Hi!

This is a sample chapter from my upcoming book, Deep Learning With Javascript[1].

The book is organized into two sections, *Inference* and *Training*.

This sample chapter is from the *Training* section. It first discusses what Training, and then demonstrates how to build a Neural Network from the ground up.

I would love any feedback, as this is very much a work in progress.

— Kevin

* * *

[1] https://dljsbook.com

1. Training Your Neural Network

> "You want to know how to paint a perfect painting? It's easy. Make yourself perfect and then just paint naturally." - Robert M. Pirsig, Zen and the Art of Motorcycle Maintenance: An Inquiry Into Values

As users, people mostly engage with Neural Networks running in the **Inference** phase. These Networks are already trained to consume data and produce appropriate predictions for their use case.

You'll likely spend a majority of your time in the **Training** phase: teaching the Network how to produce the appropriate predictions you wish to see.

When training a Neural Network, you feed it data as well as the ground truths associated with that data so that it can learn how to produce those truths itself.

When a Network is running in Inference mode and making predictions, data flows one way - in the front, out the back. When training a Network, data flows *forward and backwards* through the Network. The Network takes some input and makes some predictions, compares those to the real truth, and tweaks its internal structure so that future predictions are more accurate.

There's three components to an effective training process: **Data**, **Rewards**, and **Practice**.

We've seen examples of **Data** in previous chapters. Data can be anything representable in numeric form. Images, audio, and video are all valid data, as is text, tabular data, financial information, LIDAR recordings, and more.

For our purposes, all the data we provide during training must have an associated label, indicating the thing we wish the network to learn. If you wish for your Image Detection Network to differentiate between pictures of cats and dogs, you must provide it a collection of images of cats and dogs with labels for each image indicating its species.

The type of training we cover in this book is **Supervised Learning**: training done in the presence of labeled data. There is another type of training, **Unsupervised Learning**, where the Network attempts to learn from unlabeled data.

Rewards guide a Network in improving itself, and Networks are incentivized to improve by reducing a thing called loss.

Loss is a measure of how wrong a network's predictions are. The bigger the loss, the more wrong the predictions. The Neural Network seeks to shrink the loss over time; the closer it gets to 0, the better.

Prediction	Actual	Absolute Value Error Difference
1	2	1
2	0	2
3	1	2
Total		5

In the above chart, if we sum up the error values - the absolute difference between the actual value and the predicted value - we get a loss of 5. This gives our Network a clear metric of how well it's doing, along with a direction with which to improve itself. How you choose to measure the loss is an important piece of your Neural Network' architecture. A number of commonly used loss functions come standard in all Deep Learning libraries, and there are general guidelines for which loss functions to use for what type of problem, though you can always experiment with new ones.

Loss tends to be correlated with accuracy, in that as loss goes down, accuracy goes up (though accuracy is always bounded between 0-100%, and loss can be of any scale).

`tensorflow.js` exposes a number of loss functions out of the box, including `meanSquaredError`, `meanAbsoluteError`, and `meanAbsolutePercentageError`. A full list of loss functions can be found in the documentation.[1]

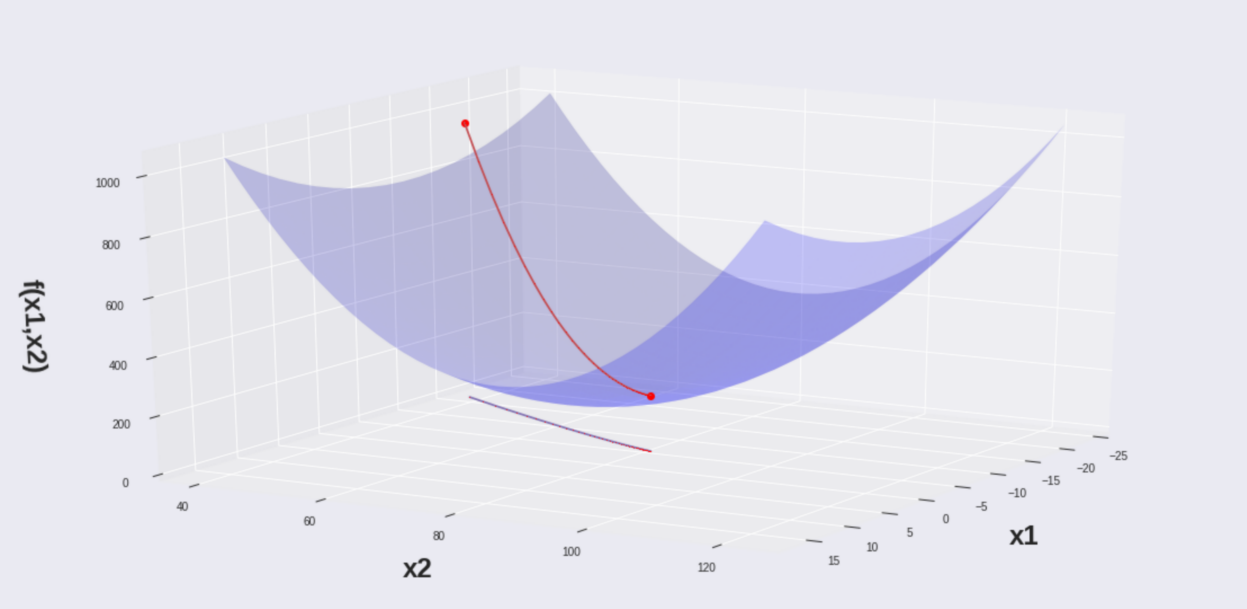
Just like you wouldn't expect a dog to learn a trick on the first try, your Neural Network needs to work and fail its way through many attempts before it's able to land on the correct answers.

The training process is an **iterative** one. It takes multiple cycles of training to get better. The more data, and the more complex the data, the more cycles of training you will need. The more complex the model, the more cycles of training you need. The more accurate you want your model to be, the more cycles of training you will need (up to a point - too many cycles can actually lead to overfitting which we'll touch on later).

[1] https://js.tensorflow.org/api/0.14.2/#Training-Losses

Why iterate? Why not just take a big leap and see where we land up?

Neural Networks learn via a process called **Gradient Descent**.



A visualization of gradient descent in action over a topology of Loss

In this image, the height of the topology represents Loss. The Network's goal is to descend the gradient to the lowest point - representing the lowest amount of loss (and conversely, the highest amount of accuracy).

You don't start off knowing this topology. The training process is in part an exploration and discovery of this space.

