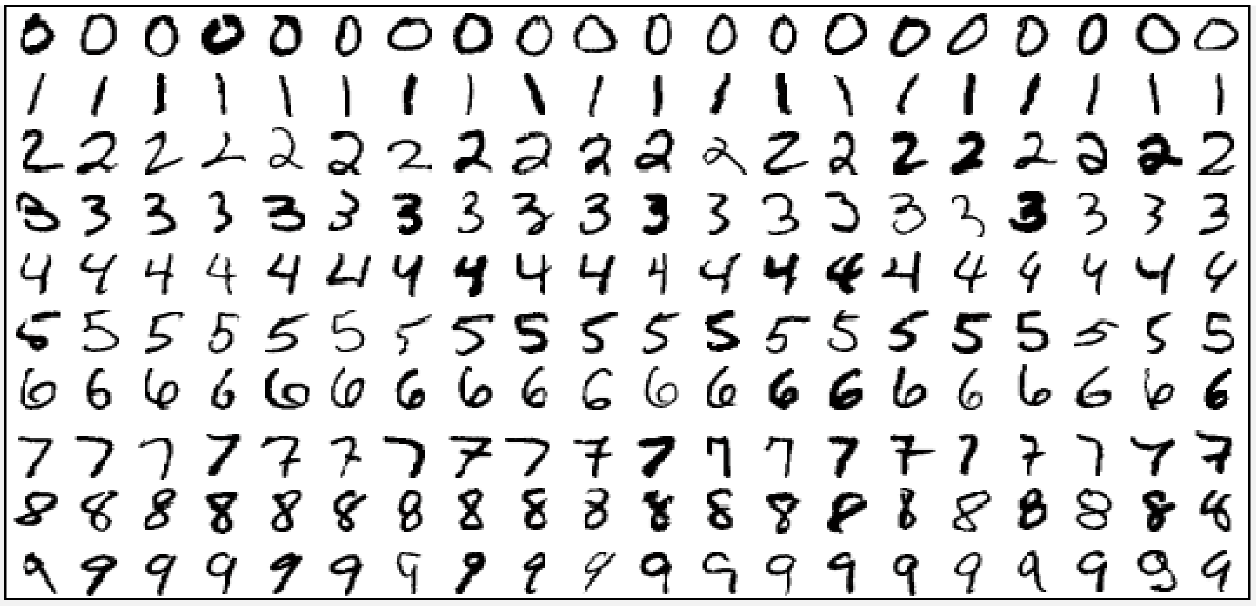
In my previous post I outlined how machine learning works by demonstrating the central role that cost functions and gradient descent play in the learning process. This post builds on these concepts by exploring how neural networks and deep learning work. This post is light on explanation and heavy on code. The reason for this is that I cannot think of any way to elucidate the internal workings of a neural network more clearly that the incredible videos put together by three blue one brown — see the full playlist here.

These videos show how neural networks can be fed raw data — such as images of digits — and can output labels for these images with amazing accuracy. The videos highlight the underlying mathematics of neural networks in a very accessible way, meaning even those without a heavy math background can begin to understand what goes on underneath the hood in deep learning.



This post is intended as a “code-along” supplement to these videos (full Tensorflow and Keras scripts are available at the end of the post). The purpose is to demonstrate how a neural network can be defined and executed in Tensorflow such that it can identify digits such as those shown above.

TensorFlow (for those who do not know) is Google’s deep learning library and while it is quite low-level (I typically use the higher-level Keras library for my deep learning projects), I think it is a great way to learn. This is simply because, although it does incredible amounts of magical things behind the scenes, it requires you (yes you!) to explicitly define the architecture of the NN. In doing so you’ll gain a better understanding of how these networks work.

