CNNs for NLP in the Browser: Client-Side Deployment and Visualization Opportunities

Yiyun Liang, Zhucheng Tu, Laetitia Huang, and Jimmy Lin
David R. Cheriton School of Computer Science
University of Waterloo
{yiyun.liang, michael.tu, laetitia.huang, jimmylin}@uwaterloo.ca

Abstract

We demonstrate a JavaScript implementation of a convolutional neural network that performs feedforward inference completely in the browser. Such a deployment means that models can run completely on the client, on a wide range of devices, without making backend server requests. This design is useful for applications with stringent latency requirements or low connectivity. Our evaluations show the feasibility of JavaScript as a deployment target. Furthermore, an in-browser implementation enables seamless integration with the JavaScript ecosystem for information visualization, providing opportunities to visually inspect neural networks and better understand their inner workings.

1 Introduction

Once trained, feedforward inference using neural networks (NNs) is straightforward: just a series of matrix multiplications, application of non-linearities, and other simple operations. With the rise of model interchange formats such as ONNX, we now have clean abstractions that separate model training from model inference. In this context, we explore JavaScript as a deployment target of neural networks for NLP applications. To be clear, we are *not* concerned with training, and simply assume the existence of a pre-trained model that we wish to deploy for inference.

Why JavaScript? We provide two compelling reasons. First, JavaScript is the most widely deployed platform in the world since it resides in every web browser. An implementation in JavaScript means that a NN can be embedded in any web page for client-side execution on any device that has a browser—from laptops to tablets to mobile phones to even potentially "smart home" gadgets. Performing inference on the client also obviates the need for server requests and the associated latencies. With such a deployment, NLP applications that have high demands on responsiveness (e.g., typeahead prediction, grammar correction) or suffer from low connectivity (e.g., remote locations or developing countries) can take advantage of NN models. Such a deployment also protects user privacy, since user data does not leave the client. Second, the browser has emerged as the dominant platform for information visualization, and JavaScript-based implementations support seamless integration with modern techniques and existing toolkits (e.g., D3.js). This provides opportunities to visually inspect neural networks.

We demonstrate a prototype implementation of a convolutional neural network for sentence classification, applied to sentiment analysis—the model of Kim (2014)—in JavaScript, running completely inside a web browser. Not surprisingly, we find that inference performance is significantly slower compared to code running natively, but the browser is nevertheless able to take advantage of GPUs and hardware acceleration on a variety of platforms. Our implementation enables simple visualizations that allow us to gain insights into what semantic n-gram features the model is extracting. This is useful for pedagogy (teaching students about neural networks) as well as research, since understanding a model is critical to improving it. Overall, our visualizations contribute to an emerging thread of research on *interpretable* machine learning (Lipton, 2016).

2 Background and Related Work

Browser-based neural networks are by no means new. Perhaps the best illustration of their potential is the work of Smilkov et al. (2016a), who illustrate backpropagation on simple multi-layer perceptrons with informative visualizations. This work, however, can be characterized as focusing on toy problems and useful primarily for pedagogy. More recently, Google Brain introduced TensorFlow.js (formerly deeplearn.js), an open-source library that brings NN building blocks to JavaScript. We take advantage of this library in our implementation.

Our work overcomes a technical challenge that to date remains unaddressed—working with word embeddings. Most NNs for NLP applications begin by converting an input sentence into an embedding matrix that serves as the actual input to the network. Embeddings can be quite large, often gigabytes, which makes it impractical to store directly in a web page. Following Lin (2015), we overcome this challenge by using an HTML standard known as IndexedDB, which allows word vectors to be stored locally for efficient access (more details below).

3 Technical Implementation

The convolutional neural network of Kim (2014) is a sentence classification model that consists of convolutions over a single sentence input embedding matrix with a number of feature maps and pooling followed by a fully-connected layer with dropout and softmax output. Since it has a straightforward architecture, we refer the reader to Kim's original paper for details. The starting point for this work is a PyTorch reimplementation of the model, which achieves accuracy comparable to the original reported results.[1]

Since our demonstration focuses on inference performance, we simply used a pre-trained model for sentiment analysis based on the Stanford Sentiment Treebank. We manually exported all weights from PyTorch and hand-coded the model architecture in TensorFlow.js, which at startup imports these weights. Since Kim CNN has a relatively simple architecture, this implementation is straightforward, as TensorFlow.js provides primitives that are quite similar to those in PyTorch. Because our implementation is pure JavaScript, the entire model can be directly included in the source of any web page, and, for example, connect to text entry boxes for input and DOM-manipulating code for output. However, there is one additional technical hurdle to overcome:

