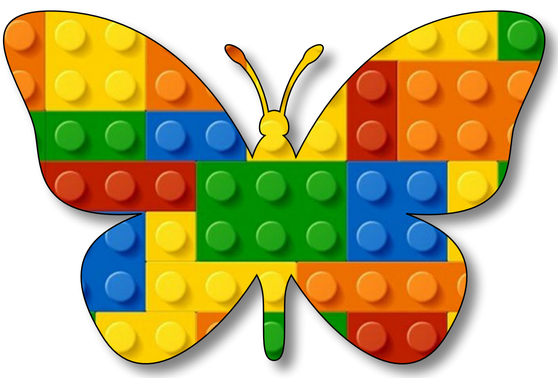
Artificial Intelligence information a gentle overview of ml5.js



Content

Introduction to ml5.js

In this section we will be introducing:

- the ml5.js neural network
- tasks
- the Debug tool
- the datasets
- the callback() function

As well as changing various hyperparameters:

- Epochs
- Batch Size
- Hidden Layers
- Number of Nodes (Neurons)
- Activation functions

Introduce the key functions within ml5.js:

- .addData()
- .normalizeData()
- .train()
- .predict()
- .classify()
- .save()
- .load()

Guide to adding ml5.js

We will be building some neural networks. The beauty of modern machine learning is that you don't have to build the neural network from scratch. We will be using a library called ml5.js, which is based on TensorFlow. This means that much of the heavy lifting is already done for you. Even so, there is so much to learn, experiment with and develop so don't think it is a complete doddle, you still need to know what you are doing.

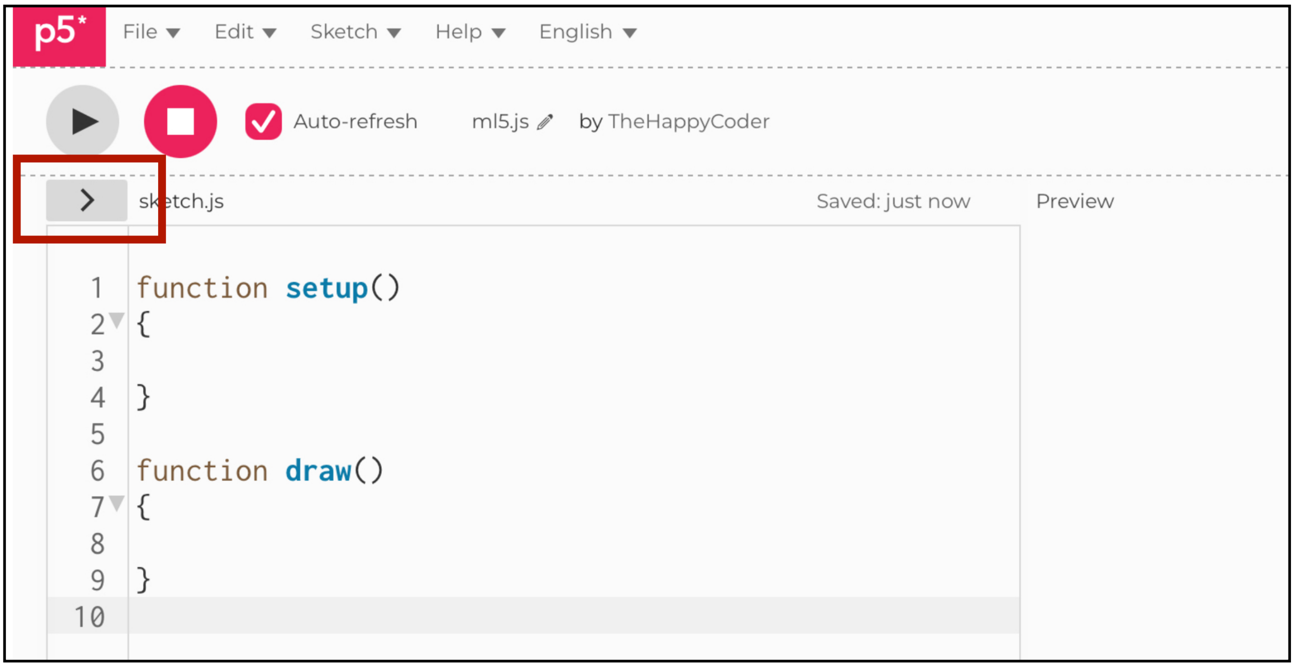
The units you will work through will provide opportunities to build your neural network for a variety of tasks. They are all fun activities but they are good learning opportunities as well. They will introduce you to some key concepts and deepen your understanding of the possibilities and limitations of machine learning.