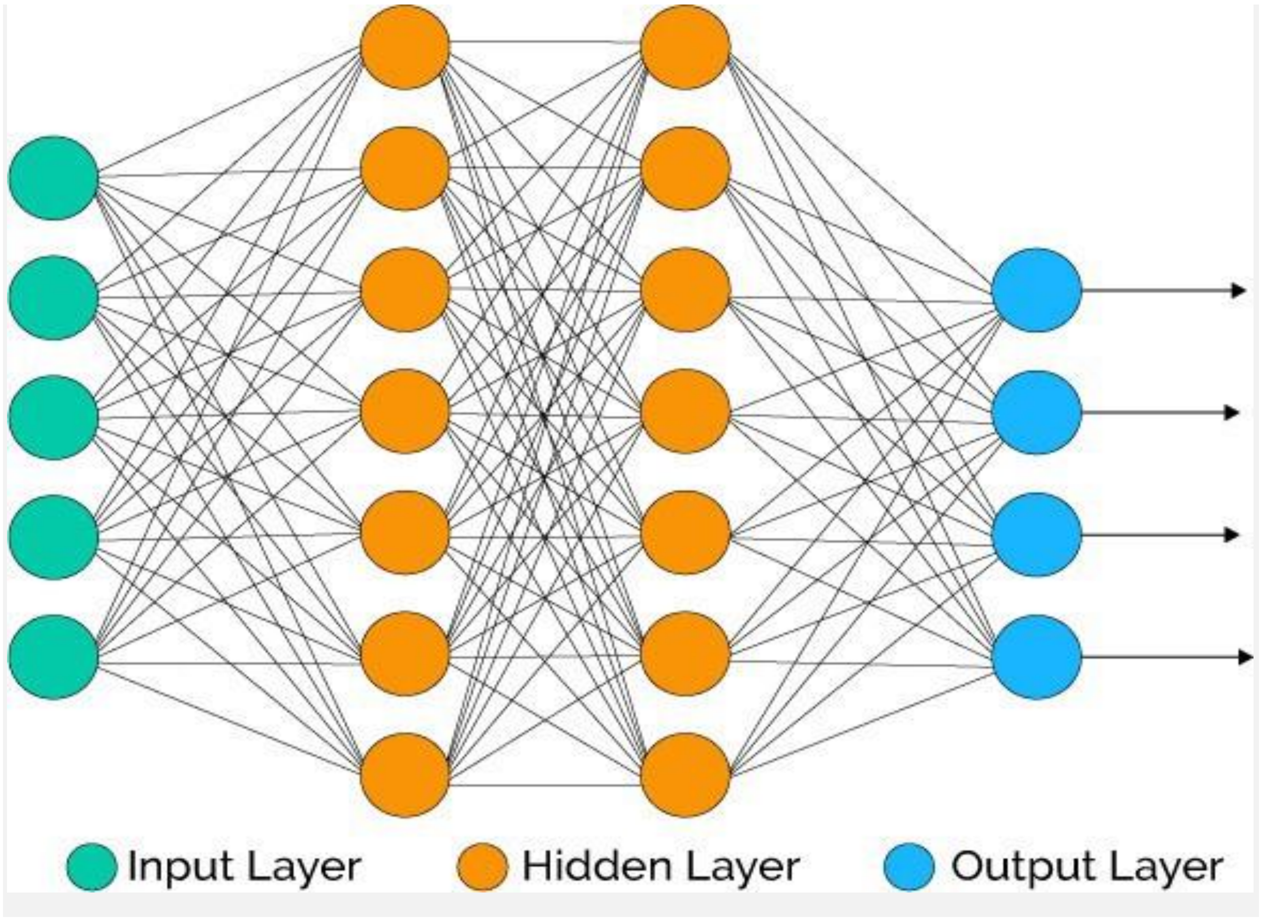
Neural Networks

Neural networks are mathematical and computational abstractions of biological processes that take place in the brain. Specifically, they loosely mimic the “firing” of interconnected neutrons in response to stimuli — such as new incoming information. I don’t find biological analogies particularly helpful for understanding neural networks, so I won’t continue down this path.

