ECE324 Assignment 2
Benjamin Cheng – 1004838045

Section 3
For brevity I will depict the weights w_i as a matrix in the following form:

$$W = \begin{bmatrix} w_0 & w_1 & w_2 \\ w_3 & w_4 & w_5 \\ w_6 & w_7 & w_8 \end{bmatrix}$$

i.

$$W = \begin{bmatrix} 0.2 & 0 & 0.2 \\ 0 & 0.2 & 0 \\ 0.2 & 0 & 0.2 \end{bmatrix} \qquad b = -0.5$$

ii. The answer is not unique, another possible set of weights and bias is:

$$W = \begin{bmatrix} 0 & -0.25 & 0 \\ -0.25 & 0 & -0.25 \\ 0 & -0.25 & 0 \end{bmatrix} \qquad b = 0.5$$

ii. Each input I_i has two possible values and there are 9 such inputs. This gives $2^9 = 512$ unique inputs.

iii. To find a single X-type pattern in a 4x4 grid the solution is similar. In fact it scales to any $N \times N$ grid. If you let M be the number of inputs that must be 1 in the X pattern, the weights corresponding to the inputs that need to be 1 should be set to $1/M$, and the weights corresponding to inputs that need to be 0 should be set to zero. Then setting $b = -0.5$ creates a classifier that is 100% accurate.

iv. Creating the single-neuron parameters to solve a 5x5 grid with translations is not as simple. In the prior parts to this question there was only one input that returned a positive result. This meant that I could construct the parameters by looking at solely making the one input return a positive result. However, the addition of the translations creates multiple inputs, which complicates this "solution by inspection" greatly. I suppose it is possible for me to create, however I would not do so willingly.

Section 6

Epoch Variation
The following tests were performed with the linear activation function and a learning rate of 0.001.

Epochs	**Final Training Accuracy**	**Final Validation Accuracy**	**Description**
5	0.63	0.55	The training and validation loss appear to be approaching zero, but the accuracy curves do not appear to be following any trend.
10	0.945	0.90	The training and validation loss appear to be flattened out at a value close to zero. The accuracy curves shot up to ~0.90

			but don’t appear to have settled yet.
25	0.960	0.95	The loss curves have flattened and both accuracies appears to be flattening as well.
100	0.975	0.95	Both the loss curves have flattened. The training accuracy appears to be creeping up very slowly, but the validation accuracy has flattened out.

Learning Rate Variation
The following tests were performed with the linear activation function and 25 epochs.

Learning Rate	**Final Training Accuracy**	**Final Validation Accuracy**	**Description**
0.05	0.335	0.65	Both validation and training accuracy oscillate rapidly between 0.65 and 0.35. Note that these are complements are each other (1 – 0.65 = 0.35).
0.005	0.335	0.65	Accuracy curves appear to be the same as 0.05.
0.001	0.96	0.95	Accuracy curves are now converging, with the training accuracy slightly higher than the validation accuracy.
0.0005	0.95	0.90	Accuracy curves are still converging as in 0.001, but the accuracies appear lower than for 0.001.
0.00005	0.48	0.55	Accuracy curves appear to have not plateaued and are still climbing.
0.000005	0.335	0.35	Both accuracy curves appear to be a constant value. The validation accuracy is higher than the training accuracy.

Activation Function Variation
The following tests were performed with 100 epochs, a 0.001 learning rate.

Activation Function	**Final Training Accuracy**	**Final Validation Accuracy**	**Description**
ReLU	0.665	0.65	Due to the dying ReLU problem this network fails to train. After one update, the weights constantly create negative values which ReLU sets to zero, meaning the gradient is zero also. The training accuracy gets stuck at 0.665 immediately after the first epoch and the validation accuracy is constant at 0.65.
Sigmoid	0.98	0.95	Both accuracy curves look very similar, almost like the sigmoid function itself (coincidence). It starts off at a low accuracy and doesn’t change until about 35 epochs. It then

			grows quickly until it reaches its maximum value at which it plateaus again. The validation accuracy is higher than the training accuracy after they both surpass about 0.60.
Linear	0.975	0.95	The accuracy is initially very erratic, but quickly shoots up to close to the maximum value. After this, the accuracies are mostly stable, with the training accuracy increasing very slowly, but the validation accuracy does not change.