To get started we need to import the ml5.js library and to do so you need to follow the next steps carefully to include the line of code needed in the index.html. If you want to I will give you the starting sketch for you to duplicate and save (see the button on my website).

STEP 1️⃣ the arrow

There is a grey arrow on the top left hand side (as indicated in the image below). Click on that arrow to bring up a list of files. You will see, `index.html`, `sketch.js` and `style.css`.

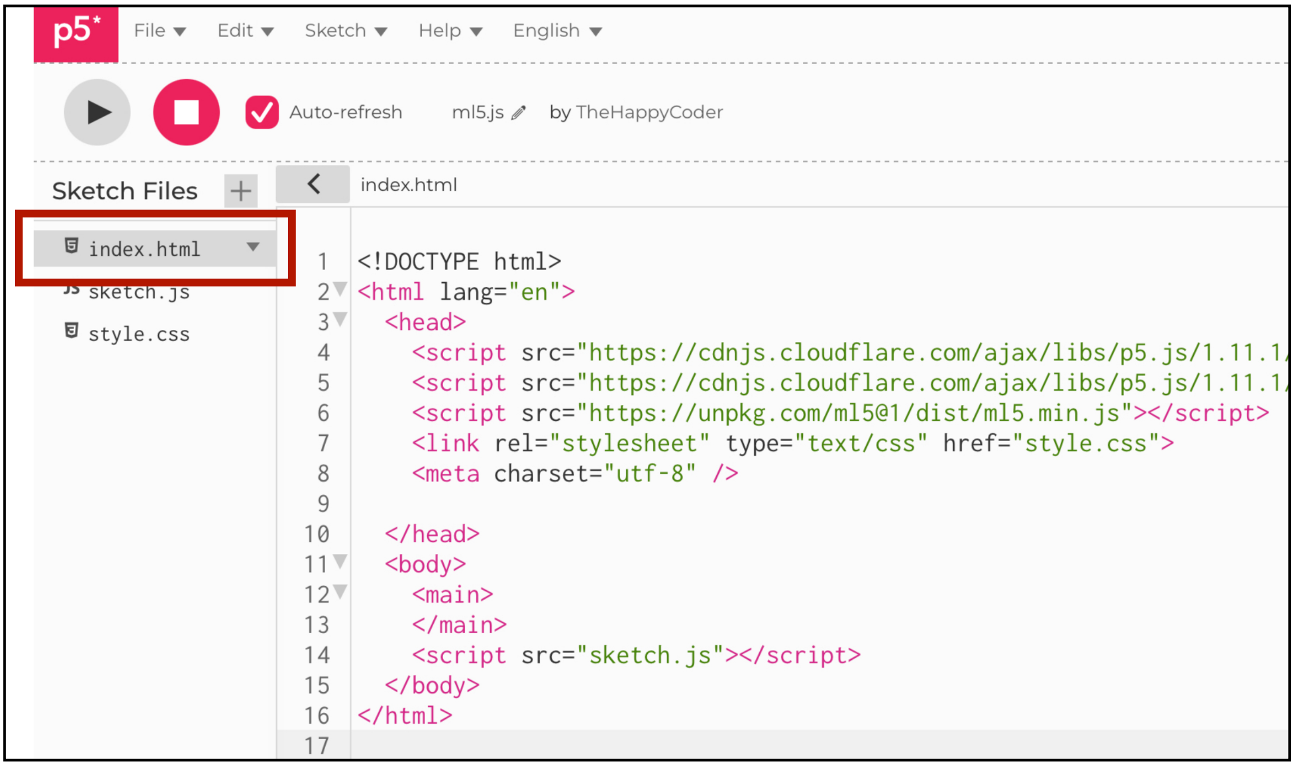
Figure 32 finding the index.html file



STEP 2️⃣ the index file

Find the file that is called `index.html` and click on that file. Once you have selected the index.html file you can close the menu by clicking on the grey arrow again. Then grab the side of the coding section and drag it so that the code fills your screen.

Figure 33 found the index.html file



STEP 3️⃣ ml5.js the line of code

Finally type in the line of code exactly as below which will allow your browser to access the module.

```
<script src="https://unpkg.com/ml5@1/dist/ml5.min.js"></script>
```

Then go back to sketch.js by clicking on it in the side tab and the arrow to exit the list of files. Hopefully you typed everything in OK. You will find out in due course.

Figure 34 adding ml5.js

The index.html file

The new line of code can slot in anywhere inside the <head> tags along with the other similar lines of code.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/
1.11.1/p5.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/
1.11.1/addons/p5.sound.min.js"></script>
    <script src="https://unpkg.com/ml5@1/dist/ml5.min.js"></
script>
    <link rel="stylesheet" type="text/css" href="style.css">
    <meta charset="utf-8" />