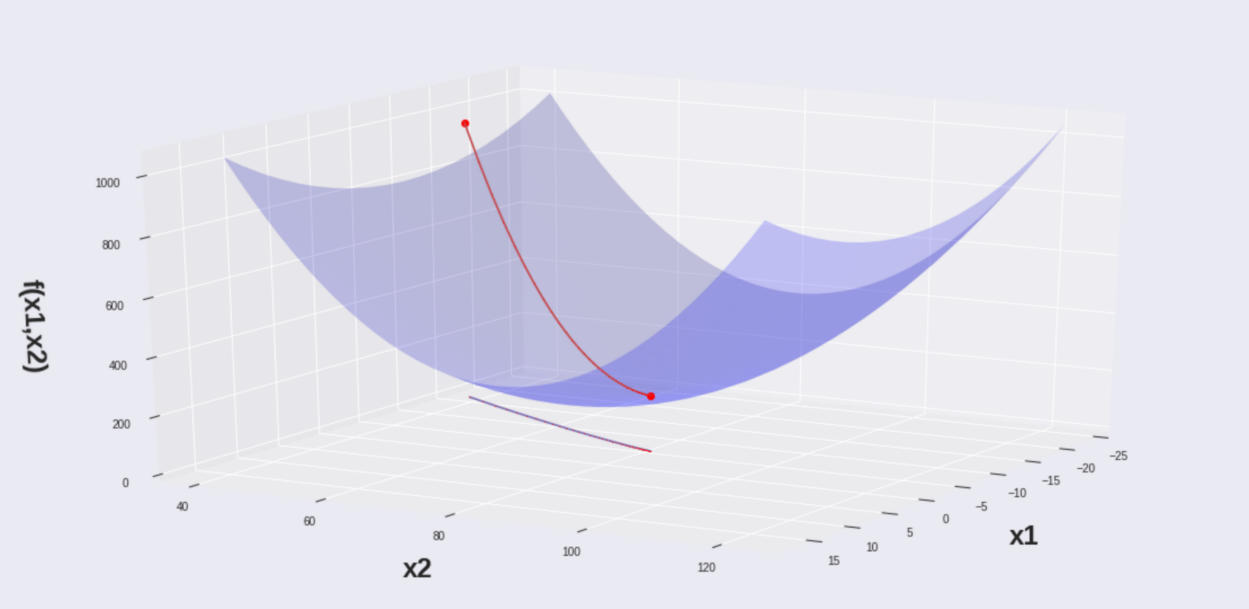
When you start, all you know is the initial point you start at, and which direction the slope goes. The only way to learn the topology is to take small baby steps. It's exploring a room in pitch darkness and navigating by feel; you know where you've been and what, roughly, is around you, but the majority of the room remains invisible to you.

The size of the step you take is called the **Learning Rate**. Once you've figured out which direction to go, the learning rate determines how big of a step your Network will take in that direction.

Let's see a number of different learning rates applied to the Network from the following chapter:

Learning Rate	Loss After 5 Cycles
0.001	25.33
0.01	11.64
0.1	0.38
1	355915424

With small learning rates, the Network learns slowly. This makes sense. If you force the Network to take many baby steps, it doesn't improve very quickly. However, the larger learning rate of 1 is almost comically inaccurate; this indicates the Network has *overshot* and is unable to improve itself effectively. Look at our hypothetical topology map:



Gradient Descent

If you're at a point on the left side looking to the right, a small step will take you downhill towards the bottom. But if you take a giant step in that direction you'll end up on the other side, potentially worse off than you were before. The equations tells you the direction in which to move, but not how far.

The Learning Rate of a Neural Network is called a **hyperparameter**, a value you can tweak during training to achieve better performance in your Network.

For any problem, there tends to be a sweet spot for the learning rate. You can and should experiment with different learning rates, and Tensorflow.js and other frameworks make it easy to play with it.

A good rule of thumb is to start with a simple dataset and simple model, and try to solve the simplest possible version of your problem. Once you've got something that works, you can make your model more complex, with increasingly complex architecture and increasing amounts of data. As your Network grows, you also have an opportunity to tweak various hyperparameters and observe their effects on your model.

We're going to take this iterative approach ourselves in the following chapter. We'll start with as simple a model as possible to teach the basics, and tackle harder and harder problems iteratively. Let's get started.

2. Lines and Nonlinear Data

Open https://dljsbook.com/i/line for an interactive repo

> "Machine learning is in the primordial soup phase. It's similar to database in the early '80s or late '70s. You really had to be a world's expert to get these things to work." — Gil Arditi, Product Lead for Lyft's Machine Learning Platform [1]

In this chapter, we're going to build a Neural Network from scratch using `Tensorflow.js`. Along the way we'll learn how to build a simple Neural Network from scratch, as well as encounter a few new topics related to effective Training.

We'll use a dataset consisting of a simple line, and build a Network that can predict Y points on that line for some given X. You can play with the dataset with the following code:

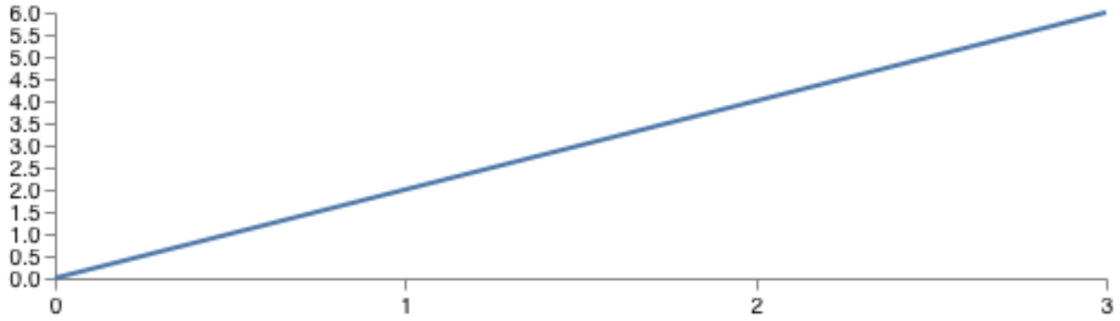
```
import { Line } from '@dljsbook/data'

const line = new Line()

const {
  data, // an array of x points
  labels, // an array of y points
  print, // a function to print a line chart to tfjs-vis
} = line.get()
```

We can call `print()` to see the data:

[1] https://venturebeat.com/2017/10/24/lyfts-biggest-ai-challenge-is-getting-engineers-up-to-speed/

```
print()
```

This will write out a graph to `tfvis` if available, or your console if not.

