Neural Networks in JavaScript with deeplearn.js

DECEMBER 05, 2017 BY ROBIN WIERUCH - EDIT THIS POST

A couple of my recent articles gave an introduction into a subfield of artificial intelligence by implementing foundational machine learning algorithms in JavaScript (e.g. linear regression with gradient descent, linear regression with normal equation or logistic regression with gradient descent). These machine learning algorithms were implemented from scratch in JavaScript by using the math.js node package for linear algebra (e.g. matrix operations) and calculus. You can find all of these machine learning algorithms grouped in a GitHub organization. If you find any flaws in them, please help me out to make the organization a great learning resource for others. I intend to grow the amount of repositories showcasing different machine learning algorithms to provide web developers a starting point when they enter the domain of machine learning.

Personally, I found it becomes quite complex and challenging to implement those algorithms from scratch at some point. Especially when combining JavaScript and neural networks with the implementation of forward and back propagation. Since I am learning about neural networks myself

at the moment, I started to look for libraries doing the job for me. Hopefully I am able to catch up with those foundational implementations to publish them in the GitHub organization in the future. However, for now, as I researched about potential candidates to facilitate neural networks in JavaScript, I came across deeplearn.js which was recently released by Google. So I gave it a shot. In this article / tutorial, I want to share my experiences by implementing with you a neural network in JavaScript with deeplearn.js to solve a real world problem for web accessibility.

I highly recommend to take the Machine Learning course by Andrew Ng. This article will not explain the machine learning algorithms in detail, but only demonstrate their usage in JavaScript. The course on the other hand goes into detail and explains these algorithms in an amazing quality. At this point in time of writing the article, I learn about the topic myself and try to internalize my learnings by writing about them and applying them in JavaScript. If you find any parts for improvements, please reach out in the comments or create a Issue/Pull Request on GitHub.

WHAT'S THE PURPOSE OF THE NEURAL NETWORK?

The neural network implemented in this article should be able to improve web accessibility by choosing an appropriate font color regarding a background color. For instance, the font color on a dark blue background should be white whereas the font color on a light yellow background should be black. You might wonder: Why would you need a neural network for the task in the first place? It isn't too difficult to compute an accessible font color depending on a background color programmatically, is it? I quickly found a solution on Stack Overflow for the problem and adjusted it to my needs to facilitate colors in RGB space.

```
function getAccessibleColor(rgb) {
  let [ r, g, b ] = rgb;

  let colors = [r / 255, g / 255, b / 255];

  let c = colors.map((col) => {
    if (col <= 0.03928) {
      return col / 12.92;
    }
    return Math.pow((col + 0.055) / 1.055, 2.4);
  });

  let L = (0.2126 * c[0]) + (0.7152 * c[1]) + (0.0722 * c[2]);

  return (L > 0.179)
    ? [ 0, 0, 0 ]
    : [ 255, 255, 255 ];
}
```

The use case of the neural network isn't too valuable for the real world because there is already a programmatic way to solve the problem. There isn't a need to use a machine trained algorithm for it. However, since there is a programmatic approach to solve the problem, it becomes simple to validate the performance of a neural network which might be able to solve the problem for us too. Checkout the animation in the GitHub repository of a learning neural network to get to know how it will perform eventually and what you are going to build in this tutorial.

If you are familiar with machine learning, you might have noticed that the task at hand is a classification problem. An algorithm should decide a binary output (font color: white or black) based on an input (background color). Over the course of training the algorithm with a neural network, it will eventually output the correct font colors based on background colors as inputs.

The following sections will give you guidance to setup all parts for your neural network from scratch. It is up to you to wire the parts together in your own file/folder setup. But you can consolidate the previous referenced GitHub repository for the implementation details.

DATA SET GENERATION IN JAVASCRIPT

A training set in machine learning consists of input data points and output data points (labels). It is used to train the algorithm which will predict the output for new input data points outside of the training set (e.g. test set). During the training phase, the algorithm trained by the neural network adjusts its weights to predict the given labels of the input data points. In conclusion, the trained algorithm is a function which takes a data point as input and approximates the output label.

