Portfolio project: food label scanning via machine learning

Dr. Terence Vockerodt

August 2021

Contents

Part I

Introduction

1.1 Background

This is a portfolio project dedicated to addressing the problem of scanning labels and ingredients lists in order to determine whether or not a food product is vegan, which is of significance to me as I try to eat a more plant-based diet in my remaining years.

This project will mainly focus on using machine learning to solve the problems, with other more sensible methods used only in cases of major convenience. This is because the aim of this project is to expand my knowledge of machine learning algorithms specifically, as well as give me a practical application of my machine learning training.

1.2 Rough algorithm design

As is the beauty with many machine learning projects, a good starting point is to consider what a human would do to solve the problem, and then convert that into an algorithm for the computer to use. When I encounter a label, I look for the following things in order:

1. Look at the front of the product or the name to see if the word 'vegan' appears.
2. Look for a logo or symbol or message indicating that this product is 'vegan society approved' or along the same ilk. This is known as 'The Vegan Trademark' [1]
3. Look at the ingredients. If no ingredients are non-vegan, then the product is vegan.

Overall, looking at the ingredients will always be able to determine if a product is vegan or not. This means that at some level, there should be a machine learning algorithm that looks at the list of ingredients. However, we may be able to save time by looking at the logos first for the vegan trademark [1], which can use a different machine learning setup in order to look at the patterns of pixels. This may be more numerically economical, because typically the ingredients list is a much smaller font size than the vegan trademark, and therefore we can divide the original ingredients image into larger sub-images, which is much easier to do and to iterate over than trying to identify small text characters which may need a much smaller discretisation scheme.

1.3 Preliminary stages

There are several interesting avenues to consider for a project like this. Some first initial steps would be to design the following algorithms:

1. An algorithm which takes in an image of an ingredients list, and translates that into text (via machine learning). It then parses the text (using regex on a database of non-vegan ingredients, for instance) to see if any ingredients are non-vegan.
2. An algorithm which looks at various sub-images of a food label image, and determines whether or not a vegan trademark is contained in that image.

Whilst the latter step seems reasonably close to the ideal final algorithm we need, the former is very contrived, since it requires an image of the ingredients list specifically to be generated. However, writing an algorithm that generates a list of ingredients from a label is something that can be addressed at a later date, since we need more accessible problems to deal with first. Therefore, we will start by looking at reading in an ingredients list. Both will be attempted at some point. If the project goes far, then they can be combined.

1.4 Language choice

For this project, I am partial to using python, because it will tie in nicely with a course I am currently taking regarding machine learning in python. In my Stanford course, I used MATLAB/Octave to program the machine learning, so I already have some experience there.

Part II

Theory of artificial neural networks

2.1 What is an artificial neural network?

From Arthur Samuel (1959), machine learning is the field of study that gives computers the ability to learn without being explicitly programmed [2]. This is useful in many fields where there is not great intuition on what the relationship between certain data sets is, such as nuclear physics, where the exact quantitative nature of the strong nuclear force has not been explicitly calculated with undisputed accuracy, nor can it be analytically derived due to the many body nature of nuclear physics.

One of the major algorithms under the wing of machine learning is the artificial neural network (ANN), which aims to model the neurons in the brain [2]. The typical structure of an ANN consist of several layers. The first is the input layers, which consists of input nodes that the ANN uses to calculate the outputs of interest [2]. In machine learning language, the features of the machine learning model are provided to the input nodes [2]. Them, there are a sequence of hidden layers, which carry information forward from the previous layers into the output layer [2]. Each hidden layer node calculates the activation function of a weighted sum of the nodes in the previous layer that are connected to it, with the weighting of each node being variables of the model [2]. Eventually, this leads to the output layer [2].

During training, the output layer is checked against the result that it should be, which requires a training data set [2]. An algorithm known as back-propagation is then used to correct all of the weights in the network such that the network can generate the correct output next time it is shown said input [2].

2.2 What do the hidden layers do?

To answer this question, I will be using the logic from reference [3]. In the video, the ANN is applied to the problem of recognising digits, but the logic is similar for recognising characters and other edge detection problems. When we see certain characters, we know what they are because of the distinct lines and curves that differ them from one another. The letter 'A' can be thought of as two slanted lines, joined together by a horizontal smaller line. This is similar to the letter 'H', except the two bigger lines in H are no longer slanted, but vertical. The letter 'O' has no horizontal or vertical or slanted lines in the same way that A and H do, but is a continuous curve.