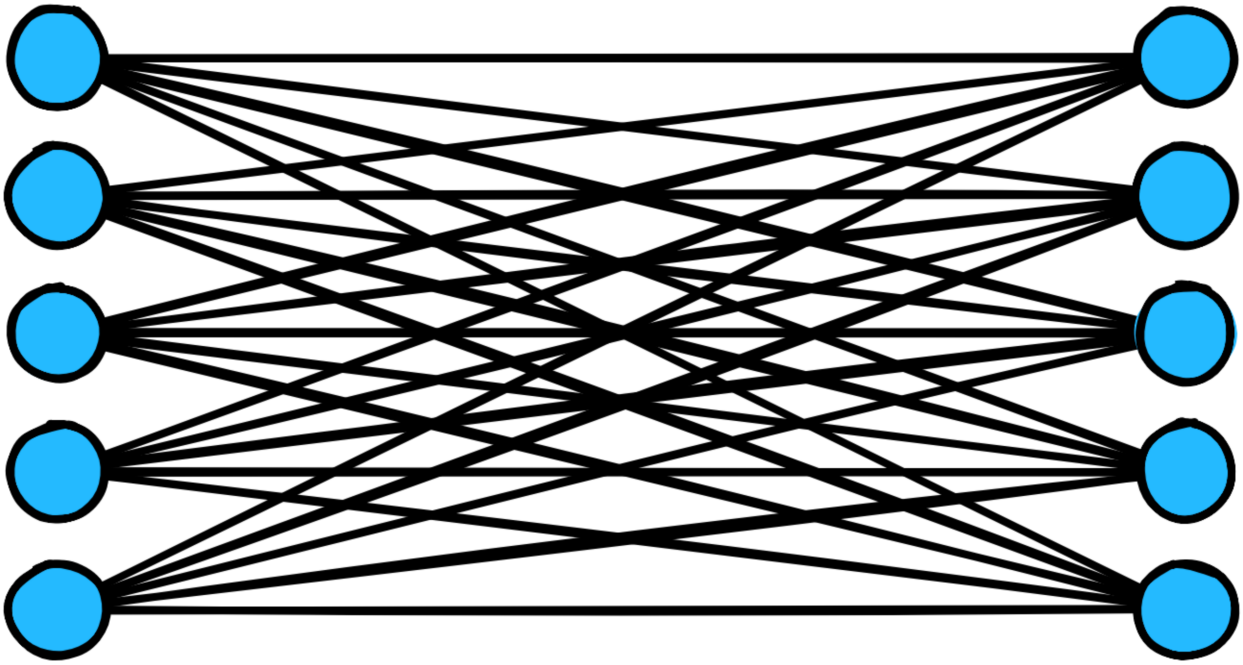


A simple line

We'll start by deciding which *type* of Network to use. Tensorflow.js offers us two options: `tf.sequential` and `tf.model`. Sequential models are more common, and that's we'll use. I like to think of a Sequential model like a pancake, because I like pancakes.

A sequential model reminds me of a delicious, delicious pancake

A Sequential model has some number of layers, all stacked on top of one another, with the Neurons in one layer connected to the Neurons in the following layer.



Two layers in a Sequential Network

Let's build our Network. We can create a Sequential model with:

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

Next, we'll add a single **dense** layer. A dense layer means that every Neuron in the layer is connected to every Neuron in the previous layer. To add a single dense layer, we can write:

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

This code specifies that the layer has an input shape of 1 (it's 1 because we pass it a single number, the "x" coordinate), and that the layer has a single Neuron, or "unit" (unit is another name for Neuron).

Our Network now looks like this:

A Neural Network built for predicting points on a line

Finally, we can compile our model with:

```
model.compile({
  loss: 'meanSquaredError',
  optimizer: 'sgd',
  metrics: ['acc']
})
```

`compile` is the command for "packaging" up your network and it allows us to specify the precise mechanisms through which the Network will improve itself. Compiling a model indicates to the framework that the model is ready to begin training. (We'll discuss each of the options in `compile` later on.)

Let's try and make a prediction without any training.

```
function predict(x) {
  const prediction = model.predict(tf.tensor2d([x], [1,1]))
  return prediction.dataSync()[0]
}
const x = 5 // an X value. We want to predict Y
const y = predict(x)
console.log(y)

// your value will be different from this one
> -0.5574913024902344
```

The Neural Network returns a wildly wrong prediction. This is to be expected.

When a Neural Network is first created, its weights are randomly initialized. So if you try and make a prediction before you train it, your prediction will effectively be a random number.

There are ways of controlling how a Network is initialized, that can in turn affect how the Network trains. See the `kernelInitializer` documentation[2] in Tensorflow.js.

Let's train our model so we can see some more accurate predictions. Get line data with:

```
const { data, labels } = (new Line()).get()
```

When training, we consistently separate our data into `data` and `labels`. `data`, also commonly known as "features", is the stuff we train our Network upon, and `labels` are the ground truth that our Network aims to predict.

We can now train the model. We do that by calling `model.fit`:

```
model.fit(data, labels, {
  epochs: 10,
  shuffle: true,
})
```

An **epoch** stands for an *entire single training cycle through your full dataset*. In this case, we're asking our Network to train for 10 cycles, so it will see the same data 10 times.

`shuffle` means that the data will be shuffled each training cycle, ensuring the Network does not learn to make predictions based on the order of the data.

Once the model is done training, you can make a prediction using the same code from above:

[2] https://js.tensorflow.org/api/0.14.2/#layers.dense

```
model.fit(data, labels, {
  epochs: 10,
  shuffle: true,
}).then(() => {
  const y = predict(model, 5)
  console.log(y)
})
```

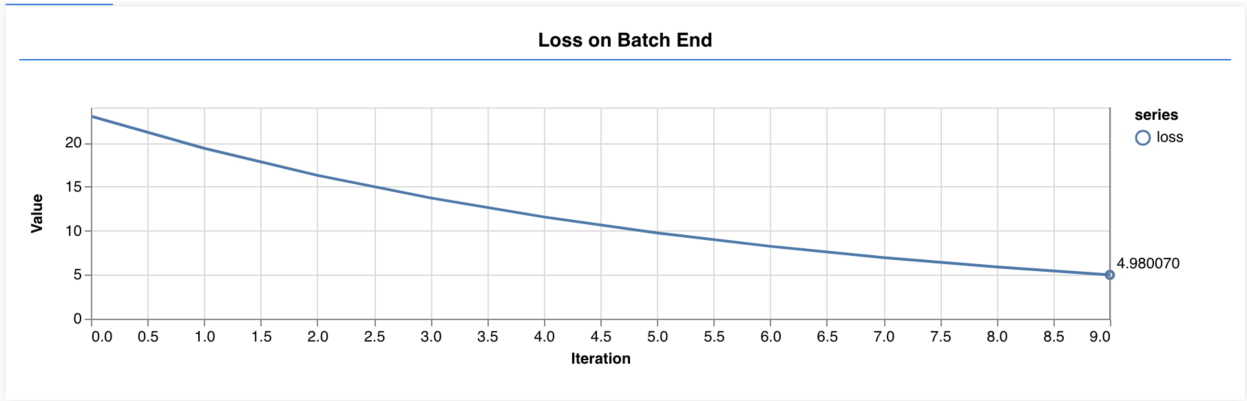
We should see a prediction closer to 10 than before.

Assuming `tfvis` is installed, we can chart out the network's improvement over time by passing some metrics.

```
import * as tfvis from '@tensorflow/tfjs-vis'
const { data, labels } = (new Line()).get()
const model = getModel()
const batchLoss = []
model.fit(data, labels, {
  epochs: 10,
  shuffle: true,
  callbacks: {
    onBatchEnd: (batch, h) => {
      batchLoss.push(h)

      // if tfvis is installed
      tfvis.show.history(
        { name: 'Loss on Batch End', tab: 'Training' },
        batchLoss,
        ['loss', 'acc'],
      )

      // if tfvis is not installed
      // console.log(batchLoss)
    },
  },
```

```
}).then(() => {
  const y = predict(model, 5)
  console.log(y)
})
```

We should see a smooth descending loss measurement as our Network trains:



A graph of the loss

Every cycle of training, the loss goes down. The network is "learning". Here's a table of predictions for every cycle of training (your numbers will be different but should point in the same direction):

Epoch	Prediction	Loss
0	-4.24	33.27
1	-3.17	27.99
2	-2.19	23.56
3	-1.28	19.84
4	-0.46	16.72
5	0.30	14.10
6	1.00	11.90
7	1.63	10.05
8	2.22	8.50
9	2.76	7.20

Every cycle of training, the network gets closer and closer to the correct prediction: 10. We may find that after training, the network's accuracy is still pretty bad. If this is the case, there's three ways to improve it:

1. Increase the number of training cycles.

2. Increase the amount of data.
3. Change the model's architecture.

It's best to start with one of the first two approaches and see if they can eke out additional gains. Wherever possible, let the computer work harder, not you.

