Part 1: Logistic Regression with Numpy

Question 1: Loss Function and Gradient

```python
def loss(W, b, x, y, reg):
    y_hat = sigmoid(np.matmul(W.T,x)+b)
    log_1 = np.log(y_hat+0.00001)
    log_2 = np.log(1-(y_hat-0.00001))
    loss = -np.dot(log_1,y) - np.dot(log_2,1-y)
    loss = loss/y_hat.shape[1] + reg/2*(np.linalg.norm(W)**2)
    return loss[0,0]

def grad_loss(W, b, x, y, reg):
    y_hat = sigmoid(np.matmul(W.T,x)+b)
    grad = np.matmul(x,y_hat.T-y)
    b_grad = np.sum(y_hat.T-y)/x.shape[1]
    grad = grad/x.shape[0] + reg*W
    return grad, b_grad
```

$$L = L_{CE} + L_W \Rightarrow \frac{dL}{dW} = \frac{dL_{CE}}{dW} + \frac{dL_W}{dW}$$

$$\frac{dL_{CE}}{dW} = \frac{dL_{CE}}{d\hat{y}} \cdot \frac{d\hat{y}}{dz} \cdot \frac{dz}{dx} \qquad \frac{dL_W}{dW} = \lambda W$$

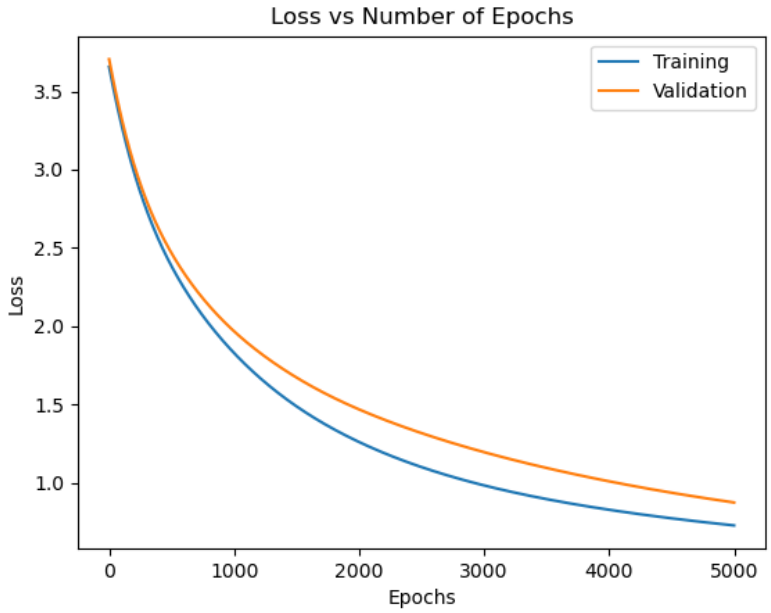
$$= \frac{\hat{y}-y}{\hat{y}(1-\hat{y})} \cdot \hat{y}(1-\hat{y}) \cdot x$$