After the algorithm is trained with the help of the neural network, it can output font colors for new background colors which weren't in the training set. Therefore you will use a **test set** later on. It is used to verify the accuracy of the trained algorithm. Since we are dealing with colors, it isn't difficult to generate a sample data set of input colors for the neural network.

```
function generateRandomRgbColors(m) {
  const rawInputs = [];

  for (let i = 0; i < m; i++) {
    rawInputs.push(generateRandomRgbColor());
  }

  return rawInputs;
}

function generateRandomRgbColor() {
  return [
    randomIntFromInterval(0, 255),
    randomIntFromInterval(0, 255),
    randomIntFromInterval(0, 255),
```

```
  ];
}

function randomIntFromInterval(min, max) {
  return Math.floor(Math.random() * (max - min + 1) + min);
}
```

The `generateRandomRgbColors()` function creates partial data sets of a given size m. The data points in the data sets are colors in the RGB color space. Each color is represented as a row in a matrix whereas each column is a **feature** of the color. A feature is either the R, G or B encoded value in the RGB space. The data set hasn't any labels yet, so the training set isn't complete (also called unlabeled training set), because it has only input values but no output values.

Since the programmatic approach to generate an accessible font color based on a color is known, an adjusted version of the functionality can be derived to generate the labels for the training set (and the test set later on). The labels are adjusted for a binary classification problem and reflect the colors black and white implicitly in the RGB space. Therefore a label is either [0, 1] for the color black or [1, 0] for the color white.

```
function getAccessibleColor(rgb) {
  let [ r, g, b ] = rgb;

  let color = [r / 255, g / 255, b / 255];

  let c = color.map((col) => {
    if (col <= 0.03928) {
      return col / 12.92;
    }
    return Math.pow((col + 0.055) / 1.055, 2.4);
  });

  let L = (0.2126 * c[0]) + (0.7152 * c[1]) + (0.0722 * c[2]);

  return (L > 0.179)
    ? [ 0, 1 ] // black
    : [ 1, 0 ]; // white
}
```

Now you have everything in place to generate random data sets (training set, test set) of (background) colors which are classified either for black or white (font) colors.

```
function generateColorSet(m) {
  const rawInputs = generateRandomRgbColors(m);
  const rawTargets = rawInputs.map(getAccessibleColor);

  return { rawInputs, rawTargets };
}
```

Another step to give the underlying algorithm in the neural network a better time is feature scaling. In a simplified version of feature scaling, you want to have the values of your RGB channels between 0 and 1. Since you know about the maximum value, you can simply derive the normalized value for each color channel.

```
function normalizeColor(rgb) {
  return rgb.map(v => v / 255);
}
```

It is up to you to put this functionality in your neural network model or as separate utility function. I will put it in the neural network model in the next step.

SETUP PHASE OF A NEURAL NETWORK MODEL IN JAVASCRIPT

Now comes the exciting part where you will implement a neural network in JavaScript. Before you can start implementing it, you should install the deeplearn.js library. It is a framework for neural networks in JavaScript. The official pitch for it says: "*deeplearn.js is an open-source library that brings performant machine learning building blocks to the web, allowing you to train neural networks in a browser or run pre-trained models in inference mode.*" In this article, you will train your model yourself and run it in inference mode afterward. There are two major advantages to use the library:

First, it uses the GPU of your local machine which accelerates the vector computations in machine learning algorithms. These machine learning computations are similar to graphical computations and thus it is computational efficient to use the GPU instead of the CPU.

Second, deeplearn.js is structured similar to the popular Tensorflow library which happens to be also developed by Google but is written in Python. So if you want to make the jump to machine learning in Python, deeplearn.js might give you a great gateway to the whole domain in JavaScript.

Let's get back to your project. If you have set it up with npm, you can simply install deeplearn.js on the command line. Otherwise check the official documentation of the deeplearn.js project for installation instructions.

```
npm install deeplearn
```

Since I didn't build a vast number of neural networks myself yet, I followed the common practice of architecting the neural network in an object-oriented programming style. In JavaScript, you can use a

JavaScript ES6 class to facilitate it. A class gives you the perfect container for your neural network by defining properties and class methods to the specifications of your neural network. For instance, your function to normalize a color could find a spot in the class as method.

```
class ColorAccessibilityModel {

  normalizeColor(rgb) {
    return rgb.map(v => v / 255);
  }

}

export default ColorAccessibilityModel;
```

Perhaps it is a place for your functions to generate the data sets as well. In my case, I only put the normalization in the class as class method and leave the data set generation outside of the class. You could argue that there are different ways to generate a data set in the future and thus it shouldn't be defined in the neural network model itself. Nevertheless, that's only a implementation detail.