The hope is that the hidden layers will learn to recognise these sub-components like horizontal and vertical lines, and then activate/fire if that sub-component is present. For example, if a hidden layer detects a line slanted to the right, joined at the top to a line slanted to the left, and a horizontal line between them, then the most likely output of the ANN is an 'A'. This 'detection' layer can also be broken down into smaller layers that detect smaller components of these sub-components. Fig. 2.1 is a diagram that showcases this logic.

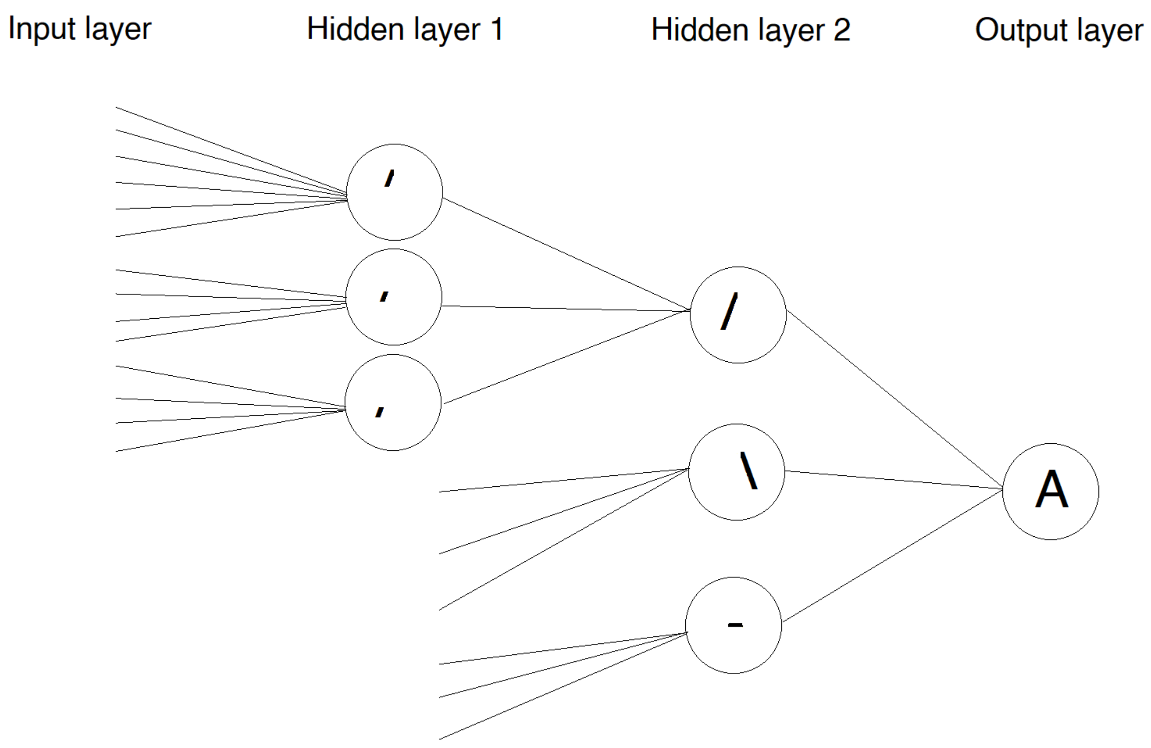

Figure 2.1: Diagram showing what the hidden layers may be trained to become in an edge detection problem. When all three neurons shown in the second hidden layer fire, then the character is the letter ‘A’. The first hidden layer breaks the task down into smaller sub-tasks, such as detecting pieces or sub-components of a slanted line.