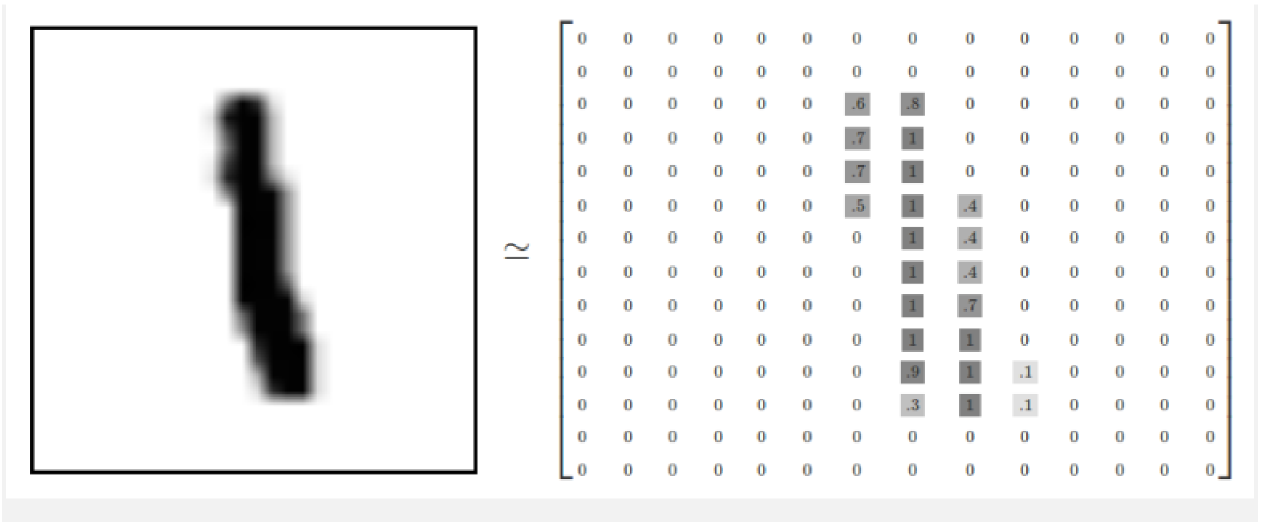
Neural networks work by computing weighted summations of input vectors that are then passed through non-linear activation functions, thereby creating a mapping from input to output via a non-linear transformation layer. The weights (represented by neutrons) in the transformation, or *hidden*, layer are iteratively adjusted such that they come to represent relationships in the data that map inputs to outputs.

Defining Layers and activations

In the first step we define the architecture of our network. We will create a four layer network comprised of one input layer, two hidden layers and one output layer. Note how the output from one layer is the input to the next. This model is quite simple as far as neural nets go, it is comprised of dense *or* fully connected layers, but is still quite powerful.



The **input layer** — also sometimes referred to as the **visible layer** — is the layer of the model that represents the data in its raw form. For example, for the digit classification task the visible layer is represented by numbers corresponding to pixel values.



In TensorFlow *(all code is below)* we need to create a placeholder variable to represent this input data, we will also create a placeholder variable for the correct label corresponding to each input. This effectively sets up the training data — the *X* values and *y* labels we will use to train the neural network.

The **hidden layer(s)** enable the neural network to create new representations of the input data that the model uses to learn complex and abstract relationships between the data and the labels. Each hidden layer is comprised of neurons, each representing a scalar value. This scalar value is used to compute a weighted sum of the input plus a bias (essentially y1 ~ wX + b) — creating a linear (or more specifically an affine) transformation.

In Tensorflow you have to explicitly define the variables for the weights and bias that comprise this layer. We do so by wrapping them in the *tf.Variable* function — these are wrapped as *variables* because the parameters will update as the model learns the weights and bias that best represent relationships in the data. We instantiate the weights with random values with very low variance, and fill the bias variable with zeros. We then define the matrix multiplication that takes place in the layer.

