Course: DD2424- Assignment 2

Sevket Melih Zenciroglu – smzen@kth.se

Question-1: The code for your assignment assembled into one file.

Answer-1: Find it in the end of the document.

Question-2.i: State how you checked your analytic gradient computations and whether you think that your gradient computations were bug free. Give evidence for these conclusions.

Answer-2.i: We calculated the gradients numerically as it was suggested in the assignment and compared these values with the ones I derived analytically. The MEAN of the differences for W and b values were smaller than 1e-11 which means that the error is very small. According to the reference given from Standford, having error values smaller than 1e-7 should make us happy.

Parameter	Value	Description
n_epochs	200	number of times we iterate on the entire data
batch_size	2	the size of the mini batch. in other words, number of images in 1 mini-batch. (in this specific example, we only used 2 images – it was suggested to use 1 image in the assignment)
eta	0.001	learning rate (step-size)
lambda_cost	0	regularization coefficient (punishment)
d	3072	dimension of X_train (input)... 3072 = 32 x 32 x 3 (20 suggested for this exercise but since the calculations are fast enough, we used all dimensions)
m	50	number of nodes in the hidden layer
h	1e-5	Precision value

Table-1: Parameters used

Another reason for not using dimension as 20 was the below results. Somehow, the small dimension couldn't help us to get the results that we are after:
d = 20, N=100 >> Cost from 2.54 to 2.38 >> Accuracy from 0.08 to 0.15 >> time: 0.15 seconds
d = 3072, N=100 >> Cost from 2.439 to 1.199 >> Accuracy from 0.15 to 0.74 >> time: 1.23 seconds
d = 3072, N=10000 >> Cost from 2.347 to 1.323 >> Accuracy from 0.1906 to 0.543 >> time: 3.08 minutes
N: Number of images used to train

	abs_MEAN (grad_numerical – grad_analytic)
grad_W1, grad_W1_num	7.448042807619361e-12
grad_W2, grad_W2_num	7.532225861799947e-12
grad_b1, grad_b1_num	8.697536371671256e-12
grad_b2, grad_b2_num	1.3584279534573085e-11

Table-2: Mean of errors

Moreover, we have done the Gradient Sanity check:

We were able to achieve an overfitting with a very small loss (J_epochs_train) (1.12059233) and a very high accuracy (0.82) on the training data.

Only 1 batch (100 images) is used:

Parameter	Value
n_epochs	200
batch_size	100
eta	0.001
lambda_cost	0
d	3072
m	50

Table-3: Parameters used

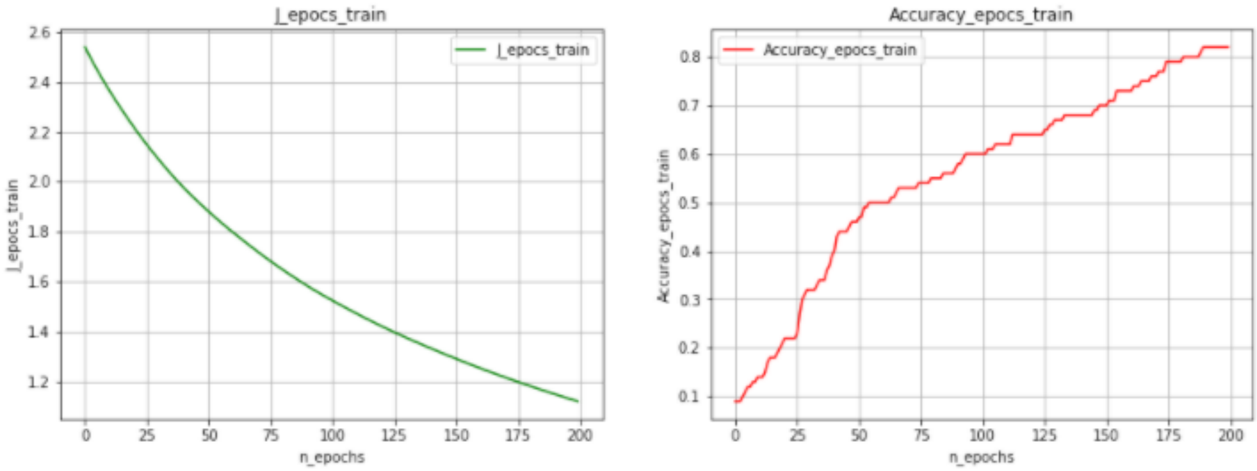


Figure-1: Overfitting

Question-2.ii: The curves for the training and validation loss/cost when using the cyclical learning rates with the default values, that is replicate figures 3 and 4. Also comment on the curves.

Answer-2.ii: We trained our algorithm to evaluate the cost with 3 different methods

a) the entire training data:
all 10.000 images were used in 'Dataset/data_batch_1' for **training** and
all 10.000 images in 'Dataset/data_batch_2' were used for **validation** to evaluate the cost

b) batch aggregation: Basically, this is kind of the average of batch costs.
Each time, a cost was calculated per batch (100 images). Then, this was added to the previous cost calculated. Finally, the sum was divided by the number of batches added until now. For a better understanding, you might check the function: Train_Cyclical

```
J_train = network1.Cost(X_batch, Y_batch, W, b, lambda_cost)
J_train_sum += J_train
smooth_cost = J_train_sum/(cost_record + 1)
```

c) batch aggregation ratio: Basically, this will add the new batch's cost to the previous cost with a ratio. For a better understanding, you might check the function: Train_Cyclical

```
J_train = network1.Cost(X_batch, Y_batch, W, b, lambda_cost)
if n_records == 0:
  smooth_cost = J_train
else:
  #smooth_cost = 0.999*smooth_cost + 0.001 * J_train
  smooth_cost = 0.99*smooth_cost + 0.01 * J_train
```

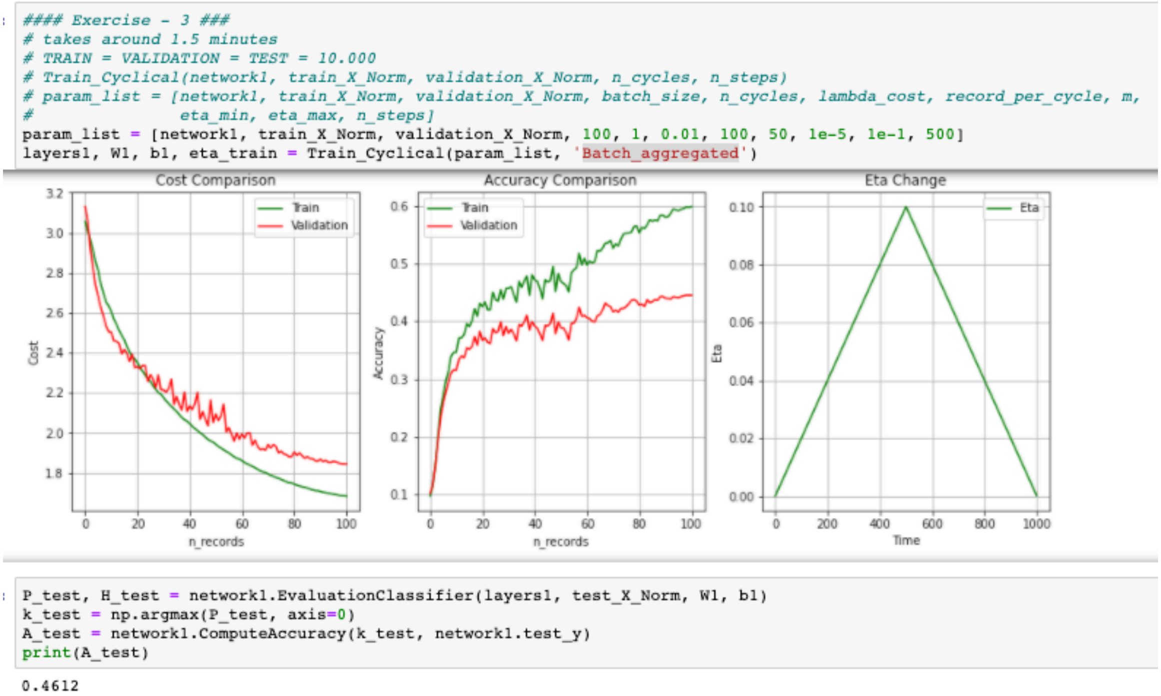
Cost Method	batch_size	n_cycles	lambda_cost	Record Per cycle	m	eta_min	eta_max	n_steps	Test Accuracy	Calculation Time
ALL_Data	100	1	0.01	100	50	1e-5	1e-1	500	0.454	2 min 39 sec
Batch_aggregated	100	1	0.01	100	50	1e-5	1e-1	500	0.4612	1 min 23 sec
Batch_aggregated_ratio	100	1	0.01	100	50	1e-5	1e-1	500	0.4526	1 min 25 sec
ALL_Data	100	3	0.01	100	50	1e-5	1e-1	800	0.4599	7 min 59 sec
Batch_aggregated	100	3	0.01	100	50	1e-5	1e-1	800	0.4641	4 min 14 sec
Batch_aggregated_ratio	100	3	0.01	100	50	1e-5	1e-1	800	0.4636	4 min 15 sec
Batch_aggregated	100	3	0.01	100	100	1e-5	1e-1	800	0.4763	4 min 47 sec

Table-4: Method and Parameters used

n_epochs = int(2 * n_cycles * (n_steps / total_batch))
d = 3072 for all
Validation data = all data in 'Dataset/data_batch_2'
'Dataset/test_batch' was used to calculate the test data set's accuracy.

Above methods were tested because once we moved to the next exercise, the calculation was taking too much time. So, instead of using the entire training data to make the cost calculations, batch-based calculations were considered. It was observed that the calculation time reduced while the calculated values were more or less the same in each 3 methods used. That was kind of a validation of the methods used. So, for the rest of the assignment, "Batch_aggregated" method is decided to be utilized to save time.

***NOTE: The x-axis (n_records) shows how many times we calculated the cost, it does not represent the number of steps. To be able to know the number of steps, you need to multiply "n_records" by the "record_per_cycle" which is usually picked as 100.

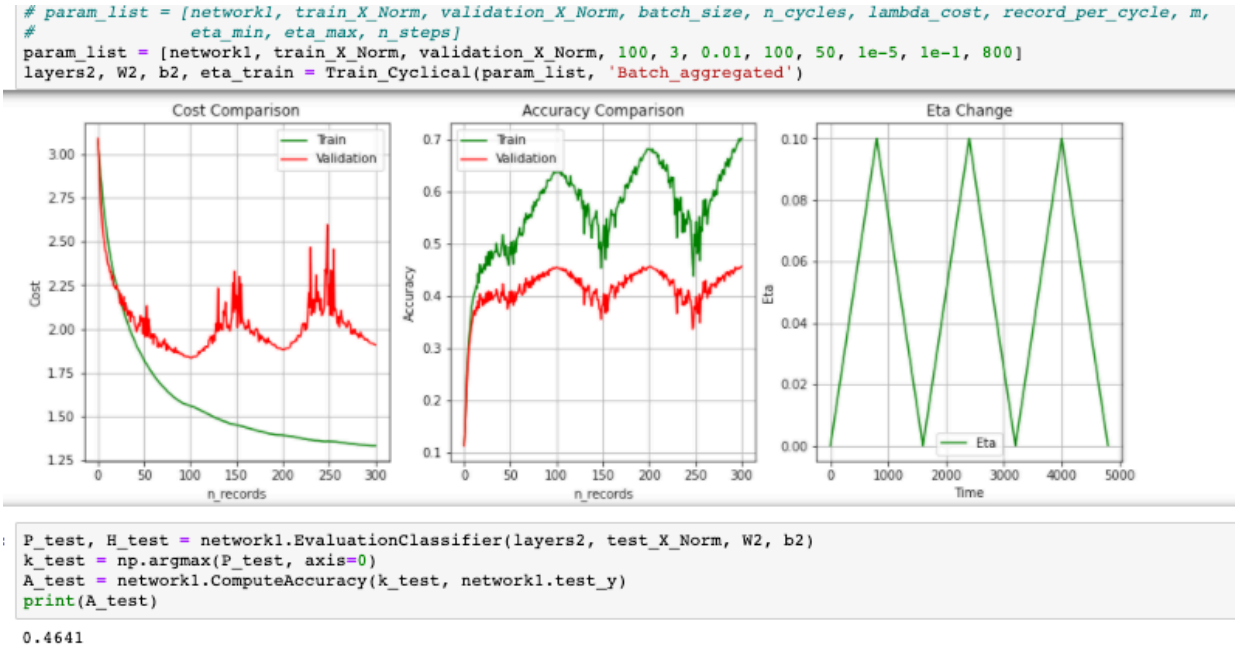


```
#### Exercise - 3 ###
# takes around 1.5 minutes
# TRAIN = VALIDATION = TEST = 10.000
# Train_Cyclical(network1, train_X_Norm, validation_X_Norm, n_cycles, n_steps)
# param_list = [network1, train_X_Norm, validation_X_Norm, batch_size, n_cycles, lambda_cost, record_per_cycle, m,
#               eta_min, eta_max, n_steps]
param_list = [network1, train_X_Norm, validation_X_Norm, 100, 1, 0.01, 100, 50, 1e-5, 1e-1, 500]
layers1, W1, b1, eta_train = Train_Cyclical(param_list, 'Batch_aggregated')
```

```
P_test, H_test = network1.EvaluationClassifier(layers1, test_X_Norm, W1, b1)
k_test = np.argmax(P_test, axis=0)
A_test = network1.ComputeAccuracy(k_test, network1.test_y)
print(A_test)
```

```
0.4612
```

Figure-2: Cost comparison and accuracy change for 1 cycle

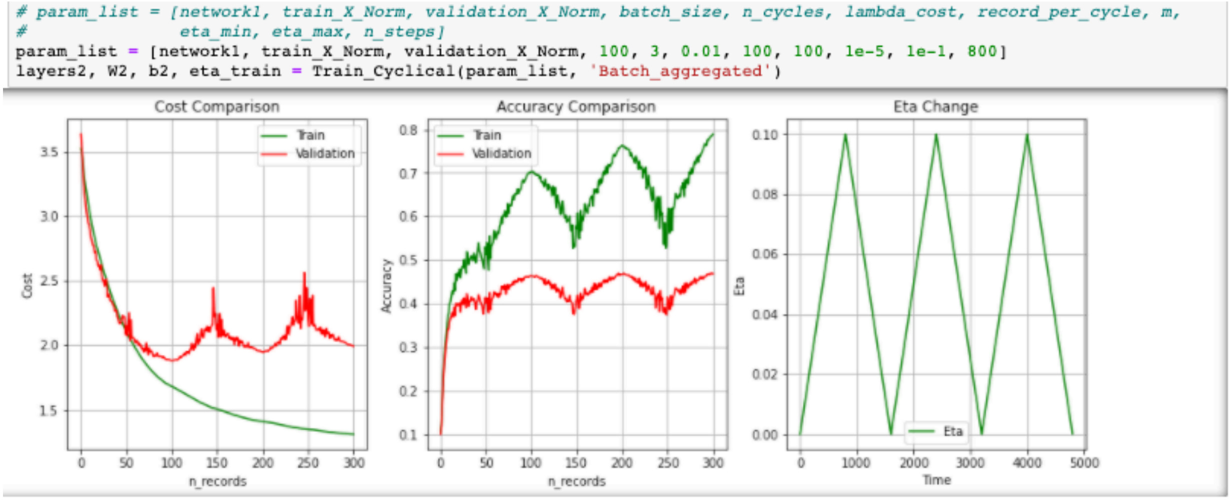


```
# param_list = [network1, train_X_Norm, validation_X_Norm, batch_size, n_cycles, lambda_cost, record_per_cycle, m,
#               eta_min, eta_max, n_steps]
param_list = [network1, train_X_Norm, validation_X_Norm, 100, 3, 0.01, 100, 50, 1e-5, 1e-1, 800]
layers2, W2, b2, eta_train = Train_Cyclical(param_list, 'Batch_aggregated')
```

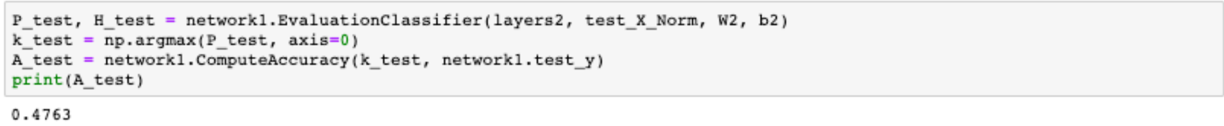
```
P_test, H_test = network1.EvaluationClassifier(layers2, test_X_Norm, W2, b2)
k_test = np.argmax(P_test, axis=0)
A_test = network1.ComputeAccuracy(k_test, network1.test_y)
print(A_test)
```

```
0.4641
```

Figure-3: Cost comparison and accuracy change for 3 cycles



```
# param_list = [network1, train_X_Norm, validation_X_Norm, batch_size, n_cycles, lambda_cost, record_per_cycle, m,
#               eta_min, eta_max, n_steps]
param_list = [network1, train_X_Norm, validation_X_Norm, 100, 3, 0.01, 100, 100, 1e-5, 1e-1, 800]
layers2, W2, b2, eta_train = Train_Cyclical(param_list, 'Batch_aggregated')
```

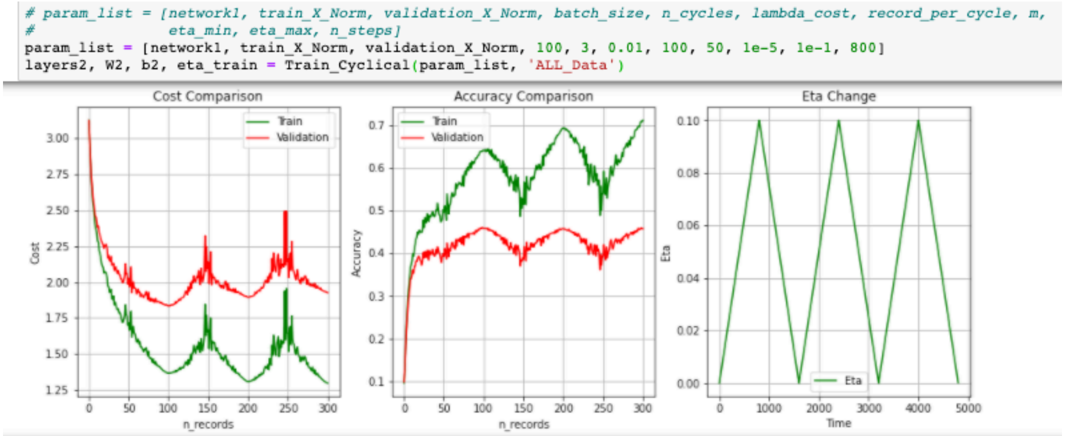


```
P_test, H_test = network1.EvaluationClassifier(layers2, test_X_Norm, W2, b2)
k_test = np.argmax(P_test, axis=0)
A_test = network1.ComputeAccuracy(k_test, network1.test_y)
print(A_test)
```

```
0.4763
```

Figure-4: Cost comparison and accuracy change for 3 cycles & 100 hidden layers

*** NOTE: The line for Training cost is not representing the 3 cycles properly since the calculation is done in an aggregated way but not using the ALL_Data each time. If we use ALL_Data as in the below figure (Figure-5), the cycles are easily observed:



```
# param_list = [network1, train_X_Norm, validation_X_Norm, batch_size, n_cycles, lambda_cost, record_per_cycle, m,
#               eta_min, eta_max, n_steps]
param_list = [network1, train_X_Norm, validation_X_Norm, 100, 3, 0.01, 100, 50, 1e-5, 1e-1, 800]
layers2, W2, b2, eta_train = Train_Cyclical(param_list, 'ALL_Data')
```

Figure-5: Cost comparison and accuracy change for 3 cycles & 50 hidden layers by using ALL_Data for cost calculations

Adding more cycles has a positive impact on the accuracy & cost for training but it doesn't have a big impact on the validation and test data. The impact was more significant once we increased the number of nodes in the hidden layer in comparison to increasing the number of cycles.

