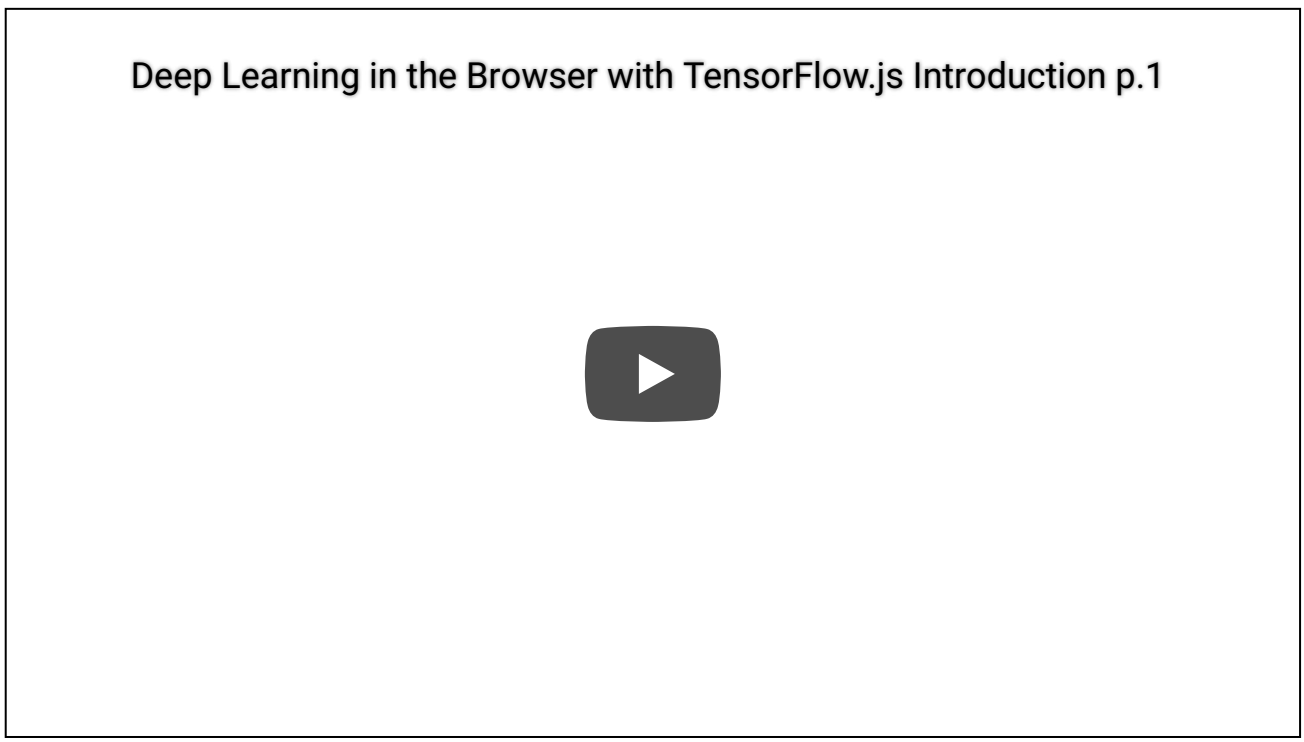
Deep Learning in the Browser with TensorFlow.js Tutorial Introduction - TensorFlow.js Tutorial p.1



Hello and welcome to a TensorFlow.js tutorial series. TensorFlow.js, previously deeplearn.js, is a JavaScript library for training and deploying ML models in the browser.

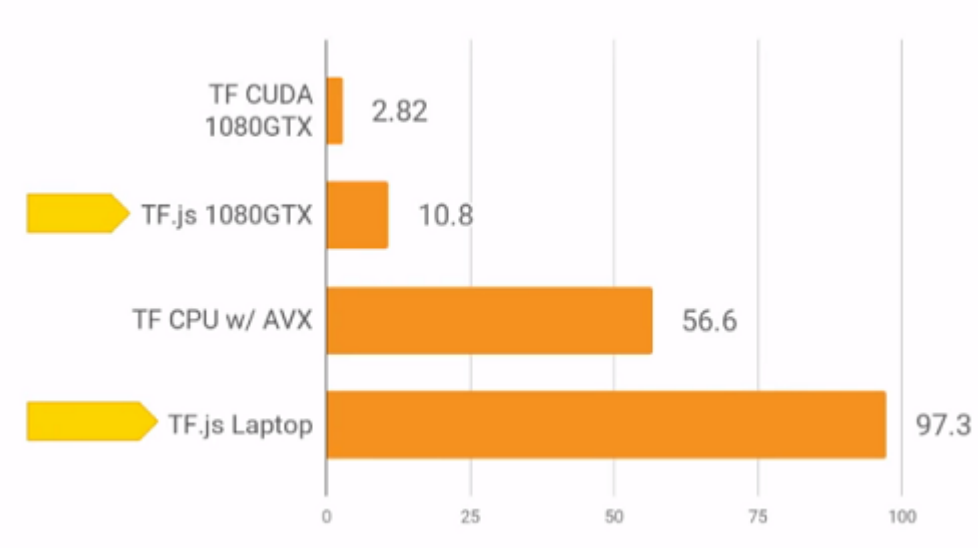
Why TensorFlow.js over something like TensorFlow lite, which also can run on things like phones?

