NN_Partition_Problem

October 1, 2020

1 MLP for Partition Problem

Edison Gu

We first set up the environment:

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import time
```

Construct the features and target values for this partition problem:

[2]:
```
feature = np.array([],dtype=int)
for i in range(2):
    for j in range(2):
        for m in range(2):
            for n in range(2):
                feature = np.append(feature,[i,j,m,n])

feature = np.reshape(feature,(16,4))

target = np.array([[s%2 for s in feature.sum(axis=1)]]).T
```

[3]:
```
print(feature)
print()
print(target)
```

```
[[0 0 0 0]
 [0 0 0 1]
 [0 0 1 0]
 [0 0 1 1]
 [0 1 0 0]
 [0 1 0 1]
 [0 1 1 0]
 [0 1 1 1]
 [1 0 0 0]
 [1 0 0 1]
 [1 0 1 0]
 [1 0 1 1]
```

```
 [1 1 0 0]
 [1 1 0 1]
 [1 1 1 0]
 [1 1 1 1]]

[[0]
 [1]
 [1]
 [0]
 [1]
 [0]
 [0]
 [1]
 [1]
 [0]
 [0]
 [1]
 [0]
 [1]
 [1]
 [0]]
```

Create a class for a single layer, call it "Dense":

[4]:
```
class Dense:
    def __init__(self, n_output_nodes):
        self.n_output = n_output_nodes

    def build(self, n_input):
        self.weight = np.random.uniform(
            low=-1, high=1, size=(n_input + 1, self.n_output))
        self.output = np.zeros((1, self.n_output))
        self.prev_wght_update = np.zeros((n_input + 1, self.n_output))

    def call(self, x):
        self.input = np.append(x, 1)
        v = np.matmul(self.input, self.weight)
        self.output = 1 / (1 + np.exp(-v))

        return self.output
```

[5]:
```
np.random.seed(2020)
layer = Dense(4)
layer.build(4)
layer.call(feature[1])
```

[5]:
```
array([0.4767296 , 0.45863976, 0.25550647, 0.49327017])
```

Create a class for the MLP model:

[6]:
```
class Multilayers:
    def __init__(self, x_shape, layers):
        self.model = layers
        self.n_layers = len(layers)

        n_input = x_shape
        for layer in self.model:
            layer.build(n_input)
            n_input = layer.n_output

    def predict(self, x):

        input_signal = x
        for i, layer in zip(range(self.n_layers), self.model):
            output_signal = layer.call(input_signal)

            input_signal = output_signal

        self.y = output_signal
        return self.y

    def back_prop(self, target, learn_rate, momentum):
        prev_update = 0

        i = 0
        for layer in reversed(self.model):

            prime = layer.output * (1 - layer.output)

            if i == 0:
                deltas = (target - self.y) * prime
            else:
                deltas = prime * np.inner(deltas, prev_weight[:-1])
            i += 1
            prev_weight = np.copy(layer.weight)

            update = learn_rate * np.outer(layer.input, deltas)

            layer.weight += momentum * layer.prev_wght_update + update

            layer.prev_wght_update = update
```

[7]:
```
np.random.seed(2020)

model = Multilayers(4,[
    Dense(4),
```

```
    Dense(1)
])

print('Before back prop:', model.predict(feature[0]))
model.back_prop(target[0], learn_rate=0.1, momentum=0.9)
print('After 1 back prop:', model.predict(feature[0]))
model.back_prop(target[0], learn_rate=0.1, momentum=0.9)
print('After 2 back prop:', model.predict(feature[0]))
model.back_prop(target[0], learn_rate=0.1, momentum=0.9)
print('After 3 back prop:', model.predict(feature[0]))
```

