Q1HW4 ESE546

November 16, 2020

0.1 Q1

[338]:
```
from sklearn.datasets import fetch_openml
from sklearn import preprocessing
import numpy as np
import cv2
import matplotlib.pyplot as plt
```

[339]:
```
ds = fetch_openml('mnist_784')
x, y = ds.data, ds.target
```

[340]:
```
cutoff = 60000
training_x = x[:cutoff]
training_y = y[:cutoff]
validation_x = x[cutoff:]
validation_y = y[cutoff:]
```

[341]:
```
train_ix = np.logical_or(training_y == '0', training_y == '1')
val_ix = np.logical_or(validation_y == '0', validation_y == '1')
```

[342]:
```
train_x = training_x[train_ix]
train_y = training_y[train_ix]

val_x = validation_x[val_ix]
val_y = validation_y[val_ix]
```

[343]:
```
print(train_x.shape, val_x.shape)
```

```
(12665, 784) (2115, 784)
```

[344]:
```
def resize_images(df, old, new):
    df_temp = df
    num_samples = df_temp.shape[0]
    df_temp = df_temp.reshape((num_samples, old, old))
    downsized = np.zeros((num_samples, new, new))
    for ix in range(num_samples):
        downsized[ix] = cv2.resize(df_temp[ix], (new, new))
    downsized = downsized.reshape((num_samples, new**2))
    return downsized
```

[345]:
```
resized_train_x = resize_images(train_x, 28, 14)
resized_val_x = resize_images(val_x, 28, 14)
```

[581]:
```
label_changer = lambda x: 1 if x == '0' else -1
vector_func = np.vectorize(label_changer)
labeled_train_y = vector_func(train_y)
labeled_val_y = vector_func(val_y)
```

[582]:
```
w = np.random.rand(resized_train_x.shape[1])
b = np.random.rand(1)
```

[583]:
```
class linear_t:
    def __init__(self):

        self.w = np.random.rand(resized_train_x.shape[1] + 1)

        self.hl = None
        self.dw = 0.0

    def forward(self, h_l):

        bias_row = np.ones((h_l.shape[0], 1))
        appended_x = np.concatenate((h_l, bias_row), axis = 1)
        h_l = appended_x

        h_l1 = np.matmul(h_l, np.transpose(self.w))
        self.hl = h_l
        return h_l1

    def backward(self, dh_l1):
        dh_l = np.matmul(dh_l1, self.w)
        dw = np.matmul(dh_l1.T, self.hl)

        self.dw = dw
        return dh_l

    def zero_grad(self):
        self.dw = 0.0 * self.dw
```

[584]:
```
class log_loss:
    def __init__(self):
        self.y = 0.0
        self.yhat = 0.0
        self.w = 0.0
        self.lam = 0.0
    def forward(self, x, y, yhat, w, lam):
        loss = 0

        for i in range(len(y)):
            loss_i = np.log(1 + np.exp(-y[i] * yhat[i]))
            loss += loss_i
        loss /= len(y)

        lambda_loss = lam/2 * np.linalg.norm(w, 2)
        loss += lambda_loss

        self.y = y
        self.yhat = yhat
        self.w = w
        self.lam = lam

        bias_row = np.ones((x.shape[0], 1))
        appended_x = np.concatenate((x, bias_row), axis = 1)
        self.x = appended_x

        pred_val = lambda x: 1 if x < 0 else -1
        vector_func = np.vectorize(pred_val)
        pred = vector_func(yhat)
        acc = np.sum(pred == y) / len(pred)
        return 1 - acc, loss

    def backward(self):
        dw = 0.0
        for i in range(len(self.y)):
            e_term = np.exp(-self.y[i]) * self.yhat[i]
            dw += self.x[i]*((self.y[i] * e_term) / (1 + e_term))
        dw /= len(y)
        dw += self.lam * np.linalg.norm(self.w, 1)
        return dw

    def zero_grad(self):
        return
```

[585]:
```
lr = 1e-1
lam = 1e-3
```

[586]:
```
l1, l3 = linear_t(), log_loss()
net = [l1, l3]
```

[587]:
```
train_error_list, train_loss_list, val_error_list, val_loss_list = [], [], [],␣
 ↪[]
```

[588]:
```
x, y = preprocessing.scale(resized_train_x), labeled_train_y
x_val, y_val = preprocessing.scale(resized_val_x), labeled_val_y
```

[589]:
```
for t in range(30):

    for l in net:
        l.zero_grad()

    hl = l1.forward(x)
    ell, error = l3.forward(x, y, hl, l1.w, lam)

    dw = l3.backward()

    if(t % 5 == 1):
        print(t, ell, error)

    train_loss_list.append(ell)
    train_error_list.append(error)

    hl = l1.forward(x_val)

    val_loss, val_error = l3.forward(x_val, y_val, hl, l1.w, lam)

    val_loss_list.append(val_loss)
    val_error_list.append(val_error)

    l1.w = l1.w - lr * dw
```

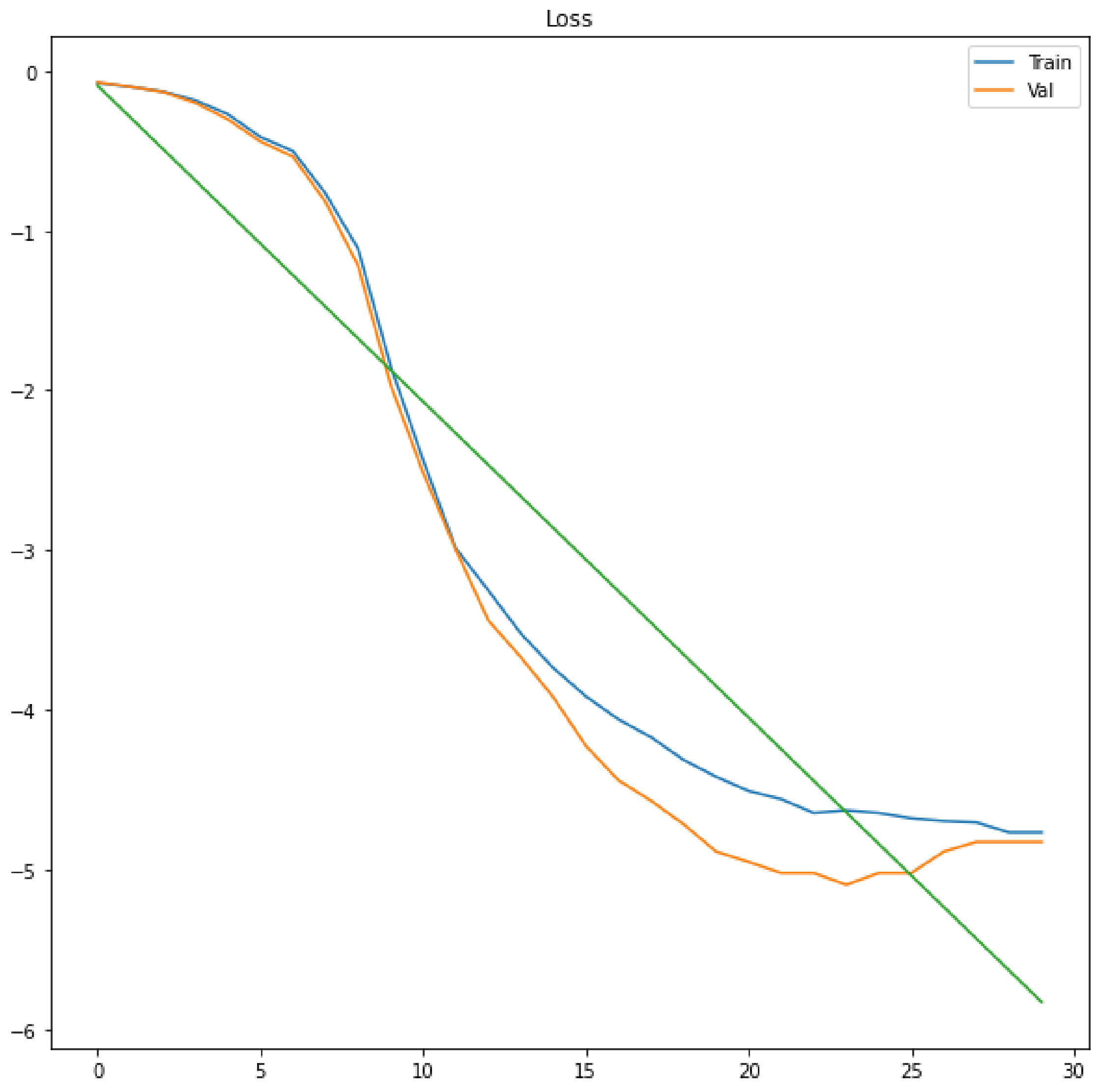
```
1 0.9039084090011844 0.7198352076509086
6 0.6045795499407817 2.521019022140901
11 0.050454007106198184 14.552846728523551
16 0.01729174891433083 29.511400363092307
21 0.010501381760757988 44.567836214650846
26 0.009159099881563382 59.82563312145311
```

[590]:
```
log_train_loss = [np.log(x) for x in train_loss_list]
log_val_loss = [np.log(x) for x in val_loss_list]
```

[593]:
```
m, b = np.polyfit(np.arange(len(train_loss_list)), np.log(train_loss_list), 1)
print(m, b)
```

```
-0.19761873027105417 -0.09378724894707781
```

[595]:
```
plt.plot(np.log(train_loss_list), label='Train')
plt.plot(np.log(val_loss_list), label='Val')
plt.plot([x * m + b for x in np.arange(len(train_loss_list))])
plt.legend()
plt.title('Loss')
plt.show()
```



0.1.1 Part (b)

Slope of the above curve can be estimated at $-.192$ using the line of best fit. This implies

$$\kappa = -1/-.192 = 5.06024909$$

0.1.2 Part (c)

We know that the first derivative of the loss function with respect to some w_i is

$$\frac{1}{n}\sum_{i=1}^{n} x_i[j]y[i] * \frac{e^{-y_i * w^T * x_i}}{1 + e^{-y_i * w^T * x_i}}$$

where $x_i[j]$ is the j^{th} feature of the i^{th} sample.

To find the Hessian, we take the next partial derivatives of the above equation.

