Jia Ming (Carter) Huang (Student ID: 1003893245)

	P1	P2	Total
Mark			

Table 1: Marking Table

Submission Date: Monday, February 7, 2022, 11:59 PM
Number of Pages: 35

Contents

1 Logistic Regression with Numpy [20 points]

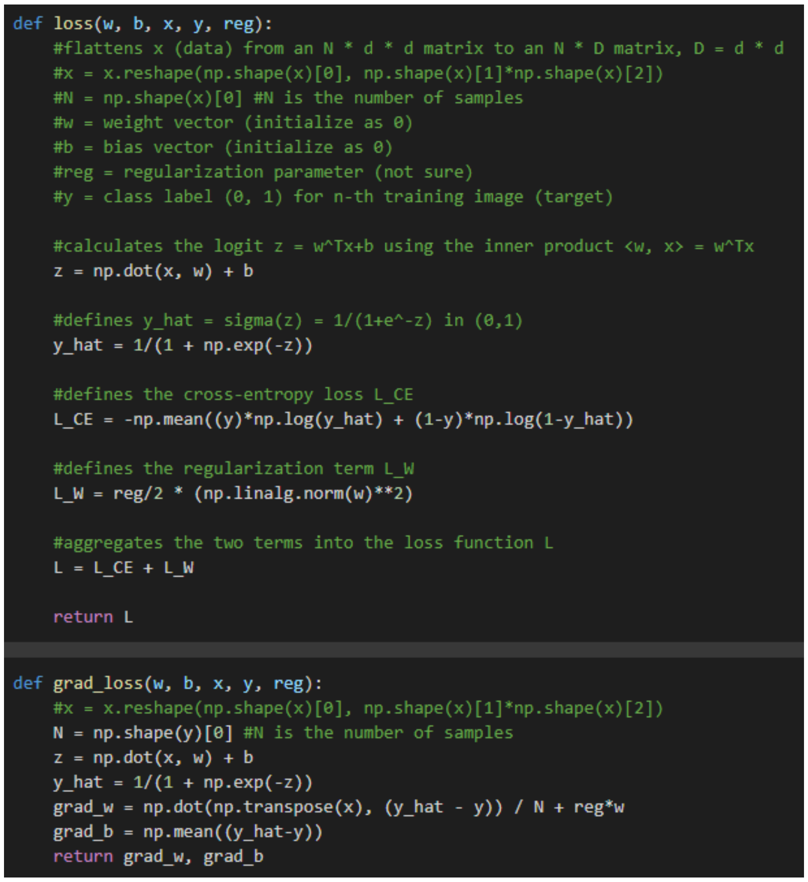
The cross-entropy loss $\mathcal{L}_{\text{CE}}$ and the regularization term $\mathcal{L}_{\text{w}}$ will form the total loss function as:

$$\begin{aligned}\mathcal{L} &= \mathcal{L}_{\text{CE}} + \mathcal{L}_{\text{w}} \\ &= \frac{1}{N}\sum_{n=1}^{N}\left[-y^{(n)}\log \hat{y}\left(\mathbf{x}^{(n)}\right) - \left(1-y^{(n)}\right)\log\left(1-\hat{y}\left(\mathbf{x}^{(n)}\right)\right)\right] + \frac{\lambda}{2}\|\mathbf{w}\|_2^2\end{aligned}$$

In logistic regression, we model the probability of a sample $\mathbf{x}$ belonging to the positive class as

$$\hat{y}(\mathbf{x}) = \sigma\left(\mathbf{w}^\top\mathbf{x} + b\right),$$

where $z = \mathbf{w}^\top\mathbf{x} + b$, also called logit, is basically the linear transformation of input vector $\mathbf{x}$ using weight vector $\mathbf{w}$ and bias scalar b. $\sigma(z) = 1/(1+\exp(-z))$ is the sigmoid or logistic function: it "squashes" the real-valued logits to fall between zero and one. Figure 1 shows the loss and grad_loss implementation in Python.



```python
def loss(w, b, x, y, reg):
    #flattens x (data) from an N * d * d matrix to an N * D matrix, D = d * d
    #x = x.reshape(np.shape(x)[0], np.shape(x)[1]*np.shape(x)[2])
    #N = np.shape(x)[0] #N is the number of samples
    #w = weight vector (initialize as 0)
    #b = bias vector (initialize as 0)
    #reg = regularization parameter (not sure)
    #y = class label (0, 1) for n-th training image (target)

    #calculates the logit z = w^Tx+b using the inner product <w, x> = w^Tx
    z = np.dot(x, w) + b

    #defines y_hat = sigma(z) = 1/(1+e^-z) in (0,1)
    y_hat = 1/(1 + np.exp(-z))

    #defines the cross-entropy loss L_CE
    L_CE = -np.mean((y)*np.log(y_hat) + (1-y)*np.log(1-y_hat))

    #defines the regularization term L_W
    L_W = reg/2 * (np.linalg.norm(w)**2)

    #aggregates the two terms into the loss function L
    L = L_CE + L_W

    return L

def grad_loss(w, b, x, y, reg):
    #x = x.reshape(np.shape(x)[0], np.shape(x)[1]*np.shape(x)[2])
    N = np.shape(y)[0] #N is the number of samples
    z = np.dot(x, w) + b
    y_hat = 1/(1 + np.exp(-z))
    grad_w = np.dot(np.transpose(x), (y_hat - y)) / N + reg*w
    grad_b = np.mean((y_hat-y))
    return grad_w, grad_b
```

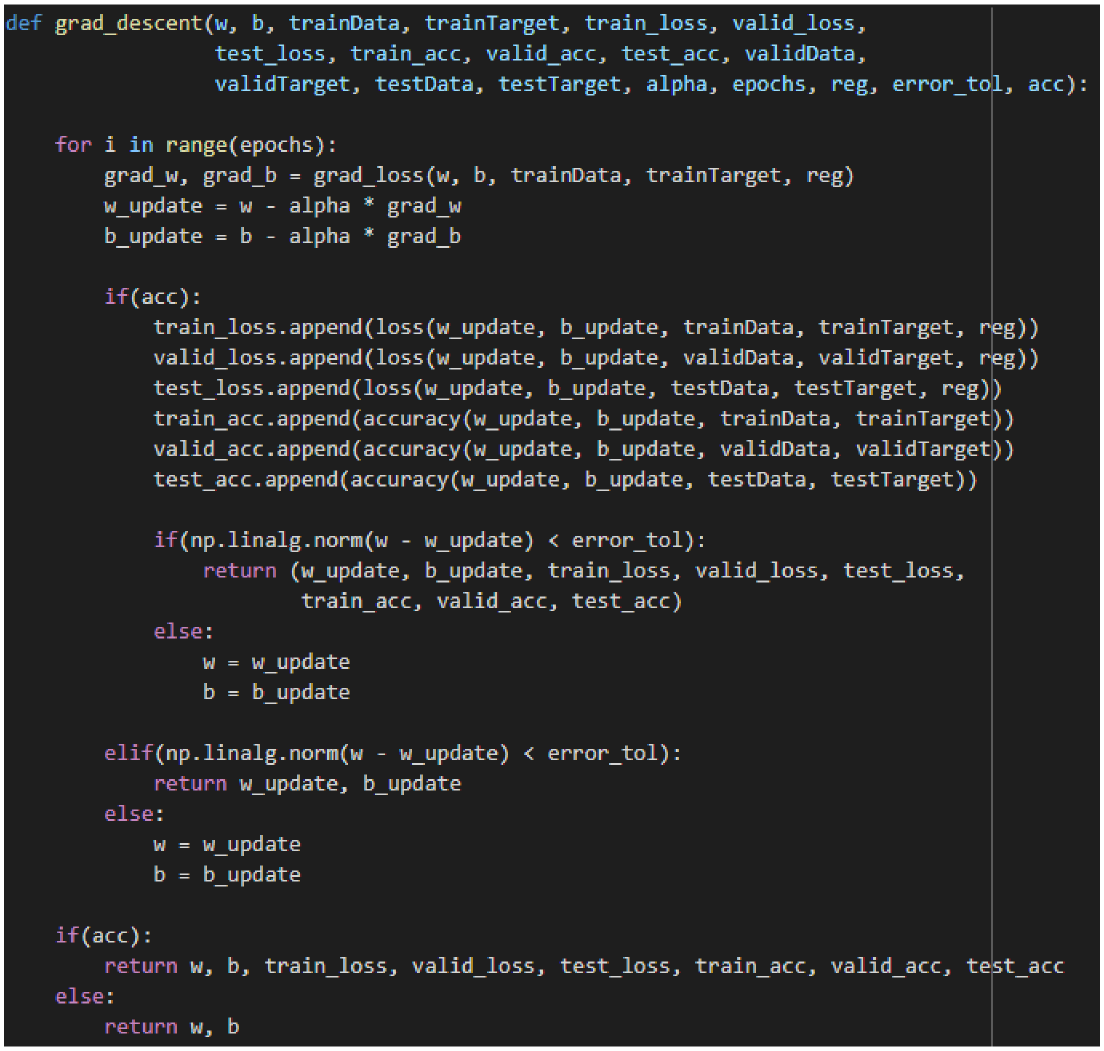
Figure 1: Implementation of the loss() and grad_loss() functions in Python

The gradient loss function implemented in Figure 1 is derived as follows:

$$
\begin{aligned}
\frac{\partial\mathcal{L}}{\partial \mathrm{w}} &= \frac{\partial\mathcal{L}_{\mathrm{CE}}}{\partial \mathrm{w}} + \frac{\partial\mathcal{L}_{\mathrm{w}}}{\partial \mathrm{w}} \\
&= \frac{\partial}{\partial \mathrm{w}}\frac{1}{N}\sum_{n=1}^{N}[-y\log(\hat{y}(\mathbf{x})) - (1-y)\log(1-\hat{y}(\mathbf{x}))] + \frac{\lambda}{2}\|\mathbf{w}\|_2^2 \\
&= \frac{1}{N}\sum_{n=1}^{N}\frac{\partial}{\partial \mathrm{w}}[-y\log(\hat{y}(\mathbf{x})) - (1-y)\log(1-\hat{y}(\mathbf{x}))] + \frac{\lambda}{2}\|\mathbf{w}\|_2^2 \\
&= \frac{1}{N}\sum_{n=1}^{N}\frac{\partial}{\partial \mathrm{w}}[-y\log(\frac{1}{1+e^{-(wx+b)}}) - (1-y)\log(1-\frac{1}{1+e^{-(wx+b)}})] + \frac{\lambda}{2}\|\mathbf{w}\|_2^2 \\
&= \frac{1}{N}\sum_{n=1}^{N}\frac{\partial}{\partial \mathrm{w}}[-y\log(\frac{1}{1+e^{-z}}) - (1-y)\log(1-\frac{1}{1+e^{-z}})] + \frac{\lambda}{2}\|\mathbf{w}\|_2^2 \\
&= [\frac{\partial\mathcal{L}_{\mathrm{CE}}}{\partial \mathrm{z}}][\frac{\partial \mathrm{z}}{\partial \mathrm{w}}] + \mathrm{w} \\
&= [\frac{1}{N}\sum_{n=1}^{N}\frac{\partial}{\partial \mathrm{z}}[-y\log(\frac{1}{1+e^{-z}}) - (1-y)\log(1-\frac{1}{1+e^{-z}})]][\frac{\partial}{\partial \mathrm{w}}(wx+b)] + \mathrm{w} \\
&= [\frac{1}{N}\sum_{n=1}^{N}((-y(1+e^{-z})(\frac{e^{-z}}{(1+e^{-z})^2})) - (\frac{1-y}{1-\frac{1}{1+e^{-z}}} - \frac{e^{-z}}{(1+e^{-z})^2}))][x] + \mathrm{w} \\
&= [\frac{1}{N}\sum_{n=1}^{N}((-y(\frac{e^{-z}}{1+e^{-z}})) + (\frac{1-y}{1-\frac{1}{1+e^{-z}}} - \frac{e^{-z}}{(1+e^{-z})^2}))][x] + \mathrm{w} \\
&= [\frac{1}{N}\sum_{n=1}^{N}((-\frac{e^{-z}}{1+e^{-z}})(y - (\frac{1-y}{1-\frac{1}{1-e^{-z}}})(\frac{1}{1+e^{-z}})))][x] + \mathrm{w} \\
&= [\frac{1}{N}\sum_{n=1}^{N}((-\frac{e^{-z}}{1+e^{-z}})(y - (\frac{1-y}{e^{-z}})))][x] + \mathrm{w} \\
&= [\frac{1}{N}\sum_{n=1}^{N}((-\frac{e^{-z}}{1+e^{-z}})(\frac{y(1+e^{-z})-1}{e^{-z}}))][x] + \mathrm{w} \\
&= [\frac{1}{N}\sum_{n=1}^{N}(\frac{1-y(1+e^{-z})}{1+e^{-z}})][x] + \mathrm{w} \\
&= [\frac{1}{N}\sum_{n=1}^{N}(\frac{1}{1+e^{-z}} - y)][x] + \mathrm{w} \\
&= \frac{1}{N}\sum_{n=1}^{N}[(\hat{y}-y)x] + \mathrm{w} \\
\frac{\partial\mathcal{L}}{\partial \mathrm{b}} &= \frac{1}{N}\sum_{n=1}^{N}\hat{y}-y
\end{aligned}
$$

QED.