```
Before back prop: [0.59519486]
After 1 back prop: [0.58891059]
After 2 back prop: [0.5769365]
After 3 back prop: [0.56499567]
```

The function below will present the whole training set to the MLP once - one epoch - and returns the error from the network:

[8]:
```
def train(model, feature, target, learn_rate=0.1, momentum=0):
    epoch_size = len(feature)
    order = np.random.choice(range(epoch_size),size=epoch_size, replace=False)
    error = np.array([])
    for i in order:
        output = model.predict(feature[i])
        model.back_prop(target[i], learn_rate, momentum)
        error = np.append(error, target[i] - output)
        absolute_error = np.absolute(error)
    return absolute_error.max(), absolute_error.mean()
```

Set up some parameters for our specific problem:

[14]:
```
tolerance = 0.05
max_iteration = 1E7
periodic_output = 5E4
plt.rcParams['figure.figsize'] = (16,6)
```

Below we will perform the MLP training with our dataset each with a different η for learning rate and a different α for momentum constant:

From the output below, the **red** curve indicate the maximum error for each epoch, and the **blue** curve indicates the average error for each epoch. We see that -

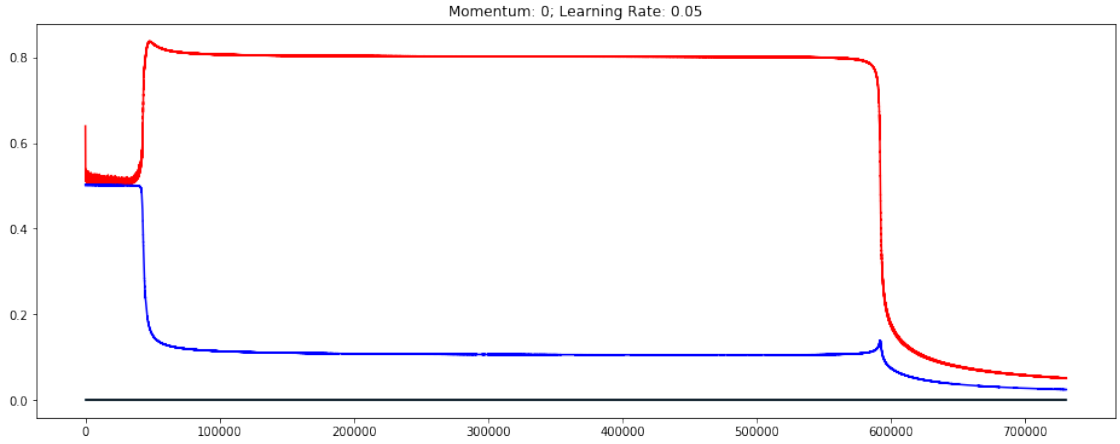
- Without the momentum constant, the learning time is comparatively longer - they usually start to converge after 500K ~ 900K epochs.
- With the momentum constant, the learning speed is faster - they converge under 90K epochs.
- For a given momentum constant, zero to exclude the momentum term or some other constant (0.9 here), the learning time needed gets shorter, i.e. the algorithm completes in less epochs, as the learning rate increases.

- This makes sense because the learning rate controls … the rate of learning … so a larger learning rate will lead to a faster learning.
- However, notice that maximum error oscillate a lot more for higher learning rate - shown as a wider red "ribbon" on the graph. This indicates that for a higher learning rate, the learning path on the error surface will possibly be zigzagging since it is takign larger steps.

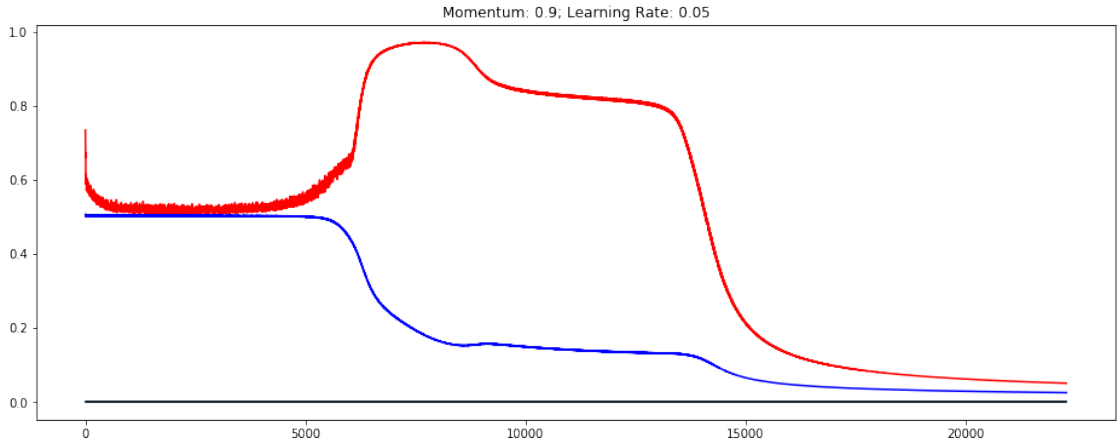
[10]:
```
for learn_rate in np.arange(0.05, 0.55, 0.05):
    for momentum in (0,0.9):
        print(f'Momentum: {momentum}; Learning Rate: {learn_rate:2}')
        print()
        model = Multilayers(4,[
            Dense(4),
            Dense(1)
        ])

        max_error, avg_error = 1, 1
        history_max, history_avg = [], []

        start_time = time.time()
        i = 0
        while max_error > tolerance and i < max_iteration:
            max_error, avg_error = train(model, feature, target, learn_rate,␣
↪momentum)
            history_max, history_avg = np.append(history_max, max_error), np.
↪append(history_avg, avg_error)

            if i % periodic_output == 0:
                print(i, max_error, avg_error)

            i += 1

        print("--- %4s seconds ---" % (time.time() - start_time))
        print(i, max_error, avg_error)

        plt.title(f'Momentum: {momentum}; Learning Rate: {learn_rate:2}')
        plt.plot(history_max, color='red')
        plt.plot([0, len(history_max)],[0,0])
        plt.plot(history_avg, color='blue')
        plt.plot([0, len(history_avg)],[0,0], color='black')
        plt.show()
        print()
```

```
Momentum: 0; Learning Rate: 0.05

0 0.6396836067248847 0.5030100311871979
50000 0.8334371032769167 0.15141498264231001
100000 0.8061233041766372 0.11375868135042304
```

