Explaining neural network concepts through an interactive visualization

Using a neural network for complementary color prediction

ROBERTO STELLING PPGI/UFRJ
ADRIANA VIVACQUA PPGI/UFRJ

Abstract

In this article we explore interactive data visualization as a mean to help users understand a training method of a Neural Network. The visualization let the users interact with the neural network training procedure, changing parameters and viewing results as they are shown on the browser. As most machine learning and neural network methods are opaque and hard to understand by the public in general, we explore this visualization as a way to teach the techniques and concepts around neural network model training, visually exposing the behaviour of the training algorithm to the users playing with the visualization. In this article we describe one particular exploration, using the problem of predicting the complementary color of a RGB color, implemented as a neural network model.

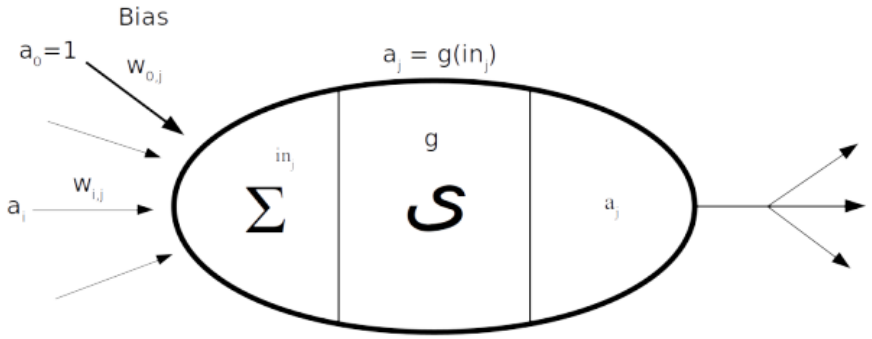
Motivation

- Machine Learning, Artificial Intelligence and Neural Networks techniques are being used in a broad spectrum of applications.
- Some techniques are hard to understand, even for advanced users [1].
- This has led to a push for novel forms of interaction that make it easier for a user, at different roles, to understand how solutions are reached.
- Visualization techniques have been successfully used to help understand large databases and complex processes.
- We are investigating the possibilities of using interactive visualizations to explain machine learning techniques, providing a better understanding of how they function.
- In this paper, we present an interactive visualization to show how neural networks work. We selected a simple and scalable problem, calculating complementary colors, and developed an interactive visualization that allows a user to explore the effects of changing some parameters and view the training process as it happens

We believe that the best way to understand neural network concepts is by interacting with real-life neural network implementations. In this paper, we present an interactive web application, where it is possible to train a neural network to predict a complementary color in RGB, change its parameters, see the network converge, start over and run the model over and over again, adjusting parameters at will. Hopefully the user will eventually develop insights about neural network concepts after some time playing with the application. By the way, the parameters of the demo are a bit off to start with, so it is up to the user to play and improve them. There is an explanation of the interface *here*, and the application is live at the end of the article.

Neural Networks

Neural network concepts are neither easily understood, nor easily explained. Most explanations start with the McCulloch/Pitts [2] neuron, build the idea of layers and weights, define backpropagation [3] and may, or may not, end the discussion with learning rates, batch size, epochs, activation functions and other intricacies of neural network implementations.



But when an student or researcher has to implement a neural network, he or she will have to manipulate the very last layer of abstraction, as almost every library deliver layers, weights and backpropagation as well wrapped black boxes. It is during the very first neural network implementation that the beginner will have a better grasp of learning rates, batch sizes, activation units, network architecture, epochs and other concepts.

Keeping this student or researcher in mind, we developed a web based application, on top of tensorflow.js [4], a javascript inception of TensorFlow [5] library and a spinoff of deeplearn.js (now Tensorflow.js core [6]). In this application the user can play with some of the model parameters (more options and functionality will appear in later versions) of an application and see the model training and predictions on a web browser, as it happens.

Visualizations and Explanations

Visualizations enable users to better understand and explore large sets of data or processes.

Different types of users require different types of explanations and learning strategies. We divide users into three categories, according to how knowledgeable they are:

1. Layperson: has no knowledge of the process or algorithms being explained. This covers most of the general public;
2. Knowledgeable: knows the concepts and algorithms, is familiar with the techniques involved;
3. Designer: this is a person involved in the design of the solution. Although a knowledgeable individual, the designer might also need to better understand the system, in order to fine tune parameters.

