ECE421 Assignment 1 Report

Name: Deniz Akyildiz
Student number: 1003102224

1. Logistic Regression with Numpy

1.1 Loss Function and Gradient

The analytical expression for the loss function is:

$$L = \frac{1}{N}\sum_{n=1}^{N}[-y_n * log(\widehat{y_n}) - (1-y_n) * log(1-\widehat{y_n})] + (\frac{\lambda}{2} * w^T w)$$

where $\widehat{y} = \sigma(w^T x + b)$. The code snippet for the calculation of the loss function is given in Figure-1.

```
# To avoid repetition, defined a y hat calculation function here
def y_hat_calculation(x, W, b):
    return (1 / (1 + np.exp( -(np.matmul(x,W) + b) )))

def loss(W, b, x, y, reg):
    y_hat = y_hat_calculation(x, W, b)

    loss = (np.sum ( -(y * np.log(y_hat)) - \
        (1 - y) * np.log(1 - y_hat))) / (np.shape(y)[0]) \
        + ((reg/2) * np.matmul(np.transpose(W), W))

    return loss[0][0]
```

Figure-1: Loss calculation

To derive the gradient, we first calculate the derivative of the loss function with respect to a single weight, w_j

$$\frac{\partial L}{\partial w_j} = \frac{\partial}{\partial w_j}(\frac{1}{N}\sum_{n=1}^{N}[-y_n\, log(\widehat{y_n}) - (1-y_n)\, log(1-\widehat{y_n})] + (\frac{\lambda}{2} * \sum_{i=1}^{N} {w_i}^2))$$

$$= \frac{1}{N}\sum_{n=1}^{N}[\frac{-y_n}{\widehat{y_n}}\frac{\partial}{\partial w_j}(\widehat{y_n}) + \frac{(1-y_n)}{1-\widehat{y_n}}\frac{\partial}{\partial w_j}(1-\widehat{y_n})] + (\lambda w_j)$$

$$= \frac{1}{N}\sum_{n=1}^{N}[(\frac{-y_n}{\widehat{y_n}} + \frac{(1-y_n)}{1-\widehat{y_n}})\frac{\partial}{\partial w_j}(\widehat{y_n})] + (\lambda w_j) \quad (1)$$

Now, using the derivative $\frac{\partial}{\partial w_j}\widehat{y_n} =$

$$\frac{\partial}{\partial w_j}\sigma(w^T x_n + b) = \sigma(w^T x_n + b)\,[1 - \sigma(w^T x_n + b)]\frac{\partial}{\partial w_j}(w^T x_n + b)$$
$$= \sigma(w^T x_n + b)\,[1 - \sigma(w^T x_n + b)]\,x_{nj} = \widehat{y_n}\,(1 - \widehat{y_n})\,x_{nj}$$

Plugging in to the equation (1) we end up with:

$$\frac{\partial L}{\partial w_j} = \frac{1}{N}\sum_{n=1}^{N}\left[\left(\frac{-y_n + \widehat{y_n}}{\widehat{y_n}\,(1-\widehat{y_n})}\right)\widehat{y}_n\,(1 - \widehat{y_n})\,x_{nj}\right] + (\lambda w_j)$$
$$= \frac{1}{N}\sum_{n=1}^{N}\left[(\widehat{y}_n - y_n)\,x_{nj}\right] + (\lambda w_j)$$

And finally the analytical expression for the gradient of the loss function is :

$$\nabla L(w) = \frac{1}{N}\,[x^T\,(\widehat{y} - y)] + \lambda w$$

Following similar steps, but since this time bias term does not have a multiplier x:

$$\nabla L(b) = \frac{1}{N}\sum_{n=1}^{N}(\widehat{y_n} - y_n)$$

The code snippet for the calculation of the gradient of the loss function is given in Figure-2.

```
def grad_loss(W, b, x, y, reg):
    y_hat = y_hat_calculation(x, W, b)

    # Included derivations in the report
    grad_loss_w = np.matmul(np.transpose(x), (y_hat - y)) \
        / np.shape(y)[0] + reg*W

    # Essentially same derivation as the weight gradient, but now instead
    # of wTx, we have b so the x in the expression no longer exists
    grad_loss_b = np.sum((y_hat - y))/(np.shape(y)[0])

    return grad_loss_w, grad_loss_b
```

Figure-2: Python implementation of grad_loss() function

1.2 Gradient Descent Implementation

The algorithm is taken from the textbook [1], with the addition of specified termination condition from the assignment handout, which is to stop when the norm of the difference between the old and updated weights are smaller than the error tolerance. See Figure-3 for the algorithm reference, and Figure-4 for the Python implementation.

Fixed learning rate gradient descent:

1: Initialize the weights at time step $t = 0$ to $\mathbf{w}(0)$.
2: **for** $t = 0, 1, 2, \ldots$ **do**
3: Compute the gradient $\mathbf{g}_t = \nabla E_{\text{in}}(\mathbf{w}(t))$.
4: Set the direction to move, $\mathbf{v}_t = -\mathbf{g}_t$.
5: Update the weights: $\mathbf{w}(t+1) = \mathbf{w}(t) + \eta \mathbf{v}_t$.
6: Iterate to the next step until it is time to stop.
7: Return the final weights.

Figure-3: Gradient descent algorithm with no termination condition [1]

The function in Figure-4 is shortened for simplicity. The code also has the calculation of loss and accuracy in the algorithm in order to plot the graphs later on.

```
def grad_descent(W, b, x, y, alpha, epochs, reg, error_tol):

    # Logistic regression algorithm from textbook page 95
    for i in range(epochs):
        gradient_w, gradient_b = grad_loss(W, b, x, y, reg)
        diff_w = -alpha * gradient_w
        diff_b = -alpha * gradient_b
        W += diff_w
        b += diff_b
        if np.linalg.norm(diff_w) <= error_tol:
            break

    return W, b, valid_accur, test_accur, train_accur, valid_loss, test_loss, train_loss
```

Figure-4: Python implementation of gradient descent algorithm

1.3 Tuning the Learning Rate

As suggested in the handout, gradient descent implementation was tested with 5000 epochs and λ = 0, for different learning rates. The results could be seen in Figures 5, 6, 7, 8, 9, and 10. From these figures, it is possible to observe the effect of the learning rate.

The validation accuracies for learning rates of α = {0.005, 0.001, 0.0001} were {0.96, 0.93, 0.51} respectively. The learning rate of 0.005 was found to converge to the desired accuracy range fastest while being able to generalize. Decreasing the learning rate to 0.001 did not have a significant effect other than making the learning process slower, which was expected, without sacrificing a significant amount of accuracy. When the learning rate was set to 0.0001, the process was significantly slower and the model was not able to converge to the desired accuracy levels after the same amount of iterations. Therefore the learning rate **α = 0.005 with the final test accuracy of 0.972** was chosen since it had the fastest convergence, no overfitting, and the highest accuracy.

