COMP9444 Assignment 1

z5257541 - Marcus Oates

Term 3 - 2021

Part 1: Japanese Character Recognition

1. The confusion matrix and accuracy for the implemented linear function `NetLin` are in Figure 1.

```
[[765.   5.   8.  13.  31.  67.   2.  61.  31.  17.]
 [  7. 669. 107.  17.  29.  22.  57.  15.  25.  52.]
 [  9.  62. 695.  25.  25.  22.  45.  36.  44.  37.]
 [  4.  38.  58. 758.  15.  54.  16.  17.  27.  13.]
 [ 59.  54.  76.  20. 625.  20.  33.  37.  20.  56.]
 [  8.  28. 127.  16.  20. 725.  27.   8.  32.   9.]
 [  5.  23. 145.  10.  25.  25. 723.  21.  11.  12.]
 [ 16.  29.  26.  13.  84.  17.  55. 620.  92.  48.]
 [ 10.  36.  97.  40.   8.  30.  45.   7. 705.  22.]
 [  8.  50.  88.   3.  52.  31.  20.  30.  41. 677.]]

Test set: Average loss: 1.0091, Accuracy: 6962/10000 (70%)
```

Figure 1: Confusion Matrix and Accuracy for `NetLin`

The number of independent parameters in `NetLin` are computed by,

$$\begin{aligned}
\text{weights per neuron (out)} &= 1 + \text{number of neurons (in)} \\
&= 1 + 28 \times 28 \\
&= 785 \\
\text{total independent parameters} &= \text{number of neurons (out)} \times \text{weights per neuron (out)} \\
&= 10 \times 785 \\
&= 7850.
\end{aligned}$$

2. The confusion matrix and accuracy for the implemented fully connected 2-layer network `NetFull` are in Figure 2.

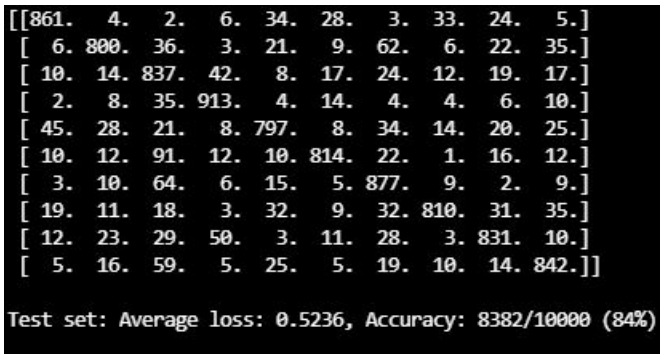


Figure 2: Confusion Matrix and Accuracy for `NetFull`

The number of independent parameters in `NetFull` are computed as follows,

$$\begin{aligned}
\text{weights per neuron (hid)} &= 1 + \text{number of neurons (in)} \\
&= 1 + 28 \times 28 \\
&= 785, \\
\text{independent parameters (hid1)} &= \text{number of neurons (hid1)} \times \text{weights per neuron (hid1)} \\
&= 120 \times 785 \\
&= 94200, \\
\text{weights per neuron (out)} &= 1 + \text{number of neurons (hid1)} \\
&= 1 + 120 \\
&= 121, \\
\text{independent parameters (out)} &= \text{number of neurons (out)} \times \text{weights per neuron (out)} \\
&= 10 \times 121 \\
&= 1210, \\
\text{total independent parameters} &= \text{independent parameters (hid)} + \text{independent parameters (out)} \\
&= 94200 + 1210 \\
&= 95410.
\end{aligned}$$

3. The confusion matrix and accuracy for the implemented convolution network `NetConv` are in Figure 3.

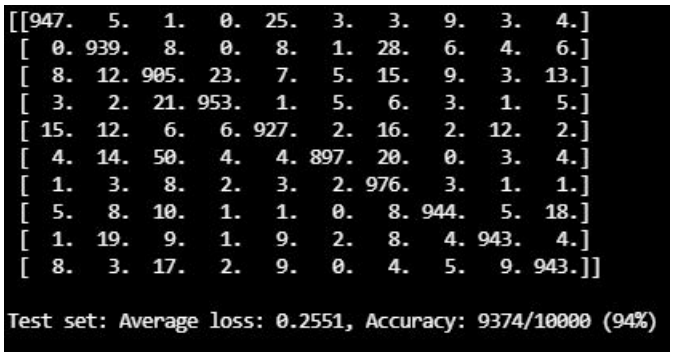


Figure 3: Confusion Matrix and Accuracy for `NetConv`

In order to calculate the number of independent parameters in `NetConv` the following variables are defined; images are of size $J \times K$, filters are of size $M \times N$, padding size is given by P, stride is given by S and L represents the number of channels in an image. These variables are used henceforth where appropriate.

$$\begin{aligned}
\text{weights per filter (conv1)} &= 1 + M_{\text{conv1}} \times N_{\text{conv1}} \times L_{\text{in}} \\
&= 1 + 5 \times 5 \times 1 \\
&= 26, \\
J_{\text{conv1}}, K_{\text{conv1}} &= 1 + (J_{\text{in}} + 2 \times P_{\text{conv1}} - M_{\text{conv1}})/S_{\text{conv1}} \\
&= 1 + (28 + 2 \times 2 - 5)/1 \\
&= 28, \\
\text{independent parameters (conv1)} &= \text{number of filters (conv1)} \times \text{weights per filter (conv1)} \\
&= 12 \times 26 \\
&= 312, \\
J_{\text{pool}} = K_{\text{pool}} &= J_{\text{conv1}}/2 \\
&= 28/2 \\
&= 14, \\
\text{weights per filter (conv2)} &= 1 + M_{\text{conv2}} \times N_{\text{conv2}} \times L_{\text{conv1}} \\
&= 1 + 5 \times 5 \times 12 \\
&= 301, \\
J_{\text{conv2}}, K_{\text{conv2}} &= 1 + (J_{\text{pool}} + 2 \times P_{\text{conv2}} - M_{\text{conv2}})/S_{\text{conv2}} \\
&= 1 + (14 + 2 \times 2 - 5)/1 \\
&= 14, \\
\text{independent parameters (conv2)} &= \text{number of filters (conv2)} \times \text{weights per filter (conv2)} \\
&= 24 \times 301
\end{aligned}$$

$$
\begin{aligned}
&= 7224, \\
\text{number of neurons (conv2)} &= \text{number of filters (conv2)} \times J_{\text{conv2}} \times K_{\text{conv2}} \\
&= 24 \times 14 \times 14 \\
&= 4704, \\
\text{weights per neuron (full)} &= 1 + \text{number of neurons (conv2)} \\
&= 1 + 4704 \\
&= 4705, \\
\text{independent parameters (full)} &= \text{number of neurons (full)} \times \text{weights per neuron (full)} \\
&= 140 \times 4705 \\
&= 658700, \\
\text{weights per neuron (out)} &= 1 + \text{number of neurons (full)} \\
&= 1 + 140 \\
&= 141, \\
\text{independent parameters (out)} &= \text{number of neurons (out)} \times \mathit{textweightsperneuron}(\mathit{out}) \\
&= 10 \times 141 \\
&= 1410, \\
\text{total independent parameters} &= \text{independent parameters (conv1)} + \text{independent parameters (conv2)} \\
&\quad + \text{independent parameters (full)} + \text{independent parameters (out)} \\
&= 312 + 7224 + 658700 + 1410 \\
&= 667646
\end{aligned}
$$

4. a. The linear function implemented in `NetLin` performs poorly, with an accuracy of approximately 70%. This means that the ten character labels are not linearly separable in the $\mathbb{R}^{28}$ input space, as the first layer, which for `NetLin` is the output layer, of a basic feed forward neural network attempts to separate the classes by learning a linear function [3], in this instance a hyper-plane, for each output neuron, with this neuron representing a particular character. In comparison with `NetFull`, which has an accuracy of 84%, the limitations of classifying the characters through a linear function are evident. The fully connected two layer neural network `NetFUll` increases the accuracy by a substantial margin as it is able to learn classification thresholds that are convex in the input space [1], allowing for greater complexity in the final learned decision boundaries between the ten classes. However, these two models learn classification based on the individual pixels of the image alone, whereas the convolution network `NetConv` samples the input through a filter, thereby allowing it to learn similar features between the characters that are then passed to a two layer fully connected network. This means that `NetConv` has the highest accuracy of the models, at 94%. While varying in size, each of the models builds on the previous, allowing for greater complexity in the decision boundary, and for `NetConv` more sophisticated input types, for the neural networks.

b. The neural network `NetLin` has the fewest independent parameters, at 7850, since it implements a simple linear function from input to output layers. Increasing the number of layers in `NetFull` and determining the optimal number of hidden neurons of 120 means that the number of independent parameters increases substantially to 95410, which is approximately 12 times the number in `NetLin`. However, while the convolution network adds two convolution layers, weight sharing between neurons that receive input from the same filter means that the number of independent parameters increases to 667646, which is only 7 times the number in `NetFull`. An increase in the number of independent parameters in a neural network may allow for greater accuracy to be obtained, as their are more weights that can be tuned to learn a more accurate decision boundary. However, at a point increasing the number of neurons in each layer or the number of filters, and the subsequent increase in the number of independent parameters does not add any more benefit to the models accuracy. Interestingly, the majority of the independent parameters of `NetConv` are those in the fully connected hidden layer, as all the neurons in the second filter must be connect to this layer.

c. For `NetLin` the three most missclassified characters were 'ma' as 'su', 'ha' as 'su' and 'ki' as 'su'. The characters 'ma', 'ha' and 'ki' all exhibit variations in the way that they are written, but most of these variations include a long vertical line through the center of the image with offshoots from this center line, which is similar to the character 'su' which does not vary as much between the different samples as the missclassified characters. As a result, `NetLin` incorrectly predicts 'su' as it has a similar structure to the other characters variations. The most correctly recognised character by `NetLin` is 'o' at 765 correct predictions. The same argument for missclassified characters is applicable to the `NetFull` model, which missclassifies 'ha' as 'su', 'ma' as 'su' and 'ki' as 'ma'. since the `NetFull` model is able to learn more complex decision boundaries, the character 'ki' is now mislabelled as 'ma', as these two character resemble each other more closely than 'ki' and 'su', as was misclassified in `NetLin`. NetFull correctly classifies 'tsu' the most, with 913 true positives. Interestingly, `NetConv` has a similar missclassification to the other models, although to a lesser extent, as it missclassifies 'ha' as 'su', 'ki' as 'ma' and 'o' as 'na'. The fact that `NetFull` and `NetConv` both missclassify 'ki' as 'ma' demonstrates the limitations of `NetLin` in learning complex decision boundaries, even if the learned decision boundaries are often wrong. Also, the missclassification of 'o' as 'na' demonstrates the convolution of the input characters, as while the overall shapes of 'o' and 'na' a dissimilar, they have many small features in common, which the convolution layers feed into the fully connected layers. For `NetConv`, the most accurately classified character is 'ma' at 976 true positives. Comparing the three confusion matrices, it is evident that correctly classifying 'ha' is difficult as not only do all three models missclassify this character frequently, but the convolution network offers little benefit over `NetFull`, as it lowers the missclassification rate from 91 to 50, which is still a high rate of confusion for the network compared to the other missclassified characters. Note that while the images of the characters may have complex features these are fed into the networks as either a vector in $\mathbb{R}^{738}$ for `NetLin` and `NetFull` or as a vector in $\mathbb{R}^{4705}$ in `NetConv`'s first fully connected layer, the difficulty in creating more complex decision boundaries in the input space does

not mean that more complex characters written in $\mathbb{R}^2$ require more analysis than those that appear to be more simple, rather, separation of the class labels in the vector space is difficult.