Question-2.iii: State the range of the values you searched for lambda, the number of cycles used for training during the coarse search and the hyper-parameter settings for the 3 best performing networks you trained.

Answer-2.iii: Starting from this question, 45.000 images are used for **training**, 5.000 for **validation** and 10.000 for **testing**.

As it was suggested in the assignment, the test accuracy was tested by searching for lambda on a log scale by generating one random sample in the range of 1e-5 and 1e-1 (10^l_min to 10ˆl_max):

l = l_min + (l_max - l_min) * np.random.uniform(0,1)
lambda_coarse = pow(10, l)

lambda_cost	Test Accuracy	batch_ size	n_cycles	Record Per cycle	m	eta_min	eta_max
0.003772863865559457	0.499	100	1	100	50	1e-5	1e-1
0.0004930704361181606	0.4996	100	1	100	50	1e-5	1e-1
0.0033665965785468826	0.4975	100	1	100	50	1e-5	1e-1
0.001305094636607304	0.5022	100	1	100	50	1e-5	1e-1
0.01757893078506135	0.4828	100	1	100	50	1e-5	1e-1
0.0007012053067447107	0.499	100	1	100	50	1e-5	1e-1
9.177497906795191e-05	0.502	100	1	100	50	1e-5	1e-1
0.000310949509136683	0.4969	100	1	100	50	1e-5	1e-1

Table-5: Parameters used for the coarse search

The value of lambda starting from 1e-3 to 1e-5 have better test accuracy values. Besides those numbers above, more tests were conducted, and we decided to narrow the search down between 1e-4 to 1e-5 in the next step.

Question-2.iv: State the range of the values you searched for lambda, the number of cycles used for training during the fine search and the hyper-parameter settings for the 3 best performing networks you trained.

Answer-2.iv: Here, you will have the results for the narrowed down lambda values. This time we executed for 16 lambda values:

lambda_cost	Test Accuracy	batch_ size	n_cycles	Record Per cycle	m	eta_min	eta_max
9.862809522185001e-05	0.498	100	1	100	50	1e-5	1e-1
9.054812798313612e-05	00.5001	100	1	100	50	1e-5	1e-1
1.9651064567590842e-05	0.4968	100	1	100	50	1e-5	1e-1
3.544673306817798e-05	**0.5056**	**100**	**1**	**100**	**50**	**1e-5**	**1e-1**
2.653016524046826e-05	0.4986	100	1	100	50	1e-5	1e-1
2.055764159216546e-05	0.4959	100	1	100	50	1e-5	1e-1

3.614606936334905e-05	0.5019	100	1	100	50	1e-5	1e-1
2.6196012000341928e-05	0.5001	100	1	100	50	1e-5	1e-1
2.406038940258964e-05	0.4992	100	1	100	50	1e-5	1e-1
5.9831933161984584e-05	0.495	100	1	100	50	1e-5	1e-1
3.516860763871908e-05	0.4995	100	1	100	50	1e-5	1e-1
4.459893452752506e-05	0.4944	100	1	100	50	1e-5	1e-1
1.1067150412880694e-05	0.5033	100	1	100	50	1e-5	1e-1
1.4001893917970237e-05	0.4947	100	1	100	50	1e-5	1e-1
1.2967366157634313e-05	0.4972	100	1	100	50	1e-5	1e-1
2.7575325697602636e-05	0.5029	100	1	100	50	1e-5	1e-1

Table-6: Parameters used for the fine search

Question-2.v: For your best found **lambda** setting (according to performance on the validation set), train the network on all the training data (all the batch data), except for 1000 examples in a validation set, for ˜3 cycles. Plot the training and validation loss plots and then report the learnt network's performance on the test data.

Answer-2.v: 3.544673306817798e-05 gave us the best accuracy result in the fine search, so we will use it as lambda. Once we used higher number of nodes in the hidden layer (m=200), we achieved a higher accuracy of 0.5336:

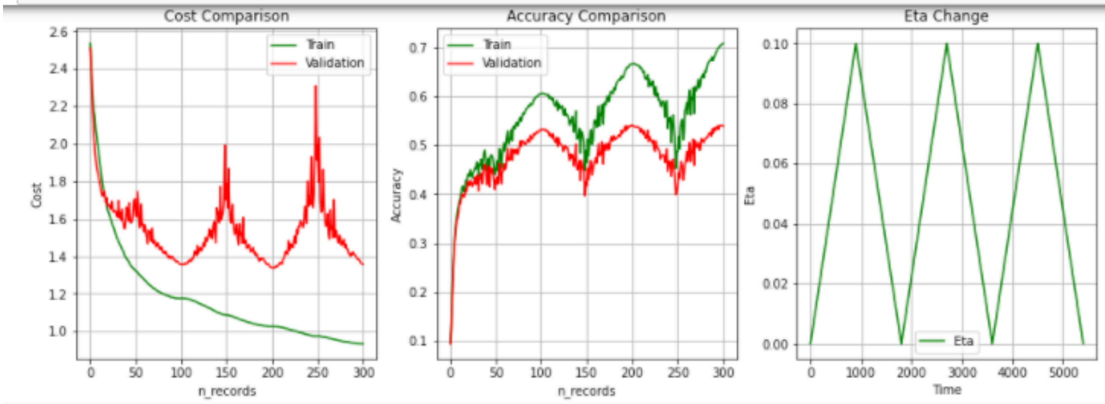
The best parameter setting:

lambda_cost	Test Accuracy	batch_ size	n_cycles	Record Per cycle	m	eta_min	eta_max
3.544673306817798e-05	**0.5336**	200	3	100	200	1e-5	1e-1

Results for some other settings changing the number of nodes in the hidden layer:

lambda_cost	Test Accuracy	batch_ size	n_cycles	Record Per cycle	m	eta_min	eta_max
3.544673306817798e-05	0.5186	100	3	100	200	1e-5	1e-1
3.544673306817798e-05	0.5051	50	3	100	200	1e-5	1e-1
3.544673306817798e-05	0.4141	10	3	100	200	1e-5	1e-1

```
lambda_coarse = 3.544673306817798e-05  #1.8544671883635666e-05
# N_CYCLEs changed >> from 1 to 3
# param_list = [network1, train_X_Norm, validation_X_Norm, batch_size, n_cycles, lambda_cost, record_per_cycle, m,
#               eta_min, eta_max]

param_list = [network1, train_X_Norm, validation_X_Norm, 100, 3, lambda_coarse, 100, 200, 1e-5, 1e-1]
layers, W, b, eta_train = Train_Cyclical_Coarse_lambda_smooth_aggregated_2(param_list)
```



```
P_test, H_test = network1.EvaluationClassifier(layers, test_X_Norm, W, b)
k_test = np.argmax(P_test, axis=0)
A_test = network1.ComputeAccuracy(k_test, network1.test_y)
print(A_test)
```

```
0.5336
```

DL_assignment2_melih_E123_NO_PRINT

September 5, 2020

[1]:

```
import numpy as np
import pickle
import matplotlib.pyplot as plt
import scipy.io as sio
from sklearn import preprocessing

import gradient
import dataset
import computations
import layer
#from layer import Linear, Softmax, Gradient
import network

import datetime
import time
```

[2]:

```
np.random.seed(400)
np.seterr(over='raise');
plt.rcParams['figure.figsize'] = (15.0, 5.0)
```

[3]:

```
network1 = network.Network()
cifar = dataset.CIFAR_IMAGES()
#asgn1.test_batch_images(cifar_batch1)
```

[4]:

```
#### Exercise - 1 ###
# Read in the data & initialize the parameters of the network

filePathLocal_labels = 'Dataset/batches.meta'
filePathLocal_batch = 'Dataset/data_batch_1'
filePathLocal_data_TRAIN = 'Dataset/data_batch_1'
filePathLocal_data_VALIDATION = 'Dataset/data_batch_2'
filePathLocal_data_TEST = 'Dataset/test_batch'

filePathList = (filePathLocal_data_TRAIN, filePathLocal_data_VALIDATION,␣
 ↪filePathLocal_data_TEST)
```

```
# Read TRAIN, VALIDATION, TEST data into numpy arrays (numpy.ndarray) from␣
 ↪local files
network1.ReadData(cifar, filePathList)
# X = (d, N), Y = (K, N), y = (N,)   # N=number of total images in X
# X = (3072, 10000), Y = (10, 10000), y = (10000,)

# Find the MEAN and STD of trainX and broadcast them for matrix calculations
# trainX_Broadcast_MeanStd = (mean_train_X_broadcast, std_train_X_broadcast)
trainX_Broadcast_MeanStd = network1.MeanStd_Train_X(network1.train_X)

# Transform the INPUT to have zero mean ** Check that one if we need to␣
 ↪transform all of them separately or
# only having the normalization as in here??
# Normalize all INPUT data by using MEAN and STD of TRAIN DATA
# OR should we normalize each of them using their own MEAN and STD ** selected␣
 ↪that one
# NORMALIZE by using TRAINING DATA
train_X_Norm = network1.NormalizeData(network1.train_X,␣
 ↪trainX_Broadcast_MeanStd)
#validation_X_Norm = network1.NormalizeData(network1.validation_X,␣
 ↪trainX_Broadcast_MeanStd)
# NORMALIZE by using VALIDATION DATA
validation_X_Norm = network1.NormalizeData_Per_DataSet(network1.validation_X)
#test_X_Norm = network1.NormalizeData(network1.test_X, trainX_Broadcast_MeanStd)
# NORMALIZE by using TEST DATA
test_X_Norm = network1.NormalizeData_Per_DataSet(network1.test_X)

# mu = 0; d = network1.train_X.shape[0]; m = 50; K = network1.train_Y.shape[0]
# we will use only 20 of 3072 to have a dimension reduction in comparing␣
 ↪grad_analytic and grad_Numerical
mu = 0; d = 20; m = 50; K = network1.train_Y.shape[0]

initial_sizes = (mu, d, m, K)

#sigma1 = 1 / int(np.sqrt(d))
sigma1 = 1 / np.sqrt(d)
sigma2 = 1 / np.sqrt(m)

# Generate W1, W2, b1, b2 matrices with initial values
#(W1, W2, b1, b2) = network1.Initialize_W_b(d, m, K, sigma1, sigma2)
(W1, W2, b1, b2) = network1.Initialize_W_b(initial_sizes, sigma1, sigma2)
# W1 = (m, d),      W2 = (K, m),     b1 = (m, 1),    b2 = (K, 1)
# W1 = (50, 3072), W2 = (10, 50),   b1 = (50, 1),   b2 = (10, 1) # if we use the␣
 ↪whole dimensions/features
# W1 = (50, 20),    W2 = (10, 50),   b1 = (50, 1),   b2 = (10, 1) # if we use 20␣
 ↪dimensions/features
```

[5]:

```
'''
print("Mean-STD:\n␣
↪trainX_Broadcast_MeanStd[0]=\n{}\n\ntrainX_Broadcast_MeanStd[1]=\n{}".\
      format(trainX_Broadcast_MeanStd[0], trainX_Broadcast_MeanStd[1]))
print("\n\nMean-STD:\n train_X_Norm.mean(axis=1)={}\n\n".format(train_X_Norm.
↪mean(axis=1)))
'''
print("Shape check:\n train_X_Norm={}\t validation_X_Norm={}\t test_X_Norm={}".
↪format(train_X_Norm.shape, validation_X_Norm.shape, test_X_Norm.shape))
print(" train_Y={}\t\t validation_Y={}\t\t test_Y={}".format(network1.train_Y.
↪shape, network1.validation_Y.shape, network1.test_Y.shape))
print(" train_y={}\t\t validation_y={}\t\t\t test_y={}".format(network1.train_y.
↪shape, network1.validation_y.shape, network1.test_y.shape))
print(" W1={}\t\t\t W2={}\t\t\t\t b1={}\t\t\t b2={}".format(W1.shape, W2.shape,␣
↪b1.shape, b2.shape))
```

```
Shape check:
 train_X_Norm=(3072, 10000)      validation_X_Norm=(3072, 10000)
test_X_Norm=(3072, 10000)
 train_Y=(10, 10000)             validation_Y=(10, 10000)
test_Y=(10, 10000)
 train_y=(10000,)                validation_y=(10000,)
test_y=(10000,)
 W1=(50, 20)                     W2=(10, 50)                             b1=(50,
1)                      b2=(10, 1)
```

[6]:

```
#### Exercise - 2 ###
# Compute the gradients for the network parameters

# only 20 features are used for gradient-TEST calculations and
# 2 images only not the whole batch (10k images)
# X_batch = train_X_Norm[0:d, :]                 # if we would like to test␣
↪with the entire batch
# num_image = number of images to use in gradient comparison calculations
num_image = 2
X_batch = train_X_Norm[0:d, 0:num_image]
Y_batch = network1.train_Y[:, 0:num_image]
y_batch = network1.train_y[0:num_image]

linearLayer1 = layer.Linear()
reluLayer = layer.ReLU()             # not an exact layer but operational step..
linearLayer2 = layer.Linear()
softmaxLayer = layer.Softmax()       # not an exact layer but operational step..

layers = [linearLayer1, reluLayer, linearLayer2, softmaxLayer]

'''
```

```
S1 = linearLayer1.Forward(X_batch, W1, b1)
# h = X(layer_no) ... X(0)=represents the input
H = ReLUlayer.Forward(S1)
S = linearLayer1.Forward(H, W2, b2)
P = softmaxLayer.Forward(S)
'''
W = [W1, W2]
b = [b1, b2]

P, H = network1.EvaluationClassifier(layers, X_batch, W, b)
```

[7]:

```
#### Exercise - 2 ###
# let's test the same process with a loop. this time, in the loop, one layer's␣
↪output will be
# the input of the following layer
num_image = 2
X_batch = train_X_Norm[0:d, 0:num_image]
Y_batch = network1.train_Y[:, 0:num_image]
y_batch = network1.train_y[0:num_image]

linearLayer1 = layer.Linear()
reluLayer = layer.ReLU()             # not an exact layer but operational step..
linearLayer2 = layer.Linear()
softmaxLayer = layer.Softmax()

layers = [linearLayer1, reluLayer, linearLayer2, softmaxLayer]

W = [W1, W2]
b = [b1, b2]


P, H = network1.EvaluationClassifier_loop(layers, X_batch, W, b)
```

[8]:

```
def GradientComparison_Analytical_Numerical(X, H, Y, lambda_cost = 0, h = 1e-5):
    grad = gradient.Gradient()
    G = -np.subtract(Y, P) # only for 2 images (change num_image value if you␣
↪want to use different # of images)
    N = Y_batch.shape[1]

    (grad_W2, grad_b2, G) = grad.ComputeGradients_Linear_HiddenLayer(N, G, H,␣
↪lambda_cost, W2)
    #print('grad_W2.shape={}'.format(grad_W2.shape))
    #print('grad_b2.shape={}'.format(grad_b2.shape))

    G = reluLayer.Backward(G, H)
    N = H.shape[1]
    (grad_W1, grad_b1) = grad.ComputeGradients_Linear_FirstLayer(N, G, X_batch,␣
↪lambda_cost, W1)

    start_time = datetime.datetime.now()

    W = [W1, W2]
    b = [b1, b2]
    #X_all = [X_batch, H]
    #(grad_W_num, grad_b_num) = grad.ComputeGradsNumSlow(layers, X_batch,␣
↪Y_batch, W, b, lambda_cost, h=1e-5)
    (grad_W_num, grad_b_num) = grad.ComputeGradsNumSlow(X, Y, W, b,␣
↪lambda_cost, h)

    grad_W1_num = grad_W_num[0]
    grad_W2_num = grad_W_num[1]

    grad_b1_num = grad_b_num[0]
    grad_b2_num = grad_b_num[1]

    end_time = datetime.datetime.now()

    print("Calculation time of GradsNumSlow: " + str(end_time - start_time))

    print('\n\n***** grad_W1, grad_W1_num')
    grad.CompareGradients_W(grad_W1, grad_W1_num)

    print('\n\n***** grad_W2, grad_W2_num')
    grad.CompareGradients_W(grad_W2, grad_W2_num)

    print('\n\n***** grad_b1, grad_b1_num')
    grad.CompareGradients_b(grad_b1, grad_b1_num)

    print('\n\n***** grad_b2, grad_b2_num')
    grad.CompareGradients_b(grad_b2, grad_b2_num)
```

[9]:

```
# takes less than a second
GradientComparison_Analytical_Numerical(X_batch, H, Y_batch, lambda_cost = 0, h␣
↪= 1e-5)
```

```
Calculation time of GradsNumSlow: 0:00:00.282604


***** grad_W1, grad_W1_num

grad_W_difference_MEAN = 7.448042807619361e-12
grad_W_difference_MIN = 0.0
```

```
grad_W_difference_MAX = 4.180217977323153e-11

grad_W_MIN = 0.0
grad_W_num_MIN = 0.0
grad_W_MAX = 0.263187266439426
grad_W_num_MAX = 0.26318726642493573


***** grad_W2, grad_W2_num

grad_W_difference_MEAN = 7.532225861799947e-12
grad_W_difference_MIN = 0.0
grad_W_difference_MAX = 4.598055269866563e-11

grad_W_MIN = 0.0
grad_W_num_MIN = 0.0
grad_W_MAX = 0.6481946016209915
grad_W_num_MAX = 0.648194601637897


***** grad_b1, grad_b1_num

grad_b_difference_MEAN = 8.697536371671256e-12
grad_b_difference_MIN = 0.0
grad_b_difference_MAX = 2.8872293444948127e-11

grad_b_MIN = 0.0
grad_b_num_MIN = [0.]

grad_b_MAX = 0.2321359378763498
grad_b_num_MAX = [0.23213594]


***** grad_b2, grad_b2_num

grad_b_difference_MEAN = 1.3584279534573085e-11
grad_b_difference_MIN = 1.8527263057066534e-12
grad_b_difference_MAX = 3.675354465215719e-11

grad_b_MIN = 0.08160652448507566
grad_b_num_MIN = [0.08160652]

grad_b_MAX = 0.4297535107822109
grad_b_num_MAX = [0.42975351]
```

