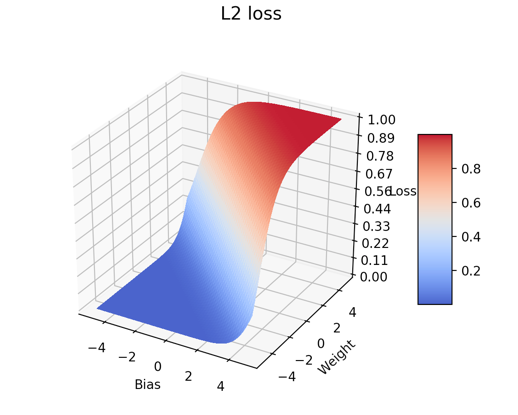
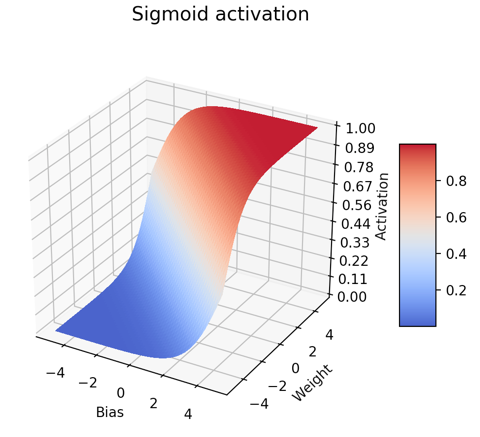
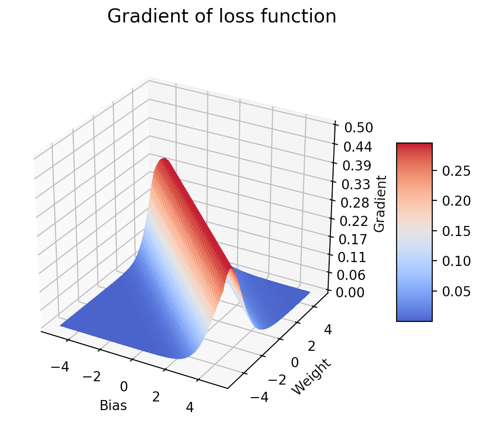
**Question 1**

The algorithm converges after 66 epochs. We find the weights as follows:

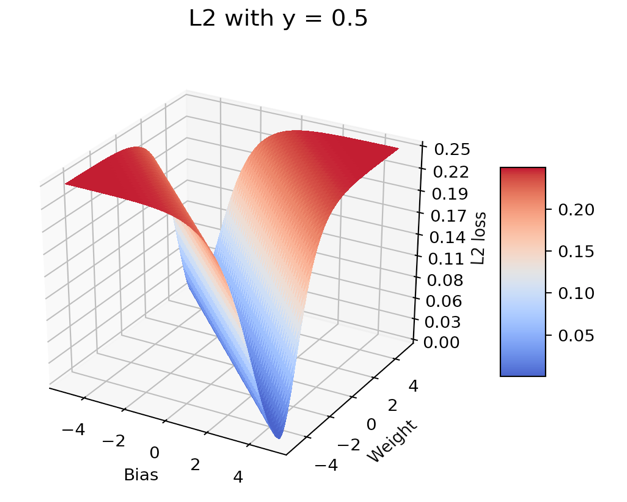
|  |  |
| --- | --- |
| **Sad images** | **Happy images** |
| -22, -10, -22, -30, -22, -22, -22, -17, -32, -27 | 26, 23, 26, 25, 13, 34, 17, 33, 19, 23 |

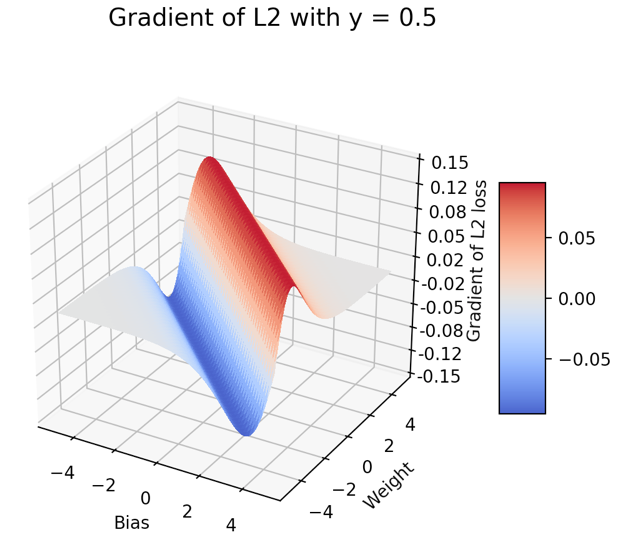
As expected, happy images have positive weights, while the sad images have negative weights. The weights are higher in absolute value for the images which the perceptron classified wrongly multiple times. Extremely recognizable images like 😓 which have a strong feature (blue droplet) in a location with low variance (top left corner) are assigned very low weights as they are easy to classify and the perceptron succeeds on them early on.

**Question 2**

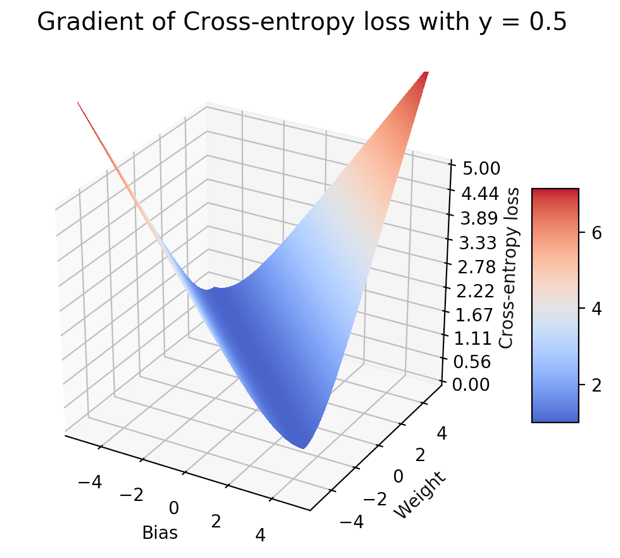
1. We plot the sigmoid function and its corresponding L2 loss, and the gradient of that loss, for x = 0, y = 0.