Part 2: Fractal Classification Task

1. Code for `Full2net` is supplied in the file `frac.py`.

2. The output plot for `Full2Net` is in Figure 4 .The total number of independent parameters in `Full2Net` is computed by,

$$\begin{aligned}
\text{weights per neuron (hid1)} &= 1 + \text{number of neurons (in)} \\
&= 1 + 2 \\
&= 3, \\
\text{independent parameters (hid1)} &= \text{weights per neuron (hid1)} \times \text{number of neurons (hid1)} \\
&= 3 \times 30 \\
&= 90, \\
\text{weights per neuron (hid2)} &= 1 + \text{number of neurons (hid1)} \\
&= 1 + 30 \\
&= 31, \\
\text{independent parameters (hid2)} &= \text{weights per neuron (hid2)} \times \text{number of neurons (hid2)} \\
&= 31 \times 30 \\
&= 930, \\
\text{weights per neuron (out)} &= 1 + \text{number of neurons (hid3)} \\
&= 1 + 30 \\
&= 31, \\
\text{independent parameters (out)} &= \text{weights per neuron (out)} \times \text{number of neurons (out)} \\
&= 31 \times 1 \\
&= 31, \\
\text{total independent parameters} &= \text{independent parameters (hid1)} + \text{independent parameters (hid2)} + \\
&\quad \ldots \text{independent parameters (out)} \\
&= 90 + 930 + 31 \\
&= 1051
\end{aligned}$$

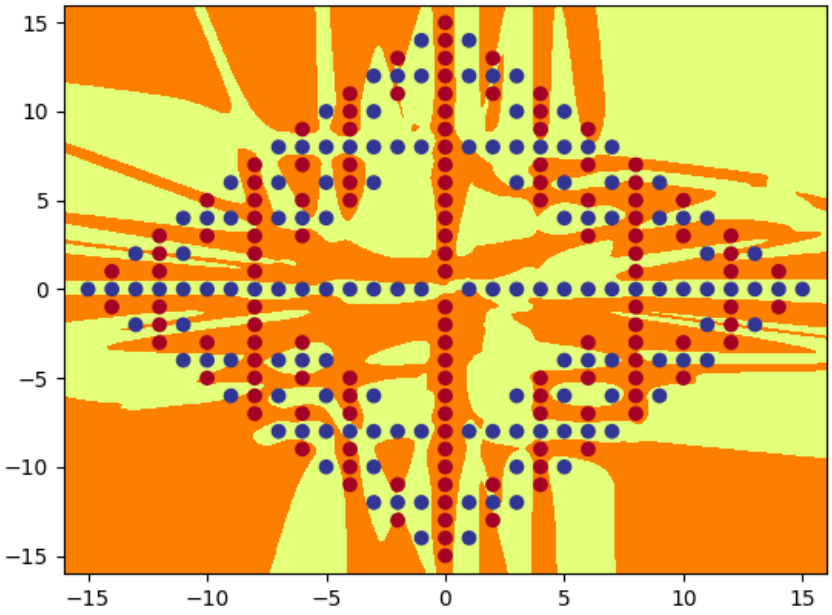
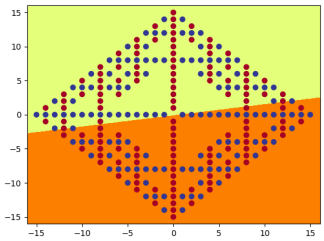


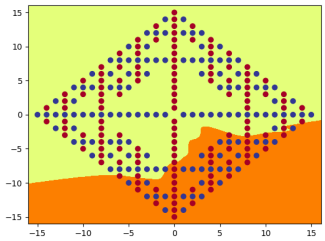
Figure 4: `Full2Net` Output

3. Code for `Full3net` is supplied in the file `frac.py`.

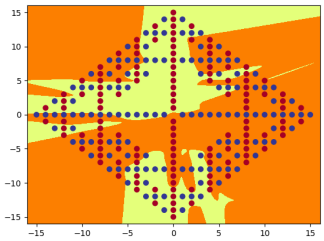
4. The plots for the output and each hidden node of `Full3Net` are in Figure 5.