0.1.3 Part (d)

[376]:
```
x, y = ds.data, ds.target
```

[377]:
```
cutoff = 60000
training_x = x[:cutoff]
training_y = y[:cutoff]
validation_x = x[cutoff:]
validation_y = y[cutoff:]
```

[378]:
```
train_ix = np.logical_or(training_y == '0', training_y == '1')
val_ix = np.logical_or(validation_y == '0', validation_y == '1')
```

[379]:
```
train_x = training_x[train_ix]
train_y = training_y[train_ix]

val_x = validation_x[val_ix]
val_y = validation_y[val_ix]
```

[380]:
```
print(train_x.shape, val_x.shape)
```

```
(12665, 784) (2115, 784)
```

[381]:
```
resized_train_x = resize_images(train_x, 28, 14)
resized_val_x = resize_images(val_x, 28, 14)
```

[382]:
```
label_changer = lambda x: 1 if x == '0' else -1
vector_func = np.vectorize(label_changer)
labeled_train_y = vector_func(train_y)
labeled_val_y = vector_func(val_y)
```

[383]:
```
w = np.random.rand(resized_train_x.shape[1])
b = np.random.rand(1)
```

[384]:
```
class log_loss_nesterov:
    def __init__(self):
        self.y = 0.0
        self.yhat = 0.0
        self.w = 0.0
        self.lam = 0.0

    def forward(self, x, y, yhat, w, lam):
        loss = 0
        for i in range(len(y)):
            loss_i = np.log(1 + np.exp(-y[i] * yhat[i]))
            loss += loss_i
        loss /= len(y)

        lambda_loss = lam/2 * np.linalg.norm(w, 2)
        loss += lambda_loss

        self.y = y
        self.yhat = yhat
        self.w = w
        self.lam = lam

        bias_row = np.ones((x.shape[0], 1))
        appended_x = np.concatenate((x, bias_row), axis = 1)
        self.x = appended_x

        pred_val = lambda x: 1 if x < 0 else -1
        vector_func = np.vectorize(pred_val)
        pred = vector_func(yhat)
        acc = np.sum(pred == y) / len(pred)
        return 1 - acc, loss

    def backward(self, new_w):
        dw = 0.0
        for i in range(len(self.y)):
            e_term = np.exp(-self.y[i] * np.matmul(self.x[i], np.
↪transpose(new_w)))
            dw += self.x[i] *((self.y[i] * e_term) / (1 + e_term))
        dw /= len(y)
        dw += self.lam * np.linalg.norm(new_w, 1)
        return dw

    def zero_grad(self):
        return
```

[397]:
```
lr = 1e-2
lam = 1e-4
k = 0.0732
L = 0.054
gamma = L / k
```

[398]:
```
l1, l3 = linear_t(), log_loss_nesterov()
last_update = l1.w
net = [l1, l3]
```

```
[399]: train_error_list, train_loss_list, val_error_list, val_loss_list = [], [], [],␣
        ↪[]
```

```
[400]: x, y = preprocessing.scale(resized_train_x), labeled_train_y
       x_val, y_val = preprocessing.scale(resized_val_x), labeled_val_y
```

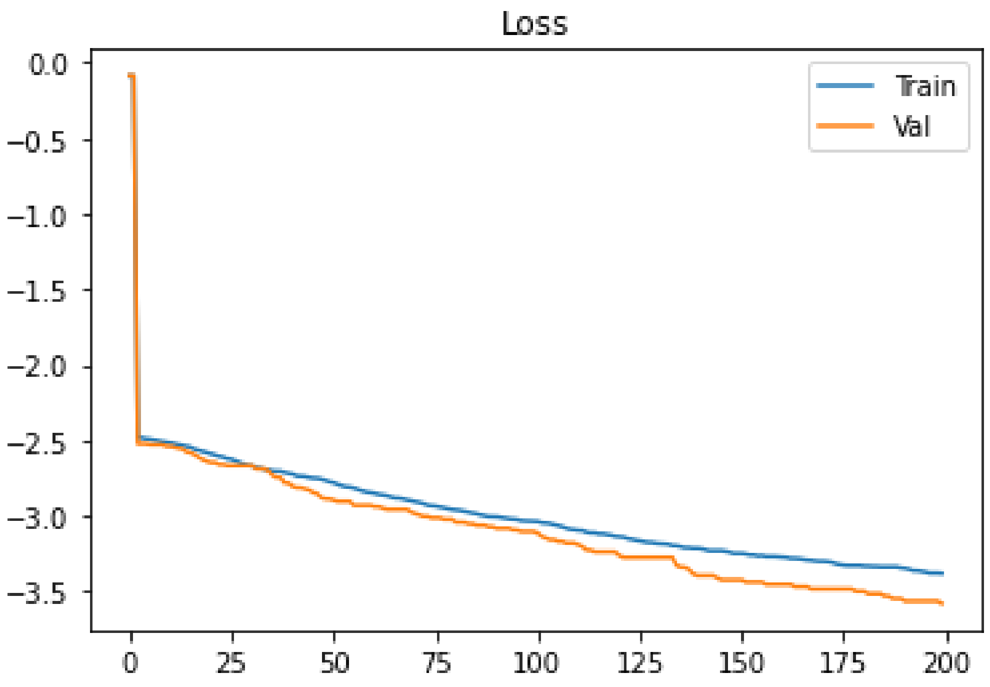
```
[401]: for t in range(200):

           for l in net:
               l.zero_grad()

           hl = l1.forward(x)
           ell, error = l3.forward(x, y, hl, l1.w, lam)

           if(t % 10 == 1):
               print(t, ell, error)

           train_loss_list.append(ell)
           train_error_list.append(error)

           hl = l1.forward(x_val)

           val_loss, val_error = l3.forward(x_val, y_val, hl, l1.w, lam)

           val_loss_list.append(val_loss)
           val_error_list.append(val_error)

           momentum = gamma * last_update
           dw = l3.backward(l1.w - momentum)
           nesterov_dw = momentum + lr * dw
           last_update = nesterov_dw

           l1.w = l1.w - nesterov_dw
```

```
1 0.9155151993683379 0.24286608532762216
11 0.08030003947887876 29.643727219345458
21 0.07453612317410185 32.94696812358209
31 0.06861429135412556 34.81105555716599
41 0.06490327674694041 36.621302906929806
51 0.06142913541255424 38.442387185685796
61 0.057402289774970416 40.276043054045836
71 0.05440189498618242 42.12080849526941
81 0.0516383734701934 43.975422895760055
91 0.04926964074220297 45.83902158094745
101 0.04761152783260958 47.71112178031243
111 0.04524279510461904 49.591671550322374
121 0.043268851164626976 51.48068883063053
```

```
131 0.0413738649822345 53.377954648348975
141 0.039794709830240804 55.28313951408746
151 0.038689301223845285 57.19594656717
161 0.03782076589024874 59.11609939668688
171 0.03679431504145281 61.04332373113328
181 0.03568890643505729 62.977396427272865
191 0.03497828661666014 64.91823275952937
```

[402]:
```
log_train_loss = [np.log(x) for x in train_loss_list]
log_val_loss = [np.log(x) for x in val_loss_list]
```

[403]:
```
plt.plot(log_train_loss, label='Train')
plt.plot(log_val_loss, label='Val')
plt.legend()
plt.title('Loss')
plt.show()
```



Q1 (e)

[404]:
```
x, y = ds.data, ds.target
```

[405]:
```
cutoff = 60000
training_x = x[:cutoff]
training_y = y[:cutoff]
validation_x = x[cutoff:]
```

```
validation_y = y[cutoff:]
```

[406]:
```
train_ix = np.logical_or(training_y == '0', training_y == '1')
val_ix = np.logical_or(validation_y == '0', validation_y == '1')
```

[407]:
```
train_x = training_x[train_ix]
train_y = training_y[train_ix]

val_x = validation_x[val_ix]
val_y = validation_y[val_ix]
```

[408]:
```
print(train_x.shape, val_x.shape)
```

```
(12665, 784) (2115, 784)
```

[409]:
```
def shuffle_dataset(X, Y):
    indices = np.arange(X.shape[0])
    np.random.shuffle(indices)

    return X[indices], Y[indices]

def batch_generator(X, Y, batch_size):
    while True:
        X, Y = shuffle_dataset(X, Y)
        for i in range((X.shape[0] - batch_size) // batch_size):
            yield X[i * batch_size: i * batch_size + batch_size], Y[i *␣
 ↪batch_size: i * batch_size + batch_size]
    return x_batch, y_batch
```

[410]:
```
resized_train_x = resize_images(train_x, 28, 14)
resized_val_x = resize_images(val_x, 28, 14)
```

[411]:
```
label_changer = lambda x: 1 if x == '0' else -1
vector_func = np.vectorize(label_changer)
labeled_train_y = vector_func(train_y)
labeled_val_y = vector_func(val_y)
```

[412]:
```
w = np.random.rand(resized_train_x.shape[1])
b = np.random.rand(1)
```

[414]:
```
lr = 1e-4
lam = 1e-4
k = 0.071
L = 0.054
gamma = L / k
```

[415]:
```
l1, l3 = linear_t(), log_loss_nesterov()
last_update = l1.w
net = [l1, l3]
```

[416]:
```
train_error_list, train_loss_list, val_error_list, val_loss_list = [], [], [],␣
 ↪[]
```

[417]:
```
x_full, y_full = preprocessing.scale(resized_train_x), labeled_train_y
x_val_full, y_val_full = preprocessing.scale(resized_val_x), labeled_val_y
```

[418]:
```
train_dataloader = batch_generator(x_full, y_full, batch_size=128)
val_dataloarder = batch_generator(x_val_full, y_val_full, batch_size=128)
```

[419]:
```
for t in range(int(1e4)):

    x, y = train_dataloader.__next__()
    x_val, y_val = val_dataloarder.__next__()

    for l in net:
        l.zero_grad()

    hl = l1.forward(x)
    ell, error = l3.forward(x, y, hl, l1.w, lam)

    if(not (t % 1e3)):
        print(t, ell, error)

    train_loss_list.append(ell)
    train_error_list.append(error)

    hl = l1.forward(x_val)

    val_loss, val_error = l3.forward(x_val, y_val, hl, l1.w, lam)

    val_loss_list.append(val_loss)
    val_error_list.append(val_error)

    momentum = gamma * last_update
    dw = l3.backward(l1.w - momentum)
    nesterov_dw = momentum + lr * dw
    last_update = nesterov_dw

    l1.w = l1.w - nesterov_dw
```

```
0 0.953125 0.3392207618421814
100 0.0546875 40.54117032656842
200 0.0859375 35.46330630005275
```

```
300 0.03125 46.23057947998151
400 0.0625 42.67176886537754
500 0.0234375 44.66052161805805
600 0.0546875 46.47370256713002
700 0.0703125 48.24480419290685
800 0.03125 47.496673335889376
900 0.0546875 46.89872312869431
1000 0.03125 52.43991357273287
1100 0.0625 51.01421201019508
1200 0.046875 53.55501320215133
1300 0.03125 56.56752003115949
1400 0.046875 50.1809871890456
1500 0.0390625 55.15131601991018
1600 0.03125 55.40985251168317
1700 0.0234375 60.64987483612179
1800 0.0390625 55.610592469017476
1900 0.03125 57.79175636606615
2000 0.0546875 61.11606438131098
2100 0.0234375 60.96012365122241
2200 0.015625 64.64554919139923
2300 0.0390625 65.03492000464392
2400 0.015625 64.39459844059044
2500 0.03125 64.95409658912257
2600 0.0703125 69.14907367964695
2700 0.015625 68.61333022567328
2800 0.0234375 71.3777609278034
2900 0.03125 67.70768716445397
3000 0.0390625 70.54503292054223
3100 0.03125 70.81237064800914
3200 0.0234375 77.2113548039414
3300 0.015625 73.92359368312073
3400 0.0078125 81.31596813769852
3500 0.046875 80.06265883110416
3600 0.015625 84.28312651919026
3700 0.0390625 78.87387798359924
3800 0.0234375 82.5745967253733
3900 0.0078125 84.68349430757283
4000 0.0078125 89.60030982465861
4100 0.0234375 83.44128380010123
4200 0.0078125 86.89818158721415
4300 0.0234375 90.13962549047523
4400 0.0390625 82.62774299676649
4500 0.0234375 90.14067964898894
4600 0.046875 84.93741106835722
4700 0.015625 91.87886448309119
4800 0.03125 85.43193364771734
4900 0.0234375 95.45100349339239
5000 0.0078125 98.15569254553995
```

