WebDNN: Fastest DNN Execution Framework on Web Browser

Masatoshi Hidaka*
The University of Tokyo
Tokyo, Japan
hidaka@mi.t.u-tokyo.ac.jp

Yuichiro Kikura*
The University of Tokyo
Tokyo, Japan
kikura@mi.t.u-tokyo.ac.jp

Yoshitaka Ushiku
The University of Tokyo
Tokyo, Japan
ushiku@mi.t.u-tokyo.ac.jp

Tatsuya Harada
The University of Tokyo / RIKEN
Tokyo, Japan
harada@mi.t.u-tokyo.ac.jp

ABSTRACT

Recently, deep neural network (DNN) is drawing a lot of attention because of its applications. However, it requires a lot of computational resources and tremendous processes in order to setup an execution environment based on hardware acceleration such as GPGPU. Therefore, providing DNN applications to end-users is very hard. To solve this problem, we have developed an installation-free web browser-based DNN execution framework, WebDNN. WebDNN optimizes the trained DNN model to compress model data and accelerate the execution. It executes the DNN model with novel JavaScript API to achieve zero-overhead execution. Empirical evaluations show that it achieves more than two-hundred times the unusual acceleration. WebDNN is an open source framework and you can download it from https://github.com/mil-tokyo/webdnn.

CCS CONCEPTS

• **Software and its engineering** → **Software libraries and repositories**; • **Information systems** → *Mobile information processing systems*;

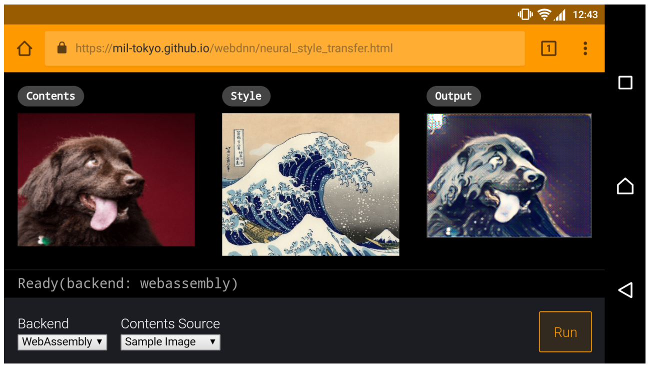
KEYWORDS

Deep Learning, Web Browser, GPGPU, Acceleration, Cross Platform, Open Source Software, Computer Vision

ACM Reference format:
Masatoshi Hidaka, Yuichiro Kikura[1], Yoshitaka Ushiku, and Tatsuya Harada. 2017. WebDNN: Fastest DNN Execution Framework on Web Browser. In *Proceedings of ACM Multimedia conference, Mountain View, CA USA, Oct 2017 (ACMMM'17),* 4 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

SUBMITTED to ACM MULTIMEDIA 2017 OPEN SOURCE SOFTWARE COMPETITION

*Both are equal contribution.



Ready(backend: webassembly)

Figure 1: Demonstration page of WebDNN. Neural Style Transfer model, in which a content image and a style image are mixed by DNN, is running on a web browser. WebDNN converts DNN models to enable fast execution on web browsers.

1 INTRODUCTION

Recently, deep neural network (DNN) is attracting a lot of attention in various fields such as image and video recognition, natural language processing and gaming AI. In these fields, DNNs are applied for various products. However, DNNs are computationally expensive and generally hardware acceleration is required for its execution, and so it is not practical to execute DNN on end-user devices such as laptops or smartphones.

One of the solutions to this is cloud computing. The applications running on end-user devices send a query to a computational server. The server processes it and sends back the result. However, this solution has two problems. First, the communication between devices and the server has a lot of latency, and it is not suitable for real-time tasks and processing large data like videos. Second, even when the data contains private information and users do not want to upload it onto the server, sending the data is inevitable and necessary. This problem has become more crucial with the development of the Internet of Things (IoT) and the increasing number of sensor devices.

As opposed to that, in another solution, DNN applications are implemented as native applications and run on end-user devices [7]. In this case, an application has to be installed to access low-level APIs and require the use of GPU acceleration. However, it is not preferable to install native applications for a good user experience.

In fact, there are few people who dedicate their time to installing research demo applications using DNN.

In this paper, we present an installation-free DNN execution framework, WebDNN. WebDNN executes DNN models on a web browser. Usually, web browsers are already installed on end-user devices and users are familiar with how to use it. Therefore, using WebDNN, DNN applications can be provided easily, without any difficulty in installing a native application. In order to accomplish maximum speed on web browsers, advanced optimizations specialized for the inference phase of DNNs are implemented.