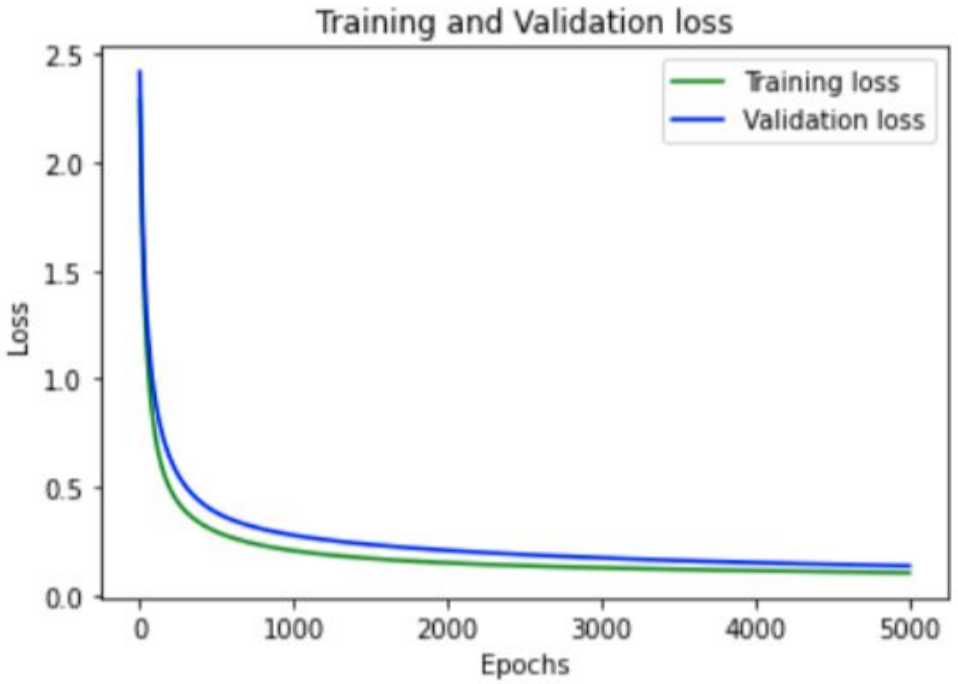


Figure-5: Loss value for validation and training data with α = 0.005

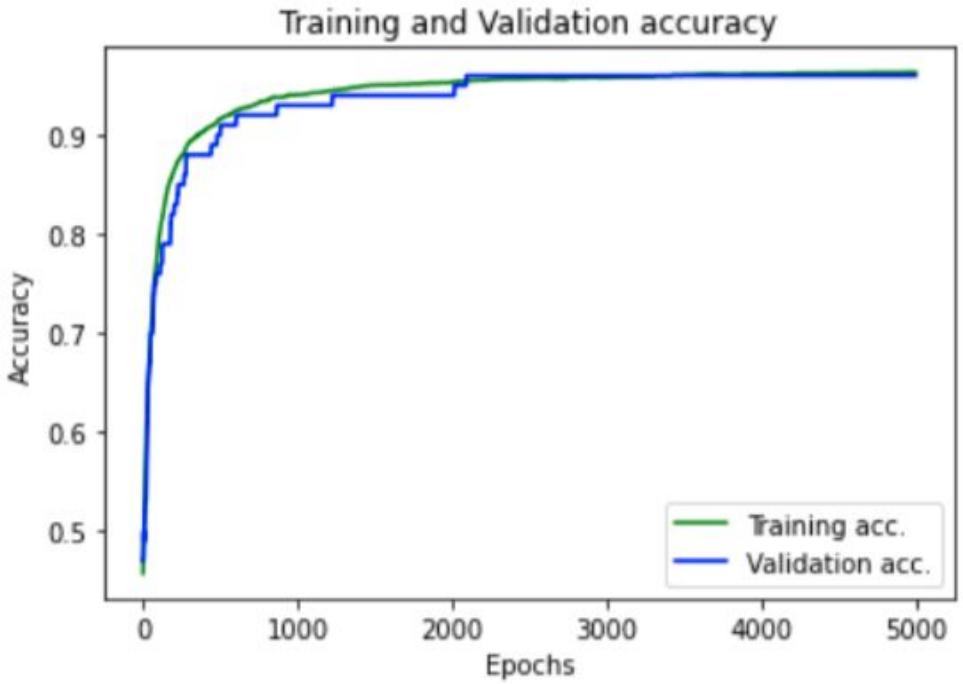


Figure-6: Accuracy value for validation and training data with α = 0.005

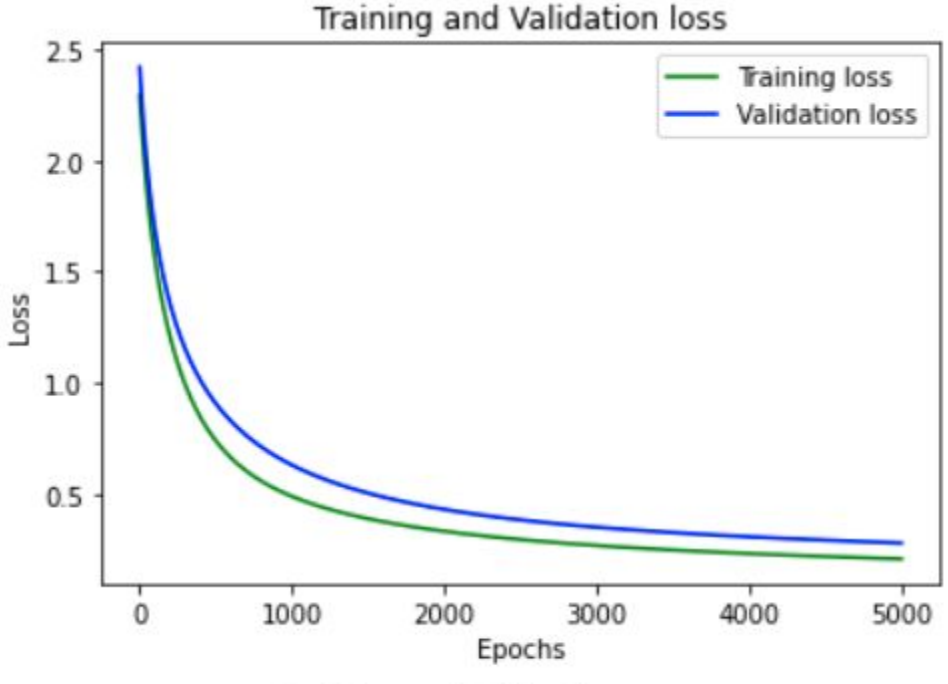


Figure-7: Loss value for validation and training data with α = 0.001

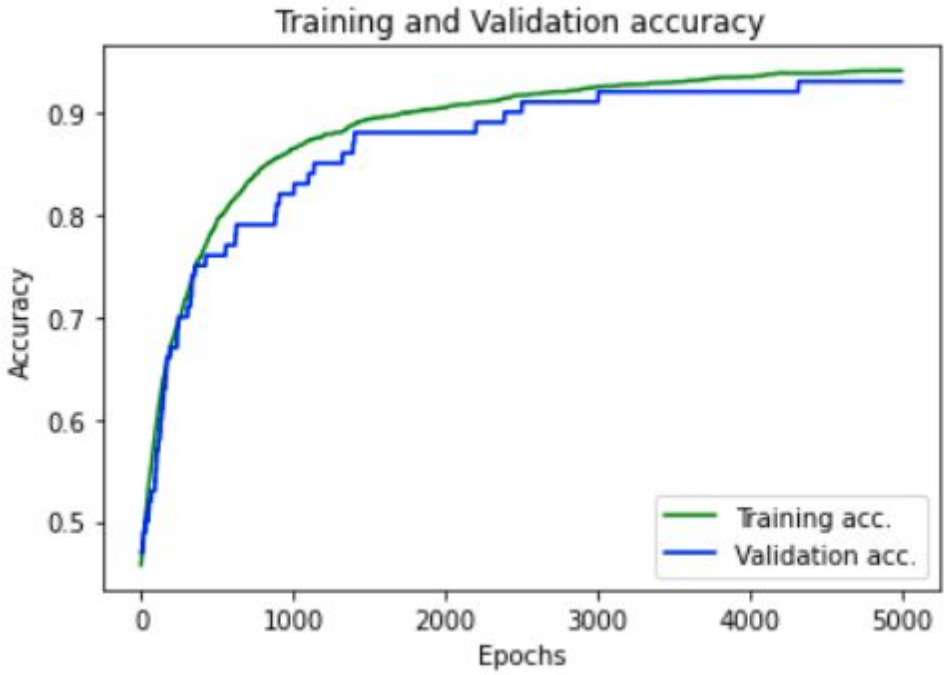


Figure-8: Accuracy value for validation and training data with α = 0.001

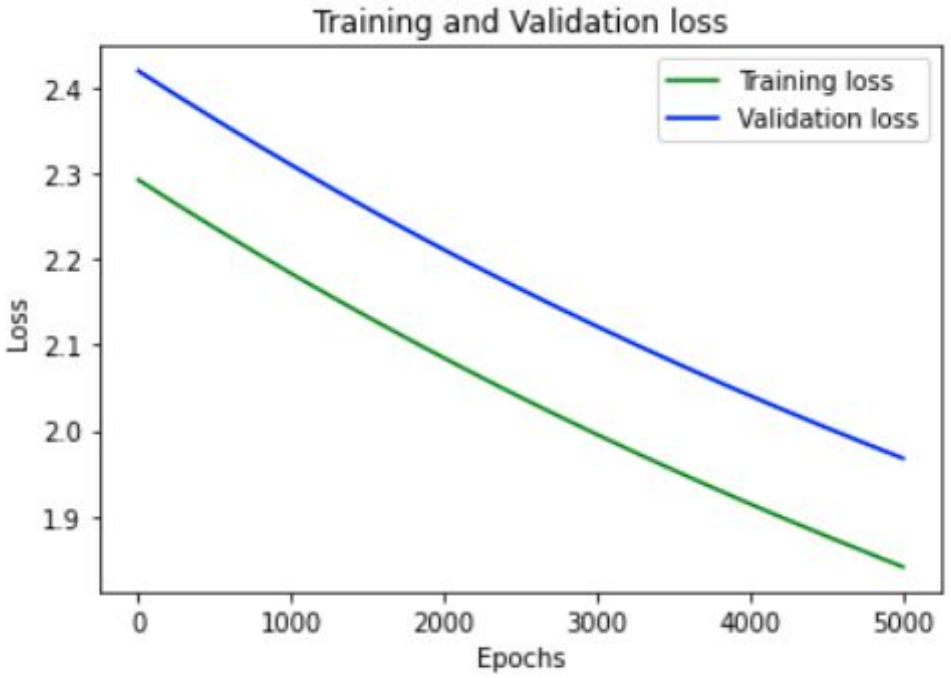


Figure-9: Loss value for validation and training data with α = 0.00001

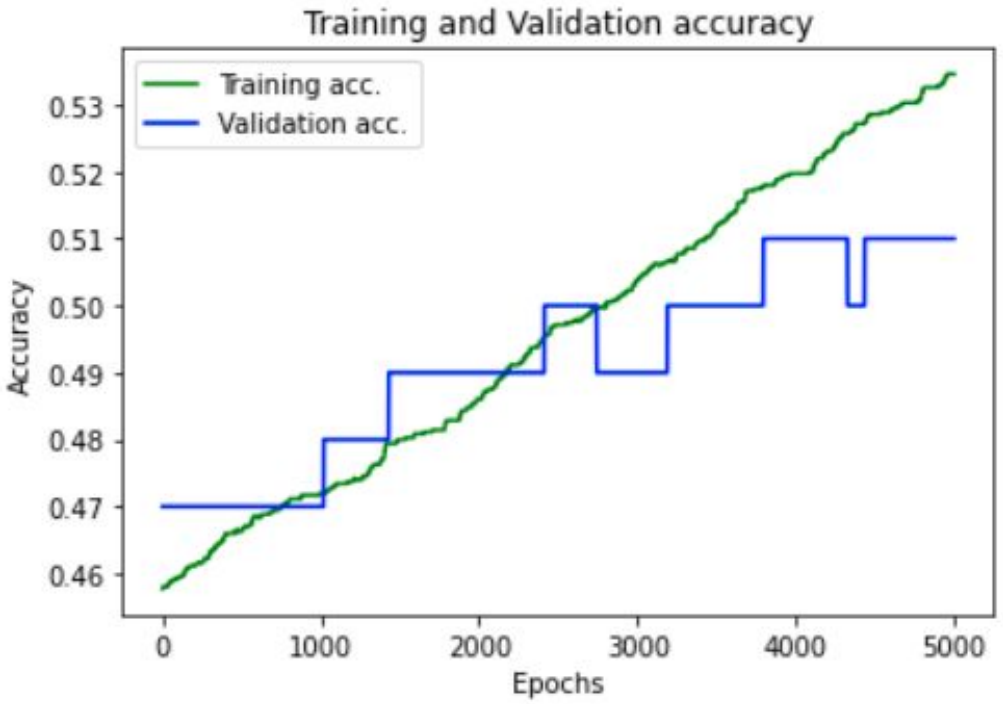


Figure-10: Accuracy value for validation and training data with α = 0.00001

1.4 Generalization

In this section, the effect of the regularization parameter on the loss/accuracy values is investigated and the results could be seen in Figures 11, 12, 13, 14, 15, and 16. The accuracy and loss values were not affected significantly. On the other hand, the time (i.e number of epochs) it takes the model to converge to the same level of accuracy and decrease the loss value was affected by the chosen regularization value. Amongst the three regularization values, λ = 0.5 had the fastest convergence and the largest decrease in loss, while maintaining the same level of validation accuracy with the rest of the values suggested. Therefore the model with **λ = 0.5 was chosen which had a final test accuracy of 0.965.**