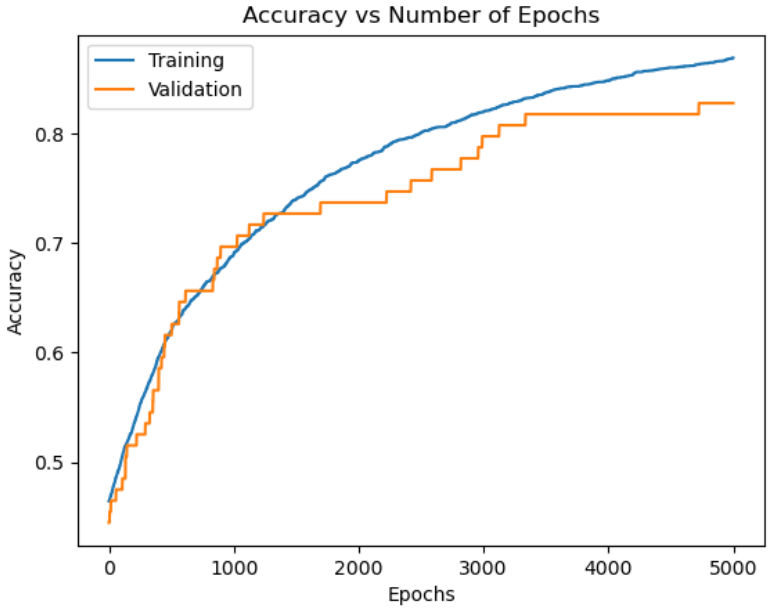
$$= x \cdot (\hat{y}-y)$$

$$\therefore \frac{dL}{dW} = x \cdot (\hat{y}(x)-y) + \lambda W$$

Question 3: Tuning the Learning Rate

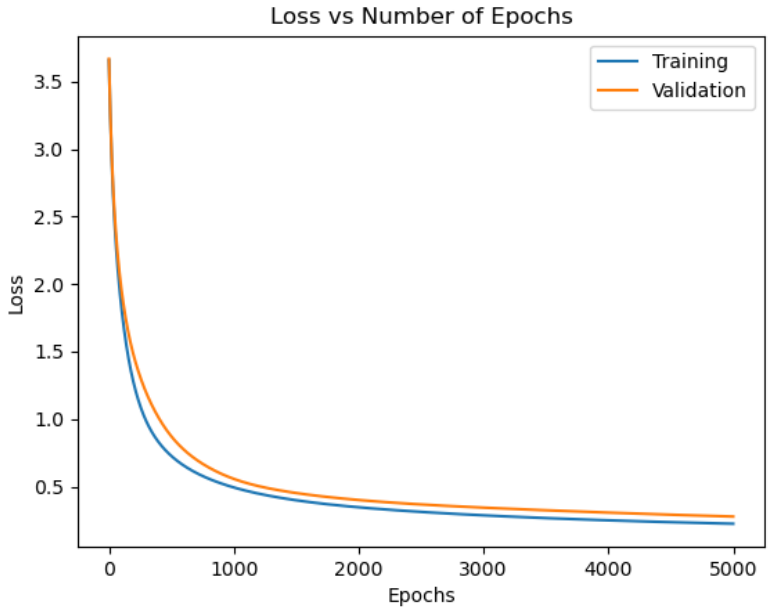
LR = 0.0001

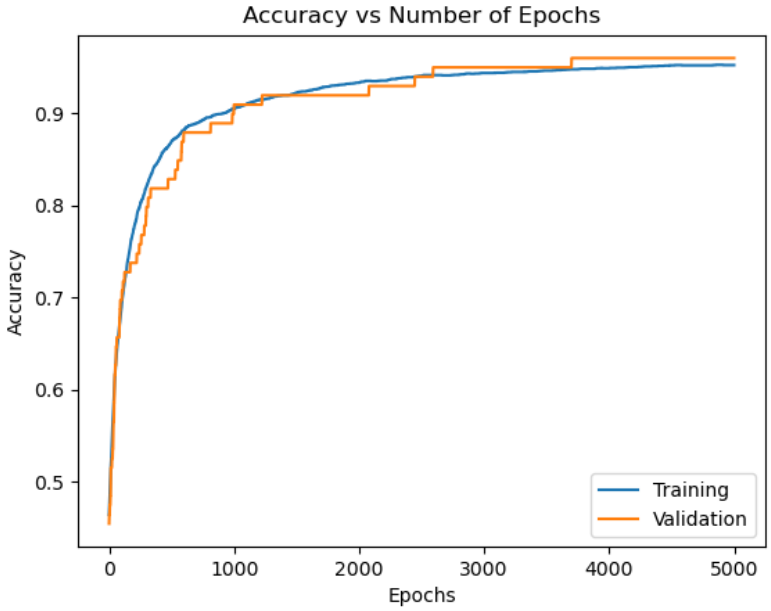




Test Accuracy: 0.8611111111111112
Test Loss: 0.6468013475364004

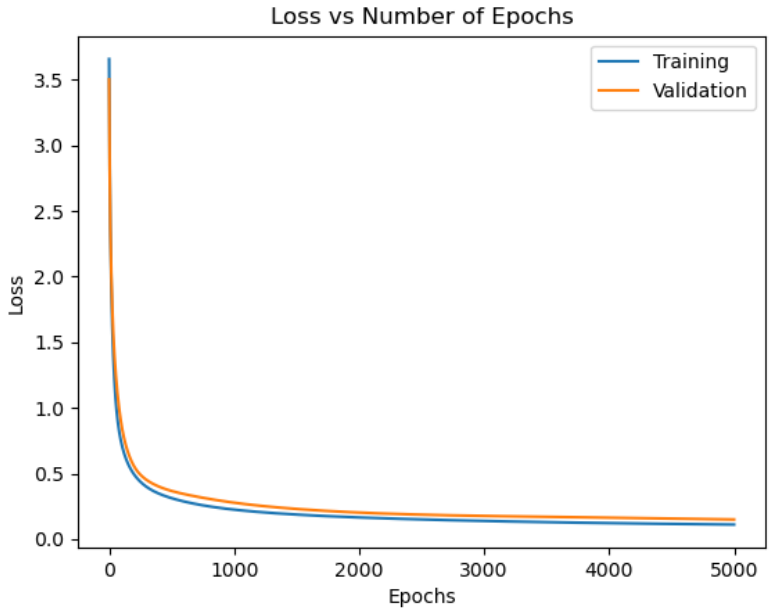
LR = 0.001

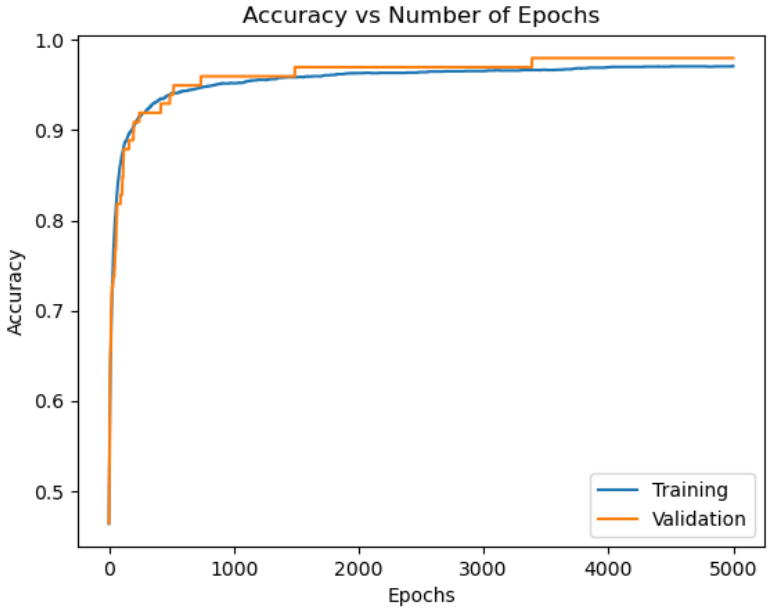




Test Accuracy: 0.9722222222222222
Test Loss: 0.25569847606392104

LR = 0.005