The main highlights of WebDNN are as follows:

Non overhead interface
JavaScript is a standard programing language running on web browsers. It is executed by an interpreter. Therefore, it requires computing overhead and it cannot completely harness the capacity of the CPU. The same problem is encountered in GPU. Modern web browsers support WebGL, which is a JavaScript API to use GPU. However, this API is designed for graphics processing and is not suitable for general purpose computation. In addition, using WebGL for general purpose computing incurs overhead costs.

Keras.js [1] is a JavaScript framework to execute DNN models on a web browser. It uses weblas [3], which is a matrix operation library such as single precision general matrix-matrix production (SGEMM), but because of the overhead in using WebGL for general purpose computing, it cannot bring out the full performance of GPU. Keras.js also supports CPU-only execution. However, this implementation is very naive and much slower than the native applications because of JavaScript's overhead costs.

In contrast, WebDNN uses next generation JavaScript API, WebGPU for GPU execution, and WebAssembly for CPU execution. These APIs help to bring out the full performance of GPU and CPU.

Inference-phase-specialized optimization
To achieve speedier execution, optimizing the computation graph of DNN models is very important. Theano [9], which is a computation graph execution framework, supports these features. Execution of DNN consists of two phases, the training phase and the inference phase. The training phase updates the parameters with the back-propagation technique. The inference phase makes predictions (forward-propagation only) for the actual task. Theano is designed to be used not only in the inference phase but in the training phase as well. In this case, the framework must preserve the auxiliary data. If the framework focuses on only the inference phase, it can optimize the computation graph more aggressively.

WebDNN focuses on only the inference phase execution on end-user devices and supports aggressive optimization. This optimization pipeline can be applied for models trained with various DNN frameworks. It is not required to edit the training codes.

In section 2, technologies used in WebDNN are described. Usages are described in section 3. Then section 4 shows empirical evaluations.

2 OVERVIEW OF WEBDNN

Figure 2 shows the pipeline of WebDNN. WebDNN consists of two modules - the graph transpiler, which transpiles and optimizes trained model into an executable format on the web browser and the descriptor runner, which executes the converted model on the web browser.

2.1 Graph Transpiler

Graph transpiler is a module which transpiles and optimizes the DNN model trained by some other framework into an executable format on the web browser.

2.1.1 Intermediate Representation. During transpilation, DNN is represented as a common intermediate representation (IR). Models trained with various frameworks are converted into IR.

Figure 3 shows an example of IR. It not only contains information on the structure of the computational graph, but additional information such as memory order, and the attributes of operation as well. This information is used in the optimization phase.

2.1.2 Model Converter. Model converter converts models trained with other DNN frameworks into IR. By using common IR format in the optimization and the generation phases, many modules are reused for various frameworks. In this paper, we have implemented converters for Caffe, Keras and Chainer.

2.1.3 Optimization Rules. IRs generated by the model converter are optimized based on several optimization rules. WebDNN has many rules to transform sub structures of a computation graph into a more optimal structure. Examples of optimization rules are as follows:

MergeElementwise
Activation function such as ReLU ($f(x) = max(0, x)$) is one of the most important operation in DNN. Most of these operations are element-wise operations and require low computation power. Therefore when doing these operations on the GPU, the dispatching operations in JavaScript become a bottleneck situation. This causes the GPU pipeline to stall and it is not able to bring out the complete performance of the GPU.

In MergeElementwise rule, these element-wise operations are concatenated with heavier operations such as fully connected layer and convolution. It reduces kernel dispatching to only once and hides the latency of dispatching these kernels.

MergeAffine
Affine transform is also frequently used in DNN. In this rule, affine transform and other operations such as fully connected layer and scaling layer (in batch normalization) are concatenated.

For example, $f(x) = diag(s)(Wx)$ where s is the scaling factor, W is the weight of the fully connected layer, the formulation transforms to $f(x) = (diag(s)W)x$ and $diag(s)W$ is pre-computed within the graph transpiler.

Inplace
For each operation, if input variables are not used after this operation, the memory allocated for these variables can be reused for output variables. This optimization reduces memory consumption and overhead time of memory allocation.