```
150000 0.8045894292842817 0.10952156199024587
200000 0.8026248831351228 0.10783246533623486
250000 0.8030786591886758 0.10657993554346595
300000 0.8017016092450687 0.10601103718135589
350000 0.8019553247555666 0.105392271906193
400000 0.8021954624351755 0.10491660574703207
450000 0.8017778508774063 0.10468051970586642
500000 0.8007839649508726 0.10476082534768115
550000 0.7997920230297448 0.10549911819386482
600000 0.17754147430738887 0.07370927682752283
650000 0.0782765665818442 0.036188851671067904
700000 0.057215604764919714 0.02697826761771399
--- 1775.3887729644775 seconds ---
730508 0.04996678635748251 0.023901312654863102
```

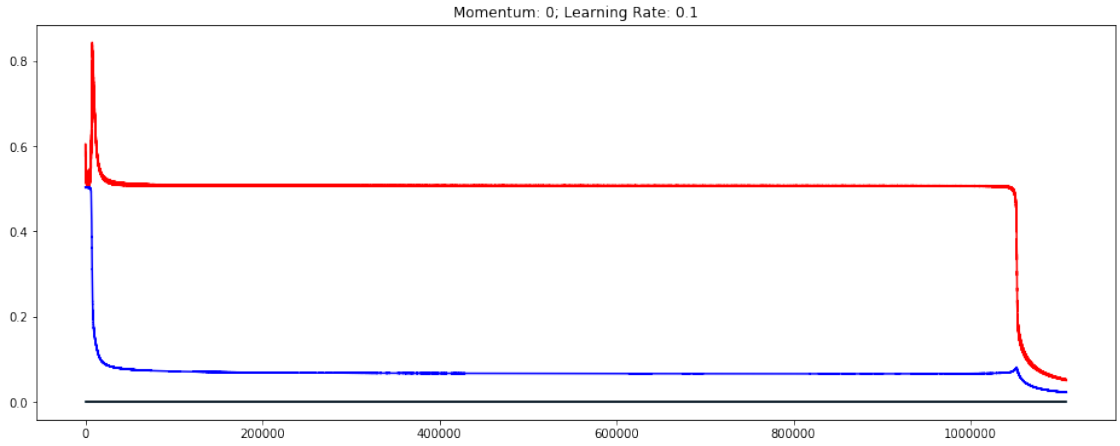



```
Momentum: 0.9; Learning Rate: 0.05

0 0.7340206558390946 0.5062517483409308
--- 27.42765212059021 seconds ---
22301 0.049998545264748014 0.024480236336623898
```

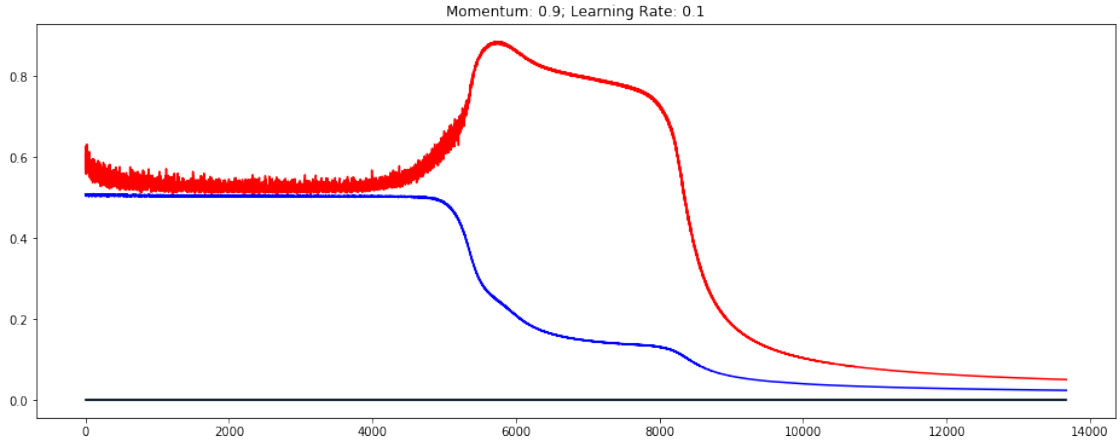


Momentum: 0; Learning Rate: 0.1

0 0.577176566537737 0.5037285789545161
50000 0.5114292210134999 0.07582145861502929
100000 0.5080309424356766 0.07118003285372818
150000 0.5077668419372441 0.06942285419361051
200000 0.506568296400423 0.06849220902670324
250000 0.5067031499584468 0.06782989849198637
300000 0.5069991884188938 0.06738192073171306
350000 0.5069086041654873 0.06699706215931359
400000 0.5068768779795692 0.06670516262115289
450000 0.5067549565848224 0.06647425099051542
500000 0.5057301084816361 0.0662756042094454
550000 0.5062927946460455 0.06609664199842345
600000 0.5062447375754889 0.06595986128971479
650000 0.5061947374329219 0.06583504500062844
700000 0.5060570756935087 0.06575003075037804
750000 0.5061422754992355 0.06564721278472083
800000 0.5056932499189074 0.06557791701391084
850000 0.506472027106023 0.06552616103889476
900000 0.5070851989215054 0.06554552615458151
950000 0.5069021354723938 0.06564610835961375
1000000 0.5064514429417155 0.06608119870573055
1050000 0.489387262541606 0.0747057665687503
1100000 0.054321671579892206 0.02353740962859994
--- 3396.8432908058167 seconds ---
1108152 0.049961639676865066 0.021935454772083786

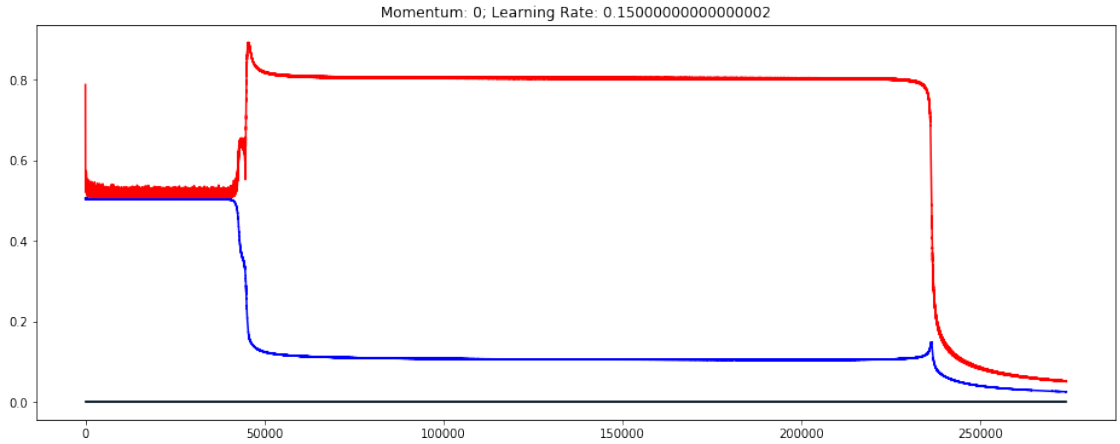