[10]:

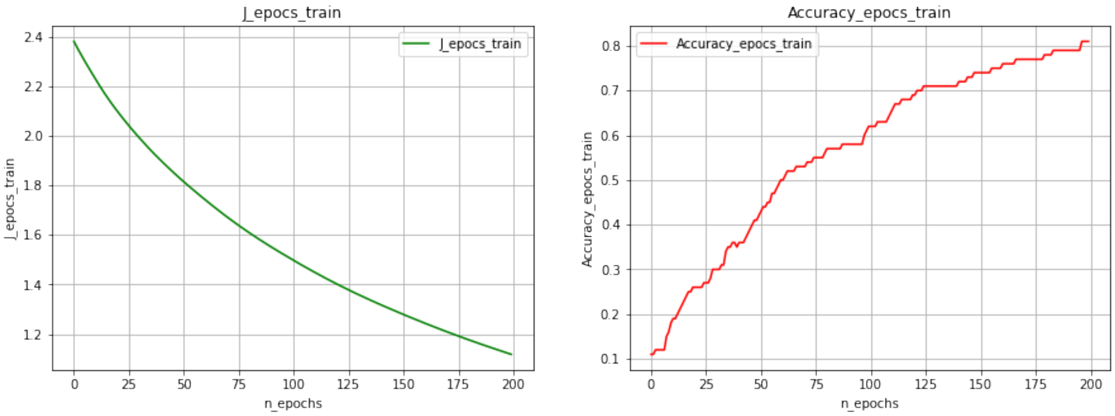
```
def GradientSanityCheck(network1, train_X_Norm):
    n_epocs = 200    # number of times we will iterate on the entire data (10K␣
↪images in our case)
    batch_size = 100    # the size of the mini-batch. in other words, number of␣
↪images in 1 mini-batch.
    eta = 0.001       # learning rate (step-size)
    lambda_cost = 0  # regularization coefficient (punishment)
    #d = train_X_Norm.shape[0]
    d = 3072
    m = 50            # number of nodes in the hidden layer
    K = network1.train_Y.shape[0]  # number of classes

    # Generate W1, W2, b1, b2 matrices with initial values
    # (W1, W2, b1, b2) = network1.Initialize_W_b(d, m, K)
    sigma1 = 1 / int(np.sqrt(d))
    sigma2 = 1 / np.sqrt(m)
    #(W1, W2, b1, b2) = network1.Initialize_W_b(d, m, K, sigma1, sigma2)
    initial_sizes = [0, d, m, K]
    (W1, W2, b1, b2) = network1.Initialize_W_b(initial_sizes, sigma1, sigma2)

    W = [W1, W2]
    b = [b1, b2]

    J_epocs_train = np.zeros(n_epocs)         # cost array     - will keep␣
↪costs per epoc (iteration)
    Accuracy_epocs_train = np.zeros(n_epocs)  # accuracy array - will keep␣
↪accuracy per epoc (iteration)

    linearLayer1 = layer.Linear()
    reluLayer = layer.ReLU()             # not an exact layer but operational␣
↪step..
    linearLayer2 = layer.Linear()
    softmaxLayer = layer.Softmax()       # not an exact layer but operational␣
↪step..
    ReLUlayer = layer.ReLU()
    grad = gradient.Gradient()

    layers = [linearLayer1, reluLayer, linearLayer2, softmaxLayer]
    # total_batch = how many mini-batches will we need to cover the entire␣
↪training set?
    #total_batch = int(train_X_normalized.shape[1] / batch_size)
    #n_sanity_batch = 1
    # we will only use 100 images for sanity check that means there will be␣
↪only 1 mini-batch
    n_sanity_batch = 1
    n_test_images = batch_size * n_sanity_batch
    start_time = datetime.datetime.now()
    for e in range(n_epocs):
        for batch in range(n_sanity_batch):
            index_list = list(range(batch * batch_size, (batch + 1) *␣
↪batch_size))
            # shuffling is not necessary but good to have
            #np.random.shuffle(index_list)
            #X_batch = train_X_Norm[:, index_list]
            X_batch = train_X_Norm[0:d, index_list]
            Y_batch = network1.train_Y[:, index_list]

            P, H = network1.EvaluationClassifier(layers, X_batch, W, b)

            G2 = -np.subtract(network1.train_Y[:, index_list], P)
            N2 = Y_batch.shape[1] #N

            # N, G
            (grad_W2, grad_b2, G1) = grad.
↪ComputeGradients_Linear_HiddenLayer(N2, G2, H, lambda_cost, W2)
            G0 = reluLayer.Backward(G1, H)
            N1 = H.shape[1] #N
            # N, G
            (grad_W1, grad_b1) = grad.ComputeGradients_Linear_FirstLayer(N1,␣
↪G0, X_batch, lambda_cost, W1)

            W1 = W1 - eta * grad_W1
            W2 = W2 - eta * grad_W2

            #W1 = Wstar1
            #W2 = Wstar2

            bstar_m1 = b1 - eta * grad_b1
            b1 = bstar_m1[:, :1]
            #b1 = bstar1

            bstar_m2 = b2 - eta * grad_b2
            b2 = bstar_m2[:, :1]
            #b2 = bstar2

        W = [W1, W2]
        b = [b1, b2]

        #J_train = network1.Cost(X_batch, Y_batch, W, b, lambda_cost)
        #J_train = network1.Cost(train_X_Norm[:, 0:], Y_batch, W, b,␣
↪lambda_cost)
        #J_train = network1.Cost(train_X_Norm[:, :batch*total_batch], network1.
↪train_y[index_list],\
         #                        W, b, lambda_cost)
        J_train = network1.Cost(train_X_Norm[0:d, 0:n_test_images], network1.
↪train_Y[:, 0:n_test_images],\
                                W, b, lambda_cost)

        J_epocs_train[e] = J_train

        #P_train = self.EvaluationClassifier(train_X_normalized, W, b)
        #k_train = np.argmax(P, axis=0)

        P, H = network1.EvaluationClassifier(layers, train_X_Norm[0:d, 0:
↪n_test_images], W, b)
        k_train = np.argmax(P, axis=0)

        A_train = network1.ComputeAccuracy(k_train, network1.train_y[0:
↪n_test_images])
        Accuracy_epocs_train[e] = A_train

        #self.Plot_Train_Validation_Cost_Accurracy(J_epocs_train,␣
↪Accuracy_epocs_train)

    end_time = datetime.datetime.now()

    print("Calculation time of GradientSanityCheck: " + str(end_time -␣
↪start_time))
    ## d = 20,   N=100    >>> Cost from 2.54  to 2.38    >>> Accuracy from 0.08␣
↪to 0.15    >> time: 0.15 seconds
    ## d = 3072, N=100    >>> Cost from 2.439 to 1.199   >>> Accuracy from 0.15␣
↪to 0.74    >> time: 1.23 seconds
    ## d = 3072, N=10000  >>> Cost from 2.347 to 1.323   >>> Accuracy from 0.
↪1906 to 0.543 >> time: 3.08 minutes

    print(J_epocs_train)
    print(Accuracy_epocs_train)

    Plot_Train_Cost_Accurracy(J_epocs_train, Accuracy_epocs_train)
```

[11]:

```
def Plot_Train_Cost_Accurracy(J_epocs_train, Accuracy_epocs_train):
        plt.subplot(1,2,1)
        plt.plot(J_epocs_train, 'g-', label='J_epocs_train')
        plt.title('J_epocs_train')
        plt.xlabel('n_epochs')
        plt.ylabel('J_epocs_train')
        plt.legend()
        plt.grid('on')
        plt.subplot(1,2,2)
        plt.plot(Accuracy_epocs_train, 'r-', label='Accuracy_epocs_train')
        plt.title('Accuracy_epocs_train')
        plt.xlabel('n_epochs')
        plt.ylabel('Accuracy_epocs_train')
        plt.legend()
        plt.grid('on')
```

[12]:
```
# takes 2 seconds
GradientSanityCheck(network1, train_X_Norm)
```

```
Calculation time of GradientSanityCheck: 0:00:01.140827
[2.38141841 2.36450693 2.34802706 2.33194751 2.31618708 2.30089071
 2.28588608 2.27091724 2.25622107 2.24157926 2.22699612 2.2126453
 2.19846949 2.18458225 2.17115511 2.15809151 2.14531877 2.13276476
 2.12052738 2.10864204 2.09704919 2.08555144 2.07424372 2.06321992
 2.05218869 2.04139736 2.03080174 2.0204127  2.01013614 2.00001399
 1.99006018 1.98020556 1.97045738 1.9608275  1.95132739 1.94191467
 1.93270478 1.92361452 1.91469319 1.90590944 1.89718644 1.88857539
 1.88003341 1.87158506 1.86323861 1.85496985 1.84682741 1.83875007
 1.83076322 1.82286238 1.81504002 1.80733752 1.79967122 1.79208482
 1.78451646 1.77705237 1.76962499 1.76226563 1.75497114 1.74765122
 1.74041076 1.73319945 1.72608571 1.71903342 1.71205125 1.70508384
 1.69808495 1.69118195 1.68434652 1.67760025 1.6709648  1.66440851
 1.6578919  1.65145517 1.64509149 1.63876215 1.63247444 1.62626872
 1.62017265 1.61407749 1.60807326 1.60208982 1.59617654 1.59030353
 1.58453471 1.57876099 1.57304723 1.56740833 1.56183348 1.55629099
 1.55078026 1.5453238  1.5399105  1.53456212 1.52923821 1.52395741
 1.51870099 1.51346296 1.50817696 1.50291301 1.49770558 1.49252834
 1.48739826 1.48231008 1.47725295 1.47222464 1.46723126 1.46227512
 1.45732934 1.452448   1.44757357 1.44275555 1.43796133 1.43316703
 1.42840992 1.42370042 1.41903333 1.41441738 1.40982201 1.4052953
 1.40077748 1.39628941 1.39185767 1.38740666 1.38301454 1.37864784
 1.37434812 1.37009831 1.36582882 1.36162834 1.3574411  1.35326575
 1.3491516  1.34505915 1.34104438 1.33701474 1.33298478 1.3290215
 1.32508    1.32117517 1.31727002 1.31340005 1.30955927 1.30569603
 1.30188087 1.2980716  1.29426344 1.29048386 1.28672573 1.28297892
 1.27927922 1.27558792 1.27192321 1.26826412 1.26464996 1.26100194
 1.25740676 1.25378203 1.25019182 1.24665256 1.24311136 1.23959565
 1.23610504 1.23263012 1.22918455 1.22573925 1.22234476 1.21890643
 1.21550129 1.21214747 1.20876068 1.20543558 1.20210198 1.19878096
 1.19550507 1.19223844 1.18899254 1.18573881 1.18251548 1.17931265
 1.17615472 1.1729755  1.16981313 1.16669174 1.16357147 1.16045891
 1.15736941 1.1543202  1.1512512  1.14822182 1.14518571 1.14216725
 1.13917123 1.13619459 1.13324875 1.13028241 1.1273352  1.12442549
 1.12151828 1.11862923]
[0.11 0.11 0.12 0.12 0.12 0.12 0.12 0.15 0.16 0.18 0.19 0.19 0.2  0.21
 0.22 0.23 0.24 0.25 0.25 0.26 0.26 0.26 0.26 0.26 0.27 0.27 0.27 0.28
 0.3  0.3  0.3  0.3  0.31 0.31 0.34 0.35 0.35 0.36 0.36 0.35 0.36 0.36
 0.36 0.37 0.38 0.39 0.4  0.41 0.41 0.42 0.43 0.44 0.44 0.45 0.45 0.47
 0.47 0.48 0.49 0.5  0.5  0.51 0.52 0.52 0.52 0.52 0.53 0.53 0.53 0.53
 0.53 0.54 0.54 0.54 0.55 0.55 0.55 0.55 0.55 0.56 0.57 0.57 0.57 0.57
 0.57 0.57 0.57 0.58 0.58 0.58 0.58 0.58 0.58 0.58 0.58 0.58 0.58 0.6
 0.61 0.62 0.62 0.62 0.62 0.63 0.63 0.63 0.63 0.63 0.64 0.65 0.66 0.67
 0.67 0.67 0.68 0.68 0.68 0.68 0.68 0.69 0.69 0.7  0.7  0.7  0.71 0.71
 0.71 0.71 0.71 0.71 0.71 0.71 0.71 0.71 0.71 0.71 0.71 0.71 0.71 0.71
 0.72 0.72 0.72 0.72 0.73 0.73 0.73 0.74 0.74 0.74 0.74 0.74 0.74 0.74
 0.74 0.75 0.75 0.75 0.75 0.75 0.76 0.76 0.76 0.76 0.76 0.76 0.77 0.77
 0.77 0.77 0.77 0.77 0.77 0.77 0.77 0.77 0.77 0.77 0.77 0.78 0.78 0.78
 0.78 0.79 0.79 0.79 0.79 0.79 0.79 0.79 0.79 0.79 0.79 0.79 0.79 0.79
 0.81 0.81 0.81 0.81]
```



```
[13]: def Vanilla_MiniBacth_GD(network1, train_X_Norm):
          # GradientSanityCheck_FullTraining(network1, train_X_Norm)
          # This one is called "Vanilla mini-batch gradient descent"
          n_epocs = 200     # number of times we will iterate on the entire data (10K␣
       ↪images in our case)
          batch_size = 100     # the size of a mini-batch. in other words, number of␣
       ↪images in 1 mini-batch.
          eta = 0.001       # learning rate (step-size)
          #eta = 0.001      # MEL: If I use an eta bigger than 1e-3, I receive an␣
       ↪overflow error for SOFTMAX
          lambda_cost = 0  # regularization coefficient (punishment)
          #lambda_cost = 0.01
          d = train_X_Norm.shape[0]
          m = 50
          K = network1.train_Y.shape[0]  # number of classes

          # Generate W1, W2, b1, b2 matrices with initial values
```

```
    # (W1, W2, b1, b2) = network1.Initialize_W_b(d, m, K)
    sigma1 = 1 / int(np.sqrt(d))
    sigma2 = 1 / np.sqrt(m)
    #(W1, W2, b1, b2) = network1.Initialize_W_b(d, m, K, sigma1, sigma2)
    initial_sizes = [0, d, m, K]
    (W1, W2, b1, b2) = network1.Initialize_W_b(initial_sizes, sigma1, sigma2)

    W = [W1, W2]
    b = [b1, b2]

    J_epocs_train = np.zeros(n_epocs)         # cost array     - will keep␣
↪costs per epoc (iteration)
    Accuracy_epocs_train = np.zeros(n_epocs)  # accuracy array - will keep␣
↪accuracy per epoc (iteration)

    linearLayer1 = layer.Linear()
    reluLayer = layer.ReLU()             # not an exact layer but operational␣
↪step..
    linearLayer2 = layer.Linear()
    softmaxLayer = layer.Softmax()       # not an exact layer but operational␣
↪step..
    ReLUlayer = layer.ReLU()
    grad = gradient.Gradient()

    layers = [linearLayer1, reluLayer, linearLayer2, softmaxLayer]
    # total_batch = how many mini-batches will we need to cover the entire␣
↪training set?
    total_batch = int(train_X_Norm.shape[1] / batch_size)
    #n_sanity_batch = 1
    # we will only use 100 images for sanity check that means there will be␣
↪only 1 mini-batch
    #n_sanity_batch = 1
    #n_test_images = batch_size * n_sanity_batch

    start_time = datetime.datetime.now()
    for e in range(n_epocs):
        for batch in range(total_batch):
            index_list = list(range(batch * batch_size, (batch + 1) *␣
↪batch_size))
            # shuffling is not necessary but good to have
            #np.random.shuffle(index_list)
            #X_batch = train_X_Norm[:, index_list]
            X_batch = train_X_Norm[:, index_list]
            Y_batch = network1.train_Y[:, index_list]

            P, H = network1.EvaluationClassifier(layers, X_batch, W, b)
            G2 = -np.subtract(network1.train_Y[:, index_list], P)  # G
            N2 = Y_batch.shape[1] #N

            # N, G
            (grad_W2, grad_b2, G1) = grad.
↪ComputeGradients_Linear_HiddenLayer(N2, G2, H, lambda_cost, W2)
            G0 = reluLayer.Backward(G1, H)
            N1 = H.shape[1] #N
            # N, G
            (grad_W1, grad_b1) = grad.ComputeGradients_Linear_FirstLayer(N1,␣
↪G0, X_batch, lambda_cost, W1)

            # W1star, W2star
            W1 = W1 - eta * grad_W1
            W2 = W2 - eta * grad_W2

            # b1star, b2star
            bstar_m1 = b1 - eta * grad_b1
            b1 = bstar_m1[:, :1]  # there's a broadcast issue needs to be␣
↪fixed, that's why we pick only 1 column

            bstar_m2 = b2 - eta * grad_b2
            b2 = bstar_m2[:, :1]  # broadcast issue, this is a quick workaround␣
↪- use 1 column

        W = [W1, W2]
        b = [b1, b2]

        J_train = network1.Cost(train_X_Norm, network1.train_Y, W, b,␣
↪lambda_cost)

        J_epocs_train[e] = J_train

        P, H = network1.EvaluationClassifier(layers, train_X_Norm, W, b)
        k_train = np.argmax(P, axis=0)

        A_train = network1.ComputeAccuracy(k_train, network1.train_y)
        Accuracy_epocs_train[e] = A_train

        #self.Plot_Train_Validation_Cost_Accurracy(J_epocs_train,␣
↪Accuracy_epocs_train)

    end_time = datetime.datetime.now()
    print("Calculation time of GradientSanityCheck: " + str(end_time -␣
↪start_time))

    print(J_epocs_train)
    print(Accuracy_epocs_train)
```