Nearly all neural networks for NLP applications make use of pre-trained word vectors to build an input representation (the embedding matrix) as the first step in inference. For a non-toy vocabulary, these word vectors consume many gigabytes. It is impractical to embed these data directly within a web page or as an external resource due to memory limitations, since all JavaScript code and associated resources are loaded into memory. Even if this were possible, all data would need to be reloaded every time the user refreshes the browser tab, leading to long wait times and an awkward user experience.

To address this challenge, we take advantage of IndexedDB, which is a low-level API for client-side storage of arbitrary amounts of data. Since IndexedDB is an HTML5 standard, it does not require additional plug-ins (assuming a standards-compliant browser). Although IndexedDB has a rich API, for our application we use it as a simple key–value store, where the key is a word and the value is its corresponding word vector. In Google's Chrome browser (which we use for our experiments), IndexedDB is supported by LevelDB, an on-disk key–value store built on the same basic design as the Bigtable tablet stack (Chang et al., 2006). In other words, inside every Chrome browser there is a modern data management platform that is directly accessible via JavaScript.

With IndexedDB, prior to using the model for inference, the user must first download and store the word embeddings locally. For convenience, the model weights are treated the same way. Note that this only needs to be performed once, and all data are persisted on the client until storage is explicitly reclaimed. This means that after a one-time setup, model inference happens locally in the browser, without any need for external connectivity. This enables tight interaction loops that do not depend on unpredictable server latencies.

4 Performance Evaluation

The first obvious question we tackle is the performance of our JavaScript implementation. How much slower is it than model inferencing performed natively? We evaluated in-browser inference on a 2015 MacBook Pro laptop equipped with an Intel Core i5-5257U processor (2 cores), running MacOS 10.13. We compared performance with the desktop machine used to train the model, which has an Intel Core i7-6800K processor (6 cores) and an NVIDIA GeForce GTX 1080

[1] https://github.com/castorini/Castor

	Latency (ms) / batch			
	1	32	64	128
PyTorch				
Desktop GPU (Ubuntu 16.04)	2.9	3.0	3.1	3.1
Desktop CPU (Ubuntu 16.04)	4.3	43	86	130
Chrome Browser				
Desktop GPU (Ubuntu 16.04)	30	56	100	135
Desktop CPU (Ubuntu 16.04)	783	47900	110000	253000
MacBook Pro GPU (MacOS 10.13)	33	180	315	702
MacBook Pro CPU (MacOS 10.13)	779	56300	126000	297000
iPad Pro (iOS 11)	170	472	786	1283
Nexus 6P (Android 8.1.0)	103	541	1117	1722
iPhone 6 (iOS 11)	400	1336	3055	7324

Table 1: Latency of our CNN running in Chrome on different devices for a batch of N sentences.

GPU (running Ubuntu 16.04). Our model was implemented using PyTorch v0.3.0 running CUDA 8.0. All experiments were performed on the Stanford Sentiment Treebank validation set. Due to the asynchronous nature of JavaScript execution in the browser, the TensorFlow.js API applies inference to batches of input sentences at a time. Thus, we measured latency per batch for batches of 1, 32, 64, and 128 sentences.

Evaluation results are shown in Table 1. The first block of the table shows the performance of PyTorch running on the desktop, with and without GPU acceleration. As expected, the GPU is able to exploit parallelism for batch inferencing, but on individual sentences, the CPU is only slightly slower. In the bottom block of the table, we report results of running our JavaScript implementation in Google Chrome (v64). We compared the desktop and the laptop, with and without GPU acceleration. For the most common case (inference on a single sentence), the browser is about an order of magnitude slower with the GPU. Without the GPU, performance drops by another $\sim 25\times$.

The above figures include only inference time. Loading the word vectors takes 7.4, 214, 459, and 1184 ms, for batch sizes of 1, 32, 64, and 128, respectively, on the MacBook Pro. As explained previously, using IndexedDB requires a one-time download of the word vectors and the model weights. This step takes approximately 16s on our MacBook Pro for 16,271 word vectors (for simplicity, we only download the vocabulary needed for our experiments). Loading the model itself takes approximately one second.