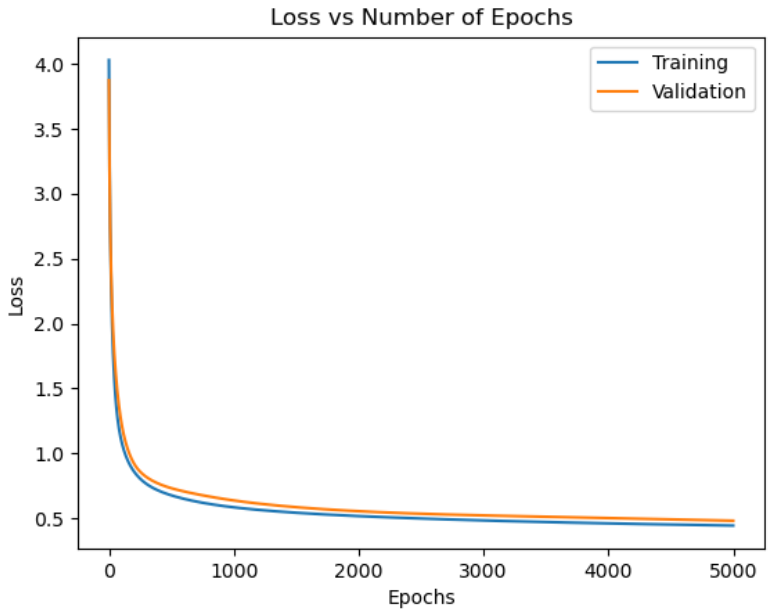


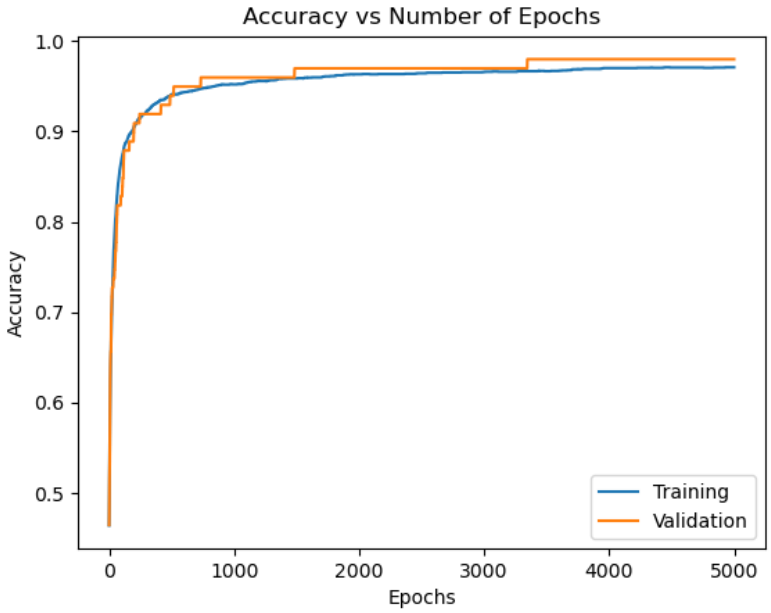
Test Accuracy: 0.9791666666666666
Test Loss: 0.1803511687509441

As shown in the above plots, the best learning rate is 0.005. 0.001 is also a decent learning rate, however, 0.005 displays the same behaviour with a quicker performance (i.e. fewer epochs to reach optimal performance). A learning rate of 0.005 also lead to the highest test accuracy and lowest test loss in comparison to the other learning rates chosen, with a test accuracy of 97.9%.