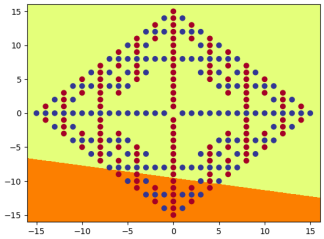
Hidden Layer 1 Node 0



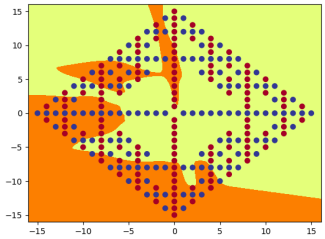
Hidden Layer 2 Node 0



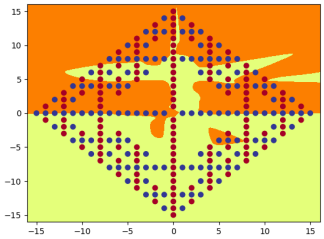
Hidden Layer 3 Node 0



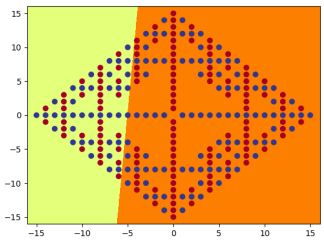
Hidden Layer 1 Node 1



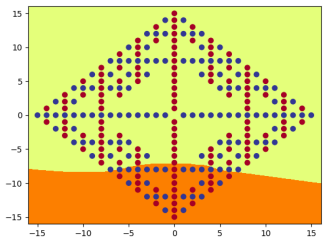
Hidden Layer 2 Node 1



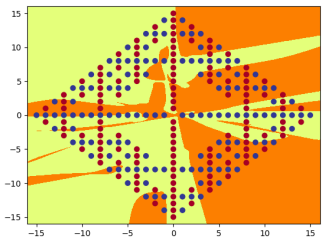
Hidden Layer 3 Node 1



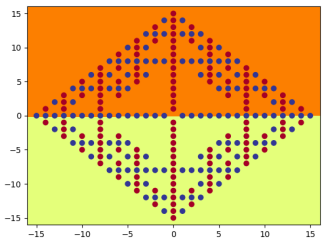
Hidden Layer 1 Node 2



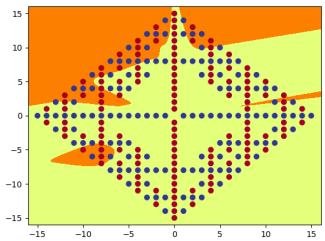
Hidden Layer 2 Node 2



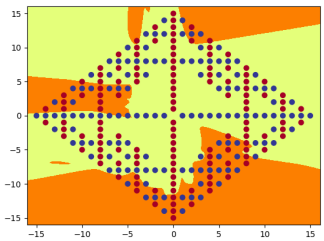
Hidden Layer 3 Node 2



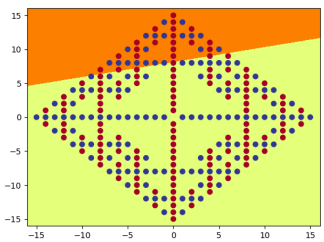
Hidden Layer 1 Node 3



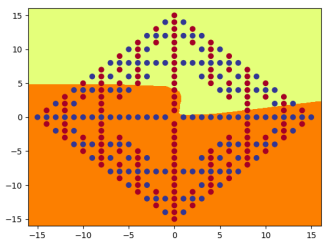
Hidden Layer 2 Node 3



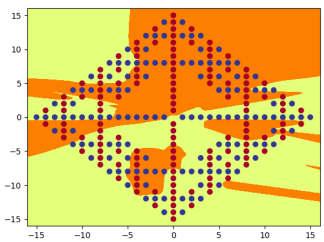
Hidden Layer 3 Node 3



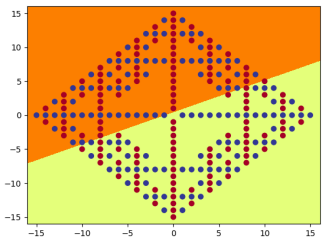
Hidden Layer 1 Node 4



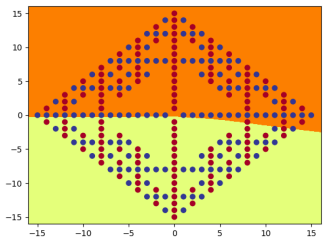
Hidden Layer 2 Node 4



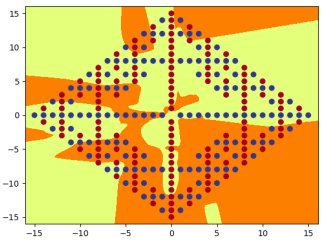
Hidden Layer 3 Node 4



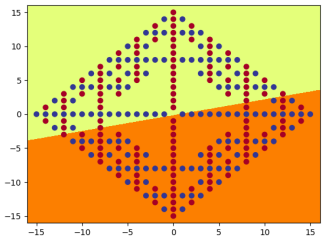
Hidden Layer 1 Node 5



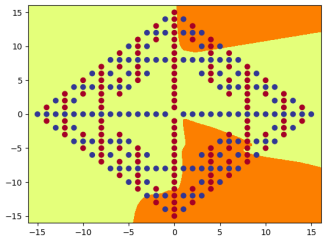
Hidden Layer 2 Node 5



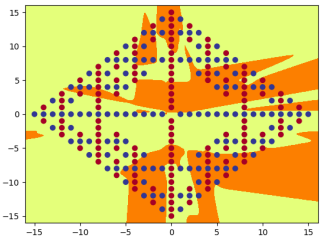
Hidden Layer 3 Node 5



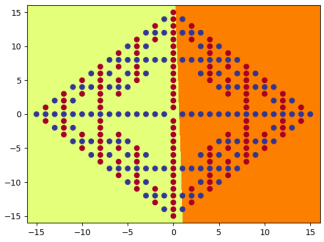
Hidden Layer 1 Node 6



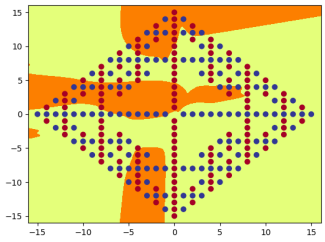
Hidden Layer 2 Node 6



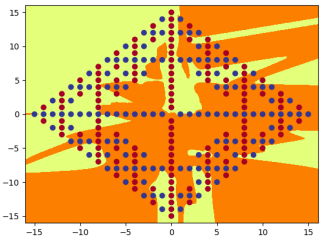
Hidden Layer 3 Node 6



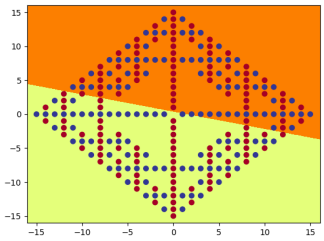
Hidden Layer 1 Node 7



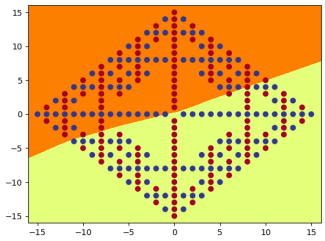
Hidden Layer 2 Node 7



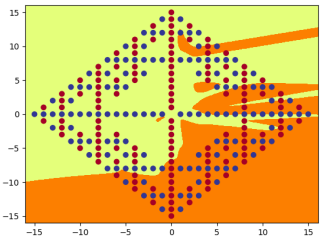
Hidden Layer 3 Node 7



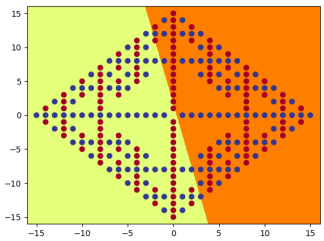
Hidden Layer 1 Node 8



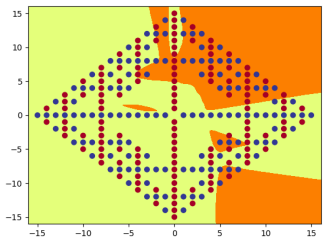
Hidden Layer 2 Node 8



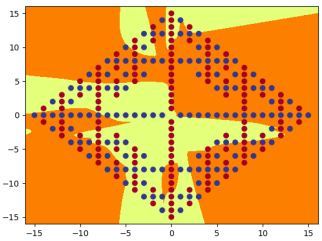
Hidden Layer 3 Node 8



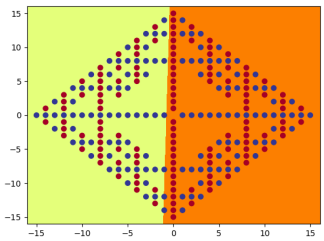
Hidden Layer 1 Node 9



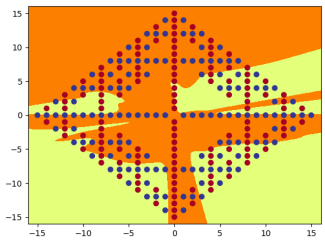
Hidden Layer 2 Node 9



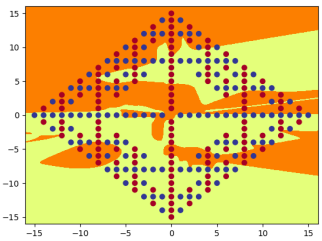
Hidden Layer 3 Node 9



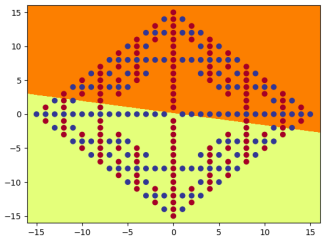
Hidden Layer 1 Node 10



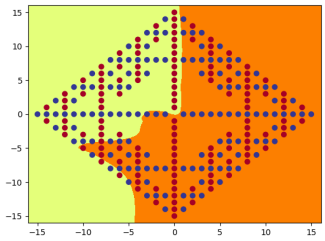
Hidden Layer 2 Node 10



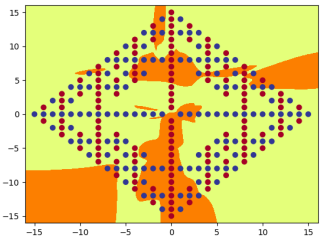
Hidden Layer 3 Node 10



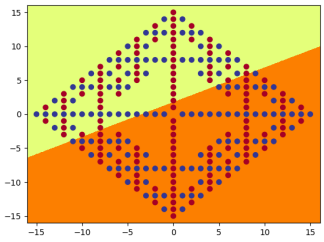
Hidden Layer 1 Node 11



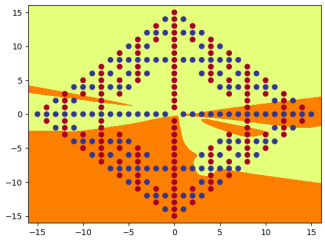
Hidden Layer 2 Node 11



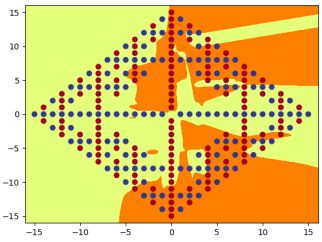
Hidden Layer 3 Node 11



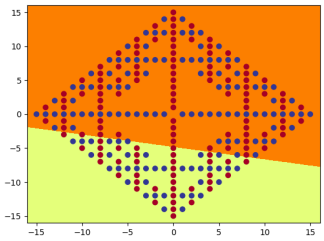
Hidden Layer 1 Node 12



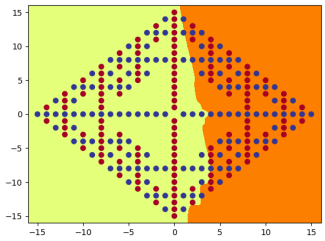
Hidden Layer 2 Node 12



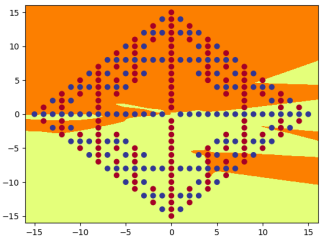
Hidden Layer 3 Node 12



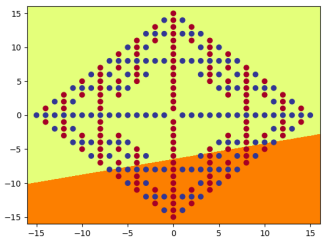
Hidden Layer 1 Node 13



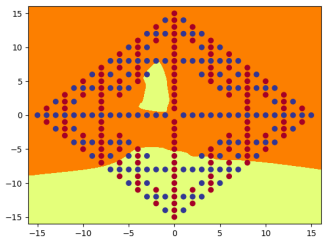
Hidden Layer 2 Node 13



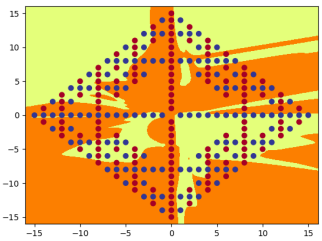
Hidden Layer 3 Node 13



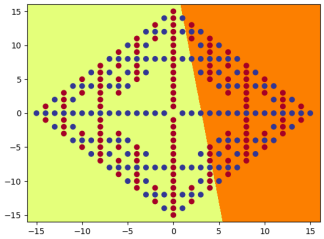
Hidden Layer 1 Node 14



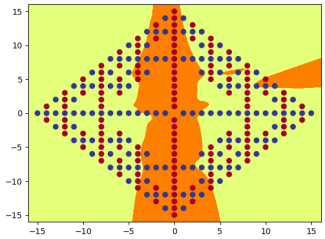
Hidden Layer 2 Node 14



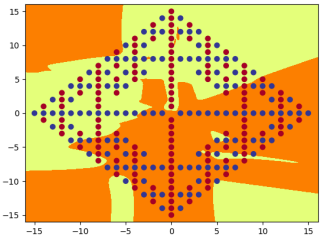
Hidden Layer 3 Node 14



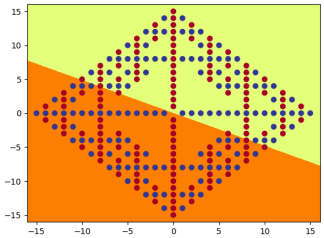
Hidden Layer 1 Node 15



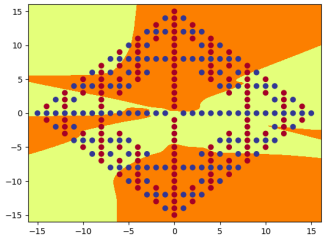
Hidden Layer 2 Node 15



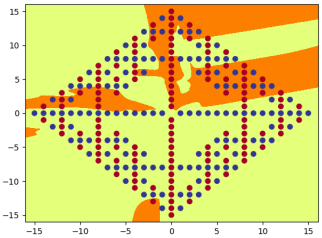
Hidden Layer 3 Node 15



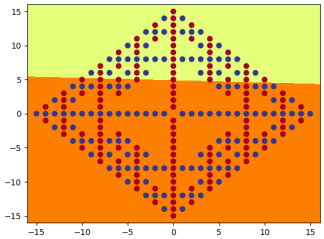
Hidden Layer 1 Node 16



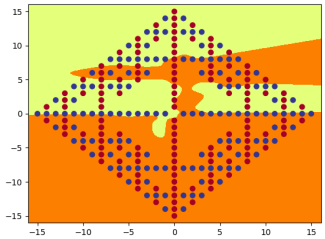
Hidden Layer 2 Node 16



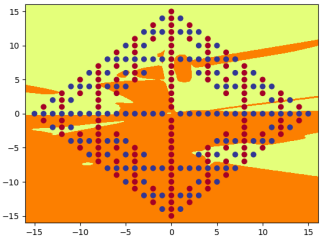
Hidden Layer 3 Node 16



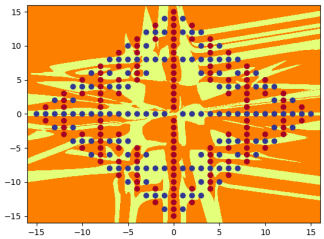
Hidden Layer 1 Node 17



Hidden Layer 2 Node 17



Hidden Layer 3 Node 17



Output

Figure 5: `Full3Net` Output and Hidden Layers

The number of independent parameters in `Full3Net` is given by,

$$
\begin{aligned}
\text{weights per neuron (hid1)} &= 1 + \text{number of neurons (in)} \\
&= 1 + 2 \\
&= 3, \\
\text{independent parameters (hid1)} &= \text{weights per neuron (hid1)} \times \text{number of neurons (hid1)} \\
&= 3 \times 18 \\
&= 54, \\
\text{weights per neuron (hid2)} &= 1 + \text{number of neurons (hid1)} \\
&= 1 + 18 \\
&= 19, \\
\text{independent parameters (hid2)} &= \text{weights per neuron (hid2)} \times \text{number of neurons (hid2)} \\
&= 19 \times 18 \\
&= 342, \\
\text{weights per neuron (hid3)} &= 1 + \text{number of neurons (hid2)} \\
&= 1 + 18 \\
&= 19, \\
\text{independent parameters (hid3)} &= \text{weights per neuron (hid3)} \times \text{number of neurons (hid3)} \\
&= 19 \times 18 \\
&= 342, \\
\text{weights per neuron (out)} &= 1 + \text{number of neurons (hid3)} \\
&= 1 + 18 \\
&= 19, \\
\text{independent parameters (out)} &= \text{weights per neuron (out)} \times \text{number of neurons (out)} \\
&= 19 \times 1 \\
&= 19, \\
\text{total independent parameters} &= \text{independent parameters (hid1)} + \text{independent parameters (hid2)} + \\
&\quad \ldots \text{independent parameters (hid3)} + \text{independent parameters (out)} \\
&= 54 + 342 + 342 + 19 \\
&= 757
\end{aligned}
$$

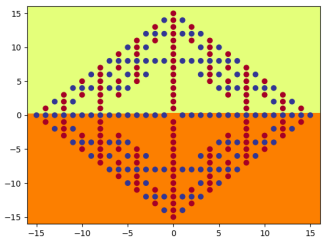
5. Code for `DenseNet` is supplied in the file `frac.py`.