Figure 2 shows the implementation of gradient descent on the notMNIST dataset. To observe tuning the learning rate, the implementation is tested for 5000 epochs with the regularization parameter $\lambda = 0$. The training and validation losses for $\alpha = 0.005$, $\alpha = 0.001$, and $\alpha = 0.0001$ are shown in Figures 3, 5, and 7, respectively. The training and validation accuracies for $\alpha = 0.005$, $\alpha = 0.001$, and $\alpha = 0.0001$ are shown in Figures 4, 6, and 8, respectively. To observe the impact of regularization, the implementation is tested for 5000 epochs with the learning rate $\alpha = 0.005$. The training and validation losses for $\lambda = 0.001$, $\lambda = 0.1$, and $\lambda = 0.5$ are shown in Figures 9, 11, and 13, respectively. The training and validation accuracies for $\lambda = 0.001$, $\lambda = 0.1$, and $\lambda = 0.5$ are shown in Figures 10, 12, and 14, respectively. Both sets of observations can be seen in Tables 2 and 3, respectively.



```python
def grad_descent(w, b, trainData, trainTarget, train_loss, valid_loss,
                 test_loss, train_acc, valid_acc, test_acc, validData,
                 validTarget, testData, testTarget, alpha, epochs, reg, error_tol, acc):

    for i in range(epochs):
        grad_w, grad_b = grad_loss(w, b, trainData, trainTarget, reg)
        w_update = w - alpha * grad_w
        b_update = b - alpha * grad_b

        if(acc):
            train_loss.append(loss(w_update, b_update, trainData, trainTarget, reg))
            valid_loss.append(loss(w_update, b_update, validData, validTarget, reg))
            test_loss.append(loss(w_update, b_update, testData, testTarget, reg))
            train_acc.append(accuracy(w_update, b_update, trainData, trainTarget))
            valid_acc.append(accuracy(w_update, b_update, validData, validTarget))
            test_acc.append(accuracy(w_update, b_update, testData, testTarget))

            if(np.linalg.norm(w - w_update) < error_tol):
                return (w_update, b_update, train_loss, valid_loss, test_loss,
                        train_acc, valid_acc, test_acc)
            else:
                w = w_update
                b = b_update

        elif(np.linalg.norm(w - w_update) < error_tol):
            return w_update, b_update
        else:
            w = w_update
            b = b_update

    if(acc):
        return w, b, train_loss, valid_loss, test_loss, train_acc, valid_acc, test_acc
    else:
        return w, b
```

Figure 2: Implementation of the grad_descent() function in Python

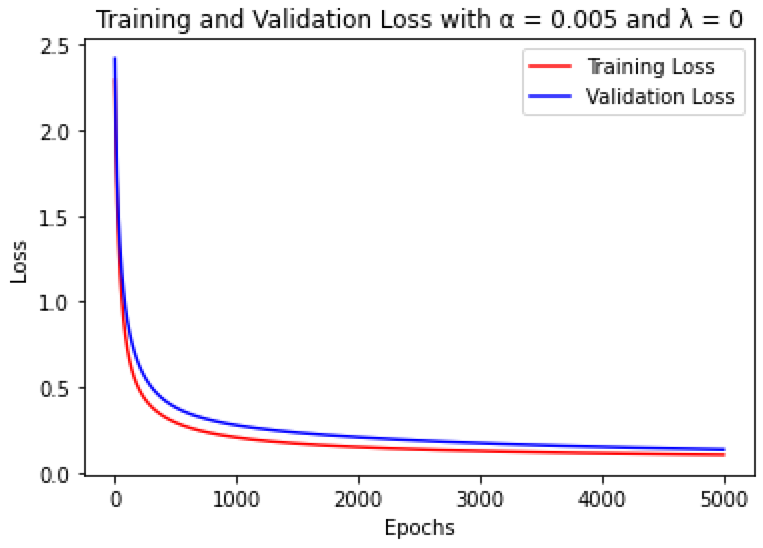


Figure 3: α = 0.005, λ = 0

.