Because our implementation is in JavaScript, our model runs in any device that has a web browser. To demonstrate this, we evaluated performance on a number of other devices we had convenient access to: an iPad Pro with an Apple A10X Fusion chip, a Nexus 6P with a Qualcomm Snapdragon 810 octa-core CPU, and an iPhone 6 with an Apple A8 chip. These results are also shown in Table 1. As expected, performance on these devices is lower than our laptop, but interestingly, batch inference on these devices is faster than batch inference in the browser without GPU acceleration. This indicates that hardware acceleration is a standard feature on many devices today. These experiments illustrate the feasibility of deploying neural networks on a wide range of devices, exploiting the ubiquity of JavaScript.

5 Visualization of Feature Maps

Another major advantage of in-browser JavaScript implementations of neural networks is seamless integration with modern browser-based information visualization techniques and toolkits (e.g., D3.js), which we describe in this section. Visualizations are useful for two purposes: First, they serve as intuitive aids to teach students how neural networks function. Although there are plenty of pedagogical resources for deep learning, nothing beats the convenience of an interactive neural network directly embedded in a web page that students can manipulate. Second, contributing to growing interest in "interpretable" machine learning (Lipton, 2016), visualizations can help us understand how various network components contribute to producing the final predictions.

Although there are many examples of neural network visualizations (Smilkov et al., 2016a; Bau et al., 2017; Olah et al., 2018), they mostly focus on vision applications, where inputs and feature maps are much easier to interpret visually. The fundamental challenge with NLP applications is that word embeddings (and by extension, feature maps) exist in an abstract semantic space that has no inherent visual meaning. How to best visualize embeddings remains an open question (Smilkov et al., 2016b; Rong and Adar, 2016).

Nevertheless, feature maps in CNNs can be thought of as n-gram feature detectors. For our sentiment analysis application (and more generally, sentence classification), we designed visualizations to answer two related questions: First, given a sentence, what feature maps are highly activated and where? Second, given a particular feature map, what token sequence activates it in a corpus of sentences?

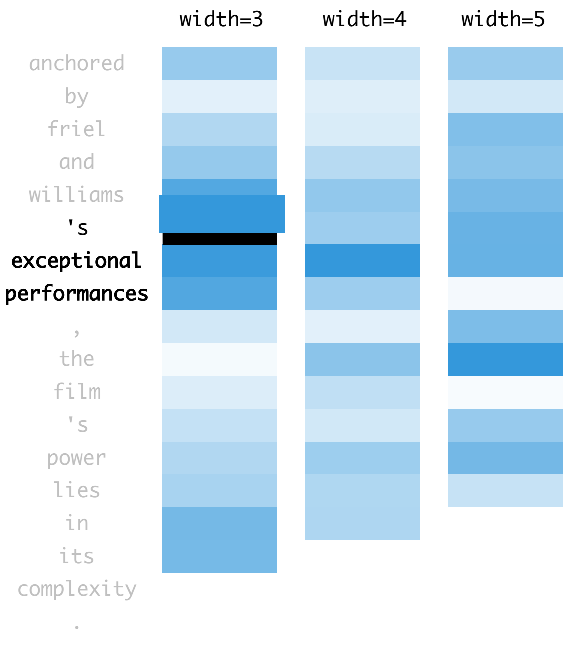
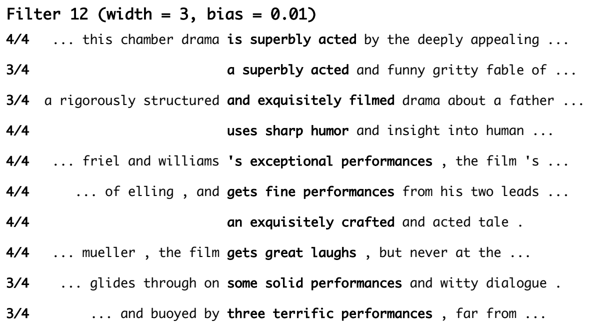


Figure 1: Visualization of feature map activations.

The visualization in Figure 1 is designed to answer the first question. Running down the left edge is the sentence that we are examining; examples in this section are taken from the development set of the Stanford Sentiment Treebank. Each column represents feature maps of a particular width; each cell is aligned with the first token from the corresponding n-gram of the sentence over which the feature map applies. The heatmap (i.e., blue saturation) corresponds to maximum activation across all feature maps on that particular n-gram in the sentence. In our interactive visualization, when we mouse over a cell, it becomes enlarged and slightly offset to indicate focus, and the corresponding n-gram in the sentence is highlighted in bold. For this sentence, we see that filters of width three are most highly activated by the n-gram sequence ['s exceptional performance]. There are multiple ways to use color to summarize the activation of the feature maps, but we have found MAX to be the most insightful. From this visualization, we can see that feature maps of widths four and five activate on different parts of the sentence.