  </head>
  <body>
    <main>
    </main>
    <script src="sketch.js"></script>
  </body>
</html>
```

The ml5.js neural network

We are going to be working with the default settings in ml5.js. I have provided the default settings for information, don't worry if you don't know what they mean, all will become clear as we go along.

The default setup for an ml5.js model for `regression` is:

- Hidden layers: 1
- Hidden layer activation function: `relu`
- Number of node: 16
- Output layer: `sigmoid`

For `classification` it is the following:

- Hidden layers: 1
- Hidden layer activation function: `relu`
- Number of node: 16
- Output layer: `softmax`

Other default settings:

- Learning Rate: `0.2`
- Batch size (?): 32
- Epochs: `10`

The task in hand

The type of task has to be identified first. This is so that you know what type of model you are going to use. There are primarily two main types, `classification` and `regression`. You need to decide which one it is going to be, this is your first job and tell that to ml5.js.

Although the neural network is almost identical for both tasks, the output gives the game away. The neural network will be predicting an output, if that is to classify something, for instance an image of a dog, then it is a `classification` task. If the prediction is a value which can change, for instance the price of a house then it is a `regression` task.

🀄 Classification

An example could be sorting images of different animals, say, cats and dogs. The model will be trained on thousands of images of cats and dogs, each image labelled accordingly. Once trained, the model will then be shown an image of a cat or dog and will `predict` the likelihood (probability) that it is a cat or a dog. There will be two outputs, one, probability of it being a cat and two, the probability of it being a dog.

🀄 Regression

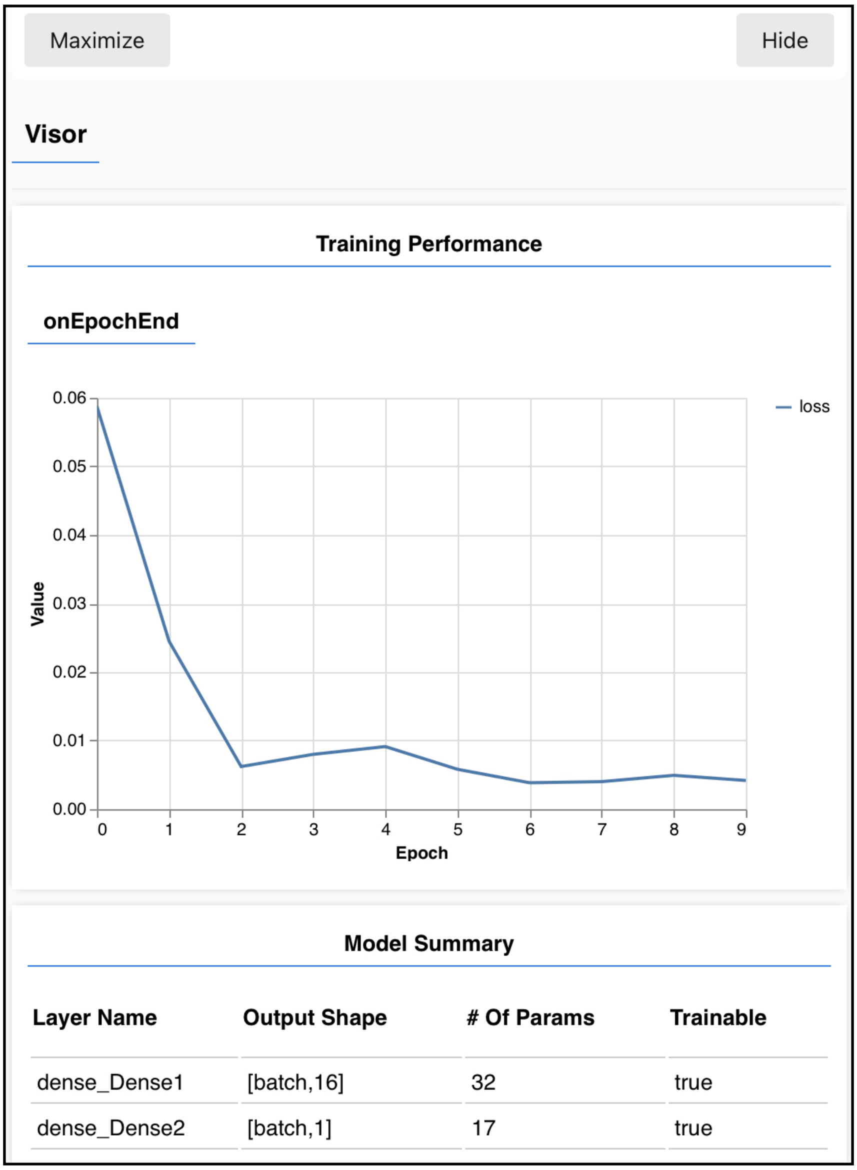
This usually only has one output. It will be a value, say, the price of house, or predicting the temperature, or the price of a cryptocurrency. This will be a value that can have a range of values and once trained it predicts what it thinks it will be, this is based on the inputs. In the case of a house price, the inputs may include the number of bedrooms, garage or not, size of garden, number of toilets, distance from a school or train station and so on based on historical data.

The debug tool

The debug tool is very useful but not essential. You activate it by calling it `true` and deactivate it by calling it `false` or not having it at all. It tells you how well the training is going on each epoch, it is a measure of the loss (error, cost or difference between its predict output and the real output). There is a hide button which if you click on it will remove it when finished.

Also if you move your mouse over the chart you will get the actual value of the loss at any point on the chart.

Figure 35 debug and loss function



Layer Name	Output Shape	# Of Params	Trainable
dense_Dense1	[batch,16]	32	true
dense_Dense2	[batch,1]	17	true

The Datasets

Mostly we will be using what can be called synthetic data which means it is not real data, it is artificially created. Synthetic data is also used in addition to real data when training models especially in the case where there is too little data or it is too expensive to collect and label the data. It might sound like cheating but done well is very effective and cost efficient.