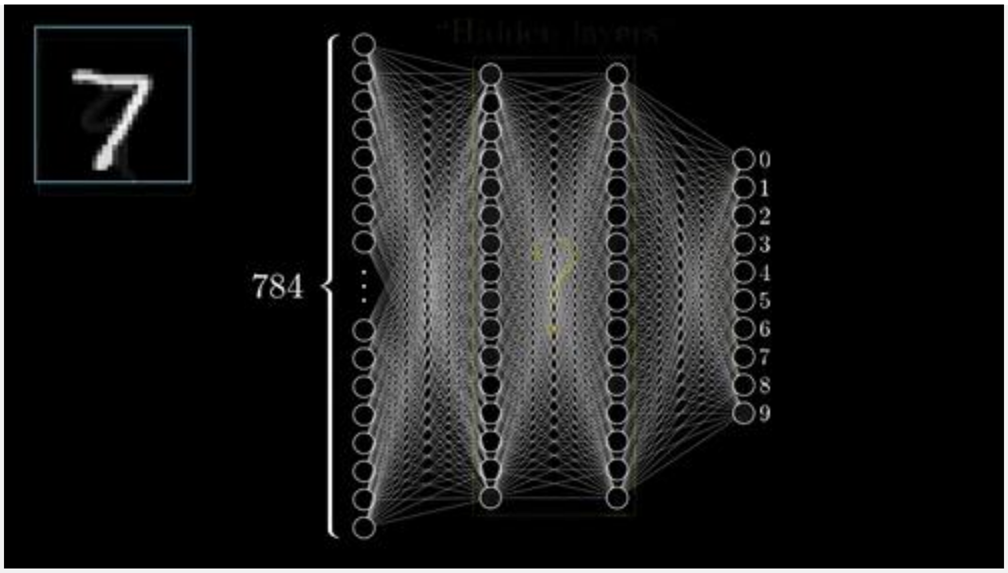
This transformation is then passed through an activation function, (here I am using **ReLU** or rectified linear units) to make make the output of the linear transformation non-linear. This allows the neural net to model complex *non-linear* relationships between input and output — check out Siraj Raval's excellent video explainer on activation functions here.

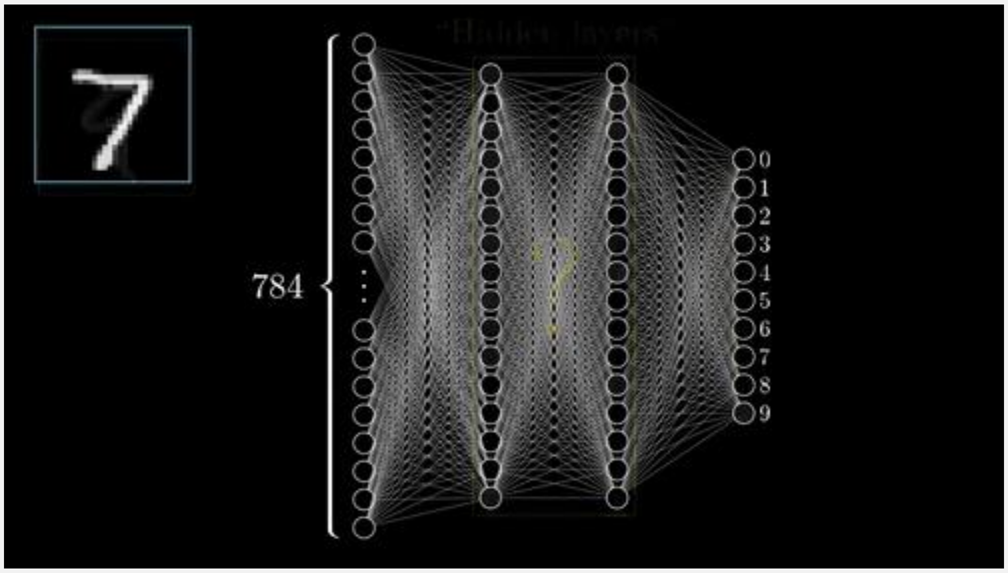
The **output layer** is the final layer in the model and, in this case, is size ten, one node for each label. We apply a **softmax activation** to this layer so that it outputs values between 0 and 1 across the final layer nodes — representing probabilities across the labels.

Cost function and optimisation

Now that the neural net architecture is defined, we set the **cost function** and **optimiser**. For this task I use categorical cross entropy. I also define an accuracy measure that can be used to evaluate the model's performance. Finally, I set the optimiser as stochastic gradient descent and call its minimise method once it is instantiated.