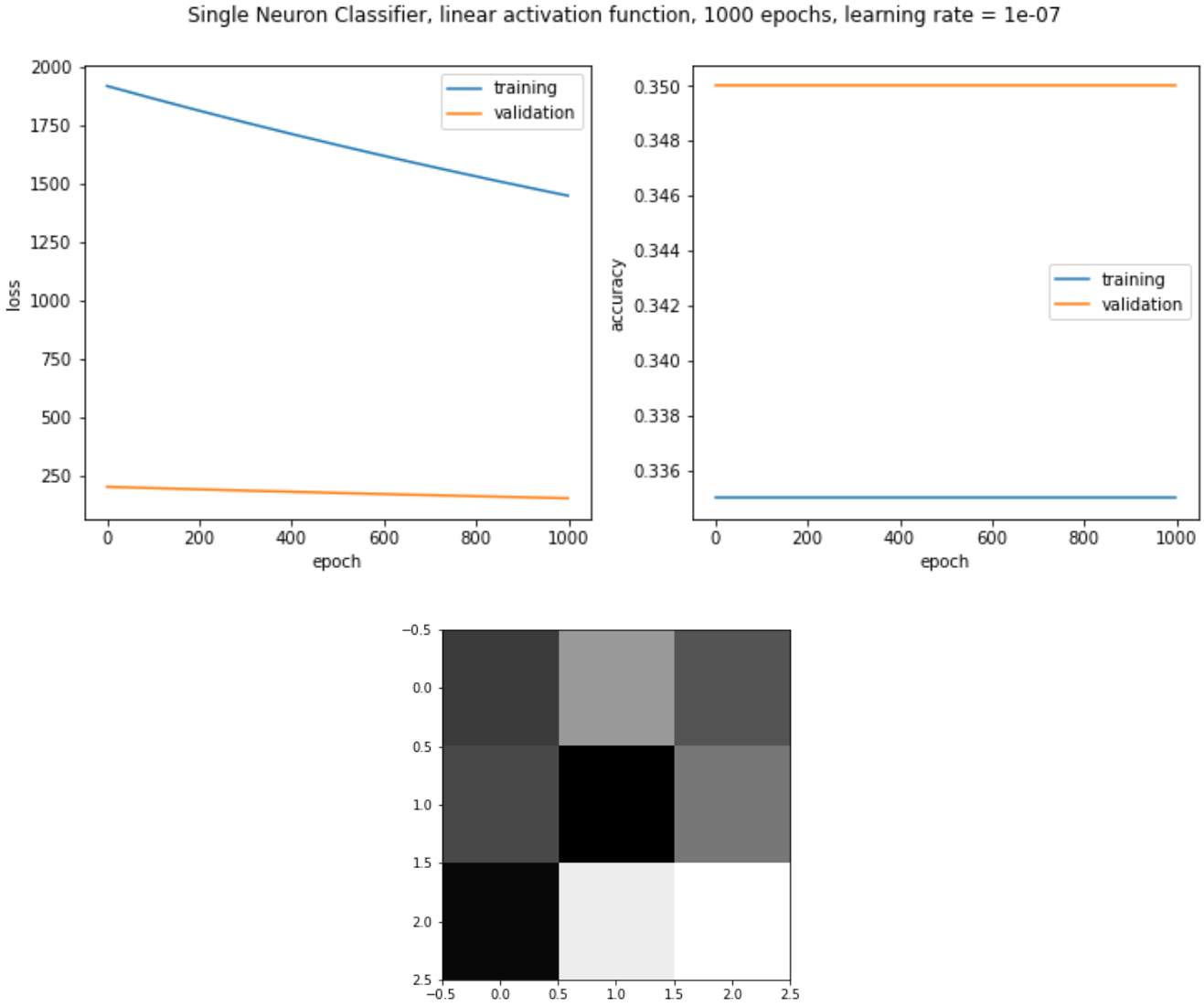
Random Seed Variation
The following tests were performed with 25 epochs, a learning rate of 0.001, and the linear activation function.

Random Seed	0	1	2	100	169
Final Training Accuracy	0.96	0.975	0.96	0.965	0.95
Final Validation Accuracy	0.95	0.90	1.00	0.90	0.95

The random seed determines the initial parameters of the neuron. However the value of the seed isn't porportional to the parameters in any way, as the parameters are still initiallized pseudorandomly. The difference in the initial parameters lead to different gradients being computed, which manifests itself in a different "pathway" down the gradient. We can see this in the different training and validation accuracies observed.