[14]:
```
#GradientSanityCheck_FullTraining(network1, train_X_Norm)
# takes around 3-3.5 minutes
Vanilla_MiniBacth_GD(network1, train_X_Norm)
```

```
Calculation time of GradientSanityCheck: 0:03:02.023809
[2.22799015 2.27064244 2.4520189  3.14684817 2.60470707 2.40477466
 2.16056023 2.11753492 2.15546103 2.21632305 2.13731107 2.04741237
 1.93581681 1.86861164 1.82401917 1.81911749 1.79649215 1.76987219
 1.75516484 1.74479231 1.7532359  1.76821934 1.76079228 1.72981642
 1.70511877 1.69049981 1.68224735 1.6796235  1.67076782 1.66707202
 1.65842817 1.65348078 1.64515917 1.63966521 1.63320274 1.62901528
 1.62523961 1.62268987 1.62187167 1.62013767 1.62130716 1.62033927
 1.617709   1.6124351  1.60391631 1.59500496 1.58477652 1.57682676
 1.56834903 1.56267671 1.55585043 1.55172019 1.54641864 1.54351545
 1.53966646 1.5392158  1.53870426 1.54062236 1.54395728 1.54485049
 1.54871393 1.54264392 1.53819469 1.53021933 1.52306363 1.51802995
 1.51288988 1.51011819 1.50489087 1.50306641 1.49840728 1.49735075
 1.49259763 1.49129901 1.48688049 1.48434576 1.47969212 1.47578124
 1.47163702 1.46748457 1.46403305 1.45959272 1.45681042 1.4526378
 1.45084687 1.4469448  1.44677586 1.44369647 1.44728259 1.44599759
 1.45539208 1.45929676 1.4714222  1.47739409 1.48314469 1.47693004
 1.48039385 1.47474528 1.48127296 1.4734616  1.47371473 1.45878825
 1.45025074 1.43766938 1.43035042 1.42291519 1.41779358 1.41391613
 1.41038976 1.40896994 1.40709876 1.40860744 1.40967189 1.41325373
 1.42014307 1.41964775 1.42906693 1.42154886 1.42080723 1.40831631
 1.40036712 1.38978331 1.38301162 1.37798607 1.37404107 1.37157536
 1.37026637 1.37020441 1.37299729 1.37779775 1.38639137 1.39624021
 1.41172698 1.42344956 1.42822314 1.42979194 1.41153308 1.4060618
 1.38372528 1.38458939 1.36677129 1.37563589 1.36529414 1.37926028
 1.37955704 1.38978922 1.39481793 1.38882246 1.37672979 1.36218713
 1.34866067 1.33985801 1.33449301 1.32935948 1.32690426 1.32305325
 1.32189347 1.3189639  1.31817025 1.31643731 1.31669224 1.31542941
 1.3188911  1.31724704 1.32411332 1.32473622 1.3348261  1.34543679
 1.3512816  1.36740609 1.35652146 1.37557173 1.35798642 1.36783537
 1.35339595 1.35904702 1.34395477 1.36163897 1.33616533 1.34113189
 1.31856066 1.31761015 1.30569634 1.30772825 1.29979169 1.30392937
 1.30388559 1.31294976 1.31899469 1.33215762 1.32287894 1.33276329
 1.31411774 1.32140904 1.29655636 1.30521278 1.28906507 1.29854473
 1.29679222 1.29927874]
[0.2037 0.2054 0.2138 0.1835 0.2457 0.2478 0.2812 0.2813 0.2777 0.2871
 0.3032 0.3017 0.3343 0.3472 0.3552 0.3576 0.3654 0.3747 0.3794 0.3828
 0.3773 0.3842 0.3884 0.3982 0.4033 0.4074 0.4077 0.4129 0.4135 0.4186
 0.4207 0.4246 0.4267 0.4289 0.4315 0.4314 0.433  0.4323 0.4333 0.4312
 0.4317 0.4282 0.4298 0.4337 0.438  0.44   0.4459 0.4495 0.454  0.4545
 0.4607 0.4598 0.4629 0.4651 0.4656 0.467  0.4644 0.4639 0.4659 0.4626
 0.4644 0.462  0.4698 0.4689 0.4768 0.4753 0.4794 0.4777 0.4832 0.4818
 0.4862 0.4837 0.4873 0.4869 0.4898 0.4914 0.4913 0.4962 0.4933 0.4975
 0.4958 0.4999 0.4987 0.502  0.501  0.5042 0.5044 0.5058 0.5033 0.5025
 0.5013 0.4956 0.4965 0.4895 0.492  0.4886 0.4901 0.4834 0.4877 0.4829
 0.4885 0.4891 0.4987 0.5004 0.508  0.5089 0.5155 0.5141 0.5209 0.5153
 0.5209 0.5143 0.52   0.5126 0.5148 0.5078 0.5114 0.5076 0.5164 0.5132
 0.5249 0.5216 0.5314 0.5259 0.5335 0.5277 0.5367 0.5301 0.5363 0.5267
 0.5268 0.519  0.5116 0.5066 0.5009 0.5052 0.506  0.5147 0.5197 0.5254
 0.531  0.5317 0.5334 0.5279 0.526  0.5233 0.5214 0.5245 0.5293 0.5351
 0.5376 0.5402 0.5432 0.5466 0.5445 0.5501 0.5463 0.5515 0.5467 0.5528
 0.548  0.5553 0.5465 0.5537 0.5448 0.5504 0.5384 0.5402 0.5299 0.535
 0.5286 0.5319 0.5248 0.5333 0.5287 0.5344 0.5351 0.5386 0.5398 0.5428
 0.5476 0.5489 0.5537 0.5543 0.5571 0.5537 0.5563 0.551  0.5489 0.5452
 0.5444 0.5462 0.5475 0.5488 0.5539 0.5559 0.5554 0.5595 0.5552 0.5575]
```

[15]:

```
#def Plot_Train_Validation_Cost_Accurracy(self, Cost_Train, Cost_Validation,␣
 ↪Acc_Train, Acc_Validation):
def Plot_Train_Validation_Cost_Accurracy(Cost_Train, Cost_Validation,␣
 ↪Acc_Train, Acc_Validation, Eta):
        plt.subplot(1,3,1)
        plt.plot(Cost_Train, 'g-', label='Train')
        plt.plot(Cost_Validation, 'r-', label='Validation')
        plt.title('Cost Comparison')
        plt.xlabel('n_records')
        plt.ylabel('Cost')
        plt.legend()
        plt.grid('on')

        plt.subplot(1,3,2)
        plt.plot(Acc_Train, 'g-', label='Train')
        plt.plot(Acc_Validation, 'r-', label='Validation')
        plt.title('Accuracy Comparison')
        plt.xlabel('n_records')
        plt.ylabel('Accuracy')
        plt.legend()
        plt.grid('on')

        plt.subplot(1,3,3)
        plt.plot(Eta, 'g-', label='Eta')
        plt.title('Eta Change')
        plt.xlabel('Time')
        plt.ylabel('Eta')
        plt.legend()
        plt.grid('on')
```

[17]:

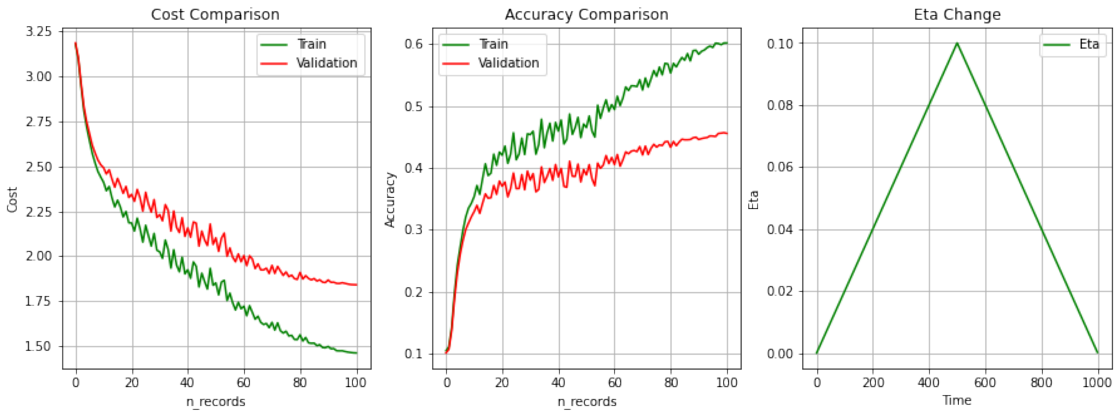
```
# this function uses the FULL training data to calculate the ACCURACY & COST
def Train_Cyclical(param_list, cost_calculation_method = 'Batch_aggregated'):
    # param_list = [network1, train_X_Norm, validation_X_Norm, batch_size,␣
 ↪n_cycles, lambda_cost,
    #               record_per_cycle, m, eta_min, eta_max, n_steps]
    '''
    Up until now, we have trained our networks with Vanilla mini-batch gradient␣
 ↪descent.
    To help speed up training times and avoid time-consuming searches for good␣
 ↪values of eta, we will now
    implement mini-batch-GD training where the learning rate at each update␣
 ↪step is defined in a cylical way
    check equations (14) and (15) in the assignment.
    '''
    network1 = param_list[0]
    train_X_Norm = param_list[1]        # 45.000 images
    validation_X_Norm = param_list[2]   # 5.000 images
    # the size of a mini-batch. in other words, number of images in 1 mini-batch
    batch_size = param_list[3]          # 100
    # number of triangles that you want to use to test different eta values
    n_cycles = param_list[4]            # 1
    # regularization coefficient (punishment)
    lambda_cost = param_list[5]         # 0.01
    # how many calculations do you want to see per cycle? = 10? 20? 30? to plot␣
 ↪the graph
    # record_per_cycle = 100  (+1 is to show the first calculation, as well)
    record_per_cycle = param_list[6]

    d = train_X_Norm.shape[0]           # dimension of X_train... 3072 = 32 x␣
 ↪32 x 3
    m = param_list[7]                   # number of nodes in the hidden layer...
 ↪ 50
    K = network1.train_Y.shape[0]       # number of classes... 10
    eta_min = param_list[8]             # learning rate (step-size)   i.e. 1e-5
    eta_max = param_list[9]             # learning rate (step-size)   i.e. 1e-1
    # n_s = k * np.floor(n/n_batch) by definition in the assignment
    # so, k corresponds to the number of epochs to complete a half cycle, for a␣
 ↪full-cycle,
    # we need to run the calculations 2*k times. i.e. if k=5, we need 10 epochs
    n_steps = param_list[10]            # number of steps to complete a half␣
 ↪cycle(triangle)

    # Generate W1, W2, b1, b2 matrices with initial values
    # (W1, W2, b1, b2) = network1.Initialize_W_b(d, m, K)
    sigma1 = 1 / int(np.sqrt(d))
    sigma2 = 1 / np.sqrt(m)
    #(W1, W2, b1, b2) = network1.Initialize_W_b(d, m, K, sigma1, sigma2)
    initial_sizes = [0, d, m, K]
    (W1, W2, b1, b2) = network1.Initialize_W_b(initial_sizes, sigma1, sigma2)

    W = [W1, W2]
    b = [b1, b2]

    linearLayer1 = layer.Linear()
    reluLayer = layer.ReLU()             # not an exact layer but operational step..
    linearLayer2 = layer.Linear()
    softmaxLayer = layer.Softmax()       # not an exact layer but operational step..
    ReLUlayer = layer.ReLU()
    grad = gradient.Gradient()

    layers = [linearLayer1, reluLayer, linearLayer2, softmaxLayer]
    # total_batch = how many mini-batches will we need to cover the entire training set?
    total_batch = int(np.floor(train_X_Norm.shape[1] / batch_size))

    # A full cycle once go up (from eta_min to eta_max) and once go down (from eta_max to eta_min)
    # so we multiply by 2
    #n_cycles = 1   # corresponds to "L array" in 2.L.ns in the assignment ..
    # L=the current cycle number ... n_cycles=the total number of cycles to be applied
    # so L is an element of n_cycle >> i.e. L = {0, 1, 2, 3} if n_cycles = 4
    # n_epocs=10 for 1 full cycle having n_steps=500 and batch_size=100
    # n_s = k * np.floor(n/n_batch) by definition in the assignment.. see the previous explanation above for n_steps
    n_epochs = int(2 * n_cycles * (n_steps / total_batch))
    #n_sanity_batch = 1
    # we will only use 100 images for sanity check that means there will be only 1 mini-batch
    #n_sanity_batch = 1
    #n_test_images = batch_size * n_sanity_batch

    n_records = total_batch * n_epochs         # number of batches will be used IN TOTAL for ALL calculations
    # i.e. 450 * 4 = 1800

    # how many points in total we will have in the COST graph
    cost_record = record_per_cycle * n_cycles + 1  # 100 * 1 + 1 = 101
    ###comparison_number = total_batch/cost_record
    # to be able to have that many points, at which batch we should run the␣
↪COST functions (e.g. in every 18 batch)
    comparison_number = int(n_records / (cost_record - 1))  # 1800/100 = 18

    eta_train = np.zeros(n_records)              # we will check if we properly␣
↪make a triangle

    J_epocs_train = np.zeros(cost_record)        # cost array to plot and see␣
↪how cost changes
    Accuracy_epocs_train = np.zeros(cost_record) # accuracy array
    J_epocs_validation = np.zeros(cost_record)
    Accuracy_epocs_validation = np.zeros(cost_record)


    #print('batch_size: {}, lambda_cost : {}, d: {}, m: {}, K: {}'.
↪format(batch_size, lambda_cost, d, m, K))
    ##print('\ntotal_batch: {}, n_steps : {}, n_epochs: {}, n_records: {},␣
↪cost_record: {}'.format(total_batch, n_steps, n_epochs, n_records,␣
↪cost_record))
    #print('\ntotal_batch: {}, n_steps : {}, n_epochs: {}, n_records: {}'.
↪format(total_batch, n_steps, n_epochs,
     #                                                                      ␣
↪ n_records))
    #print('\nrecord_per_cycle: {}, cost_record : {}, comparison_number: {}'.
↪format(record_per_cycle, cost_record,
      #                                                                     ␣
↪    comparison_number))

    # cycleID in use.. if n_cycles=0 then cycle max will be 0. else it will be␣
↪incremented by 1 at each 2*n_steps
    cycle_no = 0
    J_train_sum = 0
    n_records = 0       # this will basically show the number of batches␣
↪processed until now
    list_params = []
    smooth_cost = 0
    cost_record = 0
    # step number in the current cycle, it resets with a new cycle.
    cycle_step = 0   # just to see where we are in the current triangle

    start_time = datetime.datetime.now()
    for e in range(n_epochs):
        for batch in range(total_batch):
            index_list = list(range(batch * batch_size, (batch + 1) *␣
↪batch_size))
```

```
            # shuffling is not necessary but good to have
            np.random.shuffle(index_list)
            X_batch = train_X_Norm[:, index_list]
            Y_batch = network1.train_Y[:, index_list]

            P, H = network1.EvaluationClassifier(layers, X_batch, W, b)

            G2 = -np.subtract(network1.train_Y[:, index_list], P)
            N2 = Y_batch.shape[1]

            (grad_W2, grad_b2, G1) = grad.
↪ComputeGradients_Linear_HiddenLayer(N2, G2, H, lambda_cost, W2)
            G0 = reluLayer.Backward(G1, H)
            N1 = H.shape[1]

            (grad_W1, grad_b1) = grad.ComputeGradients_Linear_FirstLayer(N1,␣
↪G0, X_batch, lambda_cost, W1)

            # n_cycles = 1   # corresponds to "L array" in 2.L.ns in the␣
↪assignment ..
            # L=the current cycle number ... n_cycles=the total number of␣
↪cycles to be applied
            # so L is an element of n_cycle >> i.e. L = {0, 1, 2, 3} if␣
↪n_cycles = 4
            #if 2*cycle_no*n_steps <= t and t <= (2*cycle_no+1)*n_steps:
            # n_records corresponds to t in the formula
            if 2*cycle_no*n_steps <= n_records and n_records <=␣
↪(2*cycle_no+1)*n_steps:
                #eta = eta_min + (t - 2*cycle_no*n_steps)/
↪n_steps*(eta_max-eta_min)
                eta = eta_min + (n_records - 2*cycle_no*n_steps)/
↪n_steps*(eta_max-eta_min)
            #elif (2*cycle_no+1)*n_steps <= t and t <= 2*(cycle_no+1)*n_steps:
            elif (2*cycle_no+1)*n_steps <= n_records and n_records <=␣
↪2*(cycle_no+1)*n_steps:
                #eta = eta_max - (t - (2*cycle_no+1)*n_steps)/
↪n_steps*(eta_max-eta_min)
                eta = eta_max - (n_records - (2*cycle_no+1)*n_steps)/
↪n_steps*(eta_max-eta_min)

            eta_train[n_records] = eta

            # W1star, W2star
            W1 = W1 - eta * grad_W1
            W2 = W2 - eta * grad_W2
            # b1star, b2star
            bstar_m1 = b1 - eta * grad_b1
            b1 = bstar_m1[:, :1]  # there's a broadcast issue needs to be␣
↪fixed, that's why we pick only 1 column

            bstar_m2 = b2 - eta * grad_b2
            b2 = bstar_m2[:, :1]  # broadcast issue, this is a quick workaround␣
↪- use 1 column

            W = [W1, W2]
            b = [b1, b2]

            #if n_records % 10 == 0:
            if n_records == 0 or (n_records + 1) % comparison_number == 0:
                #print('\nepoch: {}, batch: {}, n_records: {}, cycle_step: {}'.
↪format(e, batch, n_records, cycle_step))

                if cost_calculation_method == 'ALL_Data':
                    J_train = network1.Cost(train_X_Norm, network1.train_Y, W,␣
↪b, lambda_cost)
                    J_epocs_train[cost_record] = J_train

                    P, H = network1.EvaluationClassifier(layers, train_X_Norm,␣
↪W, b)
                    k_train = np.argmax(P, axis=0)
                    A_train = network1.ComputeAccuracy(k_train, network1.
↪train_y)
                    Accuracy_epocs_train[cost_record] = A_train

                    J_validation = network1.Cost(validation_X_Norm , network1.
↪validation_Y, W, b, lambda_cost)
                    J_epocs_validation[cost_record] = J_validation

                    P_val, H_val = network1.EvaluationClassifier(layers,␣
↪validation_X_Norm, W, b)
                    k_validation = np.argmax(P_val, axis=0)
                    A_validation = network1.ComputeAccuracy(k_validation,␣
↪network1.validation_y)
                    Accuracy_epocs_validation[cost_record] = A_validation

                elif cost_calculation_method == 'Batch_aggregated':
                    J_train = network1.Cost(X_batch, Y_batch, W, b, lambda_cost)
                    J_train_sum += J_train
                    smooth_cost = J_train_sum/(cost_record + 1)
                    #J_epocs_train[n_records] = J_train
                    J_epocs_train[cost_record] = smooth_cost
                    ## If we use the ALL TRAINING data to calculate the␣
↪training accuracy:
                    # it feels like better to use that one
                    P, H = network1.EvaluationClassifier(layers, train_X_Norm,␣
↪W, b)
                    k_train = np.argmax(P, axis=0)
                    A_train = network1.ComputeAccuracy(k_train, network1.
↪train_y)
                    Accuracy_epocs_train[cost_record] = A_train

                    J_validation = network1.Cost(validation_X_Norm , network1.
↪validation_Y, W, b, lambda_cost)
                    J_epocs_validation[cost_record] = J_validation

                    P_val, H_val = network1.EvaluationClassifier(layers,␣
↪validation_X_Norm, W, b)
                    k_validation = np.argmax(P_val, axis=0)
                    A_validation = network1.ComputeAccuracy(k_validation,␣
↪network1.validation_y)
                    Accuracy_epocs_validation[cost_record] = A_validation

                elif cost_calculation_method == 'Batch_aggregated_ratio':
                    J_train = network1.Cost(X_batch, Y_batch, W, b, lambda_cost)
                    if n_records == 0:
                        smooth_cost = J_train
                    else:
                        #smooth_cost = 0.999*smooth_cost + 0.001 * J_train
                        smooth_cost = 0.99*smooth_cost + 0.01 * J_train
                    #J_epocs_train[n_records] = J_train
                    J_epocs_train[cost_record] = smooth_cost

                    ## If we use the ALL TRAINING data to calculate the␣
↪training accuracy:
                    # it feels like better to use that one
                    P, H = network1.EvaluationClassifier(layers, train_X_Norm,␣
↪W, b)
                    k_train = np.argmax(P, axis=0)
                    A_train = network1.ComputeAccuracy(k_train, network1.
↪train_y)
                    Accuracy_epocs_train[cost_record] = A_train

                    J_validation = network1.Cost(validation_X_Norm , network1.
↪validation_Y, W, b, lambda_cost)
                    J_epocs_validation[cost_record] = J_validation
                    P_val, H_val = network1.EvaluationClassifier(layers,␣
↪validation_X_Norm, W, b)
                    k_validation = np.argmax(P_val, axis=0)
                    A_validation = network1.ComputeAccuracy(k_validation,␣
↪network1.validation_y)
                    Accuracy_epocs_validation[cost_record] = A_validation

                cost_record += 1

                #print('cycle_no: {} ... eta: {} ... time: {}'.format(cycle_no,␣
↪eta, datetime.datetime.now()))
                #print('J_train: {} ... smooth_cost : {} ... J_validation: {}'.
↪format(J_train, smooth_cost, J_validation))


            n_records += 1
            cycle_step += 1

            if n_records % (2 * n_steps) == 0:
                cycle_no += 1
                cycle_step = 0

    end_time = datetime.datetime.now()

    #print("Calculation time of Train_Cyclical: " + str(end_time - start_time))
    #print("x-axis (steps) must be multiplied by 10 since the values recorded␣
↪once in every 10 Cycle")

    Plot_Train_Validation_Cost_Accurracy(J_epocs_train, J_epocs_validation,␣
↪Accuracy_epocs_train,
                                         Accuracy_epocs_validation, eta_train)
    #print('batch_size: {}, lambda_cost : {}, d: {}, m: {}, K: {}'.
↪format(batch_size, lambda_cost, d, m, K))
    #print('\ntotal_batch: {}, n_steps : {}, n_epochs: {}, n_records: {},␣
↪cost_record: {}'.format(total_batch, n_steps, n_epochs, n_records,␣
↪cost_record))
    #print('\ntotal_batch: {}, n_steps : {}, n_epochs: {}, n_records: {}'.
↪format(total_batch, n_steps,
     #                                                                      ␣
↪ n_epochs, n_records))
    #print('\nrecord_per_cycle: {}, cost_record : {}, comparison_number: {}'.
↪format(record_per_cycle,
      #                                                                     ␣
↪    cost_record, comparison_number))
    return layers, W, b, eta_train
```

```
```