This case was developed with knowledgeable in mind but a layperson can also explore it, as it assumes little prior knowledge of Neural Networks.

The model

To demonstrate a real neural network in action we selected the problem of predicting a complementary color [7] [8] from the RGB [9] color space. The idea was adapted from a deeplearn.js (now Tensorflow.js core [6]) tutorial, called Complementary Color Prediction, developed by Chi Zeng, from Google. The original tutorial was developed in TypeScript [10] but as the library was eventually deprecated, the code, apparently, is not available anymore. Initially we've built a D3.js [11] visualization for the neural network output and corrected an small issue with activation units. Later, when deeplearn.js moved to tensorflow.js, we rewrote the whole application in JavaScript with tensorflow.js and built the demo described in this article.

Why complementary colors?

The main reasons for choosing a model to predict complementary colors are:

- The predictions can be shown visually, without much numerical information (although we also show the RGB codes of the colors).
- There is no direct formula to produce the complementary of a color in RGB, but there is an algorithm to generate the complementary of a RGB color. As a bonus, we show that a properly constructed neural network can learn the calculations of a complex algorithm.
- The problem has a good balance between complexity and scalability to fit in a browser based solution.

Implementation

The model was implemented in JavaScript with an architecture of 3 hidden layers. One batch of data/label pairs is generated randomically for each training round, as shown in the code below.

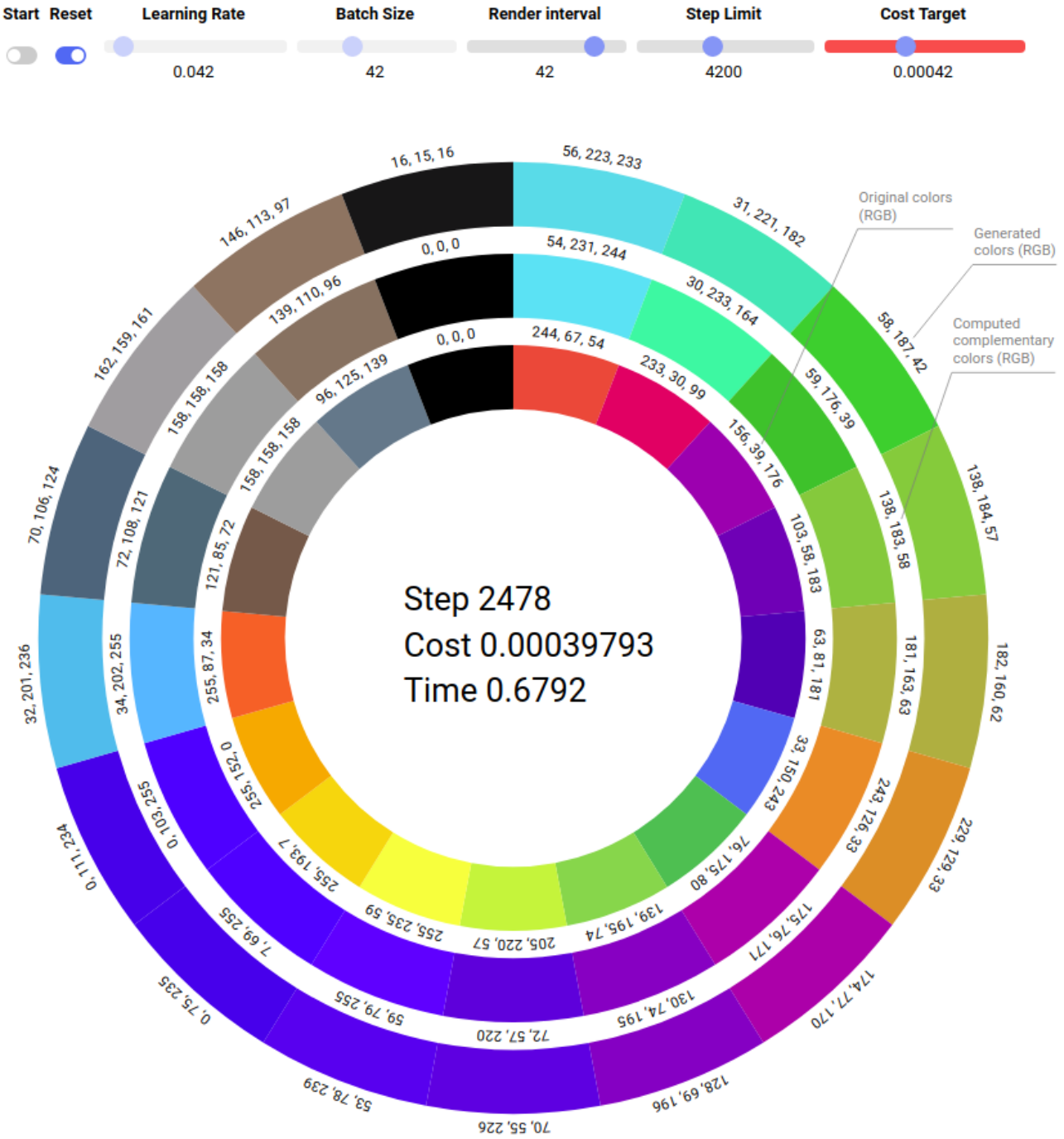
```
async function train1Batch() {
  // Reduce the learning rate by 85% every 42 steps
  //currentLearningRate = initialLearningRate * Math.pow(0.85, Math.floor(step/42));
  //model.optimizer.learningRate = currentLearningRate;
  const batchData = generateData(batchSize);
  const dataTensor = color2tensor(batchData[0]);
  const labelTensor = color2tensor(batchData[1]);
  const history = await model.fit(dataTensor, labelTensor,
           {batchSize: batchSize,
            epochs: epochs
           });

  cost = history.history.loss[0];
  tf.dispose(dataTensor, labelTensor);
  return step++;
}
```