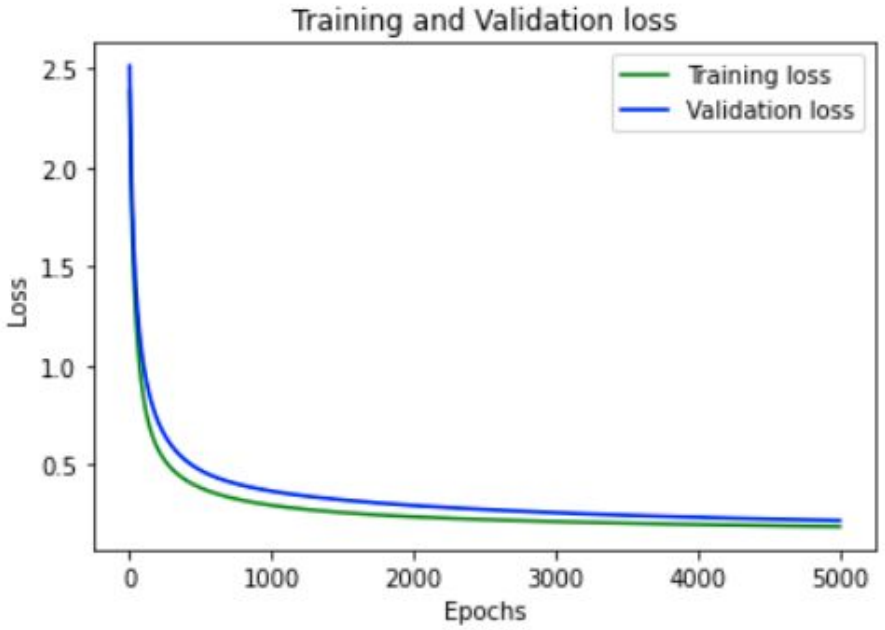


Figure-11: Training and validation loss with λ = 0.001

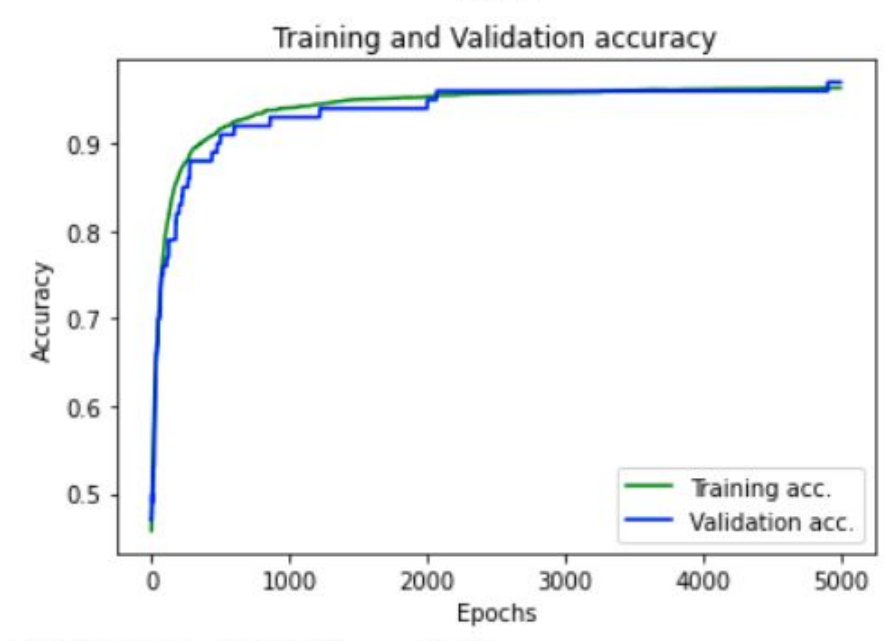


Figure-12: Training and validation accuracy with λ = 0.001

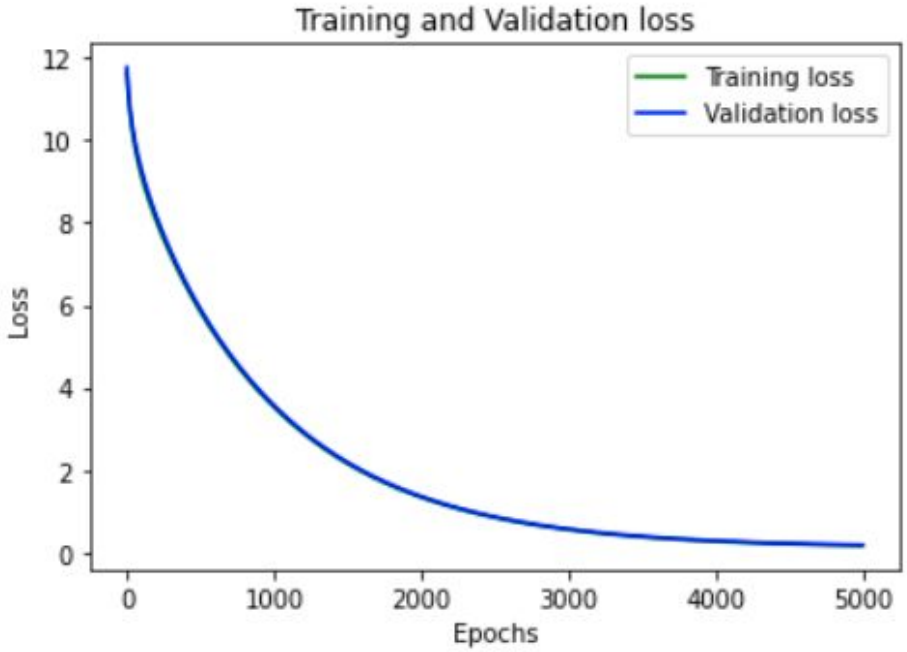


Figure-13: Training and validation loss with λ = 0.1

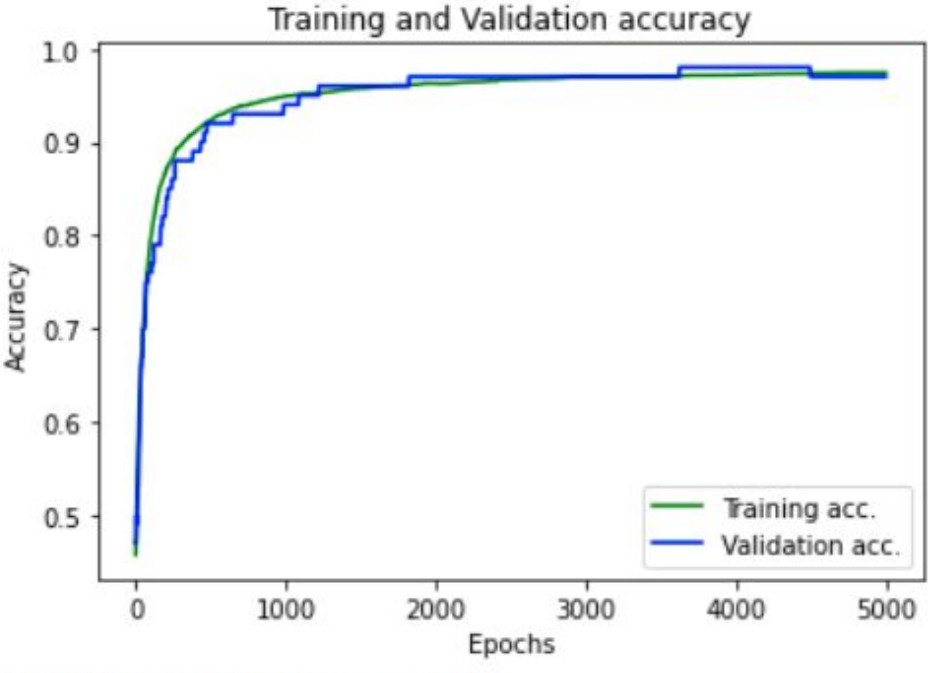


Figure-14: Training and validation accuracy with λ = 0.1

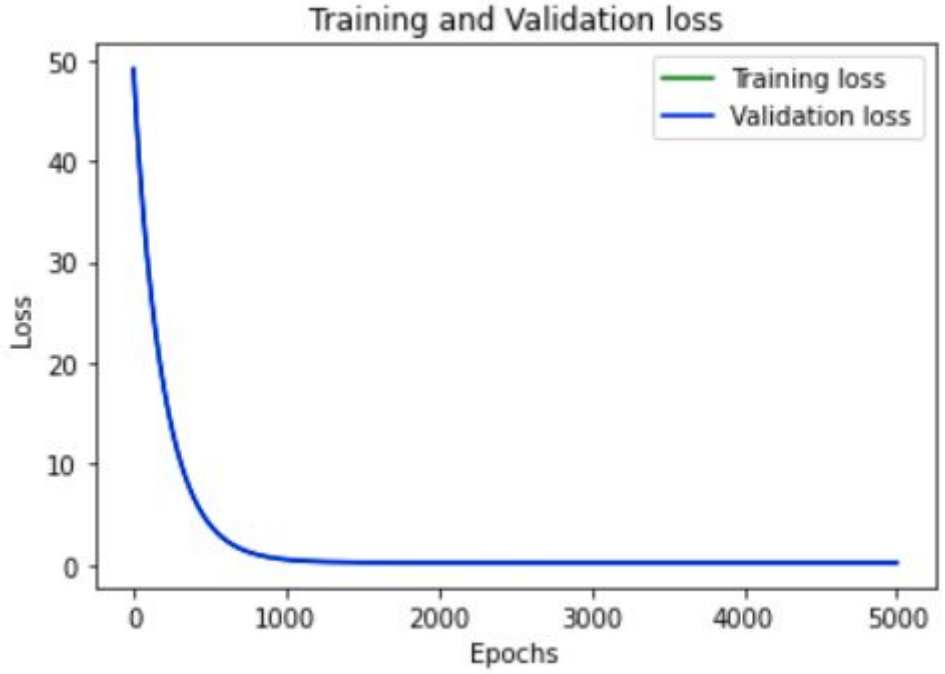


Figure-15: Training and validation loss with λ = 0.5

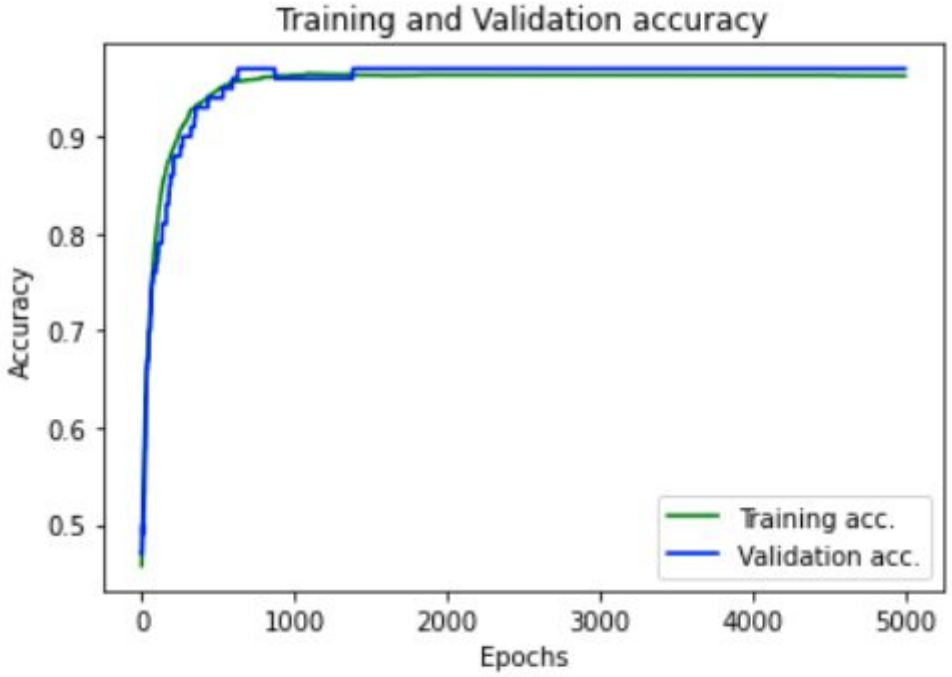


Figure-16: Training and validation accuracy with λ = 0.5

2. Logistic Regression in TensorFlow

2.1 Building the Computational Graph

The initialization of variables and placeholders in the graph could be seen in Figure 17. After allocating the necessary variables and placeholders, the prediction value is calculated, in order to be used in the loss function. Finally, the loss value is passed on to the optimizer to minimize by changing the variables, and the desired values are returned from the graph.

```
def buildGraph():
    W = tf.Variable(tf.random.truncated_normal(shape=(784,1), \
        mean=0.0, stddev=1.0, dtype=tf.dtypes.float32, seed=None, name=None))
    b = tf.Variable(0.0)
    x = tf.placeholder(tf.float32, name='x')
    y = tf.placeholder(tf.float32, name='y')
    reg = tf.placeholder(tf.float32, name='reg')
    prediction = tf.matmul(x,W) + b

    loss = tf.losses.sigmoid_cross_entropy(y, prediction) \
        + ((reg/2) * tf.matmul(W,tf.transpose(W)))

    optimizer = tf.train.AdamOptimizer(learning_rate = 0.001)
    training_op = optimizer.minimize(loss=loss)

    return W, b, x, prediction, y, loss, optimizer, training_op, reg
```

Figure-17: Building the computational graph in Tensorflow

2.2 Implementing SGD

As the first step, the number of batches to iterate over is calculated as seen in Figure-18.

```
W, b, x, prediction, y, loss, optimizer, train, reg= buildGraph()
epochs = 700
batch_size = 500
reg_val = 0
# calculate the number of batches
batches = np.shape(trainData)[0] / batch_size
```

Figure-18: Number of batches to iterate over

Then, the SGD algorithm is implemented. Following the suggestion in the handout, the input array and labels are shuffled before every new iteration. The Python implementation of the SGD algorithm could be seen in Figure-19, which utilizes the build graph, but with the addition of validation and test data for plotting purposes.

```
init_op = tf.global_variables_initializer()
# Train loop here
with tf.Session() as sess:
    sess.run(init_op)
    random_indexes = np.random.permutation(len(trainData))
    for step in range(epochs):
        # Shuffle here
        trainData_shuffled = trainData[random_indexes]
        trainTarget_shuffled = trainTarget[random_indexes]
        batch_start = 0
        for batch in range(int(batches)):
            train_batch = trainData_shuffled[batch_start:batch_start + batch_size, :]
            target_batch = trainTarget_shuffled[batch_start:batch_start + batch_size, :]
            batch_start += batch_size

            feed_dict={x:train_batch, y:target_batch, reg: reg_val, val_x: validData, \
                val_y: validTarget, test_x: testData, test_y: testTarget}

            _, current_loss, current_prediction, current_weight, current_bias, \
                cur_val_loss, cur_val_prediction, cur_test_loss, cur_test_prediction = \
                sess.run([train, loss, prediction, W, b, val_loss, val_prediction, \
                test_loss, test_prediction], feed_dict=feed_dict)
```

Figure-19: SGD Algorithm implementation with Tensorflow

The SGD algorithm was run with $\lambda = 0$, $\alpha = 0.001$, and a minibatch size of 500 optimizing over 700 epochs. The results could be seen in Figure-20 and Figure-21.