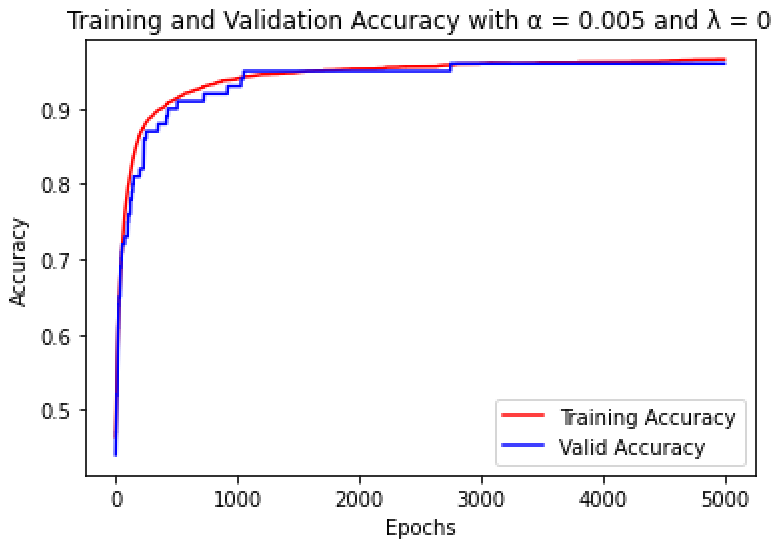


Figure 4: α = 0.005, λ = 0

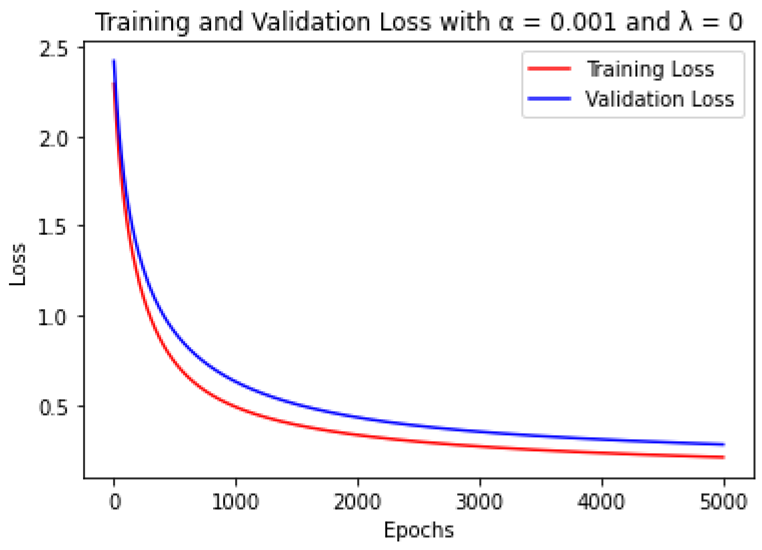


Figure 5: $\alpha = 0.001, \lambda = 0$

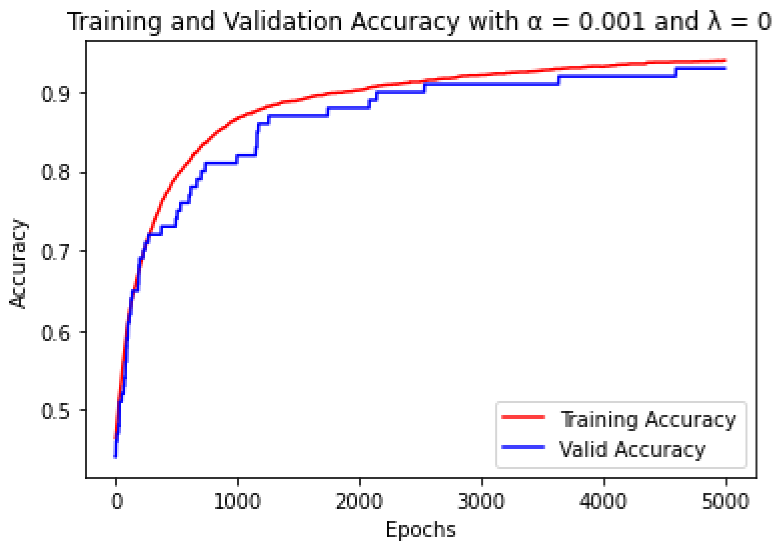


Figure 6: $\alpha = 0.001, \lambda = 0$

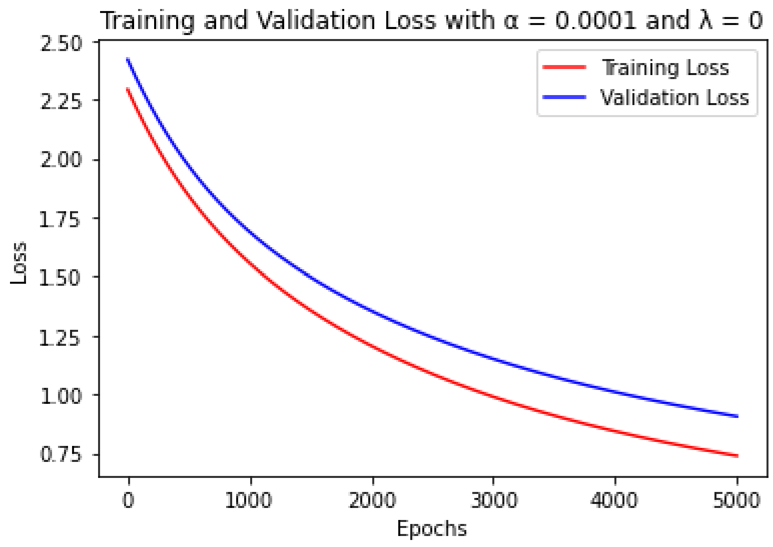


Figure 7: $\alpha = 0.0001, \lambda = 0$

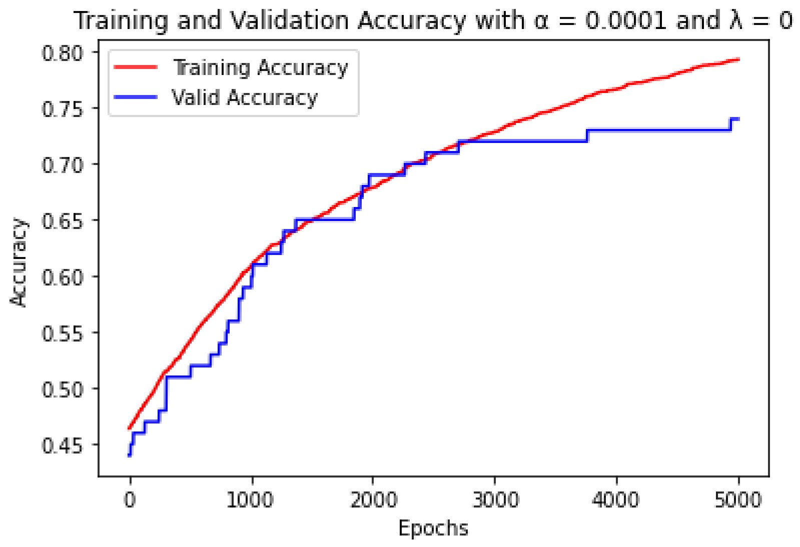


Figure 8: $\alpha = 0.0001, \lambda = 0$

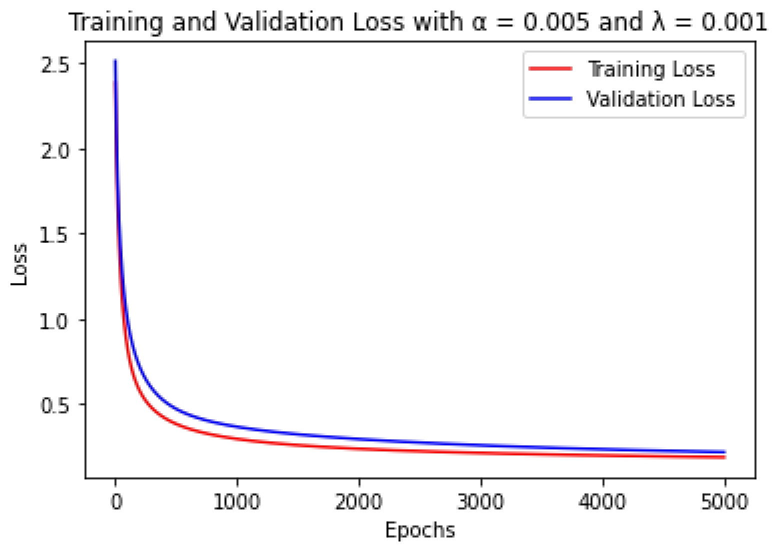


Figure 9: $\alpha = 0.005, \lambda = 0.001$

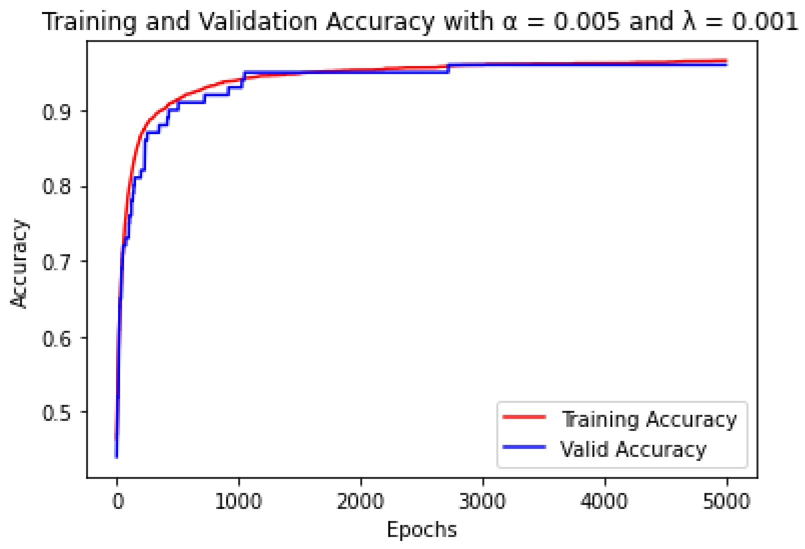


Figure 10: $\alpha = 0.005, \lambda = 0.001$

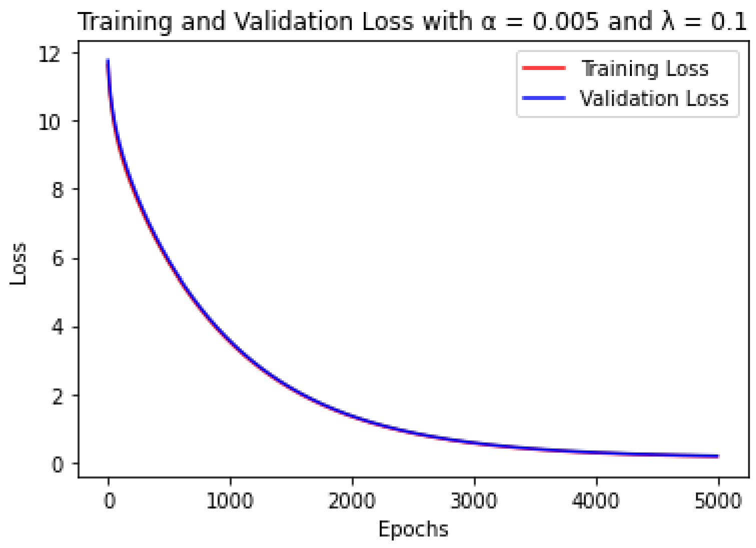


Figure 11: α = 0.005, λ = 0.1

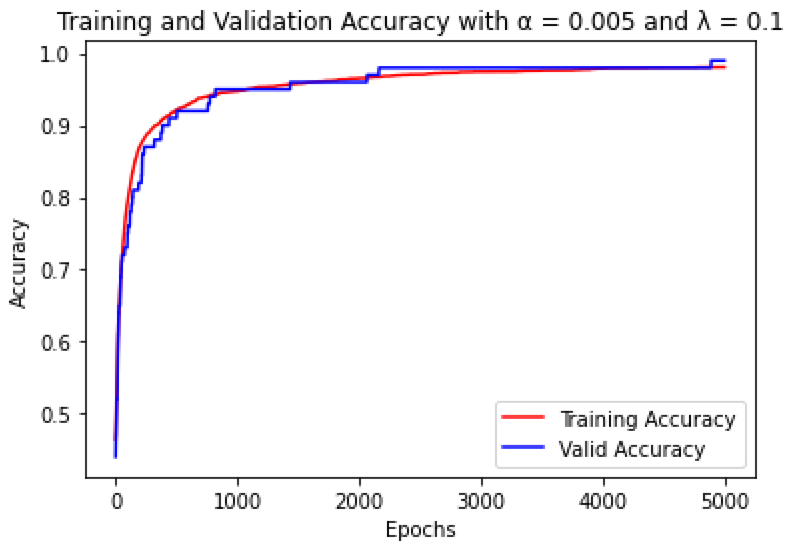


Figure 12: α = 0.005, λ = 0.1

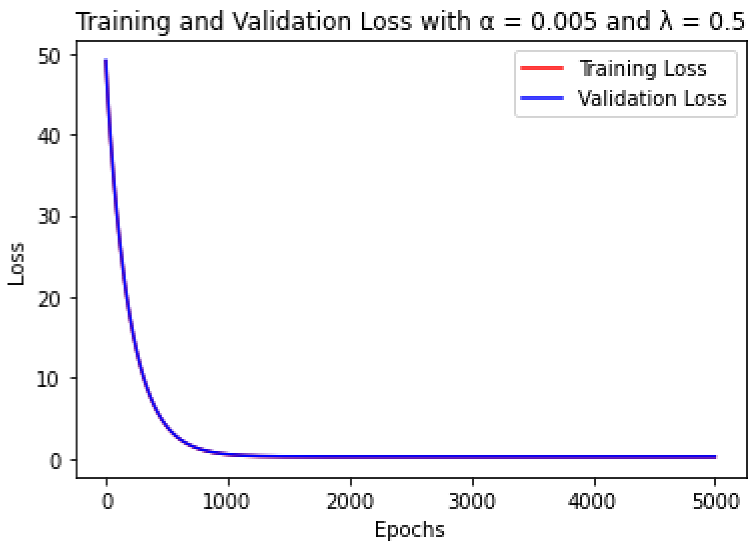


Figure 13: α = 0.005, λ = 0.5

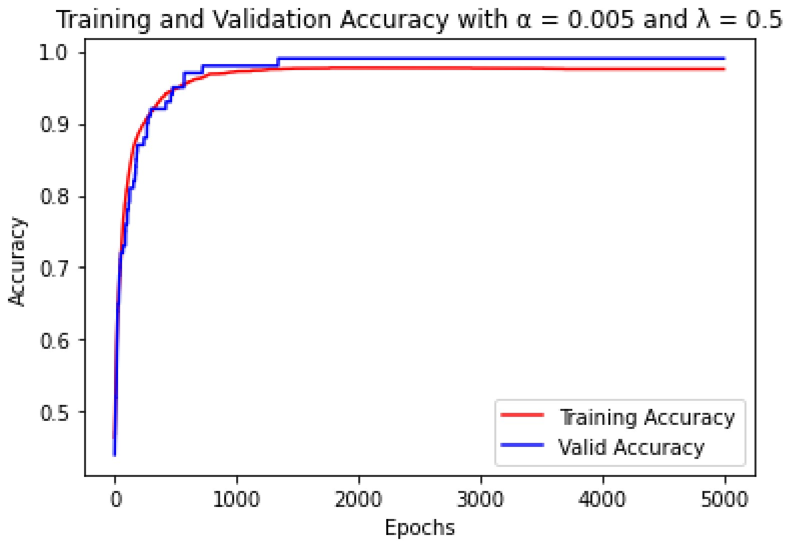


Figure 14: α = 0.005, λ = 0.5

λ = 0	α = 0.005	α = 0.001	α = 0.0001
Training Accuracy	0.9654285714285714	0.94	0.7931428571428571
Validation Accuracy	0.96	0.93	0.74

Table 2: Tuning the Learning Rate

α = 0.005	λ = 0.001	λ = 0.1	λ = 0.5
Training Accuracy	0.9657142857142857	0.9808571428571429	0.9754285714285714
Validation Accuracy	0.96	0.99	0.99

Table 3: Impact of the Regularization Parameter

Based on our observations from Figures 3-8 and Table 2, inclusive, we find that the best learning rate of α = 0.005 reports an accuracy of 96.5%. Because of our update rule $w\_update = w - \alpha * grad\_w$, we can see that a higher learning rate allows the model to converge in fewer epochs with a convex shape.

Based on our observations from Figures 9-14 and Table 3, inclusive, we find that the best regularization parameter of λ = 0.1 reports an accuracy of 98.1%. As we increase the regularization parameter, the model converges in less epochs with a convex shape and prevents overfitting. However, it seems that the middle parameter reports a higher accuracy, which can be seen from the training and validation accuracy curves.

2 Logistic Regression in TensorFlow [20 points]

Figure 15 and 16 show the training/validation loss/accuracy of the Stochastic Gradient Descent (SGD) algorithm with a minibatch size of 500 optimizing over 700 epochs using the default parameters $\alpha = 0.001$, $\lambda = 0$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 1 * 10^{-4}$. Table 4 shows the training, and validation accuracy for these settings.

Default	
Training Accuracy	0.9928571428571429
Validation Accuracy	0.98

Table 4: Stochastic Gradient Descent with TensorFlow

Figures 17-22 show the training/validation loss/accuracy of the SGD algorithm with default settings, while modifying batch size from [100, 700, 1750]. Table 5 shows the training, and validation accuracy from varying batch size. It seems that larger batch size has lower accuracy, as expected, but not by much. The larger gradient steps however, mean that the batch size of 1750 has a much smoother loss curve than a batch size of 100. Therefore, larger batch sizes can reduce noise, whereas smaller batch sizes have a regularization effect.

Default	Batch Size = 100	Batch Size = 700	Batch Size = 1750
Training Accuracy	0.9997142857142857	0.9908571428571429	0.9811428571428571
Validation Accuracy	0.98	0.97	0.99

Table 5: Impact of Batch Size

Figures 23-26, 27-30, and 31-34 show the training/validation loss/accuracy of the SGD algorithm with default settings, while modifying β_1 from [0.95, 0.99], β_2 from [0.99, 0.9999], and ϵ from $[1 * 10^{-9}, 1 * 10^{-4}]$, respectively. Table 6 shows the training, validation, and test accuracy from varying hyperparameters.

Beta 1 is the first moment used to calculate the running average of the gradient. We notice that as beta 1 increases, the accuracy decreases slightly, while producing less noise. I think 0.95 is the better value for beta 1.

Beta 2 is the second moment use the calculate the running average of the gradient squared. Similarly, higher beta 2 results in lower accuracy, and less noise. However, the effect is more pronounced, so I think 0.99 is a better choice for beta 2, as it provides near perfect training accuracy.

Finally, epsilon can help prevent a divide by zero error when the gradient converges. This epsilon value can improve stability of our outputs, but also significantly speeds up the training process. We can see from Figure 33 that the training curve due to a large epsilon value is particularly steep. Although the smaller epsilon value provides better accuracy as expected, it is almost negligibly better, while sacrificing instability, as we can see from the validation curves in Figures 31 and 32. Therefore, I think $1 * 10^{-4}$ is a good epsilon value to use.

Default	$\beta_1 = 0.95$	$\beta_1 = 0.99$	$\beta_2 = 0.99$	$\beta_2 = 0.9999$	$\epsilon = 1 * 10^{-9}$	$\epsilon = 1 * 10^{-4}$
Training Accuracy	0.9905	0.99	0.998	0.9897	0.9945	0.9914
Validation Accuracy	0.97	0.97	0.99	0.96	0.98	0.97
Testing Accuracy	0.9724	0.99	0.9724	0.9655	0.9793	0.9724

Table 6: Hyperparameter Investigation

Against batch gradient descent implemented in Section 1, Adam performs much better. In less epochs (700 vs. 3500), it is able to converge quickly to a lower loss value and to a higher accuracy. There is some instability as we can see from the plots, solely because of the mathematics involved in Adam, but there are many hyperparameters we can choose to mitigate different issues. Overall, the Adam optimizer is a better choice, and offers greater flexibility and less cost to achieve the same training results.