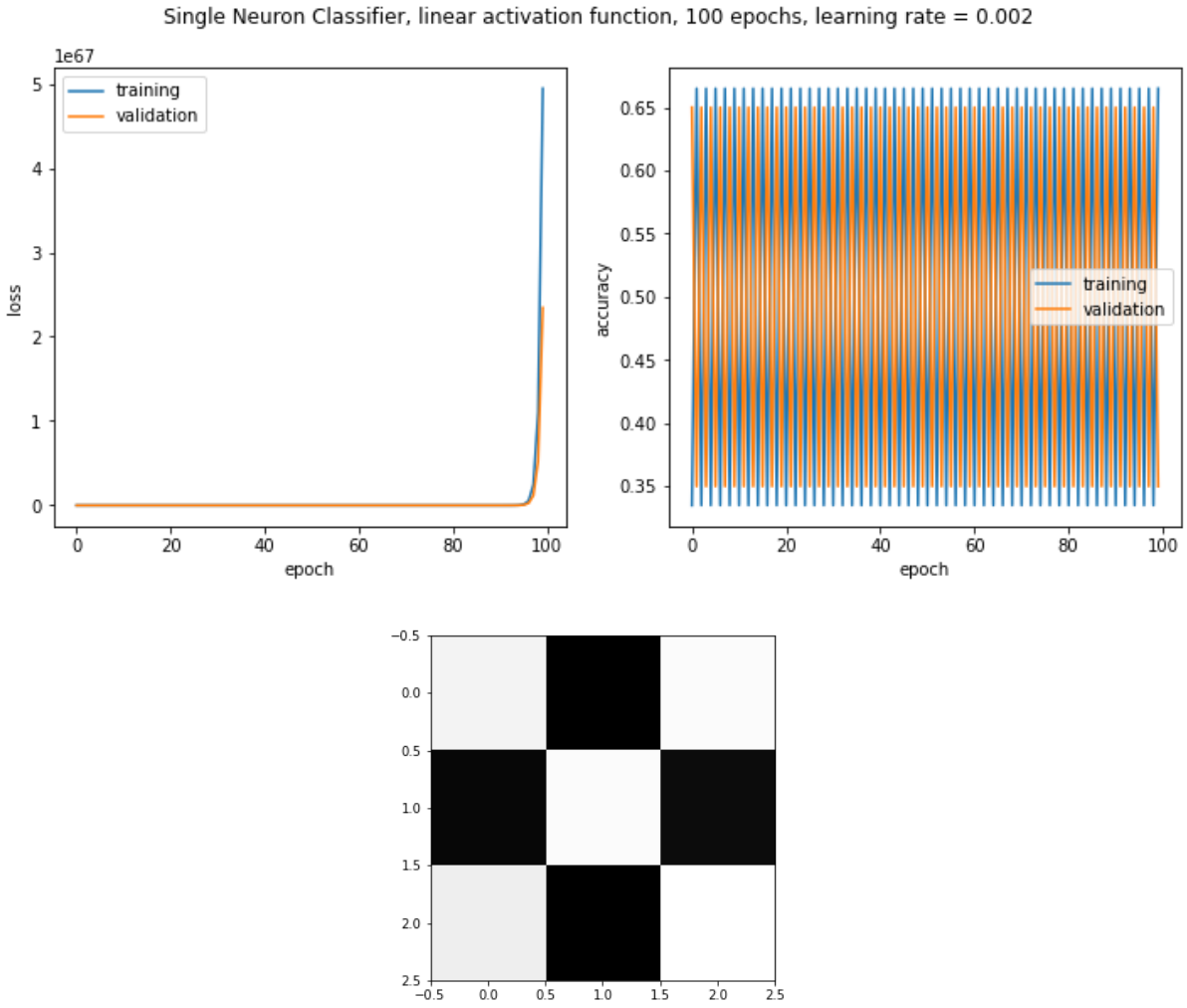
Best Set of Hyperparameters
I achieved a 99% training accuracy and 100% validation accuracy with the sigmoid activation function, a learning rate of 0.1, and only 10 epochs.