What you want is as much real data as possible, if it is images then tens of thousands will be needed for the model to be able to train on them with some accuracy. This as you can imagine, is very expensive as each image needs to labelled, a very laborious task. People are employed to do just that and other similar data preparations.

It can be easy to just focus on the neural network architecture and the training phase but the quality as well as the quantity of data is just as important. For instance, if you are doing facial recognition, but all your data is based on white, male, middle aged faces the model will have an inherent bias which may not be obvious until it is deployed at a later date when it struggles with non-white or female faces (this did happen). This was also true of accents in voice recognition software in the early days.

The trouble is identifying potential issues when there are tens of thousands bits of data, so just be aware and don't take the data for granted, check it out, test the model well before deploying it or using it for real.

The callback() function

Even though we haven't started coding yet I want to mention the callback function. When we run a particular function in `ml5.js` we often incorporate what is called a callback function. This can be simply to let us know when something has finished or it could carry data such as the predicted results. Callbacks can also be used to check for errors. We will use them quite frequently even though they are often optional. Their job is to call a function inside another function.

The Hyperparameters

These are the parameters that you can fiddle around with to get the best results. Although we can use the default settings and they will probably be OK for some situations, they are unlikely to offer the best solution. Knowing what each hyperparameter does is a skill you will need to develop. Experience will help you develop those skills, another reason for this tutorial, to get a feel for them. The main hyperparameters you will be using are listed below. If you were a data scientist then you would have even more hyperparameters to play with, but you would also have huge messy datasets but, thankfully, we will be using simple and relatively small datasets.

Training Hyperparameters

- Epochs
- Batch Size

Model Architecture Hyperparameters

- Hidden Layers
- Nodes (Neurons)
- Activation functions

I will give you a brief description for each of these hyperparameters but they will become more apparent when you actually start changing them. There is an element of trial and error but after a while you get a feel for what works well. Usually the ml5.js default settings aren't far off the mark. All the above functions are options available during either the training phase or when initially designing the neural network.

Epochs

The epoch is usually the first `hyperparameter` you change and it is the easiest to understand. In machine learning, an `epoch` refers to one complete pass through the entire training dataset. It's a fundamental concept in training neural networks. Here's a breakdown of what happens during an `epoch`:

1️⃣ The entire training dataset is fed into the model. This dataset contains the examples the model will learn from.