The training and inference phase are summarized under the umbrella term **session** in machine learning. You can setup the session for the neural network in your neural network class. First of all, you can import the NDArrayMathGPU class from deeplearn.js which helps you to perform mathematical calculations on the GPU in a computational efficient way.

```
import {
  NDArrayMathGPU,
} from 'deeplearn';

const math = new NDArrayMathGPU();

class ColorAccessibilityModel {
  ...
}

export default ColorAccessibilityModel;
```

Second, declare your class method to setup your session. It takes a training set as argument in its function signature and thus it becomes the perfect consumer for a generated training set from a previous implemented function. In the third step, the session initializes an empty graph. In the next steps, the graph will reflect your architecture of the neural network. It is up to you to define all of its properties.

```
import {
  Graph,
  NDArrayMathGPU,
```

```
} from 'deeplearn';

class ColorAccessibilityModel {

  setupSession(trainingSet) {
    const graph = new Graph();
  }

  ..

}

export default ColorAccessibilityModel;
```

Fourth, you define the shape of your input and output data points for your graph in form of a **tensor**. A tensor is an array (of arrays) of numbers with a variable number of dimensions. It can be a vector, a matrix or a higher dimensional matrix. The neural network has these tensors as input and output. In our case, there are three input units (one input unit per color channel) and two output units (binary classification, e.g. white and black color).

```
class ColorAccessibilityModel {

  inputTensor;
  targetTensor;

  setupSession(trainingSet) {
    const graph = new Graph();

    this.inputTensor = graph.placeholder('input RGB value', [3]);
    this.targetTensor = graph.placeholder('output classifier', [2]);
  }

  ...

}

export default ColorAccessibilityModel;
```

Fifth, a neural network has hidden layers in between. It's the blackbox where the magic happens. Basically, the neural network comes up with its own cross computed paramaters which are trained in the session. After all, it is up to you to define the dimension (layer size with each unit size) of the hidden layer(s).

```
class ColorAccessibilityModel {

  inputTensor;
  targetTensor;

  setupSession(trainingSet) {
```

```
    const graph = new Graph();

    this.inputTensor = graph.placeholder('input RGB value', [3]);
    this.targetTensor = graph.placeholder('output classifier', [2]);

    let connectedLayer = this.createConnectedLayer(graph, this.inputTensor,
    connectedLayer = this.createConnectedLayer(graph, connectedLayer, 1, 32)
    connectedLayer = this.createConnectedLayer(graph, connectedLayer, 2, 16)
  }

  createConnectedLayer(
    graph,
    inputLayer,
    layerIndex,
    units,
  ) {
    ...
  }

  ...

}

export default ColorAccessibilityModel;
```

Depending on your number of layers, you are altering the graph to span more and more of its layers. The class method which creates the connected layer takes the graph, the mutated connected layer, the index of the new layer and number of units. The layer property of the graph can be used to return a new tensor that is identified by a name.

```
class ColorAccessibilityModel {

  inputTensor;
  targetTensor;

  setupSession(trainingSet) {
    const graph = new Graph();

    this.inputTensor = graph.placeholder('input RGB value', [3]);
    this.targetTensor = graph.placeholder('output classifier', [2]);

    let connectedLayer = this.createConnectedLayer(graph, this.inputTensor,
    connectedLayer = this.createConnectedLayer(graph, connectedLayer, 1, 32)
    connectedLayer = this.createConnectedLayer(graph, connectedLayer, 2, 16)
  }

  createConnectedLayer(
    graph,
    inputLayer,
    layerIndex,
    units,
  ) {
```

```
    return graph.layers.dense(
      `fully_connected_${layerIndex}`,
      inputLayer,
      units
    );
  }

  ...

}

export default ColorAccessibilityModel;
```

Each neuron in a neural network has to have a defined **activation function**. It can be a **logistic activation function** that you might know already from logistic regression and thus it becomes a **logistic unit** in the neural network. In our case, the neural network uses **rectified linear units** as default.

```
class ColorAccessibilityModel {

  inputTensor;
  targetTensor;

  setupSession(trainingSet) {
    const graph = new Graph();

    this.inputTensor = graph.placeholder('input RGB value', [3]);
    this.targetTensor = graph.placeholder('output classifier', [2]);

    let connectedLayer = this.createConnectedLayer(graph, this.inputTensor,
    connectedLayer = this.createConnectedLayer(graph, connectedLayer, 1, 32)
    connectedLayer = this.createConnectedLayer(graph, connectedLayer, 2, 16)
  }

  createConnectedLayer(
    graph,
    inputLayer,
    layerIndex,
    units,
    activationFunction
  ) {
    return graph.layers.dense(
      `fully_connected_${layerIndex}`,
      inputLayer,
      units,
      activationFunction ? activationFunction : (x) => graph.relu(x)
    );
  }

  ...

}
```

```
export default ColorAccessibilityModel;
```