Question 4: Generalization

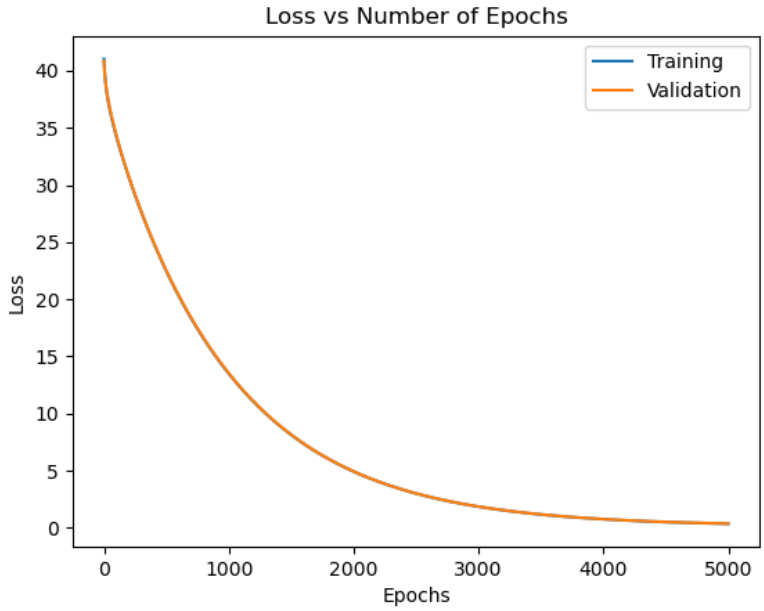
Reg = 0.001

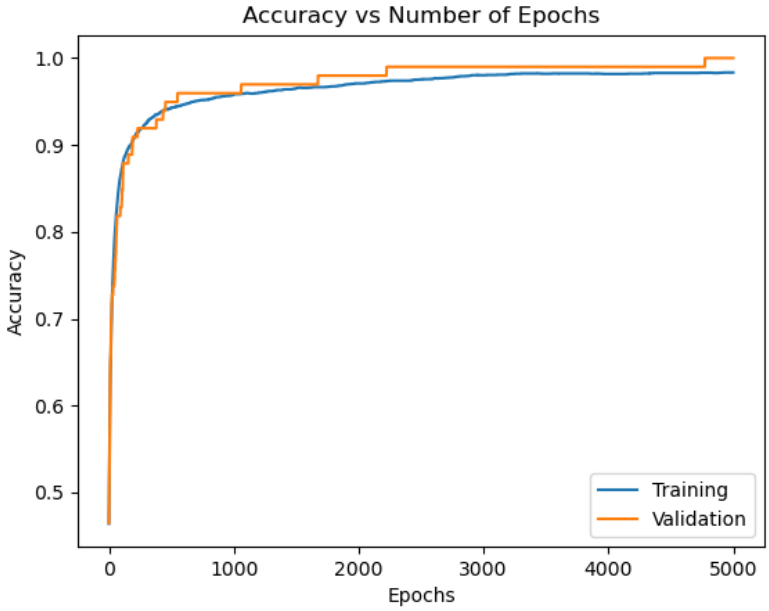




Test Accuracy: 0.9791666666666666
Test Loss: 0.5118681093108216

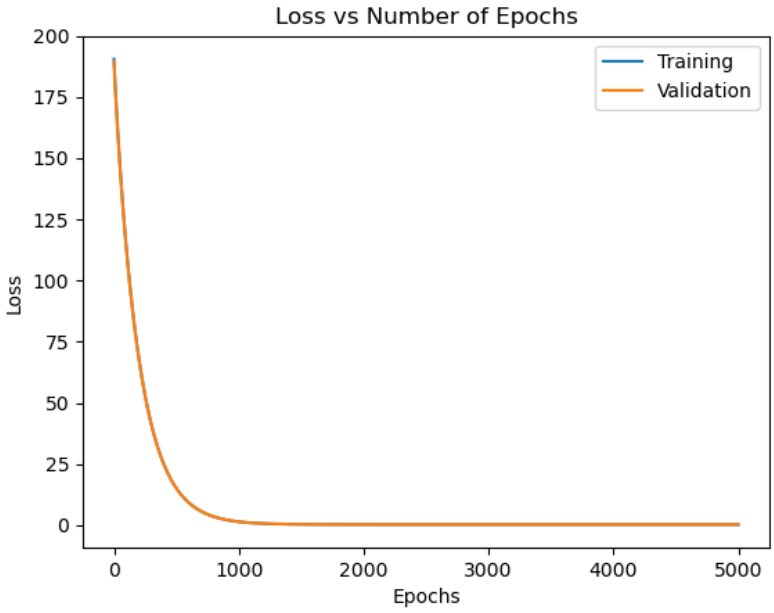
Reg = 0.1

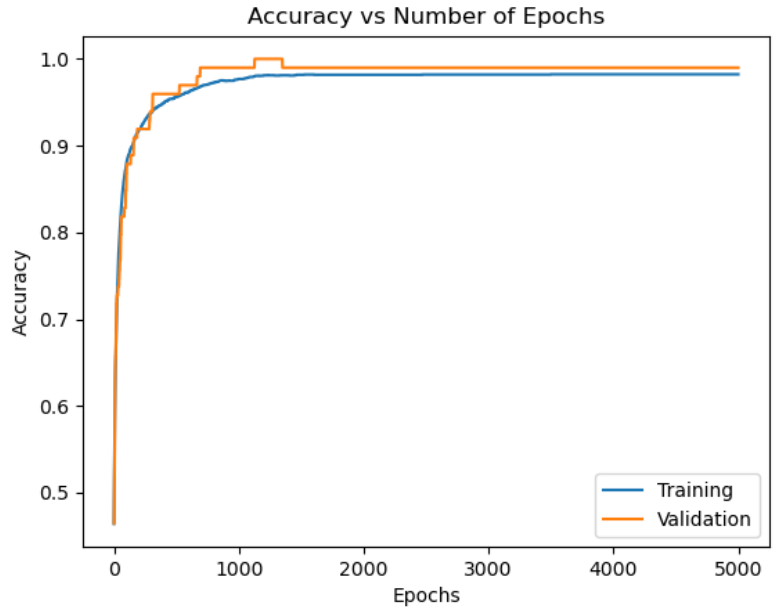




Test Accuracy: 0.9861111111111112
Test Loss: 0.3911504227801265

Reg = 0.5





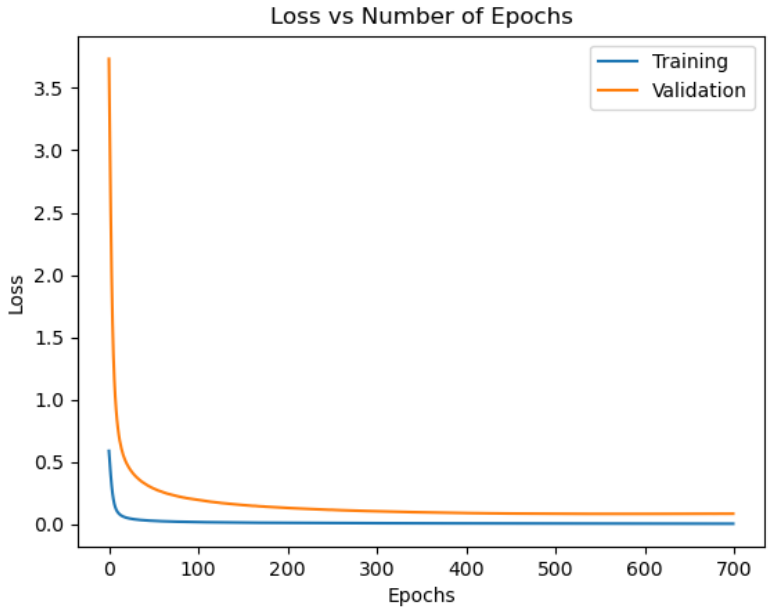
Test Accuracy: 0.9861111111111112
Test Loss: 0.26327151712332336

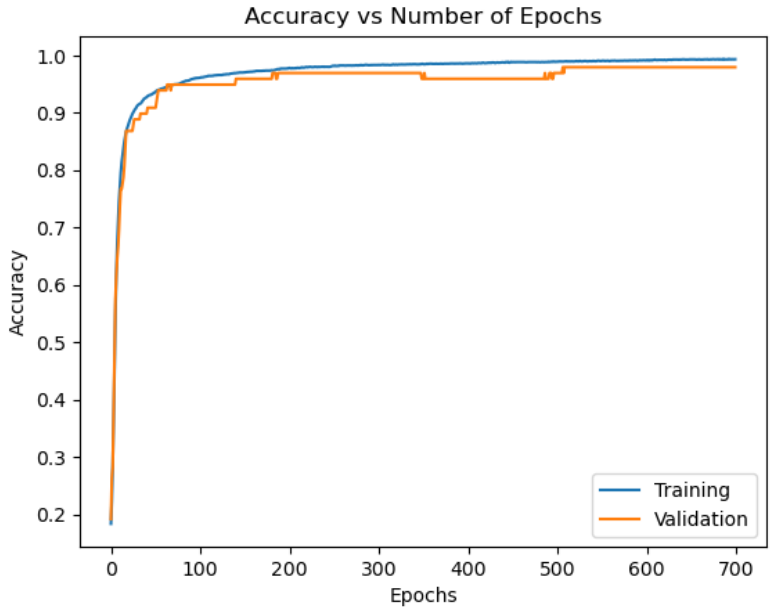
By altering the learning parameter with a learning rate of 0.005, the behaviour of each of the models were relatively similar with little variance, however, a regularization parameter of 0.5

lead to the best performance in terms of highest test accuracy and lowest test loss. The loss plots emphasize the initial effect of the regularization parameter as higher regularization parameters lead to a higher initial loss. The accuracy with a 0.5 regularization parameter reached its peak accuracy relatively quickly which displays optimal performance. A regularization parameter of 0.5 lead to a test accuracy of 98.6%.