We notice that the sigmoid activation and L2-loss are very similar. In fact, since we take the label to be 0, the L2 loss is exactly the square of the sigmoid function. Meanwhile, the gradient graph demonstrates the diminishing gradient issue of the sigmoid function – learning maxes out at 0.5 when near the ambiguous region, and drops off dramatically when the label is very wrong / right.

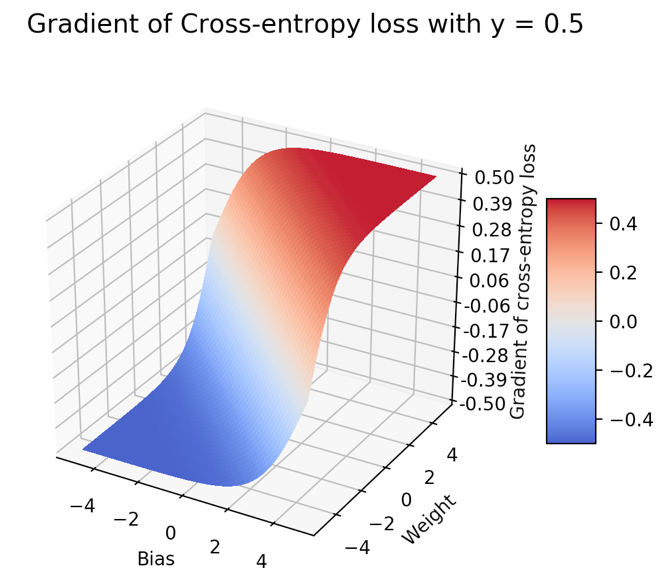
1. Now we set y = 0.5. We plot and observe a great dip when *wx + b* is approximately 0. Indeed, the sigmoid returns 0.5 when *wx + b = 0*. The loss function is symmetric around this axis, as expected.



1. Now we compute the gradient of that loss function found in (b), and see the same axis of symmetry.



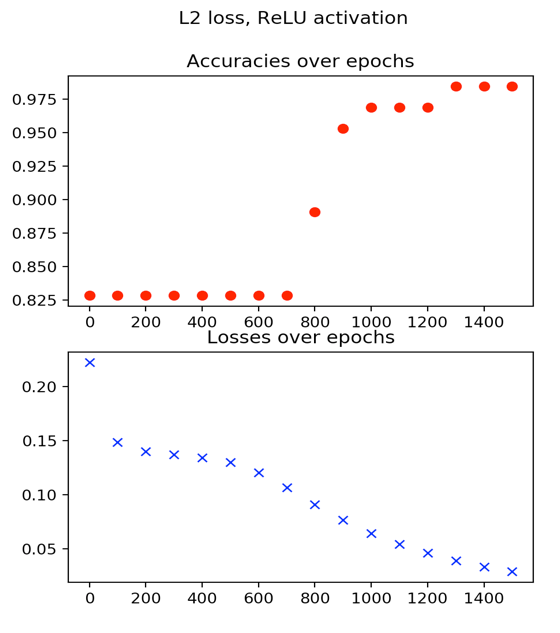
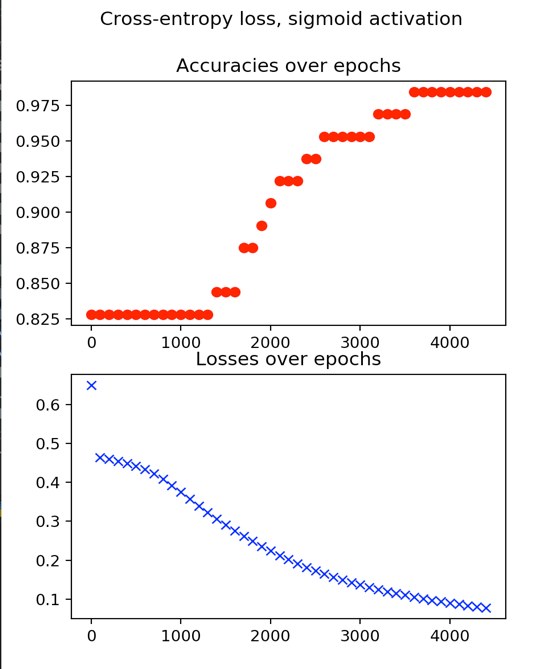
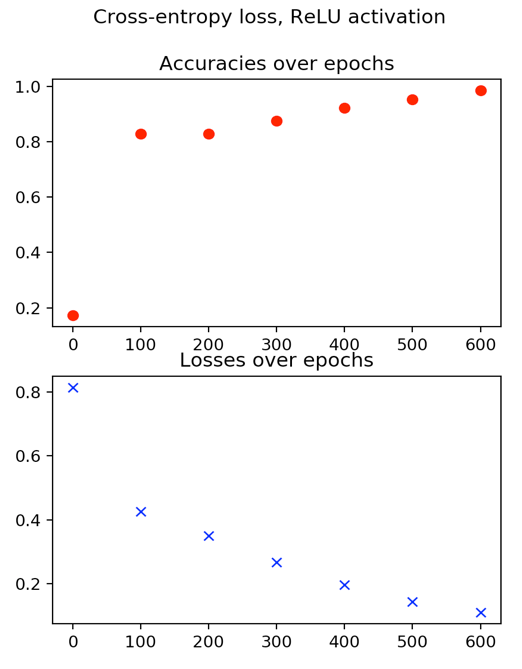
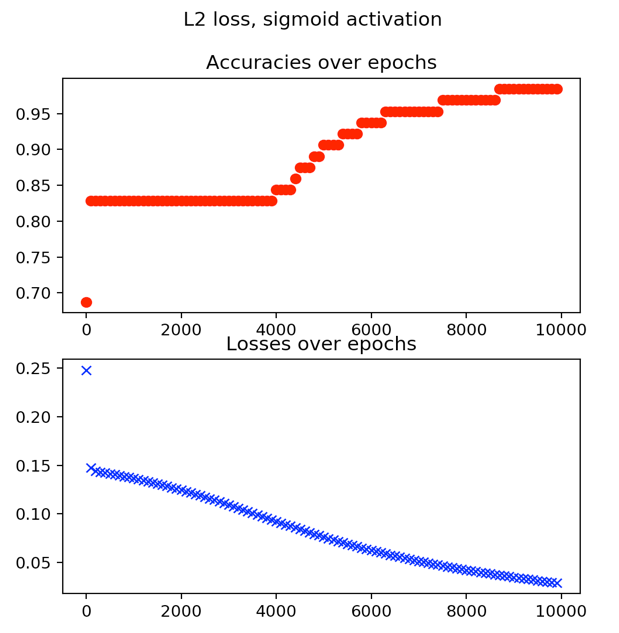
1. Now we analyze the cross-entropy loss function. We observe that it does not ever reach 0 for *y* = 0.5