We can increase the amount of data and the number of epochs with:

```
// increase the amount of data
const { data, labels } = (new Line()).get(50)
model.fit(data, labels, {
  epochs: 20, // increase the number of epochs
  shuffle: true,
})
```

And these should produce better predictions. In my testing, I saw:

Epoch	Prediction	Loss
15	7.80	0.45
16	7.91	0.39
17	8.02	0.34
18	8.11	0.30
19	8.20	0.26

These predictions are much closer to the truth, and the losses much closer to zero, than before.

* * *

Open https://dljsbook.com/i/nonlinear for an interactive repo

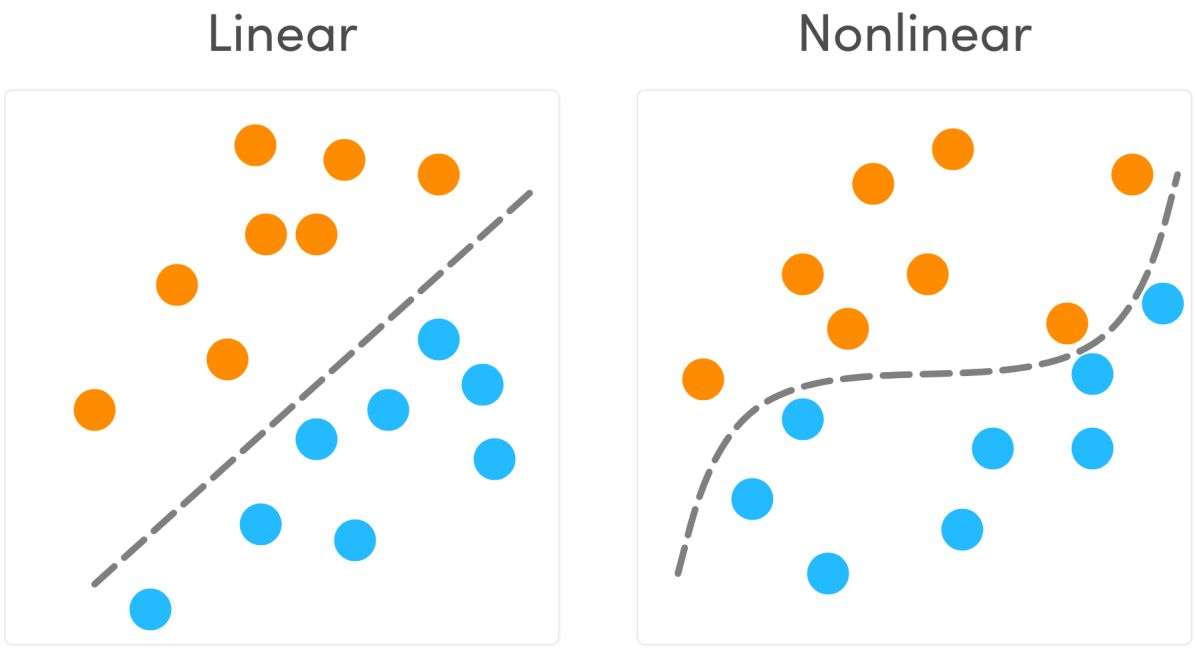
Congratulations! You've built your first Neural Network from scratch. The principles outlined in the previous section are the same ones you'll use to train much more complicated Networks later on.

This is about as simple as a Neural Network can be - a single Neuron in each layer, with only a single hidden layer, and a very simple dataset. However, the simplicity of the dataset means that this particular problem is not the best fit for a Neural Network.

Despite all the hype around Deep Learning, the truth is that for any problem for which there's an equally valid approach - a statistical method, an algorithm, a heuristic - you're better off using that method instead. The downsides of Neural Networks are that they can require more data, be slower to train, and often produce uninterpretable models.

One use case Neural Networks *do* excel at is handling non-linear data.

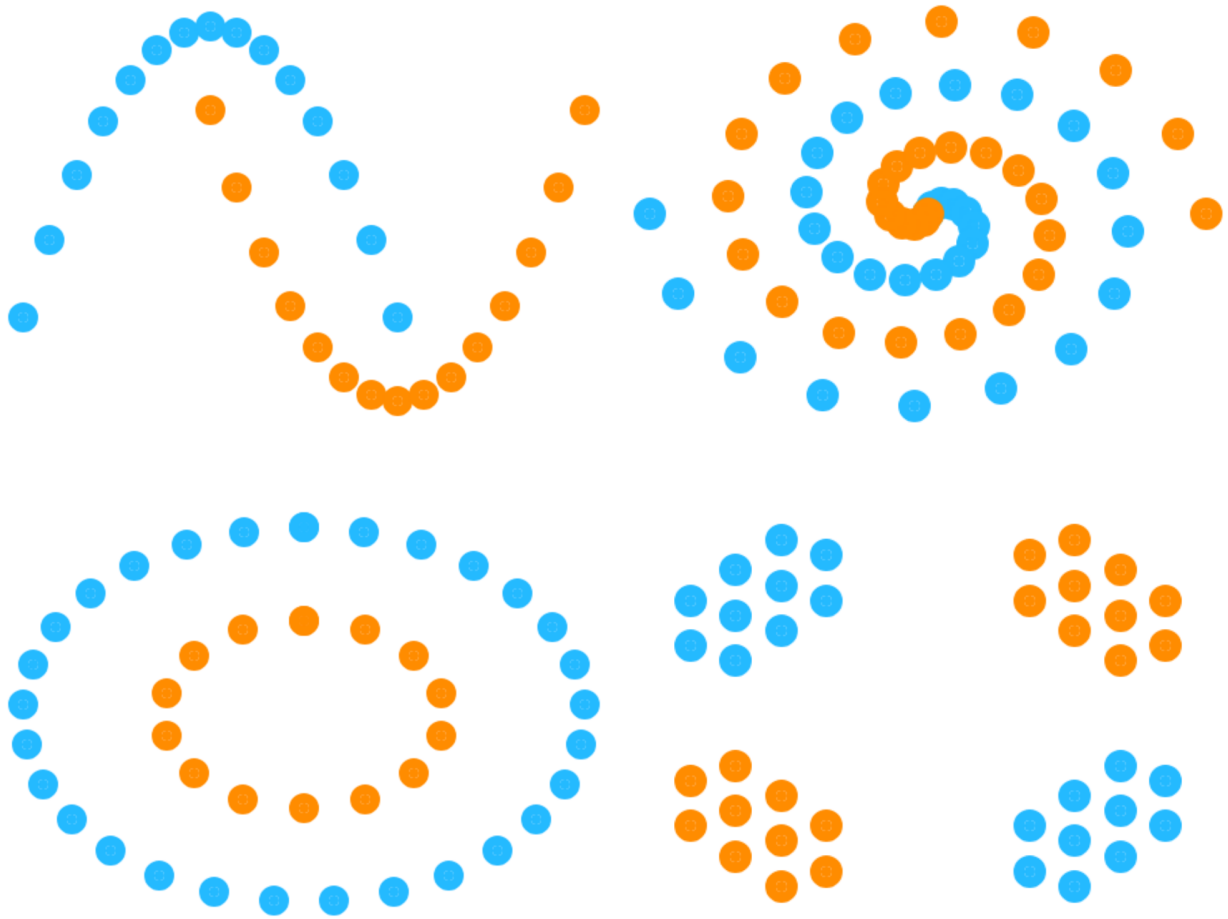
Non-linear data is data that is not "linearly separable". You can't draw a straight line through it.