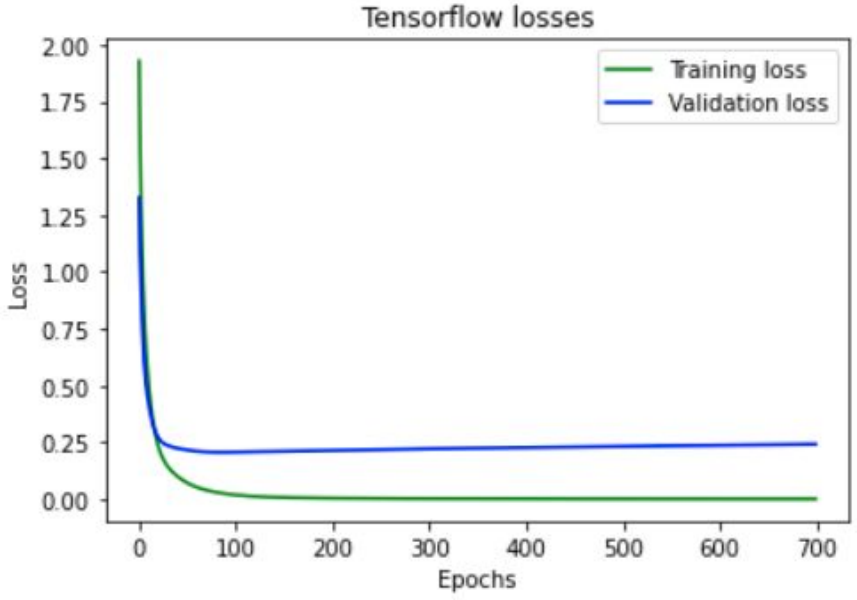
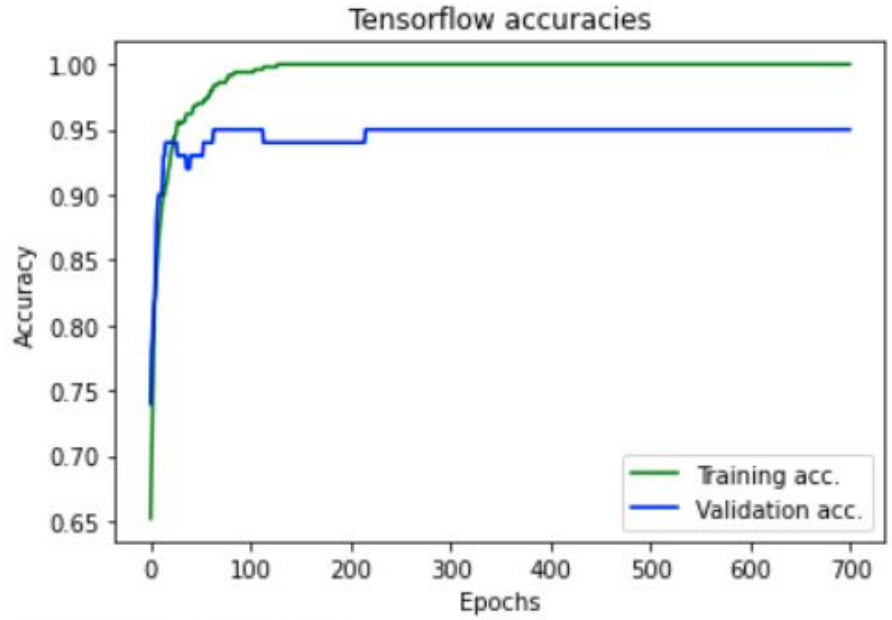


Figure-20: SGD initial loss results




```
Final train accuracy =  1.0
Final validation accuracy =  0.95
Final test accuracy =  0.9724137931034482
```

Figure-21: SGD initial accuracy results

2.3 Batch Size Investigation

The model was run with batch sizes of B = {100, 700, 1750}, λ = 0, and α = 0.001 over 700 epochs. It was observed that the model with B = 700 had the highest final accuracy. Taking a too small or too large of a batch size ended up reducing the accuracy. Final test accuracies were {0.94, 0.97, 0.95} respectively. The first observation was the training time, since there was a clear increase in the inference time of the model as the batch size decreased.

Having a large batch size is sometimes desirable since it might help parallelize the operations performed and provide faster convergence. However, it is possible to end up with a lower accuracy, because with larger batch size, there will be fewer iterations to optimize the weights. Thus, it is understandable that having B = 1750 has a lower accuracy than the B = 700. It is important to note that this trade-off might provide value for larger datasets, to reduce the training time.

On the other side of the spectrum, having a batch size that is too small might decrease the model's accuracy as well. If the batch the model is optimizing over is too small, it might not reflect the general features that are needed to "learn", and would take steps in the wrong direction due to the inaccurate gradient value.

The results are shown in Figures 22, 23, 24, 25, 26, and 27.