OptimizeSgemm
SGEMM (Single-precision General Matrix-Matrix product) is one of the most heavy operations in DNN. The performance is largely

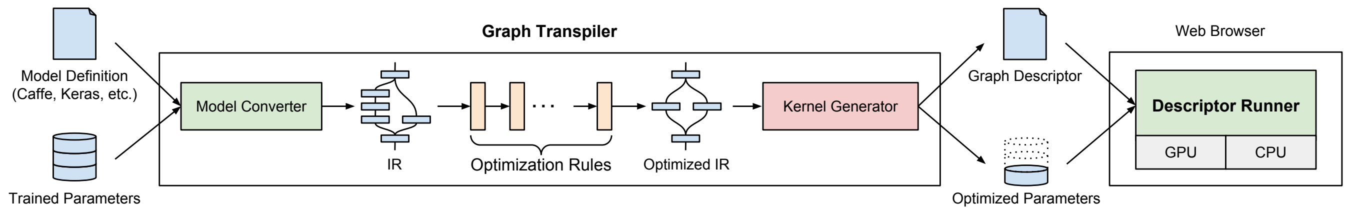


Figure 2: Pipeline of WebDNN

```
<Linear inputs={
  x: <Variable shape=[1, 2048] order=NC>,
  w: <Constant shape=[2048, 1000], order=CN>
 }, outputs={
  y: <Variable shape=[1, 1000], order=NC>
 }, attributes=[PostElementwise]>
<AxiswiseBias inputs={
  x: <Variable shape=[1, 1000] order=NC>,
  b: <Constant shape=[1000], order=C>
 }, outputs={
  y: <Variable shape=[1, 1000], order=NC>
 }, attributes=[
  PostElementwise, Inplace, Axiswise[C]]>
```

Figure 3: Example of intermediate representation of WebDNN.

affected by the layout of matrices in the memory. This rule optimizes the data order of input and the output variables.

2.1.4 Kernel Generator. Kernel generator generates the graph descriptor based on IR. A graph descriptor contains information about DNN model, which is required for it to execute on the web browser. Each kernel generator supports one backend. We have implemented three types of kernel generators: WebGPU, WebAssembly, and Fallback. By using these three kernel generators, WebDNN can work on all modern browsers.

2.1.5 Weight Compression. The weights data of DNN models are very large. ResNet50 is popular image recognition model and its weights data is about 100 MB. Transmitting these data on internet requires a lot of time. Therefore, WebDNN supports model compression by 8bit approximation [2] and "deflate" compression (using zlib library). Using this feature, the weight data of ResNet50 can be compressed to about 20 MB [1].

2.2 Descriptor Runner

Descriptor runner is a JavaScript library to load a generated graph descriptor and execute the corresponding DNN model on the web browser.

2.2.1 WebGPU. WebGL has been widely used as GPU API in JavaScript. However, this API is designed for graphics processing and there are many constraints in general purpose computing. For example, all matrix data should be converted to the image texture format. These constraints become serious overhead.

WebDNN uses new GPU API called WebGPU. It supports compute shader, which is a shader for general purpose computing. By using WebGPU, we can use GPU for general purpose computing with only little overhead. We have implemented this API on the open source web browser, WebKit and it has already been shipped in Safari Technology Preview.

2.2.2 Multi backend support. When Descriptor runner runs on web browsers which do not support WebGPU, it uses CPU-based environment called WebAssembly. WebAssembly is supported by many modern web browsers, and can execute computationally expensive tasks more efficiently than JavaScript. In graph descriptor for WebAssembly, Eigen [4] is used to perform matrix multiplication as fast as possible. Moreover, a WebWorker thread is launched for executing DNN models in order to avoid blocking of the user interface.

Fallback backend works when the web browser does not WebGPU nor WebAssembly. This backend is compatible with ECMAScript 3, so almost all web browsers can interpret it.

Backends that are compatible to the user's web browser are automatically identified and the corresponding graph descriptor is loaded. Therefore, application developers can implement it with a single interface regardless of what backend is used.

3 USAGE

In this section, the typical usage of WebDNN is illustrated with the example of converting ResNet50 model from Keras for WebDNN. ResNet50 accepts a color image of 224×224 pixels and classifies it into 1,000 object classes.

3.1 Graph transpilation

The first step is to export the model in the standard format of Keras.

```
from keras.applications import resnet50
model = resnet50.ResNet50(include_top=True,\
  weights='imagenet')
model.save("resnet50.h5")
```

Then, transpile the model file trained with Keras into the graph descriptor. The following single command accomplishes it.

```
python bin/convert_keras.py resnet50.h5 \
  --input_shape '(1,224,224,3)'
```

For Caffe, almost the same procedure can be applied to Caffemodel file. For Chainer, there are no separate model file. Graph

[1]The size may be still large for mobile network to load every time, but HTML5 offline application feature helps to cache it on the device permanently.

transpiler analyzes the on-memory computational graph generated by the "define-by-run" scheme.

