MIL WebDNN Documentation

Release 1.0.0

MIL

Jul 09, 2017

Contents

This is the WebDNN documentation.

CHAPTER 1

Contents

Tutorial

Introduction to WebDNN

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. As another solution, WebDNN highly optimizes the DNN models and executes them on the web browsers in end-user devices.

Key features of WebDNN are the follows.

- Installation-free
- Non overhead interface
- Inference-phase-specialized optimization

Installation-free

WebDNN executes DNN models on a web browser. Usually, web browsers are already installed on end-user devices already 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.

There are a few number of major web browsers, and they have different set of features that can be used for acceleration. WebDNN have several sterategies to execute DNN model as speedy as possible in each web browser.

Non overhead interface

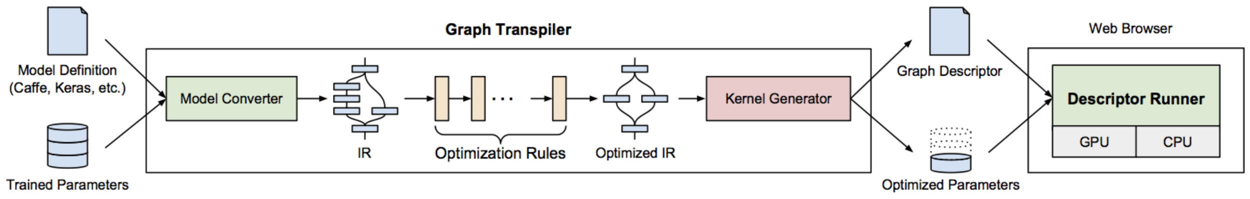
JavaScript is a standard programing language running on web browsers. It is executed by an interpreter. Therefore, it requires computing overhead and it cannot completely harness the capacity of the CPU. The same problem is encountered in GPU. Modern web browsers support WebGL, which is a JavaScript 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.

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. Execution of DNN consists of two phases, the training phase and the inference phase, and they requires different optimization sterategies. 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.

Framework structure



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.

Graph transpiler is the offline module to transpile the model. It is implemented in python (version 3.6) and only application developers need to run it. It outputs the 'graph descriptor' files, which consist of JavaScript and binary weight data.

Descriptor runner is the online module to run the graph descriptor on the web browser of the end-users. It is JavaScript files. Application developers have to use the API provided by the library to supply input to the model and display output.

Setting up the application development environment is described in setup page. You can find examples of steps to use models from Caffe, Keras, Chainer in WebDNN in the example directory.

Setup Guide (for Mac / Linux)

For Windows users, jump to setup_windows

Downloading code

```
git clone https://github.com/mil-tokyo/webdnn
```

Installing WebGPU environment

WebDNN runs fastest on browsers which support WebGPU. Currently, only Safari Technology Preview on macOS supports it.

https://developer.apple.com/safari/technology-preview/

If you don't have such environment, WebAssembly backend can be used. It is supported by most modern browsers. (Note: IE and Safari does not support WebAssembly, but asm.js code is automatically generated along with WebAssembly code, and gives similar performance.)

Installing python package

This framework requires python3.6+.

```
cd webdnn
python3 setup.py install
```

This will install `webdnn`.

If you want to convert models of Caffe or Chainer, install chainer package.

```
pip install chainer
```

(Currently, tested with `chainer==2.0` and `chainer==1.23`)

Installing Emscripten and Eigen

If you want to enable WebAssembly backend, em++ command from Emscripten is required. You can skip this section if you try WebGPU backend only.

To setup Emscripten which supports WebAssembly,

```
git clone https://github.com/juj/emsdk.git
cd emsdk
./emsdk install sdk-incoming-64bit binaryen-master-64bit
./emsdk activate sdk-incoming-64bit binaryen-master-64bit
```

(see also http://webassembly.org/getting-started/developers-guide/)

To enable em++ command, you need to type command on the shell.

```
source ./emsdk_env.sh
```

Eigen is needed as the library.

```
wget http://bitbucket.org/eigen/eigen/get/3.3.3.tar.bz2
tar jxf 3.3.3.tar.bz2
```

To enable Eigen to be included on compile, you need to type command on the shell.

```
export CPLUS_INCLUDE_PATH=$PWD/eigen-eigen-67e894c6cd8f
```

Notes on python environment

Emscripten requires `python2` command, you need to setup python environment which `python` (or `python3`) is python 3.6+ and `python2` is python 2.7. pyenv may help to setup such environment (see also).