[18]:

```
#### Exercise - 3 ###
# takes around 2.5 minutes
# TRAIN = VALIDATION = TEST = 10.000
# Train_Cyclical(network1, train_X_Norm, validation_X_Norm, n_cycles, n_steps)
# param_list = [network1, train_X_Norm, validation_X_Norm, batch_size,␣
 ↪n_cycles, lambda_cost, record_per_cycle, m,
#                 eta_min, eta_max, n_steps]
param_list = [network1, train_X_Norm, validation_X_Norm, 100, 1, 0.01, 100, 50,␣
 ↪1e-5, 1e-1, 500]
layers1, W1, b1, eta_train = Train_Cyclical(param_list, 'ALL_Data')
```



[19]:

```
P_test, H_test = network1.EvaluationClassifier(layers1, test_X_Norm, W1, b1)
k_test = np.argmax(P_test, axis=0)
A_test = network1.ComputeAccuracy(k_test, network1.test_y)
print(A_test)
```

```
0.454
```

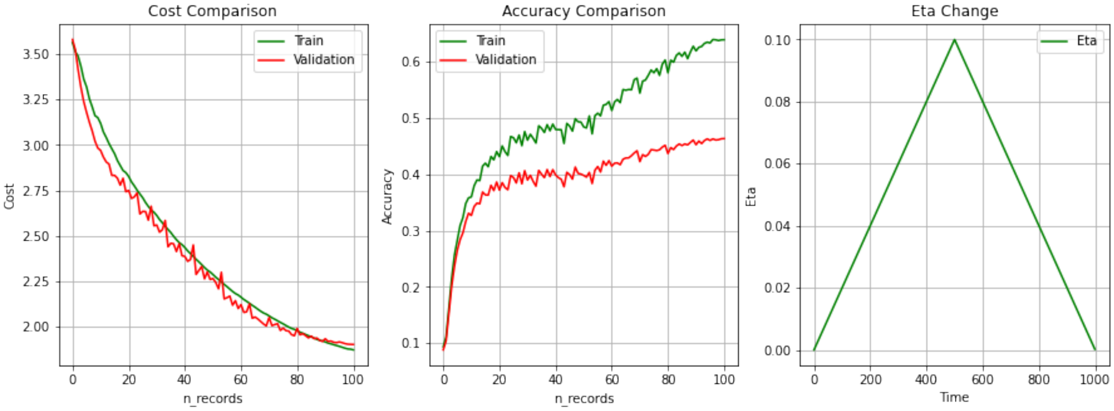
[20]:

```
print('Eta_Train\nmin: {}\tmax: {}\tmean: {}\tmedian: {}\nstd: {}\tvar: {}'.
 ↪format(min(eta_train),
        max(eta_train), np.mean(eta_train), np.median(eta_train), np.
 ↪std(eta_train), np.var(eta_train)))
```

```
Eta_Train
min: 1e-05      max: 0.1        mean: 0.050004999999999994      median:
0.050005000000000001
std: 0.02886474216641126        var: 0.0008331733403334001
```

[21]:

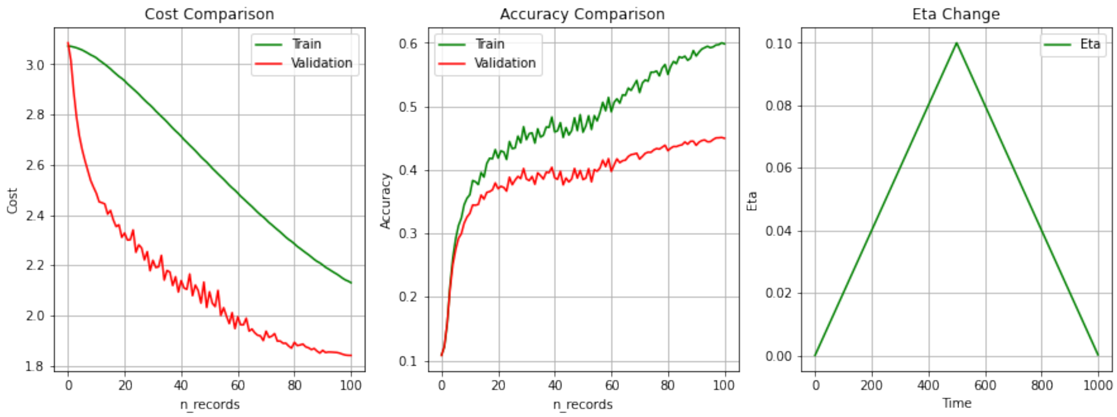
```
#### Exercise - 3 ###
# takes around 1.5 minutes
# TRAIN = VALIDATION = TEST = 10.000
# Train_Cyclical(network1, train_X_Norm, validation_X_Norm, n_cycles, n_steps)
# param_list = [network1, train_X_Norm, validation_X_Norm, batch_size,␣
 ↪n_cycles, lambda_cost, record_per_cycle, m,
#                 eta_min, eta_max, n_steps]
param_list = [network1, train_X_Norm, validation_X_Norm, 100, 1, 0.01, 100,␣
 ↪100, 1e-5, 1e-1, 500]
layers1, W1, b1, eta_train = Train_Cyclical(param_list, 'Batch_aggregated')
```



[22]:
```
P_test, H_test = network1.EvaluationClassifier(layers1, test_X_Norm, W1, b1)
k_test = np.argmax(P_test, axis=0)
A_test = network1.ComputeAccuracy(k_test, network1.test_y)
print(A_test)
```

```
0.4736
```

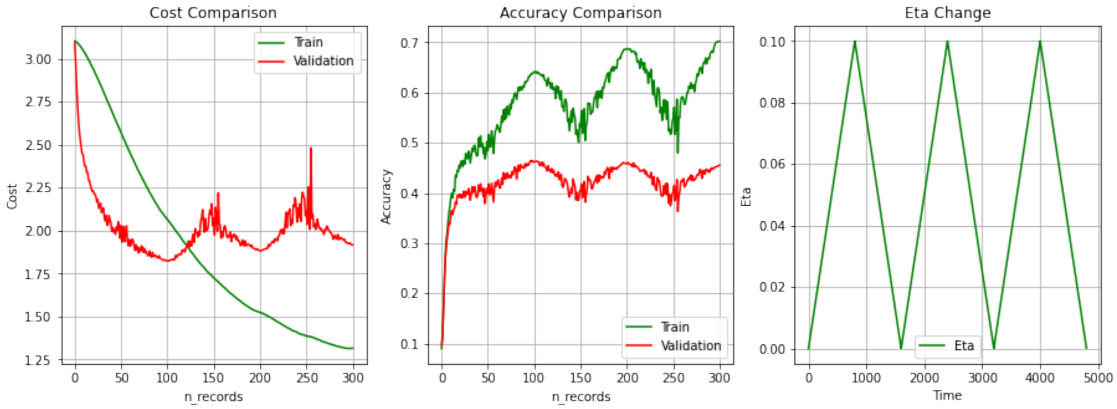
[23]:
```
#### Exercise - 3 ###
# takes around 1.5 minutes
# TRAIN = VALIDATION = TEST = 10.000
# Train_Cyclical(network1, train_X_Norm, validation_X_Norm, n_cycles, n_steps)
# param_list = [network1, train_X_Norm, validation_X_Norm, batch_size,␣
 ↪n_cycles, lambda_cost, record_per_cycle, m,
#                 eta_min, eta_max, n_steps]
param_list = [network1, train_X_Norm, validation_X_Norm, 100, 1, 0.01, 100, 50,␣
 ↪1e-5, 1e-1, 500]
layers1, W1, b1, eta_train = Train_Cyclical(param_list,␣
 ↪'Batch_aggregated_ratio')
```



```
[24]: P_test, H_test = network1.EvaluationClassifier(layers1, test_X_Norm, W1, b1)
      k_test = np.argmax(P_test, axis=0)
      A_test = network1.ComputeAccuracy(k_test, network1.test_y)
      print(A_test)
```

```
0.4574
```

```
[25]: # takes around 4.5 minutes
      # param_list = [network1, train_X_Norm, validation_X_Norm, batch_size,␣
       ↪n_cycles, lambda_cost, record_per_cycle, m,
      #                 eta_min, eta_max, n_steps]
      param_list = [network1, train_X_Norm, validation_X_Norm, 100, 3, 0.01, 100, 50,␣
       ↪1e-5, 1e-1, 800]
      layers2, W2, b2, eta_train = Train_Cyclical(param_list,␣
       ↪'Batch_aggregated_ratio')
```



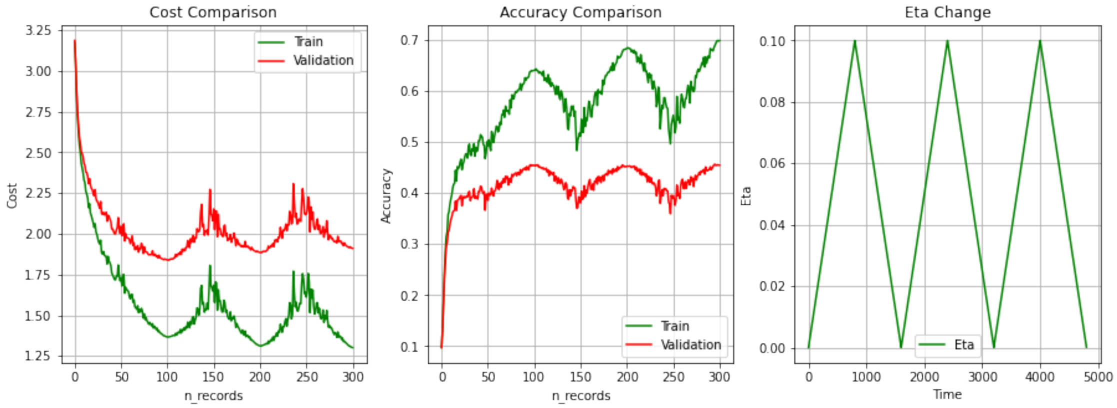
[26]:

```
P_test, H_test = network1.EvaluationClassifier(layers2, test_X_Norm, W2, b2)
k_test = np.argmax(P_test, axis=0)
A_test = network1.ComputeAccuracy(k_test, network1.test_y)
print(A_test)
```

```
0.4652
```

[27]:

```
# takes around 8.5 minutes
# param_list = [network1, train_X_Norm, validation_X_Norm, batch_size,␣
 ↪n_cycles, lambda_cost, record_per_cycle, m,
#                 eta_min, eta_max, n_steps]
param_list = [network1, train_X_Norm, validation_X_Norm, 100, 3, 0.01, 100, 50,␣
 ↪1e-5, 1e-1, 800]
layers2, W2, b2, eta_train = Train_Cyclical(param_list, 'ALL_Data')
```



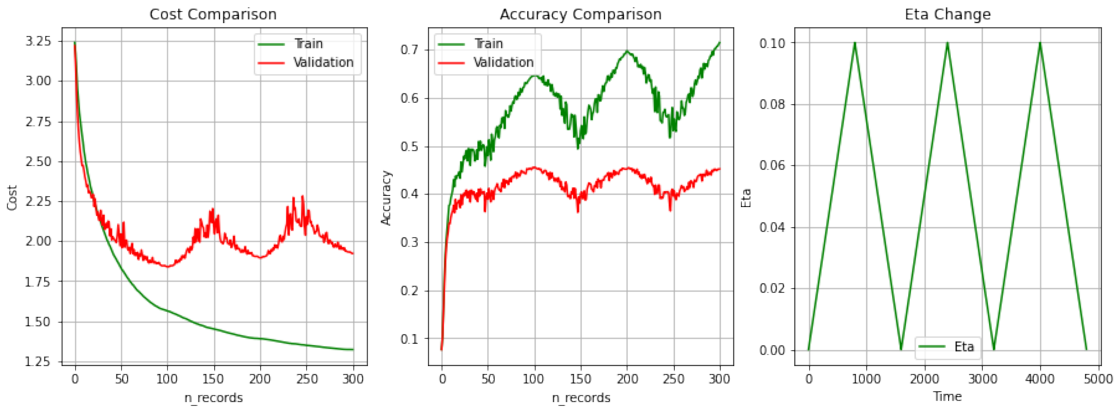
[28]:

```
P_test, H_test = network1.EvaluationClassifier(layers2, test_X_Norm, W2, b2)
k_test = np.argmax(P_test, axis=0)
A_test = network1.ComputeAccuracy(k_test, network1.test_y)
print(A_test)
```

```
0.4576
```

[29]:

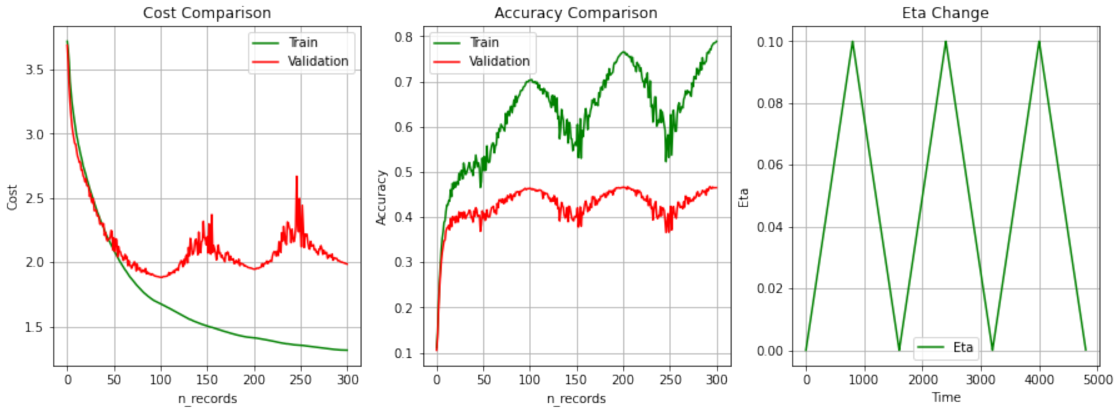
```
# takes around 4.5 minutes
# param_list = [network1, train_X_Norm, validation_X_Norm, batch_size,␣
 ↪n_cycles, lambda_cost, record_per_cycle, m,
#                 eta_min, eta_max, n_steps]
param_list = [network1, train_X_Norm, validation_X_Norm, 100, 3, 0.01, 100, 50,␣
 ↪1e-5, 1e-1, 800]
layers2, W2, b2, eta_train = Train_Cyclical(param_list, 'Batch_aggregated')
```



[30]:
```
P_test, H_test = network1.EvaluationClassifier(layers2, test_X_Norm, W2, b2)
k_test = np.argmax(P_test, axis=0)
A_test = network1.ComputeAccuracy(k_test, network1.test_y)
print(A_test)
```

```
0.4653
```

[31]:
```
# takes around 4 minutes 45 seconds
# param_list = [network1, train_X_Norm, validation_X_Norm, batch_size,␣
 ↪n_cycles, lambda_cost, record_per_cycle, m,
#                 eta_min, eta_max, n_steps]
param_list = [network1, train_X_Norm, validation_X_Norm, 100, 3, 0.01, 100,␣
 ↪100, 1e-5, 1e-1, 800]
layers2, W2, b2, eta_train = Train_Cyclical(param_list, 'Batch_aggregated')
```



[32]:
```
P_test, H_test = network1.EvaluationClassifier(layers2, test_X_Norm, W2, b2)
k_test = np.argmax(P_test, axis=0)
A_test = network1.ComputeAccuracy(k_test, network1.test_y)
```

```
print(A_test)
```

```
0.4781
```

DL_assignment2_melih_E4_v2.6_NO_PRINT

September 5, 2020

[1]:

```
import numpy as np
import pickle
import matplotlib.pyplot as plt
import scipy.io as sio
from sklearn import preprocessing

import gradient
import dataset
import computations
import layer
#from layer import Linear, Softmax, Gradient
import network

import datetime
import time
```

[2]:

```
np.random.seed(400)
np.seterr(over='raise');
plt.rcParams['figure.figsize'] = (15.0, 5.0)
```

[3]:

```
network1 = network.Network()
cifar = dataset.CIFAR_IMAGES()
#asgn1.test_batch_images(cifar_batch1)
```