The labels are generated using an implementation of a complementary color algorithm based on an answer by Edd [12] in Stack Overflow. The full code is rather long and is available at our application repository in github [13]. The computeComplementaryColor function converts an RGB color input to HSL [14], compute the complementary color in this color space and converts the result back to RGB.

At each round the model runs for 10 epochs (epoch is another parameter that can be easily exposed to the application interface, but we decided to omit it in the first version). After a user defined number of training rounds, the model is called to predict a set of test colors and the results can be visually compared, on the fly, with the reference RGB labels generated by Edd's algorithm.

The output of each training round is displayed on a color wheel using D3.js [11], as shown in the figure below.



Neural network architecture

The neural network has an input layer for the RGB colors, 3 hidden fully connected layers with 64, 32 and 16 relu units, respectively, and the output layer, for the model predictions. The code below shows how the neural network was created.

```
function modelInit() {
  //Add input layer
  // First layer must have an input shape defined.
  model.add(tf.layers.dense({units: 3,
                             activation: 'tanh',
                             inputShape: [3]}));
  // Afterwards, TF.js does automatic shape inference.
  model.add(tf.layers.dense({units: 64,
                             activation: 'relu'
                            }));
  // Afterwards, TF.js does automatic shape inference.
  model.add(tf.layers.dense({units: 32,
                             activation: 'relu'
                            }));
  // Afterwards, TF.js does automatic shape inference.
  model.add(tf.layers.dense({units: 16,
                             activation: 'relu'
                            }));
  // Afterwards, TF.js does automatic shape inference.
  model.add(tf.layers.dense({units: 3,
                             activation: 'tanh'
                            }));
}
```

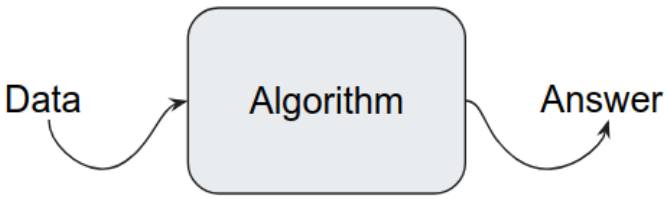
Optimizer, metric and loss

We used a momentum [15] optimizer with mean squared error loss and accuracy metric. The loss function was implemented manually, as seen on the code below, instead of using the library provided Mean Squared Error loss function. This was a design choice that allowed us to change/tweak the loss function at will during development time. On a future version we plan to provide the user with a selection of loss functions, as a configuration option.