6. The graphs of each of the hidden nodes of `DenseNet` are in Figure **??**. The following calculations

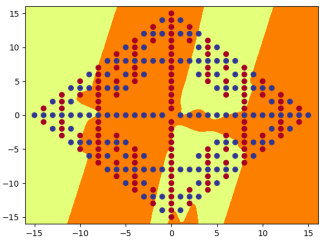
give the total number of independent parameters in `DenseNet`,

$$
\begin{aligned}
\text{weights per neuron (hid1)} &= 1 + \text{number of neurons (in)} \\
&= 1 + 2 \\
&= 3, \\
\text{independent parameters (hid1)} &= \text{weights per neuron (hid1)} \times \text{number of neurons (hid1)} \\
&= 3 \times 15 \\
&= 45, \\
\text{weights per neuron (hid2)} &= 1 + \text{number of neurons (in)} + \text{number of neurons (hid1)} \\
&= 1 + 2 + 15 \\
&= 18, \\
\text{independent parameters (hid2)} &= \text{weights per neuron (hid2)} \times \text{number of neurons (hid2)} \\
&= 18 \times 15 \\
&= 270, \\
\text{weights per neuron (out)} &= 1 + \text{number of neurons (in)} + \text{number of neurons (hid1)} + \\
&\quad \ldots \text{number of neurons (hid2)} \\
&= 1 + 2 + 15 + 15 \\
&= 33, \\
\text{independent parameters (out)} &= \text{weights per neuron (out)} \times \text{number of neurons (out)} \\
&= 33 \times 1 \\
&= 33, \\
\text{total independent parameters} &= \text{independent parameters (hid1)} + \text{independent parameters (hid2)} + \\
&\quad \ldots \text{independent parameters (out)} \\
&= 45 + 270 + 33 \\
&= 348
\end{aligned}
$$

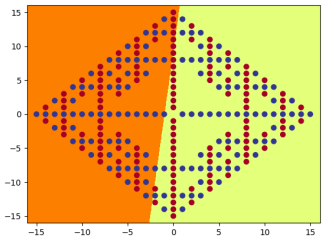
7.a. In order for `Full2Net` to be consistently trained successfully, approximately 30 hidden neurons in each of the hidden layers are required, using the default parameters, with a total of 1051 independent parameters. Since the network only consists of three layers, the number of epochs during training is about 57000. For `Full3Net`, the number of neurons in each of the three hidden layers must be approximately 18 to achieve consistency when training. As such, the network only has 757 independent parameters, which is far less than `Full2Net`. However, the training requires roughly 108800 epochs. Therefore, while `Full3Net` is able to correctly classify the fractal with fewer independent parameters than `Full2Net`, the number of epochs required is far greater, due to the added complexity of the added hidden layer. In `Densenet`, only 15 neurons are required at the two hidden layers to consistently train the network, which takes approximately 130000 epochs. The added