Learning Rate – Too Low



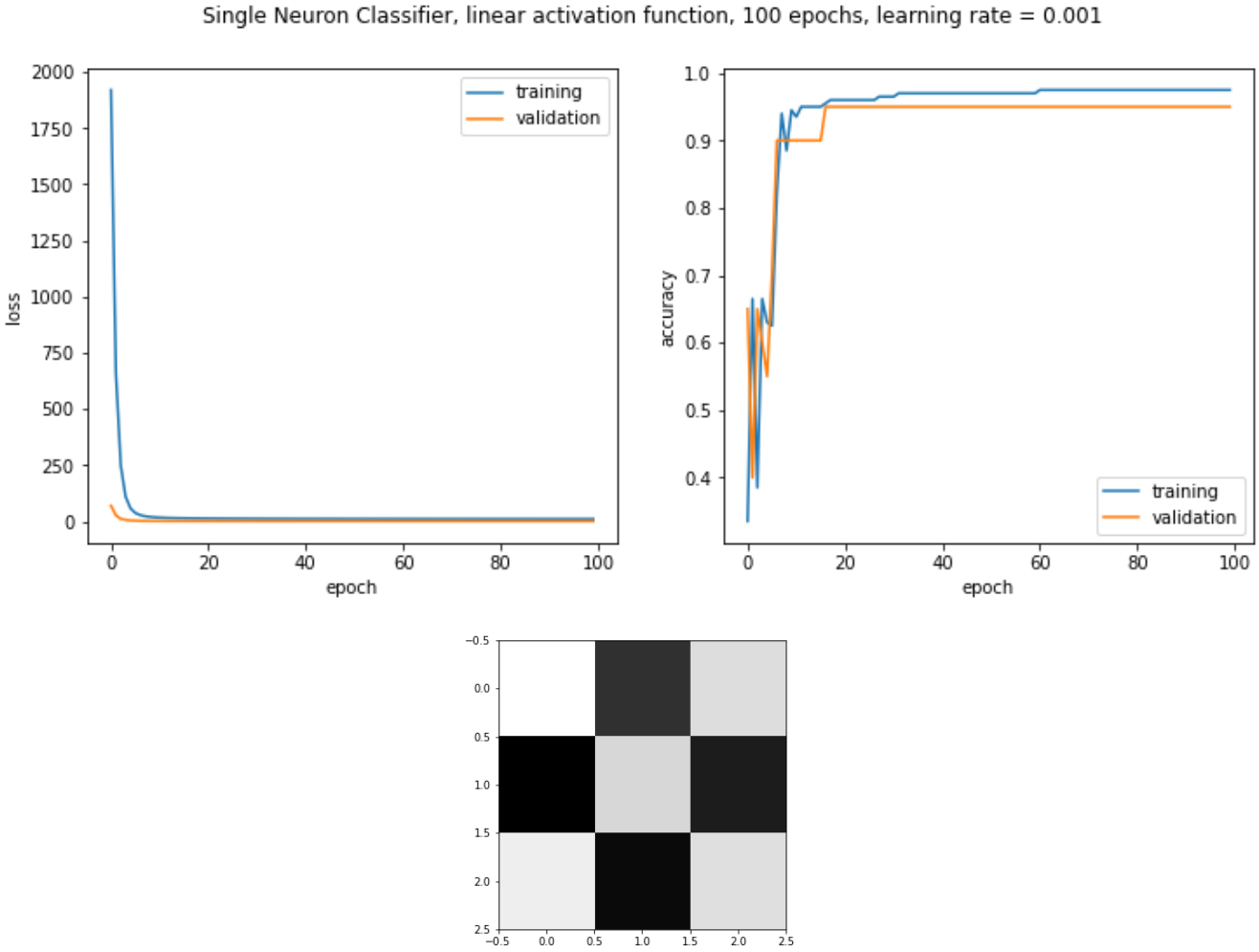
With this extremely low learning rate we can see that both losses are decreasing, with the validation loss decreasing at a slower rate. However the accuracy remains almost constant over 1000 epochs. This occurs because the loss is measuring how far the output is from the label. We are approaching closer to the desired label, the low learning rate is making the steps very small. Despite the output moving slighty closer, the parameters haven't crossed the 0.5 threshold needed to change the classification output, and thus the accuracy remains the same. The weight visualization shows that the weights don't resemble any specific pattern.