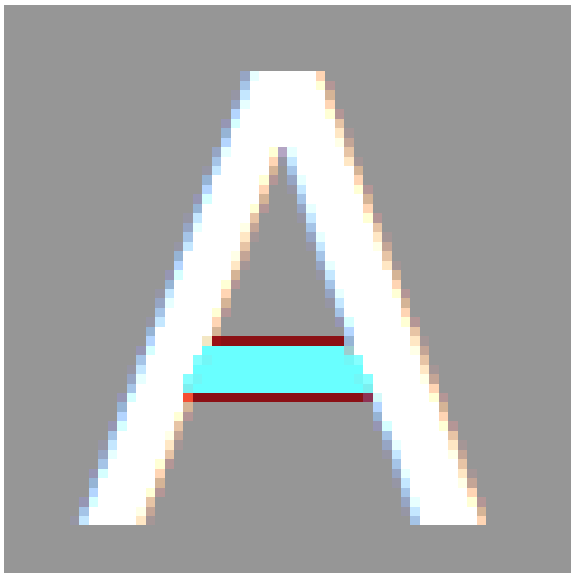

Figure 2.2: (Colour online) Diagram showing what the the weights may look like in an edge detection problem. Positive weights are shown in cyan, negative in red. White pixels hold positive values and black pixels are zero. This setup of the initial weights can be used to detect the middle of the letter ‘A’.

This at least is the hope of the ANN hidden layers, but one can qualitatively measure whether or not an ANN has this sort of behaviour by plotting the weights joining the input layer to the first hidden layer, after convergence is achieved. As explained in reference [3], for edge detection problems we expect weights to form bands, where the central band weights are positive, and the outer band weights are negative. Assuming that white pixels are positive in value and black are zero, when these weights ‘line up’ perfectly with a component of the letter that the ANN is learning, the white pixels are affected by the positive central band weights, and the result of the negative outer band weights on the black pixels will be zero. If instead the white pixels are larger than the band, then the negative outer band weights will conflict with the positive inner band weights and the node may not fire, which requires either more bands to be used or the initial band will become wider due to back-propagation. An example of the initial weights setup is shown in Fig. 2.2.

One can construct a neural network by considering what ‘tasks’ each node should be responsible for. Once I have boiled the problem down to a reasonably minimal set of sub-components, I can tune the initial weights feeding into said node to help it accomplish that task. For example, detecting the horizontal line in the letter ‘A’ will require a horizontal band of weights, centered around the center of the image (like in Fig. 2.2). We expect these weights to change over the learning period, but if every node is organised in this way then the time for convergence hopefully will decrease. Again, this can be investigated, quantified and reported on.

Part III

Mini-project: edge detection basics using ANN

3.1 Fundamental questions

Whilst a great appeal of ANN is the lack of explicit programming needed, it is interesting to me to see whether or not a more explicit approach to setting the initial weights can affect the outcome of the ANN. As an ex-scientist, I have several questions I would like to answer:

- How much time does it take for a more prepared/a custom network (a network with specific initial weights) to 'converge' (i.e: perform perfectly on the training set), compared to using a random initial seed of the weights?
- As the network learns, does a more prepared network qualitatively retain the structure of the node weights that were initially set?
- Does a randomised network eventually converge into the prepared/custom network? Are there other viable configurations? Can randomised networks be prone to better generalisation?
- Does the quality of convergence differ with a more prepared network on test sets?

To do this, I can construct two ANNs, one with random initial weights and another with custom initial weights. This can be done using the Keras library in python.

3.2 Reading in initial weights

This section is a summary of what is discussed in the Jupyter notebook entitled 'MP1: Reading in input weights.' Within the notebook, I look into using a GUI in order to draw out the initial weights. In the end, I settled on using MS Paint to draw effective images of the weights, and then created a python program to convert these pictures into arrays. This can be achieved by projecting the colour wheel onto a two-dimensional axis, or in other words: assigning a 2-D vector to each colour. This is shown in Fig. 3.1.

In the coordinate system in Fig. 3.1, I express the unit vectors for the red, green and blue colours as

$$\hat{\mathbf{R}} = \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \ \hat{\mathbf{G}} = \begin{bmatrix} \cos(300°) \\ \sin(300°) \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 1 \\ -\sqrt{3} \end{bmatrix}, \ \hat{\mathbf{B}} = \begin{bmatrix} \cos(60°) \\ \sin(60°) \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 1 \\ \sqrt{3} \end{bmatrix}, \tag{3.1}$$

where $\hat{\mathbf{R}}$, $\hat{\mathbf{G}}$ and $\hat{\mathbf{B}}$ are the unit vectors in the red, green and blue directions respectively. When we look at a pixel using python, it gives us a tuple containing the red,