1. We see that the gradient asymptotes to -0.5 or 0.5 as the error in prediction grows, while it penalizes very little for values closer to the correct label of 0.5.
2. The difference between the loss functions is that cross-entropy loss penalizes increasingly more the further away from the correct label we predict, whereas L2 loss peaks out very rapidly. This makes the gradient function very poor under L2 loss, since we have 2 big peaks near the true value, and suffer from diminishing gradient when our labels are very far from truth.

**Question 3**

We initialize with the respective functions and set it to terminate when accuracy hits 100%.

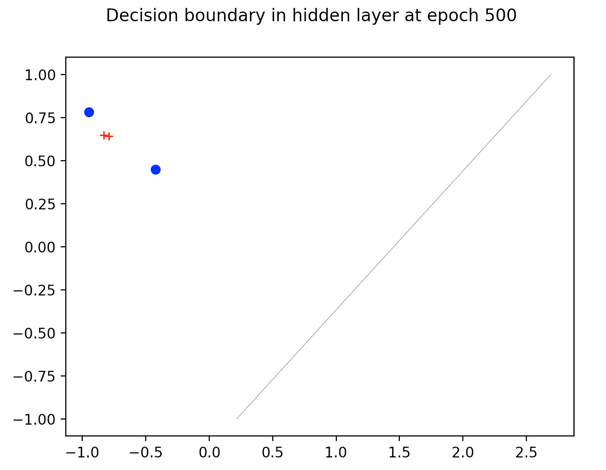
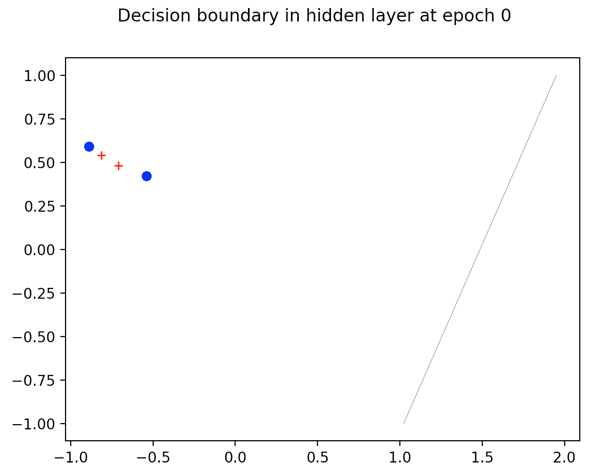


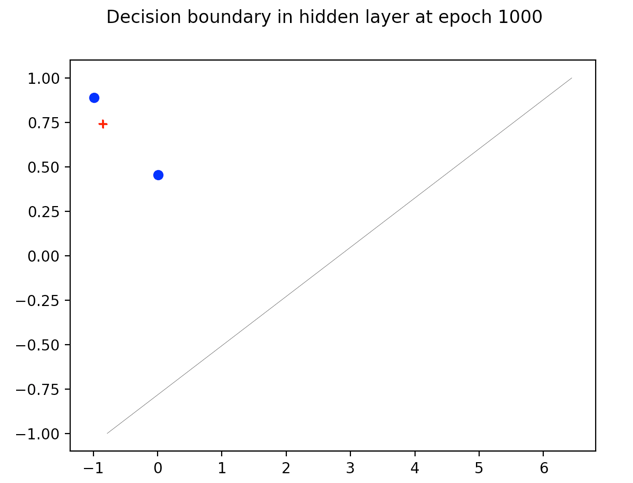
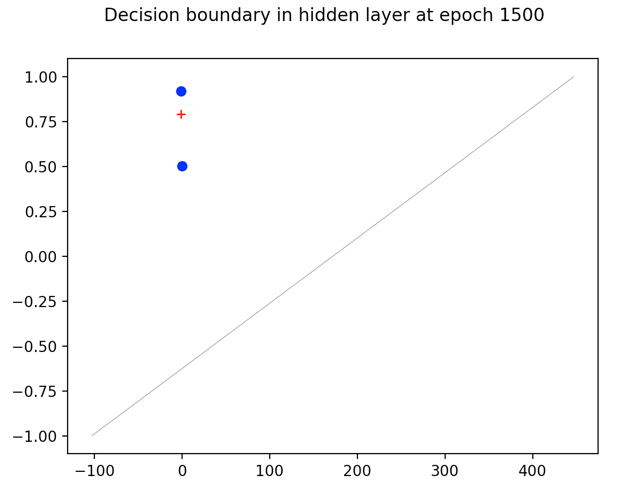
In terms of convergence rates, ReLU activation consistently out-performs sigmoid activation in the network. Moreover, cross-entropy loss allows the network to converge faster than L2-loss. This is as expected from Problem 2, where the diminishing gradient causes the sigmoid to converge slowly, and cross-entropy loss can penalize bad mistakes heavier than L2-loss.