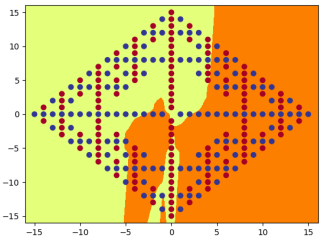
Hidden Layer 1 Node 0



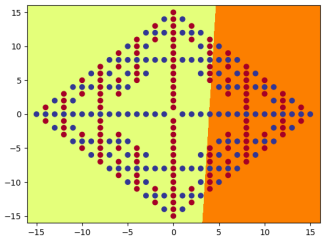
Hidden Layer 2 Node 0



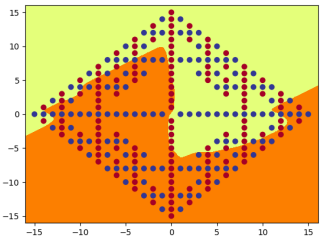
Hidden Layer 1 Node 1



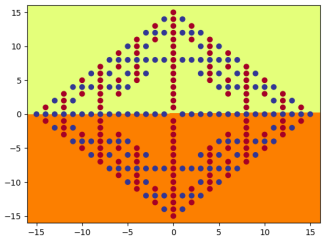
Hidden Layer 2 Node 1



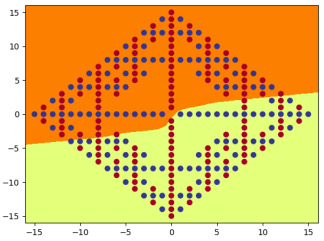
Hidden Layer 1 Node 2



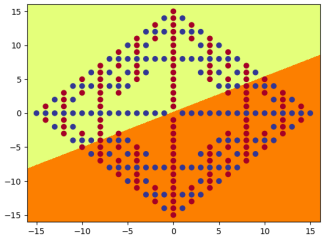
Hidden Layer 2 Node 2



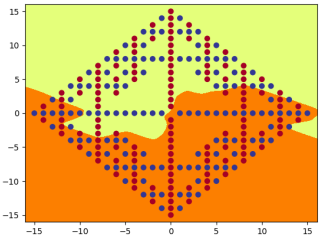
Hidden Layer 1 Node 3



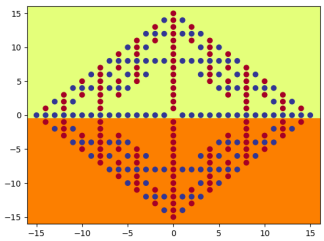
Hidden Layer 2 Node 3



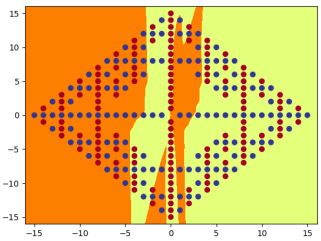
Hidden Layer 1 Node 4



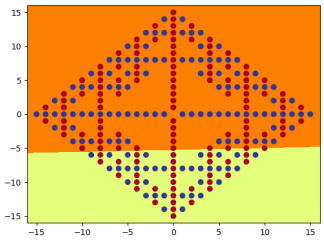
Hidden Layer 2 Node 4



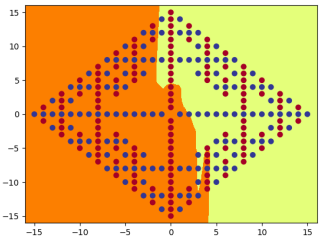
Hidden Layer 1 Node 5



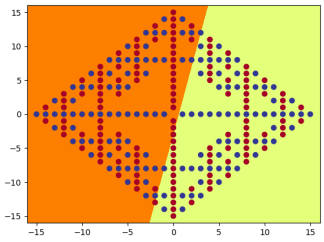
Hidden Layer 2 Node 5



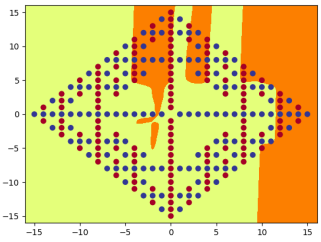
Hidden Layer 1 Node 6



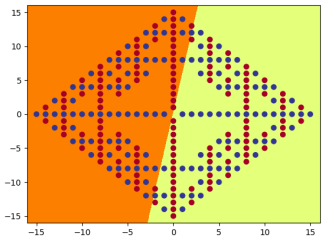
Hidden Layer 2 Node 6



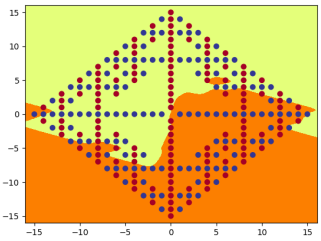
Hidden Layer 1 Node 7



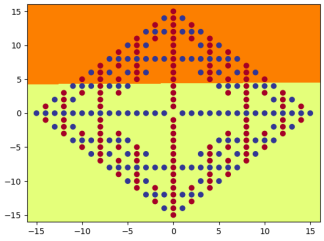
Hidden Layer 2 Node 7



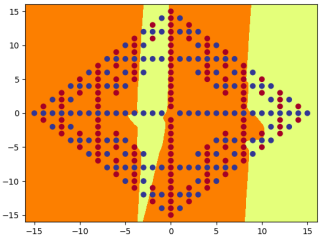
Hidden Layer 1 Node 8



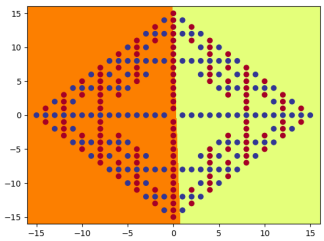
Hidden Layer 2 Node 8



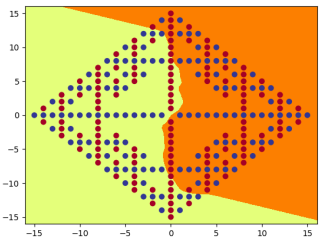
Hidden Layer 1 Node 9



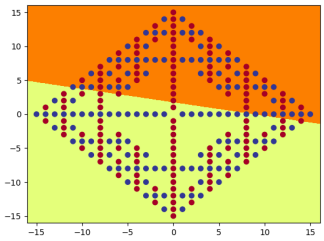
Hidden Layer 2 Node 9



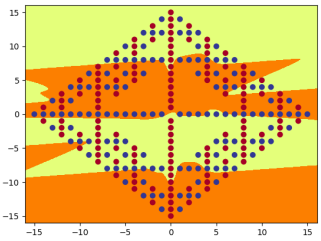
Hidden Layer 1 Node 10



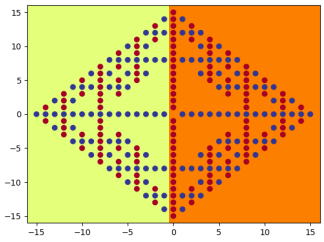
Hidden Layer 2 Node 10



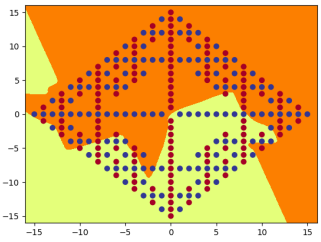
Hidden Layer 1 Node 11



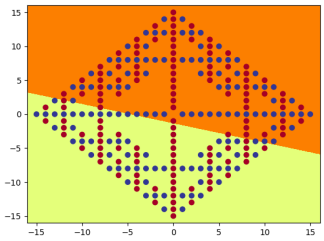
Hidden Layer 2 Node 11



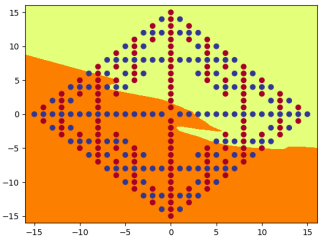
Hidden Layer 1 Node 12



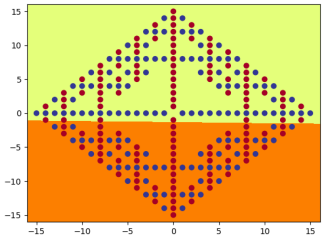
Hidden Layer 2 Node 12



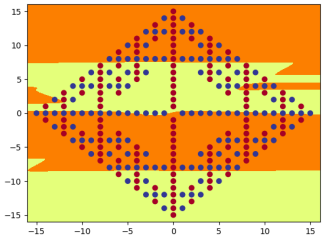
Hidden Layer 1 Node 13



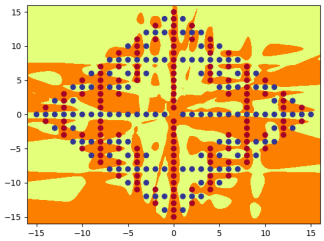
Hidden Layer 2 Node 13



Hidden Layer 1 Node 14



Hidden Layer 2 Node 14



Output

Figure 6: `DenseNet` Output and Hidden Layers

shortcut connections in `DenseNet` evidently adds more complexity to the base network `Full2Net` than adding an extra hidden layer, as was added in `Full3Net`, as the number of training epochs for `DenseNet` is far greater than `Full3Net`. However, while the training epochs required increase with the complexity of the network, the number of independent parameters is reduced significantly, to only 348 in `DenseNet`, and this is true for all three models. For example, the 30 neurons in the hidden layers of `Full2Net` allow the input fractal to be correctly classified by decision boundary in $\mathbb{R}^{30}$, whereas with the added residual components, whether that be one or both of the hidden layers, in `DenseNet` allow for successful training with only 15 neurons, that is, defining a decision boundary at the output node in $\mathbb{R}^{15}$ that correctly classifies the fractal.

b. There are two input neurons, 18 neurons in each of the three hidden layers with `tanh` activation and one output neuron with `sigmoid` activation in `Full3Net`. prior to the application of the activation function, neurons in the first hidden layer learns linear decision boundaries as the hyperplane $h_i^1(\mathbf{x}) : \mathbb{R}^2 \to \mathbb{R}$ in three dimensional space [1]. After the application of the activation function tanh, the decision boundary becomes a curved surface in the third dimension, but the decision boundary remains linear across the plane $x = y = 0$, which is evident in Figure 5, where all the boundaries are linear in the first hidden layer plots. Together the neurons form a complex boundary in $\mathbb{R}^3$. These neurons take the input from the input layer and produce an output in one dimension, which serves as input for the second hidden layer. The second hidden layer learns concave features by defining non-linear decision boundaries $h_j^2(\mathbf{h}^1) : \mathbb{R}^{18} \to \mathbb{R}$ at each neuron that exist in nineteen dimensional space [1]. At each of the nodes the input space from the first hidden layer is eighteen dimensional, and after applying the activation function this gives a one dimensional output. The graphs of the activations of the second hidden layer in Figure 5 are complex, as they represent the decision curve in two dimensional space, while the boundaries exist in nineteen dimensions, and these depicted boundaries are only those surfaces that cross $x = y = 0$. Similarly, the third hidden layer takes these activations and is able to learn features in input space that are convex [1], through output functions $\text{out}_k(\mathbf{h}^2) : \mathbb{R}^{18} \to \mathbb{R}$. As this layer builds upon the activations of the second layer, the decision boundaries in Figure 5 are more complex than those in the other two layers. These boundaries are also non-linear and are represented in two dimensions, hence the odd shapes. Finally, the outputs of the third layer serve as input into the output layer. This single neuron takes the eighteen

dimensional input and produces an output in one dimension, combining the decision boundaries of the third hidden layer to produce a complex non-linear decision function in nineteen dimensional space, of which the two dimensional representation across $x = y = 0$ is in Figure 5.

The network `DenseNet` consists of two input neurons, 15 hidden neurons at the two hidden layers with `tanh` activation and one output neuron with `sigmoid` activation, as well as skip connections from input to the second hidden layer, input to the output layer and from the first hidden layer to the output layer. The first hidden layer takes the two dimensional input and produces a linear decision function $h_i^1(\mathbf{x}) : \mathbb{R}^2 \to \mathbb{R}$ at each of the neurons before application of the activation function [1]. Applying `tanh` means that the decision boundary is a curve in three dimensional space, but as was in `Full3Net`, the boundary is still linear across the horizontal plane, as displayed in the first hidden layer boundaries in Figure 6. Together, the neurons of the first hidden layer create a decision boundary in sixteen dimensional space that is able to learn all linearly separable features. The activations are then used as input for the second hidden layer, along with the two input neurons. Individual neurons in the second hidden layer take input in 17 dimensions and output a one dimensional activation response. The non-linear functions $h_j^2(\mathbf{x}, \mathbf{h}^1) : \mathbb{R}^{17} \to \mathbb{R}$ exist in 18 dimensional space and learn concave features in the data [1], but unlike `Full2Net` or `Full3Net`, the neurons in the second hidden layer of `DenseNet` include the inputs. The shortcut connections from the input to the second layer mean that the second layer is able to learn a function that incorporates the information passed from the input and combines it with the functions learned in the first hidden layer, instead of only being able to formulate a function based off of the first hidden layers activations, as was in `Full2Net` and `Full3net`. The output neuron in `DenseNet` takes the activations from the hidden layers and the input layer, which totals an input space of 32 dimensions, and outputs a single dimensional response with the sigmoid activation. The non-linear decision functions $\text{out}_k(\mathbf{x}, \mathbf{h}^1, \mathbf{h}^2) : \mathbb{R}^{32} \to \mathbb{R}$ exist in 33 dimensional space, and has the added benefit of being able to include the inputs and the activations of the first hidden layer.

Note that all neurons learn a linear function prior to activation, this is true for all basic feed-forward neural networks, irrespective of what dimensional space this is in.

c. The network `Full2Net` produces an output function in 31 dimensional space that is able to correctly predict the given fractal by learning convex features [1]. While similar in appearance, the output function in `Full3Net` exists in 19 dimensional space, and is far more complex than the output function in `Full2Net`, as it is able to learn concave features [1], due to the additional hidden layer. As such, these two functions differ not only in the dimensions in which they exist, but also the complexity of the functions themselves, with respect to the types of features they are able to learn. In comparison, `DenseNet` learns an output function that exists in 33 dimensional space, which is more than the other two networks. Yet, being able to access inputs from not just the previous layer, like in `Full2Net` and `Full3Net`, means that the output decision function of `DenseNet` is far more complex than that of `Full2Net`, despite consisting of an equal number of hidden layers. Observation of the output plots shows that the functions of `Full2net` and `Full3Net` are relatively similar in appearance, which is logical as `Full3Net` is an extension of `Full2Net`, albeit with fewer

neurons in each hidden layer. The plot of `DenseNet` is far more different when compared with the other two networks plots. Instead of skewed regions in the decision boundaries, as is in `Full2Net` and `Full3Net`, the `DenseNet` plot exhibits a boundaries that often blocks, that is, with vertical and horizontal bounds, and consists of fewer skewed boundaries. This is evidence of the presence of the shortcut connections, as each hidden node is influenced by all the previous layers, creating a function that is less sporadic than the other two networks, as at the second hidden output layers it is able to temper the complex non-linear decision boundaries of the previous hidden layers with the input data.

Part 3: Encoder Networks

1. The image obtained by training the encoder network on the created tensor is in Figure 7. Note that the generated image must be first rotated 90 degrees anti-clockwise and then flipped about the vertical axis to easily compare it with the given image.