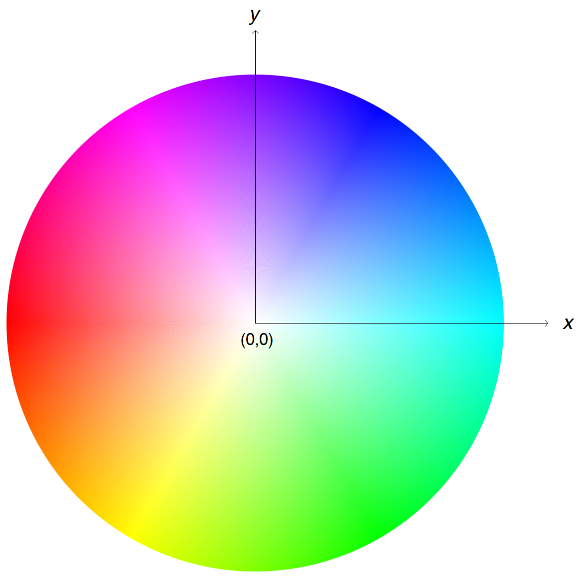

Figure 3.1: Colour wheel with the x axis aligned along the cyan direction. Black and white map to the origin (0,0).

green and blue content of the pixel, which just tells us the amount of the unit vectors $\hat{\mathbf{R}}$, $\hat{\mathbf{G}}$ and $\hat{\mathbf{B}}$ that we have in that pixel. If I label the pixel tuple as (R, G, B), then the colour vector $\mathbf{C}_{R,G,B}$ in our coordinate system can be expressed as

$$\mathbf{C}_{R,G,B} = \frac{R \cdot \hat{\mathbf{R}} + G \cdot \hat{\mathbf{G}} + B \cdot \hat{\mathbf{B}}}{255}. \tag{3.2}$$

The division by 255 ensures that cyan maps to $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$, and red to $\begin{bmatrix} -1 \\ 0 \end{bmatrix}$. Both white and black in this instance map to $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$. Since I only plan on using cyan and red to represent positive and negative weights respectively, I can take the x-component of the vector $\mathbf{C}_{R,G,B}$ as the value of the weight. Now, I can create images with a black background, with cyan to represent positive weights and red for negative, across my entire pixel grid, such as those in Fig. 2.2. To prevent the black background weights (which have measure 0) from not changing/learning during back-propagation, I set them to a small value in the x component of the colour vector is less than said small value.

3.3 Network setup

Unlike in the video of reference [3] and Fig. 2.1, I opted to use one hidden layer of size 92, which connects the 3600 size input layer of the pixels to the 62 size output layer corresponding to the English alphanumeric characters. I created 92 custom nodes, with unique weight patterns like those in Fig. 2.2. I wanted to use only one hidden layer, so that we could test properly if the structure of the custom node weights is maintained over training, since if I included other hidden layers then they may be responsible for most of the work of training and identifying the alphanumeric characters.

The activation function for nodes in the hidden layer was chosen to be a ReLU, and for the output layer a softmax activation was chosen. Due to having 62 possible one-hot outputs, I chose to use the categorical cross entropy as a loss function. The networks used were constructed using the Keras API.

3.4 Results

I show discrete histogram plots of the number of epochs needed for convergence of the custom and random networks in Fig. 3.2. In this instance, convergence is defined as the number of epochs needed for the train and test accuracies to both equal 1, which was a common occurrence. The average number of epochs needed for the custom network (13) is much less than the random network (28), and the variance of the data is much less also (stdev of 3 for the custom network, and 12 for the random one). There is a distribution for the custom network due to the random initial weights from the hidden layer to the output layer. This is solid evidence that our custom weights are out-performing the random weights in converging the network. For much larger data-sets, this is expected to save time in training the network.