[4]:

```
#### Exercise - 4 ###
# Read in the data & initialize the parameters of the network
# TRAIN = 45.000 - VALIDATION = 5.000  ... TEST = 10.000
filePathLocal_labels = 'Dataset/batches.meta'
#filePathLocal_data_TRAIN = 'Dataset/data_batch_1'
#filePathLocal_data_VALIDATION = 'Dataset/data_batch_2'
filePathLocal_data_ALL = ['Dataset/data_batch_1', 'Dataset/data_batch_2',␣
 ↪'Dataset/data_batch_3', 'Dataset/data_batch_4', 'Dataset/data_batch_5']
filePathLocal_data_TEST = 'Dataset/test_batch'

filePathList = (filePathLocal_data_ALL, filePathLocal_data_TEST)
```

```
# Read TRAIN, VALIDATION, TEST data into numpy arrays (numpy.ndarray) from␣
 ↪local files
network1.ReadData_Exercise4(cifar, filePathList)
# X = (d, N), Y = (K, N), y = (N,)   # N=number of total images in X
# X = (3072, 10000), Y = (10, 10000), y = (10000,)

# Find the MEAN and STD of trainX and broadcast them for matrix calculations
# trainX_Broadcast_MeanStd = (mean_train_X_broadcast, std_train_X_broadcast)
#trainX_Broadcast_MeanStd = network1.MeanStd_Train_X(network1.train_X)

# MEL
# Transform the INPUT to have zero mean ** Check that one if we need to␣
 ↪transfer all of them separately or
# only having the normalization as in here??
# Normalize all INPUT data by using MEAN and STD of TRAIN DATA
'''
train_X_Norm = network1.NormalizeData(network1.train_X,␣
 ↪trainX_Broadcast_MeanStd)
validation_X_Norm = network1.NormalizeData(network1.validation_X,␣
 ↪trainX_Broadcast_MeanStd)
test_X_Norm = network1.NormalizeData(network1.test_X, trainX_Broadcast_MeanStd)
'''

train_X_Norm = network1.NormalizeData_Broadcast(network1.train_X, network1.
 ↪train_X)
#validation_X_Norm = network1.NormalizeData_Broadcast(network1.validation_X,␣
 ↪network1.train_X)
validation_X_Norm = network1.NormalizeData_Per_DataSet(network1.validation_X)
#test_X_Norm = network1.NormalizeData_Broadcast(network1.test_X, network1.
 ↪train_X)
test_X_Norm = network1.NormalizeData_Per_DataSet(network1.test_X)

# mu = 0; d = network1.train_X.shape[0]; m = 50; K = network1.train_Y.shape[0]
# we will use only 20 of 3072 to have a dimension reduction in comparing␣
 ↪grad_analytic and grad_Numerical
#mu = 0; d = 20; m = 50; K = network1.train_Y.shape[0]
mu = 0; d = network1.train_X.shape[0]; m = 50; K = network1.train_Y.shape[0]

initial_sizes = (mu, d, m, K)

sigma1 = 1 / int(np.sqrt(d))
sigma2 = 1 / np.sqrt(m)

# Generate W1, W2, b1, b2 matrices with initial values
#(W1, W2, b1, b2) = network1.Initialize_W_b(d, m, K, sigma1, sigma2)
(W1, W2, b1, b2) = network1.Initialize_W_b(initial_sizes, sigma1, sigma2)
```

```
# W1 = (50, 3072), W2 = (10, 50), b1 = (50, 1), b2 = (10, 1) # if we use the␣
 ↪whole dimensions/features
# W1 = (50, 20), W2 = (10, 50), b1 = (50, 1), b2 = (10, 1) # if we use 20␣
 ↪dimensions/features
# W1 = (m, d), W2 = (K, m), b1 = (m, 1), b2 = (K, 1)
```

[5]:

```
'''
print("Mean-STD:\n␣
 ↪trainX_Broadcast_MeanStd[0]=\n{}\n\ntrainX_Broadcast_MeanStd[1]=\n{}".\
      format(trainX_Broadcast_MeanStd[0], trainX_Broadcast_MeanStd[1]))
print("\n\nMean-STD:\n train_X_Norm.mean(axis=1)={}\n\n".format(train_X_Norm.
 ↪mean(axis=1)))
'''
print("Shape check:\n train_X_Norm={}\t validation_X_Norm={}\t test_X_Norm={}".
 ↪format(train_X_Norm.shape, validation_X_Norm.shape, test_X_Norm.shape))
print(" train_Y={}\t\t validation_Y={}\t\t test_Y={}".format(network1.train_Y.
 ↪shape, network1.validation_Y.shape, network1.test_Y.shape))
print(" train_y={}\t\t validation_y={}\t\t\t test_y={}".format(network1.train_y.
 ↪shape, network1.validation_y.shape, network1.test_y.shape))
print(" W1={}\t\t\t W2={}\t\t\t\t b1={}\t\t\t b2={}".format(W1.shape, W2.shape,␣
 ↪b1.shape, b2.shape))
```

```
Shape check:
 train_X_Norm=(3072, 45000)      validation_X_Norm=(3072, 5000)
test_X_Norm=(3072, 10000)
 train_Y=(10, 45000)             validation_Y=(10, 5000)
test_Y=(10, 10000)
 train_y=(45000,)                validation_y=(5000,)
test_y=(10000,)
 W1=(50, 3072)                   W2=(10, 50)                             b1=(50,
1)                      b2=(10, 1)
```

[6]:

```
#def Plot_Train_Validation_Cost_Accurracy(self, Cost_Train, Cost_Validation,␣
 ↪Acc_Train, Acc_Validation):
def Plot_Train_Validation_Cost_Accurracy(Cost_Train, Cost_Validation,␣
 ↪Acc_Train, Acc_Validation, Eta):
        plt.subplot(1,3,1)
        plt.plot(Cost_Train, 'g-', label='Train')
        plt.plot(Cost_Validation, 'r-', label='Validation')
        plt.title('Cost Comparison')
        plt.xlabel('n_records')
        plt.ylabel('Cost')
        plt.legend()
        plt.grid('on')

        plt.subplot(1,3,2)
        plt.plot(Acc_Train, 'g-', label='Train')
        plt.plot(Acc_Validation, 'r-', label='Validation')
        plt.title('Accuracy Comparison')
        plt.xlabel('n_records')
        plt.ylabel('Accuracy')
        plt.legend()
        plt.grid('on')

        plt.subplot(1,3,3)
        plt.plot(Eta, 'g-', label='Eta')
        plt.title('Eta Change')
        plt.xlabel('Time')
        plt.ylabel('Eta')
        plt.legend()
        plt.grid('on')
```

[7]:

```
def Coarse_Search(l_min, l_max):
    l = l_min + (l_max - l_min) * np.random.uniform(0,1)
    lambda_coarse = pow(10, l)
    return lambda_coarse
```

[8]:

```
# this function uses the BATCH data with a SMOOTH_COST to calculate the␣
 ↪ACCURACY & COST
def Train_Cyclical_Coarse_lambda_smooth_aggregated_2(param_list):
    # param_list = [network1, train_X_Norm, validation_X_Norm, batch_size,␣
 ↪n_cycles, lambda_cost, record_per_cycle, m,
    #                 eta_min, eta_max]
    '''
    Up until now, we have trained our networks with Vanilla mini-batch gradient␣
 ↪descent.
    To help speed up training times and avoid time-consuming searches for good␣
 ↪values of eta, we will now
    implement mini-batch-GD training where the learning rate at each update␣
 ↪step is defined in a cylical way
    check equations (14) and (15) in the assignment.
    '''
    network1 = param_list[0]
    train_X_Norm = param_list[1]      # 45.000 images
    validation_X_Norm = param_list[2] # 5.000 images
    # the size of a mini-batch. in other words, number of images in 1 mini-batch
    batch_size = param_list[3]        # 100
    # number of triangles that you want to use to test different eta values
    n_cycles = param_list[4]         # 1
    # regularization coefficient (punishment)
    lambda_cost = param_list[5]      # 0.01
    # how many calculations do you want to see per cycle? = 10? 20? 30? to plot␣
↪the graph
    # record_per_cycle = 100  (+1 is to show the first calculation, as well)
    record_per_cycle = param_list[6]

    d = train_X_Norm.shape[0]       # dimension of X_train... 3072 = 32 x 32 x 3
    m = param_list[7]               # number of nodes in the hidden layer... 50
    K = network1.train_Y.shape[0]  # number of classes... 10
    eta_min = param_list[8] # learning rate (step-size)   i.e. 1e-5
    eta_max = param_list[9] # learning rate (step-size)   i.e. 1e-1

    # Generate W1, W2, b1, b2 matrices with initial values
    sigma1 = 1 / int(np.sqrt(d))    # std to initialize W1
    sigma2 = 1 / np.sqrt(m)         # std to initialize W2
    #(W1, W2, b1, b2) = network1.Initialize_W_b([mu, d, m, K], sigma1, sigma2)
    initial_sizes = [0, d, m, K]
    (W1, W2, b1, b2) = network1.Initialize_W_b(initial_sizes, sigma1, sigma2)

    W = [W1, W2]
    b = [b1, b2]

    linearLayer1 = layer.Linear()
    reluLayer = layer.ReLU()              # not an exact layer but operational␣
↪step..
    linearLayer2 = layer.Linear()
    softmaxLayer = layer.Softmax()        # not an exact layer but operational␣
↪step..
    ReLUlayer = layer.ReLU()
    grad = gradient.Gradient()

    layers = [linearLayer1, reluLayer, linearLayer2, softmaxLayer]
    # total_batch = how many mini-batches will we need to cover the entire␣
↪training set
    # we will use 45000 images for training = train_X_Norm.shape[1]
    # total_batch = 45000/100 = 450
    #total_batch = int(np.ceil(train_X_Norm.shape[1] / batch_size))
    total_batch = int(np.floor(train_X_Norm.shape[1] / batch_size))

    # A full cycle: once goes up (from eta_min to eta_max) and once goes down␣
↪(from eta_max to eta_min)
    # so we multiply by 2
    n_steps = 2 * total_batch  # 2 * 450 = 900      # this was given by␣
↪definition in the assignment
    # n_steps / total_batch = 2 is picked (before, this value was used as k in␣
↪exercise-3)
```

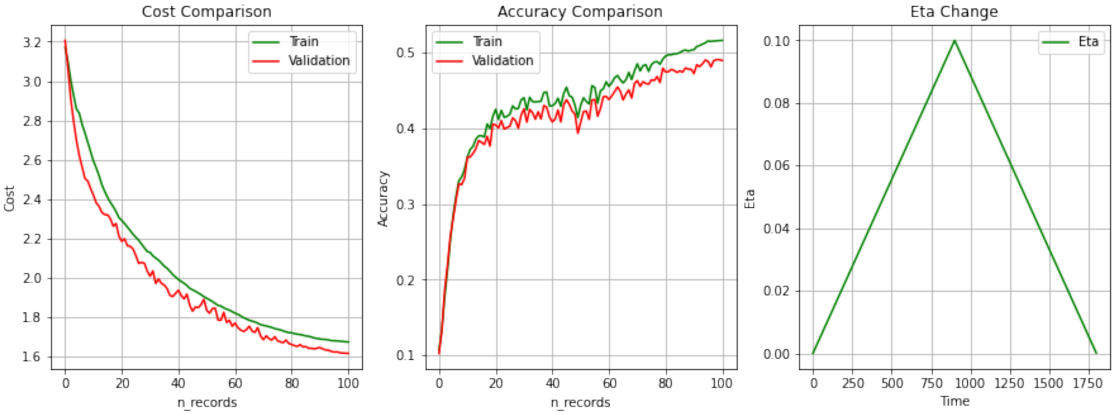
```
    # but in exercise-3, we were given the n_steps, now, n_steps is calculated␣
↪by the above equation and
    # k is given as 2 according to the equation
    ##### n_epochs = int(2 * n_cycles * (n_steps / total_batch))
    # k=2 is accepted as in this formula n_s = k[n/n_batch] ... 1 cycle= 2 * n_s
    # n_epochs = 2 * k * n_cycles = 2 * 2 * 1 = 4 in the first example
    n_epochs = 2 * 2 * n_cycles

    # !!!! CAUTION !!!! the usage of n_records and t is different than the␣
↪previous functions - Do not confuse!
    n_records = total_batch * n_epochs    # number of batches will be used IN␣
↪TOTAL for ALL calculations
    # i.e. 450 * 4 = 1800

    # how many points in total we will have in the COST graph
    cost_record = record_per_cycle * n_cycles + 1  # 100 * 1 + 1 = 101
    ###comparison_number = total_batch/cost_record
    # to be able to have that many points, at which batch we should run the␣
↪COST functions (e.g. in every 18 batch)
    comparison_number = int(n_records / (cost_record - 1))  # 1800/100 = 18

    eta_train = np.zeros(n_records)              # we will check if we properly␣
↪make a triangle

    J_epocs_train = np.zeros(cost_record)        # cost array to plot and see␣
↪how cost changes
    Accuracy_epocs_train = np.zeros(cost_record) # accuracy array
    J_epocs_validation = np.zeros(cost_record)
    Accuracy_epocs_validation = np.zeros(cost_record)

    #print('batch_size: {}, lambda_cost : {}, d: {}, m: {}, K: {}'.
↪format(batch_size, lambda_cost, d, m, K))
    ##print('\ntotal_batch: {}, n_steps : {}, n_epochs: {}, n_records: {},␣
↪cost_record: {}'.format(total_batch, n_steps, n_epochs, n_records,␣
↪cost_record))
    #print('\ntotal_batch: {}, n_steps : {}, n_epochs: {}, n_records: {}'.
↪format(total_batch, n_steps, n_epochs,
    #                                                                           ␣
↪ n_records))
    #print('\nrecord_per_cycle: {}, cost_record : {}, comparison_number: {}'.
↪format(record_per_cycle, cost_record,
     #                                                                          ␣
↪    comparison_number))

    # cycleID in use.. if n_cycles=0 then cycle max will be 0. else it will be␣
↪incremented by 1 at each 2*n_steps
```

```
    cycle_no = 0
    J_train_sum = 0
    n_records = 0       # this will basically show the number of batches␣
↪processed until now
    list_params = []
    smooth_cost = 0
    cost_record = 0
    # step number in the current cycle, it resets with a new cycle.
    cycle_step = 0   # just to see where we are in the current triangle

    start_time = datetime.datetime.now()
    for e in range(n_epochs):
        for batch in range(total_batch):
            index_list = list(range(batch * batch_size, (batch + 1) *␣
↪batch_size))
            # shuffling is not necessary but good to have
            np.random.shuffle(index_list)
            X_batch = train_X_Norm[:, index_list]
            Y_batch = network1.train_Y[:, index_list]

            P, H = network1.EvaluationClassifier(layers, X_batch, W, b)

            G2 = -np.subtract(Y_batch, P)  #
            N2 = Y_batch.shape[1]          #

            (grad_W2, grad_b2, G1) = grad.
↪ComputeGradients_Linear_HiddenLayer(N2, G2, H, lambda_cost, W2)
            G0 = reluLayer.Backward(G1, H)
            N1 = H.shape[1]                #

            (grad_W1, grad_b1) = grad.ComputeGradients_Linear_FirstLayer(N1,␣
↪G0, X_batch, lambda_cost, W1)

            # n_cycles = 1   # corresponds to "L array" in 2.L.ns in the␣
↪assignment ..
            # L=the current cycle number ... n_cycles=the total number of␣
↪cycles to be applied
            # so L is an element of n_cycle >> i.e. L = {0, 1, 2, 3} if␣
↪n_cycles = 4
            #if 2*cycle_no*n_steps <= t and t <= (2*cycle_no+1)*n_steps:
            # n_records corresponds to t in the formula
            if 2*cycle_no*n_steps <= n_records and n_records <=␣
↪(2*cycle_no+1)*n_steps:
                #eta = eta_min + (t - 2*cycle_no*n_steps)/
↪n_steps*(eta_max-eta_min)
```

```
                eta = eta_min + (n_records - 2*cycle_no*n_steps)/
↪n_steps*(eta_max-eta_min)
            #elif (2*cycle_no+1)*n_steps <= t and t <= 2*(cycle_no+1)*n_steps:
            elif (2*cycle_no+1)*n_steps <= n_records and n_records <=␣
↪2*(cycle_no+1)*n_steps:
                #eta = eta_max - (t - (2*cycle_no+1)*n_steps)/
↪n_steps*(eta_max-eta_min)
                eta = eta_max - (n_records - (2*cycle_no+1)*n_steps)/
↪n_steps*(eta_max-eta_min)

            eta_train[n_records] = eta

            # W1star, W2star
            W1 = W1 - eta * grad_W1
            W2 = W2 - eta * grad_W2

            # b1star, b2star
            bstar_m1 = b1 - eta * grad_b1
            b1 = bstar_m1[:, :1]  # there's a broadcast issue needs to be␣
↪fixed, that's why we pick only 1 column

            bstar_m2 = b2 - eta * grad_b2
            b2 = bstar_m2[:, :1]  # broadcast issue, this is a quick workaround␣
↪- use 1 column

            W = [W1, W2]
            b = [b1, b2]

            #if n_records % 10 == 0:
            if n_records == 0 or (n_records + 1) % comparison_number == 0:
                #print('\nepoch: {}, batch: {}, n_records: {}, cycle_step: {}'.
↪format(e, batch, n_records, cycle_step))

                J_train = network1.Cost(X_batch, Y_batch, W, b, lambda_cost)
                J_train_sum += J_train
                smooth_cost = J_train_sum/(cost_record + 1)
                #J_epocs_train[n_records] = J_train
                J_epocs_train[cost_record] = smooth_cost

                ## If we use the ALL TRAINING data to calculate the training␣
↪accuracy:
                # it feels like better to use that one
                P, H = network1.EvaluationClassifier(layers, train_X_Norm, W, b)
                k_train = np.argmax(P, axis=0)
                A_train = network1.ComputeAccuracy(k_train, network1.train_y)
                Accuracy_epocs_train[cost_record] = A_train
                J_validation = network1.Cost(validation_X_Norm , network1.
↪validation_Y, W, b, lambda_cost)
                J_epocs_validation[cost_record] = J_validation

                P_val, H_val = network1.EvaluationClassifier(layers,␣
↪validation_X_Norm, W, b)
                k_validation = np.argmax(P_val, axis=0)
                A_validation = network1.ComputeAccuracy(k_validation, network1.
↪validation_y)
                Accuracy_epocs_validation[cost_record] = A_validation

                cost_record += 1

                #print('cycle_no: {} ... eta: {} ... time: {}'.format(cycle_no,␣
↪eta, datetime.datetime.now()))
                #print('J_train: {} ... smooth_cost : {} ... J_validation: {}'.
↪format(J_train, smooth_cost, J_validation))
                #print('\neta: {}, cycle_step: {}'.format(eta, cycle_step))

            #if n_records == 0 or (n_records+1) % 100 == 0:
             #   print('\ncycle_no: {}, epoch: {}, batch: {}, n_records: {},␣
↪cycle_step: {}'.format(cycle_no, e, batch, n_records, cycle_step))        ␣
↪

            n_records += 1
            cycle_step += 1

            #if t % (2 * n_steps) == 0:
            if n_records % (2 * n_steps) == 0:
                cycle_no += 1
                cycle_step = 0

    end_time = datetime.datetime.now()

    #print("Calculation time of Train_Cyclical: " + str(end_time - start_time))
    #print("x-axis (steps) must be multiplied by 10 since the values recorded␣
↪once in every 10 Cycle")

    Plot_Train_Validation_Cost_Accurracy(J_epocs_train, J_epocs_validation,␣
↪Accuracy_epocs_train,
                                         Accuracy_epocs_validation, eta_train)
    #print('batch_size: {}, lambda_cost : {}, d: {}, m: {}, K: {}'.
↪format(batch_size, lambda_cost, d, m, K))
```

```
    #print('\ntotal_batch: {}, n_steps : {}, n_epochs: {}, n_records: {},␣
↪cost_record: {}'.format(total_batch, n_steps, n_epochs, n_records,␣
↪cost_record))
    #print('\ntotal_batch: {}, n_steps : {}, n_epochs: {}, n_records: {}'.
↪format(total_batch, n_steps,
     #                                                                  ␣
↪ n_epochs, n_records))
    #print('\nrecord_per_cycle: {}, cost_record : {}, comparison_number: {}'.
↪format(record_per_cycle,
     #                                                                  ␣
↪    cost_record, comparison_number))
    return layers, W, b, eta_train
```