```
Momentum: 0.9; Learning Rate: 0.1

0 0.6071568213470728 0.5083488462480137
--- 17.861434936523438 seconds ---
13668 0.0499871001743468 0.02341148204582721
```



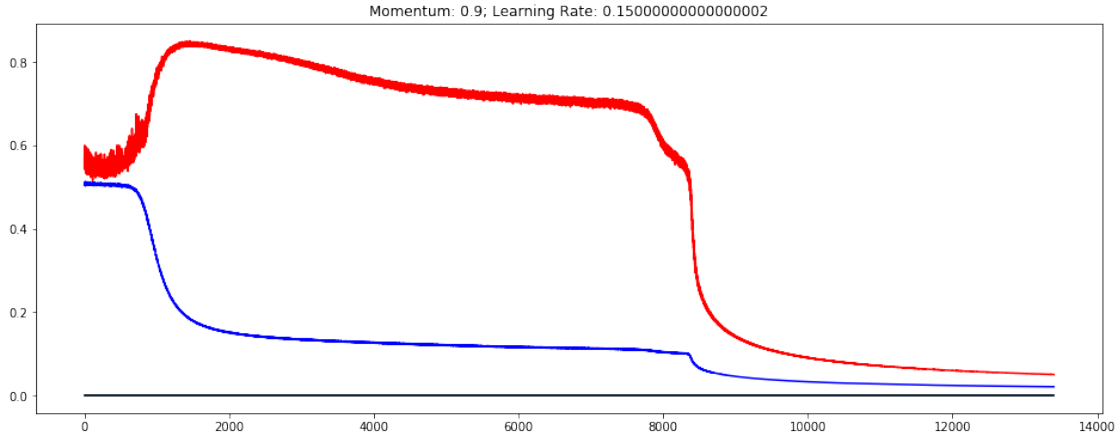