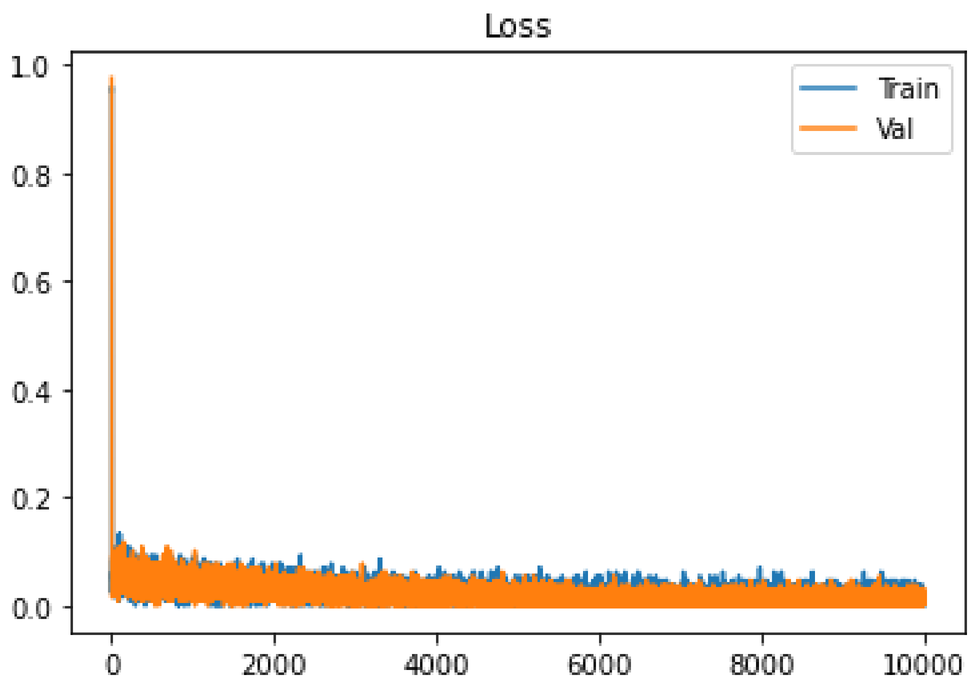
```
5100 0.0078125 96.44352549014035
5200 0.0546875 101.44272349592396
5300 0.03125 92.39381403743845
5400 0.0078125 100.8489736849586
5500 0.0234375 98.5477645342444
5600 0.0078125 99.75449777520743
5700 0.015625 103.65052999312576
5800 0.0 109.80971761243056
5900 0.015625 108.21893791046972
6000 0.0078125 104.63815432573901
6100 0.0078125 107.3260636467946
6200 0.03125 104.67033904412357
6300 0.015625 108.27348306039205
6400 0.015625 110.01986471935388
6500 0.015625 112.95948962232816
6600 0.0390625 108.1312624063106
6700 0.015625 107.31171158785664
6800 0.0078125 115.10817909833439
6900 0.0234375 116.85171048768248
7000 0.0078125 116.55821895752072
7100 0.0078125 119.98399652926516
7200 0.0 118.93554124009175
7300 0.03125 110.23210118385
7400 0.015625 123.13291313477401
7500 0.0078125 122.89509827531859
7600 0.0 125.20112590423103
7700 0.0234375 120.90719534028835
7800 0.0234375 136.95254317281368
7900 0.0078125 121.86665485683447
8000 0.0234375 131.17020406230708
8100 0.015625 129.5762070597723
8200 0.0078125 120.80006768557138
8300 0.0078125 127.75806635661944
8400 0.015625 125.39620442828893
8500 0.0 140.89280995666087
8600 0.0078125 130.54834798662594
8700 0.015625 142.53336892360102
8800 0.0390625 130.93724717910533
8900 0.015625 137.7019653245975
9000 0.015625 139.9002112460901
9100 0.015625 139.75356430104011
9200 0.0 154.58143493068096
9300 0.0234375 142.80036274304953
9400 0.015625 144.45942519597583
9500 0.015625 146.48482998591828
9600 0.015625 137.05615774804215
9700 0.0078125 145.58599401419897
9800 0.0078125 147.22134863244693
```

```
9900 0.0234375 137.45679720131875
```

[420]:
```
log_train_loss = [np.log(x) for x in train_loss_list]
log_val_loss = [np.log(x) for x in val_loss_list]
```

```
/home/sheil/miniconda3/envs/TF/lib/python3.7/site-
packages/ipykernel_launcher.py:1: RuntimeWarning: divide by zero encountered in
log
  """Entry point for launching an IPython kernel.
/home/sheil/miniconda3/envs/TF/lib/python3.7/site-
packages/ipykernel_launcher.py:2: RuntimeWarning: divide by zero encountered in
log
```

[422]:
```
plt.plot(train_loss_list, label='Train')
plt.plot(val_loss_list, label='Val')
plt.legend()
plt.title('Loss')
plt.show()
```



[ ]:
```

```

0.1.4 Training loss against number of parameter updates on a semi-log scale

Gradient descent with Nesterov (d)

[530]:
```
lr = 5e-3
lam = 1e-4
```

[531]:
```
l1, l3 = linear_t(), log_loss_nesterov()
last_update = l1.w
net = [l1, l3]
```

[532]:
```
grad_nesterov = []
```

[533]:
```
for t in range(200):

    for l in net:
        l.zero_grad()

    hl = l1.forward(x)
    ell, error = l3.forward(x, y, hl, l1.w, lam)

    if(t % 10 == 1):
        print(t, ell, error)

    grad_nesterov.append(ell)

    hl = l1.forward(x_val)

    val_loss, val_error = l3.forward(x_val, y_val, hl, l1.w, lam)

    momentum = gamma * last_update
    dw = l3.backward(l1.w - momentum)
    nesterov_dw = momentum + lr * dw
    last_update = nesterov_dw

    l1.w = l1.w - nesterov_dw
```

```
1 0.9296875 0.1966927234742865
11 0.0703125 41.76668627034822
21 0.0546875 50.04464309878662
31 0.0546875 55.91801336731703
41 0.046875 61.68004794228292
51 0.046875 67.4537125605734
61 0.046875 73.23707548119138
71 0.046875 79.0305583221826
81 0.046875 84.83976409163266
91 0.0390625 90.67641520196841
101 0.0390625 96.53984221617202
111 0.0390625 102.41494994737418
121 0.0390625 108.29825382002463
131 0.0390625 114.18935055160082
```

```
141 0.0390625 120.08820164225149
151 0.0390625 125.99487679849607
161 0.0390625 131.9095174717802
171 0.0390625 137.8321652056106
181 0.0390625 143.76284596935363
191 0.0390625 149.70158608306184
```

[534]:
```
x_grad_nesterov = np.arange(len(grad_nesterov))
print(len(x_grad_nesterov), len(grad_nesterov))
```

```
200 200
```

SGD without Nesterov (new)

[459]:
```
lr = 1e-3
lam = 1e-3
```

[460]:
```
l1, l3 = linear_t(), log_loss()
net = [l1, l3]
```

[461]:
```
sgd_train_loss_list = []
```

[462]:
```
x, y = preprocessing.scale(resized_train_x), labeled_train_y
x_val, y_val = preprocessing.scale(resized_val_x), labeled_val_y
```

[463]:
```
for t in range(int(1e4)):

    x, y = train_dataloader.__next__()
    x_val, y_val = val_dataloarder.__next__()

    for l in net:
        l.zero_grad()

    hl = l1.forward(x)
    ell, error = l3.forward(x, y, hl, l1.w, lam)

    dw = l3.backward()

    if(t % 1000 == 1):
        print(t, ell, error)

    sgd_train_loss_list.append(ell)

    hl = l1.forward(x_val)

    val_loss, val_error = l3.forward(x_val, y_val, hl, l1.w, lam)

    l1.w = l1.w - lr * dw
```

```
1 0.96875 0.30846224333362496
1001 0.046875 25.511061681100276
2001 0.0078125 62.86264440262056
3001 0.0078125 92.4026049169232
4001 0.0078125 120.570412298619
5001 0.0234375 165.59824893051197
6001 0.015625 187.68844607471496
```

```
/home/sheil/miniconda3/envs/TF/lib/python3.7/site-
packages/ipykernel_launcher.py:12: RuntimeWarning: overflow encountered in exp
  if sys.path[0] == '':
```

```
7001 0.0078125 243.21627558922552
8001 0.0078125 296.4380469048227
9001 0.0078125 inf
```

[464]:
```
x_sgd = np.arange(len(sgd_train_loss_list))
print(len(sgd_train_loss_list), len(x_sgd))
```

```
10000 10000
```

SGD with Nesterov (e)

[539]:
```
lr = 1e-4
lam = 1e-4
k = 0.071
L = 0.054
gamma = L / k
```

[540]:
```
l1, l3 = linear_t(), log_loss_nesterov()
last_update = l1.w
net = [l1, l3]
```

[541]:
```
sgd_nesterov = []
```

[542]:
```
for t in range(int(1e4)):

    x, y = train_dataloader.__next__()
    x_val, y_val = val_dataloarder.__next__()

    for l in net:
        l.zero_grad()

    hl = l1.forward(x)
    ell, error = l3.forward(x, y, hl, l1.w, lam)

    if(not (t % 1e3)):
        print(t, ell, error)
    sgd_nesterov.append(ell)

    hl = l1.forward(x_val)

    val_loss, val_error = l3.forward(x_val, y_val, hl, l1.w, lam)

    momentum = gamma * last_update
    dw = l3.backward(l1.w - momentum)
    nesterov_dw = momentum + lr * dw
    last_update = nesterov_dw

    l1.w = l1.w - nesterov_dw
```

```
0 0.9296875 0.3756032904523797
1000 0.0625 43.70161401850441
2000 0.046875 59.59140656862599
3000 0.015625 71.40436181585619
4000 0.03125 79.01258459814002
5000 0.0234375 92.8857912700847
6000 0.0078125 103.18619232934208
7000 0.0390625 107.23007692063672
8000 0.015625 131.27473808873643
9000 0.015625 146.21221001768043
```