Part 2: Logistic Regression in TensorFlow

Question 2: Implementing Stochastic Gradient Descent

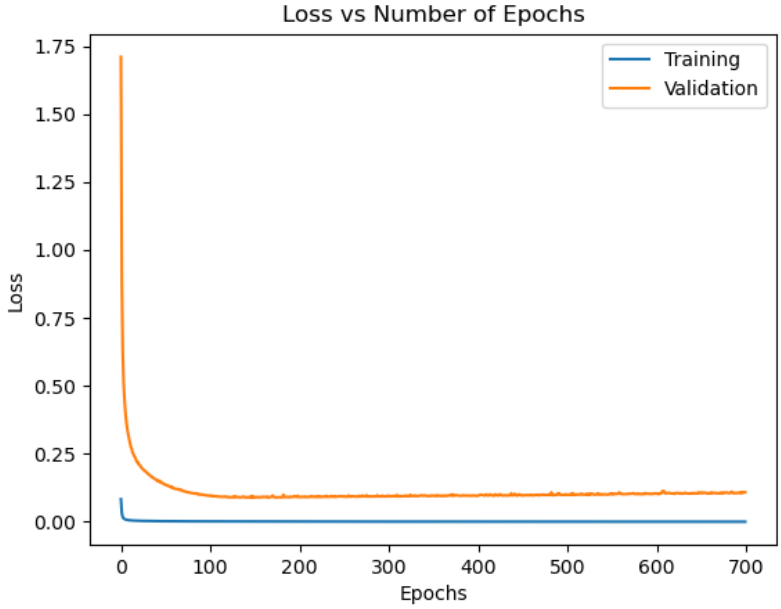


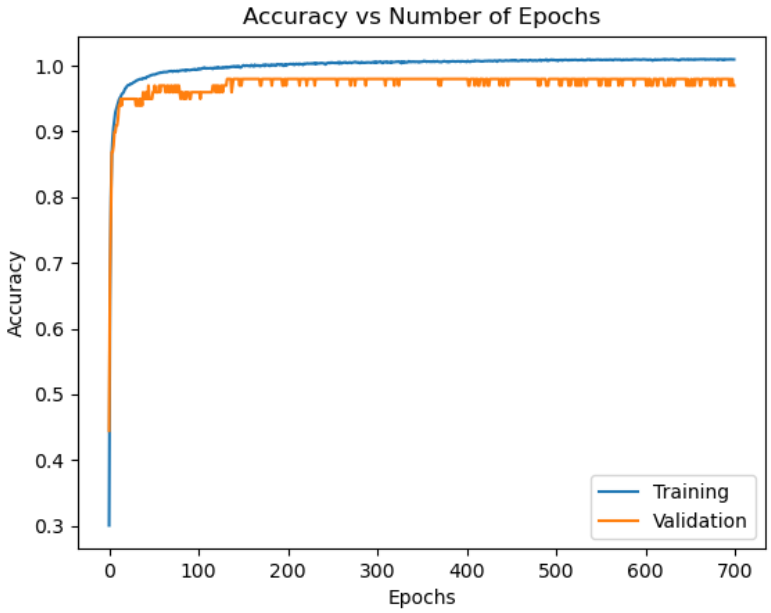


Test Accuracy: 0.9791666666666666
Test Loss: 0.11276068442841032

Question 3: Batch Size Investigation

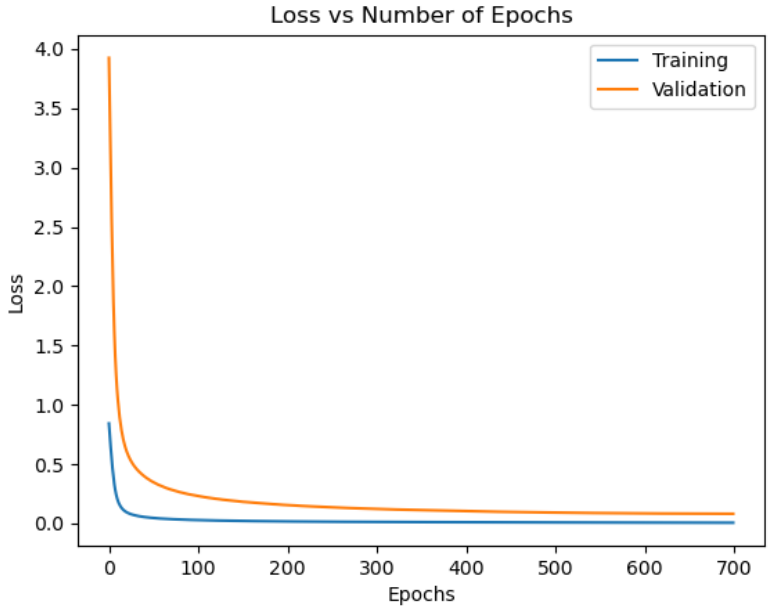
Batch Size = 100

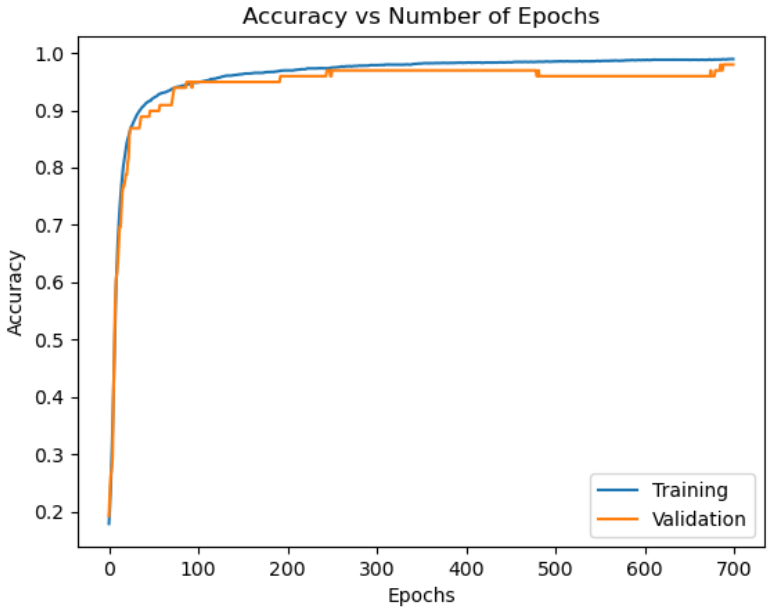




Test Accuracy: 0.9791666666666666
Test Loss: 0.13583040605958807

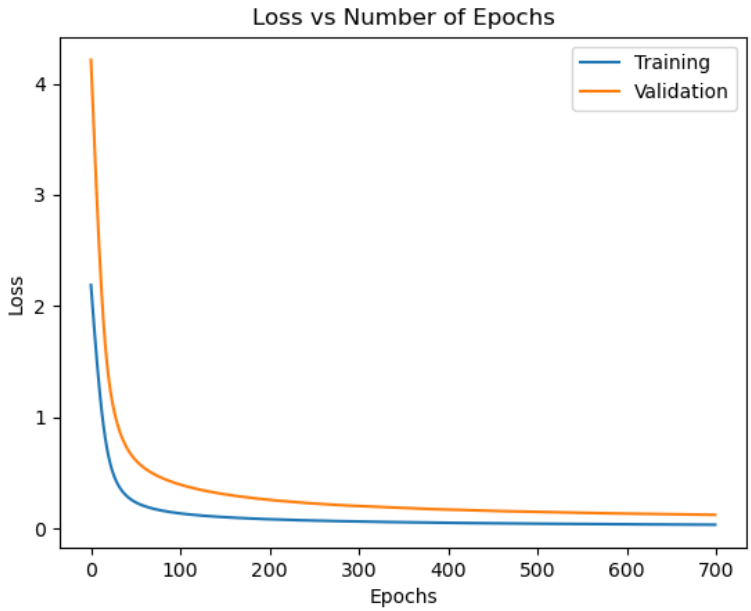
Batch Size = 700

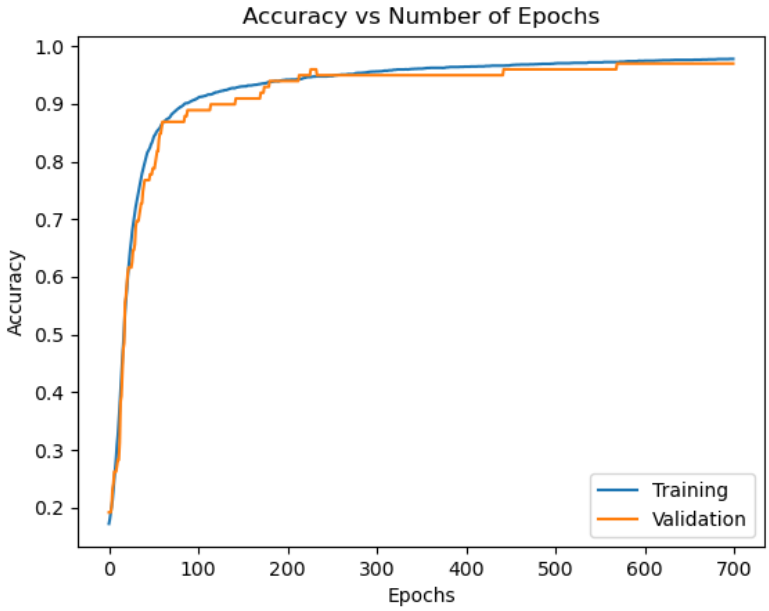




Test Accuracy: 0.9791666666666666
Test Loss: 0.11675046598544803

Batch Size = 1750



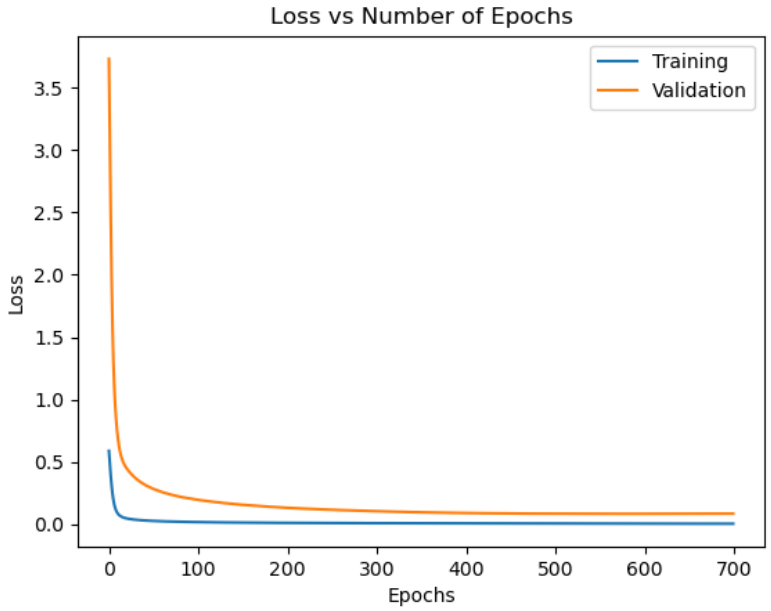


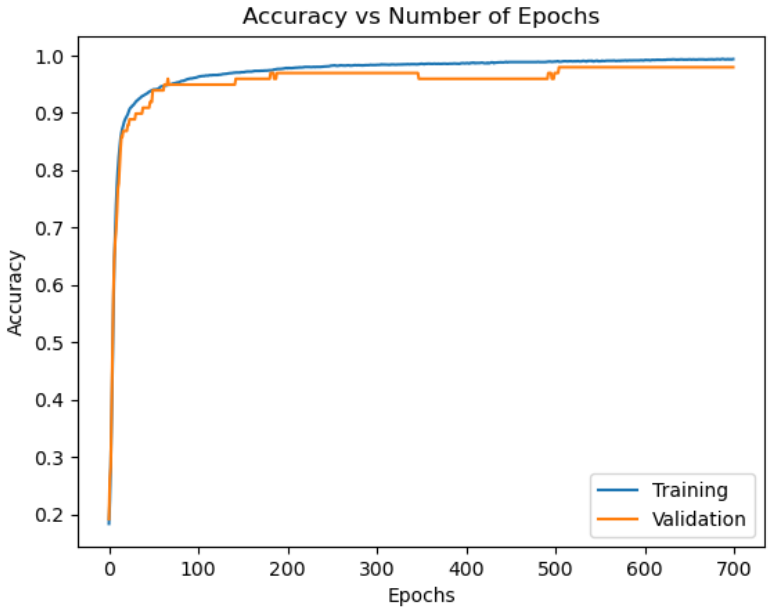
Test Accuracy: 0.9861111111111112
Test Loss: 0.12569449679390177

The effect of batch size on the performance of the model is most apparent in the accuracy plots. When using smaller batch sizes, the weights are updated more frequently for a smaller number of data points. As a result, there is a higher level of bias towards those fewer data points and when applied to the validation data, there is a much greater chance of variability in its performance. This can be seen when using a small batch size of 100 which leads to small oscillations in the accuracy of the validation data as expected. However, as the batch size is increased, fewer oscillations become noticeable because the validation data is analyzed over a larger portion of the dataset. With a larger batch size, the final accuracy becomes larger by a small margin. If these models were run successively, there is a much greater chance of the test accuracy dropping off for the smaller batch sizes. Since the weights are changing very frequently over a smaller number of test samples, the final test dataset could have been analyzed using a poor representation of the weights based on the final batch of the training loop. This is much more probable for smaller datasets since generalization over the entire dataset is worse compared to a batch size that is greater.