```
Momentum: 0; Learning Rate: 0.15000000000000002

0 0.7873311571331825 0.5051048950119819
50000 0.817131208438092 0.12394666838623619
100000 0.802558997202012 0.10687149494450468
150000 0.8048053596459898 0.10441685980076543
200000 0.8035244131695876 0.10384585316807815
250000 0.08133576228023466 0.03944588257397333
--- 442.8886218070984 seconds ---
```

274109 0.04988443581908519 0.024702013833760483



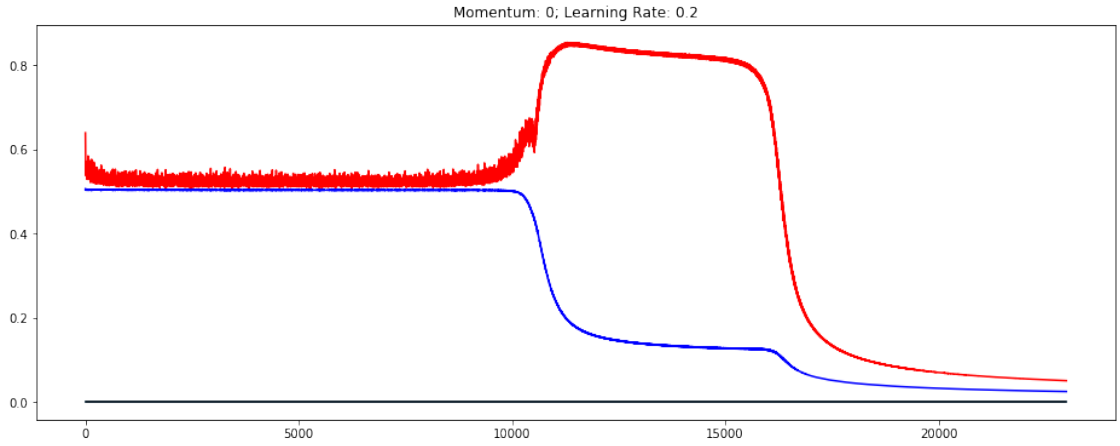
Momentum: 0.9; Learning Rate: 0.15000000000000002

0 0.5708238436464796 0.504743333627547
--- 17.50285577774048 seconds ---
13400 0.04995939574145025 0.020634621508313692

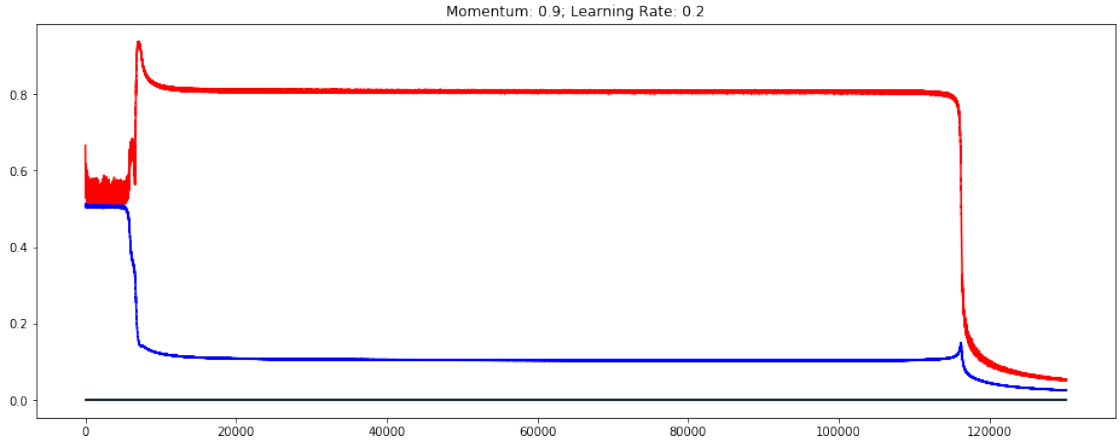


Momentum: 0; Learning Rate: 0.2

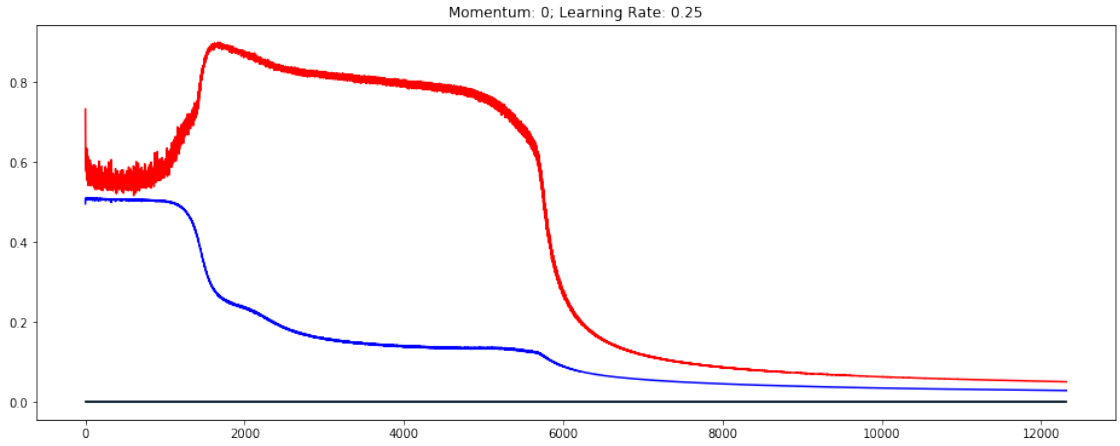
0 0.6392543297523647 0.5048873149088351
--- 30.291335821151733 seconds ---
22997 0.0499887093894899 0.024216779798973177




```
Momentum: 0.9; Learning Rate: 0.2

0 0.6661270219115218 0.5089627775940464
50000 0.8061561612562431 0.10383827552693822
100000 0.8065256151613992 0.10278848160854072
--- 179.8628749847412 seconds ---
130206 0.04964045434875282 0.02509567462688083
```

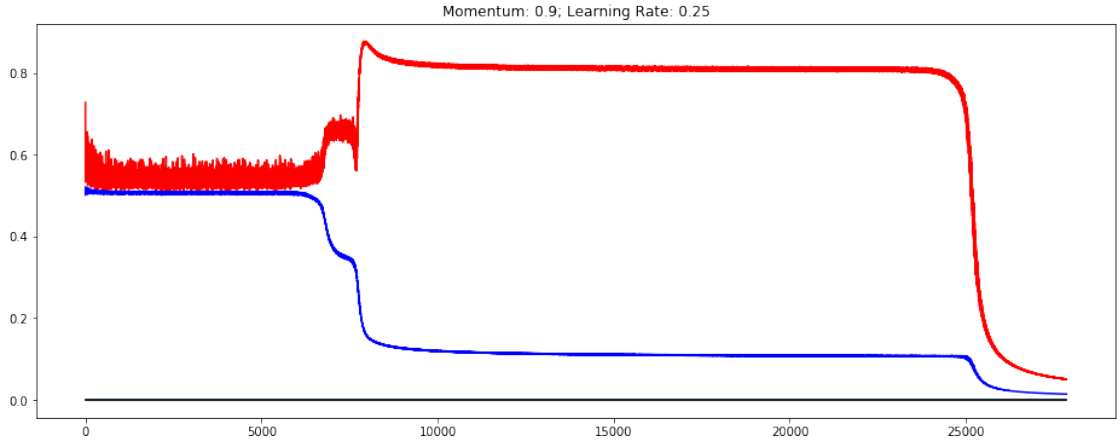