Setup Guide (for Windows)

No browser on Windows supports WebGPU, but you can still develop applications using WebDNN. Commands have to be used on command prompt.

Downloading code

```
git clone https://github.com/mil-tokyo/webdnn
```

If you do not have git, zip file is also available: https://github.com/mil-tokyo/webdnn/archive/master.zip

Installing python environment

If you do not have python environment, install python environment.

Anaconda is the popular installer: https://www.continuum.io/downloads

When installing, adding PATH to system is optional.

This framework requires python3.6+. This document is based on Anaconda 4.4.0.

If you want to convert models of Caffe or Chainer, install chainer package. Refer to Chainer document.

(Currently, tested with `chainer==2.0` and `chainer==1.23`)

Installing Emscripten and Eigen

If you want to enable WebAssembly backend, em++ command from Emscripten is required. You can skip this section if you try WebGPU backend only.

To setup Emscripten which supports WebAssembly,

Download "Emscripten SDK Offline Installer (emsdk-1.35.0-full-64bit.exe)" from http://kripken.github.io/emscripten-site/docs/getting_started/downloads.html.

When installing, you should add PATH of `emcc` to PATH, but PATH of `python` should not be added (as it conflicts with Anaconda).

Eigen is needed as the library. Download latest source from http://eigen.tuxfamily.org/index.php?title=Main_Page and decompress.

Setting for using proper python

python environment (3.6.x) of Anaconda have to be executed with `python3` command. To accompilsh it, create a file named `python3.bat` and fill with the following content.

```
"C:\ProgramData\Anaconda3\python.exe" %*
```

python environment (2.7.x) of Anaconda have to be executed with `python` command. To accompilsh it, create a file named `python.bat` and fill with the following content.

```
set CPLUS_INCLUDE_PATH=PATH\TO\EIGEN
"C:\Program Files\Emscripten\python\2.7.5.3_64bit\python.exe" %*
```

Concrete path depends on the version of Emscripten and installer configuration. `PATH\TO\EIGEN` have to be replaced by the directory where Eigen is downloaded. The file `PATH\TO\EIGEN\Eigen\Dense` should exists.

Verification of Emscripten and Eigen installation

Create a file named `hello.cpp`:

```cpp
#include <Eigen/Dense>
#include <iostream>

int main()
{
  std::cout << "hello world" << std::endl;
  return 0;
}
```

Then, try to compile it into WebAssembly:

```
em++ hello.cpp -O3 -s WASM=1 -o hello.html
```

If Emscripten works well, files such as `hello.wasm`, `hello.html` are generated.

Installing graph transpiler

Install graph transpiler for Anaconda python environment.

At the top directory of WebDNN is cloned, type

```
python3 setup.py install
```

Here, `python3.bat` have to exist on the current directory.

Running example

```
cd example\mnist
python3 train_mnist_chainer.py
```

Here, `python3.bat`, `python.bat` have to exist on the current directory.

Graph descriptor files for each backend are generated.

In Emscripten 1.35, files `before.js`, `load-wasm-worker.js` seem to be generated on current directory (it seems that latest Emscripten for linux does not require them). You have to move them to the output directory.

Use with Keras Model

In this tutorial, we'll convert ResNet50[1] classification model pretrained in Keras[2] into WebDNN execution format.

1. Export Keras pretrained model

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

[1]

11. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition", IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.

[2] https://keras.io/applications/#resnet50

2. Convert Keras model to our computation graph format

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

   At least you need to specify the model file and the shape of input array.

   Now you have the "graph descriptor" to execute on the web browsers in output directory.

3. Run on web browser

   This document illustrates the essence of running the "graph descriptor" to execute on the web browsers. In `convert_keras` example directory, the complete codes for training and converting a Keras model and running it on the web browsers can be found.

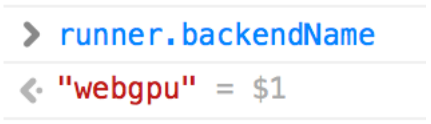
   First, You have to initialize `DescriptorRunner` and load model data.

   ```
   let runner = await WebDNN.load('./output');
   ```

   WebDNN automatically select the best backend based on Browser type and compiled model data on the server.