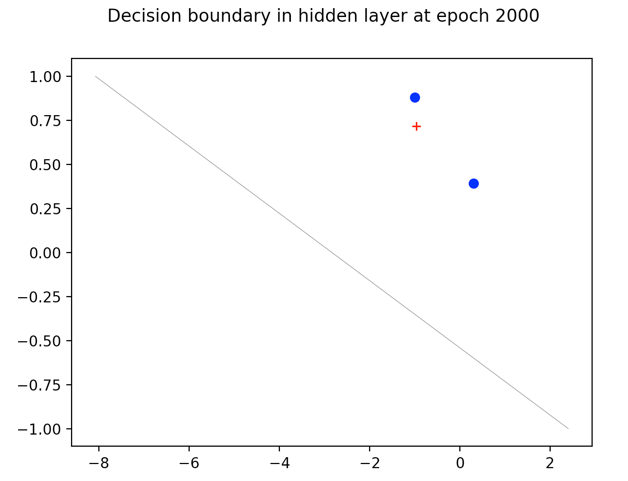
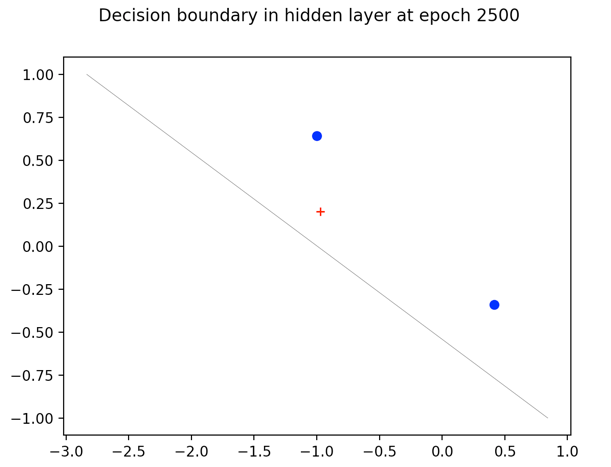
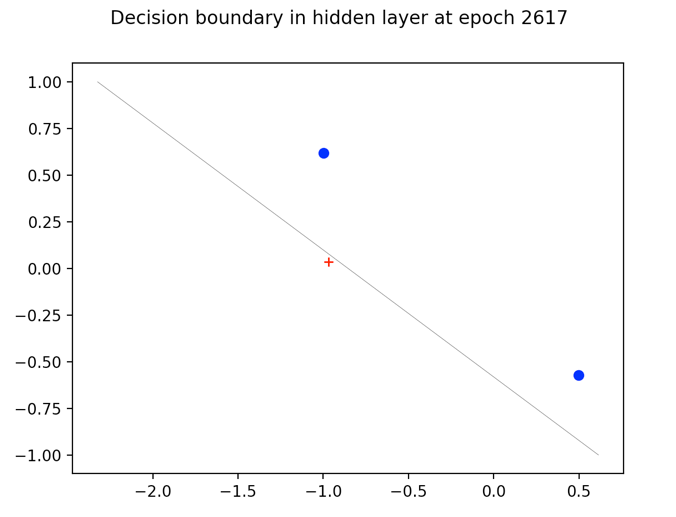
**Question 4**

1. On the dataset *X* = { (1, 0), (1, 1), (0, 1), (0, 0) }, *Y* = { 1, 0, 1, 0 }, the network output is given by:

where ***A*** is a 2 x 2 matrix and ***w*** is a 2 x 1 vector. Now, we seek to minimize:

1. We write a net and stop its iterations once it hits 100% accuracy.

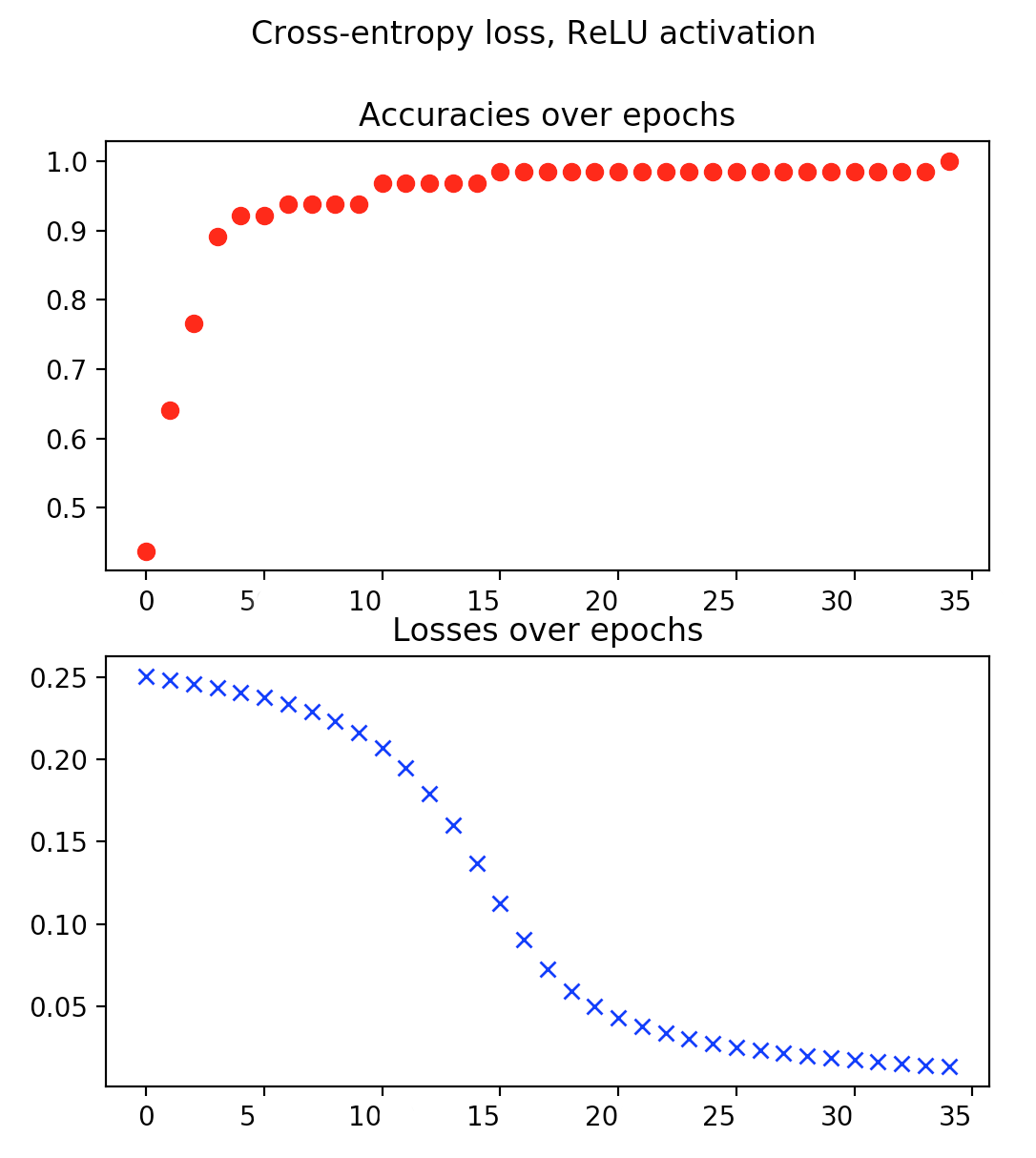
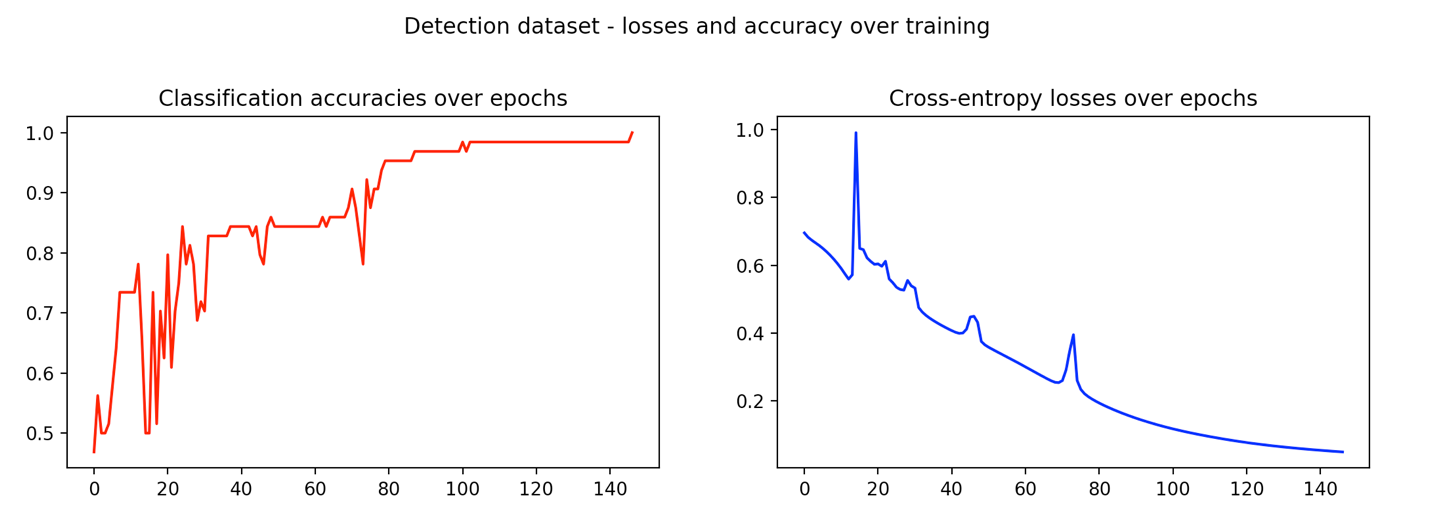




As can be seen, the net converges when the separating plane finally puts the +s on the opposite side of the –s.