Learning Rate – Too High

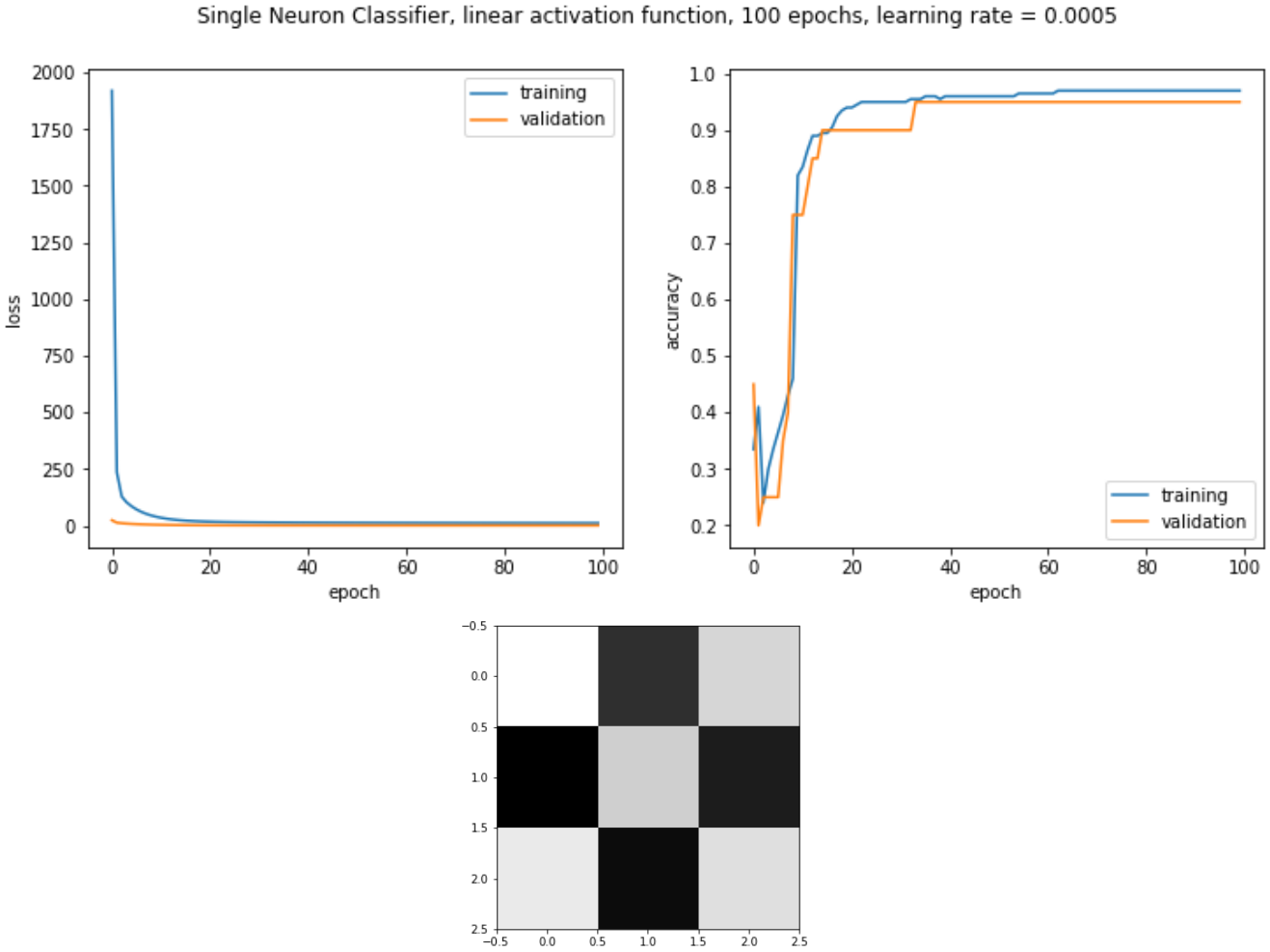


We can see the accuracy curves show rapid oscillations between a maximum and a minimum accuracy, and the losses explode exponentially. Although the weight visualization doesn't appear weird, printing out the weights reveals that the weights have exploded to values in the order of magnitude of 10^{32}. If we visualize the problem surface as a parabola, we start off at one side of the parabola. The gradient is computed and we move in that direction, but due to the high learning rate, we move so far that we run onto the other side, way past the minimum. The gradient is computed again and we move back, but because of the the high learning rate, we pass the original location. This causes us to be further from the minimum, thus increasing the gradient, and we continue in the loop with the gradients and therefore the weights exploding. This back and forth shows up in the accuracy as the oscillations.