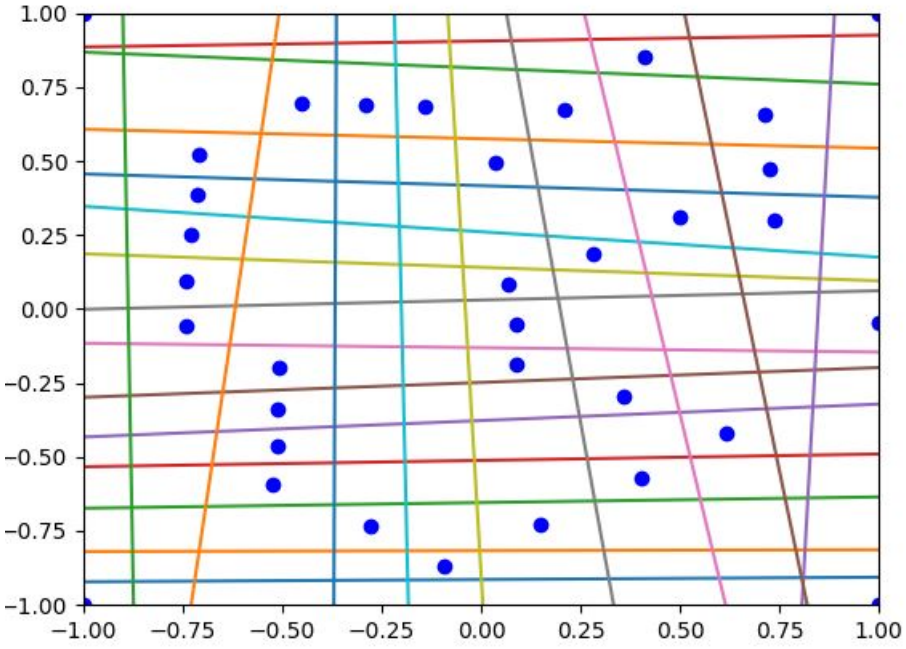


Figure 7: Encoder Network Output

Part 4: Hidden Unit Dynamics for Recurrent Networks

1. The annotated output of training the Simple Recurrent Network (SRN) on the Reber Grammar is in Figure 8.

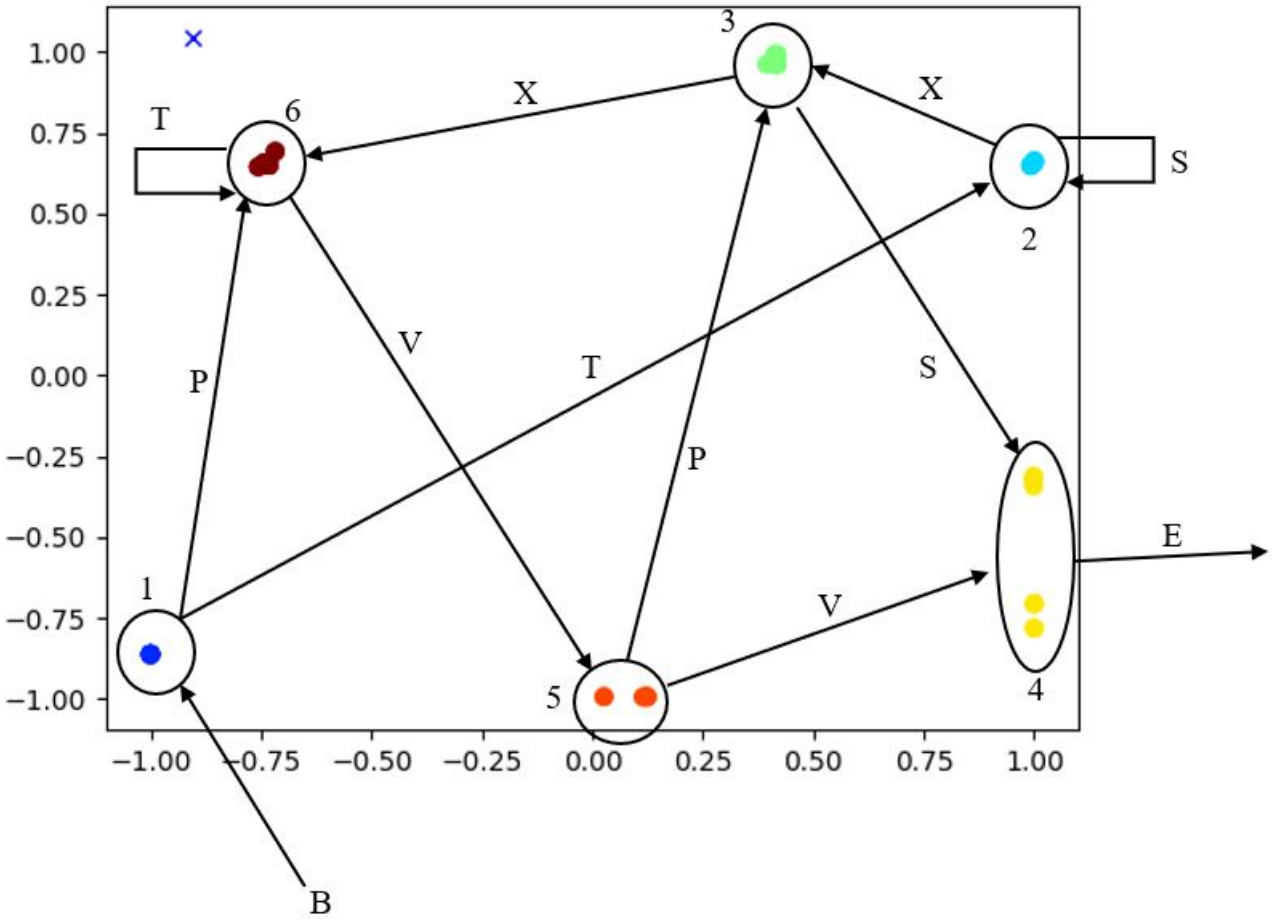


Figure 8: Annotated Hidden Unit Activations of SRN on Reber Grammar

2. The plot of the hidden unit activations from training the SRN on the $a^n b^n$ task is in Figure 9.

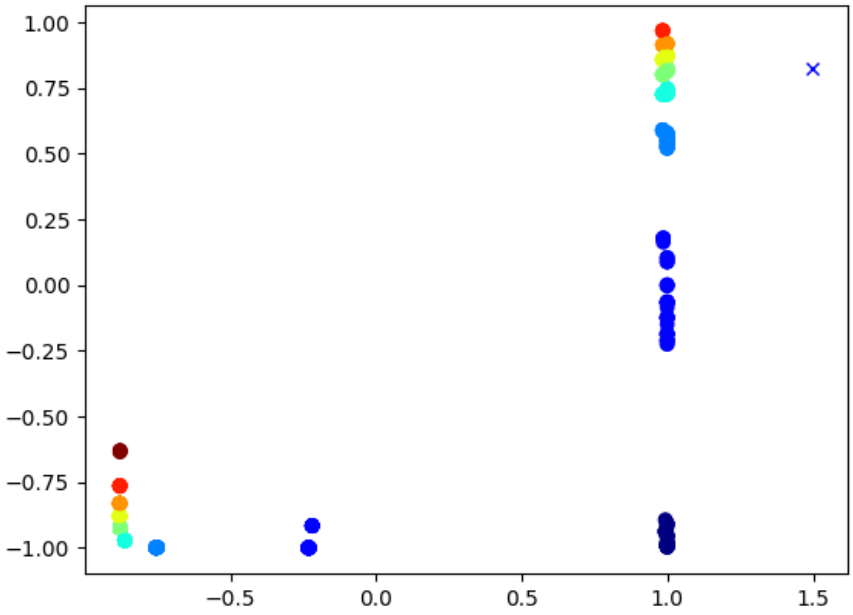


Figure 9: Hidden Unit Activations of SRN on $a^n b^n$

3. In the $a^n b^n$ task, only the b's following the first b in each sequence and the first a in the next sequence, as well as the very first a, are deterministic [5], as the number of a's following the first a in each sequence is an unknown variable. The hidden unit activations of the SRN in Figure 9 reveal that the SRN was trained by evolution instead of backpropagation, since the colour gradient of the points does not represent an oscillation towards an attractive point to count up the number a's, and then away from a repellent point to count down the remaining b's [5], rather, it counts the sequence in a monotonic fashion [5]. Observing the output in the terminal for the ten test epochs, called by the `python3 seq_plot.py --lang anbn --epoch 100` command, it is evident how the SRNs hidden unit activations allow it to correctly predict the second b through to the first a of each sequence, and a graphical example is given in Figure 10. The SRN has learned weights through training that create a linearly separable decision boundary between the hidden unit activations, one side corresponding to an a prediction and the other b. The initial a always corresponds to a hidden unit activation of $(-0.22, -0.92)$, and will always move to the next activation along the red line in the Figure, regardless of whether it is an a or b. As the SRN processes a's the hidden unit activations shift along the bottom of the plot. When the SRN encounters the first b, the activations still remain in the bottom left of the plot, that is, the SRN incorrectly predicts an a classification since the activations remain in the 'a region'. Suppose the SRN was to predict that a b would occur at a certain point in the sequence, but it is incorrect, then the weights would have to try to adjust during training for each incorrectly predicted b, and potentially at multiple points for each sub-sequence, which would be impossible with only two hidden nodes and weights that remain

constant after training.. Instead, misclassification of the first b in each sequence means that the SRN is able to learn weights that ensure the hidden unit activations are linearly separable in $\mathbb{R}^2$ for any $a^n b^n$ sequence, with the decision boundary dictated by the function of the output neurons. The color brown, which is the max on the colour map 'jet', only appears in the 'a region' since the SRN cannot predict the first b in the sequence it mistakenly predicts an a, and therefore the colour of the predictions hidden unit activations, which corresponds to the number of predicted as each sub-sequence, is one higher than when counting down the bs because of this misclassification. Now that the first b has been processed by the SRN its activation is copied to the context layer so that the next input that is processed the activations will shift to the b region, given that the next input is a b, otherwise, the activations will transition to the bottom right of the activations space to correctly predict an a. The context layer allows the SRN to count down any remaining bs along the right side of the image towards the bottom of the plot, and when it encounters the first a in the next sub sequence, the activations shift to the 'a region'. Not all the potential activations are used for each sub-sequence, as shown in Figure 10. Furthermore, note that the jump from 'a region' to 'b region' is a vector with equal orientation, no matter the length of the sub-sequence, since the weights of the SRN are constant it learns during training to transition to an activation that will allow the SRN to correctly predict the remaining bs and first a, that is, the activation must be high enough so that the SRN can incrementally move to activations that are lower in $\mathbb{R}^2$ and still remain in the 'b region' Also, the SRN is not certain, usually giving a probability of around 99%, that the final b will appear in the sequence, even though its output is always correct, as the SRN learns the sequence but is not told the rule that an equal number of b's will follow each sub-sequence of a's, so as it nears the decision boundary it becomes slightly less certain. The monotonic behaviour and spacing between average activations for both a and b predictions is similar to gradient ascent with momentum [6], where the 'maximal point' in $\mathbb{R}^2$ is the average activation of the classified letter in a sub-sequence of maximal length, more specifically, activations labelled that are coloured brown and red for a and b predictions respectively. Overall, the SRN leans weights through training that create linearly separable hidden unit activations for the $a^n b^n$ sequence.

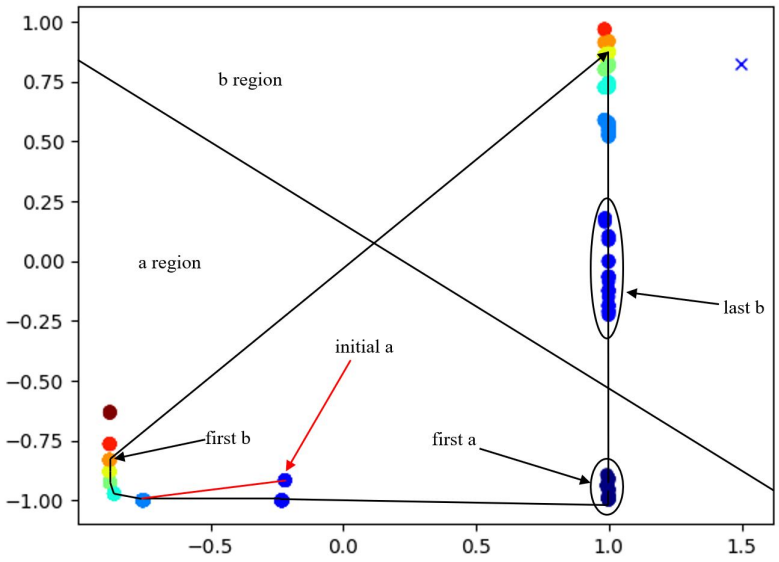


Figure 10: Hidden Unit Dynamics of SRN on $a^n b^n$ Task

4. The SRN is first trained on the $a^n b^n c^n$ task for 200 epochs, and then a separate testing set is applied to generate the hidden unit activations in Figure 11.

