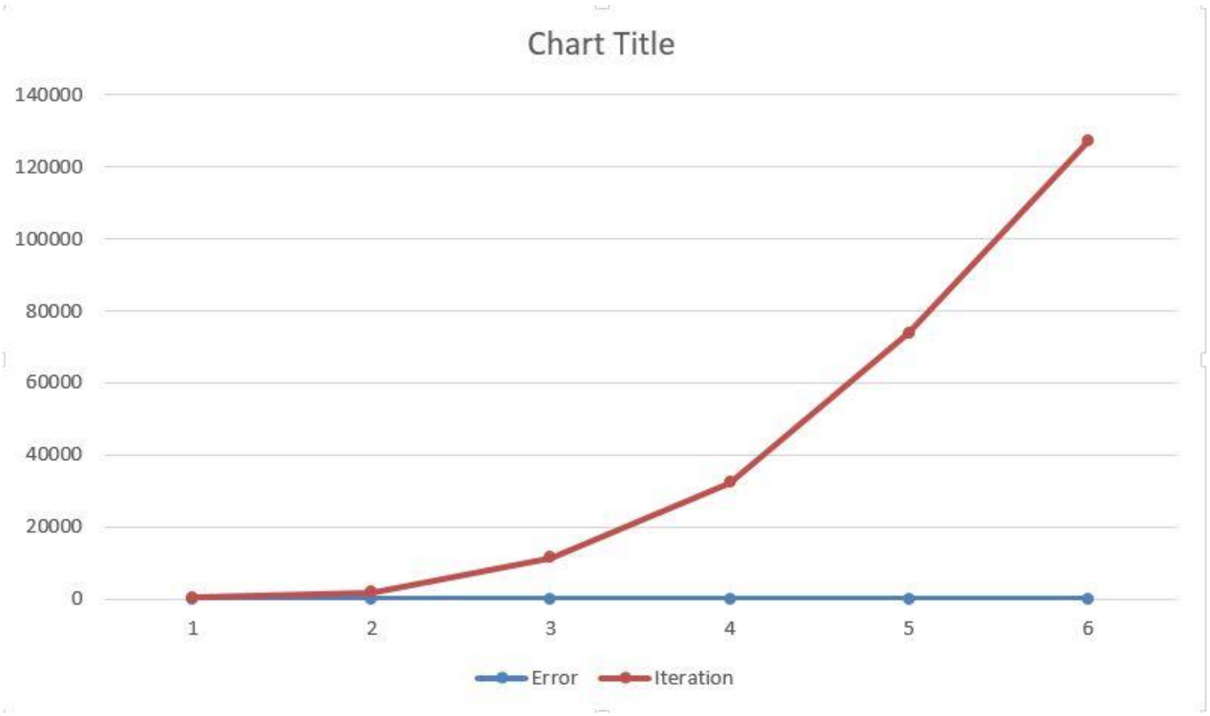
1. b) It took 386 iterations for the error to drop below 0.01. The code for the implementation is attached.

1. c)