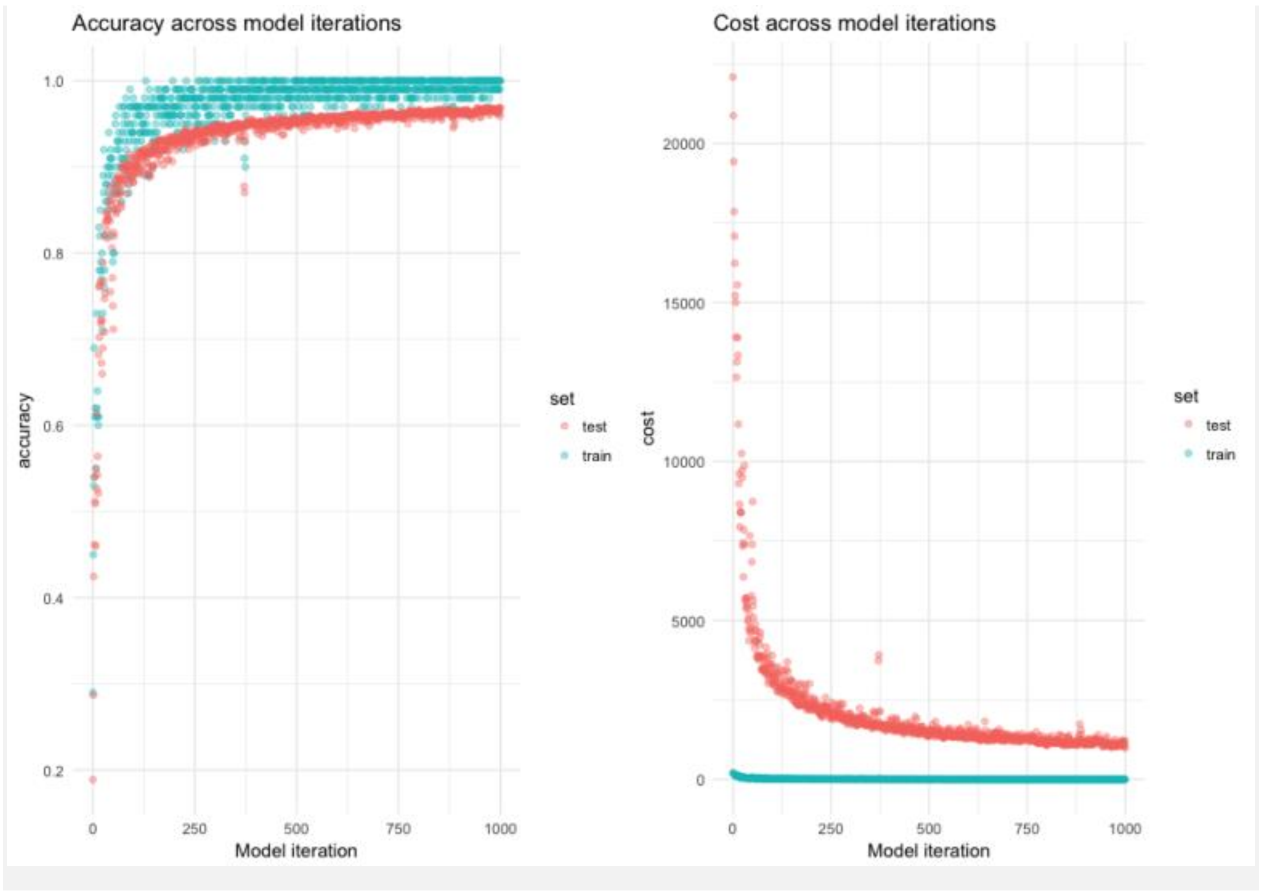
Finally, the model can be run — here 1000 iterations are run. On each iteration a mini-batch of the data is fed to the model, it makes predictions, computes the loss and through backpropogation, updates the weights and repeats the process.





from three blue one brown's "But, what is a neural network?" video

This simple(ish) model gets to around **95.5% accuracy** on the test set, which isn't too bad, but could be a lot better. In the plots below you can see the accuracy and cost for each iteration of the model, one thing that clearly stands out is the discrepancy between the performance on the train set and performance on the test set.



This is indicative of **overfitting** — that is, the model is learning the training data too well, which is limiting its **generalisability**. We can handle overfitting using **regularisation methods**, which will be the subject of my next post.