Contents

1 Browser Based Models with TensorFlow.js

1.1 Training and Inference using Tensorflow.js in JavaScript

We typically think of training a neural network using GPU's or a large data center, but modern web browsers have come a long way: they contain fully fleshed out runtime environments. One of the exciting aspects of Tensorflow.js is that it allows us to train neural networks and perform inference directly from a browser. We'll see how, for example, a user can upload a picture or grab a snapshot from a webcam and then train a neural network and/or perform inference right in the browser, without ever needing to send that image up to the cloud to be processed by a server. This saves time as we cut down on communication costs, allows us to run our models offline, and preserves user privacy.

1.1.1 Getting Your System Ready

We're going to learn how to run all the examples and exercises for this course locally on a single machine. We'll use Chrome for a browser, Brackets for an HTML editor, and the Web Server for Chrome App as our web server. We will also use github.com/lmoroney/dlaicourse as a repository to store homework assignments.

1.1.2 Visualizing API

- At the highest level, we have the *Keras Model* layers API which we've learned how to use in the deep learning specialization.
- Beneath this sits the *Core API*, which is backed by a TensorFlow saved model. Using this Core API, we can interact with either a browser or node.js.
  - A browser sits ontop of webgl, a JavaScript API for rendering graphics.
  - *Node.js* can rely on a TensorFlow CPU, GPU, or TPU.

1.1.3 Building a First Model

Let's start by creating the simplest possible web-page.

```
<html>
<head></head>
<body>
  <hl>First HTML Page</hl>
</body>
</html>
```

We'll next need to add a *script tag* below the head and above the body to load the TensorFlow.js file.

```
<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest"></script>
```

We're going to build a model that learns the relationship between two numbers x, y where their ground truth relationship is $y = 2x - 1$. We will do this in a *separate script block*, that needs to be placed above the `body` tag in your HTML page.

```
<script lang="js">
    const model = tf.sequential();
    model.add(tf.layers.dense({units: 1, inputShape: [1]}));
    model.compile({loss:'meanSquaredError',
```

```
                    optimizer:'sgd'});
    model.summary();
</script>
```

The first line defines a *sequential* model. The second line adds a single hidden layer, itself containing a single hidden neuron. We then compile the model using mean-squared-error as our loss function, which will work well to model a linear relationship $y = 2x - 1$, and we choose stochastic gradient descent as our method of optimization. Note that in the model summary, we'll be told there are two parameters in the model, since we are learning both a weight *and* a bias term. Before closing the script tag, let's insert some data that will be used to train the neural network.

```
const xs = tf.tensor2d([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0], [6, 1]);
const ys = tf.tensor2d([-3.0, -1.0, 2.0, 3.0, 5.0, 7.0], [6, 1]);
```

Notice that we're defining this as a `tensor2d`, since we don't have something like `numpy` from Python. As its name suggests, we must specify the extents of the two dimensions via the second argument.

1.1.4 Training the Model

Training should by *asynchronous*, because it takes an indeterminate amount of time to complete. Our next piece of code will call an asynchronous function called `doTraining`, which, when it completes execution will do something. Because training can take an indeterminate amount of time, we don't want to block the browser while this is taking place, so instead we specify it as an asynchronous function that calls us back when it's done.

```
doTraining(model).then(() => {
    alert(model.predict(tf.tensor2d([10], [1,1])));
});
```

We call the function by passing it a model we just created above; when it calls back, the model is trained and at that point we can call `model.predict()`. In this example, we're predicted $\hat{y}$ for an input of $x = 10$. Note that we still must use a `tensor2d`, which in this case is just a scalar i.e. a tensor of dimension 1×1. To actually define an asynchronous training function, we can do so as follows:

```
async function doTraining(model){
    const history =
          await model.fit(xs, ys,
                { epochs: 500,
                  callbacks:{
                      onEpochEnd: asynch(epoch, logs) =>{
                          console.log("Epoch:" + epoch + "␣Loss:" + logs.loss);
                      }
                  }
                });
}
```