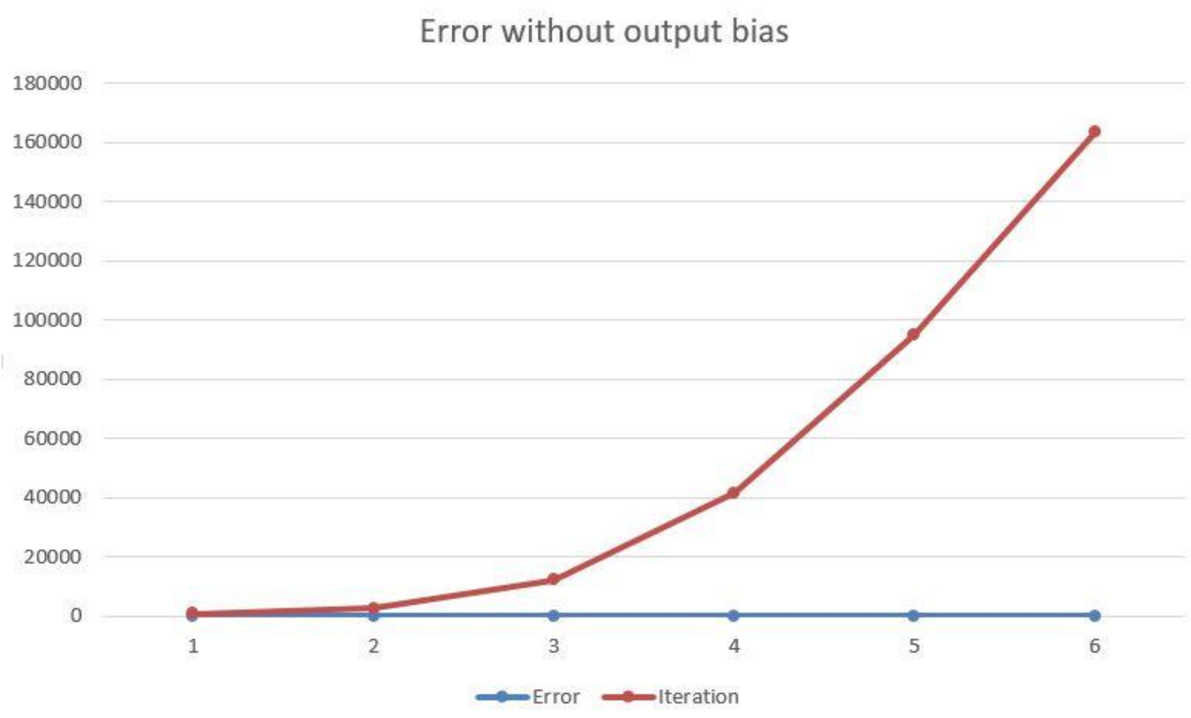
Error	Iteration
0.01	400
0.001	1900
0.0001	11400
0.00001	32300
0.000001	73800
0.0000001	127000



My observation here is that as the number of error increases, error value decreases. Which means the graph converges as the number of iterations increases and the weights get updated.

1.d)

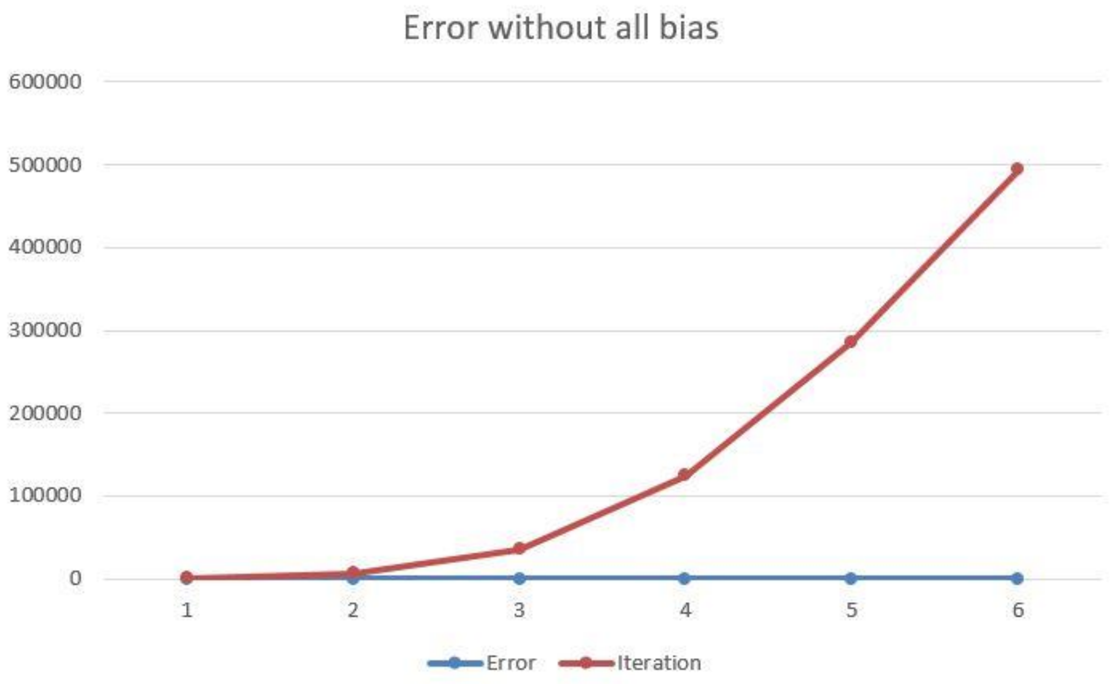
Error	Iteration
0.01	700
0.001	2700
0.0001	12300
0.00001	41500
0.000001	95000
0.0000001	163400



Here, it took more iterations than before to reach the minimum error.