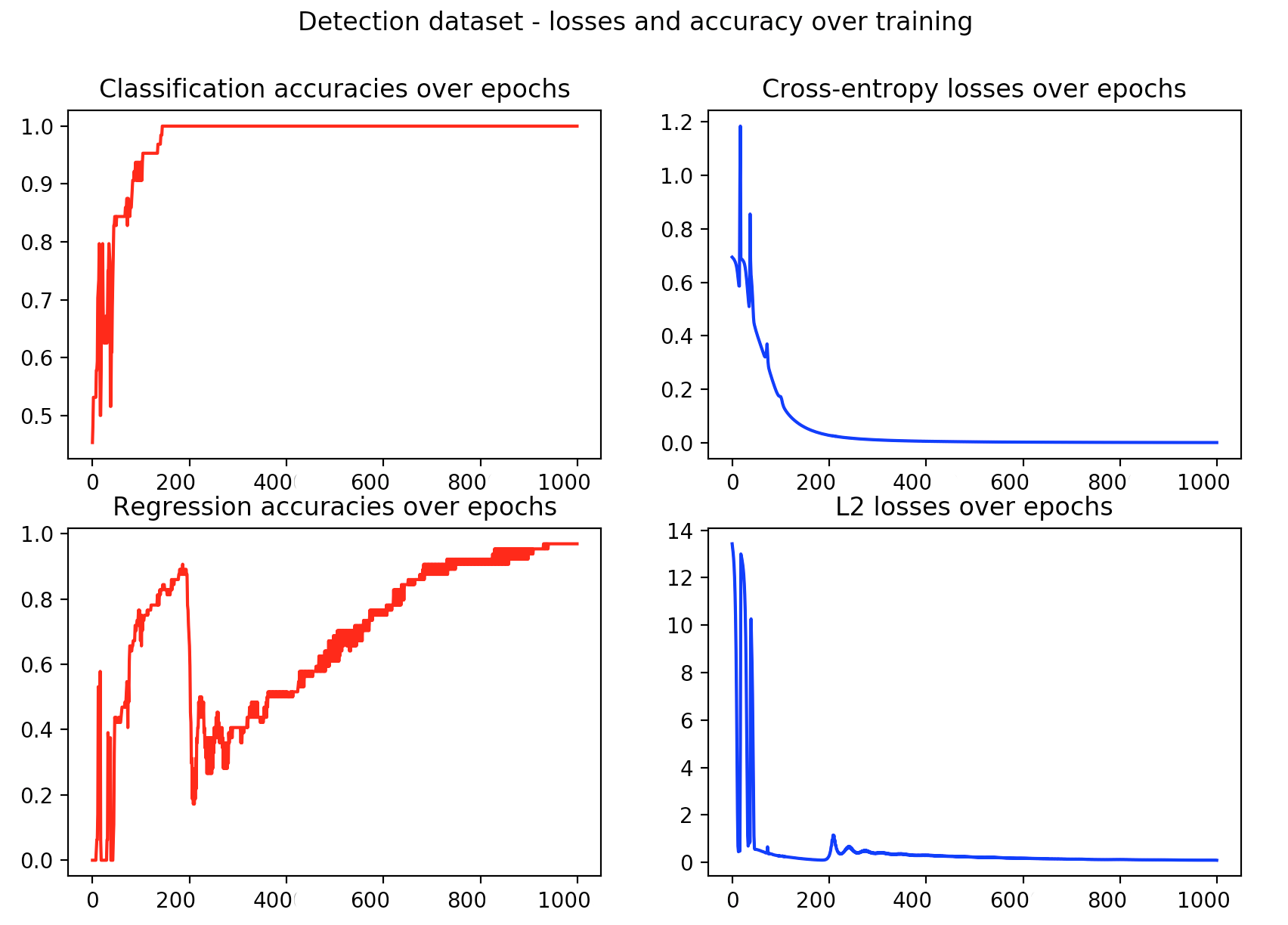
1. If we don’t use an activation function in the hidden layer, then the output will simply be a sigmoid of some linear combination of the inputs. This can never achieve 100% on the XOR, since we have not transformed the space ***x*** into a separable space ***h***.

**Question 5**

1. We run the network on the line dataset, and find that it converges relatively fast in 34 epochs.
2. We use the same network on the detection dataset, and find that it takes the same order of magnitude to converge, but slightly longer.

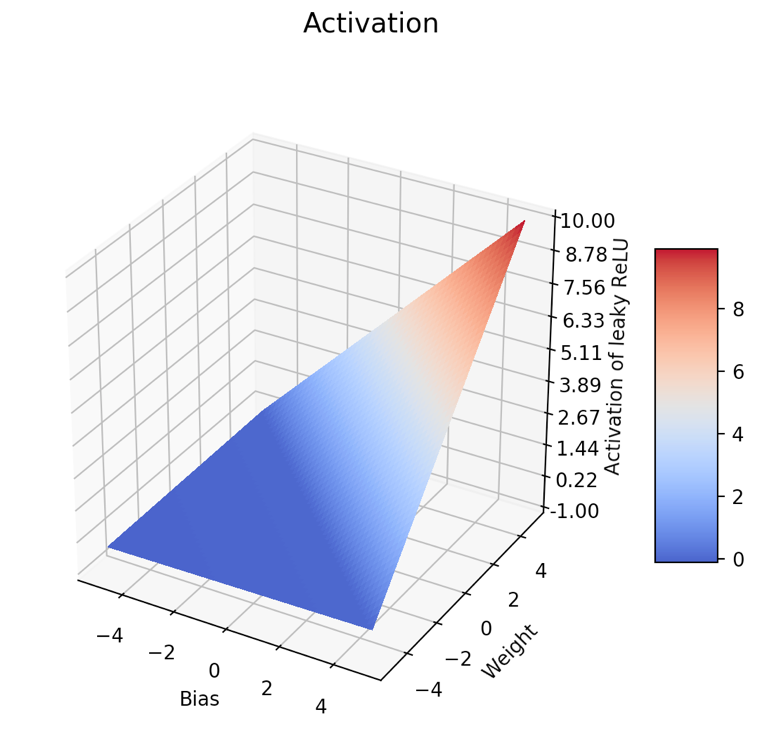
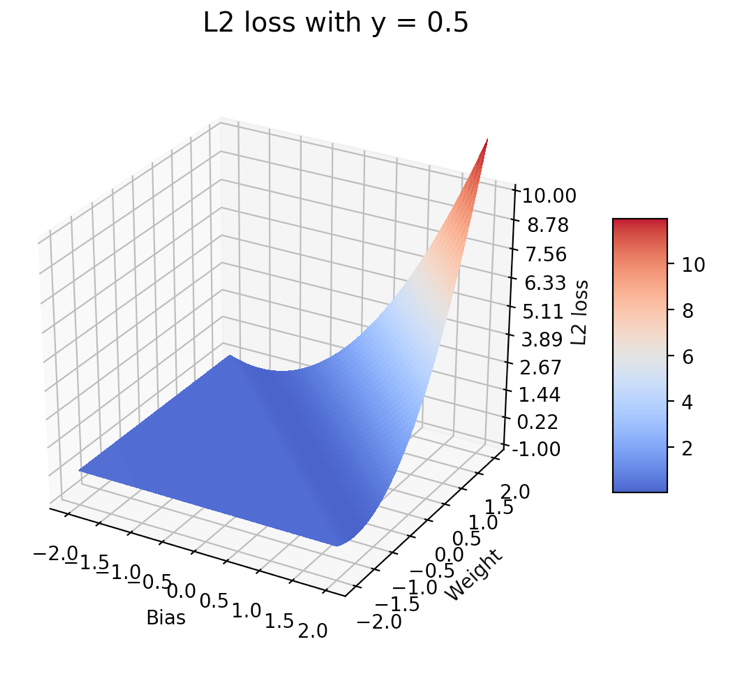
Now we add the additional linear output to a prediction on width, with a L2-loss on that output, and use back-propagation on the sum of both losses *L* = *L1 + 0.01 L2*