Sixth, create the layer which outputs the binary classification. It has 2 output units; one for each discrete value (black, white).

```
class ColorAccessibilityModel {

  inputTensor;
  targetTensor;
  predictionTensor;

  setupSession(trainingSet) {
    const graph = new Graph();

    this.inputTensor = graph.placeholder('input RGB value', [3]);
    this.targetTensor = graph.placeholder('output classifier', [2]);

    let connectedLayer = this.createConnectedLayer(graph, this.inputTensor,
    connectedLayer = this.createConnectedLayer(graph, connectedLayer, 1, 32)
    connectedLayer = this.createConnectedLayer(graph, connectedLayer, 2, 16)

    this.predictionTensor = this.createConnectedLayer(graph, connectedLayer,
  }

  ...

}

export default ColorAccessibilityModel;
```

Seventh, declare a cost tensor which defines the loss function. In this case, it will be a mean squared error. It optimizes the algorithm that takes the target tensor (labels) of the training set and the predicted tensor from the trained algorithm to evaluate the cost.

```
class ColorAccessibilityModel {

  inputTensor;
  targetTensor;
  predictionTensor;
  costTensor;

  setupSession(trainingSet) {
    const graph = new Graph();

    this.inputTensor = graph.placeholder('input RGB value', [3]);
    this.targetTensor = graph.placeholder('output classifier', [2]);
```

```
    let connectedLayer = this.createConnectedLayer(graph, this.inputTensor,
    connectedLayer = this.createConnectedLayer(graph, connectedLayer, 1, 32)
    connectedLayer = this.createConnectedLayer(graph, connectedLayer, 2, 16)

    this.predictionTensor = this.createConnectedLayer(graph, connectedLayer,
    this.costTensor = graph.meanSquaredCost(this.targetTensor, this.predicti
  }

  ...

}

export default ColorAccessibilityModel;
```

Last but not least, setup the session with the architected graph. Afterward, you can start to prepare the incoming training set for the upcoming training phase.

```
import {
  Graph,
  Session,
  NDArrayMathGPU,
} from 'deeplearn';

class ColorAccessibilityModel {

  session;

  inputTensor;
  targetTensor;
  predictionTensor;
  costTensor;

  setupSession(trainingSet) {
    const graph = new Graph();

    this.inputTensor = graph.placeholder('input RGB value', [3]);
    this.targetTensor = graph.placeholder('output classifier', [2]);

    let connectedLayer = this.createConnectedLayer(graph, this.inputTensor,
    connectedLayer = this.createConnectedLayer(graph, connectedLayer, 1, 32)
    connectedLayer = this.createConnectedLayer(graph, connectedLayer, 2, 16)

    this.predictionTensor = this.createConnectedLayer(graph, connectedLayer,
    this.costTensor = graph.meanSquaredCost(this.targetTensor, this.predicti

    this.session = new Session(graph, math);

    this.prepareTrainingSet(trainingSet);
  }

  prepareTrainingSet(trainingSet) {
    ...
```

```
  }

  ...

}

export default ColorAccessibilityModel;
```

The setup isn't done before preparing the training set for the neural network. First, you can support the computation by using a callback function in the GPU performed math context. But it's not mandatory and you could perform the computation without it.

```
import {
  Graph,
  Session,
  NDArrayMathGPU,
} from 'deeplearn';

const math = new NDArrayMathGPU();

class ColorAccessibilityModel {

  session;

  inputTensor;
  targetTensor;
  predictionTensor;
  costTensor;

  ...

  prepareTrainingSet(trainingSet) {
    math.scope(() => {
      ...
    });
  }

  ...

}

export default ColorAccessibilityModel;
```