When looking at the KDEs, it is clear that the custom network has not converged to a typical distribution (due to the peak at 22 epochs), and the random network KDE looks converged. One can take more runs to get a better idea of the probability distribution in this instance, but we expect it to be similar to a Maxwell-Bolztmann distribution [4]. This distribution is skewed such that observing values higher than the mean is more likely than observing values lower than the mean. This is sensible to assume for the random network. There are far fewer randomly generated configurations of the weights that will converge in a number of epochs less than the mean, than there are randomly generated configurations of the weights that will converge in a number of epochs more than the mean. As an example, using the mean of 28 for the random network, consider the likelihood of finding a network that converges in around 0 epochs, vs. a network that converges in around 56 (in a normal distribution, these would be the same). Since the custom network still has the weights from the hidden layer to the output layer randomised, then we expect that it would also follow a Maxwell-Boltzmann distribution (subject to more trials done), only with a much smaller mean and variance.

So what do the weights look like in custom and random networks? From Fig. 3.3, we can see that the custom weights mostly maintain their structure after training, with some of the weights developing patterns of the training data. However, these patterns are of much lower order of magnitude compared to the initial structure,

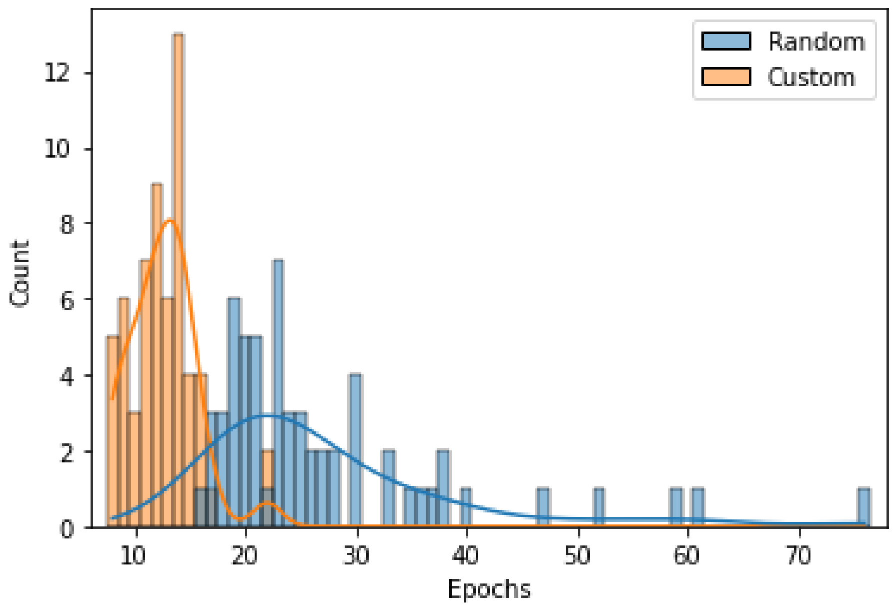

Figure 3.2: Comparison of discrete histogram plots (with KDEs as solid lines to guide the eye) of 60 runs of the number of epochs needed for convergence for the random and custom networks. In this instance, convergence is defined as the epoch when both the training and test accuracies both equal one. The random network has a higher average convergence time, and a larger variance compared to the custom network.

so they contribute to the activation less. For the randomly initialised network, we see two classes of node weights - ones which are structureless, characterised by lack of pattern, and a lower absolute magnitude of the weights, and the other with significant structure and a higher absolute magnitude of the weights. It would appear that the 92 nodes used are an excess of what we need for a random network, assuming that the structured nodes are the nodes responsible for most of the learning, and they effectively parallelise the function of more than one of the 92 custom nodes.

Since the random network appears to have parallelised the custom nodes, then a fairer comparison would be the convergence of the 92 node custom network, and a much smaller random network. Using 5 of the runs, I calculated the average number of structured nodes to be 26. However, when testing using only 26 nodes in the hidden layer of a randomly initialised network, the network failed to converge to 100% test and train accuracy within 100 epochs. This can mean that the unstructured nodes do have a purpose in training, or that only very specific initial conditions can converge the accuracies within 100 epochs for a small network, or both and beyond. I found that 64 nodes was able to start converging, and therefore did a final comparison between the 92 node custom network, and a 64 node random one. Since the number of nodes in the hidden layer are different now, we cannot look at the number of epochs as a metric for convergence time, since the epoch time is no longer the same. Therefore, I opted to use the time taken for training as a metric. Using the tf.keras.callbacks.EarlyStopping(monitor=‘accuracy’, patience=3) callback,