An example of linear vs nonlinear data

Non-linear data encompasses a startlingly large array of problems. It encompasses the rich data we've discussed in previous chapters, things like images, audio, video, and text. A lot of signal data can be non-linear as well.

Let's build a Neural Network capable of handling non-linear data. We have a few datasets at our disposal:



Examples of nonlinear data

You can load each dataset by importing it, and each dataset has the same API. The datasets are:

- `Moons`
- `Clusters`
- `Swirl`
- `Circles`

Let's load the `Moons` dataset and inspect it:

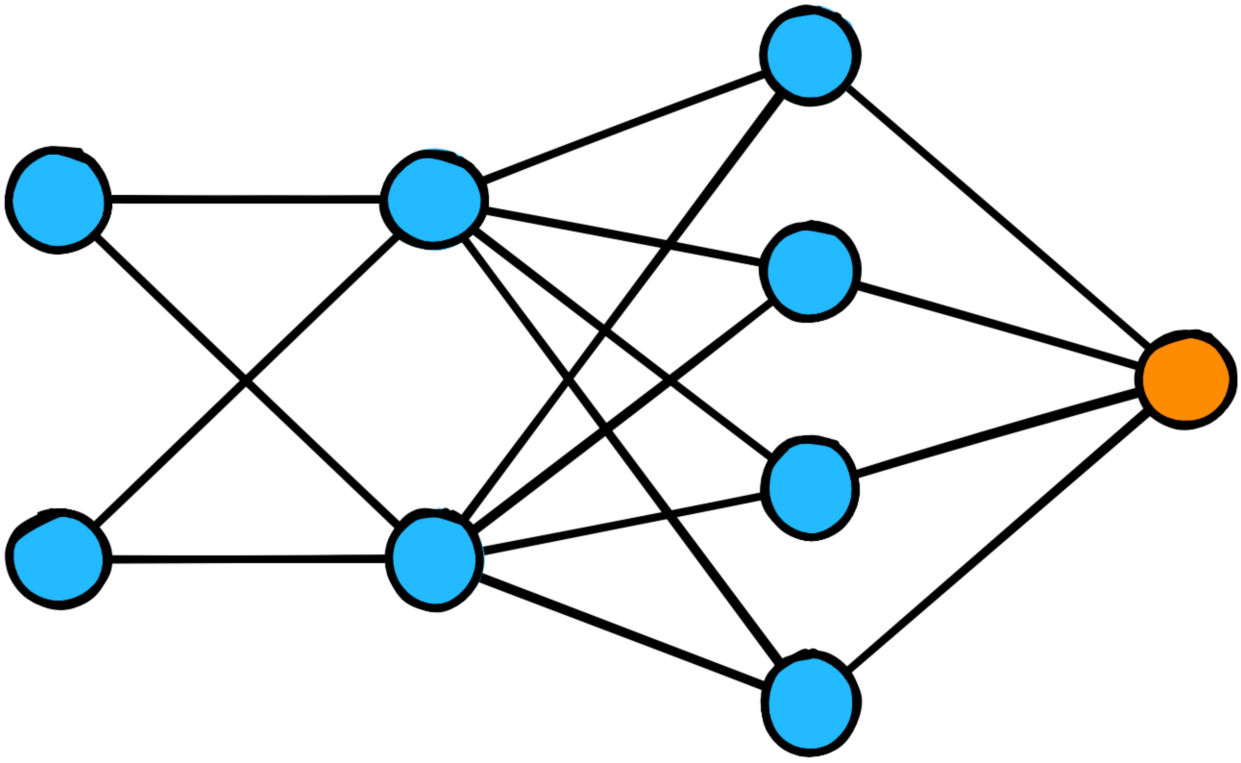
```
import { Moons, MOONS_POLARITY } from '@dljsbook/data'
const moons = new Moons()
moons.get().print()
```

This will write out a graph to `tfvis` if available, or your console. Let's build a Neural Network to handle these nonlinear datasets.

We'll start by writing a `getModel` function that sets up a sequential model, our pancake-like structure from the previous chapter:

```
const getModel = () => {
  const model = tf.sequential()
}
```

Writing a specific function for building a model allows us to later specify hyperparameters easily as arguments, giving us greater flexibility to experiment. The Network we're going to build looks like this:

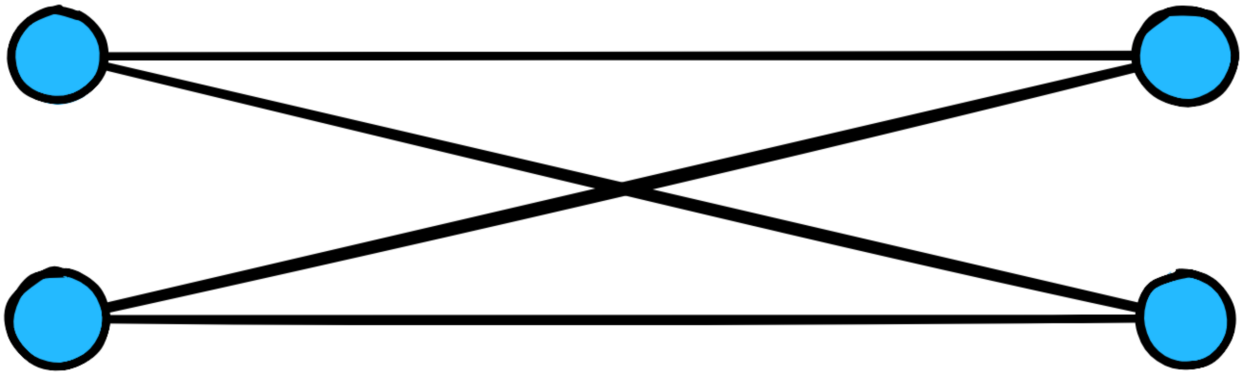


Our Neural Network structure

Each point in our dataset has an `x` and `y` coordinate, and an associated color. We want to predict the color, which is our "label". `x` and `y` make up our two input values, an `inputShape` of 2, which we can add to our function with:

```
model.add(tf.layers.dense({ units: 2, inputShape: [2] }))
```

Tensorflow.js, along with other Deep Learning frameworks, provide a shorthand for specifying your input layer and first hidden layer in one line. In the above code, our input layer has two Neurons, and our first hidden layer also has two Neurons. Let's see what this looks like so far:



Input layer and hidden layer

We can add our second hidden layer with:

```
model.add(tf.layers.dense({ units: 4, activation: 'relu' }))
```

`units` is arbitrary here. You can experiment with different numbers. More neurons can lead to better performance at a cost of speed, but will only improve your Network up to a point.