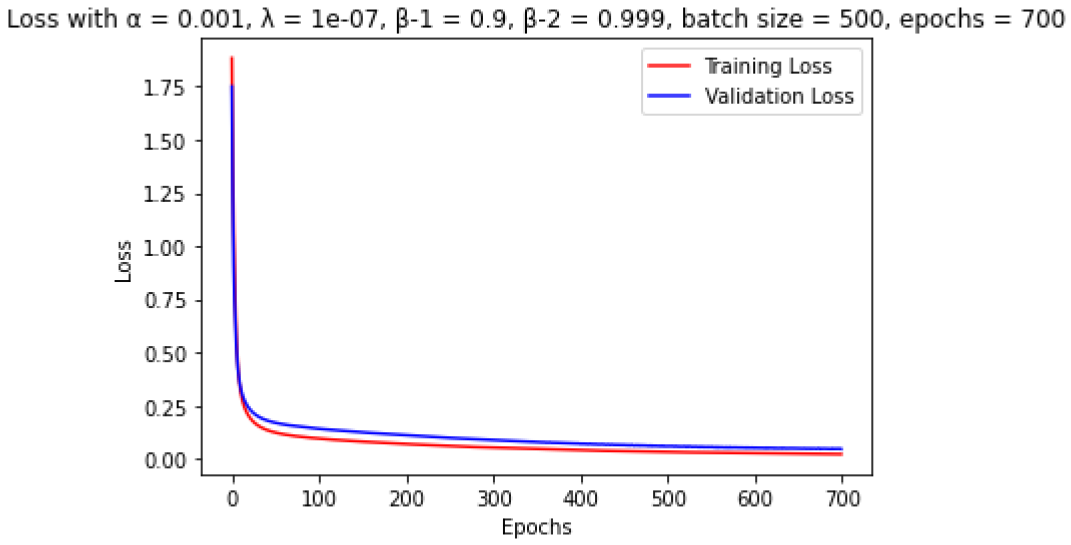


Figure 15: Default Settings Training and Validation Loss

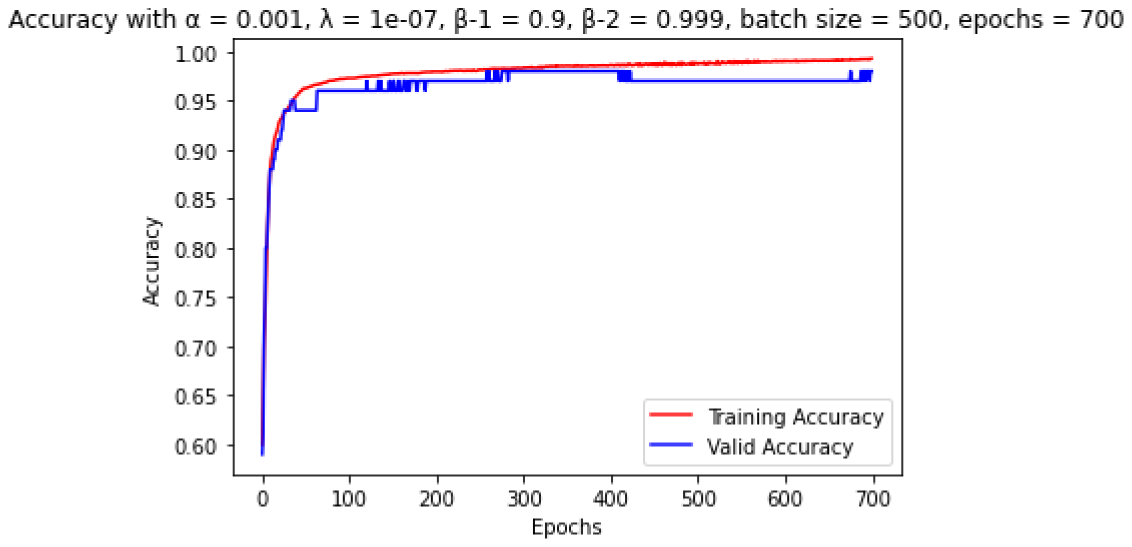


Figure 16: Default Settings Training and Validation Accuracy

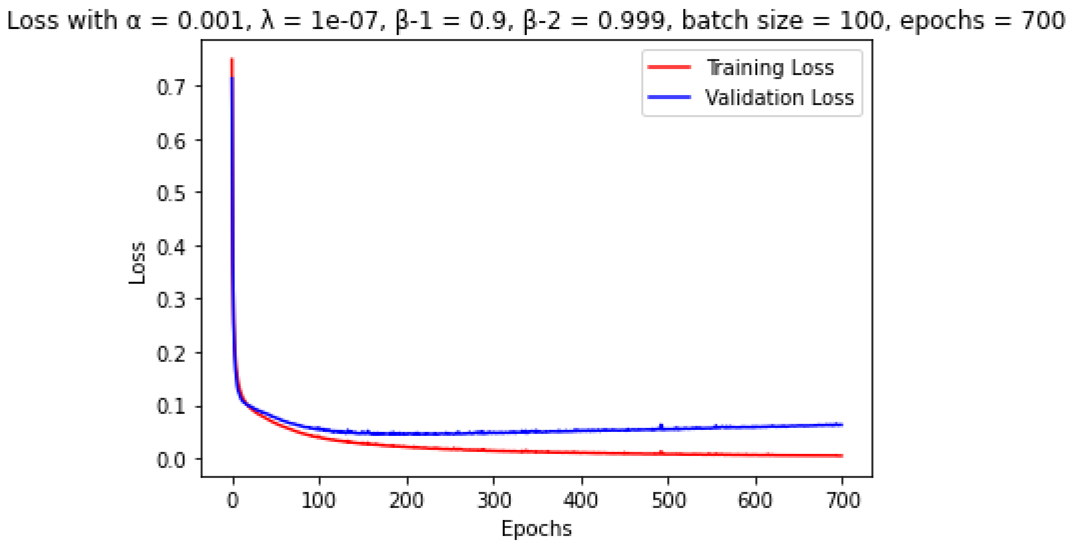


Figure 17: Batch Size = 100 Training and Validation Loss

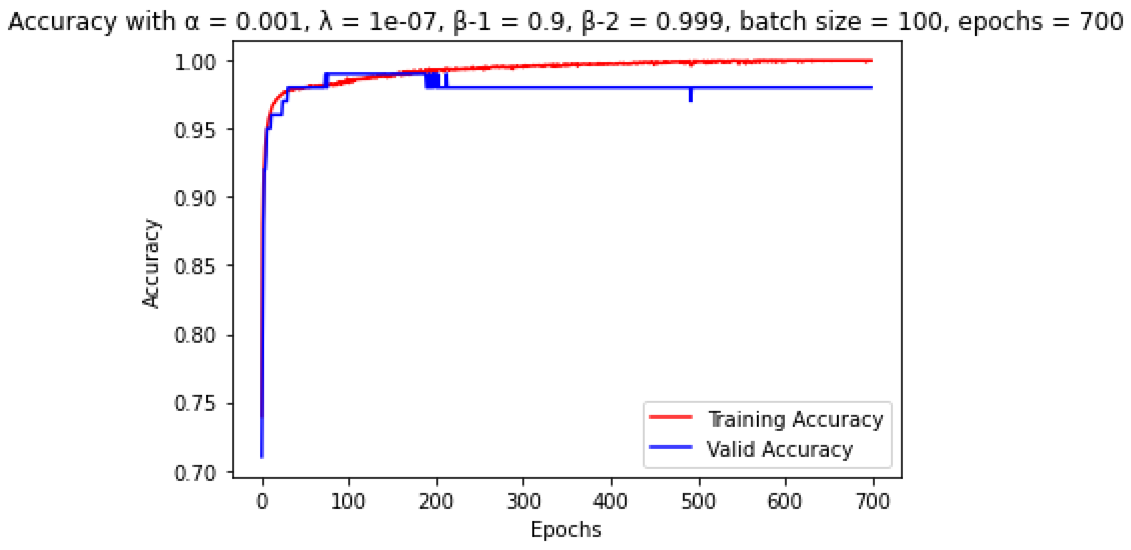


Figure 18: Batch Size = 100 Training and Validation Accuracy

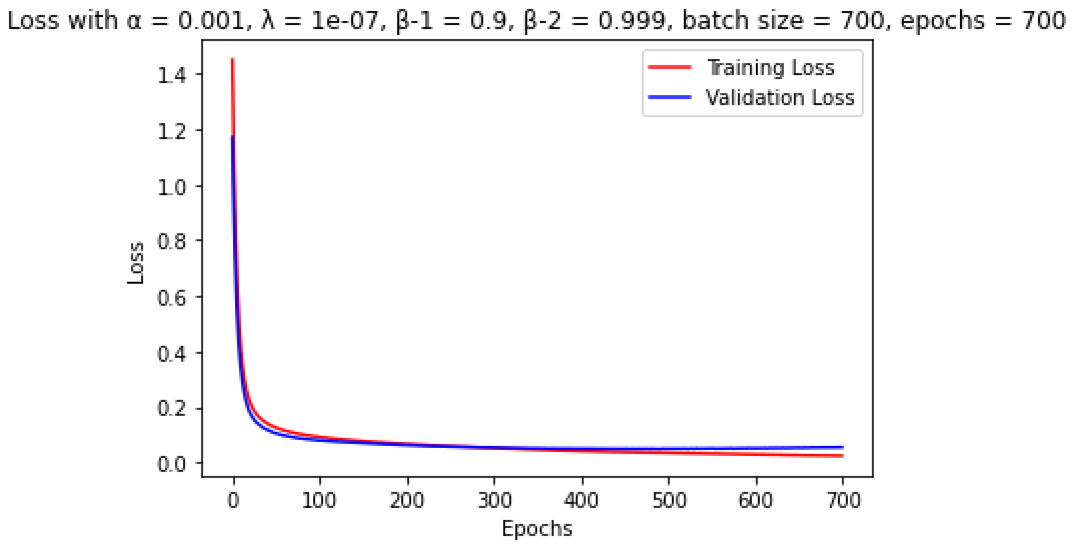


Figure 19: Batch Size = 700 Training and Validation Loss

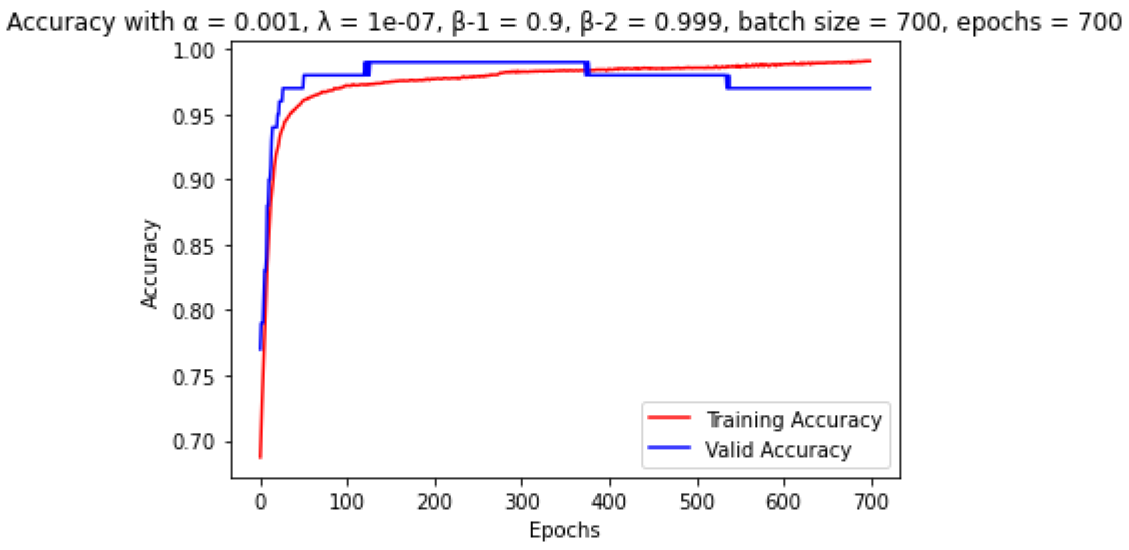


Figure 20: Batch Size = 700 Training and Validation Accuracy

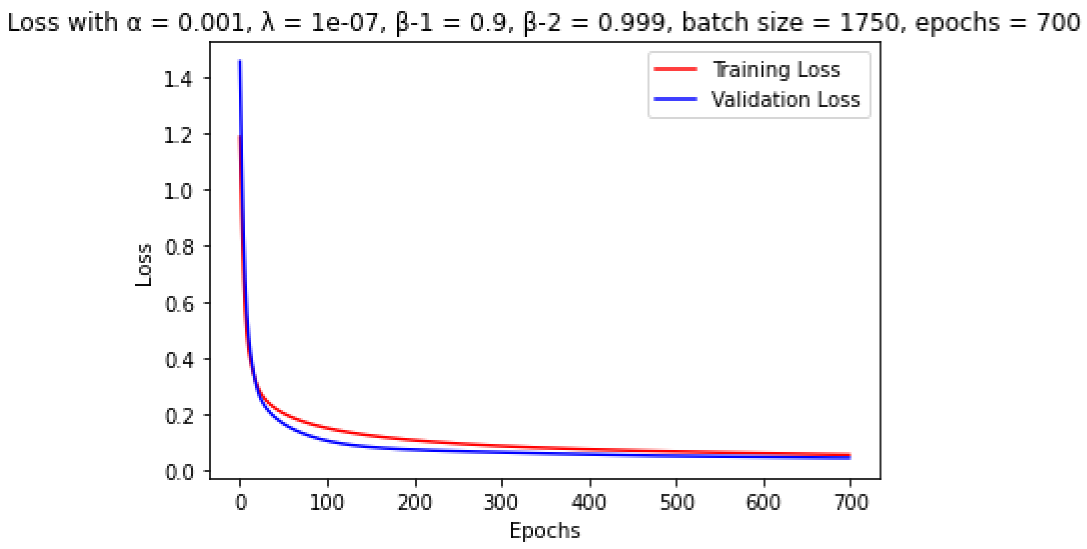


Figure 21: Batch Size = 1750 Training and Validation Loss

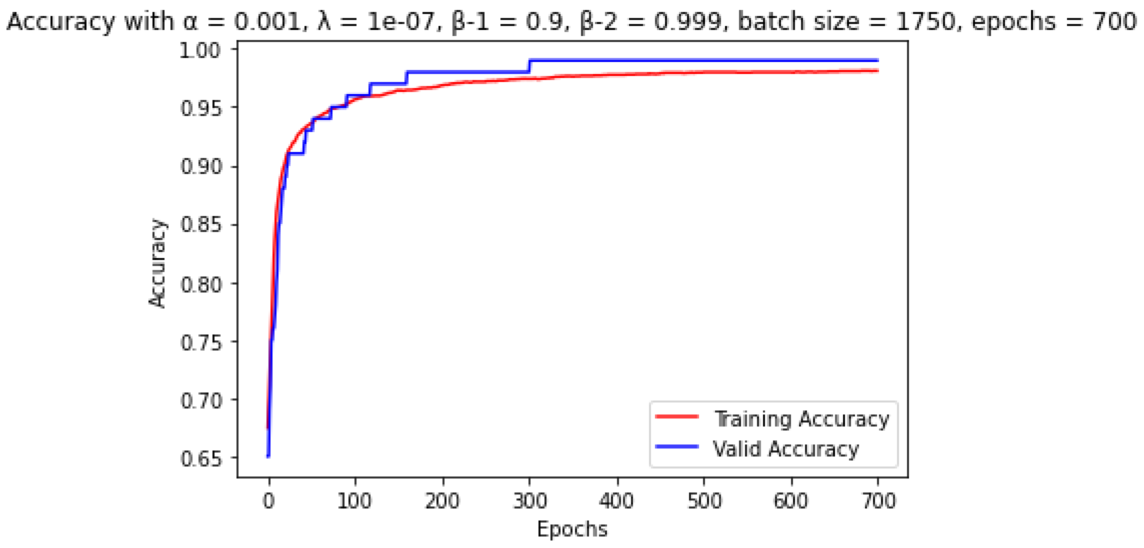


Figure 22: Batch Size = 1750 Training and Validation Accuracy

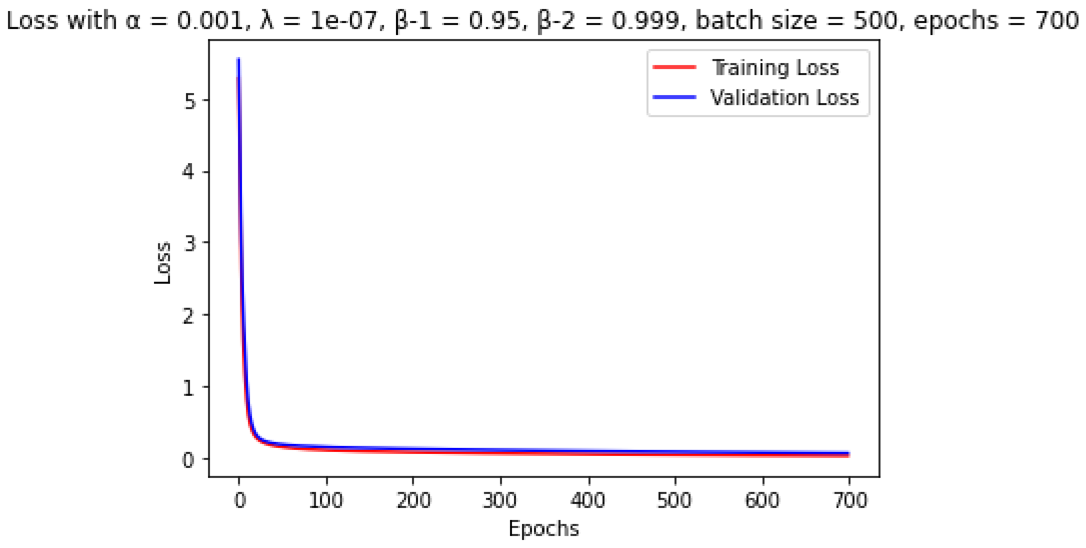


Figure 23: β_1 = 0.95 Training and Validation Loss

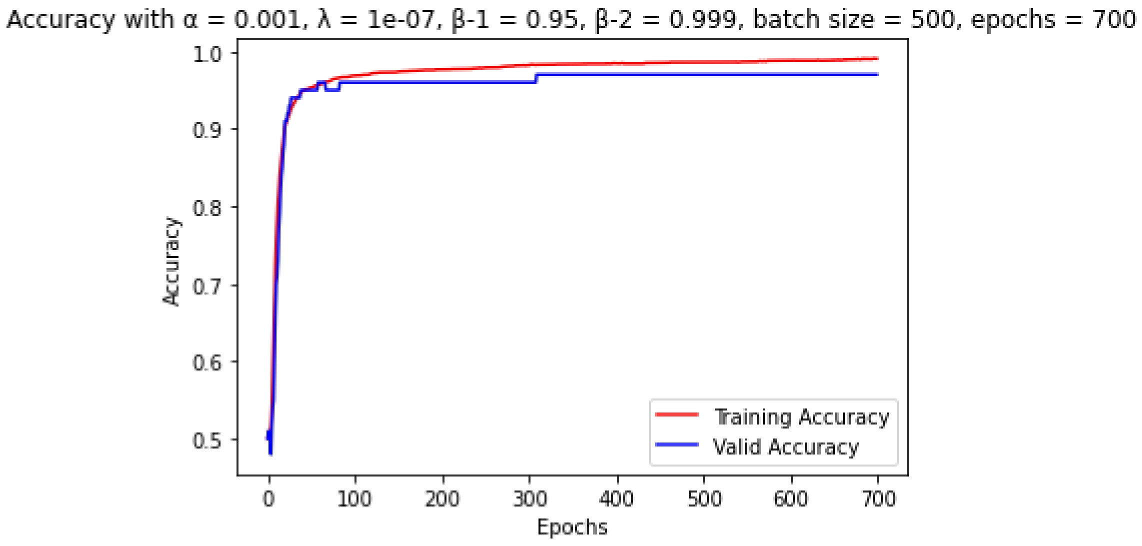


Figure 24: β_1 = 0.95 Training and Validation Accuracy

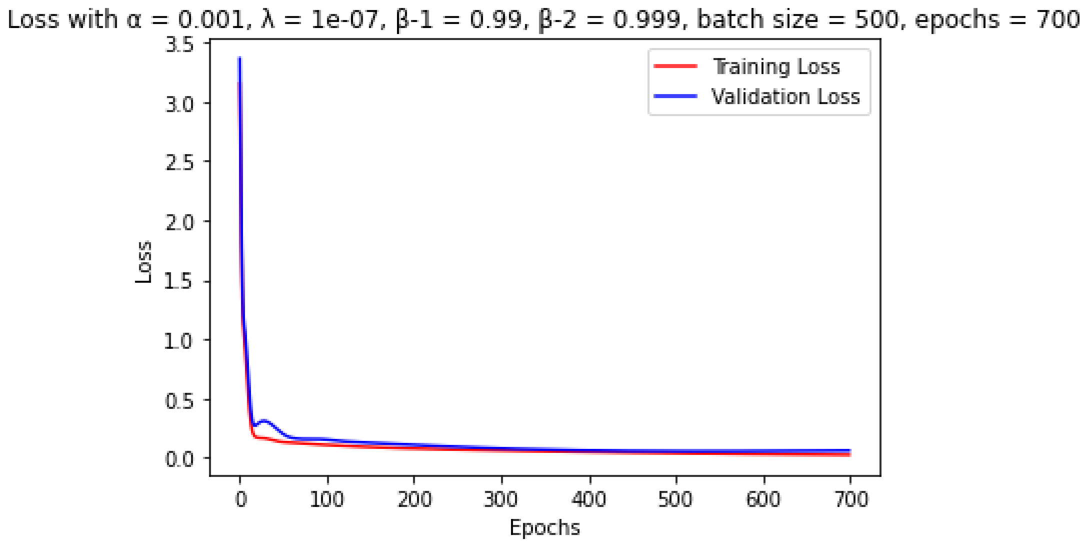


Figure 25: β_1 = 0.99 Training and Validation Loss

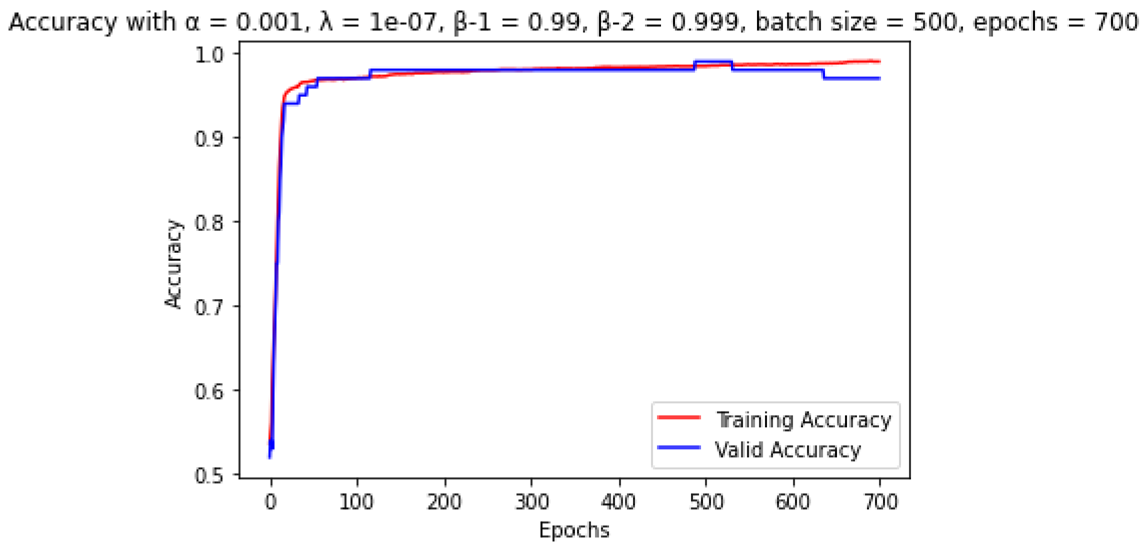


Figure 26: β_1 = 0.99 Training and Validation Accuracy

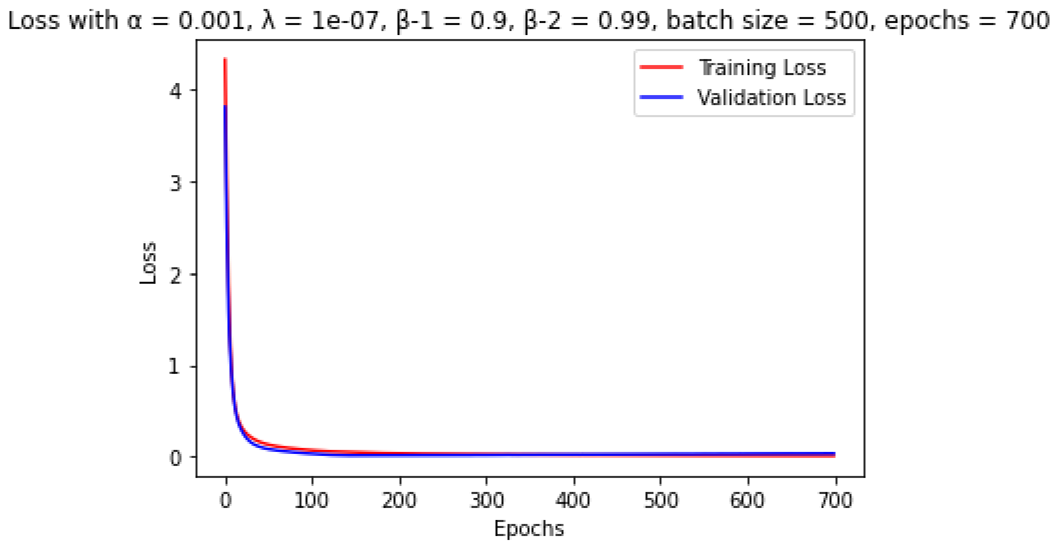


Figure 27: β_2 = 0.99 Training and Validation Loss

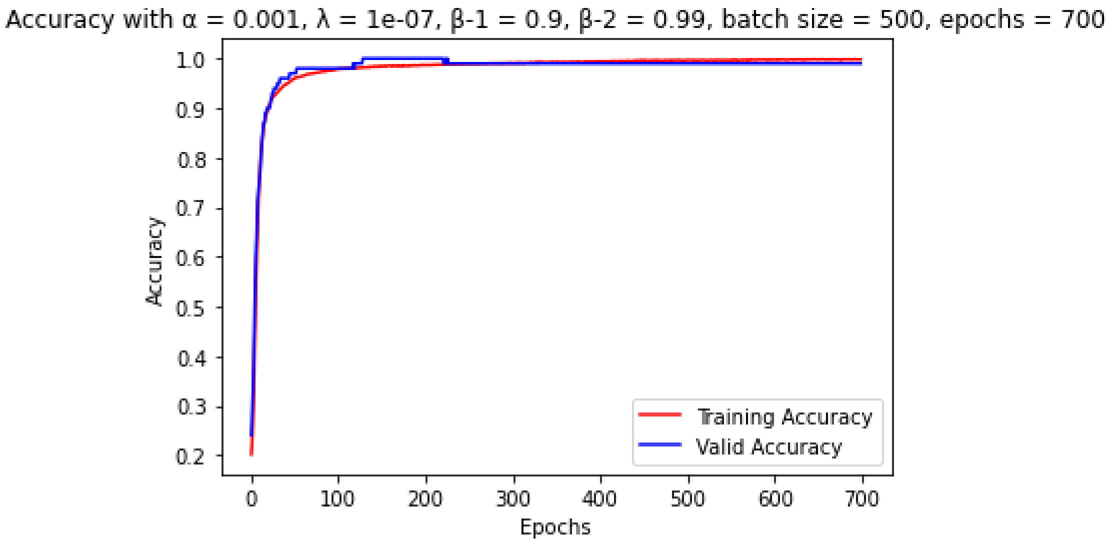


Figure 28: β_2 = 0.99 Training and Validation Accuracy

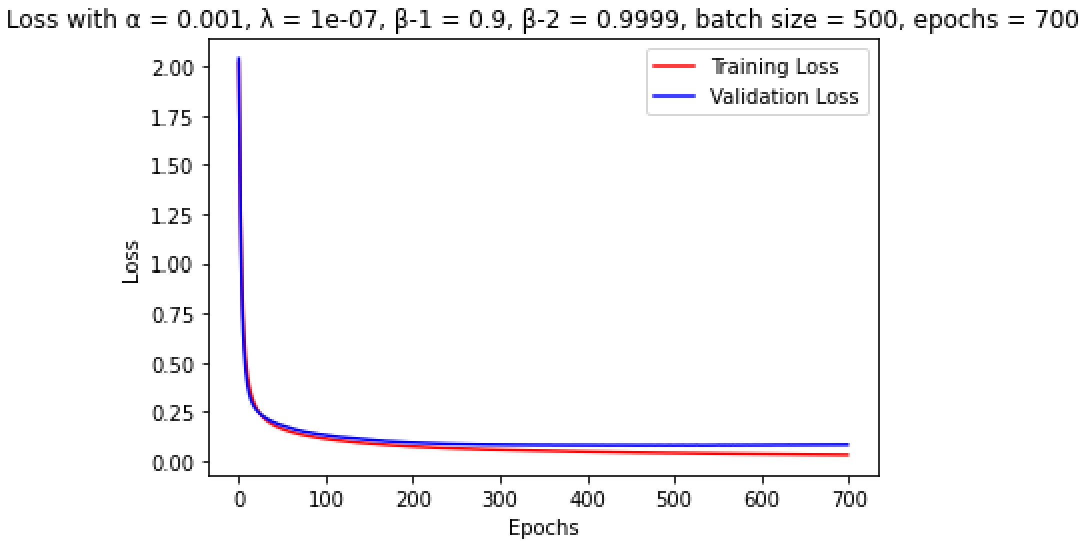


Figure 29: β_2 = 0.9999 Training and Validation Loss

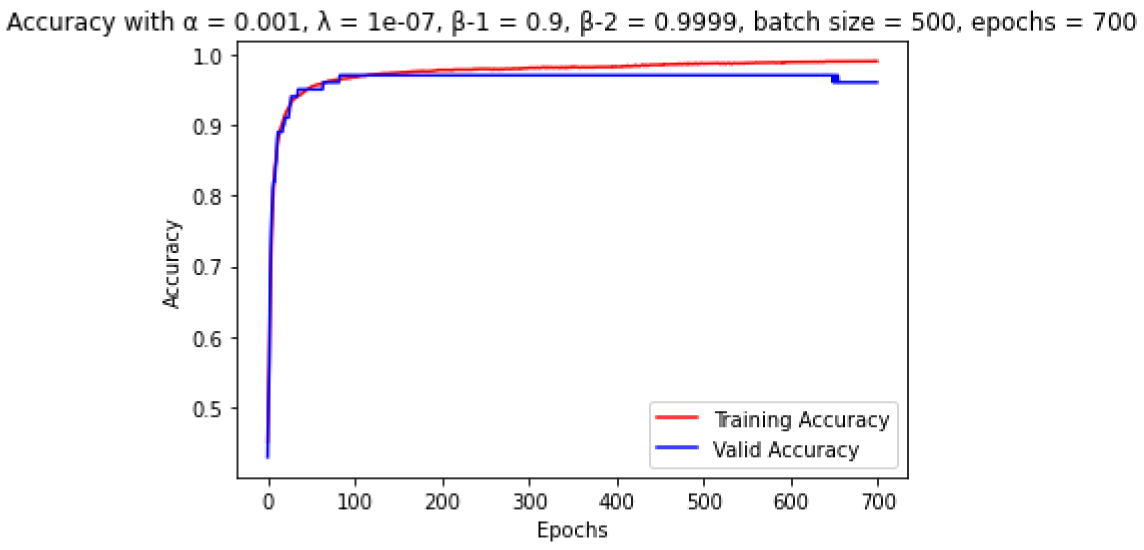


Figure 30: $\beta_2 = 0.9999$ Training and Validation Accuracy

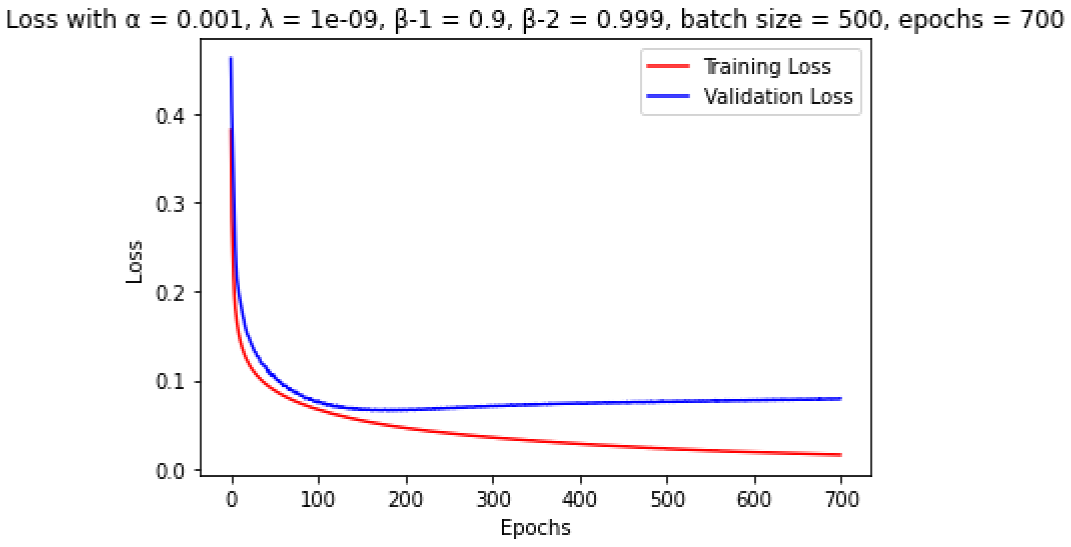


Figure 31: $\epsilon = 1 * 10^{-9}$ Training and Validation Loss

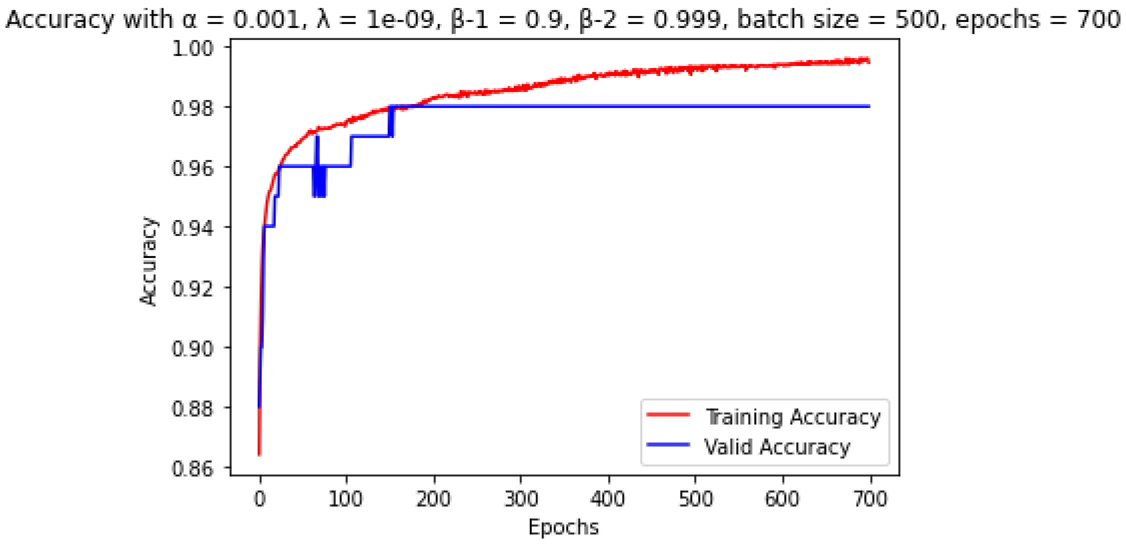


Figure 32: $\epsilon = 1 * 10^{-9}$ Training and Validation Accuracy

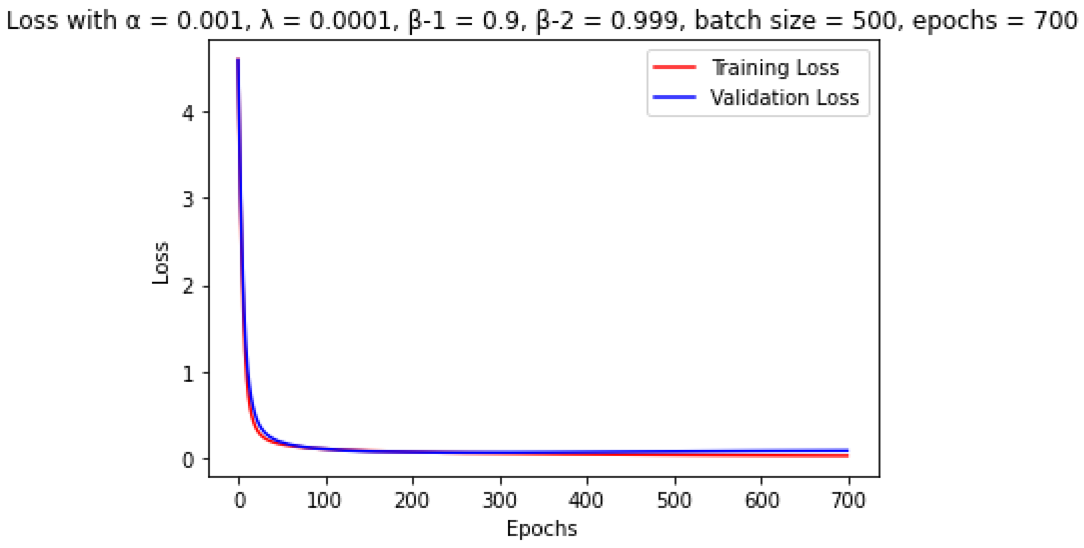


Figure 33: $\epsilon = 1 * 10^{-4}$ Training and Validation Loss

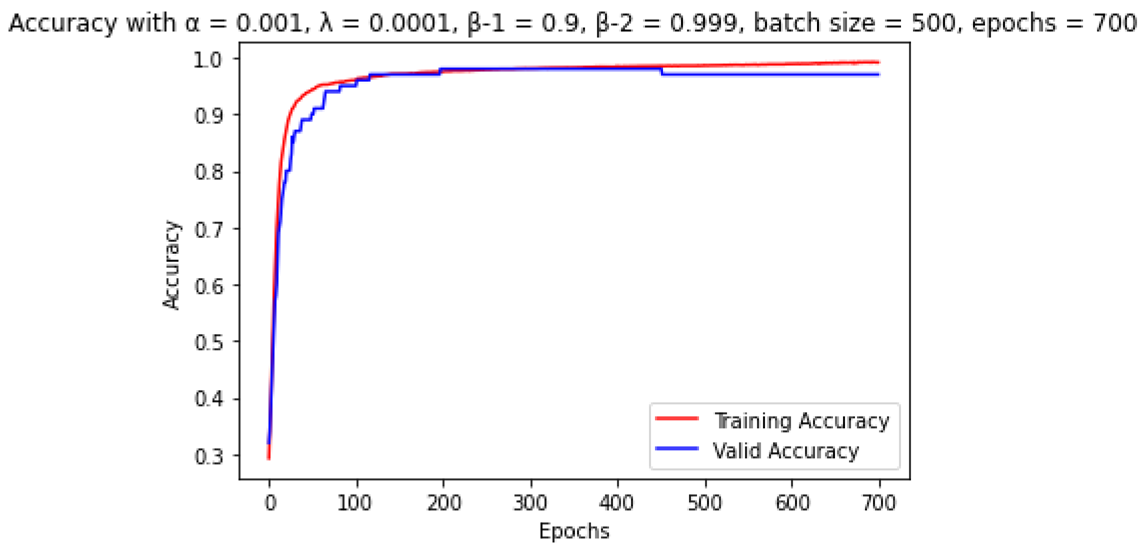


Figure 34: $\epsilon = 1 * 10^{-4}$ Training and Validation Accuracy

3 Code

```
# -*- coding: utf-8 -*-
"""A1_Logistic_Regression.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1-
        giCUkoZ6_iqtzI2jGhepMQ_CjBiyHzF
"""

#!pip install tensorflow==1.14.0
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

from google.colab import drive
drive.mount('/content/gdrive', force_remount=True)

def loadData():
    with np.load('/content/gdrive/My_Drive/ECE421/notMNIST.npz') as
        dataset:
         Data, Target = dataset['images'], dataset['labels']
         posClass = 2
         negClass = 9
         dataIndx = (Target==posClass) + (Target==negClass)
         Data = Data[dataIndx]/255.
         Target = Target[dataIndx].reshape(-1, 1)
         Target[Target==posClass] = 1
         Target[Target==negClass] = 0
         np.random.seed(421)
         randIndx = np.arange(len(Data))
         np.random.shuffle(randIndx)
         Data, Target = Data[randIndx], Target[randIndx]
         trainData, trainTarget = Data[:3500], Target[:3500]
         validData, validTarget = Data[3500:3600], Target[3500:3600]
         testData, testTarget = Data[3600:], Target[3600:]
    return trainData, validData, testData, trainTarget, validTarget,
        testTarget

def loss(w, b, x, y, reg):
    #flattens x (data) from an N * d * d matrix to an N * D matrix, D =
        d * d
    #x = x.reshape(np.shape(x)[0], np.shape(x)[1]*np.shape(x)[2])
    #N = np.shape(x)[0] #N is the number of samples
    #w = weight vector (initialize as 0)
    #b = bias vector (initialize as 0)
    #reg = regularization parameter (not sure)
    #y = class label (0, 1) for n-th training image (target)

    #calculates the logit z = w^Tx+b using the inner product <w, x> = w
        ^Tx
    z = np.dot(x, w) + b

    #defines y_hat = sigma(z) = 1/(1+e^-z) in (0,1)
    y_hat = 1/(1 + np.exp(-z))

    #defines the cross-entropy loss L_CE
    L_CE = -np.mean((y)*np.log(y_hat) + (1-y)*np.log(1-y_hat))

    #defines the regularization term L_W
    L_W = reg/2 * (np.linalg.norm(w)**2)

    #aggregates the two terms into the loss function L
    L = L_CE + L_W

    return L

def grad_loss(w, b, x, y, reg):