   You can check the backend type

   ```
   console.log(runner.backendName);
   ```



```
> runner.backendName
<· "webgpu" = $1
```

   Then you can get input and output variable references (`SymbolicFloat32Array` type).

   ```
   let x = runner.getInputViews()[0];
   let y = runner.getOutputViews()[0];
   ```

   That's all for initialization. You only have to do this at once in the application.

   Let's classify this image.

   First, set input data.

   ```
   // loadImageData() returns image data as Float32Array
   x.set(loadImageData());
   ```

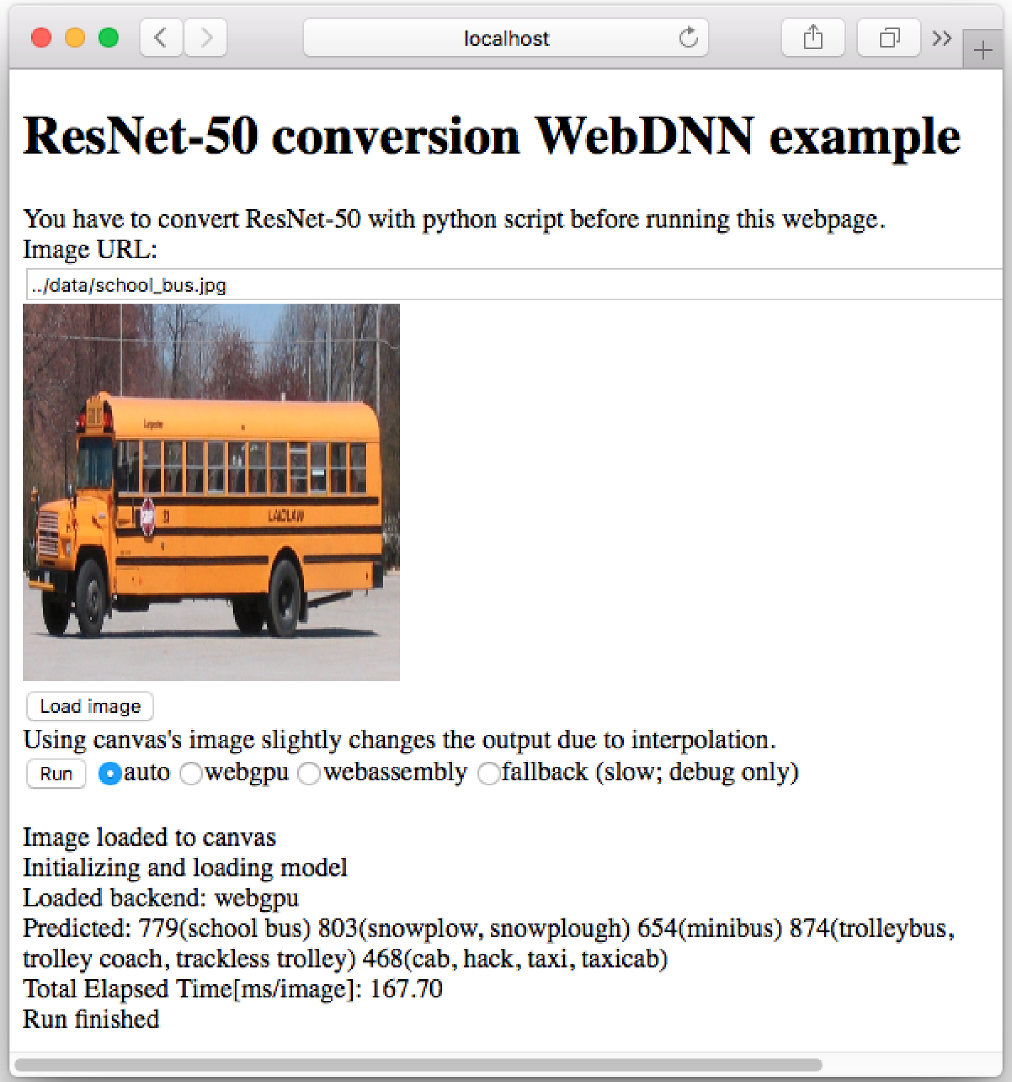
   Next, run model.

   ```
   await runner.run();
   ```

   That's all.