`activation` is something we haven't seen before, and is really the key to how Neural Networks are able to so successfully learn patterns in nonlinear data. Let's take a brief digression to discuss them.

Activation Functions

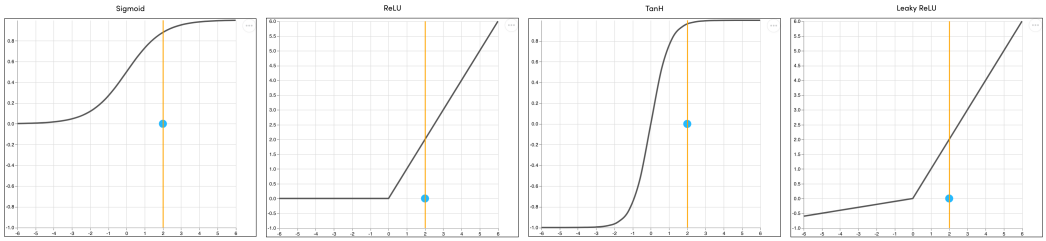
Activation functions are a large reason why Neural Networks can learn to recognize rich data like images, sound, and video.

An activation function is simply a mathematical function that transforms a numeric input into an output, non-linearly. Put together at scale, they enable Neural Networks to learn non-linear features of datasets.

> If we only allow linear activation functions in a neural network, the output will just be a linear transformation of the input, which is not enough to form a universal function approximator. Such a network can just be represented as a matrix multiplication, and you would not be able to obtain very interesting behaviors from such a network. - HelloGoodbye[3]

Activation functions introduce the concept of nonlinearity into the network. Although activation functions work within individual Neurons, we define them at the layer level, so that all Neurons in a layer use the same kind of activation function.

Here's some of the most common activation functions you'll see:

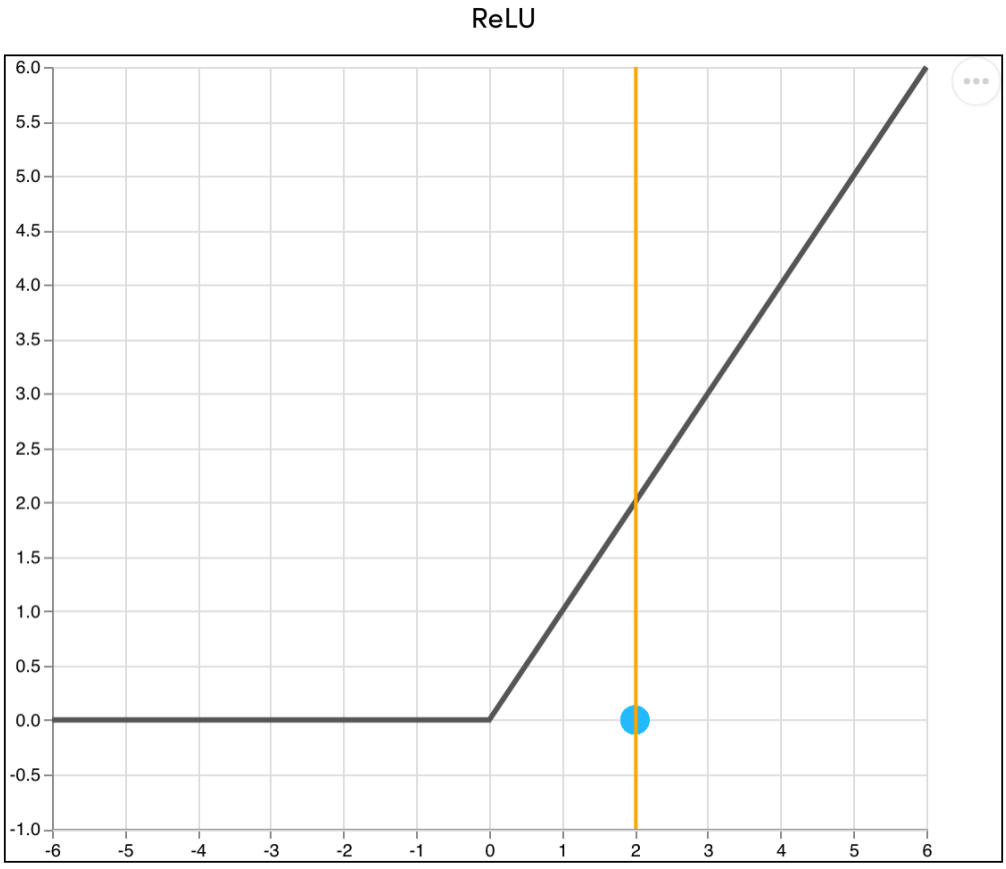


Common activation functions

You can play with these different activation functions yourself at https://dljsbook.com/activation-functions.

ReLU - which stands for Rectified Linear Unit - is one of the most common activation functions you'll see. It turns anything less than `0` into `0`, and anything greater than `0` is left alone. In graphical form, it looks like this:

[3] https://stackoverflow.com/a/34816528



ReLU

And in code form, the function would be:

```
function relu(x) {
  return (x > 0) ? x : 0
}
```

Let's return to our in-progress Network. We were building a second hidden layer, that looked like:

