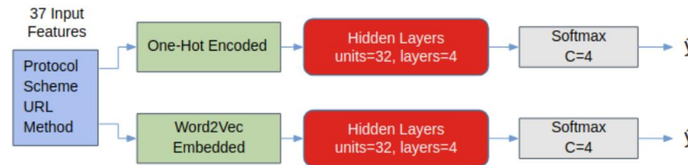

Figure 7: DNN for one-hot encoding and Word2Vec embedding

Figure [8] shows the LSTM sentiment model inspired by the Emojify-v2 assignment. The inputs are HTTP header features that are converted word vector representation. The Softmax predicts the remote latency class: *fast, normal, slow, crawl*.

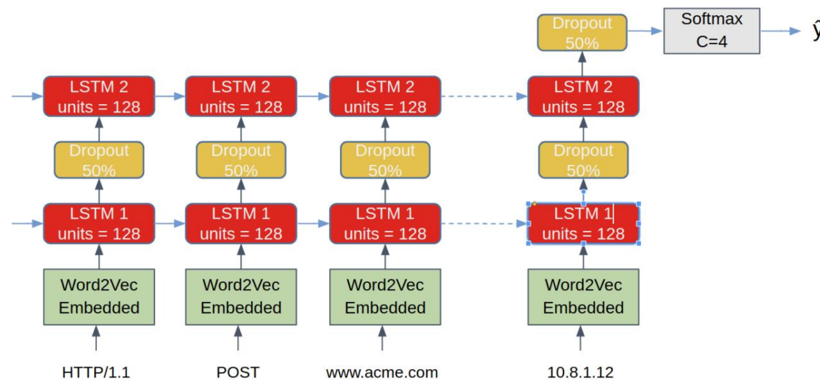

Figure 8: LSTM Sentiment Architecture

5 Experiments

To establish a base-line, we selected 17 input features and log_{10} of remote latency as our output for a DNN-regression model. We achieved $R^2 = 0.80$ with a 64-unit, 8-layer API DNN ($\lambda = 5.0$). Next, we focused on building DNN models based on remote latency labels: *fast, normal, slow, crawl*.

5.1 API Request Header One-Hot Encoding

With the full HTTP header feature set as one-hot encoded input and no regularization, we could readily over fit our data set. We used L2 regularization to close the gap between training and test accuracy values. The learning rate $\alpha = 0.0005$ was used in most one-hot encoding training runs. Higher learning rates would result in non-monotonic gradient descent. We also experimented with various dropout rates from 0.2 to 0.8, however, we did not find any significant advantages over using L2 regularization. As expected, reducing the number of features has a regularizing effect, even when $\lambda = 0$.

	$\lambda = 0$		$\lambda_{optimal}$	
Features	Train	Test	Train	Test
37	0.997	0.793	0.871	**0.851** ($\lambda = 1.065$)
17	0.952	0.834	0.852	**0.849** ($\lambda = 0.005$)
8	0.847	0.846	0.846	0.846 ($\lambda = 0$)

Table 2: One-Hot-Encoding results with regularization

Table [2] shows that we have significant feature richness to over-fit our API DNN especially with feature counts of 17 and above. For feature count of 37, post L2 regularization, systems test accuracy compared to 17 features did not improve significantly. We hypothesize that features that appear later in the HTTP headers are not standard across all input headers, are sparse, and therefore perhaps have low sensitivity to the target. We anticipate API DNNs trained with weeks or months of data, instead of ~ 45 minutes, will eventually increase sensitivity to deeper HTTP content.

5.2 API Request Header Embedding

With a large number of possible values for each feature, one-hot encoded vectors are sparsely populated and provide no relationship between features within an HTTP header. However, unlike encoding, embedding provides a relationship between features, see Figure [3].

Table [3] shows embedding accuracy results of a variety of word vector size setting, D. For 37 features, with a corpus of $\sim 20,000$ HTTP headers, excluding rare occurrence words, we extracted a vocabulary of 5215 words. A vector size, $D = vocabulary^{1/4} = 5125^{1/4} \sim 9$ is considered sufficient [8]. A promising result of this project is that a dense representation of even $D = 5$ performs well. With $D = 5$ for 37 features, our dense-to-sparse ratio of $185/19085 \sim 0.01$ provides a significant advantage in representing HTTP request headers as dense embedded vectors.