   Show computed vector and predicted label.

```
let y_typed_array = y.toActual();
console.log('Computed vector', y_typed_array);
console.log('Predicted Label', WebDNN.Math.argmax(y_typed_array));
```



ResNet-50 conversion WebDNN example

You have to convert ResNet-50 with python script before running this webpage.
Image URL:
../data/school_bus.jpg

Using canvas's image slightly changes the output due to interpolation.
[✓ auto] webgpu webassembly fallback (slow; debug only)

Image loaded to canvas
Initializing and loading model
Loaded backend: webgpu
Predicted: 779(school bus) 803(snowplow, snowplough) 654(minibus) 874(trolleybus, trolley coach, trackless trolley) 468(cab, hack, taxi, taxicab)
Total Elapsed Time[ms/image]: 167.70
Run finished

Congratulation! `LabelID:779` is `"School bus"` in ImageNet. It looks work well.

References

Use with Caffemodel

In this section, you will learn about how to convert your caffemodel into `GraphDescriptor`, and run `GraphDescriptor` on your web page.

1. Convert Caffemodel into GraphDescriptor

See jupyter notebook (caffenet_conversion.ipynb) to see how to convert model offline.

2. Run on web browser

In this section, I'll describe how to run generated descriptor on web browser. you can view complete codes (HTML and JS file) in `/example/convert_caffe`.

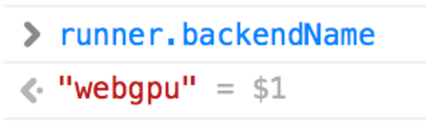
First, You have to initialize `DescriptorRunner` and load model data.

```
let runner = await WebDNN.load('./output');
```

WebDNN automatically select the best backend based on Browser type and compiled model data on the server.

You can check the backend type

```
console.log(runner.backendName);
```



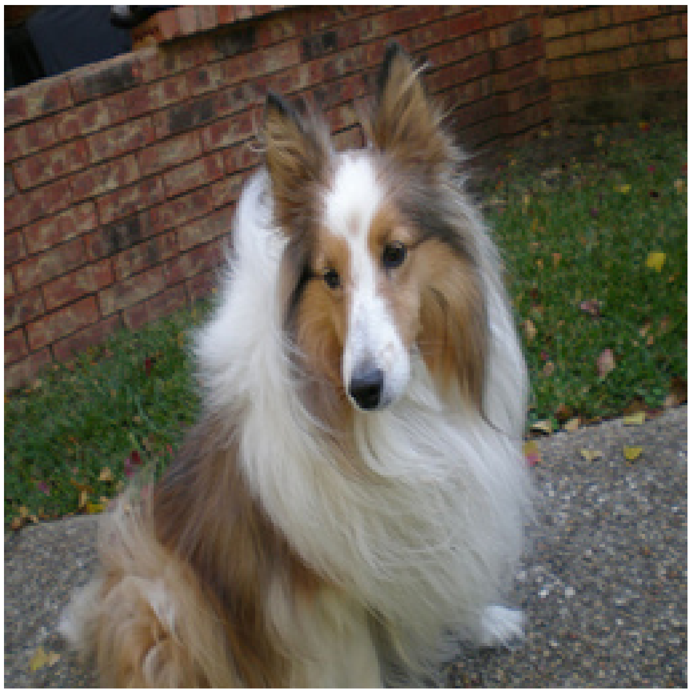
```
> runner.backendName
< "webgpu" = $1
```

Then you can get input and output variable references (`SymbolicFloat32Array` type).

```
let x = runner.getInputViews()[0];
let y = runner.getOutputViews()[0];
```

That's all for initialization. You only have to do this at once in the application.

Let's classify this image.



First, set input data.

```
// loadImageData() returns image data as Float32Array
x.set(loadImageData());
```