From the visualization in Figure 1, we see that *some* feature map activates highly on the n-gram ['s exceptional performance]. But which one? To answer this question, we can pivot over to the visualization shown in Figure 2, where we see that the answer is Filter 12: we show the n-grams that trigger the highest activation from sentences in the



```
Filter 12 (width = 3, bias = 0.01)
4/4   ... this chamber drama is superbly acted by the deeply appealing ...
3/4                         a superbly acted and funny gritty fable of ...
3/4  a rigorously structured and exquisitely filmed drama about a father ...
4/4                         uses sharp humor and insight into human ...
4/4   ... friel and williams 's exceptional performances , the film 's ...
4/4      ... of elling , and gets fine performances from his two leads ...
4/4                         an exquisitely crafted and acted tale .
4/4   ... mueller , the film gets great laughs , but never at the ...
3/4    ... glides through on some solid performances and witty dialogue .
3/4        ... and buoyed by three terrific performances , far from ...
```

Figure 2: Visualization of the n-grams that a particular feature map most highly activates on.

development set. The n-grams are aligned in a keyword-in-context format for easy browsing. To the left of each sentence we show x/y, where x is the ground truth label and y is the predicted label. Here, we clearly see that all the n-grams are related to positive aspects of performances, and this gives us some insight into the semantics captured by this feature map. From this visualization, we can click on any sentence and pivot back to the sentence-focused visualization in Figure 1.

6 Future Work and Conclusions

We describe a JavaScript implementation of a convolutional neural network that runs completely in the browser. Unsurprisingly, in-browser inference is significantly slower. However, for many applications, such a performance tradeoff may be worthwhile given the advantages of a JavaScript implementation—the ability to embed a neural network in any web page, the ability to run on a wide variety of devices and without internet connectivity, and opportunities for visualizations that help us interpret the model.

Ongoing work focuses on better integration of model training and browser deployment. Currently, porting a PyTorch model into JavaScript requires tediously rewriting the model into TensorFlow.js by hand. Although the library supports reading TensorFlow models, importers for PyTorch do not exist yet, as far as we are aware. Provided that the right adaptors are built, the ONNX model interchange format could provide an interlingua to support seamless integration, enabling a future where running neural networks in JavaScript becomes routine.

Acknowledgments. This research was supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

References

David Bau, Bolei Zhou, Aditya Khosla, Aude Oliva, and Antonio Torralba. 2017. Network dissection: Quantifying interpretability of deep visual representations. In *Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2017)*, pages 6541–6549, Honolulu, Hawaii.

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. 2006. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating System Design and Implementation (OSDI 2006)*, pages 205–218, Seattle, Washington.

Yoon Kim. 2014. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)*, pages 1746–1751, Doha, Qatar.

Jimmy Lin. 2015. Building a self-contained search engine in the browser. In *Proceedings of the ACM International Conference on the Theory of Information Retrieval (ICTIR 2015)*, pages 309–312, Northampton, Massachusetts.

Zachary C. Lipton. 2016. The mythos of model interpretability. *arXiv:1606.03490*.

Chris Olah, Arvind Satyanarayan, Ian Johnson, Shan Carter, Ludwig Schubert, Katherine Ye, and Alexander Mordvintsev. 2018. The building blocks of interpretability. *Distill*.

Xin Rong and Eytan Adar. 2016. Visual tools for debugging neural language models. In *Proceedings of the ICML 2016 Workshop on Visualization for Deep Learning*, New York, New York.

Daniel Smilkov, Shan Carter, D. Sculley, Fernanda B. Viégas, and Martin Wattenberg. 2016a. Direct-manipulation visualization of deep networks. In *Proceedings of the ICML 2016 Workshop on Visualization for Deep Learning*, New York, New York.

Daniel Smilkov, Nikhil Thorat, Charles Nicholson, Emily Reif, Fernanda B. Viégas, and Martin Wattenberg. 2016b. Embedding projector: Interactive visualization and interpretation of embeddings. *arXiv:1611.05469*.