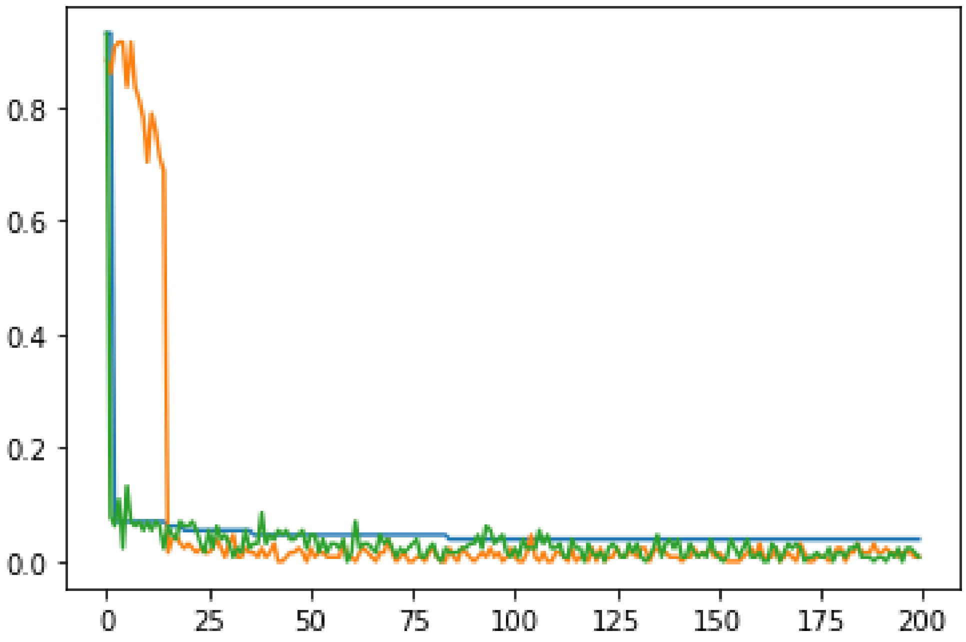
[543]:
```
x_sgd_nesterov = np.arange(len(sgd_nesterov))
print(len(sgd_nesterov), len(x_sgd_nesterov))
```

```
10000 10000
```

0.2 Plot (1e)

[553]:
```
plt.plot(x_grad_nesterov, grad_nesterov)
plt.plot(sgd_train_loss_list[::50])
plt.plot(sgd_nesterov[::50])
```

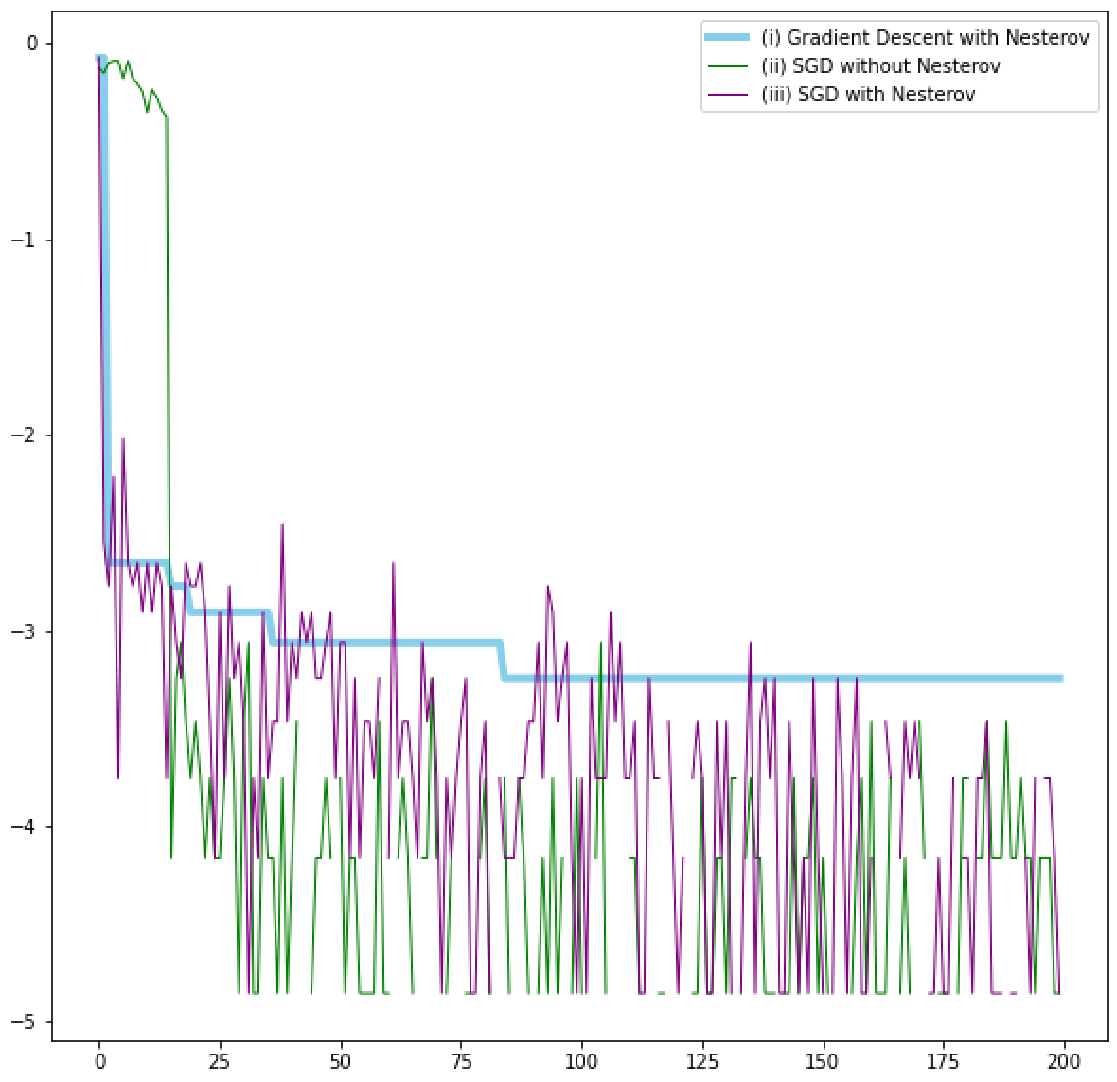
[553]:
```
[<matplotlib.lines.Line2D at 0x7fae6173cf10>]
```



```
[578]: plt.rcParams["figure.figsize"] = (10,10)
```

```
[580]: plt.plot(x_grad_nesterov, np.log(grad_nesterov),
                color='skyblue', linewidth=4,
                label='(i) Gradient Descent with Nesterov')
       plt.plot(np.log(sgd_train_loss_list[::50]),
                color='Green', linewidth=1,
                label='(ii) SGD without Nesterov')
       plt.plot(np.log(sgd_nesterov[::50]),
                color='Purple', linewidth=1,
                label='(iii) SGD with Nesterov')
       plt.legend()
       plt.show()
```

```
/home/sheil/miniconda3/envs/TF/lib/python3.7/site-
packages/ipykernel_launcher.py:4: RuntimeWarning: divide by zero encountered in
log
  after removing the cwd from sys.path.
/home/sheil/miniconda3/envs/TF/lib/python3.7/site-
packages/ipykernel_launcher.py:7: RuntimeWarning: divide by zero encountered in
log
  import sys
```



0.3 Is the convergence for (iii) faster than that of (ii)?

No, the convergence of (iii) is not faster than that of (ii). This is inline with our expectations, since we did not expect Nesterov to speed-up SGD by a significant amount.

0.4 Comment on the differences for the convergence curves of (i) and (ii)

Differences: - SGD took 10,000 iterations to converge, whereas Gradient Descent with Nesterov took only 200 to arrive at a similar training loss. - The loss was monotonically decreasing in gradient descent, whereas SGD without Nesterov did not display the same property due to some batch losses being much larger than others.

Q2HW4 ESE546

November 16, 2020

[83]:
```
# torch and torchvision imports
import torch
import torchvision
import torch.nn as nn
import torchvision.transforms as transforms
import torch.optim as optim
from torch.autograd import Variable
import tensorflow as tf
import numpy as np
import scipy.misc
try:
    from StringIO import StringIO  # Python 2.7
except ImportError:
    from io import BytesIO         # Python 3.x

import matplotlib.pyplot as plt
```

[84]:
```
class View(nn.Module):
    def __init__(self,o):
        super().__init__()
        self.o = o

    def forward(self,x):
        return x.view(-1, self.o)

class allcnn_t(nn.Module):
    def __init__(self, c1=96, c2= 192):
        super().__init__()
        d = 0.5

        def convbn(ci,co,ksz,s=1,pz=0):
            return nn.Sequential(
                nn.Conv2d(ci,co,ksz,stride=s,padding=pz),
                nn.ReLU(True),
                nn.BatchNorm2d(co))

        self.m = nn.Sequential(
            nn.Dropout(0.2),
            convbn(3,c1,3,1,1),
            convbn(c1,c1,3,1,1),
            convbn(c1,c1,3,2,1),
            nn.Dropout(d),
            convbn(c1,c2,3,1,1),
            convbn(c2,c2,3,1,1),
            convbn(c2,c2,3,2,1),
            nn.Dropout(d),
            convbn(c2,c2,3,1,1),
            convbn(c2,c2,3,1,1),
            convbn(c2,10,1,1),
            nn.AvgPool2d(8),
            View(10))

        print('Num parameters: ', sum([p.numel() for p in self.m.parameters()]))

    def forward(self, x):
        return self.m(x)
```

[84]:

```
```

[85]:

```
"""# Training Loop on CIFAR 10.
 The model currently does not achieve less than 12% validation error, you have␣
 ↪to tweak the parameters to get it.
"""

# Reading in the dataset
transform = transforms.Compose(
    [
      transforms.RandomCrop(32, padding=4),
      transforms.RandomHorizontalFlip(),
     transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=128,
                                          shuffle=True)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=128,
                                         shuffle=False)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

```
Files already downloaded and verified
Files already downloaded and verified
```

[18]:
```
# The training loop
def train(net, optimizer, criterion, train_loader, test_loader, epochs,␣
 ↪model_name, plot):
    model = net.to(device)
    total_step = len(train_loader)

    loss_values = []
    accuracy_vals = []

    overall_step = 0

    for epoch in range(epochs):
        correct = 0
        total = 0
        epoch_loss = 0

        if(epoch == 40):
            for param_group in optimizer.param_groups:
                param_group['lr'] = 0.01
        elif (epoch == 80):
            for param_group in optimizer.param_groups:
                param_group['lr'] = 0.001

        for i, (images, labels) in enumerate(train_loader):
            # Move tensors to configured device
            images = images.to(device)
            labels = labels.to(device)
            #Forward Pass
            outputs = model(images)
            loss = criterion(outputs, labels)

            epoch_loss += loss.item()