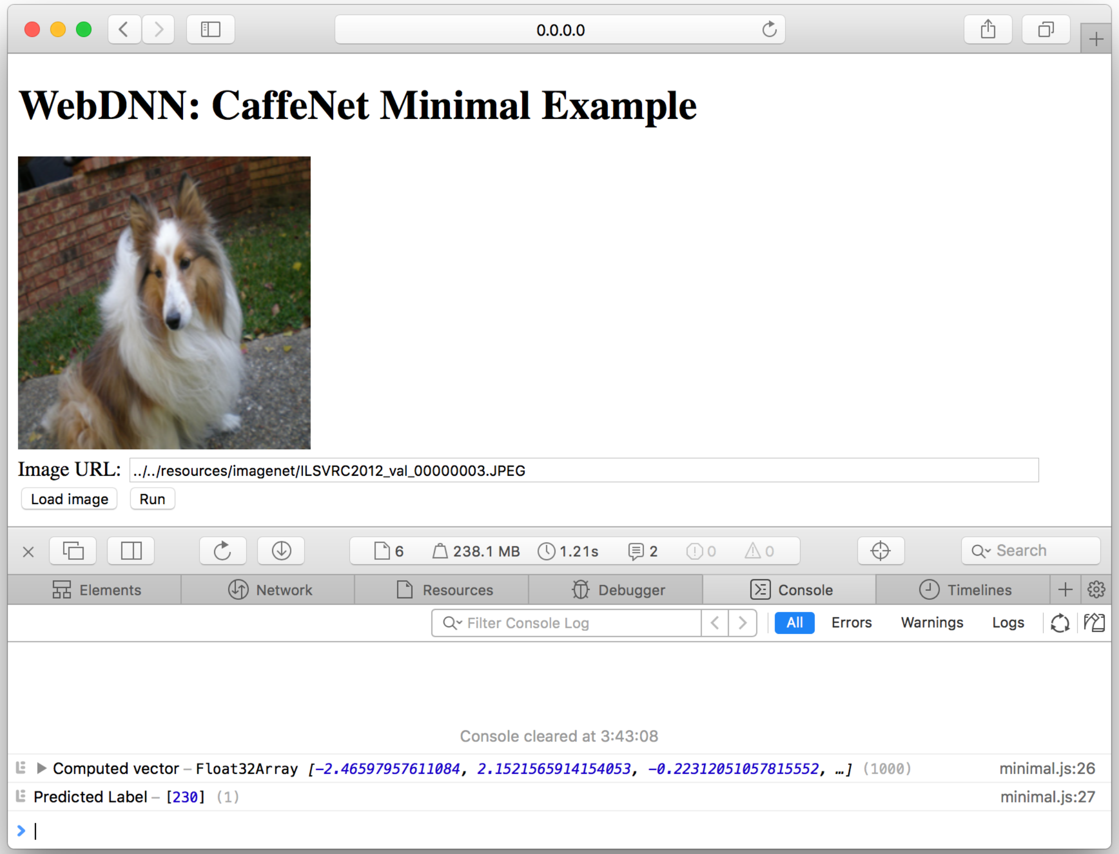
Next, run model.

```
await runner.run();
```

That's all.

Show computed vector and predicted label.

```
let y_typed_array = y.toActual();
console.log('Computed vector', y_typed_array);
console.log('Predicted Label', WebDNN.Math.argmax(y_typed_array));
```



```
Console cleared at 3:43:08
▶ Computed vector – Float32Array [-2.46597957611084, 2.1521565914154053, -0.22312051057815552, …] (1000)   minimal.js:26
Predicted Label – [230] (1)   minimal.js:27
```

Congratulation! `LabelID:230` is `"Shetland sheepdog"` in ImageNet. It looks work well.

Use with Chainer Model

In this tutorial, we'll convert ResNet50[1] classification model pretrained in Chainer[2] into WebDNN execution format.

1. Load chainer pretrained model

```
import chainer

model = chainer.links.model.vision.resnet.ResNet50Layers()
```

2. Execute model with dummy data. In chainer, computation graph are defined by run. Therefore we need execute model at least once to construct the graph.

```
import numpy as np

x = chainer.Variable(np.empty((1, 3, 224, 224), dtype=np.float32))
y = model(x, layers=["fc6"])["fc6"]
```

3. Convert chainer computation graph to our computation graph format

```
from webdnn.frontend.chainer import ChainerGraphConverter

graph = ChainerGraphConverter().convert_from_inout_vars([x], [y])
```

4. Generate and save execution information.

```
from webdnn.backend.interface.generator import generate_descriptor

exec_info = generate_descriptor("webgpu", graph)
exec_info.save("./output")
```

References

Tips

EcmaScript5 support

In some cases, you want to support older browser such as IE11, which is the default browser for Windows 7 and 8. To support IE11, the JavaScript code have to be compliant with EcmaScript5. In this document, how to convert the code is described.