3.2 Run DNN Model on Webpage

The procedure to execute a graph descriptor generated by the graph transpiler in a web browser is as follows.

```
let runner;

async function init() {
  // (1) Initialize DescriptorRunner
  runner = await WebDNN.prepareAll('./model');
}

async function run() {
  // (2.1) Set input data
  runner.inputViews[0].set(loadImageData());

  // (2.2) Run DNN model
  await runner.run();

  // (2.3) Show result
  console.log(WebDNN.Math.argmax(
              runner.outputViews[0]));
}
```

First, initialize the descriptor runner and load model data. When performing this, WebDNN automatically identifies the running environment and graph descriptors on the server, and then selects the best backend. The placeholders of input / output variables of DNN are returned. This initialization has to be called once. After this, the DNN can be executed by setting the input and call "run" method. Figure 1 shows a screenshot of the demo application which converted Neural Style Transfer model [6], which is running on Chrome for Android web browser.

4 EVALUATION

We evaluated the model execution speed of WebDNN with Keras.js. We transpiled image classification model VGG16 [8], ResNet50 [5] using these libraries and measured the model execution time for one image on a web browser. The hardware is Mac Book Pro early 2015, Intel Core i5 2.7 GHz CPU, 16 GB Memory, Intel Iris Graphics 6100 GPU. The web browser is Safari Technology Preview 30.

The experimental results are shown in Fig. 4. When WebDNN with WebGPU backend and all optimizations are enabled, 217x speedup on VGG16 and 207x speedup on ResNet50 are achieved compared to Keras.js with a CPU backend.

Additionally, we measured the speed on out-of-the-box Windows 10 with Internet Explorer 11 environment, which represents the environment in ordinary office. The hardware is Intel Core i5-3337U 1.80GHz CPU, Intel HD Graphics 4000 GPU. The execution speeds of ResNet50 are 9,495ms in WebDNN WebAssembly backend [2], 8,115ms in Keras.js GPU backend and 27,070ms in Keras.js CPU backend. In the balance of GPU / CPU power in the hardwares, WebAssembly (CPU-based) backend of WebDNN was comparable with GPU backend of Keras.js. In each DNN model and backend,

[2]IE does not have WebGPU support.

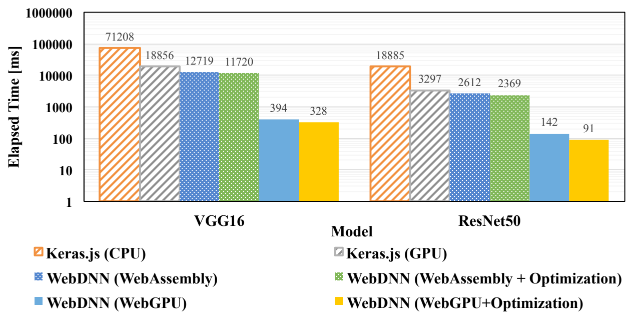


Figure 4: Processing time for each library and backend. "Optimization" indicates the optimization rules described in Sec. 2.1.3 are applied to the graph descriptor. Vertical axis is logarithmic scale and lower is better.

WebDNN obtains better results in terms of speed. More speed improvement is observed when the optimizations are applied in the graph transpiler. We think these results came from WebDNN implementing the optimization strategy specialized for DNN inference on web browsers, that is quite different from the ordinary environment.

5 CONCLUSION

In this project, we developed a framework for executing a DNN model on web browsers in order to provide an installation-free DNN execution environment. By advanced optimizations of computational graph and support from the latest web specifications, we achieved a maximum of more than 200x speedup compared to the existing framework.

ACKNOWLEDGMENTS

This work was partially supported by JST CREST Grant Number JPMJCR1403, Japan. This work was also partially supported by the Ministry of Education, Culture, Sports, Science and Technology (MEXT) as "Seminal Issue on Post-K Computer".

REFERENCES

[1] Leon Chen. 2016. Keras.js. (2016). https://github.com/transcranial/keras-js.
[2] Tim Dettmers. 2016. 8-Bit Approximations for Parallelism in Deep Learning. In *Proceedings of the International Conference on Learning Representations (ICLR).*
[3] Waylon Flinn. 2016. weblas. (2016). https://github.com/waylonflinn/weblas.
[4] Gaël Guennebaud, Benoît Jacob, and others. 2010. Eigen v3. http://eigen.tuxfamily.org. (2010).
[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR).*
[6] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. 2015. Perceptual Losses for Real-time Style Transfer and Single Image Super-Resolution. In *Proceedings of the International Conference on Machine Learning (ICML).*
[7] Seyyed Salar Latifi Oskouei, Hossein Golestani, Matin Hashemi, and Soheil Ghiasi. 2016. CNNdroid: GPU-Accelerated Execution of Trained Deep Convolutional Neural Networks on Android. In *Proceedings of the 2016 ACM on Multimedia Conference (MM '16).* 1201–1205.
[8] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *Proceedings of the International Conference on Learning Representations (ICLR).*
[9] Theano Development Team. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints* abs/1605.02688 (May 2016). http://arxiv.org/abs/1605.02688