```
Momentum: 0; Learning Rate: 0.25

0 0.7321589568341965 0.4951871201555509
--- 16.262065887451172 seconds ---
12316 0.049941162782019495 0.027941647944260985
```



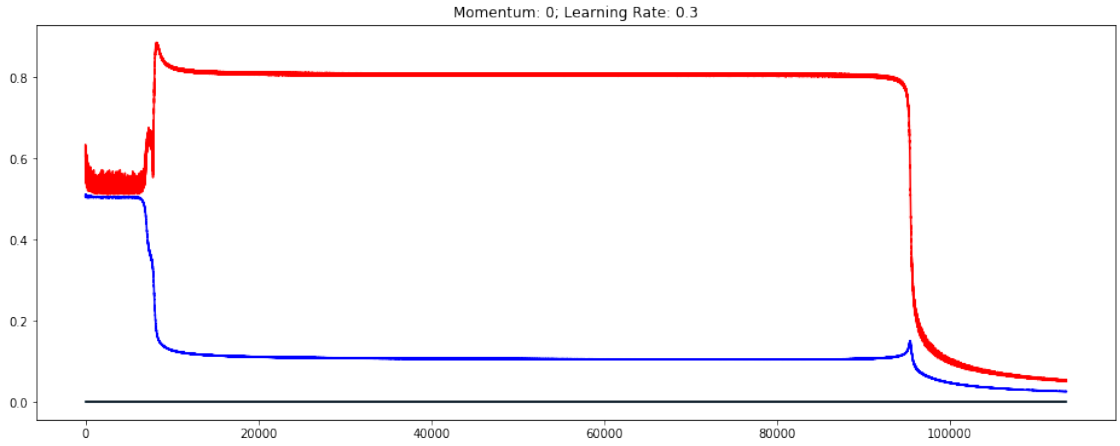
Momentum: 0.9; Learning Rate: 0.25

0 0.7277396223698753 0.501243904706183
--- 38.0995819568634 seconds ---
27851 0.04987685295733564 0.013993050239601932

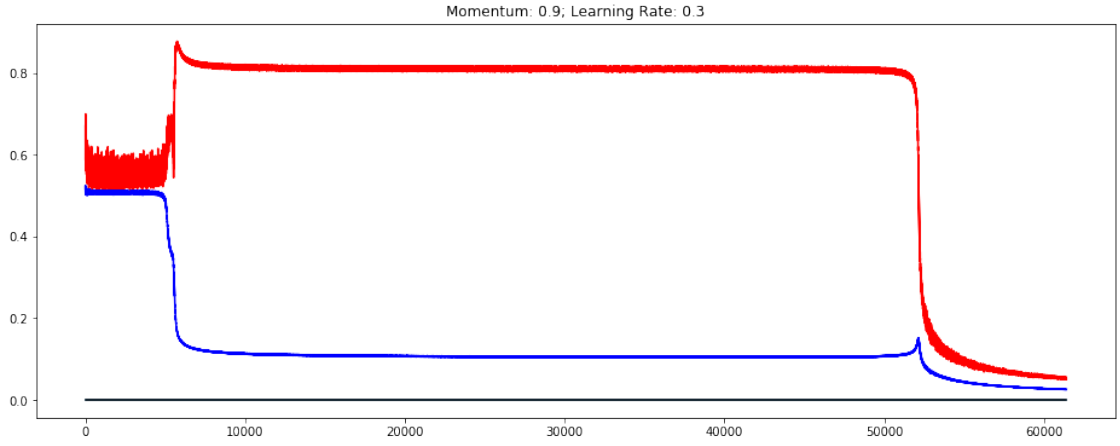


Momentum: 0; Learning Rate: 0.3

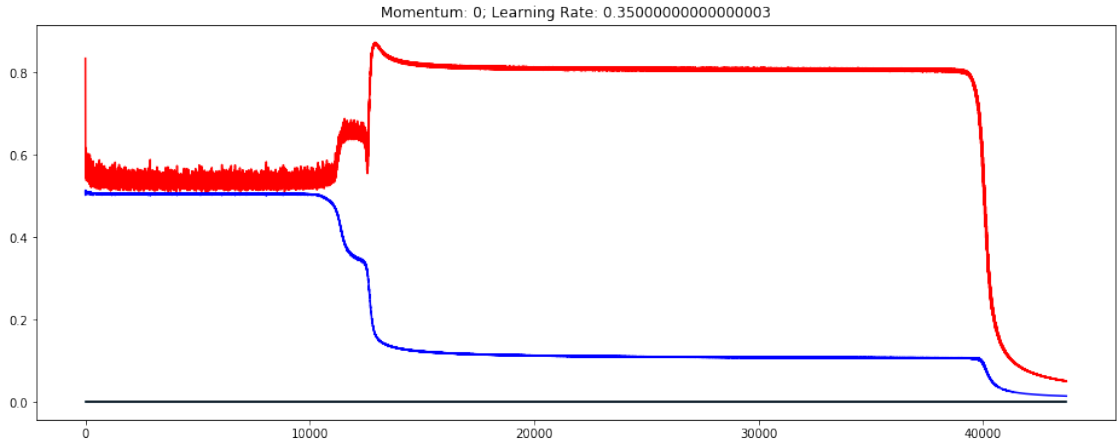
0 0.6045427730706887 0.5067085658843403
50000 0.8087199539836691 0.10485107214449851
100000 0.09845395522366907 0.04636640834248609
--- 155.83437085151672 seconds ---
113454 0.049876853722986954 0.025225274161191618



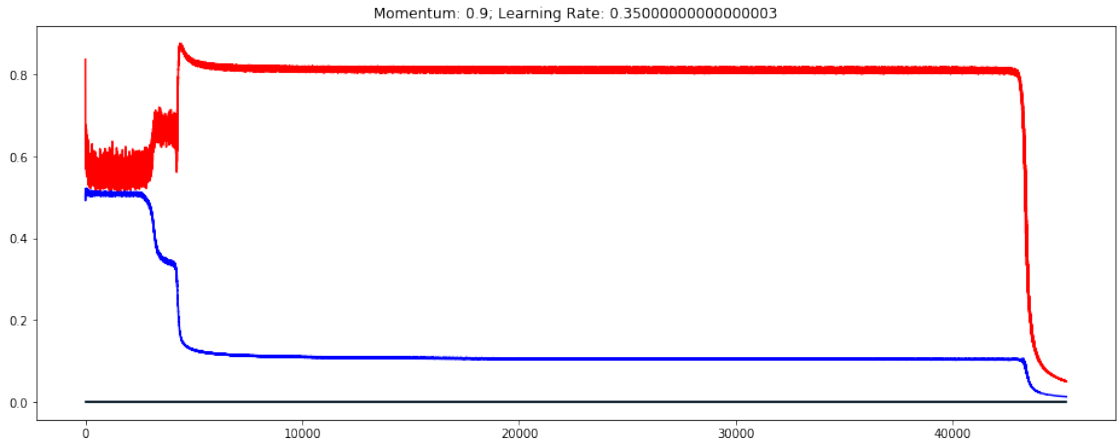