    #x = x.reshape(np.shape(x)[0], np.shape(x)[1]*np.shape(x)[2])
    N = np.shape(y)[0] #N is the number of samples
    z = np.dot(x, w) + b
    y_hat = 1/(1 + np.exp(-z))
    grad_w = np.dot(np.transpose(x), (y_hat - y)) / N + reg*w
    grad_b = np.mean((y_hat-y))
    return grad_w, grad_b

def grad_descent(w, b, trainData, trainTarget, train_loss, valid_loss,
                 test_loss, train_acc, valid_acc, test_acc, validData,
                 validTarget, testData, testTarget, alpha, epochs, reg,
                     error_tol, acc):

    for i in range(epochs):
        grad_w, grad_b = grad_loss(w, b, trainData, trainTarget, reg)
        w_update = w - alpha * grad_w
        b_update = b - alpha * grad_b

        if(acc):
            train_loss.append(loss(w_update, b_update, trainData,
               trainTarget, reg))
```

```
                valid_loss.append(loss(w_update, b_update, validData,
                    validTarget, reg))
                test_loss.append(loss(w_update, b_update, testData,
                    testTarget, reg))
                train_acc.append(accuracy(w_update, b_update, trainData,
                    trainTarget))
                valid_acc.append(accuracy(w_update, b_update, validData,
                    validTarget))
                test_acc.append(accuracy(w_update, b_update, testData,
                    testTarget))

                if(np.linalg.norm(w - w_update) < error_tol):
                    return (w_update, b_update, train_loss, valid_loss,
                        test_loss,
                            train_acc, valid_acc, test_acc)
                else:
                    w = w_update
                    b = b_update

            elif(np.linalg.norm(w - w_update) < error_tol):
                return w_update, b_update
            else:
                w = w_update
                b = b_update

    if(acc):
        return w, b, train_loss, valid_loss, test_loss, train_acc,
            valid_acc, test_acc
    else:
        return w, b

def accuracy(w, b, x, y):
    #x = x.reshape(np.shape(x)[0], np.shape(x)[1]*np.shape(x)[2])
    N = np.shape(y)[0] #N is the number of samples
    z = np.dot(x, w) + b
    y_hat = 1/(1 + np.exp(-z))
    acc = np.sum((y_hat >= 0.5) == y)/N
    return acc

def driver():

    trainData, validData, testData, trainTarget, validTarget,
        testTarget, w, b, reg, train_loss, valid_loss, test_loss,
        train_loss, train_acc, valid_acc, test_acc = initializer(reg =
        0)
```

```
w, b, train_loss, valid_loss, test_loss, train_acc, valid_acc,
    test_acc = grad_descent(
     w, b, trainData, trainTarget, train_loss, valid_loss, test_loss
         , train_acc, valid_acc, test_acc, validData, validTarget,
         testData, testTarget, alpha = 0.005, epochs = 5000, reg = 0,
          error_tol = 1e-7, acc = True)

f = plt.figure(1)
title = 'Training and Validation Loss with \u03B1 = 0.005 and \
    u03BB = 0'
plt.title(title)
plt.ylabel('Loss')
plt.xlabel('Epochs')
plt.plot(range(5001), train_loss, 'r', range(5001), valid_loss, 'b'
    )
plt.legend(["Training Loss","Validation Loss"], loc = 'best')
plt.show()

f = plt.figure(2)
title = 'Training and Validation Accuracy with \u03B1 = 0.005 and \
    u03BB = 0'
plt.title(title)
plt.ylabel('Accuracy')
plt.xlabel('Epochs')
plt.plot(range(5001), train_acc, 'r', range(5001), valid_acc, 'b')
plt.legend(["Training Accuracy","Valid Accuracy"], loc = 'best')
plt.show()

print('Training Accuracy: ', train_acc[5000])
print('Validation Accuracy: ', valid_acc[5000])
print('Test Accuracy: ', test_acc[5000])

trainData, validData, testData, trainTarget, validTarget,
    testTarget, w, b, reg, train_loss, valid_loss, test_loss,
    train_loss, train_acc, valid_acc, test_acc = initializer(reg =
    0)

w, b, train_loss, valid_loss, test_loss, train_acc, valid_acc,
    test_acc = grad_descent(
     w, b, trainData, trainTarget, train_loss, valid_loss, test_loss
         , train_acc, valid_acc, test_acc, validData, validTarget,
         testData, testTarget, alpha = 0.001, epochs = 5000, reg = 0,
          error_tol = 1e-7, acc = True)

f = plt.figure(3)
```

```
title = 'Training␣and␣Validation␣Loss␣with␣\u03B1␣=␣0.001␣and␣\