Second, you can destructure the input and output (labels, also called targets) from the training set to map them into a readable format for the neural network. The mathematical computations in deeplearn.js use their in-house NDArrays. After all, you can imagine them as simple array in array matrices or vectors. In addition, the colors from the input array are normalized to improve the performance of the neural network.

```
import {
  Array1D,
  Graph,
  Session,
  NDArrayMathGPU,
} from 'deeplearn';

const math = new NDArrayMathGPU();

class ColorAccessibilityModel {

  session;

  inputTensor;
  targetTensor;
  predictionTensor;
  costTensor;

  ...

  prepareTrainingSet(trainingSet) {
    math.scope(() => {
      const { rawInputs, rawTargets } = trainingSet;

      const inputArray = rawInputs.map(v => Array1D.new(this.normalizeColor(
      const targetArray = rawTargets.map(v => Array1D.new(v));
    });
  }

  ...

}

export default ColorAccessibilityModel;
```

Third, the input and target arrays are shuffled. The shuffler provided by deeplearn.js keeps both arrays in sync when shuffling them. The shuffle happens for each training iteration to feed different inputs as batches to the neural network. The whole shuffling process improves the trained algorithm, because it is more likely to make generalizations by avoiding over-fitting.

```
import {
  Array1D,
  InCPUMemoryShuffledInputProviderBuilder,
  Graph,
  Session,
  NDArrayMathGPU,
} from 'deeplearn';

const math = new NDArrayMathGPU();

class ColorAccessibilityModel {
```

```
  session;

  inputTensor;
  targetTensor;
  predictionTensor;
  costTensor;

  ...

  prepareTrainingSet(trainingSet) {
    math.scope(() => {
      const { rawInputs, rawTargets } = trainingSet;

      const inputArray = rawInputs.map(v => Array1D.new(this.normalizeColor(
      const targetArray = rawTargets.map(v => Array1D.new(v));

      const shuffledInputProviderBuilder = new InCPUMemoryShuffledInputProvi
        inputArray,
        targetArray
      ]);

      const [
        inputProvider,
        targetProvider,
      ] = shuffledInputProviderBuilder.getInputProviders();
    });
  }

  ...

}

export default ColorAccessibilityModel;
```

Last but not least, the feed entries are the ultimate input for the feedforward algorithm of the neural network in the training phase. It matches data and tensors (which were defined by their shapes in the setup phase).

```
import {
  Array1D,
  InCPUMemoryShuffledInputProviderBuilder
  Graph,
  Session,
  NDArrayMathGPU,
} from 'deeplearn';

const math = new NDArrayMathGPU();

class ColorAccessibilityModel {

  session;
```

```
  inputTensor;
  targetTensor;
  predictionTensor;
  costTensor;

  feedEntries;

  ...

  prepareTrainingSet(trainingSet) {
    math.scope(() => {
      const { rawInputs, rawTargets } = trainingSet;

      const inputArray = rawInputs.map(v => Array1D.new(this.normalizeColor(
      const targetArray = rawTargets.map(v => Array1D.new(v));

      const shuffledInputProviderBuilder = new InCPUMemoryShuffledInputProvi
        inputArray,
        targetArray
      ]);

      const [
        inputProvider,
        targetProvider,
      ] = shuffledInputProviderBuilder.getInputProviders();

      this.feedEntries = [
        { tensor: this.inputTensor, data: inputProvider },
        { tensor: this.targetTensor, data: targetProvider },
      ];
    });
  }

  ...

}

export default ColorAccessibilityModel;
```

The setup phase of the neural network is finished. The neural network is implemented with all its layers and units. Moreover the training set is prepared for training. Only two **hyperparameters** are missing to configure the high level behaviour of the neural network. These are used in the next phase: the training phase.

```
import {
  Array1D,
  InCPUMemoryShuffledInputProviderBuilder,
  Graph,
  Session,
  SGDOptimizer,
  NDArrayMathGPU,
```

```
} from 'deeplearn';

const math = new NDArrayMathGPU();

class ColorAccessibilityModel {

  session;

  optimizer;

  batchSize = 300;
  initialLearningRate = 0.06;

  inputTensor;
  targetTensor;
  predictionTensor;
  costTensor;

  feedEntries;

  constructor() {
    this.optimizer = new SGDOptimizer(this.initialLearningRate);
  }

  ...

}

export default ColorAccessibilityModel;
```

The first parameter is the **learning rate**. You might remember it from linear or logistic regression with gradient descent. It determines how fast the algorithm converges to minimize the cost. So one could assume it should be high. But it mustn't be too high. Otherwise gradient descent never converges because it cannot find a local optima.