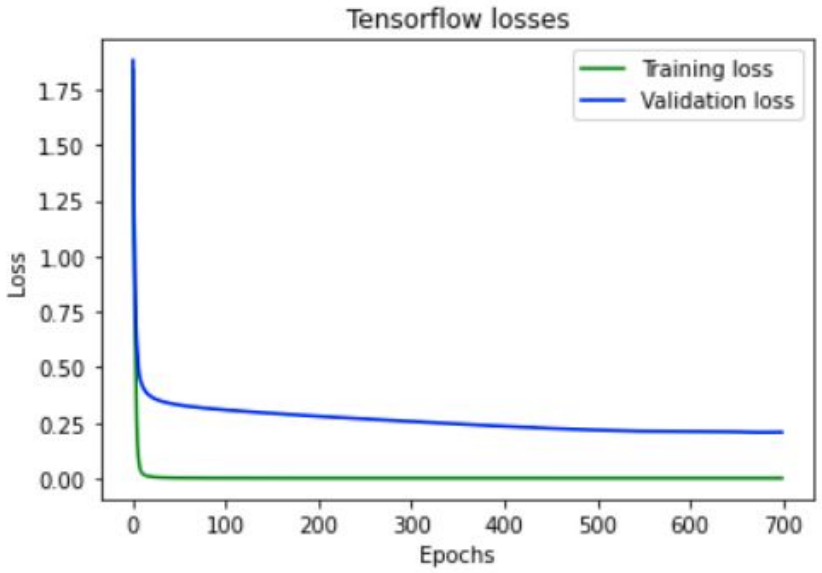


Figure-22: Training and validation loss with B = 100

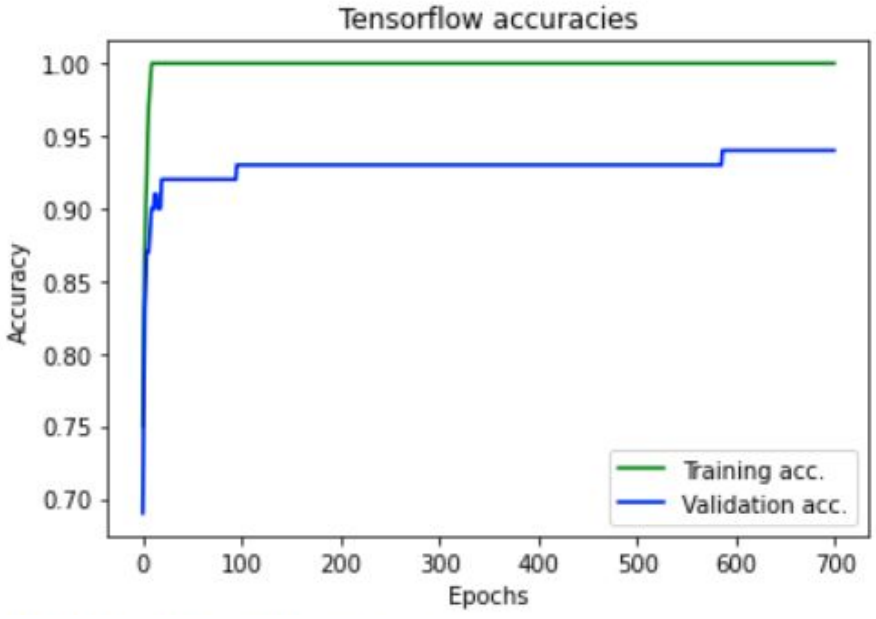


Figure-23: Training and validation accuracy with B = 100

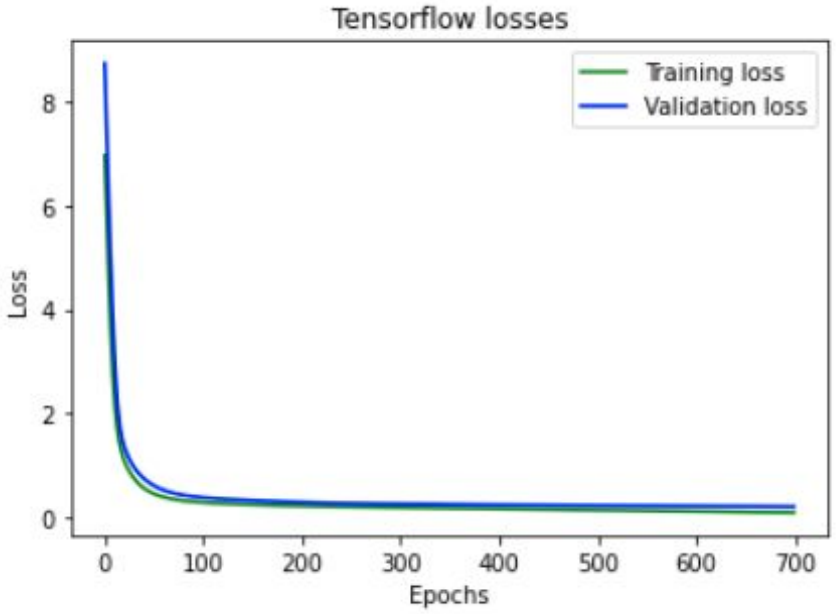


Figure-24: Training and validation loss with B = 700

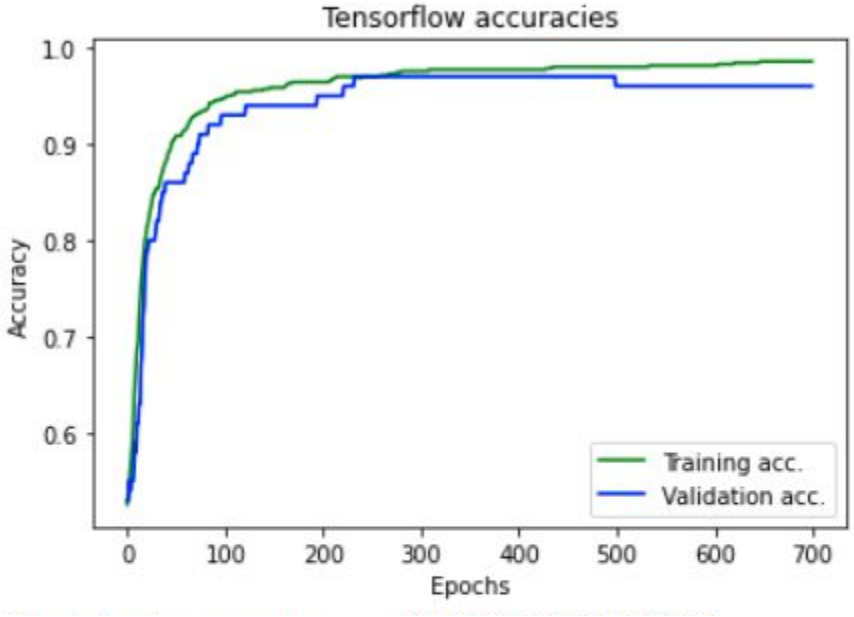


Figure-25: Training and validation accuracy with B = 700

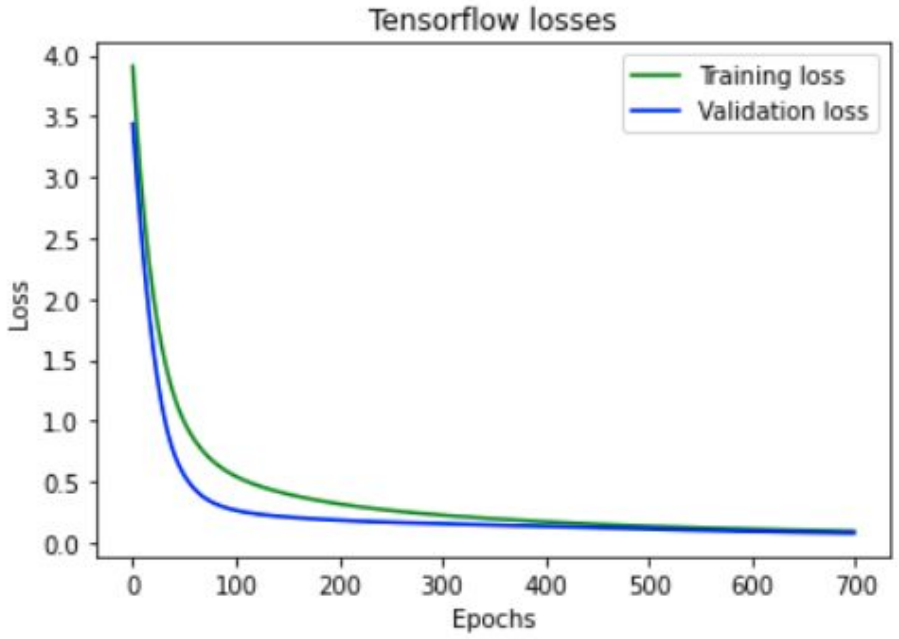


Figure-26: Training and validation loss with B = 1750

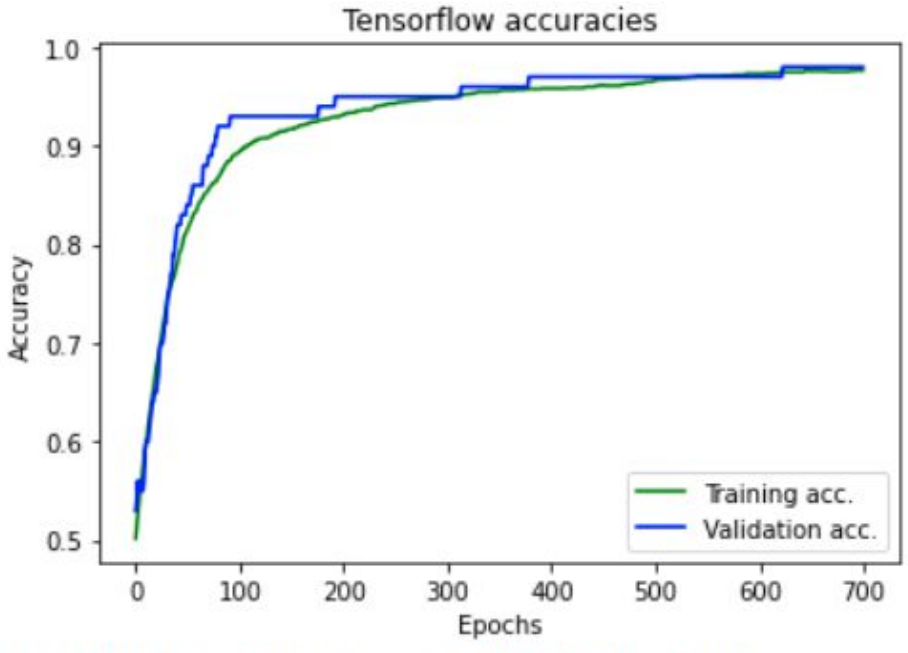


Figure-27: Training and validation accuracy with B = 1750

2.4 Hyperparameter Investigation

In this part, different values for the optimizer (Adam) parameters were tested. For the sake of simplicity, the results are shown in a table. Graphs could be found in Appendix-1.

β_1 is the first hyperparameter that was investigated. As seen in Table-1, compared to β_1 = 0.95, β_1 = 0.99 had a higher test accuracy, lower validation and test losses, while having similar validation and training accuracies, as well as a similar training loss. **Thus, β_1 = 0.99 was chosen as the first hyperparameter, with final test accuracy of 0.959.**

Next, effect of β_2 was investigated. All the accuracy values were very similar, thus the losses were inspected to decide on a value for selection of this hyperparameter. β_2 = 0.99 had smaller train, validation, and test losses. **Therefore, β_2 = 0.99 was chosen as the second hyperparameter, with final test accuracy of 0.965.**