```
Momentum: 0.9; Learning Rate: 0.3

0 0.6699263198101857 0.5165190814440226
50000 0.8024046319097002 0.10686095129335846
--- 80.81418180465698 seconds ---
61394 0.049895832948933815 0.025442957930866164
```



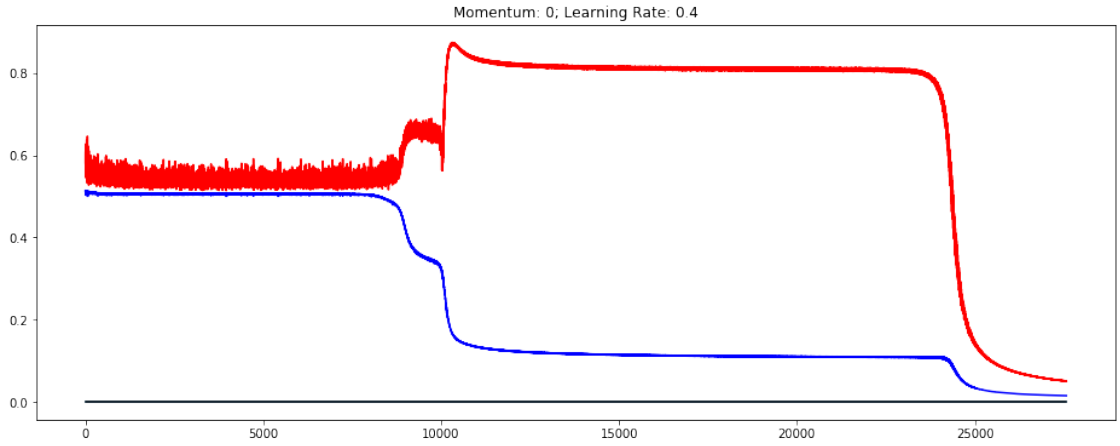

```
Momentum: 0; Learning Rate: 0.35000000000000003

0 0.8344895048661326 0.5093939412272426
--- 57.50732684135437 seconds ---
43715 0.049971118357503064 0.013696644994750967
```



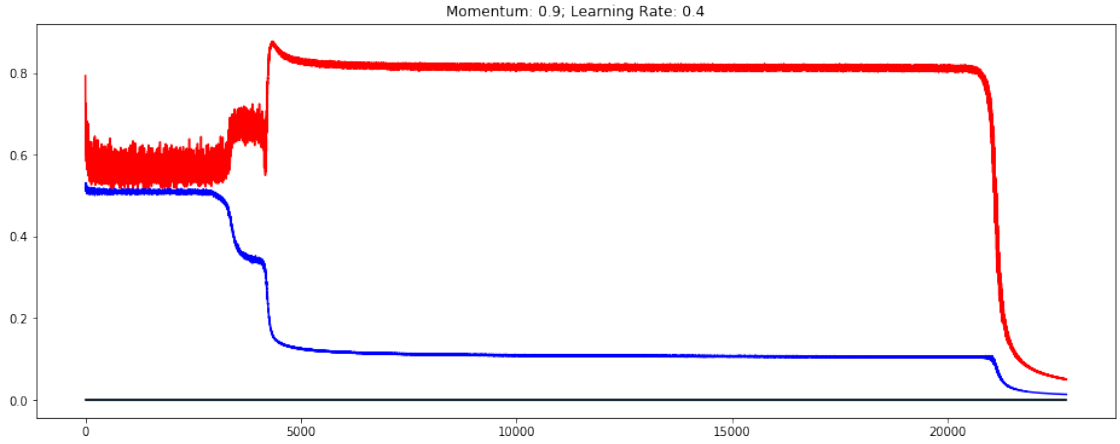