The second parameter is the **batch size**. It defines how many data points of the training set are passed through the neural network in one **epoch** (iteration). An epoch includes one forward pass and one backward pass of one batch of data points. There are two advantages to training a neural network with batches. First, it is not as computational intensive because the algorithm is trained with less data points in memory. Second, a neural network trains faster with batches because the weights are adjusted with each batch of data points in an epoch rather than the whole training set going through it.

TRAINING PHASE

The setup phase is finished. Next comes the training phases. It doesn't need too much implementation anymore, because all the cornerstones were defined in the setup phase. First of all, the **training**

phase can be defined in a class method. It is executed again in the math context of deeplearn.js. In addition, it uses all the predefined properties of the neural network instance to train the algorithm.

```
class ColorAccessibilityModel {

  ...

  train() {
    math.scope(() => {
      this.session.train(
        this.costTensor,
        this.feedEntries,
        this.batchSize,
        this.optimizer
      );
    });
  }
}

export default ColorAccessibilityModel;
```

The train method is only one epoch of the neural network training. So when it is called from outside, it has to be called iteratively. Moreover it trains only one batch. In order to train the algorithm for multiple batches, you have to run multiple iterations of the train method again.

That's it for a basic training phase. But it can be improved by adjusting the learning rate over time. The learning rate can be high in the beginning, but when the algorithm converges with each step it takes, the learning rate could be decreased.

```
class ColorAccessibilityModel {

  ...

  train(step) {
    let learningRate = this.initialLearningRate * Math.pow(0.90, Math.floor(
    this.optimizer.setLearningRate(learningRate);

    math.scope(() => {
      this.session.train(
        this.costTensor,
        this.feedEntries,
        this.batchSize,
        this.optimizer
      );
    }
  }
}

export default ColorAccessibilityModel;
```

In our case, the learning rate decreases by 10% every 50 steps. Next, it would be interesting to get the cost in the training phase to verify that it decreases over time. It could be simply returned with each iteration, but that's leads to computational inefficiency. Every time the cost is requested from the neural network, it has to access the GPU to return it. Therefore, we only access the cost once in a while to verify that it's decreasing. If the cost is not requested, the cost reduction constant for the training is defined with NONE (which was the default before).

```
import {
  Array1D,
  InCPUMemoryShuffledInputProviderBuilder,
  Graph,
  Session,
  SGDOptimizer,
  NDArrayMathGPU,
  CostReduction,
} from 'deeplearn';

class ColorAccessibilityModel {

  ...

  train(step, computeCost) {
    let learningRate = this.initialLearningRate * Math.pow(0.90, Math.floor(
    this.optimizer.setLearningRate(learningRate);

    let costValue;
    math.scope(() => {
      const cost = this.session.train(
        this.costTensor,
        this.feedEntries,
        this.batchSize,
        this.optimizer,
        computeCost ? CostReduction.MEAN : CostReduction.NONE,
      );

      if (computeCost) {
        costValue = cost.get();
      }
    });

    return costValue;
  }
}

export default ColorAccessibilityModel;
```

Finally, that's it for the training phase. Now it needs only to be executed iteratively from the outside after the session setup with the training set. The outside execution can decide on a condition if the

train method should return the cost.

INFERENCE PHASE

The final stage is the **inference phase** where a test set is used to validate the performance of the trained algorithm. The input is a color in RGB space for the background color and as output it should predict the classifier [0, 1] or [1, 0] for either black or white for the font color. Since the input data points were normalized, don't forget to normalize the color in this step as well.

```
class ColorAccessibilityModel {

  ...

  predict(rgb) {
    let classifier = [];

    math.scope(() => {
      const mapping = [{
        tensor: this.inputTensor,
        data: Array1D.new(this.normalizeColor(rgb)),
      }];

      classifier = this.session.eval(this.predictionTensor, mapping).getValu
    });

    return [ ...classifier ];
  }
}

export default ColorAccessibilityModel;
```

The method run the performance critical parts in the math context again. There it needs to define a mapping that will end up as input for the session evaluation. Keep in mind, that the predict method doesn't need to run strictly after the training phase. It can be used during the training phase to output validations of the test set.

Ultimately the neural network is implemented for setup, training and inference phase.

VISUALIZE A LEARNING NEURAL NETWORK IN JAVASCRIPT

Now it's about time using the neural network to train it with a training set in the training phase and validate the predictions in the inference phase with a test set. In its simplest form, you would set up the neural network, run the training phase with a training set, validate over the time of training the minimizing cost and finally predict a couple of data points with a test set. All of it would happen on the developer console in the web browser with a couple of console.log statements. However, since the neural network is about color prediction and deeplearn.js runs in the browser anyway, it would be much more enjoyable to visualize the training phase and inference phase of the neural network.