The third and last parameter to be investigated was ϵ. Validation accuracy values were too close to make a meaningful distinction between the two hyperparameters. However ϵ = 1e-9 had a higher test accuracy, and lower test and validation losses compared to ϵ = 1e-4. **Thus ϵ = 1e-9 was chosen, with final test accuracy of 0.979**

	Train Acc.	Val. Acc.	Test Acc.	Train Loss	Val. Loss	Test Loss
β_1 = 0.95	0.988	0.980	0.959	0.031	0.079	0.234
β_1 = 0.99	0.988	0.960	0.972	0.032	0.105	0.184
β_2 = 0.99	1.000	0.960	0.965	0.009	0.103	0.177
β_2 = 0.9999	0.982	0.960	0.965	0.055	0.168	0.231
ϵ = 1e-9	0.992	0.980	0.979	0.027	0.113	0.205
ϵ = 1e-4	0.988	0.980	0.965	0.027	0.124	0.208

Table-1: Hyperparameter comparison

2.5 Comparison against Batch GD

The clear difference between Batch GD and SGD with Adam is the number of epochs it takes for the algorithm to converge. SGD with Adam was observed to converge usually around 200 epochs, while Batch GD algorithm often took more than 1500 epochs to converge.

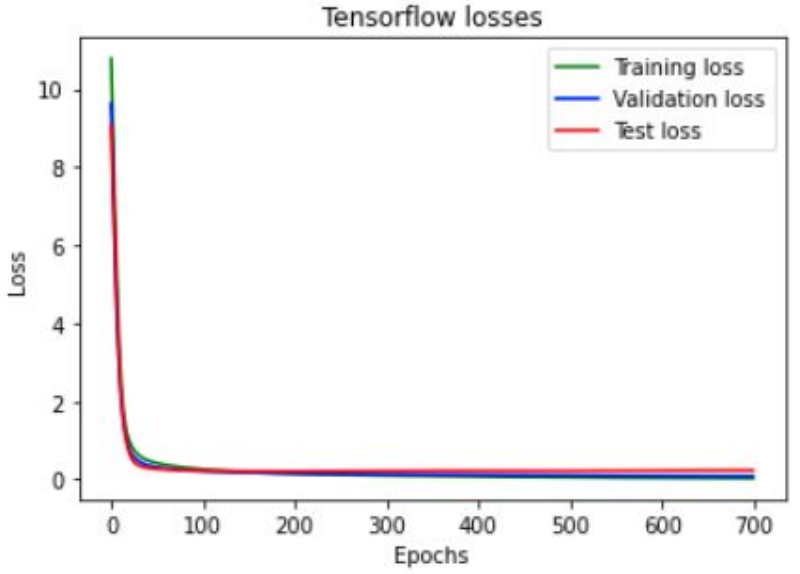
In terms of accuracy and loss values, with the ideal hyperparameters chosen as described in the previous sections, there was not a significant difference observed given enough training time.

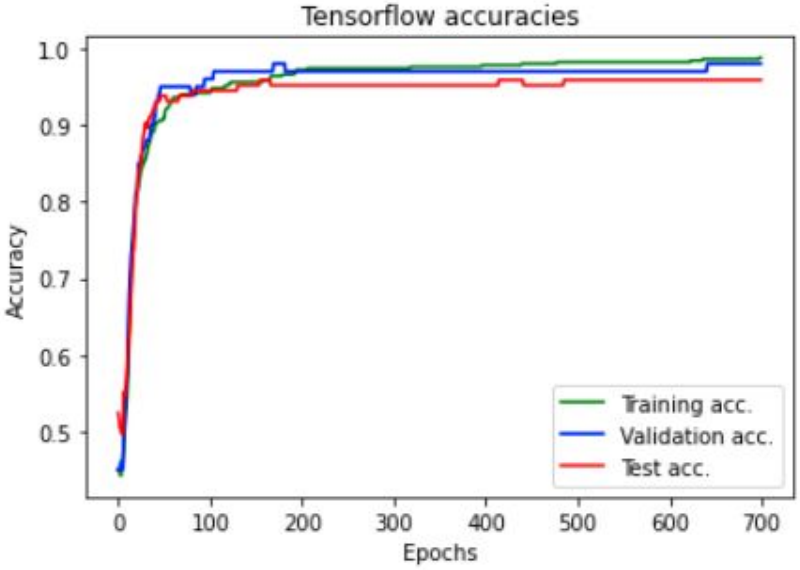
The reason behind the difference in convergence time is having smaller batches instead of the whole dataset and also using the Adam optimizer, which could choose better steps towards the correct direction. Both these factors would have an effect on decreasing the time it takes to converge.