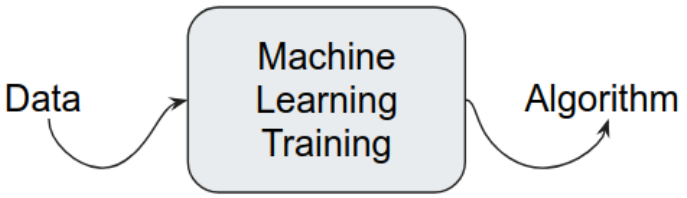
```
function loss(predictions, labels) {
  // Subtract our labels (actual values) from predictions, square the results,
  // and take the mean. Inputs are tensors.
  return tf.tidy(() => {
    const meanSquareError = predictions.sub(labels).square().mean();
    return meanSquareError;
  });
}
```

Common Misconception

We found out that a common difficulty in understanding machine learning applications arise from a simple misconception about the very framework of a machine learning application. Most people look at the field through the lenses of a traditional application, where data is fed to an algorithm that produces an answer.



To understand machine learning applications, one must first understand how they come to life: a LARGE amount of data is fed to a training algorithm and it produces, as output, an instance of a prediction algorithm trained with the input data that was provided to the machine learning algorithm. This output can then take data as input and produce predictions in accordance with the training process. There is NO guarantees that the predictions will be fully correct but they will, hopefully, be as good as possible.



An analogy for the layperson

Suppose that you are given a card with a color at random and you must give back a card with a color too, it can be the same color or a different one. In the beginning, as you have no training, the best you can do is to give back a card at random. Being a deterministic algorithm, everytime someone gives you a card with a color X, you will produce as a response, the same Y color. So if I give you red and you give me back a green card, next time I give you a red card you will give me back a green card again. At this juncture you will produce bogus answers, but answers you will produce!

Now suppose that you are given set of n card pairs, with the correct input/output cards. You can then use these pairs to correct your card selections. But there is a catch, whenever you correct the predictions for one card (for example, blue is the correct answer to red), all the other predictions are affected in a way, they change a little bit. What you try to do then, is to minimize the error of the color pairs you got as input.

The idea is that, after this training and adjusting procedure, you may be able to make good guesses on color pairs you saw, and, hopefully, you will be able to make good guesses on the colors you never saw!

Now, you can adjust the color predictions in small baby steps, changing the output colors just a little bit (remember that when you change the answer for one color, all the other answers are slightly affected), or you can adjust the colors in big steps, with a big effect on your other predictions. Or you can do something in between these two extremes.

Every time you get a color pair list, you adjust the color predictions again. When will you stop this training process? Maybe you will stop after training for 1.000 rounds, or when your data has finished, or you will stop when the error in all the colors you saw is smaller than a certain value (meaning that you're doing good predictions on this data). It is up to you.

If you are still following us, then you, likely, understood what were you doing with the color cards. We can now throw a bit of nomenclature:

- **Learning Rate**: Is the amount of change you make on each pass. Big changes correspond to high learning rate, small changes correspond to small learning rate. If the learning rate is too big, you may never be able to make good predictions. If the learning rate is too small, it may take very long to properly correct, and later predict, the color pairs.

- **Batch Size**: Is the amount of color pairs you get at each turn. The bigger the batch size, the better your adjustments, but you will take longer to process them. Notice, though, that you can process the colors in parallel, so a small batch size can be a bad thing.

- **Step Limit**: Is the maximum amount of rounds of training that you will go through.