At this point, you can decide on your own how to visualize the phases of your performing neural network. It can be plain JavaScript by using a canvas and the requestAnimationFrame API. But in the case of this article, I will demonstrate it by using React.js, because I write about it on my blog as well.

So after setting up the project with create-react-app, the App component will be our entry point for the visualization. First of all, import the neural network class and the functions to generate the data sets from your files. Moreover, add a couple of constants for the training set size, test set sizes and number of training iterations.

```
import React, { Component } from 'react';

import './App.css';

import generateColorSet from './data';
import ColorAccessibilityModel from './neuralNetwork';

const ITERATIONS = 750;
const TRAINING_SET_SIZE = 1500;
const TEST_SET_SIZE = 10;

class App extends Component {
  ...
}

export default App;
```

In the constructor of the App component, generate the data sets (training set, test set), setup the neural network session by passing in the training set, and define the initial local state of the component. Over the course of the training phase, the value for the cost and number of iterations will be displayed somewhere, so these are the properties which end up in the component state.

```
import React, { Component } from 'react';

import './App.css';

import generateColorSet from './data';
import ColorAccessibilityModel from './neuralNetwork';
```

```
const ITERATIONS = 750;
const TRAINING_SET_SIZE = 1500;
const TEST_SET_SIZE = 10;

class App extends Component {

  testSet;
  trainingSet;
  colorAccessibilityModel;

  constructor() {
    super();

    this.testSet = generateColorSet(TEST_SET_SIZE);
    this.trainingSet = generateColorSet(TRAINING_SET_SIZE);

    this.colorAccessibilityModel = new ColorAccessibilityModel();
    this.colorAccessibilityModel.setupSession(this.trainingSet);

    this.state = {
      currentIteration: 0,
      cost: -42,
    };
  }

  ...
}

export default App;
```

Next, after setting up the session of the neural network in the constructor, you could train the neural network iteratively. In a naive approach you would only need a for loop in a mounting component lifecycle hook of React.

```
class App extends Component {

  ...

  componentDidMount () {
    for (let i = 0; i <= ITERATIONS; i++) {
      this.colorAccessibilityModel.train(i);
    }
  };
}

export default App;
```

However, it wouldn't work to render an output during the training phase in React, because the component couldn't re-render while the neural network blocks the single JavaScript thread. That's where requestAnimationFrame can be used in React. Rather than defining a for loop statement

ourselves, each requested animation frame of the browser can be used to run exactly one training iteration.

```
class App extends Component {

  ...

  componentDidMount () {
    requestAnimationFrame(this.tick);
  };

  tick = () => {
    this.setState((state) => ({
      currentIteration: state.currentIteration + 1
    }));

    if (this.state.currentIteration < ITERATIONS) {
      requestAnimationFrame(this.tick);

      this.colorAccessibilityModel.train(this.state.currentIteration);
    }
  };
}

export default App;
```

In addition, the cost can be computed every 5th step. As mentioned, the GPU needs to be accessed to retrieve the cost. Thus it should be avoided to train the neural network faster.

```
class App extends Component {

  ...

  componentDidMount () {
    requestAnimationFrame(this.tick);
  };

  tick = () => {
    this.setState((state) => ({
      currentIteration: state.currentIteration + 1
    }));

    if (this.state.currentIteration < ITERATIONS) {
      requestAnimationFrame(this.tick);

      let computeCost = !(this.state.currentIteration % 5);
      let cost = this.colorAccessibilityModel.train(
        this.state.currentIteration,
        computeCost
      );

      if (cost > 0) {
```

```
        this.setState(() => ({ cost }));
      }
    }
  };
}

export default App;
```

The training phase is running once the component mounted. Now it is about rendering the test set with the programmatically computed output and the predicted output. Over time, the predicted output should be the same as the programmatically computed output. The training set itself is never visualized.

```
class App extends Component {

  ...

  render() {
    const { currentIteration, cost } = this.state;

    return (
      <div className="app">
        <div>
          <h1>Neural Network for Font Color Accessibility</h1>
          <p>Iterations: {currentIteration}</p>
          <p>Cost: {cost}</p>
        </div>

        <div className="content">
          <div className="content-item">
            <ActualTable
              testSet={this.testSet}
            />
          </div>

          <div className="content-item">
            <InferenceTable
              model={this.colorAccessibilityModel}
              testSet={this.testSet}
            />
          </div>
        </div>
      </div>
    );
  }
}

const ActualTable = ({ testSet }) =>
  <div>
    <p>Programmatically Computed</p>
  </div>

const InferenceTable = ({ testSet, model }) =>
```

```
  <div>
    <p>Neural Network Computed</p>
  </div>

export default App;
```