	$D = 5$		$D = 50$		$D = 200$	
Features	Train	Test	Train	Test	Train	Test
37	0.841	0.845	0.849	0.847	0.822	0.821
17	0.840	0.835	0.851	0.842	0.835	0.832
8	0.844	0.845	0.845	0.847	0.845	0.846

Table 3: Embedding accuracy with varying features and vector size

Table [3] shows that even with $\lambda = 0$, for embedded vectors, the testing and training accuracy values are closer than one-hot-encoded vectors. With the same DNN architecture used for encoded inputs, a change to embedded inputs seems to have a natural regularization effect. This is caused by a reduction in the number of input features when moving from encoded to embedded representations, which retains information through its dense nature.

We also concatenated encoded and embedded as a composite input. The results were in between separate encoded and embedded results as displayed in Table [2] and Table [3] and did not result in any significant advantage.

5.3 LSTM Sentiment Model

With Word2Vec embedded representation of HTTP headers, we assembled 8, 17 and 35 word sequences as inputs to the LSTM model, Table [4]. The LSTM model predicts remote latency "sentiment" well, especially for low feature count of 8. We expect that adding Attention Mechanism to the LSTM is probably necessary for higher feature counts.

	$D = 5$		$D = 50$		$D = 200$	
Features	Train	Test	Train	Test	Train	Test
37	0.827	0.834	0.821	0.822	0.828	0.835
17	0.823	0.824	0.823	0.824	0.821	0.822
8	0.833	0.828	0.834	0.839	0.833	0.835

Table 4: LSTM Sentiment Model Accuracy

5.4 Result Metrics

Table [5] shows accuracy metrics across a variety of models for 17 input features. While comparing encoding and embedding performance, we kept the size and shape of the DNN constant (units = 32, layers = 4). Embedding shows a natural regularization effect. It also reduces the size of the input features by a factor of 121 as shown in Table [1], a promising result for API traffic. Table [6] shows the classification report for encoded inputs. As seen in

Model	**Hyper-parameters**	**Train**	**Test**
Encoded-Regression	$\lambda = 5.0$	0.81	0.79
Embedded-Regression	$\lambda = 0$	0.81	0.80
Encoded-Classification	$\lambda = 0.005$	0.85	0.85
Embedded-Classification	$\lambda = 0, D = 5$	0.84	**0.84**
LSTM latency Sentiment	$dr = 0.5, ep = 50$	0.82	0.83

Table 5: Model Comparison

Figure [6], classes $\{0, 2, 3\}$ are of the same scale and ~ 4 times smaller than class $\{1\}$. Even though class $\{0\}$ has a 4x class imbalance compared to class $\{1\}$, its F-1 score is not impacted because of this imbalance. Adjusting for class imbalance did not help our results. Class $\{2\}$ which corresponds to remote latency label *slow* has the lowest F-1 score. Further analysis is required to assess the poor performance of this class.

Class	Precision	Recall	F-1
0	0.94	0.99	0.97
1	0.93	0.91	0.92
2	0.71	0.55	0.62
3	0.64	0.84	0.72

Table 6: Classification report for encoded inputs

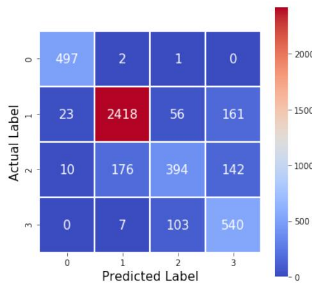

Figure 9: 1-hot encoding

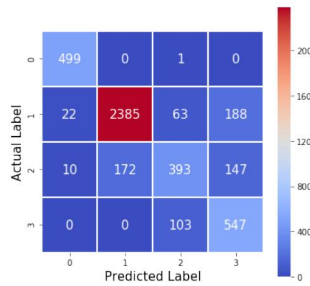

Figure 10: Word embedding

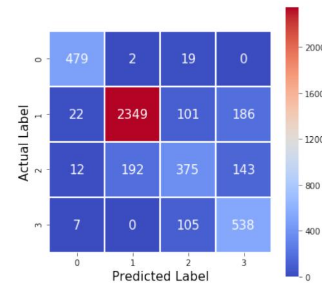

Figure 11: LSTM Sentiment

The confusion matrices for one-hot encoded, Figure [9], embedded representations, Figure [10] and LSTM model. Figure [11] show insignificant variation, giving us confidence in our models and methods.