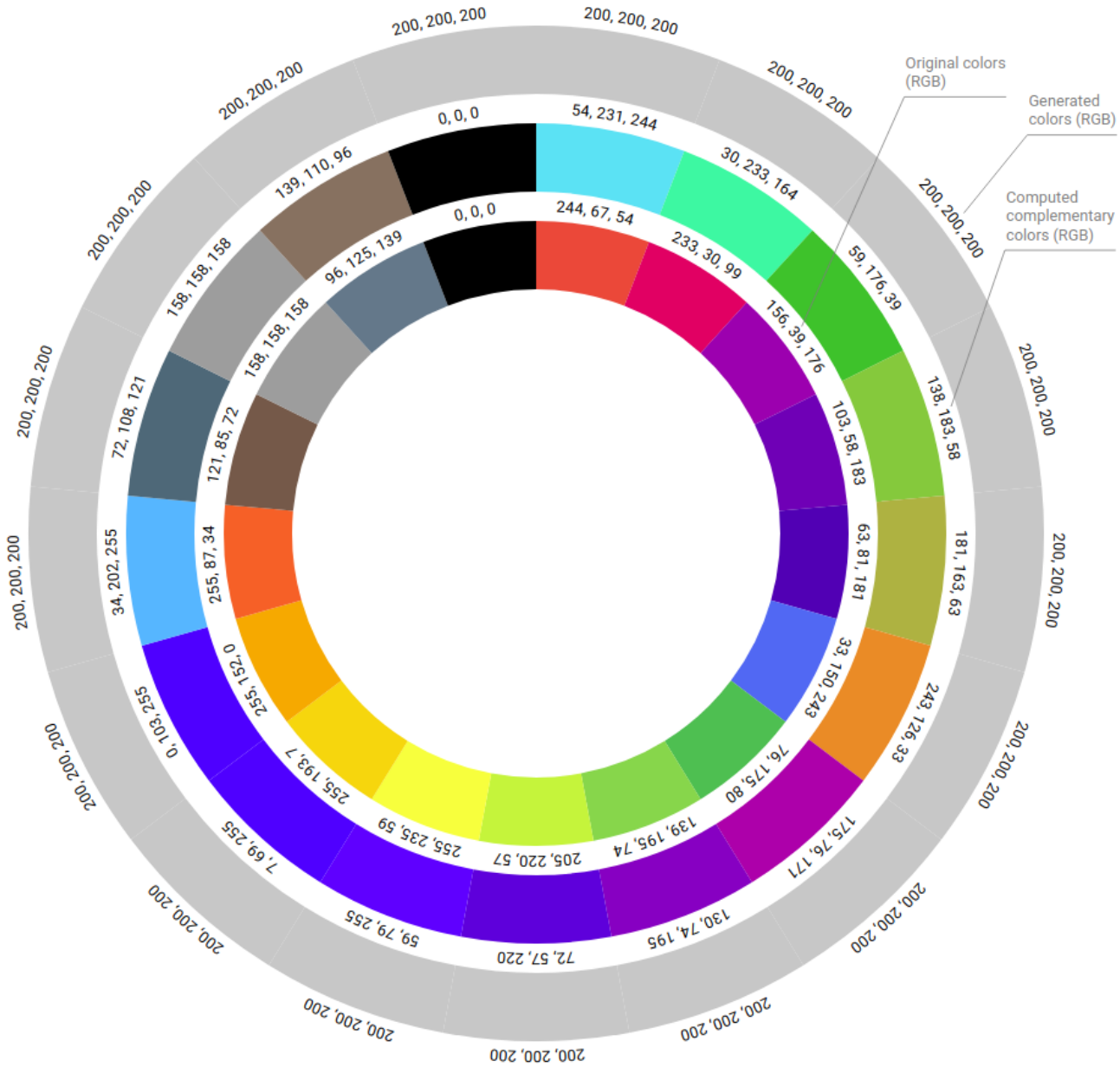
- **Cost Target**: The the minimum error that your overall card predictions must reach in order to stop the training.

There are another concepts in neural network training, but the four above are the ones you will see at work in the interactive application.

The demo

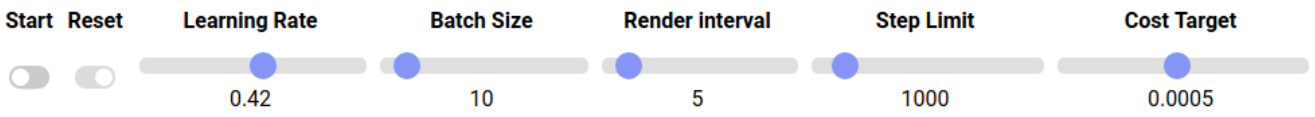
The application we present is based on one of the tutorials for the late deeplearn.js library (now tensorflow.js, a javascript inception of TensorFlow[5] and a spinoff of deeplearn.js). That tutorial, called Complementary Color Prediction, was developed by Chi Zeng, from Google.

