Machine Learning Hackathon presentation notes

Machine learning is one of the most powerful tools in computing so far. I wanted to do a session on ML because the School of Code doesn’t do anything about it, and it may well be good for our careers if we understand it. And you might be primarily a UI designer or a DevOps specialist, or some role that doesn’t ever expect to be doing the coding for making sense of huge data sets, but it would still be good to actually know what you will be asking of your colleagues on the back-end. I myself hope to be one of those people in the future.

Actually, it is one of the most controversial applications of computing, and its pretty scary what you can do with it. You can use it as a tool of control, so the Chinese government have been able to find patterns in their population’s communication with ML. They can adopt attitudes widespread in the population as official policy when it suits, or rapidly mute any social media protests. Democracy is replaced with algorithms. In the West, our militaries have more restrained versions of this, but our social media companies and streaming services use data to keep us hooked on their products and therefore their adverts. I have deliberately not put a picture of the time that Google Photos’ auto tag feature used poor datasets to label some black people as gorillas, but I think all of us feel a bit patronised by those ‘CAPTCHA’ verification systems.

But ML also has some really good uses too. Science often has problems where we need to find interesting things but there are too many different chemicals to choose from, finding a vaccine for example. The Janssen Covid-19 vaccine was found using a machine learning algorithm. We can predict the weather and earthquakes, this is from a paper using a random forest technique, link https://arxiv.org/ftp/arxiv/papers/1702/1702.05774.pdf.. I remember when I was learning to drive, that reversing was pretty annoying to get right, but there are now cars that can use live feature recognition to do the manoeuvres for you.

I imagine that the companies you work for may well be using machine learning for some other purpose. It all boils down to define suitable boundaries between sets of data points.

Now there are two classes of ML problems – one is when you know what the right answers are, and you use those to train the network to draw a boundary based on them. This is a supervised learning model and we are doing those today. An unsupervised model doesn’t have the right answers supplied, so we have to group the data points by some other method, like relation to cluster centres. I might do another hackathon on unsupervised learning if this one goes well.

We can represent a neural network using this diagram with weights and nodes and you multiply the weights by the inputs to get the output nodes.

Actually forget that, I am going to go for a weird analogy now. Imagine you have this robot taking balls from white buckets to black buckets, but you need to figure out how many balls the robot has to move from each bucket and which bucket to move them to.

The first task is to make a neural network from scratch, and if it sounds like reinventing the wheel, well you’re not wrong, but it does give you an appreciation of what is happening when you are tweaking layers and settings in the more sophisticated ML frameworks. If we work with data in 2D, we can make pretty graphs.

- Why do we want graphs for these tasks? It is because machine learning is essentially about getting your code to define suitable boundaries between sets of data points.

- Humans find this difficult when we look at big arrays of numbers, but we find it much easier when we have a picture to look at. So if your code is not doing what you want it to, it should be easy to spot on the graph.

- I have tried to put in the code that makes the Plotly graphing library work. If you wish to display multiple graphs or play around a bit more, then I encourage you to take a look at the [Plotly docs](https://plotly.com/javascript/).

Now we have a network, but we need to make it learn. What kind of ways do you think we can get our robot to get even better at moving the right balls to the right buckets? Given that you have the ‘right’ number of balls to end up at each output bucket?

Also introduce biases here, as the number of balls starting in the output/hidden buckets

We would compare the true values with what we actually end up with. And then we can change the weights, our robot instructions, to get it a little bit closer to the true values. This is where gradient descent comes in, because you need to have some method that can get closer to the true values only by changing the weights. Now there is a quick way of doing this, at least for the simple networks, by doing calculus. But not everyone wants to learn maths to do programming, so we can do a more intuitive version involving slopes of hills, well gradients.

* Analogy of going down a hill when lost – like ‘round the moors where I live, if you have no idea where you are, then it is good to go downhill, because you are likely to reach a river, and river valleys are where people live. But if you are not careful, you could end up in a bog instead and have to change your strategy.’
* Then I can point them to a graph and say that you can find which way is downhill by moving a small amount and then finding how height has changed. And if you have gone up, then you determine not to go up so give the gradient a negative sign, but if you go down, then you give it a positive sign, and if it is steep, you can assume there is much more to go down so you take a bigger step.
* And then use a line graph and pointer on zoom to show how you can get stuck in local minima – by changing the learn rate, an art rather than a science at this point. The global minimum is what you are aiming for.

Then, I need to go through the concept of learning in small batches – the idea is that it helps avoid the local minima and saddle points.

I can start to put all this into equations and pseudo-code, like this:

* Initialise weights and network
* Find the estimated outputs
  + Get hidden layer by w\_1i\*x\_1+w\_2i\*x\_2+…+b\_i
  + Apply activation function to bring it into a sensible range
  + Get output layer by m\_1j\*y\_1+…+c\_j where y are the activated hidden layer results.
  + Calculate gradient \Delta z\_j/\Delta m\_kj
  + Calculate cost function by m\_kj -\Delta z\_j/\Delta m\_kj \* m\_kj
  + Change the weights by the cost function
  + Including the biases and w weights as well
* Repeat for many epochs until learning is good enough.

Warn participants against overfitting, perhaps using the data we have and the potential for the learning process to put too much detail into the boundary, making loops where none are required. However, we can assure them that with more data in the relevant areas, it is unlikely that there will be overfitting. The surprising logic of statistics is quite beneficial here.

I could break it up into smaller tasks to refactor code – this is better practice and can help people understand what they are doing better.

* We can start with a one input, one output, one weight
* Then go to two inputs, four weights, two outputs
* Then add biases
* Then add activation function
* Then use sigmoid activation
* Then add hidden layer – this is going to be the big step

The second task is using the TensorFlow framework for JavaScript (or TFJS) to make a more complicated learning model, to classify some blurry images. This one I picked out because I have sort-of done it before in Python, and given that machine learning engineers don’t normally use JavaScript for this stuff, I hope it has some transferable syntax to TensorFlow versions in other languages. Some concepts may be familiar from task 1, but the convolutional layers are new.

The thing with convolutional layers are that they split each image into little groups of pixels next to each other, and kind of makes a little neural network to learn from these. It means it can learn the borders and details that a more brute-force network might not pick up.

We can do a similar thing with dropout layers and these are a bit easier to understand, because it just tells the network to ignore a percentage of nodes each time it runs, and hence ignores a load of weights. A weight mislead by the data doesn’t have as much effect that way.