The actual table iterates over the size of the test set size to display each color. The test set has the input colors (background colors) and output colors (font colors). Since the output colors are classified into black [0, 1] and white [1, 0] vectors when a data set is generated, they need to be transformed into actual colors again.

```
const ActualTable = ({ testSet }) =>
  <div>
    <p>Programmatically Computed</p>

    {Array(TEST_SET_SIZE).fill(0).map((v, i) =>
      <ColorBox
        key={i}
        rgbInput={testSet.rawInputs[i]}
        rgbTarget={fromClassifierToRgb(testSet.rawTargets[i])}
      />
    )}
  </div>

const fromClassifierToRgb = (classifier) =>
  classifier[0] > classifier[1]
    ? [ 255, 255, 255 ]
    : [ 0, 0, 0 ]
```

The ColorBox component is a generic component which takes the input color (background color) and target color (font color). It simply displays a rectangle with the input color style, the RGB code of the input color as string and styles the font of the RGB code into the given target color.

```
const ColorBox = ({ rgbInput, rgbTarget }) =>
  <div className="color-box" style={{ backgroundColor: getRgbStyle(rgbInput)
    <span style={{ color: getRgbStyle(rgbTarget) }}>
      <RgbString rgb={rgbInput} />
    </span>
  </div>

const RgbString = ({ rgb }) =>
  `rgb(${rgb.toString()})`

const getRgbStyle = (rgb) =>
  `rgb(${rgb[0]}, ${rgb[1]}, ${rgb[2]})`
```

Last but not least, the exciting part of visualizing the predicted colors in the inference table. It uses the color box as well, but gives a different set of props into it.

```
const InferenceTable = ({ testSet, model }) =>
  <div>
    <p>Neural Network Computed</p>
    {Array(TEST_SET_SIZE).fill(0).map((v, i) =>
      <ColorBox
        key={i}
        rgbInput={testSet.rawInputs[i]}
        rgbTarget={fromClassifierToRgb(model.predict(testSet.rawInputs[i]))}
      />
    )}
  </div>
```

The input color is still the color defined in the test set. But the target color isn't the target color from the test set. The crucial part is that the target color is predicted in this component by using the neural network's predict method. It takes the input color and should predict the target color over the course of the training phase.

Finally, when you start your application, you should see the neural network in action. Whereas the actual table uses the fixed test set from the beginning, the inference table should change its font colors during the training phase. In fact, while the ActualTable component shows the actual test set, the InferenceTable shows the input data points of the test set, but the predicted output by using the neural network. The React rendered part can be seen in the GitHub repository animation too.

. . .

The article has shown you how deeplearn.js can be used to build neural networks in JavaScript for machine learning. If you have any recommendation for improvements, please leave a comment below. In addition, I am curious whether you are interested in the crossover of machine learning and JavaScript. If that's is the case, I would write more about it.

Furthermore, I would love to get more into the topic and I am open for opportunities in the field of machine learning. At the moment, I apply my learnings in JavaScript, but I am so keen to get into Python at some point as well. So if you know about any opportunities in the field, please reach out to me :-)

IMPROVING GRADIENT DESCENT IN JAVASCRIPT

A recent article gave an introduction to the field of machine learning in JavaScript by predicting housing prices with gradient descent in a univariate regression problem. There are several…

POLYNOMIAL REGRESSION AND MODEL SELECTION

After learning about gradient descent in a linear regression , I was curious about using different kinds of hypothesis functions to improve the result of the algorithm itself. So far, the hypothesis…

Get it on Amazon.

TAKE PART

NEVER MISS AN ARTICLE ABOUT WEB DEVELOPMENT AND JAVASCRIPT.

- ✔ Join 50.000+ Developers
- ✔ Learn Web Development
- ✔ Learn JavaScript
- ✔ Access Tutorials, eBooks and Courses
- ✔ Personal Development as a Software Engineer

SUBSCRIBE ›

View our Privacy Policy.