We developed a D3.js [11] visual interface for the CCP tutorial in typescript and later, when deeplearn.js moved to tensorflow.js, we rewrote the whole code in javascript and added a bit of interactivity on top of it.



In the figure above we show the starting stage of the demo. The inner ring of the display is a set of colors that we will use to predict its complementary colors [8] [7]. In the middle ring we show the corresponding complementary colors, for reference and control. In the outer ring we display the model predicted complementary colors, for each color in the original color set (the inner ring). In the beginning the outer ring is gray, as no color was predicted yet. When running, the outer ring will be updated with the model predicted colors, and the user will be able to compare the predictions with the reference complementary colors in the middle ring.

It is possible to play with the demo using the controls below:



The five sliders on the right side let the user adjust the values for:

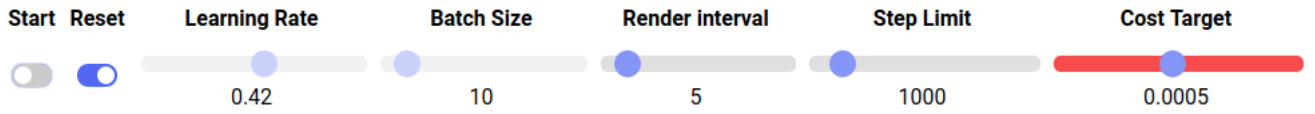
- **Learning Rate**: The learning rate of the training model. Smaller learning rates are the baby steps that may go in the right direction but will take longer to get to the right prediction, larger learning rates are big strides that may go to the right direction but will likely miss the prediction altogether, by going beyond the expected solution.

- **Batch Size**: The amount of colors that will be used for training at each round, ranging from 1 to 128. There are drawbacks in small batches and big batches, it is up to the analyst to figure out the *best batch*.

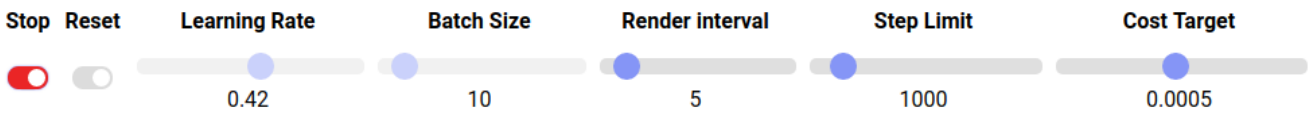
- **Render Interval**: The number of rounds between screen updates, ranging from 1 (updating every round) to 50.

- **Step Limit**: The number of rounds that the training will run, ranging from 25 to 10.000. If the training reaches this limit it will stop and this slider color will switch to red, to show that a step limit was reached. If the step limit is too small, the predictions won't be that good (the outer ring colors won't match the middle ring colors).

- **Cost Target**: The cost target of the predictions, ranging from 0.00005 to 0.001. The training will stop when the cost of the predictions is smaller than the cost target, and the cost target slider will become red in consequence, as shown in the figure below. If the cost is too big, the training may not converge, if the cost is too high, the training may overfit (becoming a very good predictor of the colors it has seen, but off the mark on the colors it hasn't seen).



In the beginning the user can adjust any of the five sliders, choosing whatever combination he/she like. The user has to click on the start button to start the training. The colors on the outer ring will change, as the training goes on. Each new update shows the predictions of the neural network algorithm for the complementary colors at that stage of the training. The use can compare the results with the colors in the middle ring. When the training is running, the Start button becomes a red Stop button, as shown in the figure below. The user can click on the stop button to halt the model training.



After the train starts, the user won't be able to change neither the Learning Rate nor the Batch Size, so these parameter must be wisely choosen. The training may stop for three reasons:

1. The user clicked on *Stop*.
2. The *Step Limit* was reached.
3. The *Cost Target* was reached.

If the training was manually halted, then the user can restart it by clicking on start again. If the *Step Limit* was reached, the user can **increase** the step limit, if possible, and restart the training. If the *Cost Target* was reached, the user can **decrease** the cost target, if possible, and restart the training. Whenever the training is halted, the user can Reset it by clicking on the *Reset* button (notice that the button is disabled when the training is running). A reset will clear all the training data (meaning that the model will "forget" the predictions made so far), but will keep the values chosen for the training session. This way the user can incrementally fine tune the training and explore the impact of each parameter change.