[9]:
```
# In this function, we didn't calculate the cost for each batch but we␣
↪calculated it for 100 batches out of 1800
# in 1 cycle and this helped us to save a lot of time with a similar result␣
↪with the one I calculated J for
# each batch
# takes ONLY around 40 seconds!
# param_list = [network1, train_X_Norm, validation_X_Norm, batch_size,␣
↪n_cycles, lambda_cost, record_per_cycle, m,
#                eta_min, eta_max]
param_list = [network1, train_X_Norm, validation_X_Norm, 100, 1, 0.01, 100, 50,␣
↪1e-5, 1e-1]
layers, W, b, eta_train =␣
↪Train_Cyclical_Coarse_lambda_smooth_aggregated_2(param_list)
```



[10]:
```
print('Eta_Train\nmin: {}\tmax: {}\tmean: {}\tmedian: {}\nstd: {}\tvar: {}'.
↪format(min(eta_train),
        max(eta_train), np.mean(eta_train), np.median(eta_train), np.
↪std(eta_train), np.var(eta_train)))
```

```
Eta_Train
min: 1e-05      max: 0.1        mean: 0.050005  median: 0.050005000000000001
std: 0.028864662343454958       var: 0.0008331687322016667
```

[11]:

```
P_test, H_test = network1.EvaluationClassifier(layers, test_X_Norm, W, b)
k_test = np.argmax(P_test, axis=0)
A_test = network1.ComputeAccuracy(k_test, network1.test_y)
print(A_test)
```

```
0.4924
```

[12]:

```
# takes around 5.5 minutes!! >> LIMIT the J (Cost) calculation per cycle instead
# of calculating J per batch
# this one uses the formula given in the assignment for the lambda_coarse values
lambda_coarse_list = []
#n_cycles = 1
#record_per_cycle = 100
start_time = datetime.datetime.now()
for i in range(8):
    lambda_coarse = Coarse_Search(-5, -1)
    # param_list = [network1, train_X_Norm, validation_X_Norm, batch_size,␣
 ↪n_cycles, lambda_cost, record_per_cycle, m,
    #               eta_min, eta_max]
    param_list = [network1, train_X_Norm, validation_X_Norm, 100, 1,␣
 ↪lambda_coarse, 100, 50, 1e-5, 1e-1]
    layers, W, b, eta_train =␣
 ↪Train_Cyclical_Coarse_lambda_smooth_aggregated_2(param_list)

    P_test, H_test = network1.EvaluationClassifier(layers, test_X_Norm, W, b)
    k_test = np.argmax(P_test, axis=0)
    A_test = network1.ComputeAccuracy(k_test, network1.test_y)
    #lambda_coarse_list.append([lambda_coarse, A_test, W, b])
    lambda_coarse_list.append([lambda_coarse, A_test])
    #print(datetime.datetime.now())
    #print('Test accuracy of {}. lambda ({})={}'.format(i, lambda_coarse,␣
 ↪A_test))

end_time = datetime.datetime.now()
print("Calculation time of lambda_coarse: " + str(end_time - start_time))
```

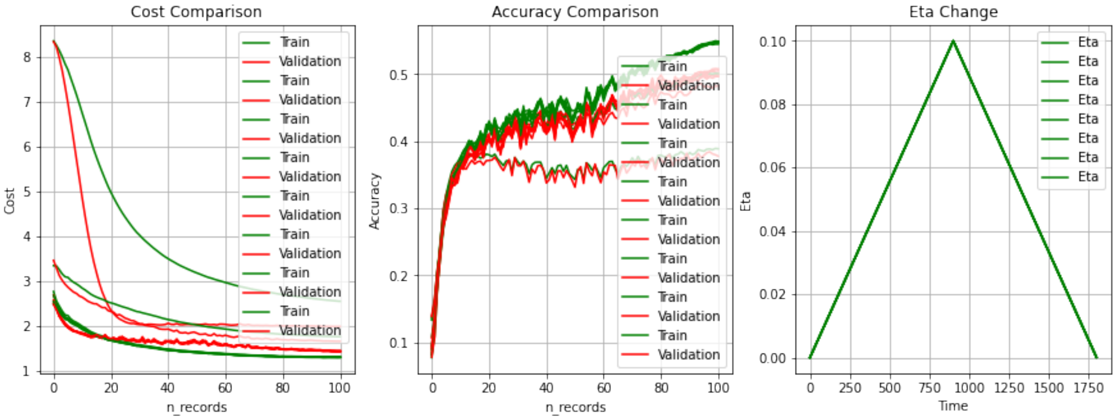
```
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
```

```
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
```

```
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
```

```
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
```

```
Calculation time of lambda_coarse: 0:05:25.496845
```



[13]:

```
# [lambda, test_accuracy]
lambda_coarse_list
```

[13]:

```
[[3.4654364956587273e-05, 0.4963],
 [0.09769685956289327, 0.3884],
 [0.0001955309850790619, 0.5001],
 [9.526945539447682e-05, 0.5024],
 [3.3350075921078503e-05, 0.5017],
 [0.014539373481118277, 0.487],
 [0.00017842001970348124, 0.4965],
 [6.973282303589172e-05, 0.4973]]
```

[14]:

```
import fileinput

def Time_Stamp():
    date_time = datetime.datetime.now()

    D = str(date_time.day)
    M = str(date_time.month)
    Y = str(date_time.year)

    h = str(date_time.hour)
    m = str(date_time.minute)
    s = str(date_time.second)

    date_array = [D, M, Y, h, m, s]

    return date_array

def FileNameUnique(prefix = "Lambda_"):
    file_name = prefix
    date_array = Time_Stamp()

    for idx, i in enumerate(date_array):
        if idx == 2:
            file_name += i + '_'
        elif idx == 5:
            file_name += i + '.txt'
        else:
            file_name += i + '.'

    #print(file_name)
    return file_name

def Save_Lambda_TestAccuracy(lambda_coarse_list):
    file_name = FileNameUnique()

    with open(file_name, "w") as output:
            for lambda_c in lambda_coarse_list:
                lambda_c_save = str(lambda_c[0]) + "," + str(lambda_c[1])
                #output.write(str(movie) + "\n")
                output.write(lambda_c_save + "\n")
```

[15]:

```
# Search of LAMBDA for a narrower range
# takes around 12 minutes
# HERE we NARROWED the range >> (-5, -1) to (-5, -4) and used 16 samples␣
 ↪instead of 8
lambda_coarse_list = []
start_time = datetime.datetime.now()
for i in range(16):
    lambda_coarse = Coarse_Search(-5, -4)
    # param_list = [network1, train_X_Norm, validation_X_Norm, batch_size,␣
 ↪n_cycles, lambda_cost, record_per_cycle, m,
    #               eta_min, eta_max]
    param_list = [network1, train_X_Norm, validation_X_Norm, 100, 1,␣
 ↪lambda_coarse, 100, 50, 1e-5, 1e-1]
    layers, W, b, eta_train =␣
 ↪Train_Cyclical_Coarse_lambda_smooth_aggregated_2(param_list)

    P_test, H_test = network1.EvaluationClassifier(layers, test_X_Norm, W, b)
    k_test = np.argmax(P_test, axis=0)
    A_test = network1.ComputeAccuracy(k_test, network1.test_y)
    #lambda_coarse_list.append([lambda_coarse, A_test, W, b])
    lambda_coarse_list.append([lambda_coarse, A_test])
    #print(datetime.datetime.now())
    #print('Test accuracy of {}. lambda ({})={}'.format(i, lambda_coarse,␣
 ↪A_test))
```

```python
Save_Lambda_TestAccuracy(lambda_coarse_list)
end_time = datetime.datetime.now()
print("Calculation time of lambda_coarse: " + str(end_time - start_time))
```

```
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
```

```
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
```

```
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
```

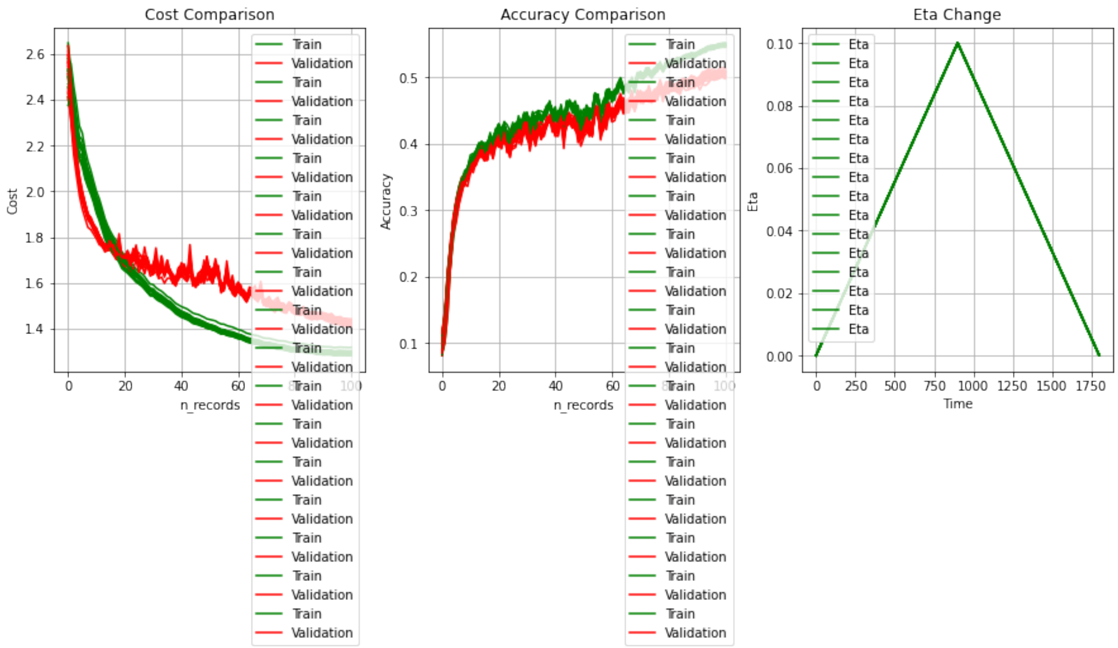
```
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
```

```
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
```

```
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
```

```
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
```

```
Calculation time of lambda_coarse: 0:10:57.812096
```



[16]:

```
lambda_coarse_list
```

[16]:

```
[[2.3623509445618744e-05, 0.4978],
 [4.412783084739363e-05, 0.4983],
 [4.560435734959645e-05, 0.5021],
 [1.1262450346489812e-05, 0.5007],
 [9.818671394260568e-05, 0.4966],
 [1.5749133438880033e-05, 0.4995],
 [1.3164243736202251e-05, 0.5054],
 [6.0485879260023e-05, 0.4998],
 [9.068077588056084e-05, 0.5012],
 [6.869406073885309e-05, 0.4931],
 [2.1980450649925166e-05, 0.5011],
 [2.6665029322969088e-05, 0.4987],
 [2.760969250853863e-05, 0.5011],
 [2.32252187405257e-05, 0.501],
 [1.0561798881093307e-05, 0.4993],
 [4.359936046517301e-05, 0.4988]]
```

[17]:

```
# Search of LAMBDA for a narrower range
# takes around 12 minutes
# HERE we NARROWED the range >> (-5, -1) to (-5, -4) and used 16 samples␣
 ↪instead of 8
lambda_coarse_list = []
start_time = datetime.datetime.now()
for i in range(16):
    lambda_coarse = Coarse_Search(-5, -3)
    # param_list = [network1, train_X_Norm, validation_X_Norm, batch_size,␣
↪n_cycles, lambda_cost, record_per_cycle, m,
    #               eta_min, eta_max]
    param_list = [network1, train_X_Norm, validation_X_Norm, 100, 1,␣
↪lambda_coarse, 100, 50, 1e-5, 1e-1]
    layers, W, b, eta_train =␣
↪Train_Cyclical_Coarse_lambda_smooth_aggregated_2(param_list)

    P_test, H_test = network1.EvaluationClassifier(layers, test_X_Norm, W, b)
    k_test = np.argmax(P_test, axis=0)
    A_test = network1.ComputeAccuracy(k_test, network1.test_y)
    #lambda_coarse_list.append([lambda_coarse, A_test, W, b])
    lambda_coarse_list.append([lambda_coarse, A_test])
    #print(datetime.datetime.now())
    #print('Test accuracy of {}. lambda ({})={}'.format(i, lambda_coarse,␣
↪A_test))

Save_Lambda_TestAccuracy(lambda_coarse_list)
end_time = datetime.datetime.now()
print("Calculation time of lambda_coarse: " + str(end_time - start_time))
```

```
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
```

```
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
```

```
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
```

```
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
```

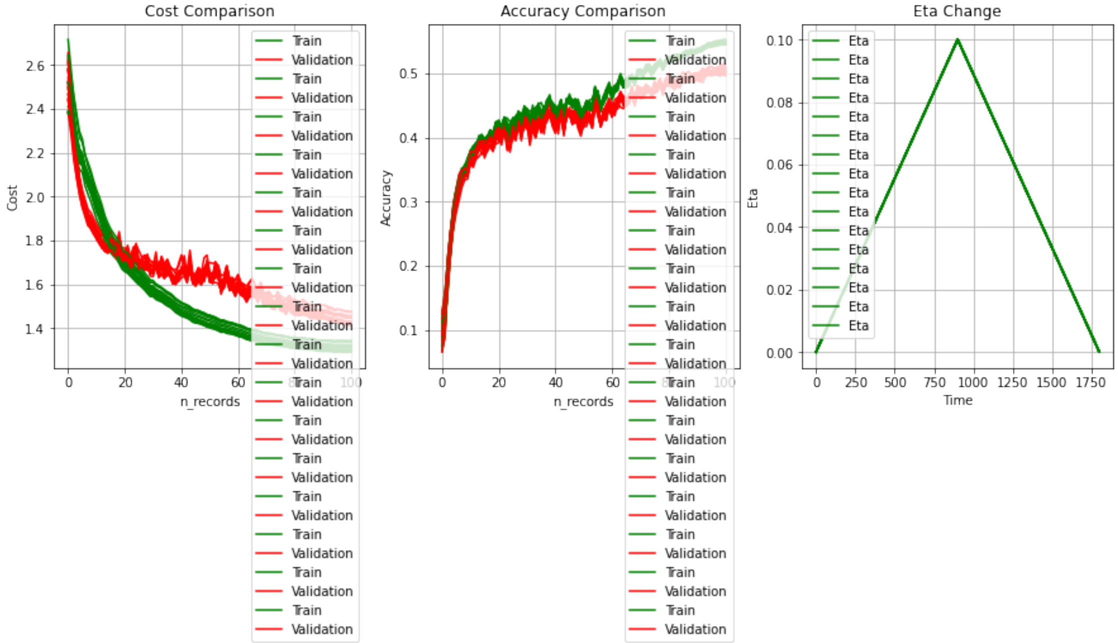
```
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
```

```
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
```

```
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
  if sys.path[0] == '':
/Users/melih/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:21:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
```

```
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
```

```
Calculation time of lambda_coarse: 0:10:48.951884
```

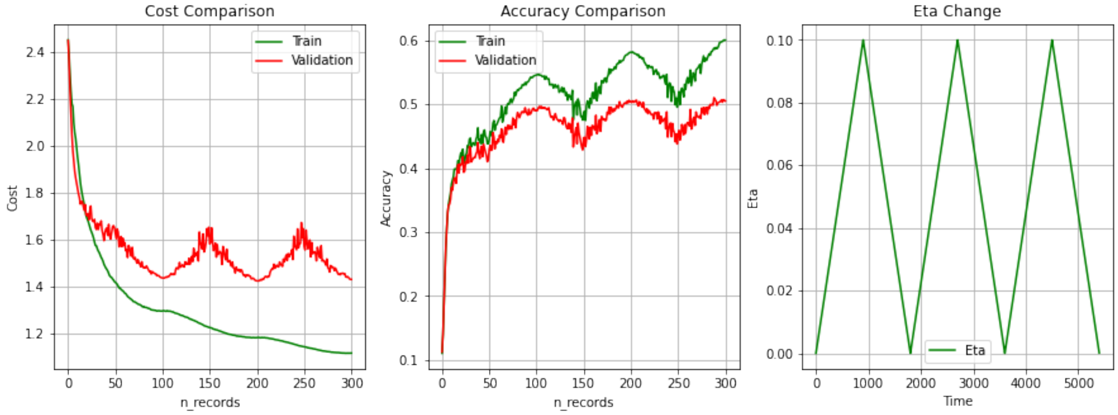


[18]:
```
lambda_coarse_list
```

[18]:
```
[[0.00032933566659033414, 0.4969],
 [0.0008484962871457188, 0.5018],
 [1.7499162934356765e-05, 0.502],
 [1.608819021269823e-05, 0.4984],
 [5.1366343447880185e-05, 0.5017],
 [0.0003867206597413132, 0.4976],
 [0.0006962310857509251, 0.4987],
 [0.0002612298487195736, 0.4954],
 [6.503713689980735e-05, 0.4973],
 [0.0004215372047767062, 0.4988],
 [1.8161231262606217e-05, 0.4991],
 [0.0003026456374925449, 0.4989],
 [5.260979986414023e-05, 0.4971],
 [0.00046666178523047065, 0.4978],
 [1.500921867517104e-05, 0.5009],
 [0.00010091058853135455, 0.495]]
```

[19]:
```
# In this functions, I didn't calculate the cost for each batch but I␣
↪calculated it for 100 batches out of 1800 in 1 cycle
```

```
# and this helped me to save a lot of time with a similar result with the one I␣
 ↪calculated J for each batch
# takes ONLY 2 minutes 12 seconds!
lambda_coarse = 3.544673306817798e-05  #1.8544671883635666e-05
# N_CYCLEs changed >> from 1 to 3
# param_list = [network1, train_X_Norm, validation_X_Norm, batch_size,␣
 ↪n_cycles, lambda_cost, record_per_cycle, m, eta_min, eta_max]

param_list = [network1, train_X_Norm, validation_X_Norm, 100, 3, lambda_coarse,␣
 ↪100, 50, 1e-5, 1e-1]
layers, W, b, eta_train =␣
 ↪Train_Cyclical_Coarse_lambda_smooth_aggregated_2(param_list)
```