Use webdnn.es5.js and polyfill

In syntax level, `webdnn.js` uses statements like `await`, which is not compatible with EcmaScript5. Instead, you can use `webdnn.es5.js` which does not use such statements. This library can be compiled with `tsc -p tsconfig.es5.js` on `src/descriptor_runner` directory.

In standard library level, `webdnn.js` uses `Promise` and `fetch`. You need to supply these objects by loading polyfill.

[1]

11. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition", IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.

[2] http://docs.chainer.org/en/latest/reference/links.html#chainer.links.ResNet50Layers

In colusion, you need to insert the following statements in the html.

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-polyfill/6.23.0/polyfill.
↪min.js"></script>
<script src="https://cdn.polyfill.io/v2/polyfill.min.js?features=fetch"></script>
<script src="../../lib/inflate.min.js"></script>
<script src="../../dist/webdnn.es5.js"></script>
```

Converting your own code

If your own JavaScript code uses newer statements like `await`, you can also convert it automatically by babel.

Install babel

```
npm install babel-cli babel-preset-env babel-preset-es2015 babel-plugin-transform-
↪regenerator
```

Converting your code

```
babel foo.js --out-file foo.es5.js --presets env,es2015 --plugins transform-
↪regenerator
```

Using web camera in Safari

Background

Most modern web browsers support WebRTC, which supports easy access to camera from web page scripts. However, Safari (on Mac) does not support WebRTC, so the workaround is to use Flash.

There is a good library "webcamjs" to wrap the browser difference. To make this library to work, security setting is needed. This document describes how to change the setting.

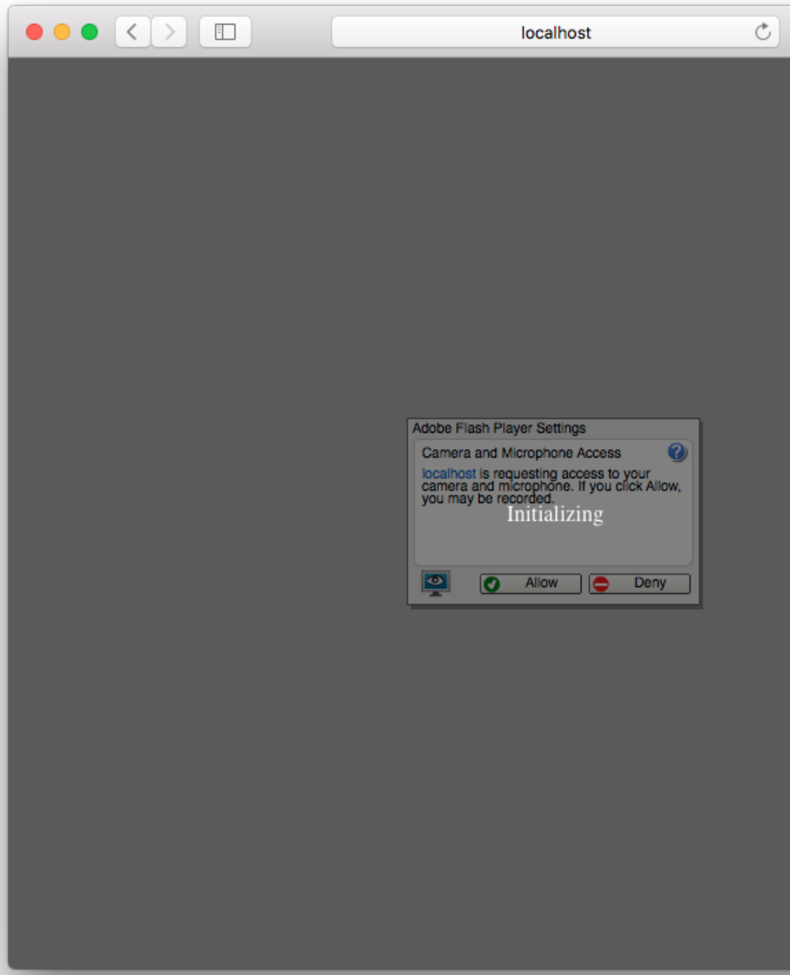
Flash player

You need to install and enable Flash player *on the target website* first.

Visit official instruction: https://helpx.adobe.com/flash-player/kb/enabling-flash-player-safari.html

Security setting for camera

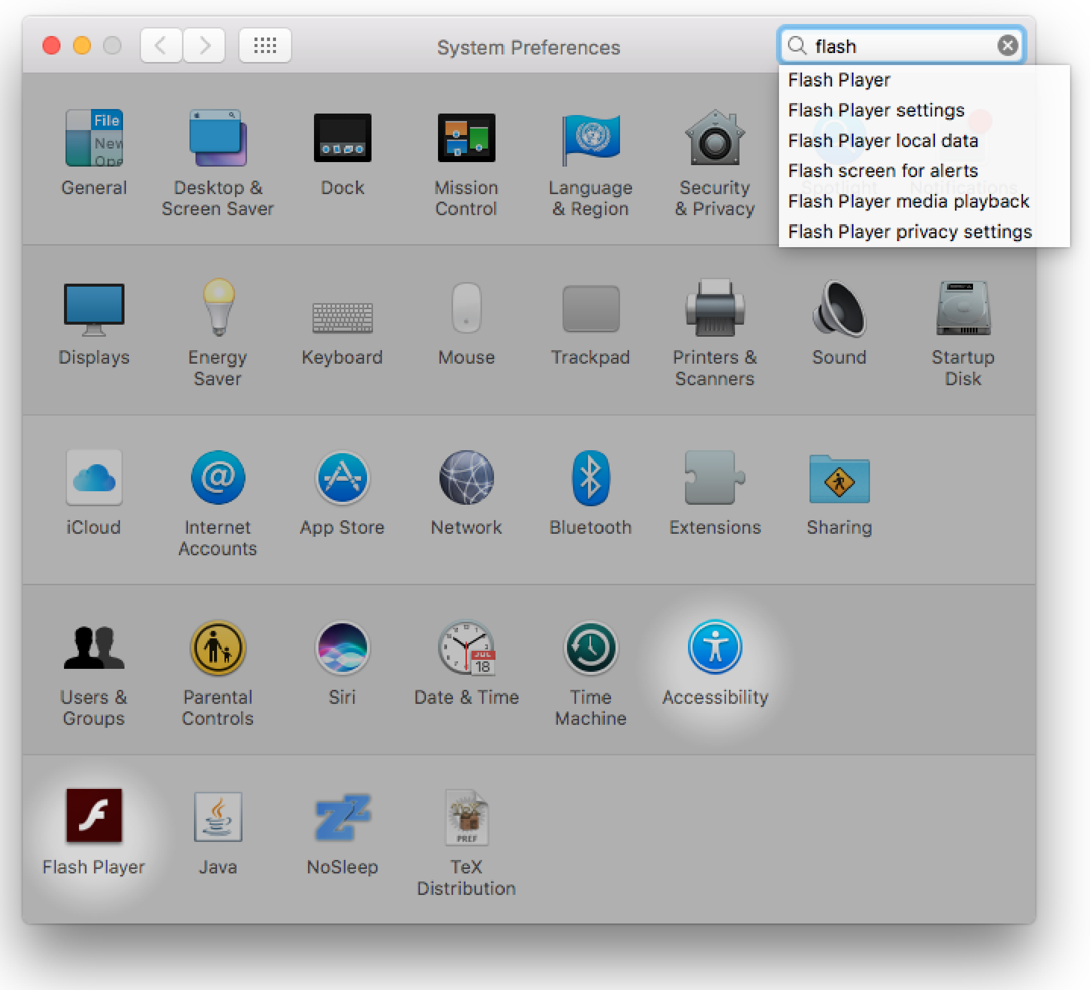
When you visit a website that want to access your camera, special permission is needed.



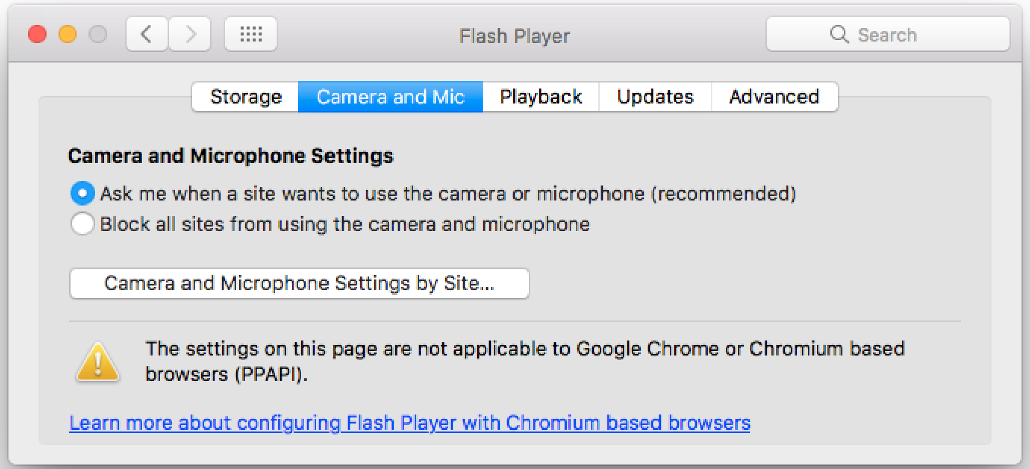
localhost is requesting access to your camera and microphone. If you click Allow, you may be recorded.

If you are lucky, you have only to click "Allow" button.

Sometimes, the button does not work. In this case, you need to change the setting from "System Preferences".



1. Open Flash Player
2. Click "Camera and Mic" tab and open "Camera and Microphone Settings by Site..." dialog



Flash Player

Storage | Camera and Mic | Playback | Updates | Advanced

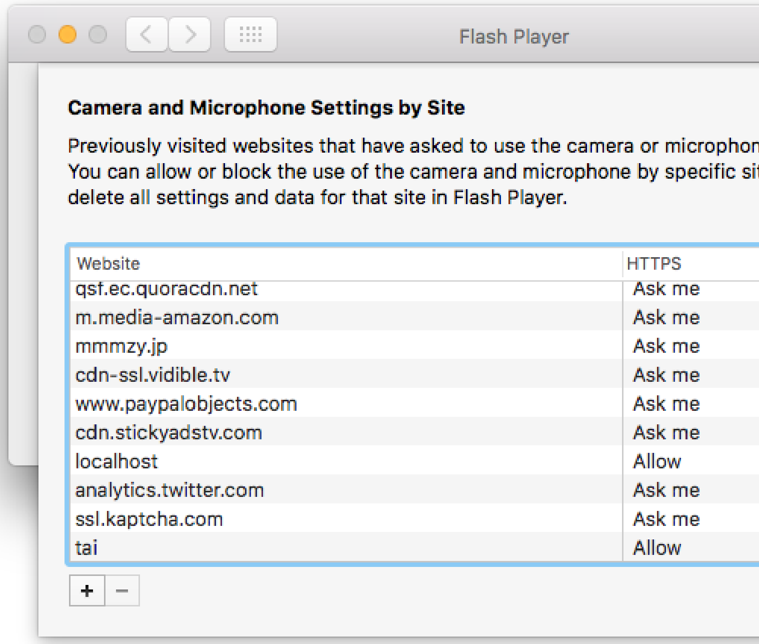