2️⃣ The model processes each data point and updates its internal `parameters` based on the errors it makes in its predictions. These `parameters` (called the `weights`) are like the knobs and dials of the model that determine its behaviour.
3️⃣ Once all data points have been processed, one `epoch` is complete.

It's important to note that multiple epochs are typically needed to train a model effectively. With each `epoch`, the model gets better at recognising patterns and making accurate predictions. However, there's a sweet spot: too few epochs and the model won't learn enough, while too many epochs can lead to overfitting, this is where the model memorises the training data too well and performs poorly on unseen data.

The number of epochs is called a `hyperparameter`, meaning it's a setting that needs to be tuned to find the best performance. The default is 10, we could increase it to say 250 or more but that may be a waste of time as there is very little learning after a certain point. That is why the loss function chart is so helpful, as you can see where the training is giving diminishing returns.

The size of the batch

This is the second of the hyperparameters that we will tackle and it is an important one even if it seems a bit nebulous.

Batch size is a term used in machine learning to describe the number of samples that are used to update the weights of a machine learning model in one iteration. For example, if you have a dataset of 100 samples and you set the batch size to 10, the model will update the weights after processing 10 samples. The model will repeat this process until all the samples in the dataset are processed.

This is another hyperparameter that you can change. Through a bit of trial and error you will get a feel for which batch size yields the best results for a specific timeframe. You could send it all through in one go (an epoch size) or send one through at a time, but the ideal will be a compromise somewhere in between. Smaller batch sizes also requires less memory as it doesn't have to process a large amount of data each time.

The default batch size for ml5.js (as far as I can gather) is 32. Choosing the batch size is trial and error. Although you can specify any size it is usual to try values of: 8, 16, 32, 64, 128, 256, etc.

The hidden layers

The hidden layers are sandwiched between the input layer and the output layer. It is another hyperparameter which you can command. The number of hidden layers has a profound impact, however, I would caution against going overboard with lots of layers to start with, it can slow the training because of the number of calculations.

This is where much of the magic happens, though actually no-one is sure what actually is happening here which may seem a bit strange. This has been one of the issues surrounding machine learning, it is the black box syndrome. A neural network has all those layers and nodes, with randomly generated weights, tweaking them as it learns from the data provided, this part is a bit of a mystery. The machine sets all these weights and there can be thousands of them all working together.

Each hidden layer extracts features from the data passed through it. As it passes through all the layers more and more complex features are identified. A bit like a series of filters each one building on the previous, it filters out irrelevant information. This is what makes neural networks so powerful.