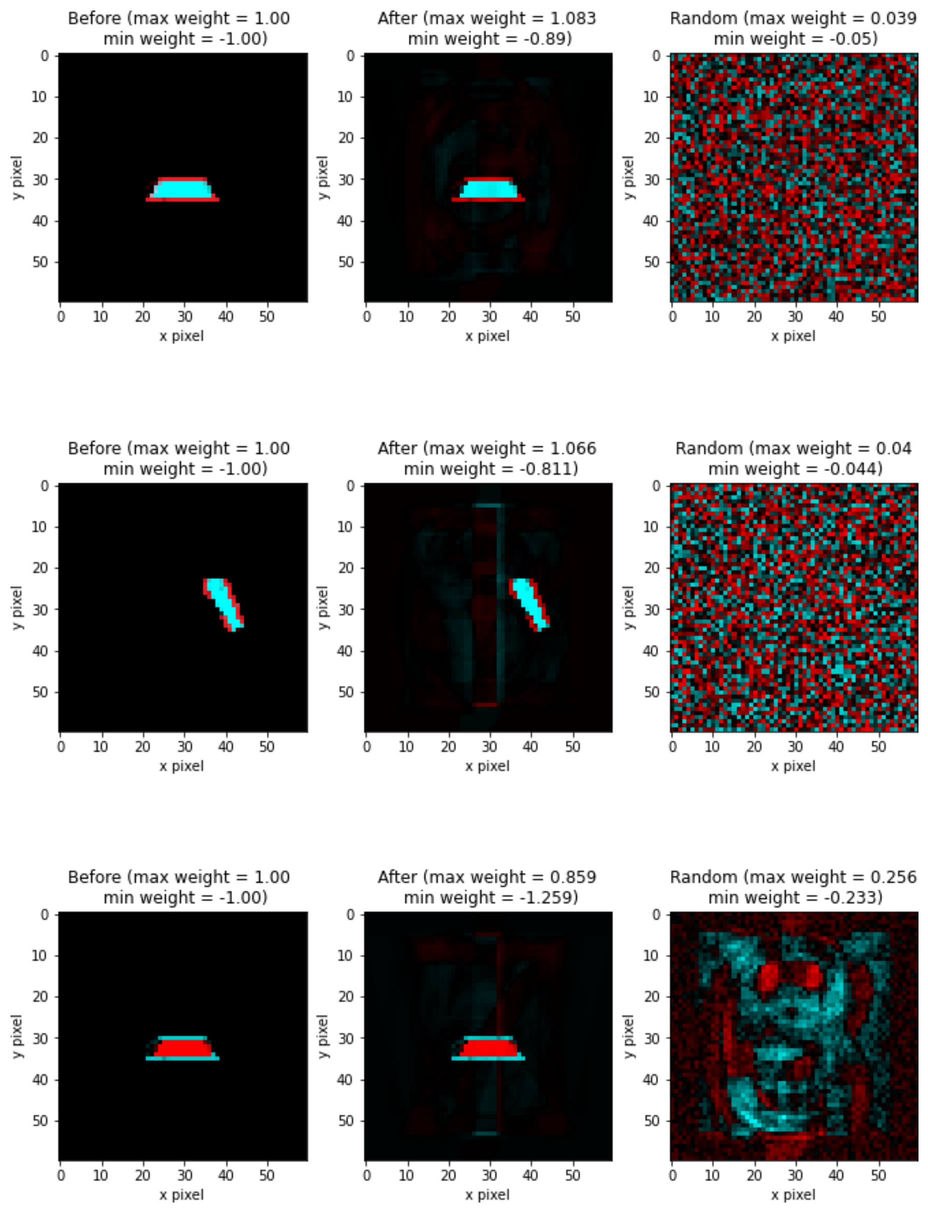

Figure 3.3: Showing custom initial weights before and after training, as well as random initial weights after training. For custom weights, the weight pattern was mostly unchanged (see top and bottom panels), with some weights gaining significant additional structure (middle). For the random weights, there are two classes of weights - structureless (top and middle) and structured (bottom). The colours have been scaled in each graph such that cyan corresponds to $\max(w_{n,\max}, |w_{n,\min}|)$, where w are the weights of the single node n in the hidden layer.