What's so special about this? Being javascript, it runs anywhere, and there are no installs required. Previously, if you wanted users to be able to run an ML model, you pretty much needed to host that model and provide the processing power, or your users needed to download and install all of the packages required which would have been absurd for releasing to the general public.

With this hurdle gone, all kinds of possibilities begin to emerge. Rather than serving ads, your site could instead use user processing power to, say, help detect cancer early.

TensorFlow.js runs on WebGL, which can either make use of a user's CPU, or GPU if available. Being WebGL, it's not as efficient as CUDA, but it's also not bad:

Single-inference speeds of Python TensorFlow vs TensorFlow.js:



Source: TensorFlow Dev Summit 2018

Sold! What do I need to start?

TensorFlow.js comes with two major ways to work with it: "core" and with "layers." If you're familiar with working more with tensorflow, then the core library is probably more your style. If you're more familiar with working with a higher-level API like Keras, then the Layers library is what you're after. We'll be using the Layers API to start.

What do I need to know to follow along?

If you already are comfortable with TensorFlow, carry along with me. I am a very basic JavaScript developer, I will do my best to explain pretty much all of the JS as it relates to our TensorFlow.js models/code.

If you do not currently know TensorFlow or Neural Networks, you might want to check out: Introduction to Neural Networks and Introduction to Deep Learning with TensorFlow to get a better idea, but here's an ultra simplified breakdown:

TensorFlow is a package created to help you perform operations on Tensors.
What's a Tensor?! An array. It's that simple. It's a type of array, and not all arrays are the same, but, it's an array.
TensorFlow as a library is here to help you form operations on tensors (arrays), like multiplication, but also when it comes to performing complex optimizations like trying to minimize loss.
In our case, we want to use TensorFlow.js for neural networks. All neural networks are, are these relationships of nodes, where each node has a connection to another node, or a few nodes. Nodes have certain numerical values, and their connections carry various weights that are multiplied against the node's value.
The whole point is to adjust those weights (and sometimes biases), in order to hopefully get the #s you're wanting at the final layer (the output layer) to be what your training data is asking for. You're hunting for some combination of weights between your nodes that best fits your data.

That's really it. It's just a challenging mathematical problem that we'd rather not work out by hand optimizing and fitting all of those weights, but machines with many cores of processors can power through it with some accompanying mathematics. If you want to dive deeper into how these things work, visit the above tutorials, I don't plan to break down neural networks here any further than what I just did.

Working with the TensorFlow.js package

As of my writing this, the TensorFlow.js library has a ~90% coverage of the entire TensorFlow library. This means, most likely, the thing you are doing in TensorFlow in, say, Python, is also possible to do in TensorFlow.js. Most of the time, I can guess the names of things without needing to even look it up, just prepare to change your naming convention to camelCase.

To train, to transfer, or just to infer?

Just doing inference, with a model you've already trained:

It might often be the case that your application is actually all about inference, not really needing to be trained. If this is the case, then you should use Keras most likely to build and train your model on your own massive machinery and bajillion samples. Then, you can use something like the package tensorflowjs to convert the keras model. More info on this here: Importing a Keras model into TensorFlow.js.

Transfer learning:

Next, maybe you want to take a model that already has been trained, but allow your users to train that model something else. An example of this would be something like object detection. You could take a pre-trained model that does object detection, then let a user take a few images of something, label it with bounding boxes, then, using transfer learning, train a model to detect this new type of object, custom to exactly whatever the user was wanting. Again here, you'd probably pre-train it with Keras, convert the model, then you'd need to write some TensorFlow.js code to train the model.

It's my guess that transfer learning will be the most meaningful and impacting capability of TensorFlow.js, since the models can still be quick to train, but also do exceptionally powerful things. Who knows though.

Training AND inference:

Finally, maybe you want to let your users create a totally fresh model, doing who knows what!

Keep in mind processing times. On a single inference, TensorFlow.js is 5x slower. Now imagine batching with 32, or 128, or whatever you might be batching with. Yikes! Keep this in mind. Despite this, you really can actually train smaller models on easier challenges on many devices.

If you want some examples of what you can do, check out the demos here: TensorFlow.js getting started and demos.

Okay, let's get started!