References:

[1]: Y. S. Abu-Mostafa, M. Magdon-Ismail , and H. Lin, "The Linear Model" in *Learning From Data.* 2012, ch 3.3, pp. 95

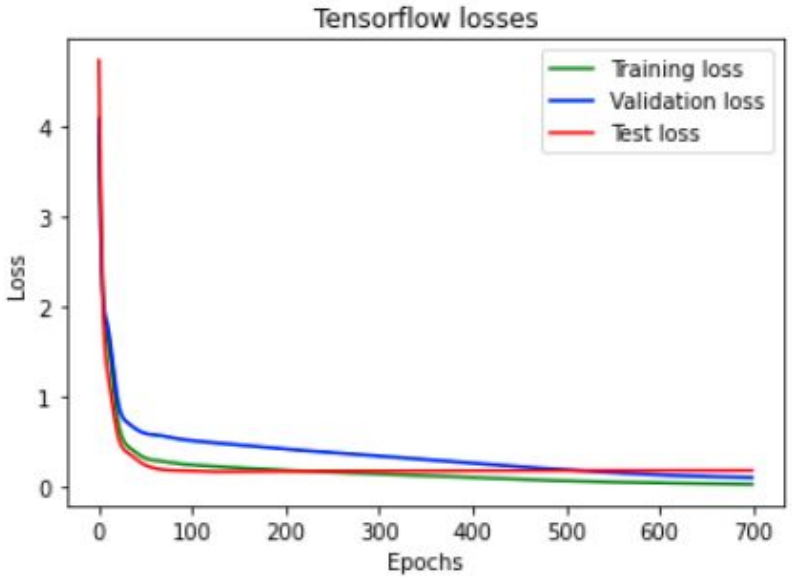
APPENDIX-1:
Figures for Hyperparameter Investigation (Part 2.4)

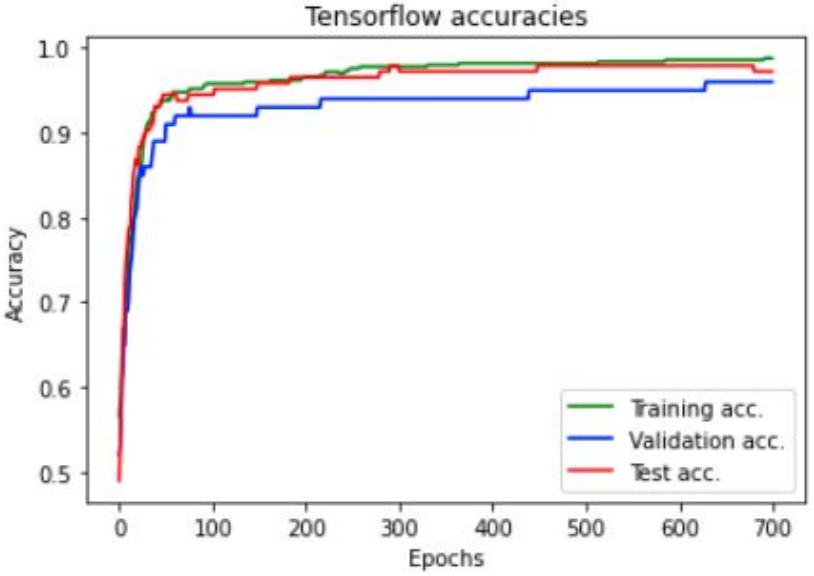





```
Final train accuracy =  0.988
Final validation accuracy =  0.98
Final test accuracy =  0.9586206896551724
Final train loss =  0.030847352
Final validation loss =  0.078612305
Final test loss =  0.23397137
```

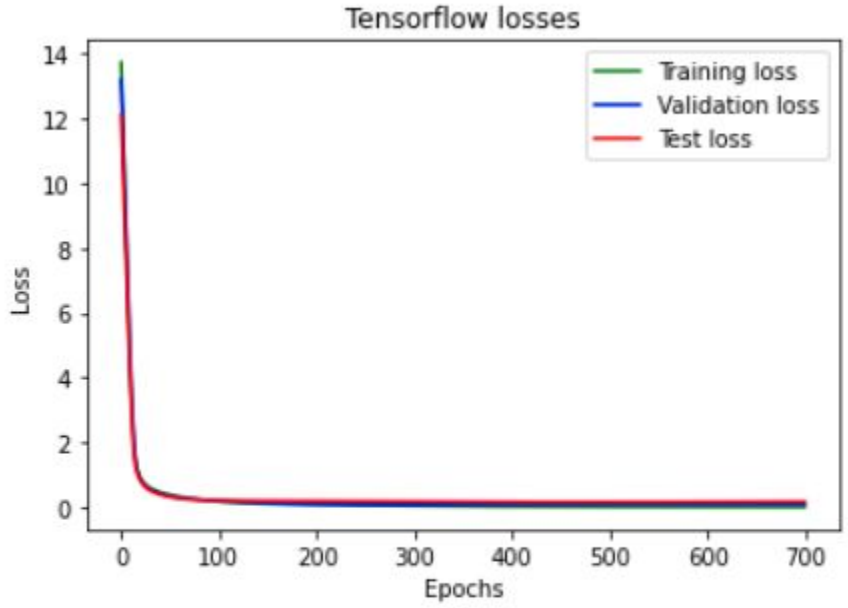
Figure A-1: $\beta_1 = 0.95$

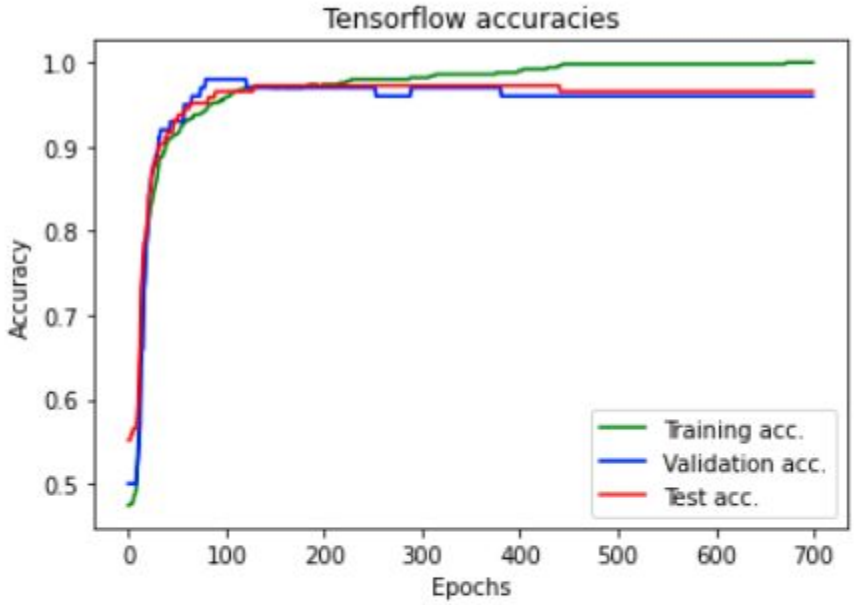





```
Final train accuracy =  0.988
Final validation accuracy =  0.96
Final test accuracy =  0.9724137931034482
Final train loss =  0.031550374
Final validation loss =  0.10653388
Final test loss =  0.18423572
```