    u03BB␣=␣0'
plt.title(title)
plt.ylabel('Loss')
plt.xlabel('Epochs')
plt.plot(range(5001), train_loss, 'r', range(5001), valid_loss, 'b'
    )
plt.legend(["Training␣Loss" ,"Validation␣Loss"], loc = 'best')
plt.show()

f = plt.figure(4)
title = 'Training␣and␣Validation␣Accuracy␣with␣\u03B1␣=␣0.001␣and␣\
    u03BB␣=␣0'
plt.title(title)
plt.ylabel('Accuracy')
plt.xlabel('Epochs')
plt.plot(range(5001), train_acc, 'r', range(5001), valid_acc, 'b')
plt.legend(["Training␣Accuracy" ,"Valid␣Accuracy"], loc = 'best')
plt.show()

print('Training␣Accuracy:␣', train_acc[5000])
print('Validation␣Accuracy:␣', valid_acc[5000])
print('Test␣Accuracy:␣', test_acc[5000])

trainData, validData, testData, trainTarget, validTarget,
    testTarget, w, b, reg, train_loss, valid_loss, test_loss,
    train_loss, train_acc, valid_acc, test_acc = initializer(reg =
    0)

w, b, train_loss, valid_loss, test_loss, train_acc, valid_acc,
    test_acc = grad_descent(
     w, b, trainData, trainTarget, train_loss, valid_loss, test_loss
         , train_acc, valid_acc, test_acc, validData, validTarget,
         testData, testTarget, alpha = 0.0001, epochs = 5000, reg =
         0, error_tol = 1e-7, acc = True)

f = plt.figure(5)
title = 'Training␣and␣Validation␣Loss␣with␣\u03B1␣=␣0.0001␣and␣\
    u03BB␣=␣0'
plt.title(title)
plt.ylabel('Loss')
plt.xlabel('Epochs')
plt.plot(range(5001), train_loss, 'r', range(5001), valid_loss, 'b'
    )
plt.legend(["Training␣Loss" ,"Validation␣Loss"], loc = 'best')
plt.show()
```

```
f = plt.figure(6)
title = 'Training and Validation Accuracy with \u03B1 = 0.0001 and 
    \u03BB = 0'
plt.title(title)
plt.ylabel('Accuracy')
plt.xlabel('Epochs')
plt.plot(range(5001), train_acc, 'r', range(5001), valid_acc, 'b')
plt.legend(["Training Accuracy","Valid Accuracy"], loc = 'best')
plt.show()

print('Training Accuracy: ', train_acc[5000])
print('Validation Accuracy: ', valid_acc[5000])
print('Test Accuracy: ', test_acc[5000])

trainData, validData, testData, trainTarget, validTarget,
    testTarget, w, b, reg, train_loss, valid_loss, test_loss,
    train_loss, train_acc, valid_acc, test_acc = initializer(reg =
    0.001)

w, b, train_loss, valid_loss, test_loss, train_acc, valid_acc,
    test_acc = grad_descent(
     w, b, trainData, trainTarget, train_loss, valid_loss, test_loss
         , train_acc, valid_acc, test_acc, validData, validTarget,
         testData, testTarget, alpha = 0.005, epochs = 5000, reg =
         0.001, error_tol = 1e-7, acc = True)

f = plt.figure(7)
title = 'Training and Validation Loss with \u03B1 = 0.005 and \
    u03BB = 0.001'
plt.title(title)
plt.ylabel('Loss')
plt.xlabel('Epochs')
plt.plot(range(5001), train_loss, 'r', range(5001), valid_loss, 'b'
    )
plt.legend(["Training Loss","Validation Loss"], loc = 'best')
plt.show()

f = plt.figure(8)
title = 'Training and Validation Accuracy with \u03B1 = 0.005 and \
    u03BB = 0.001'
plt.title(title)
plt.ylabel('Accuracy')
plt.xlabel('Epochs')
plt.plot(range(5001), train_acc, 'r', range(5001), valid_acc, 'b')
plt.legend(["Training Accuracy","Valid Accuracy"], loc = 'best')
plt.show()

print('Training Accuracy: ', train_acc[5000])
print('Validation Accuracy: ', valid_acc[5000])
print('Test Accuracy: ', test_acc[5000])

trainData, validData, testData, trainTarget, validTarget,
    testTarget, w, b, reg, train_loss, valid_loss, test_loss,
    train_loss, train_acc, valid_acc, test_acc = initializer(reg =
    0.1)

w, b, train_loss, valid_loss, test_loss, train_acc, valid_acc,
    test_acc = grad_descent(
     w, b, trainData, trainTarget, train_loss, valid_loss, test_loss
         , train_acc, valid_acc, test_acc, validData, validTarget,
         testData, testTarget, alpha = 0.005, epochs = 5000, reg =
         0.1, error_tol = 1e-7, acc = True)

f = plt.figure(9)
title = 'Training and Validation Loss with \u03B1 = 0.005 and \
    u03BB = 0.1'
plt.title(title)
plt.ylabel('Loss')
plt.xlabel('Epochs')
plt.plot(range(5001), train_loss, 'r', range(5001), valid_loss, 'b'
    )
plt.legend(["Training Loss" ,"Validation Loss"], loc = 'best')
plt.show()

f = plt.figure(10)
title = 'Training and Validation Accuracy with \u03B1 = 0.005 and \