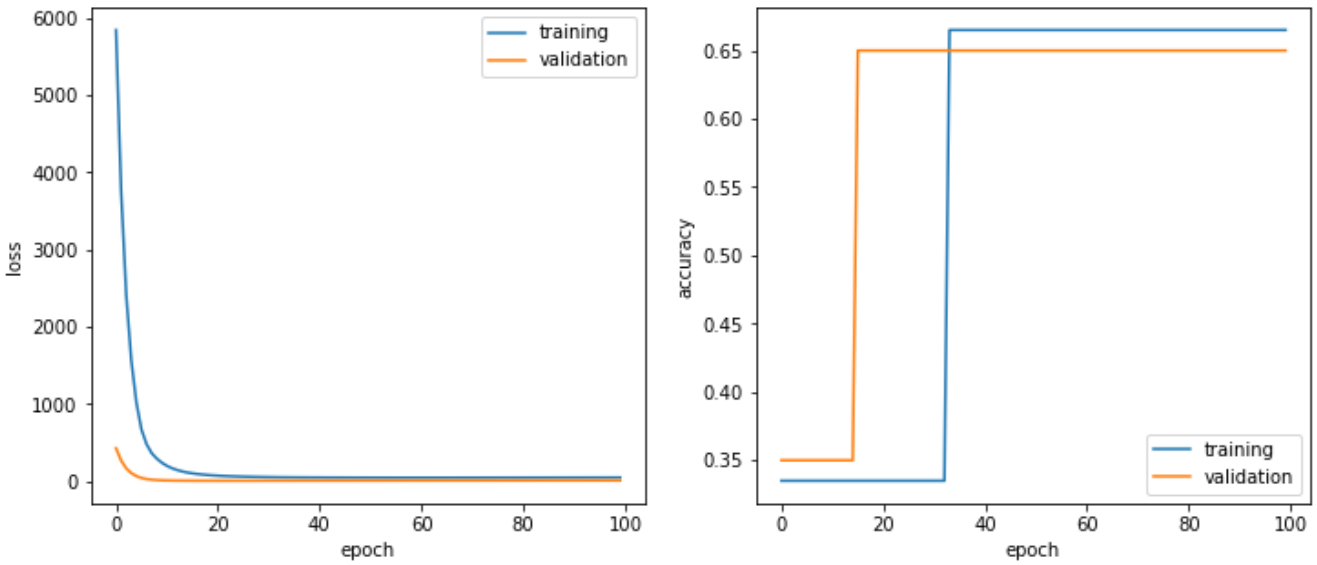
Learning Rate – Good

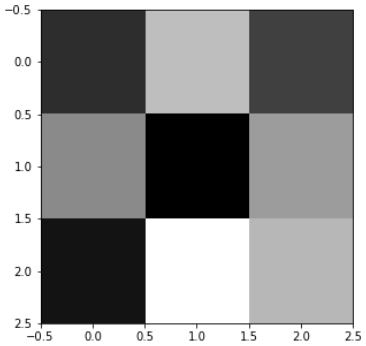


Linear Activation Function

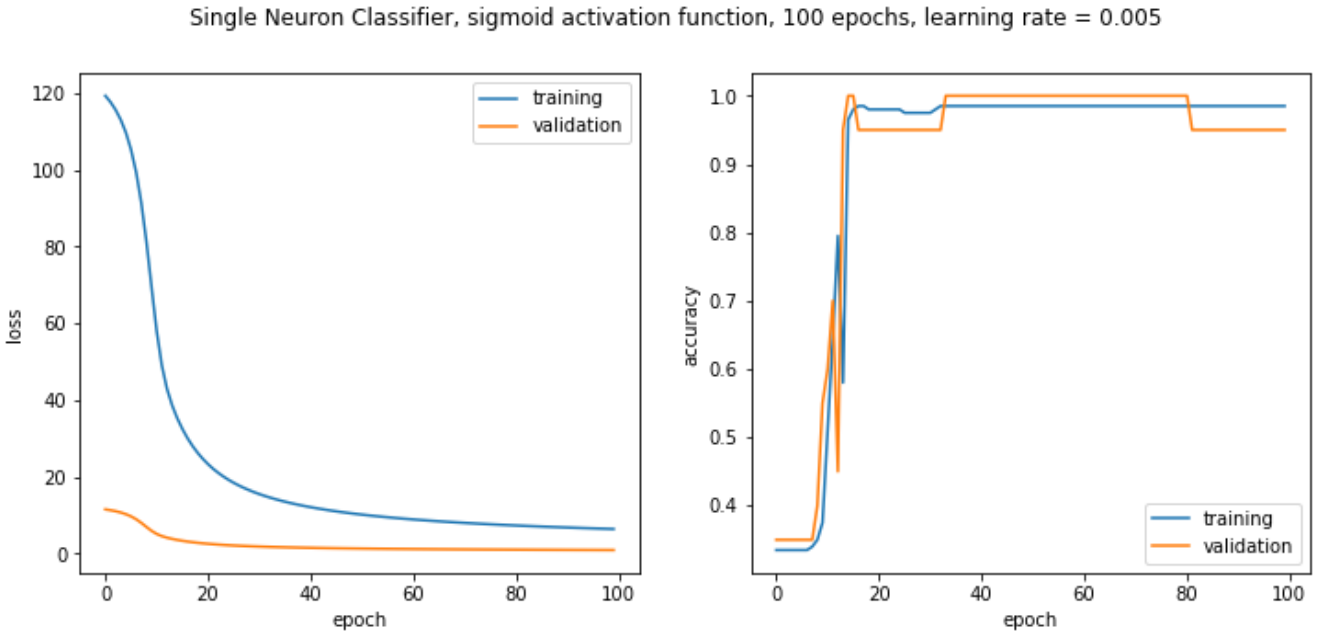


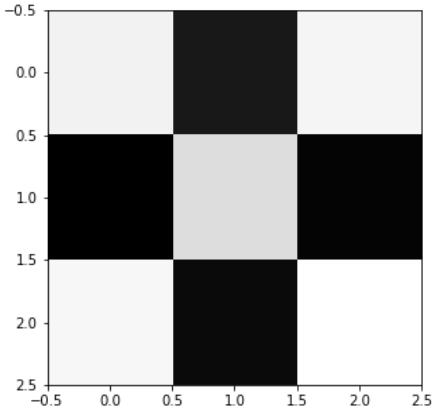
ReLU Activation Function





Sigmoid Activation Function



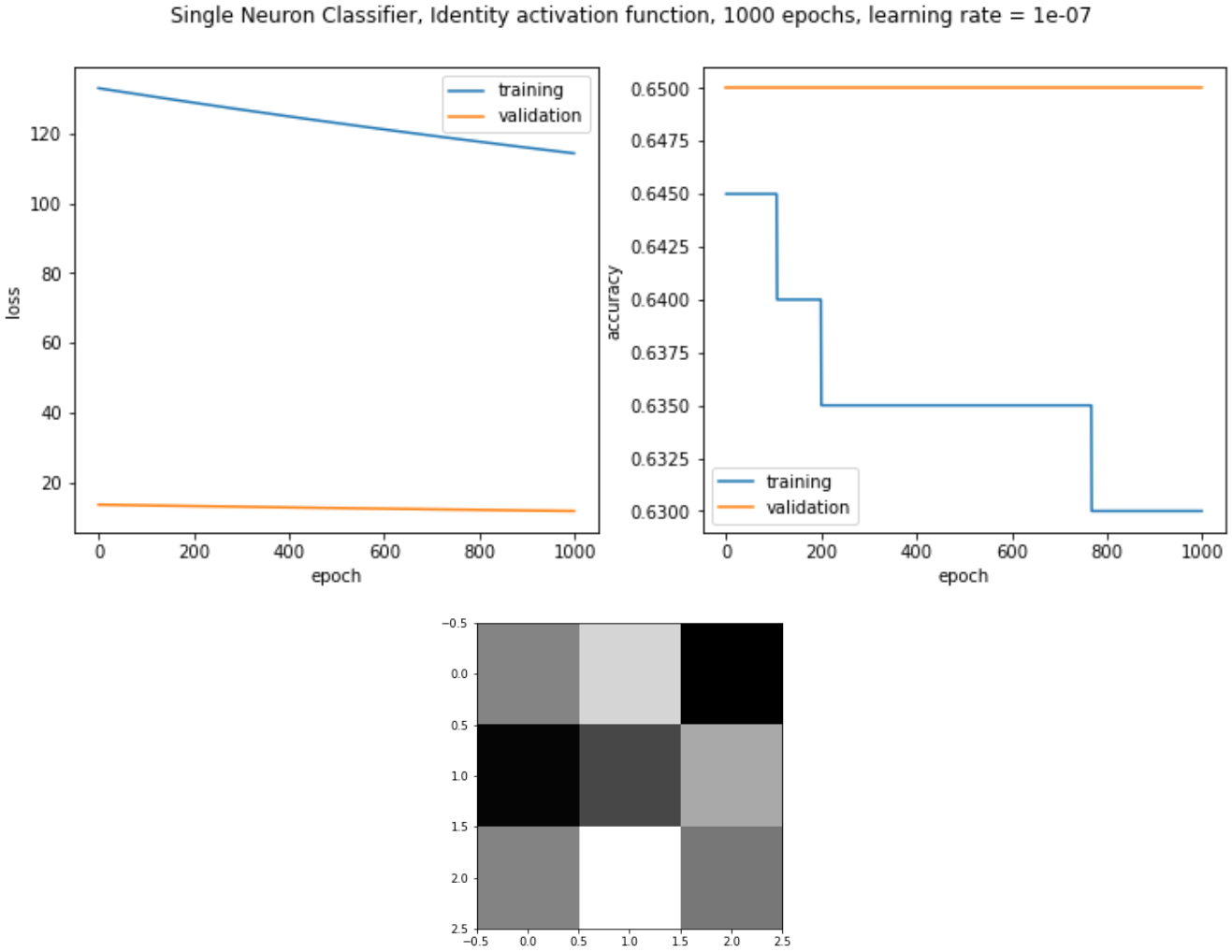


Section 7.2

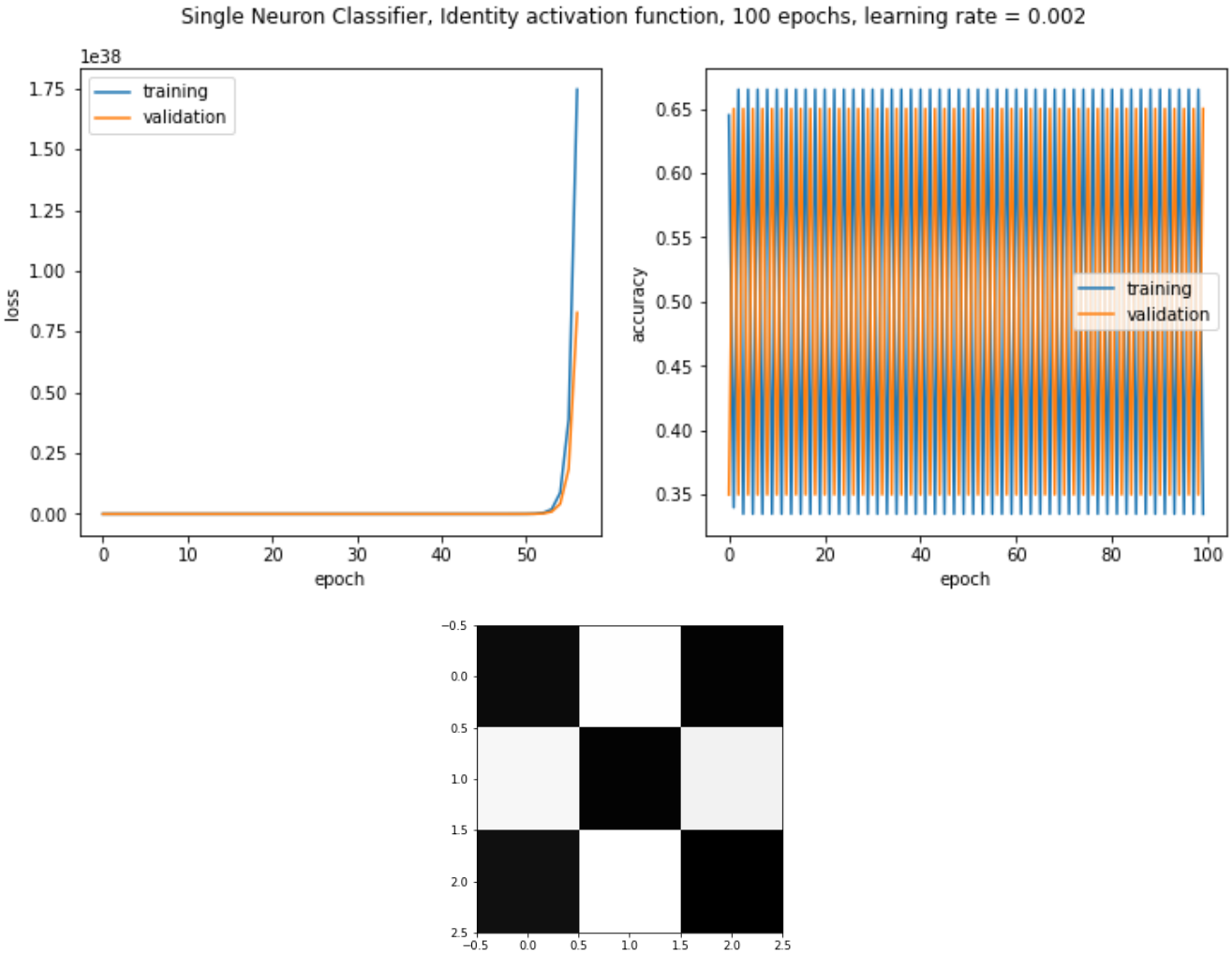
1. The `torch.nn.Linear` class contains all the weights as well as the bias for the single-neuron classifier in its `weights` and `bias` tensors respectively.

2. Every tensor, including the two mentioned in question 1 has a `grad` variable which is populated with backpropogation occurs. The gradient is stored in this variable in both the weight and bias tensors. The optimizer is passed `net.parameters()` which is a generator which yields the weights and bias tensors, letting the optimizer access these gradients.
3. The call to `loss.backward()` in the `train` function makes PyTorch do its backpropogation pass. Every tensor in PyTorch has a flag called requires_grad. When the model is in train mode (set by net.train()), PyTorch keeps track of the operations done on and values of each tensor with requires_grad set to True in a graph structure. When backward() is called, the gradients are computed using this graph. The storage of the values and operations allows PyTorch to calculate gradients for each operation indivdually, wrt the operations inputs. The graph structure allows PyTorch to intelligently chain these indivdual operations' gradients together to construct the gradient for the weights and biases.
4. The 3 cases are shown with the linear activation function (using `nn.Identity`) with 100 epochs, with the exception of the learning rate too low case. This is done to show how the learning rate is so small that it still even with 10x the epochs, no significant progress has been made.

PyTorch LR Too Low



PyTorch LR Too High



PyTorch Good LR