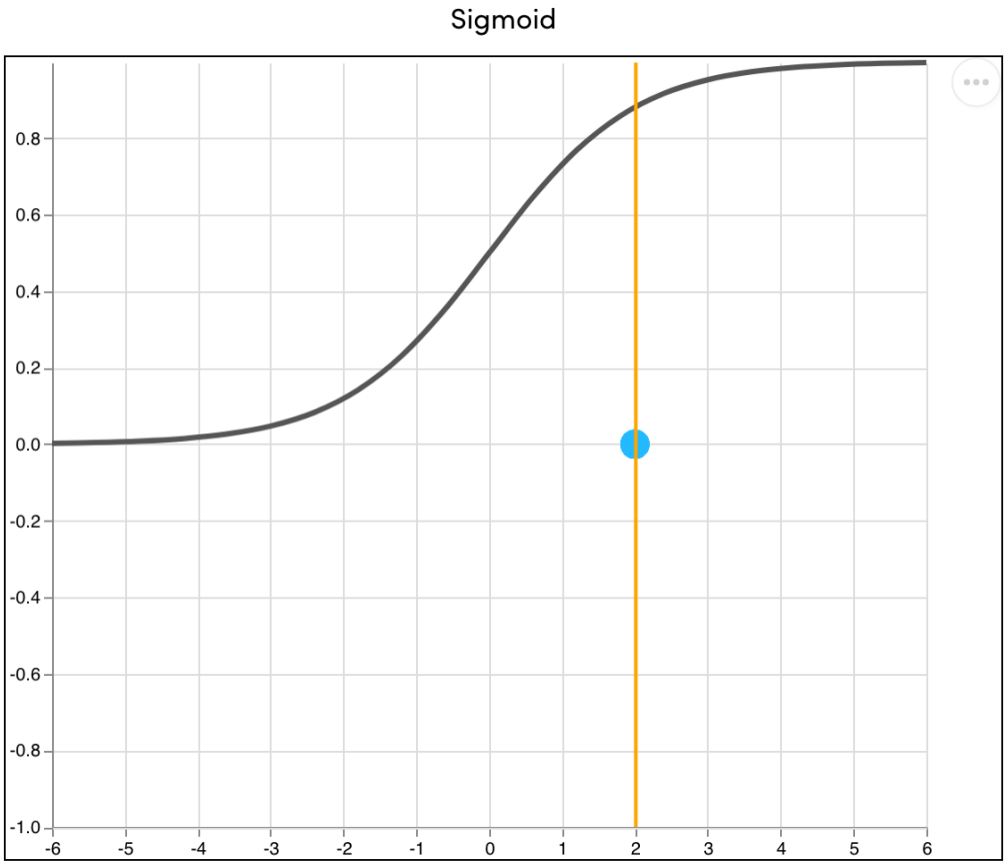
```
model.add(tf.layers.dense({ units: 4, activation: 'relu' }))
```

The activation function `relu` here will transform any incoming values less than 0 to 0, and anything above will be let through unadulterated.

Our final layer - our output layer - has a single unit. You can think of this unit as the color prediction:

```
model.add(tf.layers.dense({ units: 1, activation: 'sigmoid'}))
```

We use a Sigmoid activation, which looks like this:



Sigmoid

The Sigmoid activation function forces incoming values to be between 0 and 1. It's commonly used in the last layer of a Neural Network. Since we're predicting one of two colors, this is a *binary classification* problem. If the value is less than 0.5, we'll call it 0, otherwise we'll call it 1.

The last step is to compile the model, like before:

```
const LEARNING_RATE = 0.15
const optimizer = tf.train.adam(LEARNING_RATE)
model.compile({
  optimizer,
  loss: 'binaryCrossentropy',
  metrics: ['accuracy'],
})
return model
```

`metrics` is a parameter that tells `Tensorflow.js` what to measure. We want it to measure accuracy here.

We discussed the concept of learning rate, a hyperparameter in the previous section. Loss functions tell the network how to measure the loss, and optimizers describe how the Network should improve itself.

Side note: when I was starting out, I struggled a lot with what values to use for these hyperparameters. To be honest, even with advanced Networks, there's a lot of trial and error involved in picking these numbers. In my experience, a combination of seeing what's worked best for similar problem domains, studying up on some of the underlying math, and brute trial and error have been all I need to get pretty decent results.

For binary classification problems, `binaryCrossentropy` is an appropriate loss function to use. The `adam` optimizer is used in a lot in Neural Networks, and is usually a pretty safe default optimizer to reach for.

Our full code for building the model:

```
const getModel = () => {
  const model = tf.sequential()
  model.add(tf.layers.dense({ units: 2, inputShape: [2] }))
  model.add(tf.layers.dense({ units: 4, activation: 'relu' }))
  model.add(tf.layers.dense({ units: 1, activation: 'sigmoid'}))

  const LEARNING_RATE = 0.15
  const optimizer = tf.train.adam(LEARNING_RATE)
  model.compile({
    optimizer,
    loss: 'binaryCrossentropy',
    metrics: ['accuracy'],
  })
  return model
}
```

Before we train our Network, let's define some callbacks that let us visualize our Network learning. These make use of `tfvis`, but you can always call `console.log` if you'd prefer.

```
const batchLoss = [];

// onBatchEnd will be called at the end of every batch, and will graph
// the loss over time
function onBatchEnd(batch, h) {
  batchLoss.push(h)
  tfvis.show.history(
    { name: 'Loss on Batch End', tab: 'Training' },
    batchLoss,
    ['loss'],
  )
}

// onTrainEnd will be called when training is complete, and will make
// some predictions on the freshly trained model
function onTrainEnd() {
```

```
  for (let i = 0; i < 5; i++) {
    const type = Math.random() >= .5 ? MOONS_POLARITY.POS : MOONS_POLA\
RITY.NEG
    const { data } = Moons.getForType(type)
    const prediction = Math.round(model.predict(data).as1D().dataSync(\
))
    console.log(`${type === prediction ? '+' : 'x'} | Actual: ${type} \
| Pred: ${prediction}`)
  }
}
```

A full list of training callbacks can be found in the Tensorflow.js documentation[4].

We're ready to train our Network, and we can do that by calling `model.fit`:

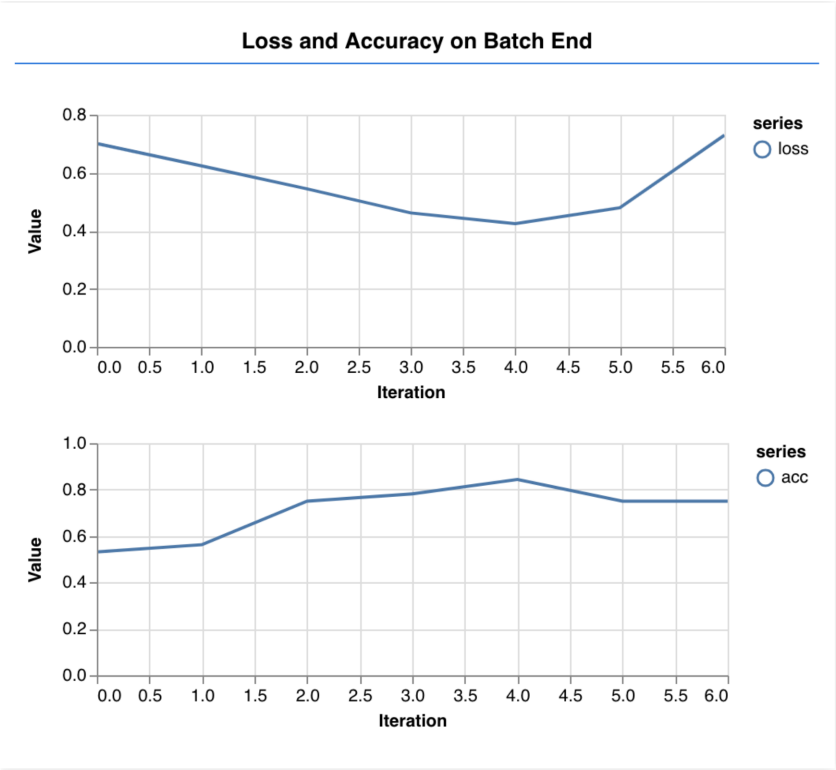
```
const model = getModel()
const {
  data,
  labels,
} = (new Moons()).get()

model.fit(data, labels, {
  shuffle: true,
  epochs: 1,
  callbacks: {
    onBatchEnd,
    onTrainEnd,
  },
})
```

I get somewhere around 80% accuracy out of the box, which is not an amazing result.