    u03BB = 0.1'
plt.title(title)
plt.ylabel('Accuracy')
plt.xlabel('Epochs')
plt.plot(range(5001), train_acc, 'r', range(5001), valid_acc, 'b')
plt.legend(["Training Accuracy" ,"Valid Accuracy"], loc = 'best')
plt.show()

print('Training Accuracy: ', train_acc[5000])
print('Validation Accuracy: ', valid_acc[5000])
print('Test Accuracy: ', test_acc[5000])

trainData, validData, testData, trainTarget, validTarget,
    testTarget, w, b, reg, train_loss, valid_loss, test_loss,
    train_loss, train_acc, valid_acc, test_acc = initializer(reg =
        0.5)

    w, b, train_loss, valid_loss, test_loss, train_acc, valid_acc,
        test_acc = grad_descent(
         w, b, trainData, trainTarget, train_loss, valid_loss, test_loss
            , train_acc, valid_acc, test_acc, validData, validTarget,
            testData, testTarget, alpha = 0.005, epochs = 5000, reg =
            0.5, error_tol = 1e-7, acc = True)

    f = plt.figure(11)
    title = 'Training and Validation Loss with \u03B1 = 0.005 and \
        u03BB = 0.5'
    plt.title(title)
    plt.ylabel('Loss')
    plt.xlabel('Epochs')
    plt.plot(range(5001), train_loss, 'r', range(5001), valid_loss, 'b'
        )
    plt.legend(["Training Loss","Validation Loss"], loc = 'best')
    plt.show()

    f = plt.figure(12)
    title = 'Training and Validation Accuracy with \u03B1 = 0.005 and \
        u03BB = 0.5'
    plt.title(title)
    plt.ylabel('Accuracy')
    plt.xlabel('Epochs')
    plt.plot(range(5001), train_acc, 'r', range(5001), valid_acc, 'b')
    plt.legend(["Training Accuracy","Valid Accuracy"], loc = 'best')
    plt.show()

    print('Training Accuracy: ', train_acc[5000])
    print('Validation Accuracy: ', valid_acc[5000])
    print('Test Accuracy: ', test_acc[5000])

def initializer(reg):

    trainData, validData, testData, trainTarget, validTarget,
        testTarget = loadData()

    trainData = trainData.reshape(np.shape(trainData)[0], np.shape(
        trainData)[1]*np.shape(trainData)[2])
    validData = validData.reshape(np.shape(validData)[0], np.shape(
        validData)[1]*np.shape(validData)[2])
    testData = testData.reshape(np.shape(testData)[0], np.shape(
        testData)[1]*np.shape(testData)[2])
    w = np.random.normal(0, 0.5, (np.shape(trainData)[1], 1))

    b = 0

    train_loss = [loss(w, b, trainData, trainTarget, reg)]
    valid_loss = [loss(w, b, validData, validTarget, reg)]
    test_loss = [loss(w, b, testData, testTarget, reg)]
    train_acc = [accuracy(w, b, trainData, trainTarget)]
    valid_acc = [accuracy(w, b, validData, validTarget)]
    test_acc = [accuracy(w, b, testData, testTarget)]

    return trainData, validData, testData, trainTarget, validTarget,
        testTarget, w, b, reg, train_loss, valid_loss, test_loss,
        train_loss, train_acc, valid_acc, test_acc

def buildGraph(beta1, beta2, epsilon, alpha):
    w = tf.Variable(tf.truncated_normal([784, 1], mean = 0, stddev =
        0.5, dtype = tf.float32))
    b = tf.Variable(tf.zeros(1))

    x = tf.placeholder(tf.float32, [None, 784])
    y = tf.placeholder(tf.float32, [None, 1])
    reg = tf.placeholder(tf.float32)

    logits = tf.matmul(x, w) + b

    loss = tf.losses.sigmoid_cross_entropy(multi_class_labels = y,
        logits = logits) + reg * tf.nn.l2_loss(w) #latter term returns
        half the l2 norm of w

    optimizer = tf.train.AdamOptimizer(alpha, beta1, beta2, epsilon).
        minimize(loss)

    return w, b, x, y, reg, loss, optimizer

def SGD(batch_size, beta1, beta2, epsilon, alpha, epochs):
    trainData, validData, testData, trainTarget, validTarget,
        testTarget = loadData()
    tf.set_random_seed(421)
    trainData = trainData.reshape(np.shape(trainData)[0],np.shape(
        trainData)[1]*np.shape(trainData)[2])
    validData = validData.reshape(np.shape(validData)[0],np.shape(
        validData)[1]*np.shape(validData)[2])
    testData = testData.reshape(np.shape(testData)[0],np.shape(testData
        )[1]*np.shape(testData)[2])
    w, b, x, y, reg, loss, optimizer = buildGraph(beta1, beta2, epsilon
        , alpha)

    num_batches = np.shape(trainData)[0] // batch_size # rounds down