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            if (i+1) % 1000 == 1:
                print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'.format(
                  epoch+1, epochs, i+1, total_step, loss.item()))
            if plot:
                info = { ('loss_' + model_name): loss.item() }
        overall_step += 1
        epoch_loss /= len(train_loader)
        print("Epoch loss: " + str(epoch_loss))
        loss_values.append(epoch_loss)

    model.eval()
    with torch.no_grad():
        correct = 0
        total = 0
        for i, (images, labels) in enumerate(test_loader):
            images = images.to(device)
            labels = labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            accuracy = 100*(correct / total)
            accuracy_vals.append(accuracy)

    print('Accuracy of the network on the test images: {} %'.format(accuracy))
    return loss_values, accuracy_vals
```

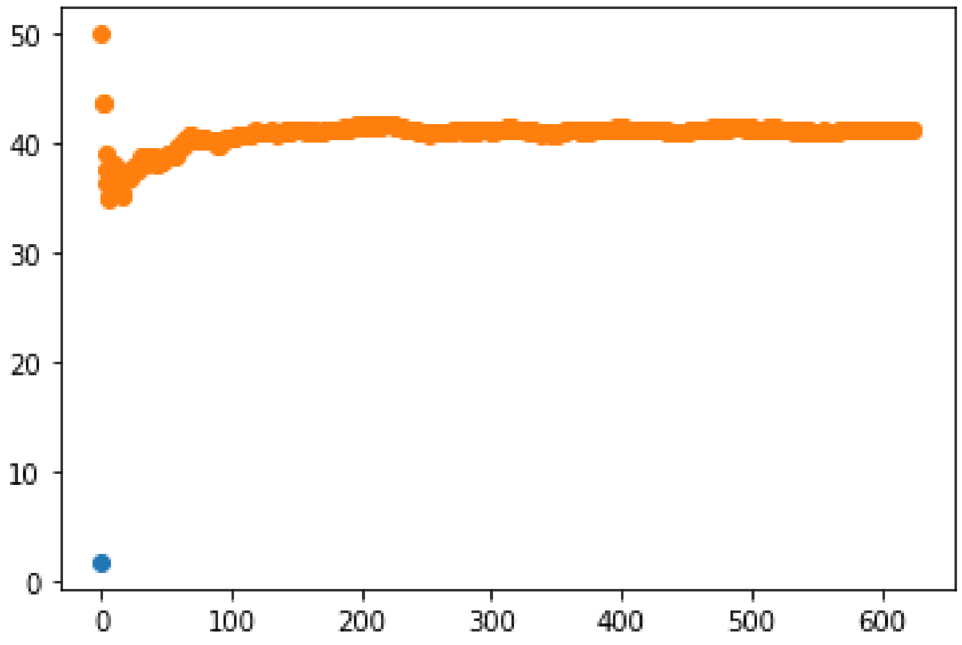
[11]:
```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

[ ]:
```
model = allcnn_t().to(device)

#TODO: Set it as number of epochs states in the question (100)
epochs = 50

#Define the loss function as asked in the question
criterion = nn.CrossEntropyLoss()

#Set parameters as stated in the question
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9,␣
 ↪weight_decay=0.001)

# Training loop called here
loss_values, accuracy_values = train(
    model, optimizer, criterion, trainloader, testloader, epochs, 'cnn', True)
```

[20]:
```
loss_values

# Plot the training and validation losses as a function of the number of epochs
plt.scatter(range(len(loss_values)), loss_values)

# Plot the training and validation errors as a function of the number of epochs
```

```
plt.scatter(range(len(accuracy_values)), accuracy_values)
```

[20]:
```
<matplotlib.collections.PathCollection at 0x7f395e1fbda0>
```



0.0.1 Part (a)

0.0.2 Since all classes start with equal probability, $-ln(0.1) = 2.3$

0.0.3 Part (b)

[69]:
```
eta = 1e-5
```

[70]:
```
training_batch_loss = []
stop_val = 150
```

[71]:
```
learning_rate = []
learning_rate.append(eta)
for i in range(stop_val):
    learning_rate.append(learning_rate[-1] * 1.1)
```

[72]:
```
# The training loop
def train_lr(net, optimizer, criterion, train_loader, test_loader, epochs,␣
 ↪model_name, plot):
    model = net.to(device)
    total_step = len(train_loader)
    loss_values = []
    accuracy_vals = []

    overall_step = 0

    for epoch in range(epochs):
        correct = 0
        total = 0
        epoch_loss = 0

        for i, (images, labels) in enumerate(train_loader):
            # Move tensors to configured device
            images = images.to(device)
            labels = labels.to(device)
            #Forward Pass
            outputs = model(images)
            loss = criterion(outputs, labels)

            epoch_loss += loss.item()

            training_batch_loss.append(loss.item())
            if(len(training_batch_loss) > stop_val):
                break

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            for g in optimizer.param_groups:
                g['lr'] *= 1.1

            if (i+1) % 1000 == 1:
                print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'.format(
                  epoch+1, epochs, i+1, total_step, loss.item()))
            if plot:
                info = { ('loss_' + model_name): loss.item() }
        overall_step += 1

        epoch_loss /= len(train_loader)
        print("Epoch loss: " + str(epoch_loss))
        loss_values.append(epoch_loss)

    model.eval()
    with torch.no_grad():
        correct = 0
        total = 0
        for i, (images, labels) in enumerate(test_loader):
            images = images.to(device)
            labels = labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            accuracy = 100*(correct / total)
            accuracy_vals.append(accuracy)

    print('Accuracy of the network on the test images: {} %'.format(accuracy))
    return loss_values, accuracy_vals
```

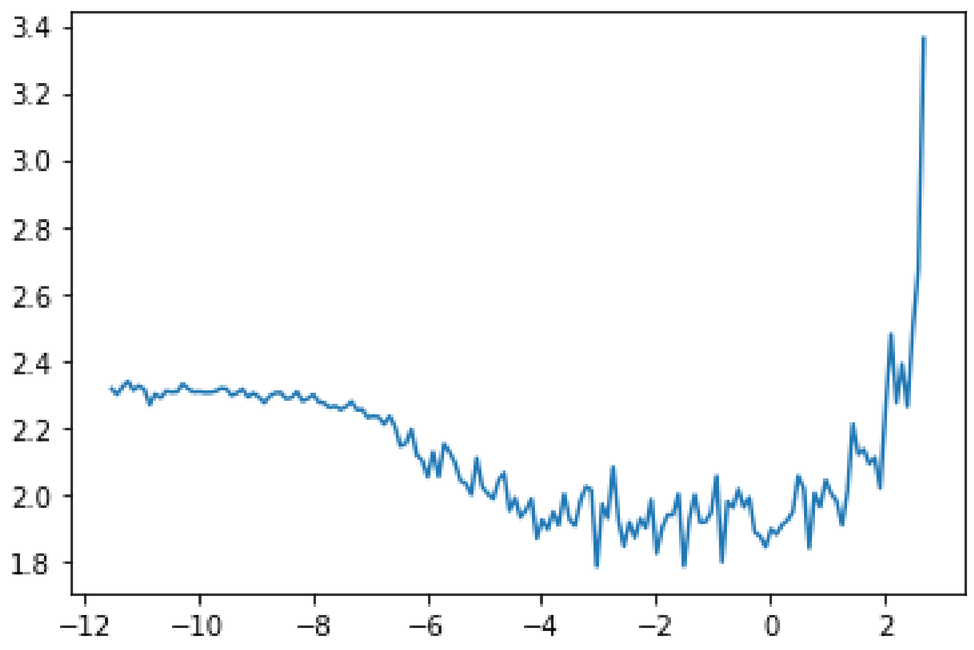
[73]:
```
model = allcnn_t().to(device)

#TODO: Set it as number of epochs states in the question (100)
epochs = 1

#Define the loss function as asked in the question
criterion = nn.CrossEntropyLoss()

#Set parameters as stated in the question
optimizer = optim.SGD(model.parameters(), lr=eta, momentum=0.9, weight_decay=0.
 ↪001)

# Training loop called here
loss_values, accuracy_values = train_lr(
    model, optimizer, criterion, trainloader, testloader, epochs, 'cnn', True)
```

```
Num parameters:  1667166
Epoch [1/1], Step [1/391], Loss: 2.3153
Epoch loss: 0.8207876328617105
Accuracy of the network on the test images: 10.11 %
```

[74]:
```
len(training_batch_loss)
```

[74]:
```
151
```

[75]:
```
plt.plot(np.log(learning_rate[:150]), training_batch_loss[:150])
plt.show()
```



[76]:
```
min_ix = np.argmin(training_batch_loss)
print(learning_rate[min_ix])
print(training_batch_loss[min_ix])
```

```
0.04830020556225739
1.7841922044754028
```

0.0.4 Local minimum seems to occur at 10^{-2}

0.1 Part (c)

[162]:
```
T0 = 19550 / 5
T = 19550
eta_max = 1e-2 # eye-balling the minimum / 10
```

[163]:
```
training_loss = []
validation_loss = []

training_error = []
validation_error = []
```

[164]:
```
# The training loop
def train_lr(net, optimizer, criterion, train_loader, test_loader, epochs,␣
 ↪model_name, plot):
    model = net.to(device)
    total_step = len(train_loader)
```

```
overall_step = 0
t = 0

for epoch in range(epochs):
    correct = 0
    total = 0
    epoch_loss = 0

    model.train()

    for i, (images, labels) in enumerate(train_loader):

        # update learning rate
        if t < T0:
            new_lr = 1e-4 + (t/T0)*nmax
        else:
            new_lr = nmax*np.cos((np.pi/2)*(t - T0)/(T - T0)) + 1e-6

        for g in optimizer.param_groups:
            g['lr'] = new_lr

        # Move tensors to configured device
        images = images.to(device)
        labels = labels.to(device)

        # Forward Pass
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
        loss = criterion(outputs, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        t += 1

        training_loss.append(loss.item())
        training_error.append(1 - correct/total)

        if (i) % 100 == 0:
          print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'.
↪format(epoch+1, epochs, i+1, total_step, loss.item()))

    epoch_loss /= len(train_loader)
    print("Epoch loss: " + str(epoch_loss))
    loss_values.append(epoch_loss)
```

```
        model.eval()

        with torch.no_grad():
            correct = 0
            total = 0
            for i, (images, labels) in enumerate(test_loader):
                images = images.to(device)
                labels = labels.to(device)
                outputs = model(images)
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()
            validation_loss.append(loss.item())
            validation_error.append(1 - correct / total)

    print('Accuracy of the network on the test images: {} %'.format(100 *␣
↪correct / total))
    return
```