Having multiple hidden layers means it can tackle non-linear problems. These are the more complex patterns and problems that aren't simple linear (straight line) relationships. The world is rarely linear.

An example might be recognising cats or dogs. The first layer will look for corners, edges (e.g. tails, ears, whiskers) or colours. The next layer will identify patterns in the coats and the next will find other subtle differences. Some of these features will not be obvious to us until it finally makes the prediction if it is a cat or a dog in the final output layer.

In ml5.js the default is one hidden layer, but as you will see we can (and will) add more hidden layers. But more, as you will find out, is not always better. Next we can decide how many nodes we want for each hidden layer.

How many nodes (neurons)?

The input layer, to be precise, isn't usually called a layer. The number of `nodes`, (`neurons`) is fixed by the input data, and this also true for the output layer. The hidden layer(s) can have as many `nodes` as you want to give them. So, surely the more the merrier, maybe, we will see.

The default is one hidden layer and it is given `16 nodes` (`neurons` or `units` to use their jargon). This is a very good starting point. You can give it fewer or more. If you have more than one hidden layer you don't even need to have the same number of `nodes` in each. Again it is something to play around with to get a feel for what works, or doesn't work or seem to make little or no difference.

Each `node` has an `activation function` and an extra `node` is added called the `bias node`. The `bias node` just makes sure that there is always a value being introduced so that it never collapses to zero which cannot be trained. This is created automatically but I thought it was worth mentioning.

Activation functions

There are quite a few activation functions. But there are two common functions used and we will explore them to start with. There are many more that can, and are used. Those who work in research are exploring ways to make neural networks better, more efficient. They explore different architectures and functions to achieve better results, but we will keep it simple for this tutorial.

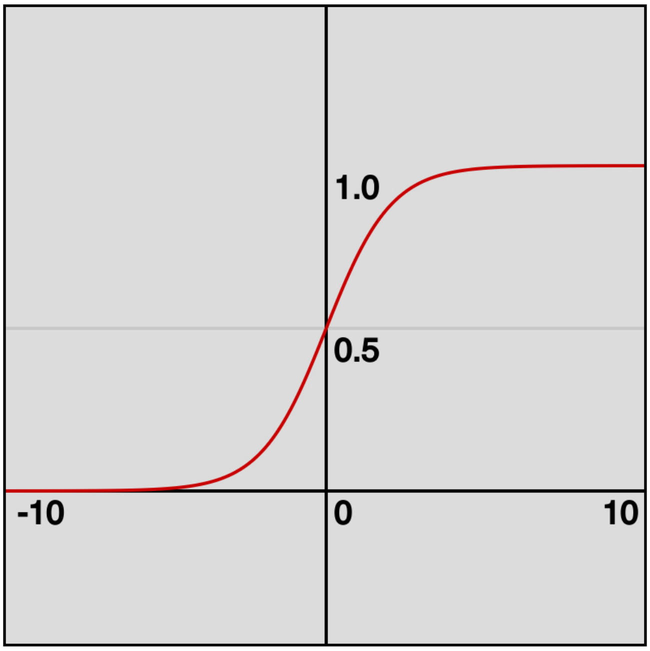
The default activation function in the hidden layer nodes is what is called the ReLU function. In the recent past the Sigmoid function was the popular one but that was costly in terms of calculation and was a poor performer compared to the ReLU function.

The Sigmoid activation function

To start with we will look at the `Sigmoid` function. It inputs any value (x axis) and the output will be between 0 and 1, however you can see that any values close to 5 are effectively 1 and -5 effectively 0.

This is great and works well but it is computationally heavy, it takes a lot of working out and if you have thousands of nodes each with their own sigmoid activation functions then this can take the training a long time. If you want outputs between -1 and 1 then you can use a variation of the `sigmoid` called `tanh`.

Sigmoid



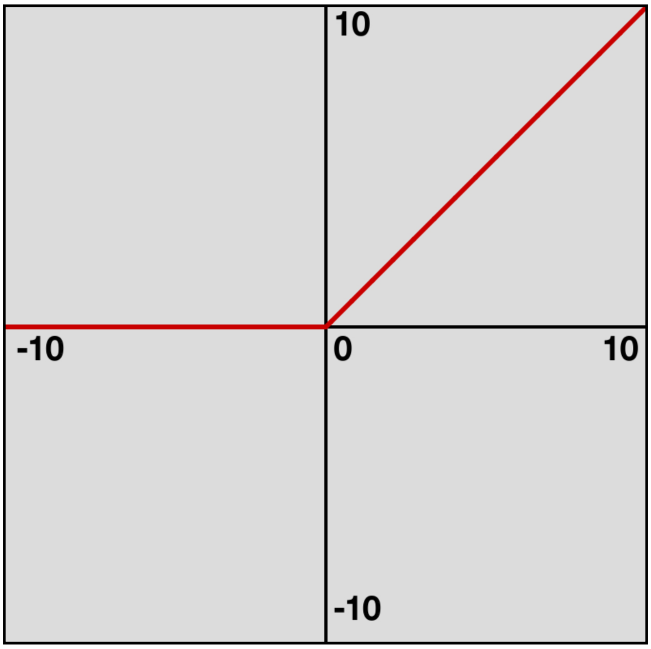
The ReLU activation function