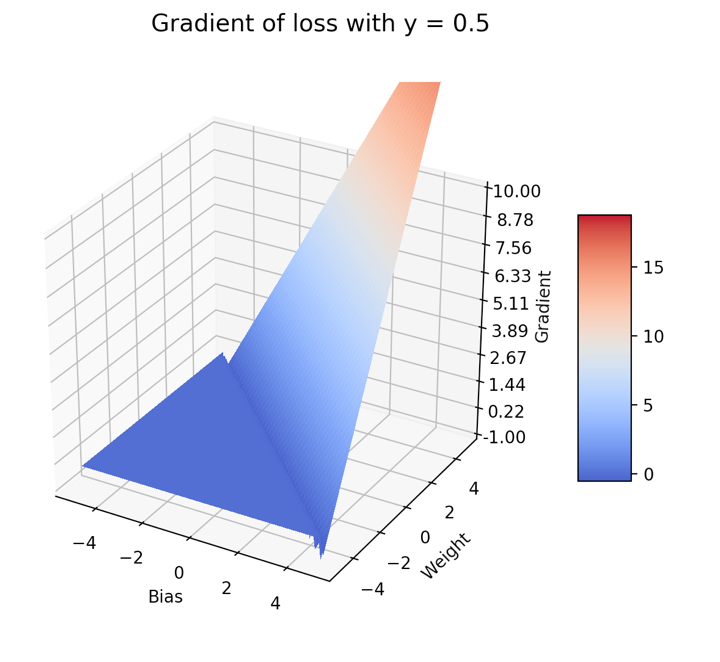
The classification takes slightly longer to converge, but basically the same amount of time. However, the width prediction is harder to learn, and takes approximately 1000 epochs.