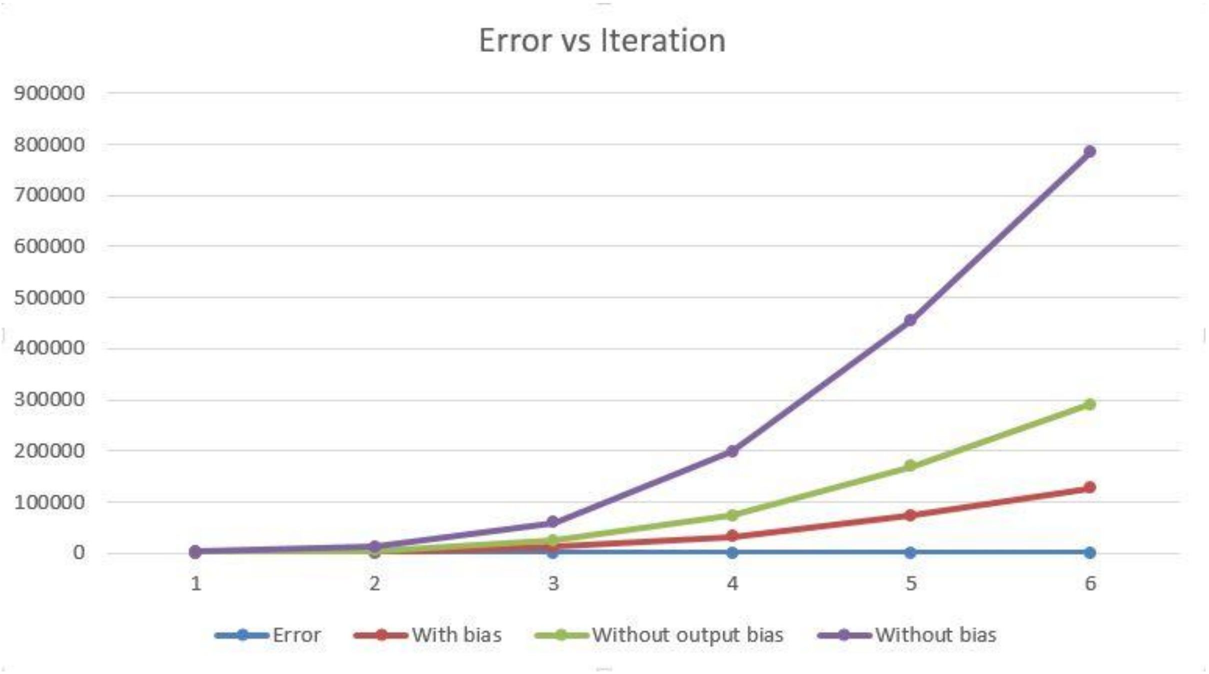
1.e)

Error	Iteration
0.01	1200
0.001	6900
0.0001	35500
0.00001	124600
0.000001	286300
0.0000001	494000