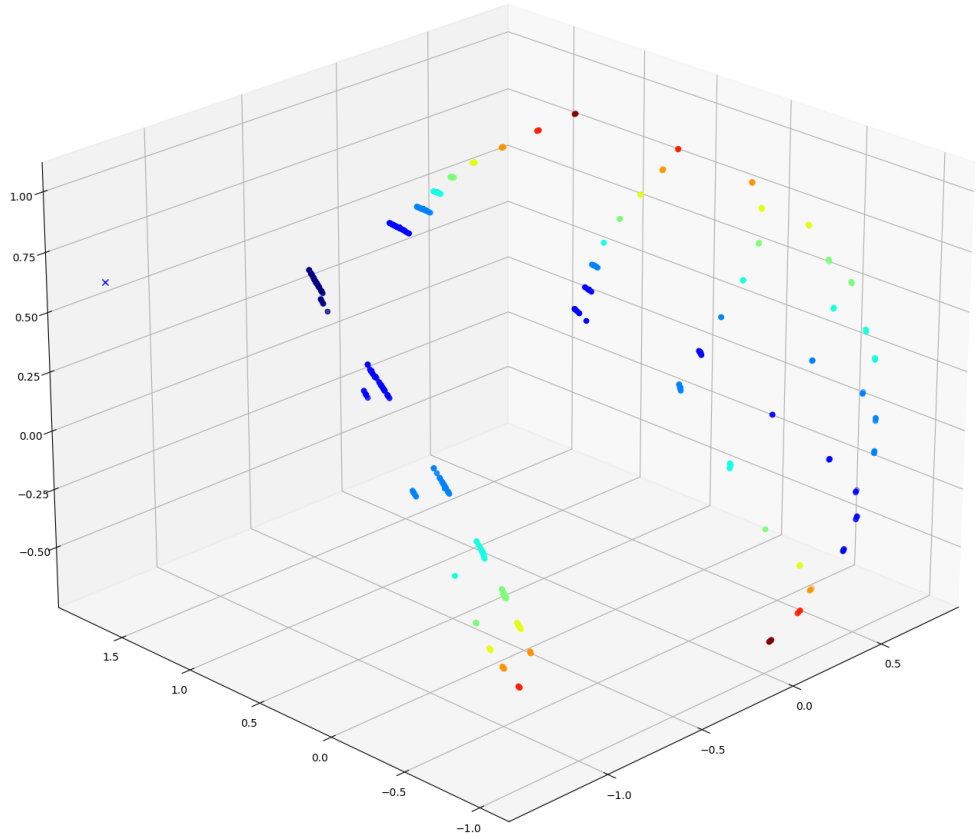


Figure 11: Hidden Unit Activations of SRN on $a^n b^n c^n$

5. After 200 training epochs the SRN has learned hidden unit activations that can accurately predict the letters in each sequence that are deterministic, that is, all letters except the second a to the first b, inclusive, of each sub-sequence [5], since the number of a's in each sub-sequence is unknown. Note that for each epoch the SRN is fed a sequence that is made up of many sub-sequences that follow the $a^n b^n c^n$ rule, excluding the trailing a. Comparing the numeric and graphic representation of the hidden unit activations of the ten test epochs, as well as reference to the colour map 'jet', it is evident as to which hidden unit activations correspond to which classification. These hidden unit activations are split into three regions in $\mathbb{R}^3$, governed by functions at the output neurons. Although there are variations in the exact activations for each possible letter index in the sub-sequences, which are grouped by colour, they tightly grouped in similar activation spaces and will be treated as single entities, for simplicity. The possible activations for the a classification are at the top of the plot, and count towards up towards the dark red activation. Although the probability that the next letter in each sub-sequence will be a b increases after the first a, if the SRN predicts when the first b will occur and this prediction is incorrect the SRN is unable to keep

track and learn the rest of the sequence as the context layer now has the activation of a b instead of the correct a activation, and as such, the weights would have to correct for this mistake which is impossible as the weights remain constant after training. Instead, the SRN can achieve a low error by misclassifying the first b in each sub-sequence. Then, as the SRN encounters the next b, it is able to learn weights that create separable activations, that is, the second b causes a shift in the hidden unit activations when processed and they move into the 'b region'. From this point, the SRN is able to count down the remaining number of b's as the learned weights keep the activations, when presented with an encoded b, in the 'b region', and this is true for any length of sequence. Since the network does not know how many b's to expect, as the number of a's after the first is variable, the activations that the string of b's can take form several paths through $\mathbb{R}^3$, and since the learned weights do not change based on the number of a's in each sub-sequence, these paths are similar in curvature, that is, the change in activation for subsequent b's in the sub-sequence are similar as the weights used to traverse these paths are the same, while it is the input and path origin in $\mathbb{R}^3$ that varies. While these paths are similar in curvature, they differ in length to ensure that the final b in each sequence lies in a separable 'b region'. When the SRN processes the first c, these activations now lie in a separable c region that is close to the 'b region'. If the sequence is greater than one, than the second input c causes a dramatic shift to the far side of the hidden unit space, if $n = 1$ then the shift occurs upon reading the first c. For the activations of the remaining c's, given that there are some yet to classify, the activations shift to the left side of Figure 12 and join a single activation pathway. Unlike the potential activations of the b's, omitting the first b in the sub-sequences, for the c's the activations exist in a thin region. This is because the a's set the length of the sub-sequence and the b's, from the second, must be separable in their own classification region, the 'b region'. But since the learned weights are shared across different sub-sequence lengths, the b paths must provide a pathway for each possible length of a's. The b's must also setup the string of c's, which is why the pathways are curve inwards towards the plane in which the activations for the c's exist. The subsequent b's in each string must perform this function, as it is impossible to have a single pathway, as seen in the a and c activations, with shared weights between different lengths of sub-sequences as the point of origin for these paths is different. As such, the pathways for the subsequent b's in each sub-sequence form surface in $\mathbb{R}^3$ that connects the activations of the a's and c's. Following, the SRN counts down the remaining c's, moving up towards the top left of Figure 12, having learnt activations that are separable. The next a is correctly predicted as the activation of the last c is always the in a similar region, and as such, the context layer and weights push the hidden unit activations for the a into the 'a region'. Overall, the SRN learns the $a^n b^n c^n$ task by updating weights by during training, which allow it to correctly classify those letters that are deterministic by creating linearly separable hidden unit activations. Notably, sub-sequences that have a length greater than one see the biggest shift in activation when the SRN processes the second letter in the string. An example of this hidden unit activation movement is seen in Figure 12.

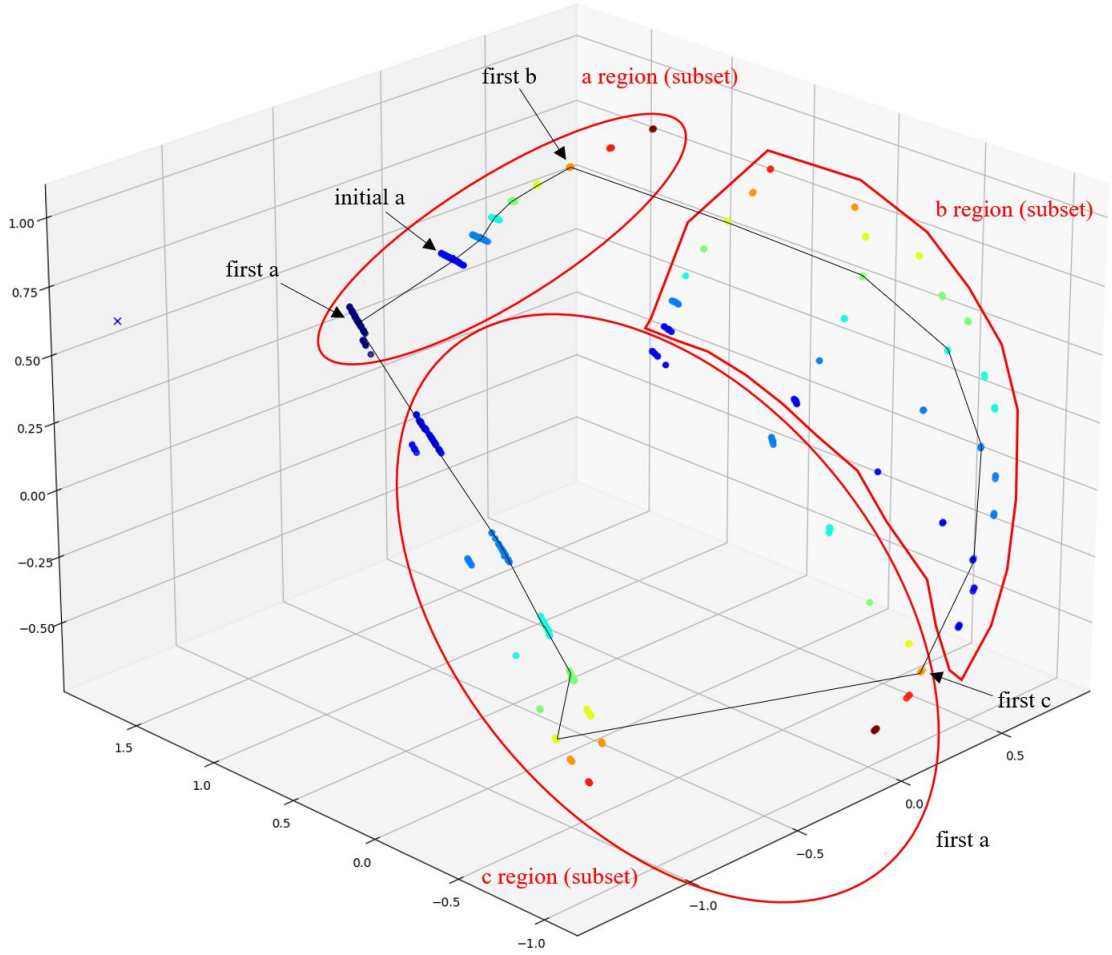


Figure 12: Hidden Unit Activations of SRN on $a^n b^n c^n$

6. A Long Short Term Memory (LSTM) network is able to learn both long and short range dependencies by updating a separate context and hidden layer at each time-step [2], which allow it to learn the Embedded Reber Grammar (ERG) task with high accuracy. The hidden layer of the LSTM provides information on short range dependencies, while the context layer retains the long range dependencies [7]. This is fundamentally different from the SRN, which simply copies its hidden unit activations to the context layer, and as such, loses previously inputted information over time [4], as the context layer has no mechanism to learn how much of the current activations it should retain. During training the weights are updated so the LSTM can determine what information needs to be retained in both the short and long term memory, based on the previously passed information and the current inputs. Note that the output prediction is taken from the hidden unit activations, not the context activations. In particular, the network must learn to retain whether the T or P input caused the transition into either of the simple Reber Grammar (RG) components, as otherwise the network will be unable to accurately predict this letter once it has reached the final state in the RG, that is, the state 9 or 17. In this instance, LSTM long term memory is performed by creating