[4] https://js.tensorflow.org/api/0.14.2/#tf.Model.fit

Neural Networks demand a paradigm shift in how we approach the craft of building software. It's often impossible to achieve perfect performance, so for any non-trivial Deep Learning task, an important first step is to determine what sort of performance is considered acceptable. Human performance often serves as a reasonable benchmark. If we know that humans can classify 90% of given images, then 90% is a pretty good baseline to shoot for.

In this case, a human could reasonably be expected to classify better than 80% datapoints. Let's see if we can do better.



A log of the loss and accuracy during training our non-linear model

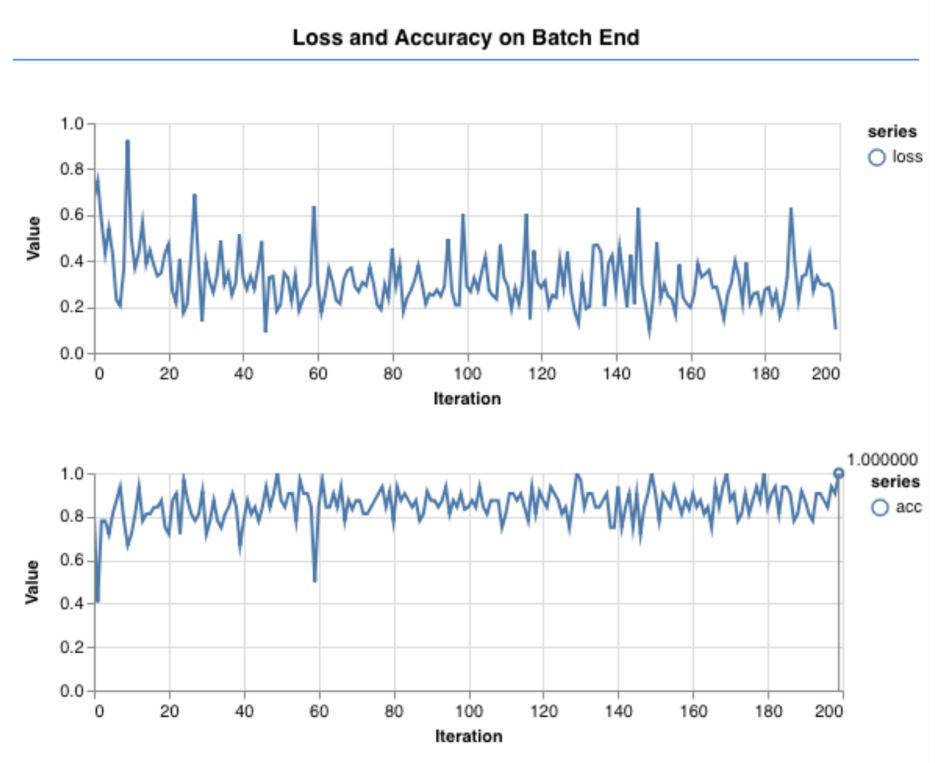
Remember, to improve performance, you can:

1. Increase the number of training cycles.
2. Increase the amount of data.
3. Change the model's architecture.

Let's start with the first two. Increase the data size, and increase the epochs:

```
const model = getModel()
const {
  data,
  labels,
} = Moons.get(500) // increase your data size

model.fit(data, labels, {
  shuffle: true,
  epochs: 20, // increase your training time
  callbacks: {
    onBatchEnd,
    onTrainEnd,
  },
})
```

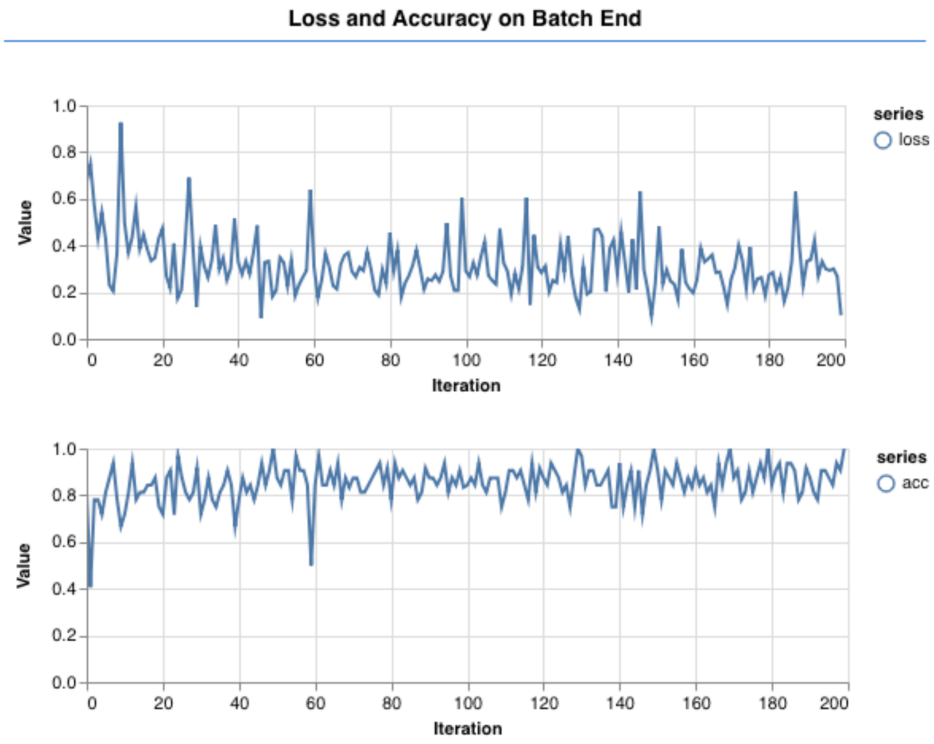
This should get performance to about 90%. An improvement, but we can do better.



A log of the loss and accuracy with more epochs and more data

What happens if we play with our Network architecture, and add another hidden layer?

```
const getModel = () => {
  const model = tf.sequential()
  model.add(tf.layers.dense({ units: 2, inputShape: [2] }))
  model.add(tf.layers.dense({ units: 4, activation: 'relu' }))
  model.add(tf.layers.dense({ units: 4, activation: 'relu' }))
  model.add(tf.layers.dense({ units: 1, activation: 'sigmoid'}))
}
```



Complex Nonlinear Training

I'm able to get well into the upper 90s with this Network.

This is generally a strategy worth following when training a Network. Build something simple, observe the results, hypothesize and tweak, and try again.

In the next chapter, we'll look at some tabular data in a spreadsheet and see if we can build

a Network to handle that.

* * *

And that's it for now! What'd ya think? Lingering questions? Missing parts?

I'd love to hear what you think and how I can improve it.

I'm releasing the book at the end of February. In return for feedback, I'd be happy to send you a digital copy of the book for free.

Write to me at:

feedback@dljsbook.com

Hope to hear from you!

— Kevin