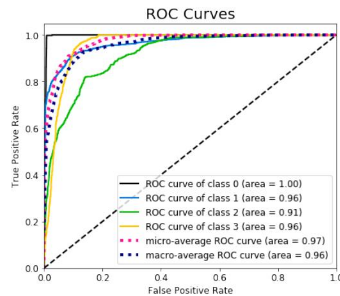

Figure 12: 1-hot encoding

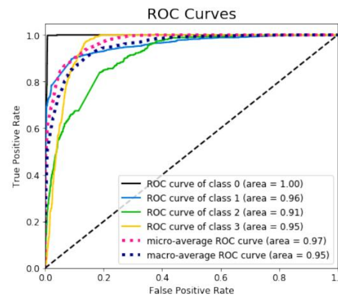

Figure 13: Word embedding

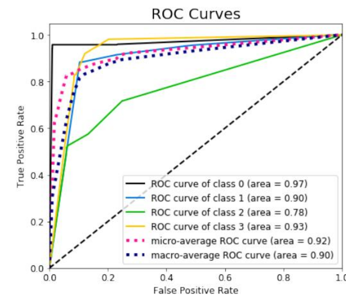

Figure 14: LSTM Sentiment

Although accuracy provides a single comparative metric, it hides details that are more evident in ROC-curves. Figure [12], [13], and [14] show variations in class class $\{2\}$ curves with LSTM sentiment model pointing to further optimization opportunities.

6 Conclusion/Future Work

Performance of dense embedded word representation of HTTP headers without requiring L2 regularization has been the most significant and valuable aspect of this project. White-listing a vocabulary of permissible HTTP header is another practical advantage of deploying word embedding functionality in API gateways. With additional data sets, we hope to extend API DNN research deeper into HTTP payloads, beyond HTTP headers and explore CNN, Attention Mechanisms and GloVe techniques to predict remote latency characteristics based on an HTTP header sequence.

7 Acknowledgement

This project has been exclusively developed by the author with valuable input from CS230 Mentor, Patrick Cho and inspiration from CS230 course materials.

References

[1] Forum Sentry API Security Gateway: *https://www.forumsys.com*

[2] Guy Levin (2018) The Role of API Security Gateways in API Security *DZone: https://dzone.com/articles/the-role-of-api-gateways-in-api-security*

[3] Artificial Intelligence and Machine Learning: A new Approach to API Security: *https://www.pingidentity.com*

[4] S. Zhang, B. Li, J. Li, M. Zhang and Y. Chen, "A Novel Anomaly Detection Approach for Mitigating Web-Based Attacks Against Clouds," 2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing, New York, NY, USA, 2015, pp. 289-294.

[5] M. Zolotukhin, T. Hämäläinen, T. Kokkonen and J. Siltanen, "Analysis of HTTP Requests for Anomaly Detection of Web Attacks," 2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing, Dalian, 2014, pp. 406-411. doi: 10.1109/DASC.2014.79

[6] Hasselmo, M.E., Schnell, E. & Barkai, E. (1995) On the log-normal distribution of network traffic. *Physica D: Nonlinear Phenomena* **167**(1-2):72-85.

[7] MacCormick, Chris (2016) Word2Vec Tutorial - The Skip-Gram Model

[8] Introducing TensorFlow Feature Columns, Monday, November 20, 2017, Posted by Tensorflow Team

[9] Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing, Internet Engineering Task Force (IETF), June 2014

Git Repository

Private: shared with project mentor, Patrick Cho. https://github.com/monogenics/deep-api