        number of batches so we get batches of same size (this is ok
        because we reshuffle the data)

    train_loss, train_acc, valid_loss, valid_acc, test_loss, test_acc =
         ([] for idx in range(6))

    init = tf.global_variables_initializer()

    with tf.Session() as sess:
        sess.run(init)
        for i in range(epochs):
            index = np.arange(trainData.shape[0])
            np.random.shuffle(index)
            trainData = trainData[index, :]
            trainTarget = trainTarget[index]
            for j in range(num_batches):
                x_batch = trainData[j * batch_size : (j + 1) *
                    batch_size, :]
                y_batch = trainTarget[j * batch_size : (j + 1) *
                    batch_size, :]
                optimizer_something, loss_now, w_now, b_now = sess.run
                    ([optimizer, loss, w, b], feed_dict = {x: x_batch, y
                    : y_batch, reg: 0})
            train_loss.append(sess.run(loss, feed_dict = {x: trainData,
                 y: trainTarget, reg: 0}))
            valid_loss.append(sess.run(loss, feed_dict = {x: validData,
                 y: validTarget, reg: 0}))
            test_loss.append(sess.run(loss, feed_dict = {x: testData, y
                : testTarget, reg: 0}))
            train_acc.append(accuracy(w_now, b_now, trainData,
                trainTarget))
            valid_acc.append(accuracy(w_now, b_now, validData,
                validTarget))
            test_acc.append(accuracy(w_now, b_now, testData, testTarget
                ))

    return train_loss, valid_loss, test_loss, train_acc, valid_acc,
        test_acc

def driver2(batch_size, beta1, beta2, epsilon, alpha, epochs):
    train_loss, valid_loss, test_loss, train_acc, valid_acc, test_acc =
         SGD(batch_size, beta1, beta2, epsilon, alpha, epochs)
    plt.title('Loss␣with␣\u03B1␣=␣{},␣\u03BB␣=␣{},␣\u03B2-1␣=␣{},␣\
        u03B2-2␣=␣{},␣batch␣size␣=␣{},␣epochs␣=␣{}'.format(alpha,
        epsilon, beta1, beta2, batch_size, epochs))
    plt.ylabel('Loss')
    plt.xlabel('Epochs')
    plt.plot(range(epochs), train_loss, 'r', range(epochs), valid_loss,
         'b')
    plt.legend(["Training␣Loss" ,"Validation␣Loss"], loc = 'best')
    plt.show()
    print('Training␣Accuracy:␣', train_acc[699])
    print('Validation␣Accuracy:␣', valid_acc[699])
    print('Test␣Accuracy:␣', test_acc[699])

    plt.title('Accuracy␣with␣\u03B1␣=␣{},␣\u03BB␣=␣{},␣\u03B2-1␣=␣{},␣\
        u03B2-2␣=␣{},␣batch␣size␣=␣{},␣epochs␣=␣{}'.format(alpha,
        epsilon, beta1, beta2, batch_size, epochs))
    plt.ylabel('Accuracy')
    plt.xlabel('Epochs')
    plt.plot(range(epochs), train_acc, 'r', range(epochs), valid_acc, '
        b')
    plt.legend(["Training␣Accuracy" ,"Valid␣Accuracy"], loc = 'best')
    plt.show()
    print('Training␣Accuracy:␣', train_acc[699])
    print('Validation␣Accuracy:␣', valid_acc[699])
    print('Test␣Accuracy:␣', test_acc[699])

def big_driver():
    #part 1
    #driver()
    #q1 SGD first try
    driver2(batch_size = 500, beta1 = 0.9, beta2 = 0.999, epsilon = 1e
        -07, alpha = 0.001, epochs = 700)
    #q3 batch sizes
    driver2(batch_size = 100, beta1 = 0.9, beta2 = 0.999, epsilon = 1e
        -07, alpha = 0.001, epochs = 700)
    driver2(batch_size = 700, beta1 = 0.9, beta2 = 0.999, epsilon = 1e
        -07, alpha = 0.001, epochs = 700)
    driver2(batch_size = 1750, beta1 = 0.9, beta2 = 0.999, epsilon = 1e
        -07, alpha = 0.001, epochs = 700)
    #q4 hyperparameters
    driver2(batch_size = 500, beta1 = 0.95, beta2 = 0.999, epsilon = 1e
        -07, alpha = 0.001, epochs = 700)
    driver2(batch_size = 500, beta1 = 0.99, beta2 = 0.999, epsilon = 1e
        -07, alpha = 0.001, epochs = 700)
    driver2(batch_size = 500, beta1 = 0.9, beta2 = 0.99, epsilon = 1e
        -07, alpha = 0.001, epochs = 700)
    driver2(batch_size = 500, beta1 = 0.9, beta2 = 0.9999, epsilon = 1e
        -07, alpha = 0.001, epochs = 700)
    driver2(batch_size = 500, beta1 = 0.9, beta2 = 0.999, epsilon = 1e
        -09, alpha = 0.001, epochs = 700)
    driver2(batch_size = 500, beta1 = 0.9, beta2 = 0.999, epsilon = 1e
        -04, alpha = 0.001, epochs = 700)

big_driver()
```