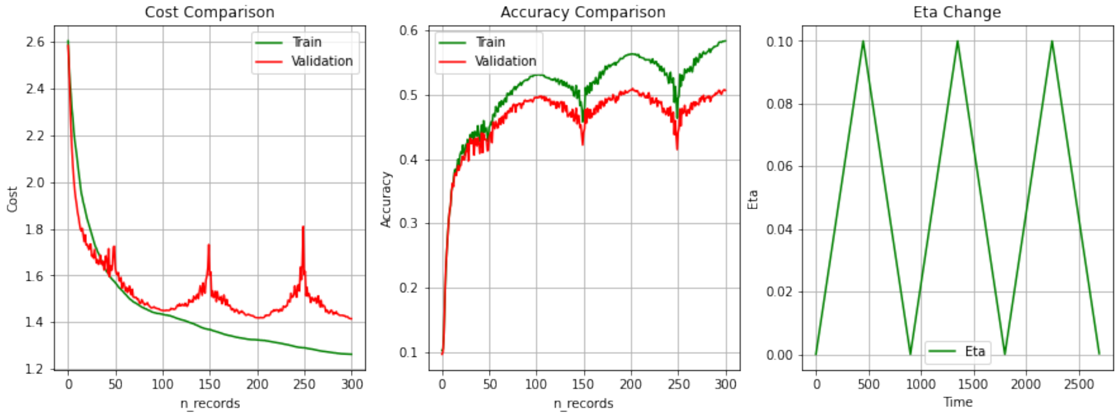


[20]:
```
P_test, H_test = network1.EvaluationClassifier(layers, test_X_Norm, W, b)
k_test = np.argmax(P_test, axis=0)
A_test = network1.ComputeAccuracy(k_test, network1.test_y)
print(A_test)
```

```
0.5004
```

[21]:
```
# In this functions, we didn't calculate the cost for each batch but
# we calculated it for 100 batches out of 1800 in 1 cycle
# and this helped me to save a lot of time with a similar result with the one I␣
 ↪calculated J for each batch
# takes 2 minutes 3 seconds!
lambda_coarse = 3.544673306817798e-05  #1.8544671883635666e-05
# N_CYCLEs changed >> from 1 to 3
# param_list = [network1, train_X_Norm, validation_X_Norm, batch_size,␣
 ↪n_cycles, lambda_cost, record_per_cycle, m,
#               eta_min, eta_max]
```

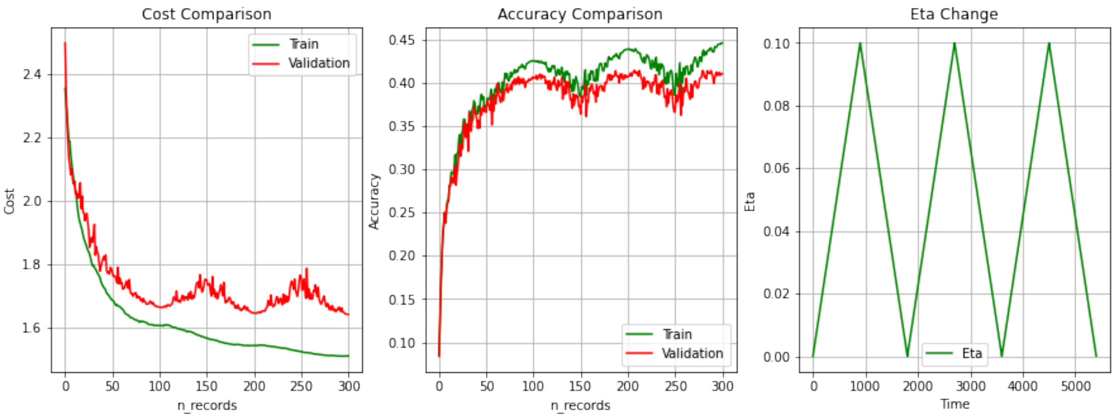
```
param_list = [network1, train_X_Norm, validation_X_Norm, 200, 3, lambda_coarse,␣
 ↪100, 50, 1e-5, 1e-1]
layers, W, b, eta_train =␣
 ↪Train_Cyclical_Coarse_lambda_smooth_aggregated_2(param_list)
```



[22]:
```
P_test, H_test = network1.EvaluationClassifier(layers, test_X_Norm, W, b)
k_test = np.argmax(P_test, axis=0)
A_test = network1.ComputeAccuracy(k_test, network1.test_y)
print(A_test)
```

```
0.5092
```

[23]:
```
# In this function, we didn't calculate the cost for each batch but
# we calculated it for 100 batches out of 1800 in 1 cycle
# and this helped me to save a lot of time with a similar result with the one I␣
 ↪calculated J for each batch
# takes 1 minute 34 seconds!  >> HERE we DECREASED the number of HIDDEN NODES␣
 ↪from 50 to 10
lambda_coarse = 3.544673306817798e-05  #1.8544671883635666e-05
# N_CYCLEs changed >> from 1 to 3
# param_list = [network1, train_X_Norm, validation_X_Norm, batch_size,␣
 ↪n_cycles, lambda_cost, record_per_cycle, m,
#                eta_min, eta_max]

param_list = [network1, train_X_Norm, validation_X_Norm, 100, 3, lambda_coarse,␣
 ↪100, 10, 1e-5, 1e-1]
layers, W, b, eta_train =␣
 ↪Train_Cyclical_Coarse_lambda_smooth_aggregated_2(param_list)
```



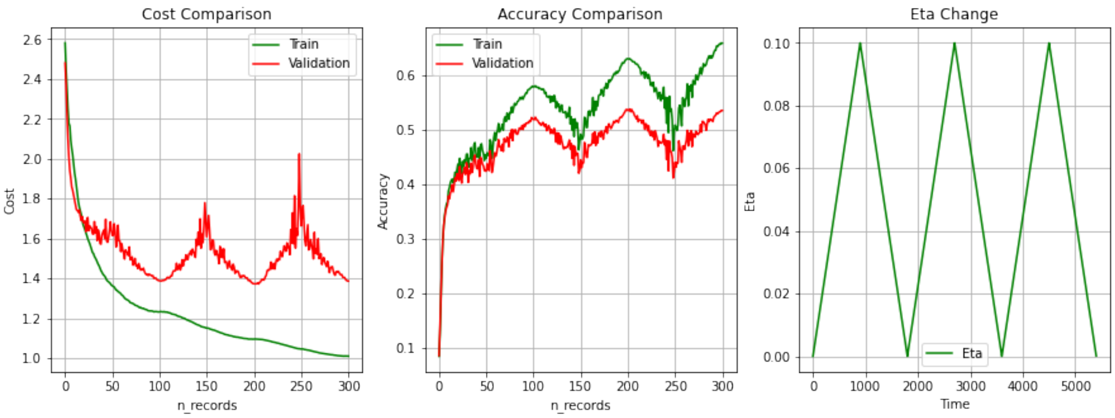
[24]:

```
P_test, H_test = network1.EvaluationClassifier(layers, test_X_Norm, W, b)
k_test = np.argmax(P_test, axis=0)
A_test = network1.ComputeAccuracy(k_test, network1.test_y)
print(A_test)
```

```
0.4122
```

[25]:

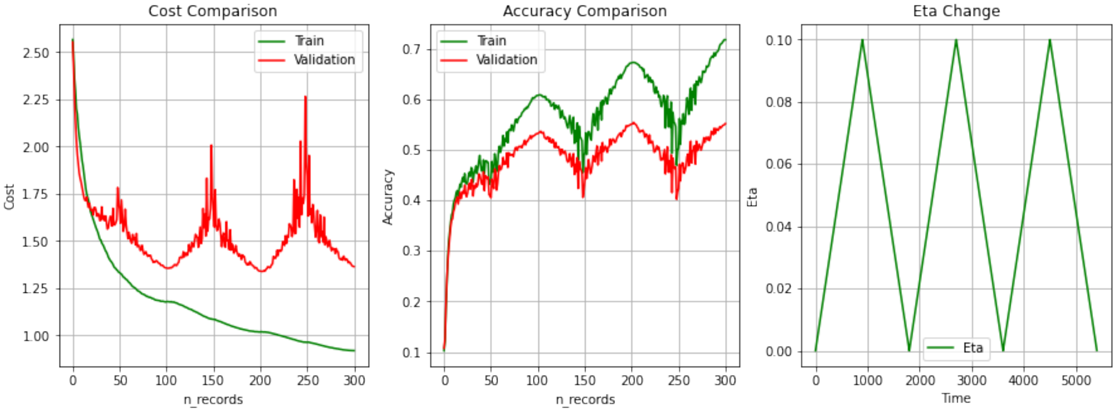
```
# In this function, we didn't calculate the cost for each batch but
# we calculated it for 100 batches out of 1800 in 1 cycle
# and this helped me to save a lot of time with a similar result with the one I␣
↪calculated J for each batch
# takes arund 3 minutes! >> HERE we INCREASED the number of HIDDEN NODES from␣
↪50 to 100
lambda_coarse = 3.544673306817798e-05   #1.8544671883635666e-05
# N_CYCLEs changed >> from 1 to 3
# param_list = [network1, train_X_Norm, validation_X_Norm, batch_size,␣
↪n_cycles, lambda_cost, record_per_cycle, m,
#                eta_min, eta_max]

param_list = [network1, train_X_Norm, validation_X_Norm, 100, 3, lambda_coarse,␣
↪100, 100, 1e-5, 1e-1]
layers, W, b, eta_train =␣
↪Train_Cyclical_Coarse_lambda_smooth_aggregated_2(param_list)
```



[26]:
```
P_test, H_test = network1.EvaluationClassifier(layers, test_X_Norm, W, b)
k_test = np.argmax(P_test, axis=0)
A_test = network1.ComputeAccuracy(k_test, network1.test_y)
print(A_test)
```

```
0.5201
```

[27]:
```
# In this function, we didn't calculate the cost for each batch but
# we calculated it for 100 batches out of 1800 in 1 cycle
# and this helped me to save a lot of time with a similar result with the one I␣
 ↪calculated J for each batch
# takes arund 3 minutes! >> HERE we INCREASED the number of HIDDEN NODES from␣
 ↪50 to 100
lambda_coarse = 3.544673306817798e-05  #1.8544671883635666e-05
# N_CYCLEs changed >> from 1 to 3
# param_list = [network1, train_X_Norm, validation_X_Norm, batch_size,␣
 ↪n_cycles, lambda_cost, record_per_cycle, m,
#                 eta_min, eta_max]

param_list = [network1, train_X_Norm, validation_X_Norm, 100, 3, lambda_coarse,␣
 ↪100, 200, 1e-5, 1e-1]
layers, W, b, eta_train =␣
 ↪Train_Cyclical_Coarse_lambda_smooth_aggregated_2(param_list)
```



[28]:

```
P_test, H_test = network1.EvaluationClassifier(layers, test_X_Norm, W, b)
k_test = np.argmax(P_test, axis=0)
A_test = network1.ComputeAccuracy(k_test, network1.test_y)
print(A_test)
```

```
0.5343
```

```python
# This is a .py file with the name "dataset.py"
import numpy as np
import pickle
import matplotlib.pyplot as plt
import scipy.io as sio
from sklearn import preprocessing

class CIFAR_IMAGES:
    def __init__(self):
        self.label_size = 10
        self.image_size = 32 * 32 * 3
        #for one_hot_encoding
        self.label_encoder = preprocessing.LabelBinarizer()
        # file path of the images on your laptop
        self.filePath = 'Dataset/data_batch_1'
        self.unique_labels = []

    # NOT USED
    def load_labels(self, filePathLocal, keyToRead=b'label_names'):
        self.filePath = filePathLocal
        with open(filePathLocal, 'rb') as fileToOpen:
            dictUniqueLabels = pickle.load(fileToOpen, encoding='bytes')
            #dictUniqueLabels_keys([b'num_cases_per_batch', b'label_names', b'num_vis'])
            #print(dictUniqueLabels.get(b'label_names'))
            #print(type(dictUniqueLabels[b'label_names'][0]))
            #<class 'bytes'>
            #[b'airplane', b'automobile', b'bird', b'cat', b'deer', b'dog', b'frog', b'horse', b'ship', b'truck']
            # labels = class names of each image. we convert the values from bytes to string
            self.unique_labels = [u_lbl.decode('ascii') for u_lbl in dictUniqueLabels[keyToRead]]
            #print(unique_labels)
            #['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
            #print(type(self.unique_labels[0]))
            #<class 'str'>

    # NOT USED
    def load_batch(self, filePathLocal):
        # requires "import pickle"
        # filePathLocal=filepath for the the required batch of images
        with open(filePathLocal, 'rb') as fileToOpen:
            dictBatch = pickle.load(fileToOpen, encoding='bytes')
            # dictBatch.keys()
            # dictBatch_keys([b'batch_label', b'labels', b'data', b'filenames'])
        return dictBatch

    # below function corresponds to [X, Y, y] = LoadBatch(filename) in the assignment, 1st task.
# This is a .py file with the name "network.py"
import numpy as np
import matplotlib.pyplot as plt
import layer
from layer import Linear, Softmax

class Network:
    def __init__(self):
        self.filePathLocal_labels = 'Dataset/batches.meta'
        self.filePathLocal_batch = 'Dataset/data_batch_1'
        self.filePathLocal_data_TRAIN = 'Dataset/data_batch_1'
        self.filePathLocal_data_VALIDATION = 'Dataset/data_batch_2'
        self.filePathLocal_data_TEST = 'Dataset/test_batch'
        self.K = 10                        # number of labels/classes
        self.d = 3072                      # number of dimensions of an image 32x32x3
        self.mu, self.sigma = 0, 0.01      # mean and standard deviation
        self.batch_length = 100            # n=100 samples picked as a subset ... N=10000=number of images in the training set
        self.lambda_cost = 0
        self.h = 1e-6
        self.eps = 1e-6

    # filePathList[0] = filePathLocal_data_TRAIN, filePathList[1] = filePathLocal_data_VALIDATION
    # consindering that all TRAIN, VALIDATION and TEST data are in different files
    def ReadData(self, cifar, filePathList):
        # Top-level: Read in and store the training, validation and test data.
        # cifar_batch = CIFAR_IMAGES()
        [self.train_X, self.train_Y, self.train_y] = cifar.load_batch_a1(filePathList[0])
        [self.validation_X, self.validation_Y, self.validation_y] = cifar.load_batch_a1(filePathList[1])
        [self.test_X, self.test_Y, self.test_y] = cifar.load_batch_a1(filePathList[2])

    def ReadData_Exercise4(self, cifar, filePathList):
        # Top-level: Read in and store the training, validation and test data.
        # cifar_batch = CIFAR_IMAGES()
        # 'filePathList[0] = ['Dataset/data_batch_1', 'Dataset/data_batch_2', 'Dataset/data_batch_3', 'Dataset/data_batch_4', 'Dataset/data_batch_5']
        list_X_all = []
        list_Y_all = []
        list_y_all = []

        for file in filePathList[0]:
            [temp_train_X, temp_train_Y, temp_train_y] = cifar.load_batch_a1(file)
            list_X_all.append(temp_train_X)
            list_Y_all.append(temp_train_Y)
```

```python
# This is a .py file with the name "gradient.py"
import numpy as np
import matplotlib.pyplot as plt
import layer
import network

import datetime
import time

class Gradient:
    def ComputeGradients_Linear_HiddenLayer(self, N, G, H, lambda_cost, W):
        # In assignment1, we used a bit different = ComputeGradients(self, Y, P, X, lambda_cost, W):
        # Y = ground_truth_labels_matrix
        # P = probabilities
        # X = image data
        # let's provide G to the function, so it will slightly be different than assisgnment1
        # G = -np.subtract(Y, P)
        # Also instead of providing Y, now we can provide N only
        # N = Y.shape[1]  # number of images in data (X)

        dL_dW = np.divide(np.dot(G, H.transpose()), N)
        dL_dB = np.divide(np.sum(G, axis=1), N)

        # grad_W = dJ_dW    ...   grad_b = dJ_db
        grad_W = dL_dW + 2 * lambda_cost * W
        grad_b = dL_dB

        # MEL
        # g = gW2          >>> pg. 29(42) - Lecture4.pdf
        # OR as below?
        # np.dot(W.T, G)   >>> pg. 32(45) - Lecture4.pdf (Gbatch = W2.T@Gbatch)
        # since we use matrix notation here, I think we should go for the below one. check the matrix sizes after the execution
        G = np.dot(W.T, G)

        return (grad_W, grad_b, G)

    def ComputeGradients_Linear_FirstLayer(self, N, G, X, lambda_cost, W):
        # In assignment1, we used a bit different = ComputeGradients(self, Y, P, X, lambda_cost, W):
        # Y = ground_truth_labels_matrix
        # P = probabilities
        # X = image data
        # let's provide G to the function, so it will slightly be different than assisgnment1
        # G = -np.subtract(Y, P)
        # Also instead of providing Y, now we can provide N only
        # N = Y.shape[1]  # number of images in data (X)

# This is a .py file with the name "layer.py"
import numpy as np
import matplotlib.pyplot as plt
import gradient

### MEL
### !!! using from requires extra attention if you use circular dependencies !!!
#from gradient import Gradient

class Linear:
    def __init__(self):
        self.gradLinear = gradient.Gradient()
        self.lambda_cost = 0 # wight regularization

    def Forward(self, X, W, b):
        # EvaluationClassifier
        # W = (Kxd) size, randomly initialized weights > then this will be updated by each batch
        # X = each column of X corresponds to an image and it has size (dxn) >> here n will be smaller since
        # it will be selected as subset of images n=100 can be selected
        # b = (Kx1) size, randomly initialized bias > then this will be updated by each batch
        #print('b.shape: ' + str(b.shape))
        #print('X.shape: ' + str(X.shape))
        b_broadcast = np.tile(b, (1, X.shape[1]))
        #b_broadcast = np.broadcast_to(b, (b.shape[0], X.shape[1]))
        #print('b_broadcast.shape: ' + str(b_broadcast.shape))
        s = np.dot(W, X) + b_broadcast
        # p = probabilities of each class to the corresponding images
        # p = self.softmax(s)
        return s

    # NOT USED
    def Backward(self, N, G, X, lambda_cost, W, eta, layer_type='hidden'):
        #P = self.EvaluationClassifier(X, W, b)

        #n_batch = GDparams[0]  # e.g. n_batch=100
        #eta = GDparams[0]      # e.g. eta=0.001
        #n_epocs = GDparams[1]    # e.g. epocs=20

        if layer_type == 'hidden':
            (grad_W, grad_b, G) = self.gradLinear.ComputeGradients_Linear_HiddenLayer(N, G, X, lambda_cost, W2)
            #(grad_W, grad_b, G) = ComputeGradients_Linear_HiddenLayer(N, G, H, lambda_cost, W2)
        else:
            # means first layer
            (grad_W, grad_b) = self.gradLinear.ComputeGradients Linear FirstLayer(N, G, X, lambda cost, W1)
```