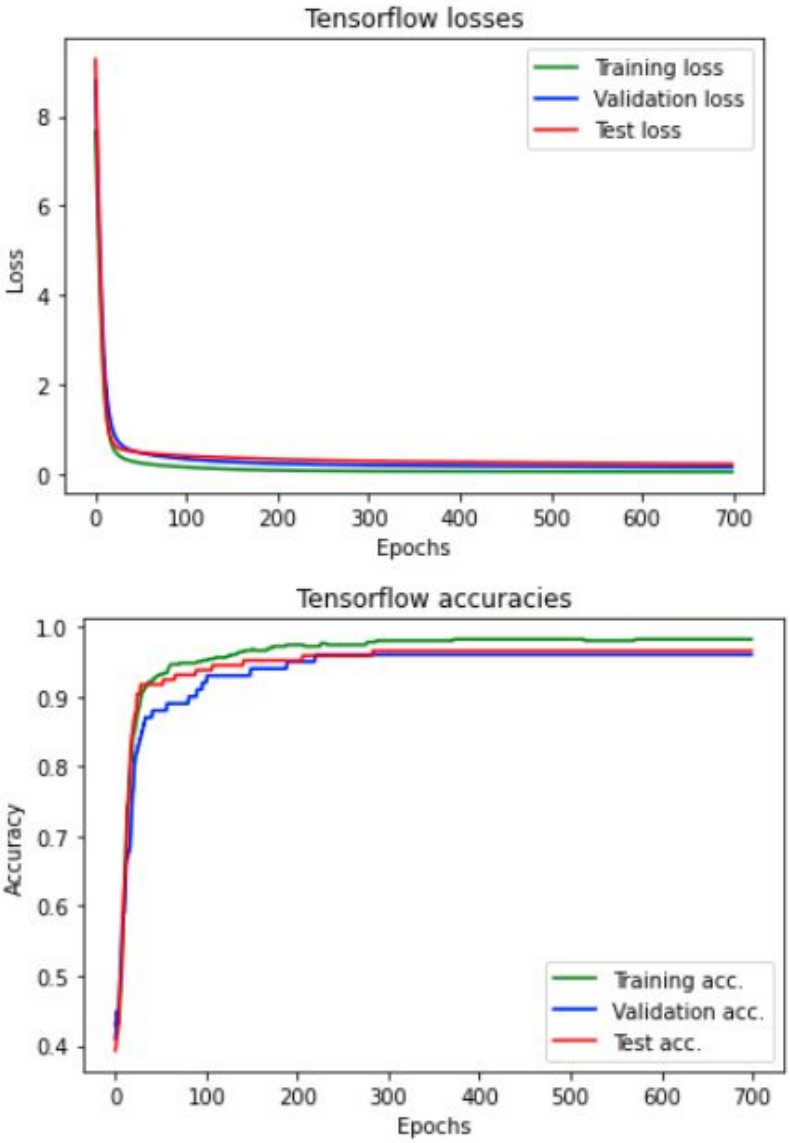
Figure A-2: $\beta_1 = 0.99$





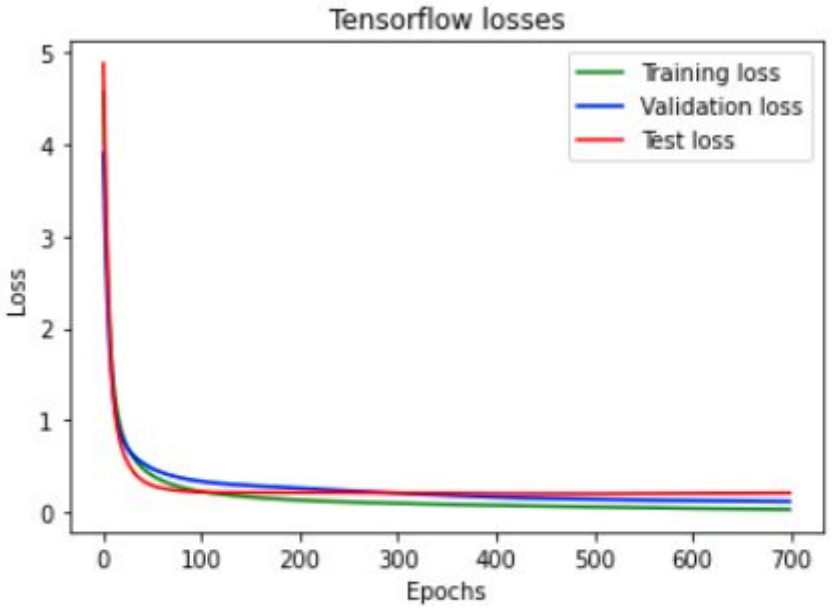

```
Final train accuracy =  1.0
Final validation accuracy =  0.96
Final test accuracy =  0.9655172413793104
Final train loss =  0.009196476
Final validation loss =  0.10330382
Final test loss =  0.17655468
```

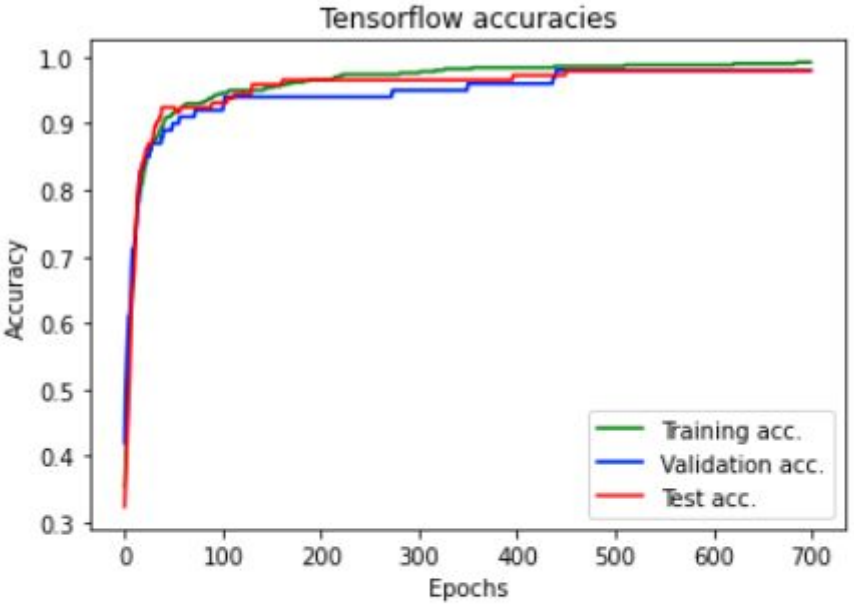
Figure A-3: $\beta_2 = 0.99$




```
Final train accuracy =  0.982
Final validation accuracy =  0.96
Final test accuracy =  0.9655172413793104
Final train loss =  0.05649092
Final validation loss =  0.1675682
Final test loss =  0.23142081
```

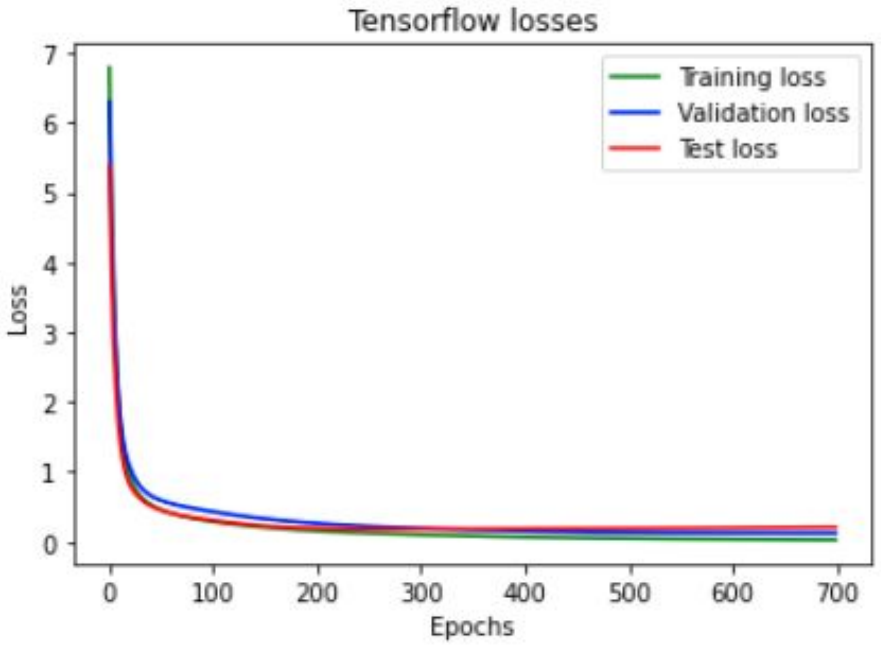
Figure A-4: $\beta_2 = 0.9999$

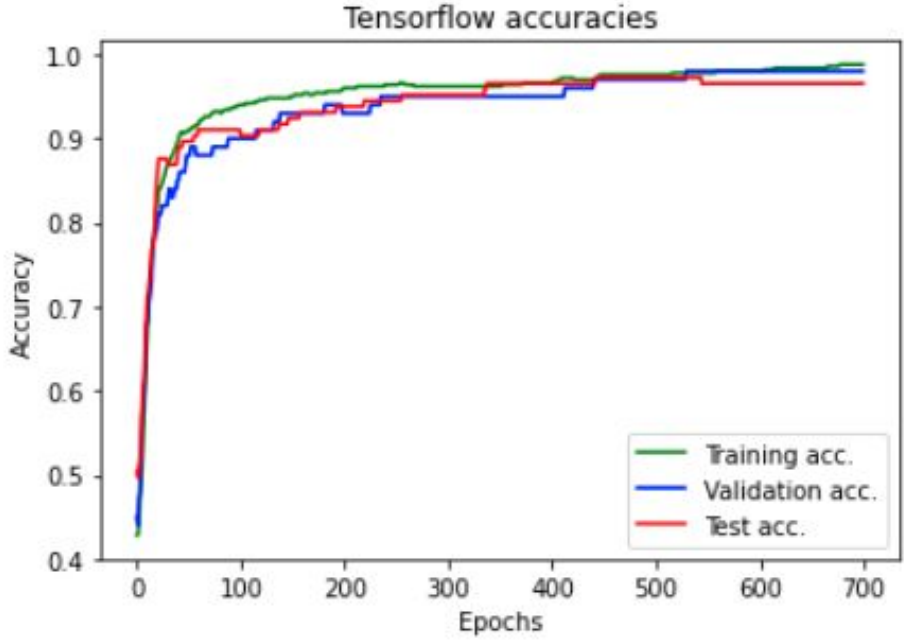





```
Final train accuracy =  0.992
Final validation accuracy =  0.98
Final test accuracy =  0.9793103448275862
Final train loss =  0.027018294
Final validation loss =  0.1132256
Final test loss =  0.20525652
```

Figure A-5: ϵ = 1e-9






```
Final train accuracy =  0.988
Final validation accuracy =  0.98
Final test accuracy =  0.9655172413793104
Final train loss =  0.026914908
Final validation loss =  0.12398939
Final test loss =  0.20792338
```

Figure A-6: ϵ = 1e-4