```
Momentum: 0.9; Learning Rate: 0.35000000000000003

0 0.8375273848105261 0.5063243136856783
--- 59.21809196472168 seconds ---
45238 0.049915817727090264 0.0126044323843701
```



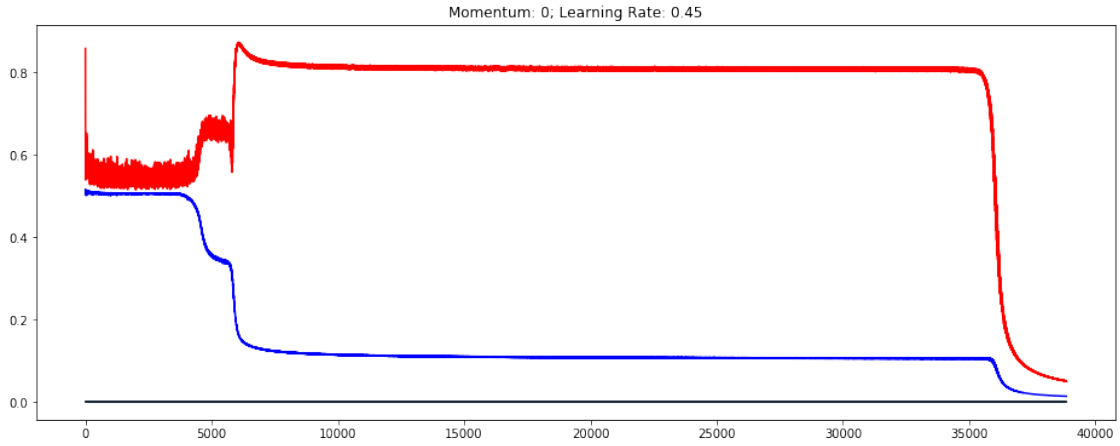

```
Momentum: 0; Learning Rate: 0.4

0 0.5813235969309379 0.5133570188975464
--- 36.43797039985657 seconds ---
27572 0.0499673319729086 0.014688106562906207
```




```
Momentum: 0.9; Learning Rate: 0.4

0 0.7934851281018227 0.5183743554601006
--- 29.585691213607788 seconds ---
22761 0.04992923903155298 0.013280297620511484
```

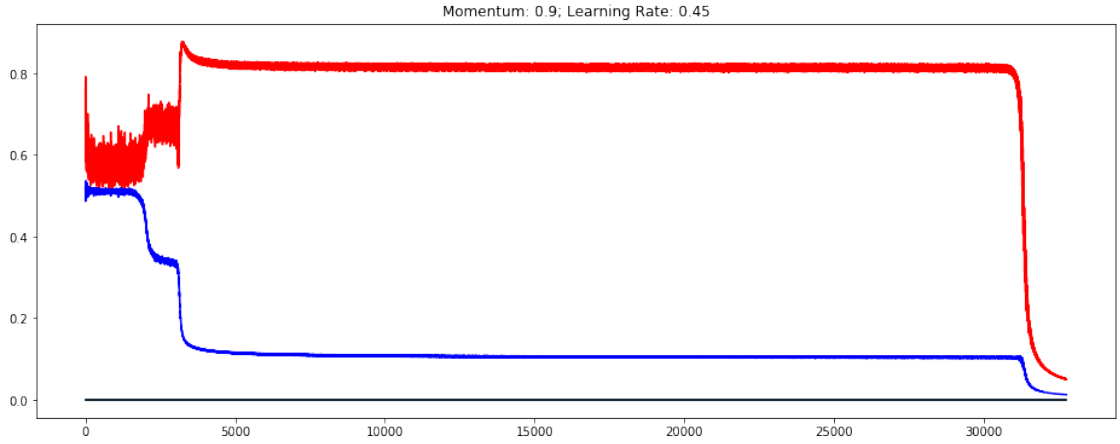



```
Momentum: 0; Learning Rate: 0.45

0 0.8584754473963924 0.5032764564763665
--- 51.66105318069458 seconds ---
38844 0.04998531307560557 0.013240740934174669
```



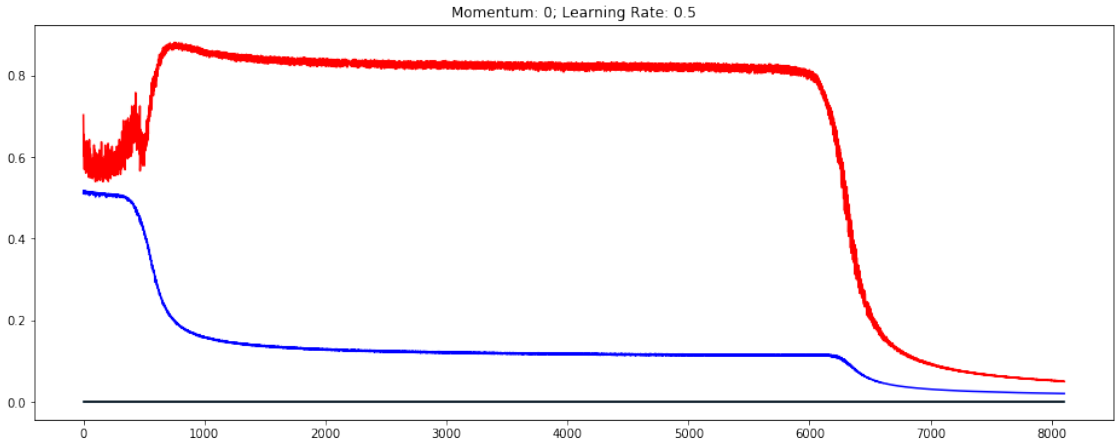
Momentum: 0.9; Learning Rate: 0.45

0 0.7635605601024634 0.5187750826016013
--- 42.864778995513916 seconds ---
32755 0.04988396015110437 0.0127179350173801



Momentum: 0; Learning Rate: 0.5

0 0.7034935670196887 0.5123311524523677
--- 10.703325748443604 seconds ---
8105 0.04994689164619359 0.019974334986965388



Momentum: 0.9; Learning Rate: 0.5

0 0.8085384895015142 0.5492445364481718
--- 33.69790816307068 seconds ---
25802 0.049602190943464006 0.012828758151698358