TensorFlow.js is, of course, a JavaScript library. To use it, we need to load it in. To do this, we need to place the following in our code, probably in the head tags of our .html document:

```
<html>
  <head>
    <!-- Loading in TensorFlow.js version 0.11.2 -->
    <script src="https://cdn.jsdelivr.net/npm/@tensorflo
  </head>
  <body>
  </body>
</html>
```

You can also use NPM. For more information on this, see the Getting started w/ TensorFlow.js docs

Next, let's make ourselves a model!

For this, you can either create another set of script tags in your html file, or, what I would suggest is to instead open your .html file right now in the browser. I am going to use Google Chrome for this, so I right click the html file and choose to open it with Google Chrome.

Once it's open, you probably see nothing, that's what you expect. Now, if you are in Google Chrome, press f12 for the deveoper's console. Next, click on the "Console" tab.

Now we can use this as an interactive session!

To begin we need a model variable to be established. To do this:

```
const model = tf.sequential();
```

Press enter, if you're in the developer console, and you will see `undefined` returned. That's fine. Nothing note-worthy yet is all. Our model is a `tf.sequential` model, which is any model that goes in order. That is, the outputs from one layer are the inputs to the next one in exact order, without skipping/branching.

Now, let's add a layer to our hypothetical model:

```
model.add(tf.layers.dense({units: 1, inputShape: [1]}));

model.add(tf.layers.dense({units: 64, inputShape: [1]}));

model.add(tf.layers.dense({units: 1, inputShape: [64]}));
```

So, for now, we're just going to have a layer that takes an input of only 1 unit. Again, this type of layer is specific, it's a `dense` layer, which is just a generic, fully connected, layer. You can have layers with dropout, recurrent layers, convolutional layers, and more.

Next, let's add some more parameters to our model, specifically the loss calculation and the optimizer of choice:

```
model.compile({loss: 'meanSquaredError', optimizer: 'sgd'});
```

Okay great, now we need our dataset. Our inputs and outputs!

```
const xs = tf.tensor2d([1, 2, 3, 4, 5], [5, 1]);
```

In the above case, we're specifying a 2-dimensional tensor, which a shape of 5x1, which just means we've got 5 samples of 1 element each. Next up, we need some targets, our output layer for training:

```
const ys = tf.tensor2d([2, 4, 6, 8, 10], [5, 1]);
```

After this, we're ready to fit:

```
model.fit(xs, ys)
```

Hey, what's this mention of a "promise?"

When you ask a model to fit, it can obviously take a few moments. This will not cause your page to hang. In fact, if you intend to wait, you need to explicitly wait! Luckily for us, this fitment is quite fast and is already done! A promise in javascript is just an object that can produce a value at some point in the future.

Once fitment is done, we can predict. We can do this with:

```
model.predict(tf.tensor2d([6], [1, 1]))
```

Again though, we get back some information that isn't quite what we were after. In the console, we could use something like .print()

```
model.predict(tf.tensor2d([6], [1, 1])).print()
```

This returns:

```
Tensor
     [[3.4681847],]
```

In our application though, we're going to want the value. To actually get the value, we can use dataSync:

```
model.predict(tf.tensor2d([6], [1, 1])).dataSync();
```

This gives us: `Float32Array [3.46818470954895]`

Not quite right (we're looking for 2x, so 12), but still, we did the thing with the code!

How might we go about getting the right answer? Well, we have an optimizer that, per sample it reads, is only going to optimize so much. We don't have very much data, so this wasn't very much optimizing. We can try to fix this with more epochs (repetitions over our entire dataset). Can one really over-fit a perfectly linear relationship anyway?

We can add parameters to our training as a 3rd parameter like so:

```
model.fit(xs, ys, {epochs:150})
```

```
model.predict(tf.tensor2d([6], [1, 1])).dataSync();
Float32Array [11.820777893066406]
```

...much closer!

If we wanted to also shuffle, we might do:

```
model.fit(xs, ys, {epochs:150, shuffle:true})
```

Let's try seriously fit this bad boy!

```
model.fit(xs, ys, {epochs:15000})
```

```
model.predict(tf.tensor2d([6], [1, 1])).dataSync();
Float32Array [11.999996185302734]
```

Okay, I'll take it, that's pretty darn close to 12! Warning, the above might take a minute or so to fully calculate.

```
model.predict(tf.tensor2d([60], [1, 1])).dataSync();
Float32Array [119.99992370605469]
model.predict(tf.tensor2d([15], [1, 1])).dataSync();
Float32Array [29.999982833862305]
model.predict(tf.tensor2d([21], [1, 1])).dataSync();
Float32Array [41.99997329711914]
```

Cool! Okay, that's an introduction to TensorFlow.js for you! Check out their docs here for more things you can do: TensorFlow.js API docs.

We've just barely scratched the surface here, but hopefully you get the idea.