Camera and Microphone Settings

- [✓ Ask me when a site wants to use the camera or microphone (recommended)]
- Block all sites from using the camera and microphone

Camera and Microphone Settings by Site...

The settings on this page are not applicable to Google Chrome or Chromium based browsers (PPAPI).

Learn more about configuring Flash Player with Chromium based browsers



Flash Player

Camera and Microphone Settings by Site

Previously visited websites that have asked to use the camera or microphon
You can allow or block the use of the camera and microphone by specific si
delete all settings and data for that site in Flash Player.

Website	HTTPS
qsf.ec.quoracdn.net	Ask me
m.media-amazon.com	Ask me
mmmzy.jp	Ask me
cdn-ssl.vidible.tv	Ask me
www.paypalobjects.com	Ask me
cdn.stickyadstv.com	Ask me
localhost	Allow
analytics.twitter.com	Ask me
ssl.kaptcha.com	Ask me
tai	Allow

3. Change configuration to "Allow" for target website

API Reference

Graph Transpiler

Base Classes

Graph

Operator

Variable

Attribute

Axis

Order

Placeholder

OptimizeRule

Operators

AveragePooling2D

AxiswiseBias

AxiswiseScale

Concat

Convolution2D

Deconvolution2D

ElementwiseSum

Elu

Flatten

Linear

LocalResponseNormalization

MaxPooling2D

Relu

ScalarAffine

Sigmoid

Softmax

Tanh

Variables

ConstantVariable

Optimize Rules

FIXME: DOCS

Descriptor Runner