To improve the efficiency of neural networks they looked at alternatives to the Sigmoid function that worked better and more efficiently. They came up with the ReLU function which looks almost too simple for it to work well and yet it does. This is especially true in a deep neural network (DNN) which has many hidden layers and nodes.

It work very efficiently and yields excellent results. It works so simply: any x value less than zero the output is zero, any value greater than zero the output is the same as the input.

A problem can arise sometimes if the input is zero exactly. Getting a zero output is not ideal and can cause problems for the network hence there are variations on this, one is called leakyReLU which is quite similar.

ReLU



The ml5.js functions

For this tutorial we call our model nn for no other reason than for simplicity. We have functions we can call on to perform a particular action. The format is the name of the model, in this case nn, followed by a dot (.) and the function we are calling. Inside the brackets is the information or data we are actioning and also where we might put a `callback()` function

nn.addData()

We use this to add the data to the neural network

nn.normalizeData()

This normalises the data taking it from a wide range of values and squashing them, in proportion between 0 and 1

nn.train()

We call this function when we are ready to train the model on the data added. Here we can include options for number of epochs and the batch size.

nn.predict()

Once it is trained we can put in some data to predict the outcome, this is used with regression tasks. It uses sigmoid and then scales up the predicted value.

nn.classify()

This is used to make the classification prediction, it uses the softmax function (to pick highest probability).

nn.save()

We can save the model once we have trained it and tested it. It is saved in three files. These files will contain the necessary information (hyperparameter settings and weights) to run it again.

🀄 nn.load()

This will load the three files that make up the model and make it available to predict or classify.

Overfitting and underfitting

Overfitting is where the model performs well in the training but fails to generalise the learning when you use new data. It has learned the correlation between the inputs and outputs too well. It has memorised the data not recognised its patterns.

Underfitting on the other hand is the opposite. It becomes obvious when the model is too simple and cannot create a relationship between the input and the output and so hasn't recognised any strong patterns.

This is why you usually have your data split three ways. Whereby you use 80% for training, 10% for validation where you fine tune your hyperparameters. Then the remaining 10% is where you test the model to check if there is any overfitting or underfitting. For the tutorial we will be using training data only and then predicting. This is for simplicity.

In the real world of machine learning models you would have the three datasets, also it is important that for all three data sets that they are representative of the whole data, to do that you take a random sample and not just creamed off the last 10% or 20% of the total dataset. This is important that the three datasets are equally representative of the total dataset.