semi-separable regions in $\mathbb{R}^4$, where context activations lie within either a T or P region for the given state of each RG.. Since the LSTM required as least four hidden nodes to be trained successfully, a static plot of the activations cannot be provided as was in provided in Part 4 Question 1. However, the output of the context and hidden layer activations describes how the LSTM predicts the ERG by learning separable hidden unit activations in $\mathbb{R}^4$, with decision boundaries determined by the output functions. Note that the prediction for the first decision between either T or P is not of concern and purely probabilistic, rather, the retention of this input in the context layer is essential. Although the context layer activations are not directly used to determine the probability of the next character, that is, the functions at the output only consider the hidden unit activations, they still occupy an activation space in $\mathbb{R}^4$ the regions in which the context activations lie allow the LSTM to retain the T or P input. Information retention in either the hidden or context layer does not mean that components of either output vectors will be similar to the previous time-steps, rather, the regions in which these different activations lie will be similar if the information is important, thereby allowing the LSTM to accurately predict the next state. Inside the LSTM, the current input, $\mathbf{x}_t$, and the previous context and hidden layer, $\mathbf{c}_{t-1}$ and $\mathbf{h}_{t-1}$, are taken in. To begin, the amount degree of information retention in the context layer is determined by the previous hidden layer and current input in,

$$\mathbf{f}_t = \sigma(U_f \mathbf{h}_{t-1} + W_f \mathbf{x}_t + \mathbf{b}_f),$$

where the values of $\mathbf{f}_t$ range from 0 to 1 and the vector $\mathbf{f_t}$ is multiplied with $\mathbf{c_{t-1}}$ component wise [2]. Following, the LSTM implements the activations $\mathbf{i}_t$ and $\mathbf{g}_t$ that follow,

$$\mathbf{i}_t = \sigma(U_i \mathbf{h}_{t-1} + W_i \mathbf{x}_t + \mathbf{b}_i),$$

$$\mathbf{g}_t = \tanh(U_o \mathbf{h}_{t-1} + W_o \mathbf{x}_t + \mathbf{b}_o),$$

where the sigmoid activation determines the amount of short range data should be implemented into the context layer from 0 to 1, an the tanh activation provides these values in a range of -1 to 1 [2] i. Finally, this new context activation not only serves as input for the next time step, but is also combined with a sigmoid activation $\mathbf{o}_t$ where,

$$\mathbf{o}_t = \sigma(U_o \mathbf{h}_{t-1} + W_o \mathbf{x}_t + \mathbf{b}_o),$$

which determines the extent to which the current context unit will affect the hidden units activations, based on the previous hidden activation and the current input [2]. Finally the current context and hidden activations are given by,

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t$$

$$\mathbf{h}_t = \mathbf{o_t} \odot \tanh(\mathbf{c}_t)$$

and both serve as input for the next time-step while the hidden activation $\mathbf{h}_t$ is also used to compute the output probabilities [2], which in the case of this LSTM is given by `log_softmax`. Unsurprisingly,

the context and hidden layers are the same for state 1, as the ERG always begins with B, with activations,

$$\mathbf{c}_0 = \begin{bmatrix} -0.77 \\ -0.97 \\ 0.97 \\ -0.95 \end{bmatrix}, \quad \mathbf{h}_0 = \begin{bmatrix} -0.63 \\ -.075 \\ 0.75 \\ -0.9 \end{bmatrix}.$$

For sequences with an initial T input the follow internal activations are consistent at state 2,

$$\mathbf{f}_1 = \begin{bmatrix} 0.8 \\ 0.37 \\ 0.32 \\ 1 \end{bmatrix}, \quad \mathbf{i}_1 = \begin{bmatrix} 0.9 \\ 0.97 \\ 0.98 \\ 0.93 \end{bmatrix}, \quad \mathbf{g}_1 = \begin{bmatrix} -0.61 \\ 0.78 \\ 0.81 \\ -0.85 \end{bmatrix}, \quad \mathbf{o}_1 = \begin{bmatrix} 0.94 \\ 0.99 \\ 0.51 \\ 0.98 \end{bmatrix},$$

Sequences that must retain that a P was inputted initially also have consistent internal activations at state 10

$$\mathbf{f}_1 = \begin{bmatrix} 0.91 \\ 0.05 \\ 1 \\ 0.97 \end{bmatrix}, \quad \mathbf{i}_1 = \begin{bmatrix} 0.97 \\ 0.99 \\ 0.97 \\ 0.76 \end{bmatrix}, \quad \mathbf{g}_1 = \begin{bmatrix} -0.36 \\ 0.98 \\ -0.25 \\ 0.13 \end{bmatrix}, \quad \mathbf{o}_1 = \begin{bmatrix} 0.96 \\ 0.98 \\ 0.45 \\ 0.86 \end{bmatrix},$$

Therefore, the following $\mathbf{f}_t \odot \mathbf{c}_{t-1}$ activations are obtained for states 2 and 10,

$$\mathbf{f}_1^2 \odot \mathbf{c}_0^2 = \begin{bmatrix} -0.62 \\ -0.36 \\ 0.31 \\ -0.95 \end{bmatrix}, \quad \mathbf{f}_1^{10} \odot \mathbf{c}_0^{10} = \begin{bmatrix} -0.7 \\ -.05 \\ 0.97 \\ -0.92 \end{bmatrix}.$$

The euclidean distances, $D(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_i (x_i - y_i)^2}$, between these altered context activations and the shared original are $D(\mathbf{c}_0^2, \mathbf{f}_1^2 \odot \mathbf{c}_0^2) = 0.91$ and $D(\mathbf{c}_0^{10}, \mathbf{f}_1^{10} \odot \mathbf{c}_0^{10}) = 0.92$ units, therefore implying that the LSTM has learned weights that shift the activation substantially when processing the first letter choice T or P. The equal shift also denotes that this is not a random occurrence, but a learned shift in activations. Furthermore, the distance between these partially formulated context activations is $D(\mathbf{f}_1^2 \odot \mathbf{c}_0^2, \mathbf{f}_1^{10} \odot \mathbf{c}_0^{10}) = 0.73$, so the LSTM has begun to separate the two embedded RG state activations in $\mathbb{R}^4$, as doing so will allow the LSTM to retain memory of which letter was inputted first, as the equivalent states in each RG will appear in different regions of $\mathbb{R}^4$, thereby allowing it to accurately predict which letter was inputted in order to escape from each embedded RG. Following, the learned weights increase the distance in the context layer since,

$$\mathbf{i}_1^2 \odot \mathbf{g}_0^2 = \begin{bmatrix} -0.55 \\ 0.76 \\ 0.79 \\ -0.79 \end{bmatrix}, \quad \mathbf{i}_1^{10} \odot \mathbf{g}_0^{10} = \begin{bmatrix} -0.35 \\ 0.97 \\ -0.24 \\ 0.1 \end{bmatrix}.$$

which gives output context layer activations of,

$$\mathbf{c}_1^2 = \begin{bmatrix} -1.17 \\ 0.4 \\ 1.11 \\ -1.74 \end{bmatrix}, \quad \mathbf{c}_1^{10} = \begin{bmatrix} -1.05 \\ 0.93 \\ 0.72 \\ -0.83 \end{bmatrix},$$

which have separation $D(\mathbf{c}_1^2, \mathbf{c}_1^{10}) = 1.13$. This confirms that the LSTM has learned weights that separate the context activations of the two RGs. It is interesting to note the context activations of the two RGs as a whole may not be linearly separable in $\mathbb{R}^4$, but since the context activations allow the LSTM is able to remember the inputted T or P letter, only corresponding states inside the two embedded RGs need to be distinct in the hidden unit space, for example, states 5 and 13 or 8 and 16 would be indistinguishable from each other without the context activations learning the long range dependency on the initial letter [2]. The context layer allows the LSTM to determine which of the two embedded RGs it is currently in based on the activation of the current state. Taking the context activations and using a simple perceptron model, it is observed that after 10,000 iterations the perceptron has failed to converge, but it able to separate upwards of approximately 50 95% of the activations, where the two classes are those activations in the RG that has an initial T and the other RG that stems from P. Yet this inseparability is of no concern, as the activations for the corresponding states between the two embedded RGs remain separate. Applying the euclidean distance to these corresponding average context activations yields the output in Figure 13.

```
dist 2->10: 1.2623496055603027
dist 3->11: 1.3534919023513794
dist 4->12: 2.736585855484009
dist 5->13: 2.046635866165161
dist 6->14: 0.7834059000015259
dist 7->15: 0.6447969079017639
dist 8->16: 0.9962063431739807
dist 9->17: 0.888801634311676
```

Figure 13: Average euclidean distance between corresponding states in the ERG task

Running the ten test epochs against the trained LSTM yields similar results for different levels of separability of the context activations of the two embedded RGs. Evidently, the context activations a separated in $\mathbb{R}^4$, and as such, the LSTM has learned weights that retain the information of which letter, T or P, was inputted first in the sequence in the context layer. While inside the either RGs the states shift their position, similarly to the shifts seen in Part4 Question 1, albeit in $\mathbb{R}^4$, but the context activations between corresponding states remain distant. By comparing the current input and previous hidden activation the LSTM is able to determine the sequence within the embedded RG, but it uses the separate context activations to determine which letter allows the LSTM to 'escape' the embedded RG and successfully complete the task.

References

[1] Alan Blair. *Deeper Networks*. URL: https://edstem.org/au/courses/7036/lessons/14961/slides/107699. (accessed: 25.10.21).

[2] Alan Blair. *Long Short Term Memory*. URL: https://edstem.org/au/courses/7036/lessons/14966/slides/107728. (accessed: 24.10.21).

[3] Alan Blair. *Perceptron Learning Algorithm*. URL: https://edstem.org/au/courses/7036/lessons/14957/slides/107677. (accessed: 25.10.21).

[4] Alan Blair. *Recognizers and Predictors: Hidden unit dynamics for $a^n b^n c^n$*. URL: https://edstem.org/au/courses/7036/lessons/14965/slides/107725. (accessed: 24.10.21).

[5] Alan Blair. *Recognizers and Predictors: Non-Regular Languages*. URL: https://edstem.org/au/courses/7036/lessons/14965/slides/107725. (accessed: 21.10.21).

[6] Alan Blair. *Weight Decay and Momentum*. URL: https://edstem.org/au/courses/7036/lessons/14960/slides/107694. (accessed: 22.10.21).

[7] Christopher Olah. *Understanding LSTM Networks*. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs/. (accessed: 23.10.21).