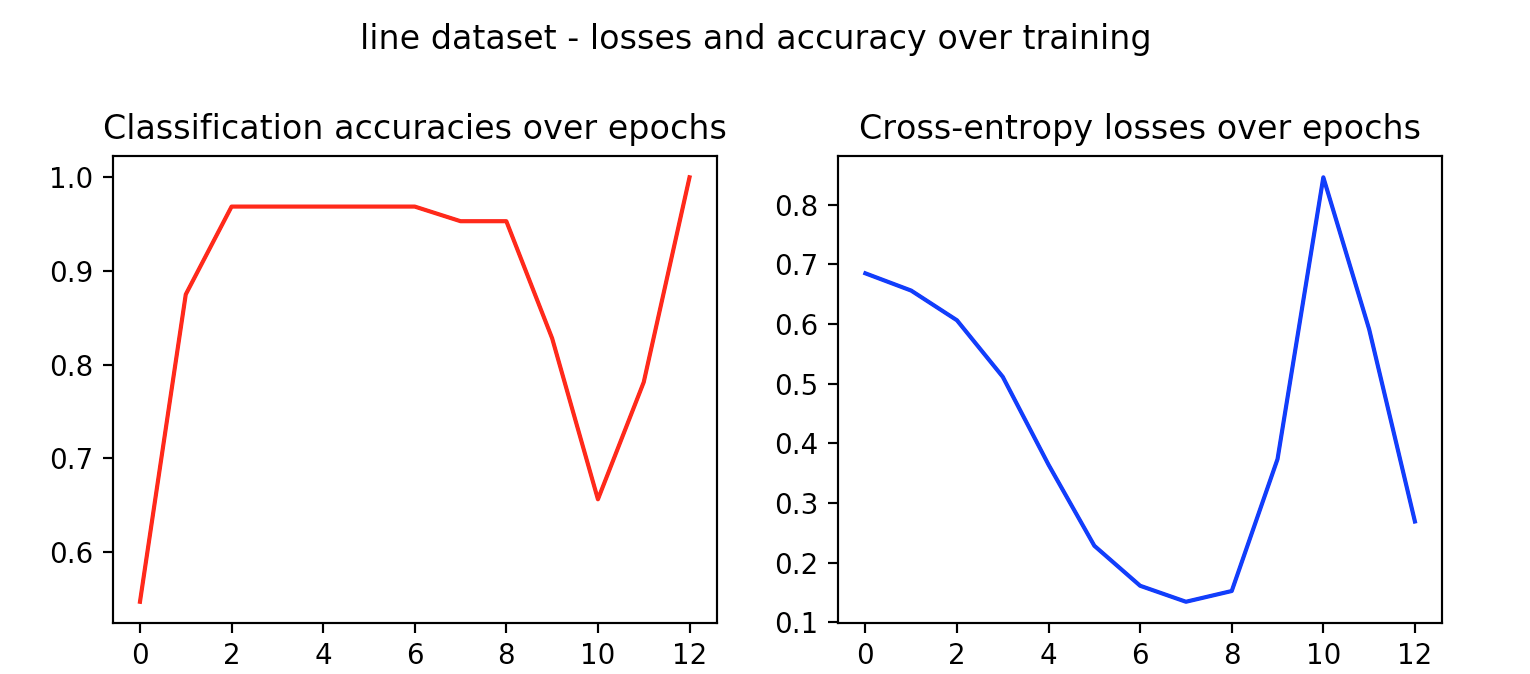
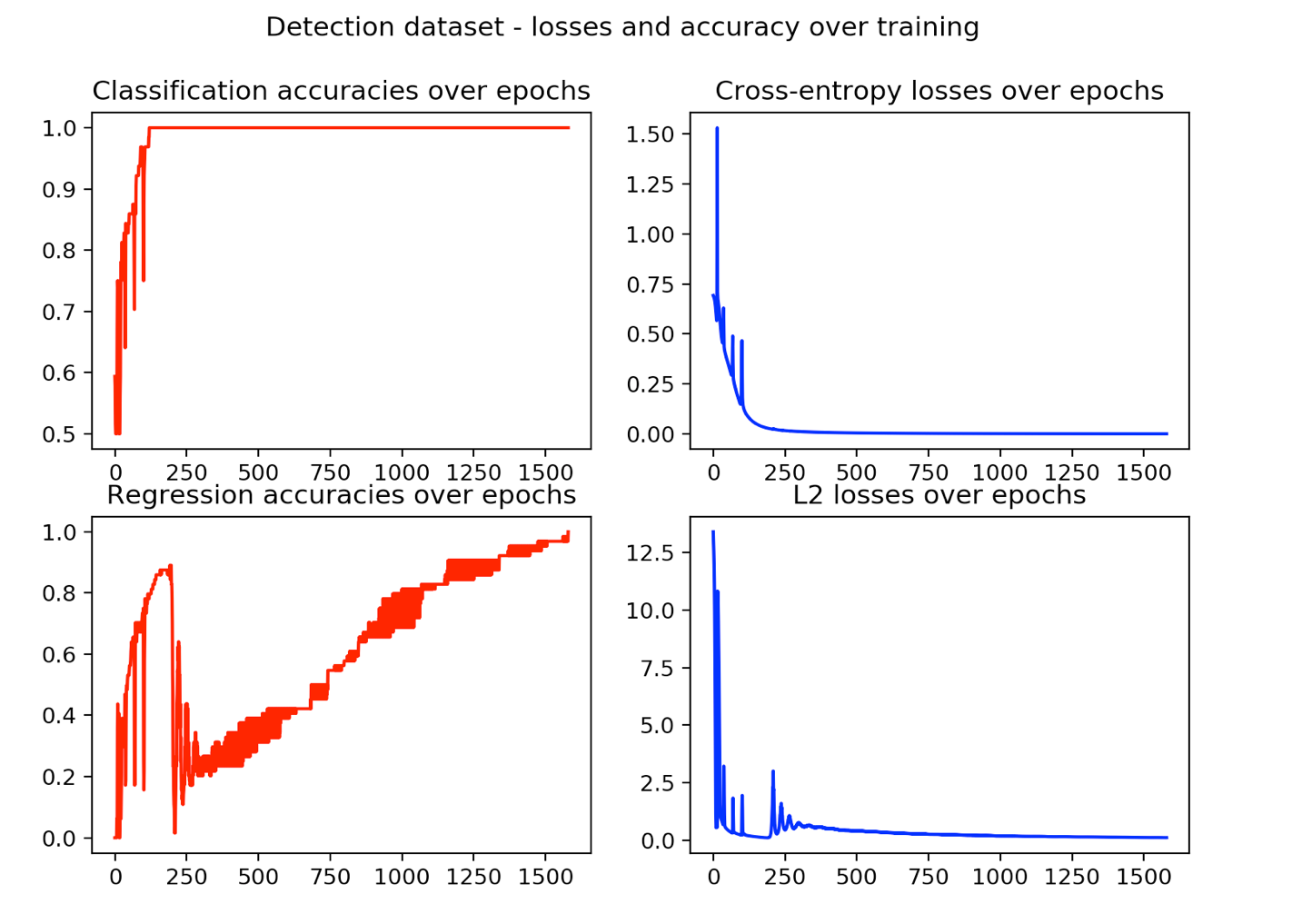


**Problem 6**

1. I implement a Leaky ReLU as ‘MyActivationFunction’. The graphs for part 1 as are follows:





1. The figures from part 3 with leaky ReLU are as follows:

It seems that the leaky ReLU did not affect the detection dataset much compared to the ReLU run, but decreased the convergence epochs required for the line dataset by an order of 3. Since I set the ‘leak’ coefficient to a small number, perhaps this explains the lack of significance.

**Problem 7**

We download the Mask RCNN as per the instructions and run a few images from the internet through. We find that it misclassifies with high confidence objects which are not commonly found, but have strong features. For example, it mislabels an image of a vintage stereo as a phone, while a glass 3D printer partially occluded behind a translucent screen is classified as a television.

It likely outputs these high-confidence misclassifications due

1. Strong features which are rare and tightly associated with the class it misclassifies as
   1. Televisions have shiny, glassy texture
2. The net does not have much data on rare objects to train on them. They can incur minimal loss by mis-classifying a rare class in order to maximize confidence on the common class.
   1. Vintage stereos are not common at all, whereas phones are much more common