[165]:
```
model = allcnn_t().to(device)

epochs = 50

#Define the loss function as asked in the question
criterion = nn.CrossEntropyLoss()

#Set parameters as stated in the question
optimizer = optim.SGD(model.parameters(), lr=eta, momentum=0.9, weight_decay=0.
↪001)

# Training loop called here
train_lr(model, optimizer, criterion, trainloader, testloader, epochs, 'cnn',␣
↪True)
```

```
Num parameters:  1667166
Epoch [1/50], Step [1/391], Loss: 2.2975
Epoch [1/50], Step [101/391], Loss: 2.1078
Epoch [1/50], Step [201/391], Loss: 1.9483
Epoch [1/50], Step [301/391], Loss: 1.9817
Epoch loss: 0.0
Epoch [2/50], Step [1/391], Loss: 1.7292
Epoch [2/50], Step [101/391], Loss: 1.8746
Epoch [2/50], Step [201/391], Loss: 1.7938
Epoch [2/50], Step [301/391], Loss: 1.7377
Epoch loss: 0.0
Epoch [3/50], Step [1/391], Loss: 1.6278
Epoch [3/50], Step [101/391], Loss: 1.5064
Epoch [3/50], Step [201/391], Loss: 1.4964
Epoch [3/50], Step [301/391], Loss: 1.5147
Epoch loss: 0.0
Epoch [4/50], Step [1/391], Loss: 1.4488
Epoch [4/50], Step [101/391], Loss: 1.4401
Epoch [4/50], Step [201/391], Loss: 1.3071
Epoch [4/50], Step [301/391], Loss: 1.3397
Epoch loss: 0.0
Epoch [5/50], Step [1/391], Loss: 1.2333
Epoch [5/50], Step [101/391], Loss: 1.1833
Epoch [5/50], Step [201/391], Loss: 1.2305
Epoch [5/50], Step [301/391], Loss: 1.0268
Epoch loss: 0.0
Epoch [6/50], Step [1/391], Loss: 1.1651
Epoch [6/50], Step [101/391], Loss: 1.0939
Epoch [6/50], Step [201/391], Loss: 1.0773
Epoch [6/50], Step [301/391], Loss: 1.0645
Epoch loss: 0.0
Epoch [7/50], Step [1/391], Loss: 1.0624
Epoch [7/50], Step [101/391], Loss: 1.0457
Epoch [7/50], Step [201/391], Loss: 0.9940
Epoch [7/50], Step [301/391], Loss: 0.9706
Epoch loss: 0.0
Epoch [8/50], Step [1/391], Loss: 0.8835
Epoch [8/50], Step [101/391], Loss: 0.9459
Epoch [8/50], Step [201/391], Loss: 0.9702
Epoch [8/50], Step [301/391], Loss: 0.9449
Epoch loss: 0.0
Epoch [9/50], Step [1/391], Loss: 0.9693
Epoch [9/50], Step [101/391], Loss: 0.8533
Epoch [9/50], Step [201/391], Loss: 1.0784
Epoch [9/50], Step [301/391], Loss: 1.0110
Epoch loss: 0.0
Epoch [10/50], Step [1/391], Loss: 0.8367
Epoch [10/50], Step [101/391], Loss: 0.8047
Epoch [10/50], Step [201/391], Loss: 0.9357
Epoch [10/50], Step [301/391], Loss: 0.9059
Epoch loss: 0.0
Epoch [11/50], Step [1/391], Loss: 0.8204
Epoch [11/50], Step [101/391], Loss: 0.9152
Epoch [11/50], Step [201/391], Loss: 0.8034
Epoch [11/50], Step [301/391], Loss: 0.9101
Epoch loss: 0.0
Epoch [12/50], Step [1/391], Loss: 0.9010
Epoch [12/50], Step [101/391], Loss: 0.8596
Epoch [12/50], Step [201/391], Loss: 0.6973
Epoch [12/50], Step [301/391], Loss: 0.8153
Epoch loss: 0.0
Epoch [13/50], Step [1/391], Loss: 0.7606
Epoch [13/50], Step [101/391], Loss: 0.7864
Epoch [13/50], Step [201/391], Loss: 0.6773
Epoch [13/50], Step [301/391], Loss: 0.6570
Epoch loss: 0.0
Epoch [14/50], Step [1/391], Loss: 0.5779
Epoch [14/50], Step [101/391], Loss: 0.5896
Epoch [14/50], Step [201/391], Loss: 0.6090
Epoch [14/50], Step [301/391], Loss: 0.8096
Epoch loss: 0.0
Epoch [15/50], Step [1/391], Loss: 0.6739
Epoch [15/50], Step [101/391], Loss: 0.6678
Epoch [15/50], Step [201/391], Loss: 0.7002
Epoch [15/50], Step [301/391], Loss: 0.5233
Epoch loss: 0.0
Epoch [16/50], Step [1/391], Loss: 0.8300
Epoch [16/50], Step [101/391], Loss: 0.7857
Epoch [16/50], Step [201/391], Loss: 0.6079
Epoch [16/50], Step [301/391], Loss: 0.6565
Epoch loss: 0.0
Epoch [17/50], Step [1/391], Loss: 0.5765
Epoch [17/50], Step [101/391], Loss: 0.5599
Epoch [17/50], Step [201/391], Loss: 0.5274
Epoch [17/50], Step [301/391], Loss: 0.5802
Epoch loss: 0.0
Epoch [18/50], Step [1/391], Loss: 0.5922
Epoch [18/50], Step [101/391], Loss: 0.5854
Epoch [18/50], Step [201/391], Loss: 0.6560
Epoch [18/50], Step [301/391], Loss: 0.5499
Epoch loss: 0.0
Epoch [19/50], Step [1/391], Loss: 0.5238
Epoch [19/50], Step [101/391], Loss: 0.5550
Epoch [19/50], Step [201/391], Loss: 0.4975
Epoch [19/50], Step [301/391], Loss: 0.5471
Epoch loss: 0.0
Epoch [20/50], Step [1/391], Loss: 0.5009
Epoch [20/50], Step [101/391], Loss: 0.5555
Epoch [20/50], Step [201/391], Loss: 0.5322
Epoch [20/50], Step [301/391], Loss: 0.6736
Epoch loss: 0.0
Epoch [21/50], Step [1/391], Loss: 0.4893
Epoch [21/50], Step [101/391], Loss: 0.6440
Epoch [21/50], Step [201/391], Loss: 0.5358
Epoch [21/50], Step [301/391], Loss: 0.3975
Epoch loss: 0.0
Epoch [22/50], Step [1/391], Loss: 0.4301
Epoch [22/50], Step [101/391], Loss: 0.5373
Epoch [22/50], Step [201/391], Loss: 0.5301
Epoch [22/50], Step [301/391], Loss: 0.5130
Epoch loss: 0.0
Epoch [23/50], Step [1/391], Loss: 0.5976
Epoch [23/50], Step [101/391], Loss: 0.5550
Epoch [23/50], Step [201/391], Loss: 0.5313
Epoch [23/50], Step [301/391], Loss: 0.4254
Epoch loss: 0.0
Epoch [24/50], Step [1/391], Loss: 0.4173
Epoch [24/50], Step [101/391], Loss: 0.4947
Epoch [24/50], Step [201/391], Loss: 0.5594
Epoch [24/50], Step [301/391], Loss: 0.3476
Epoch loss: 0.0
Epoch [25/50], Step [1/391], Loss: 0.4466
Epoch [25/50], Step [101/391], Loss: 0.4650
Epoch [25/50], Step [201/391], Loss: 0.5880
Epoch [25/50], Step [301/391], Loss: 0.5505
Epoch loss: 0.0
Epoch [26/50], Step [1/391], Loss: 0.5046
Epoch [26/50], Step [101/391], Loss: 0.3411
Epoch [26/50], Step [201/391], Loss: 0.3288
Epoch [26/50], Step [301/391], Loss: 0.4225
Epoch loss: 0.0
Epoch [27/50], Step [1/391], Loss: 0.3167
Epoch [27/50], Step [101/391], Loss: 0.4745
Epoch [27/50], Step [201/391], Loss: 0.4143
Epoch [27/50], Step [301/391], Loss: 0.4575
Epoch loss: 0.0
Epoch [28/50], Step [1/391], Loss: 0.4940
Epoch [28/50], Step [101/391], Loss: 0.4825
Epoch [28/50], Step [201/391], Loss: 0.5192
Epoch [28/50], Step [301/391], Loss: 0.3064
Epoch loss: 0.0
Epoch [29/50], Step [1/391], Loss: 0.4024
Epoch [29/50], Step [101/391], Loss: 0.3293
Epoch [29/50], Step [201/391], Loss: 0.4489
Epoch [29/50], Step [301/391], Loss: 0.5057
Epoch loss: 0.0
Epoch [30/50], Step [1/391], Loss: 0.4067
Epoch [30/50], Step [101/391], Loss: 0.5365
Epoch [30/50], Step [201/391], Loss: 0.4012
Epoch [30/50], Step [301/391], Loss: 0.4694
Epoch loss: 0.0
Epoch [31/50], Step [1/391], Loss: 0.4356
Epoch [31/50], Step [101/391], Loss: 0.2980
Epoch [31/50], Step [201/391], Loss: 0.2846
Epoch [31/50], Step [301/391], Loss: 0.4156
Epoch loss: 0.0
Epoch [32/50], Step [1/391], Loss: 0.3304
Epoch [32/50], Step [101/391], Loss: 0.4109
Epoch [32/50], Step [201/391], Loss: 0.4633
Epoch [32/50], Step [301/391], Loss: 0.4034
Epoch loss: 0.0
Epoch [33/50], Step [1/391], Loss: 0.3521
Epoch [33/50], Step [101/391], Loss: 0.4170
Epoch [33/50], Step [201/391], Loss: 0.3939
Epoch [33/50], Step [301/391], Loss: 0.4290
Epoch loss: 0.0
Epoch [34/50], Step [1/391], Loss: 0.3151
Epoch [34/50], Step [101/391], Loss: 0.3486
Epoch [34/50], Step [201/391], Loss: 0.4054
Epoch [34/50], Step [301/391], Loss: 0.4330
Epoch loss: 0.0
Epoch [35/50], Step [1/391], Loss: 0.2671
Epoch [35/50], Step [101/391], Loss: 0.3478
Epoch [35/50], Step [201/391], Loss: 0.3083
Epoch [35/50], Step [301/391], Loss: 0.2792
Epoch loss: 0.0
Epoch [36/50], Step [1/391], Loss: 0.3927
Epoch [36/50], Step [101/391], Loss: 0.3938
Epoch [36/50], Step [201/391], Loss: 0.3753
Epoch [36/50], Step [301/391], Loss: 0.3289
Epoch loss: 0.0
Epoch [37/50], Step [1/391], Loss: 0.3327
Epoch [37/50], Step [101/391], Loss: 0.2782
Epoch [37/50], Step [201/391], Loss: 0.4179
Epoch [37/50], Step [301/391], Loss: 0.3101
Epoch loss: 0.0
Epoch [38/50], Step [1/391], Loss: 0.3250
Epoch [38/50], Step [101/391], Loss: 0.4541
Epoch [38/50], Step [201/391], Loss: 0.3375
Epoch [38/50], Step [301/391], Loss: 0.3090
Epoch loss: 0.0
Epoch [39/50], Step [1/391], Loss: 0.4108
Epoch [39/50], Step [101/391], Loss: 0.4024
Epoch [39/50], Step [201/391], Loss: 0.3012
Epoch [39/50], Step [301/391], Loss: 0.4336
Epoch loss: 0.0
Epoch [40/50], Step [1/391], Loss: 0.3891
Epoch [40/50], Step [101/391], Loss: 0.3160
Epoch [40/50], Step [201/391], Loss: 0.2615
Epoch [40/50], Step [301/391], Loss: 0.3220
Epoch loss: 0.0
Epoch [41/50], Step [1/391], Loss: 0.2787
Epoch [41/50], Step [101/391], Loss: 0.3155
Epoch [41/50], Step [201/391], Loss: 0.2322
Epoch [41/50], Step [301/391], Loss: 0.2715
Epoch loss: 0.0
Epoch [42/50], Step [1/391], Loss: 0.2706
Epoch [42/50], Step [101/391], Loss: 0.3163
Epoch [42/50], Step [201/391], Loss: 0.3272
Epoch [42/50], Step [301/391], Loss: 0.2988
Epoch loss: 0.0
Epoch [43/50], Step [1/391], Loss: 0.2124
Epoch [43/50], Step [101/391], Loss: 0.2882
Epoch [43/50], Step [201/391], Loss: 0.3724
Epoch [43/50], Step [301/391], Loss: 0.2389
Epoch loss: 0.0
Epoch [44/50], Step [1/391], Loss: 0.2687
Epoch [44/50], Step [101/391], Loss: 0.2373
Epoch [44/50], Step [201/391], Loss: 0.2818
Epoch [44/50], Step [301/391], Loss: 0.2343
Epoch loss: 0.0
Epoch [45/50], Step [1/391], Loss: 0.2131
Epoch [45/50], Step [101/391], Loss: 0.2127
Epoch [45/50], Step [201/391], Loss: 0.2431
Epoch [45/50], Step [301/391], Loss: 0.2389
Epoch loss: 0.0
Epoch [46/50], Step [1/391], Loss: 0.3001
Epoch [46/50], Step [101/391], Loss: 0.1981
Epoch [46/50], Step [201/391], Loss: 0.2866
Epoch [46/50], Step [301/391], Loss: 0.3703
Epoch loss: 0.0
Epoch [47/50], Step [1/391], Loss: 0.1602
Epoch [47/50], Step [101/391], Loss: 0.2415
Epoch [47/50], Step [201/391], Loss: 0.2196
Epoch [47/50], Step [301/391], Loss: 0.3137
Epoch loss: 0.0
Epoch [48/50], Step [1/391], Loss: 0.2530
Epoch [48/50], Step [101/391], Loss: 0.2034
Epoch [48/50], Step [201/391], Loss: 0.2272
Epoch [48/50], Step [301/391], Loss: 0.2120
Epoch loss: 0.0
Epoch [49/50], Step [1/391], Loss: 0.1802
Epoch [49/50], Step [101/391], Loss: 0.1750
Epoch [49/50], Step [201/391], Loss: 0.1829
Epoch [49/50], Step [301/391], Loss: 0.2795
Epoch loss: 0.0
Epoch [50/50], Step [1/391], Loss: 0.1472
Epoch [50/50], Step [101/391], Loss: 0.2639
Epoch [50/50], Step [201/391], Loss: 0.1830
Epoch [50/50], Step [301/391], Loss: 0.2024
Epoch loss: 0.0
Accuracy of the network on the test images: 87.97 %
```

```
[166]: t_array_val = [391 * x for x in range(0, epochs)]
       t_array_test = np.arange(0, 391*epochs)
```

```
[167]: print(len(training_loss))
       print(len(validation_loss))
```

```
19550
50
```