Notice that the initial parameters are not optimal, so the user must play with them a little bit before finding out a good workable combination of parameter input. The user is encouraged to try different combinations, play with them and, with luck, acquire some insights on how these parameters influence the training and the model predictions. One important thing to notice about machine learning training is that there is a random effect going on. The user will see it in action by running the model with the same parameters more then once. He/She won't get the same results twice! The demo the be found at the end of the article.

Apendix

Acknowledgements

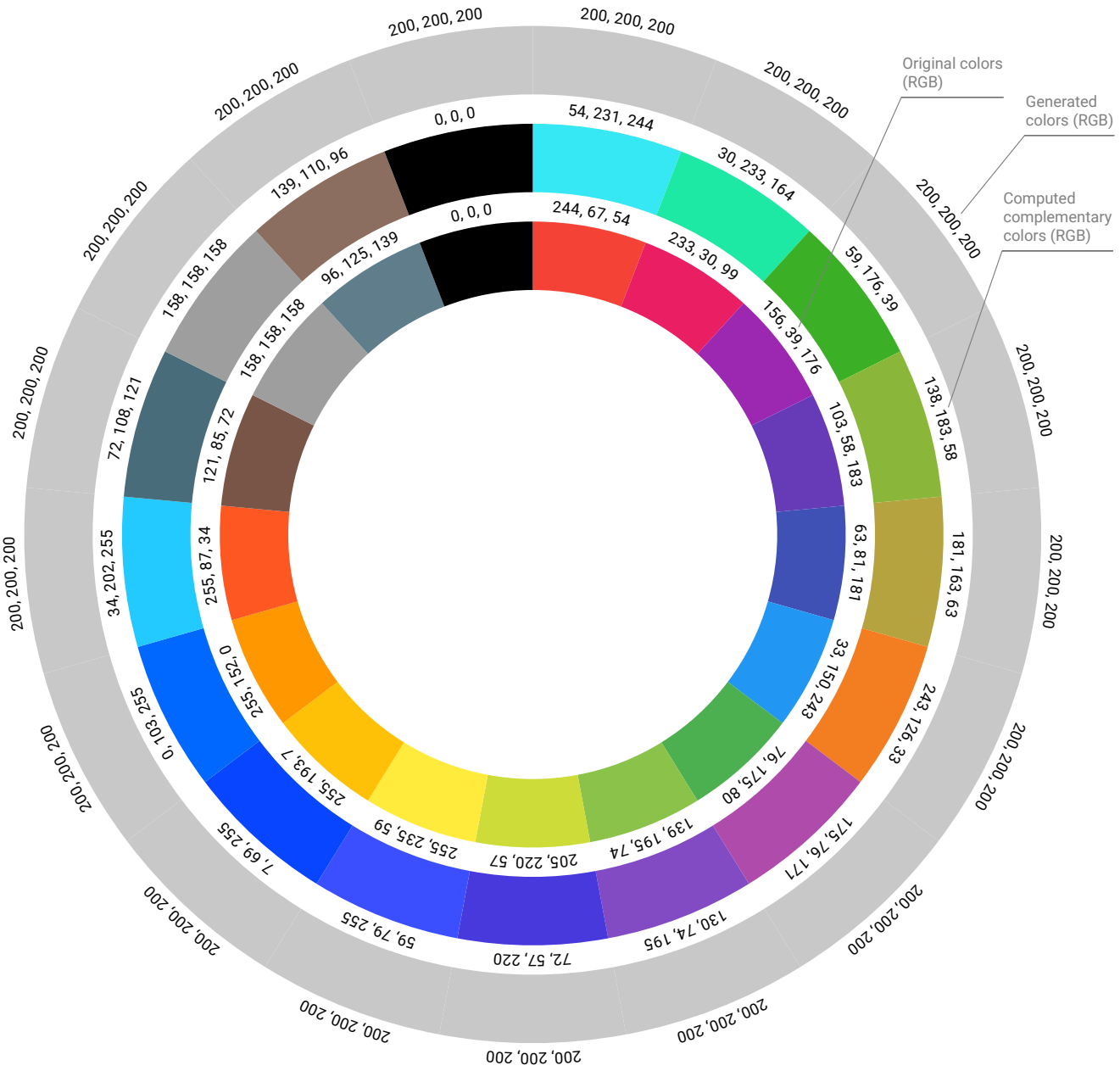
First and foremost we would like to thank the TensorFlowJS team, this is an amazing library that opens new doors to machine learning explorations on the browser. Tensorflow.js code base can be found here. A special thanks goes to Chi Zeng, whose code inspired this application.