Question 4: Hyperparameter Investigation

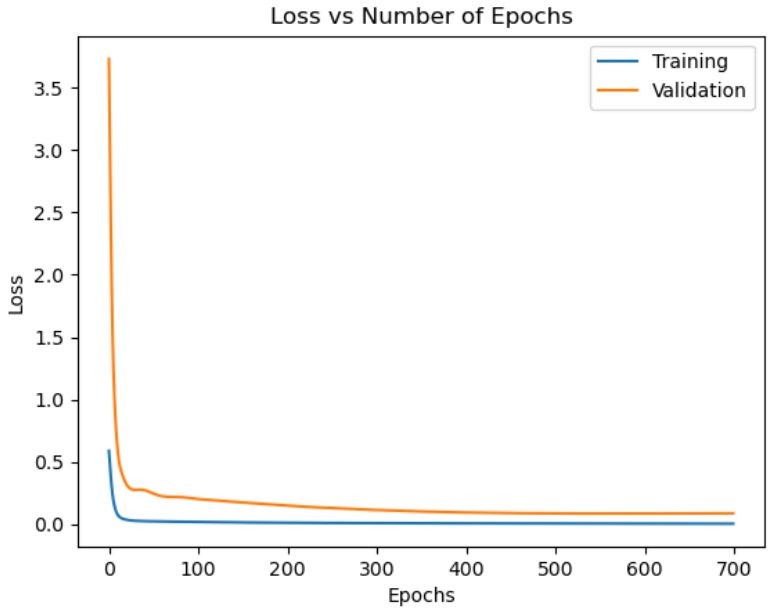
Beta_1 = 0.95

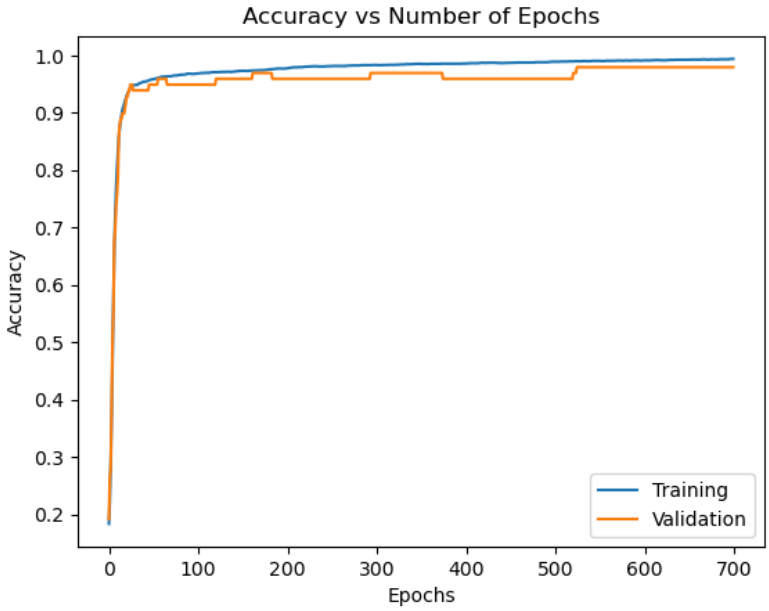




Test Accuracy: 0.9791666666666666
Test Loss: 0.11271498770846512

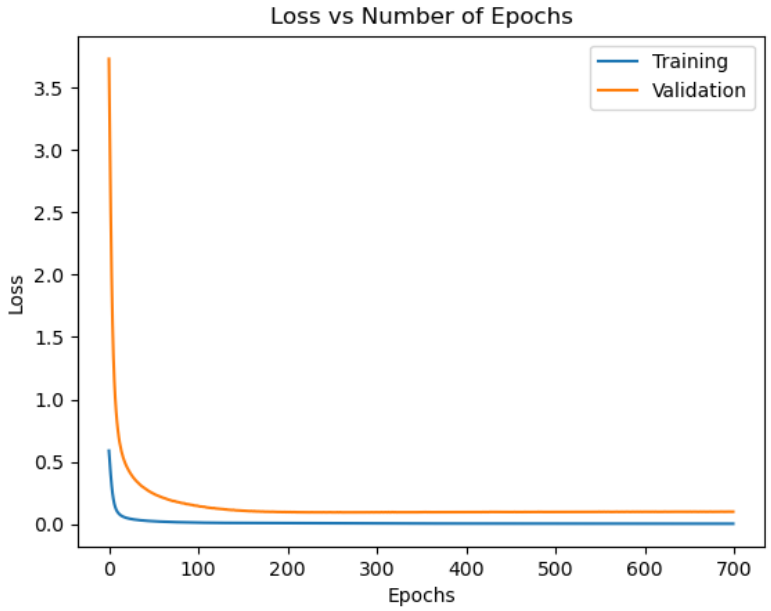
Beta_1 = 0.99

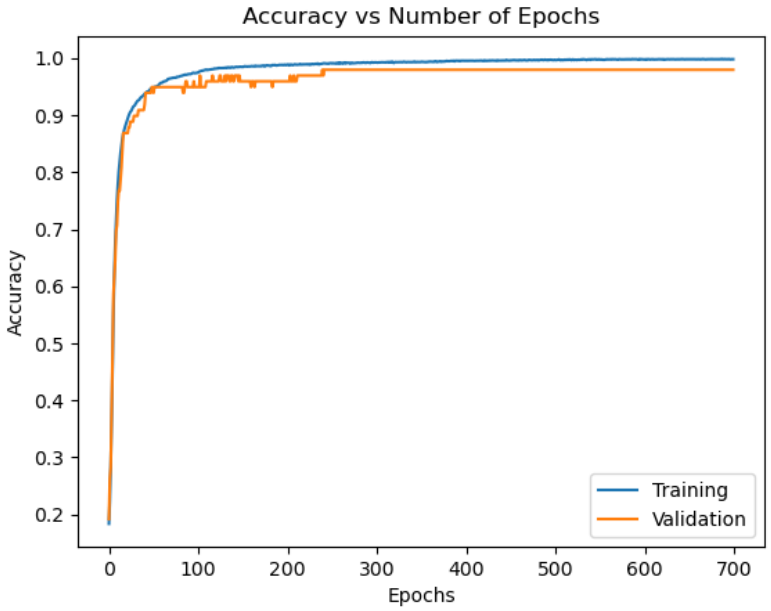




Test Accuracy: 0.9791666666666666
Test Loss: 0.11312064178232883

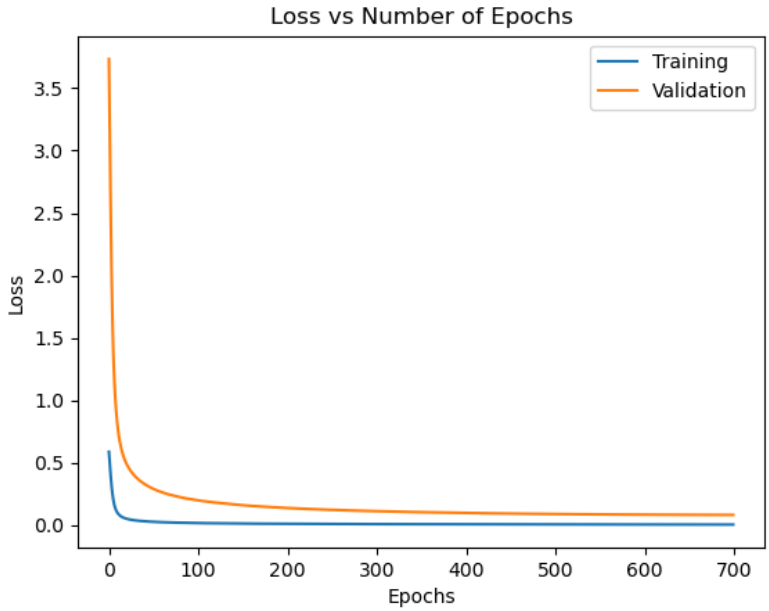
Beta_2 = 0.99

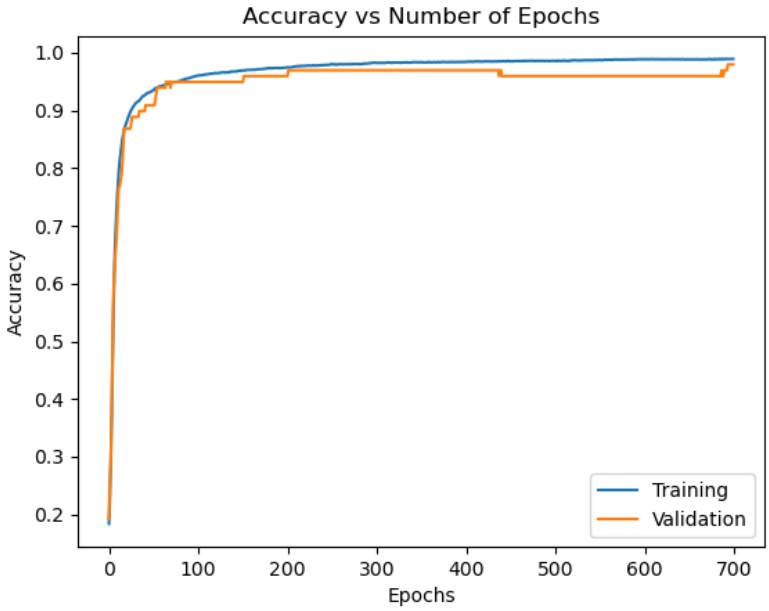




Test Accuracy: 0.9861111111111112
Test Loss: 0.1182350595098537

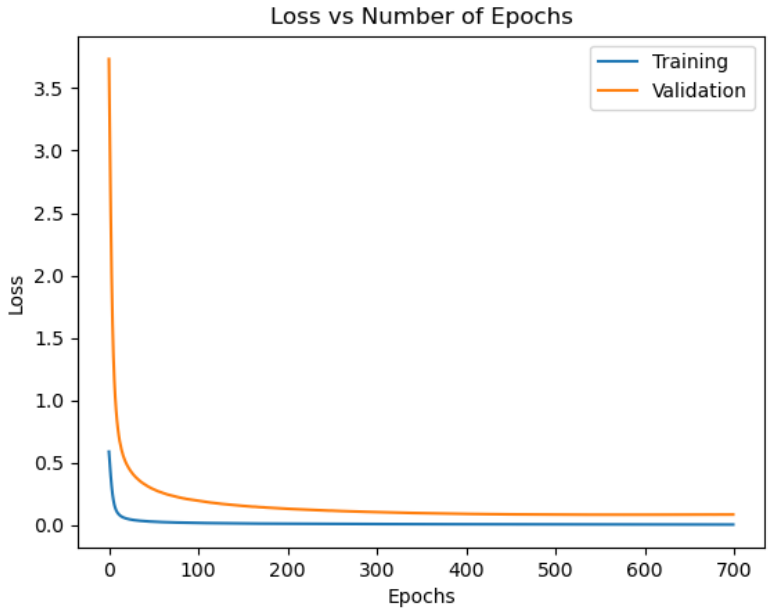
Beta_2 = 0.9999

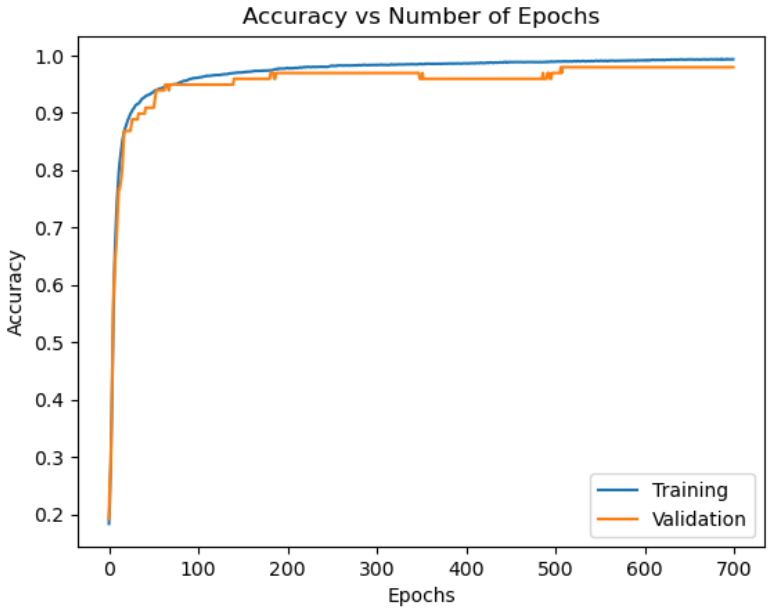




Test Accuracy: 0.9791666666666666
Test Loss: 0.11801902569586867

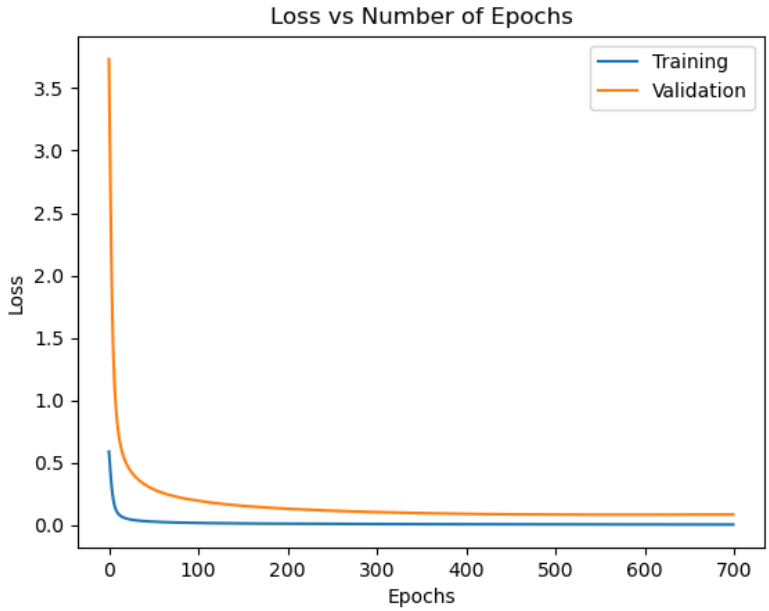
Epsilon = 1e-09

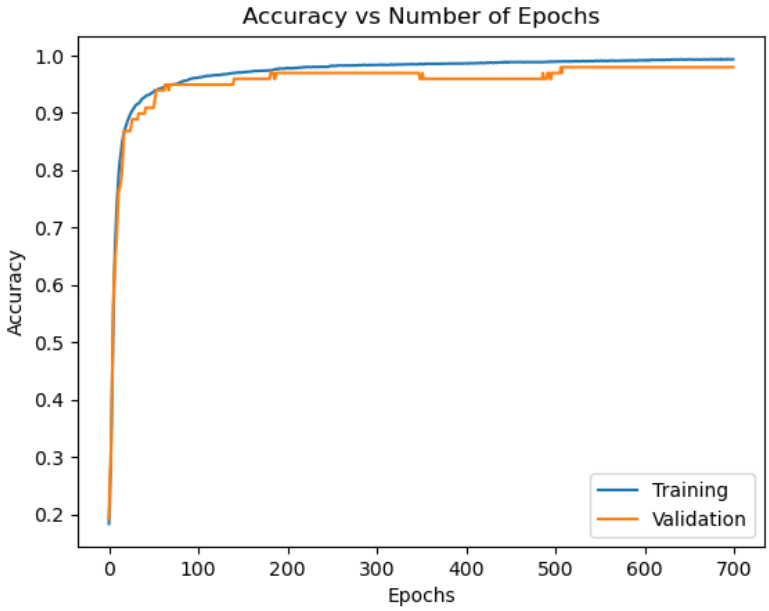




Test Accuracy: 0.9791666666666666
Test Loss: 0.11276068442841032

Epsilon = 1e-04





Test Accuracy: 0.9791666666666666
Test Loss: 0.11275993327104165

Question 5: Comparison against Batch GD

The most noticeable observation between the performance of SGD versus the batch gradient descent algorithm is the number of respective epochs required to reach optimal behaviour. Since the dataset only consists of two classes, there are various image representations of the letters

C and J which look very similar throughout the dataset. As a result, it is not necessary to update the weights through one entire pass of 3500 training samples. This claim is apparent in the behaviour of using SGD versus batch gradient descent as SGD is able to reach essentially the same values of loss and accuracy with only 700 epochs in comparison to 5000 epochs for batch gradient descent.