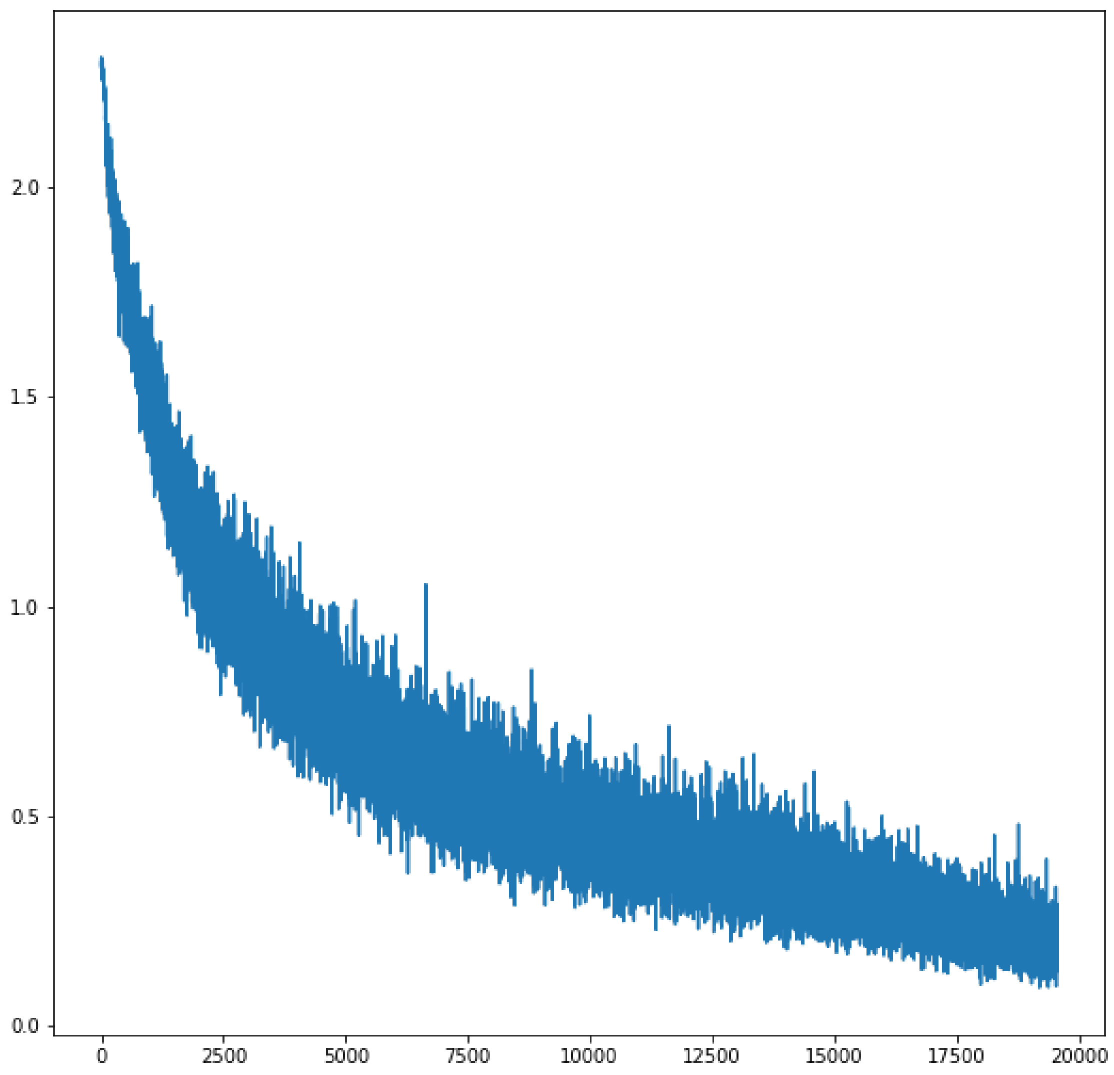
```
[168]: print(len(training_error))
       print(len(validation_error))
```

```
19550
50
```

```
[175]: plt.rcParams["figure.figsize"] = (10,10)
```

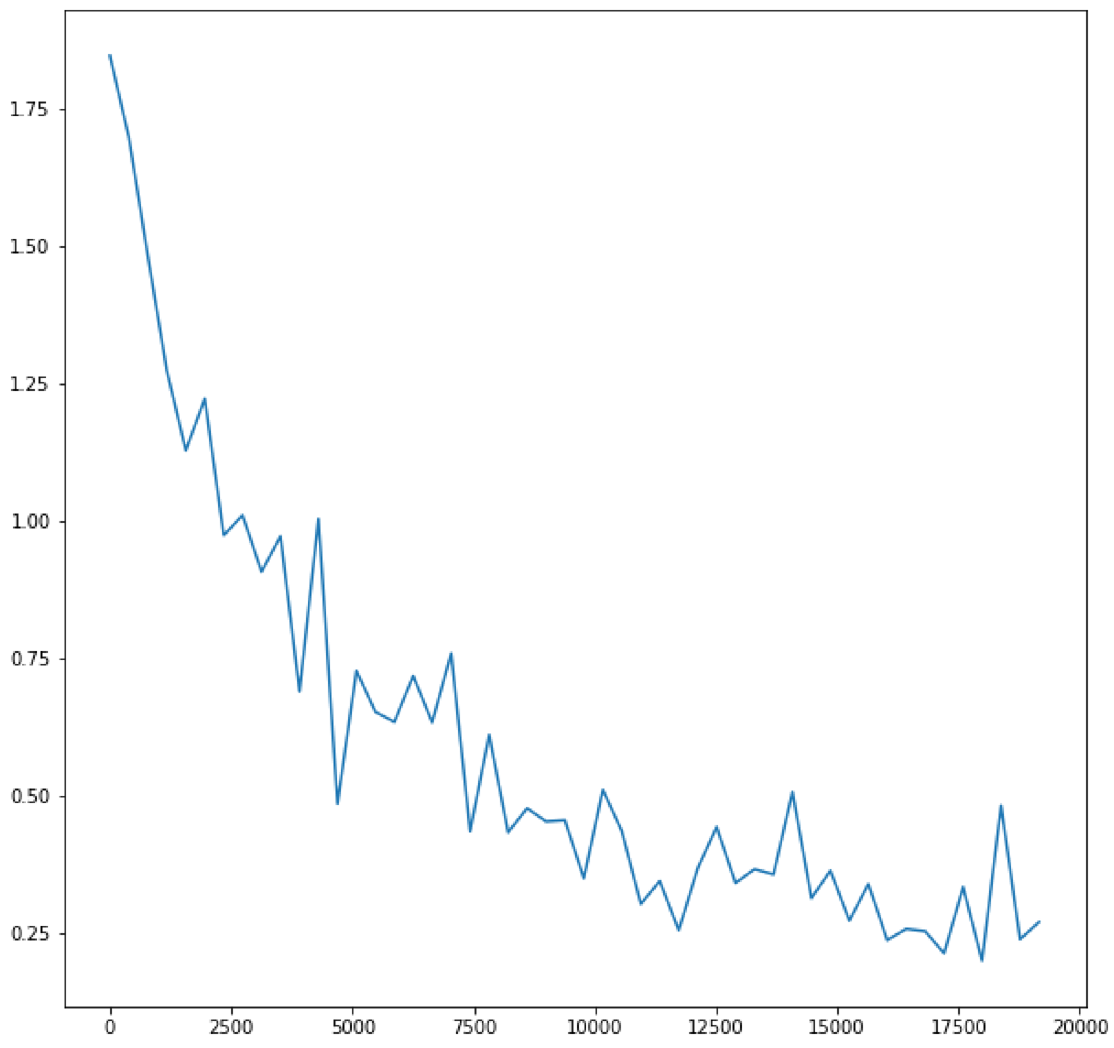
```
[176]: plt.plot(t_array_test,training_loss)
```

```
[176]: [<matplotlib.lines.Line2D at 0x7f390f819438>]
```