$t_{\text{setup,ran}}$ (s)	$t_{\text{conv,ran}}$ (s)	$t_{\text{total,ran}}$ (s)	$a_{\text{train, ran}}$	$a_{\text{test, ran}}$	$t_{\text{setup,cust}}$ (s)	$t_{\text{conv,cust}}$ (s)	$t_{\text{total,cust}}$ (s)	$a_{\text{train, cust}}$	$a_{\text{test, cust}}$
0.0156	4.3071	4.3227	0.9084	0.9061	2.4096	2.4237	4.8333	0.9986	1.0
0.0156	7.039	7.0546	0.9703	0.9607	2.3707	2.4376	4.8083	1.0	1.0
0.0156	6.5679	6.5835	0.9688	0.9626	2.3866	2.6224	5.0089	1.0	1.0
0.0156	5.6415	5.6571	0.9877	0.9866	2.3734	2.9761	5.3495	0.9995	1.0
0.0156	5.7983	5.8139	0.9991	1.0	2.4405	2.2691	4.7096	1.0	1.0

$\bar{t}_{\text{setup,ran}}$ (s)	$\bar{t}_{\text{conv,ran}}$ (s)	$\bar{t}_{\text{total,ran}}$ (s)	$\bar{a}_{\text{train, ran}}$	$\bar{a}_{\text{test, ran}}$	$\bar{t}_{\text{setup,cust}}$ (s)	$\bar{t}_{\text{conv,cust}}$ (s)	$\bar{t}_{\text{total,cust}}$ (s)	$\bar{a}_{\text{train, cust}}$	$\bar{a}_{\text{test, cust}}$
0.0181	5.3678	5.3859	0.9824	0.9814	2.4079	2.6142	5.0221	0.9993	0.9995

Table 3.1: Comparison between the convergence times and accuracies of the first 5 runs of 50 training runs of the 64 node random network, and the 92 node custom network. Averages are shown on the last row. Convergence in this instance is tested using the EarlyStopping Keras procedure described in the text.

I recorded the times taken for the early stop to be reached on training for both networks for 50 runs. Some of these are shown in Table 3.1, which also contains the averages. Whilst this callback does not mean that the network will converge to max accuracy every run, it gives us an indicator at least of how much time the network will take to reach perfect accuracy on our test and training data.

From Table 3.1, we can see that on average, the custom network trains faster, and produces more accurate results, even though we have decreased the number of nodes in the hidden layer for the random network. There is a small time cost for setting up the custom network of around 2.4 seconds, but this near constant time cost becomes insignificant if we are to train on larger data sets. The time cost for setting up the random network is negligible, and even then when comparing the average times taken to setup and train the network, the custom network still pulls ahead (although not by much). Again, if we have larger training sets, then the constant setup time associated with setting the initial weights will be less of a factor, and the enhanced training time from having custom initial weights will be worth it. The results suggest that if we can customise hidden layers intuitively then we ought to, as it can save precious time reaching the fully trained state, which is very important when considering scalability onto larger training sets.

3.5 Conclusion

I will now answer the questions that were outlined at the beginning of this mini project. Using my analysis, I determined that using custom weights improved convergence times for the training of the data on average. The custom weights were maintained through training, indicating some stability to the network structure. Said structure is not the only convergent structure of the ANN, and so the random networks in my runs never converged to the same as the custom network. With enough time, both networks can achieve 100% accuracy on training and validation data sets, but the custom weighted network gets there faster. This indicates that for future edge detection that I should use custom weights, in order to improve convergence times.

3.6 Next steps

There are a few problems with using the ANN with our custom weights to recognise letters. The custom network as it stands does not generalise well to transformations of our input letters, such as shears, rotations, and not even simple translations. In order to more accurately describe these types of phenomena, I will use a convolutional neural network (CNN) with custom filters.

Bibliography

[1] "The Vegan Trademark."
https://www.vegansociety.com/the-vegan-trademark.

[2] "Coursera: Machine Learning."
https://www.coursera.org/learn/machine-learning.

[3] "But what is a neural network? | Chapter 1, Deep learning."
https://www.youtube.com/watch?v=aircAruvnKk.

[4] "Maxwell–boltzmann distribution."
https://en.wikipedia.org/wiki/Maxwell-Boltzmann_distribution.