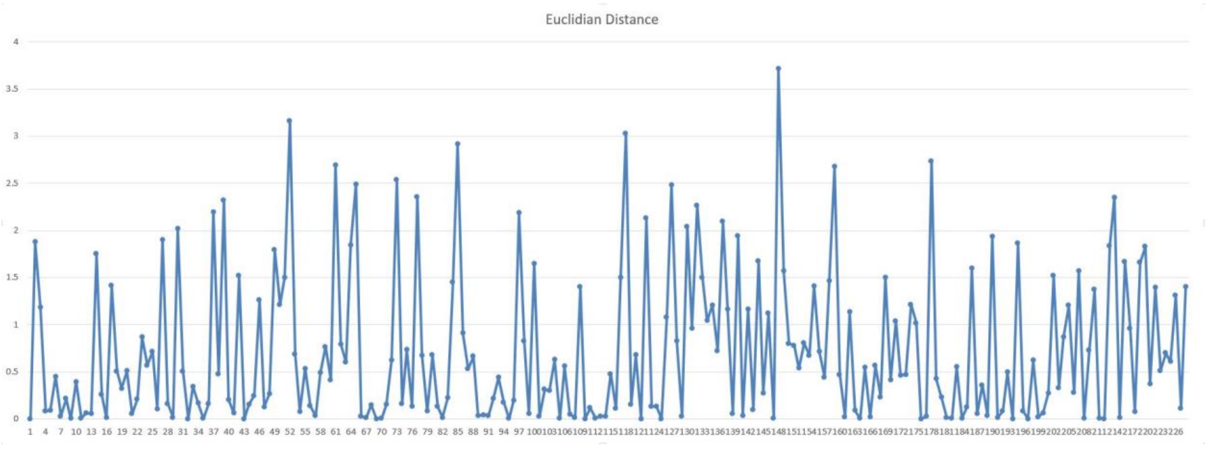


Here, it took even more iterations than without the output bias case to reach the minimum error.

1.f) From the above observations, we can say that the bias weights help the network to converge quickly. As we remove biases, the number of iterations to reach minimum error increases.



1.g) yes, the network converges with all the 20 sets of random weights. It took 1700 to 3000 iterations to reach the error 0.001. Also the convergence occurs around the same points which we can visualize from the graph.



2.a) The softmax function is the gradient-log-normalizer of the categorical probability distribution. It is used in various probabilistic multiclass classification methods including multiclass linear discriminant analysis, naive Bayes classifiers and artificial neural networks.
For multinomial classification problems (1-of-n, where n > 2) we use a network with n outputs, one corresponding to each class, and target values of 1 for the correct class, and 0 otherwise. Since these targets are not independent of each other, it is no longer appropriate to use logistic output units.

The correct generalization of the logistic sigmoid to the multinomial case is the softmax activation function.

$$f(net_i) = e^{net\_i} / \sum_{0} e^{net\_o}$$

In neural network simulations, the softmax function is often implemented at the final layer of a network used for classification. Such networks are then trained under a log loss (or cross-entropy) regime, giving a non-linear variant of multinomial logistic regression.

A probabilistic interpretation is that the softmax leads to model the joint distribution over the output variables p(x1, x2, x3, ..., xn) whereas using sigmoids leads to model the marginal distributions p(x1), p(x2), p(x3), ..., p(xn).

A more hands on way of thinking about it is that pushing up the value for output i will lead the probability of other outputs to go down if using a softmax. For a sigmoid, it won't affect the probabilities for the other outputs. An even more hands on way to think about it is that sigmoids make sense when there are lots of independent yes/no labels. For example, if there is a classifier that predicts male vs. female, young vs. old, etc. Whereas the softmax makes sense for exclusive classes.

2.b) Given the probabilistic interpretation, a network output of, say, 0.01 for a pattern that is actually in the '1' class is a much more serious error than, say, 0.1. Unfortunately the sum-squared loss function makes almost no distinction between these two cases. A loss function that is appropriate for dealing with probabilities is the cross-entropy error. For the two-class case, it is given by

$$E = -t \ln y - (1-t) \ln (1-y)$$

When logistic output units and cross-entropy error are used together in backpropagation learning, the error signal for the output unit becomes just the difference between target and output:

$$\frac{\partial \mathrm{E}}{\partial \mathrm{net}} = \ldots\ldots = y-t$$

In other words, implementing cross-entropy error for this case amounts to nothing more than omitting the f'(net) factor that the error signal would otherwise get multiplied by. This is not an accident, but indicative of a deeper mathematical connection: cross-entropy error and logistic outputs are the "correct" combination to use for binomial probabilities, just like linear outputs and sum-squared error are for scalar values.

The back-propagation algorithm computes gradient values which are derived from some implicit measure of error. Typically the implicit error is mean squared error, which gives a particular gradient equation that involves the calculus derivative of the softmax output activation function. But we can use implicit cross-entropy error instead of implicit mean squared error. This approach changes the back-propagation equation for the gradients.