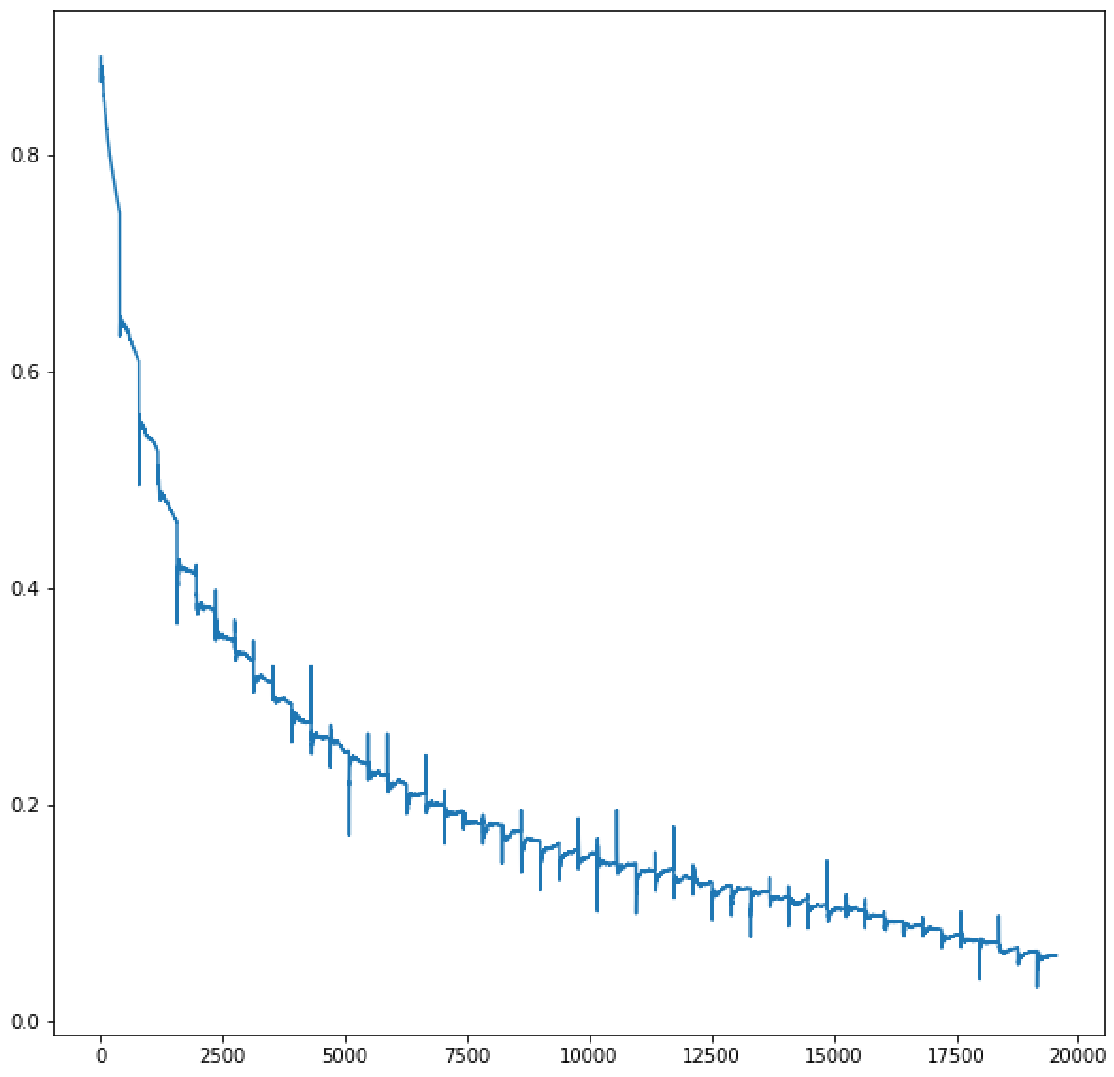
```
[177]: plt.plot(t_array_val, validation_loss)
```

```
[177]: [<matplotlib.lines.Line2D at 0x7f390f7773c8>]
```



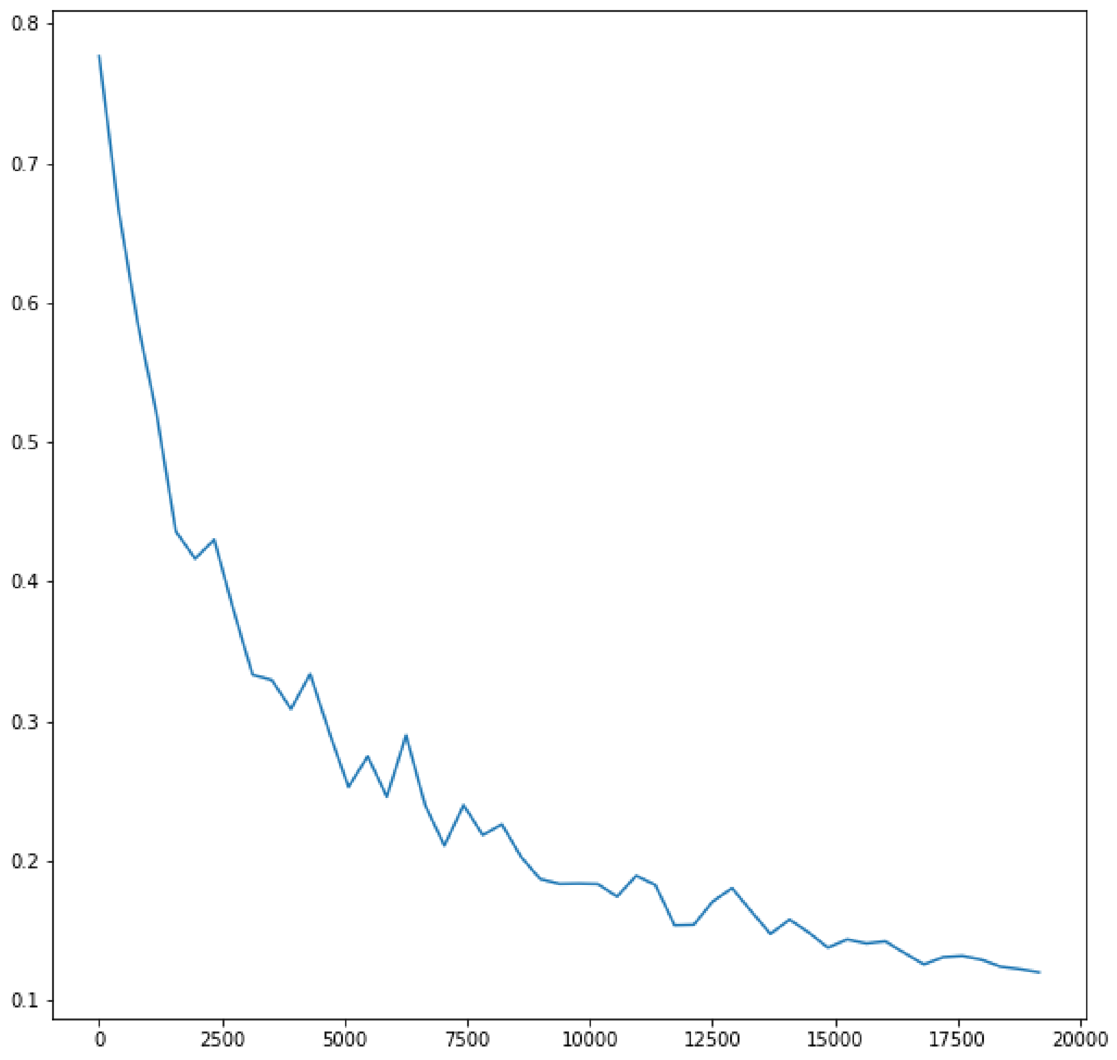
```
[178]: plt.plot(t_array_test, training_error)
```

```
[178]: [<matplotlib.lines.Line2D at 0x7f390f6611d0>]
```



```
[179]: plt.plot(t_array_val, validation_error)
```

```
[179]: [<matplotlib.lines.Line2D at 0x7f390f63f978>]
```



0.2 Part (d)

```
[180]: # The training loop
def train_scenario(net, optimizer, criterion, train_loader, test_loader,␣
 ↪epochs, max_eta):
    model = net.to(device)
    total_step = len(train_loader)

    overall_step = 0
    t = 0

    for epoch in range(epochs):
        correct = 0
        total = 0
        epoch_loss = 0

        model.train()

        for i, (images, labels) in enumerate(train_loader):

            # update learning rate
            if t < T0:
                new_lr = 1e-4 + (t/T0)*max_eta
            else:
                new_lr = max_eta*np.cos((np.pi/2)*(t - T0)/(T - T0)) + 1e-6

            for g in optimizer.param_groups:
                g['lr'] = new_lr

            # Move tensors to configured device
            images = images.to(device)
            labels = labels.to(device)

            # Forward Pass
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
            loss = criterion(outputs, labels)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            t += 1

        epoch_loss /= len(train_loader)

    model.eval()
    with torch.no_grad():
        correct = 0
        total = 0
        for i, (images, labels) in enumerate(test_loader):
            images = images.to(device)
            labels = labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        validation_loss.append(loss.item())
        validation_error.append(1 - correct / total)
    print('Validation Error: {} %'.format(100 * (1 - correct / total)))
    print('Accuracy of the network on the test images: {} %'.format(100 *␣
 ↪correct / total))
    return
```

[181]:
```
eta_max_arr = [eta_max, 5 * eta_max, eta_max]
rho = [0.9, 0.5, 0.5]
validation_error = []
validation_loss = []
```

[182]:
```
epochs = 50

for i in range(0, 3):
    model = allcnn_t().to(device)

    #Define the loss function as asked in the question
    criterion = nn.CrossEntropyLoss()

    #Set parameters as stated in the question
    optimizer = optim.SGD(model.parameters(), lr=eta, momentum=rho[i],␣
 ↪weight_decay=0.001)

    # Training loop called here
    train_scenario(model, optimizer, criterion, trainloader, testloader,␣
 ↪epochs, eta_max_arr[i])
```

```
Num parameters:  1667166
Validation Error: 12.819999999999999 %
Accuracy of the network on the test images: 87.18 %
Num parameters:  1667166
Validation Error: 11.619999999999997 %
Accuracy of the network on the test images: 88.38 %
Num parameters:  1667166
Validation Error: 20.540000000000003 %
Accuracy of the network on the test images: 79.46 %
```

0.2.1 We can see that the validation error increased from 12% to 21% for the third case, where the ration of eta-max and 1 - rho was not maintained