References

1. **Deep learning**
   LeCun, Y., Bengio, Y. and Hinton, G., 2015. nature, Vol 521(7553), pp. 436. Nature Publishing Group.
2. **A logical calculus of the ideas immanent in nervous activity**
   McCulloch, W.S. and Pitts, W., 1943. The bulletin of mathematical biophysics, Vol 5(4), pp. 115—133. Springer.
3. **Backpropagation** [link]
   Wikipedia, ., 2018.
4. **Tensorflow.js** [link]
   Team, T., 2018.
5. **{TensorFlow}: Large-Scale Machine Learning on Heterogeneous Systems** [link]
   Mart'in~Abadi, ., Ashish~Agarwal, ., Paul~Barham, ., Eugene~Brevdo, ., Zhifeng~Chen, ., Craig~Citro, ., Greg~S.~Corrado, ., Andy~Davis, ., Jeffrey~Dean, ., Matthieu~Devin, ., Sanjay~Ghemawat, ., Ian~Goodfellow, ., Andrew~Harp, ., Geoffrey~Irving, ., Michael~Isard, ., Jia, Y., Rafal~Jozefowicz, ., Lukasz~Kaiser, ., Manjunath~Kudlur, ., Josh~Levenberg, ., Dandelion~Man\'{e}, ., Rajat~Monga, ., Sherry~Moore, ., Derek~Murray, ., Chris~Olah, ., Mike~Schuster, ., Jonathon~Shlens, ., Benoit~Steiner, ., Ilya~Sutskever, ., Kunal~Talwar, ., Paul~Tucker, ., Vincent~Vanhoucke, ., Vijay~Vasudevan, ., Fernanda~Vi'{e}gas, ., Oriol~Vinyals, ., Pete~Warden, ., Martin~Wattenberg, ., Martin~Wicke, ., Yuan~Yu, . and Xiaoqiang~Zheng, ., 2015.
6. **Tensorflow.js core** [link]
   Team, T., 2018.
7. **Complementary Colors** [link]
   Wikipedia, ., 2018.

8. **Oxford Learner's Dictionary - Complementary Color** [link]
Press, O.U., 2018.
9. **RGB Color Model** [link]
Wikipedia, ., 2018.
10. **TypeScript - JavaScript that scales** [link]
{Microsoft}, ., 2018.
11. **D3.js** [link]
Bostok, M., 2018.
12. **Complementary Color in RGB - StackOverflow** [link]
Edd, ., 2016.
13. **Explaining neural network concepts through an interactive visualization** [link]
Stelling, R., 2018.
14. **HSL and HSV Color Model** [link]
Wikipedia, ., 2018.
15. **On the importance of initialization and momentum in deep learning**
Sutskever, I., Martens, J., Dahl, G. and Hinton, G., 2013. International conference on machine learning, pp. 1139—1147.

Start	Reset	Learning Rate	Batch Size	Render interval	Step Limit	Cost Target
		0.42	10	5	1000	0.0005


Original colors (RGB)
Generated colors (RGB)
Computed complementary colors (RGB)
0, 0, 0
54, 231, 244
30, 233, 164
59, 176, 39
138, 183, 58
181, 163, 63
243, 126, 33
175, 76, 171
130, 74, 195
72, 57, 220
59, 79, 255
7, 69, 255
0, 103, 255
255, 152, 0
255, 193, 7
255, 235, 59
205, 220, 57
139, 195, 74
76, 175, 80
33, 150, 243
63, 81, 181
103, 58, 183
156, 39, 176
233, 30, 99
244, 67, 54
255, 87, 34
121, 85, 72
158, 158, 158
96, 125, 139
139, 110, 96
158, 158, 158
72, 108, 121
34, 202, 255
200, 200, 200