Q1

September 25, 2018

0.1 Question 1: Implementation of basic neural netwrok without any hidden layer.

(a) Generate some random 1D test data according to a simple linear function, with Gaussian noise added. For example, your data might be generated with: y = 7x + 3 + , where is a Gaussian random variable. Include a plot showing the training data and the function that your network computes. (You can plot the function by evaluating it on a range of different inputs). This is all 1D, so easy to visualize.

Sol: Here we implement designing a neural network which will take arbitary number of inputs for arbitary dimensions. This network has no hidden layers. #### Loss Function Regression loss function as given in the question.

Input Inputs are generated randomly by setting the number of training inputs and the dimensions.

Outputs The oputputs are limited to a single output. If the there is an n dimensional input, output is a function of these n inputs to produce a single value. (Here I have used summation to add the input for n dimensions to generate a single output). I have used the function y = 7x + 3 + to produce all the outputs.

Hyper Paramters For this function there are only two Hyper paramters, that is the **Learning Rate** (Step Size) and the number of **epochs**

Implicit Hyper paramters: weights and bias are initialized to a random value.

The Netwrok has 5 main Functions

1) Network Initialization
2) feed_forward
3) back_prop
4) update_network
5) train

Now lets got thorugh each function

```
In [2]: ## Initialize the Network
        def initialize_network(n_inputs, dimensions):
            x= np.random.rand(n_inputs,dimensions) ## Randomly Genrated Inputs
            y= 7*x + 3 + np.random.randn(1)        ## Output function of x
```

```
    y = np.sum(y,axis=1,keepdims= True)     ## To produce a single output
    wts= np.random.randn(dimensions,1)      ## Weight Initialization
    b =  np.random.randn(1)                 ## Bias Initialization
    return x,y,wts,b

## Feed Forward Function
def feed_forward(x,wts,b):
    a1= np.dot(x,wts)+b                      ## calculating the Output
    return a1

## Back Propogation
def back_prop(x,y,a1):
    error = (((y-a1)**2).sum())              ## Calculating the error
    wt_err = -x.T.dot(y-a1)                  ## Calculating Weight error by differentiati
    b_err = (a1-y).sum()                     ## Claculating Bias Error by differentiating
    return wt_err, b_err, error

## Updating the Network
def update_network(wts, b, wt_err, b_err,lr):
    wts -= lr*wt_err                         ## Updating weights
    b -= lr*b_err                            ## Updating b
    return wts,b

## Training the Network
def train(net,epochs,lr):
    error = []
    wt = net[2]
    b_= net[3]
    for i in range(epochs):
        a1 = feed_forward(net[0],wt,b_)
        err= back_prop(net[0], net[1],a1)
        error.append(err[2])
        update = update_network(net[2], net[3], err[0], err[1],lr)
        wt = update[0]                       ## Weight Update
        b_ = update [1]                      ## Bias Update
    return error, update[0], update[1]
```

0.2 (a) Defining the Main Function and Implementing for 1D input

In [62]:
```
# Importing the Libraries
import numpy as np
from matplotlib import pyplot as plt
from math import exp

# Initializing the Network
n_inputs = 1000
net=initialize_network(n_inputs,1)
```

```
## Hyper Parameter Initialization
epochs= 100      ## Step Size
lr = 0.001
np.random.seed(1)

## Train
tr = train (net,epochs,lr)
```

0.2.1 Plot For Training Data

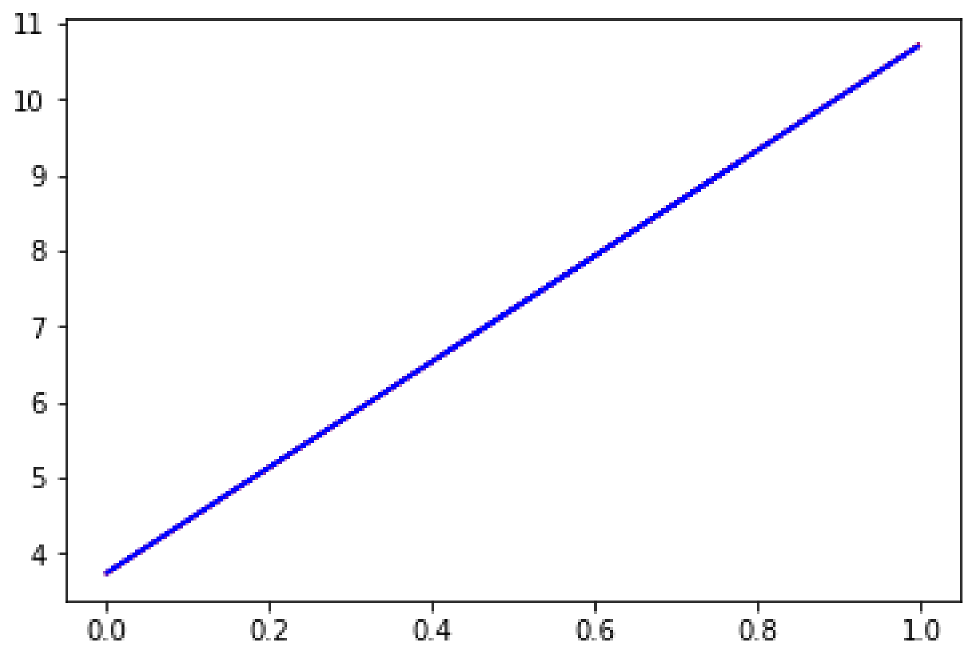
Red: Ground Truth Values
Blue: Predicted Values

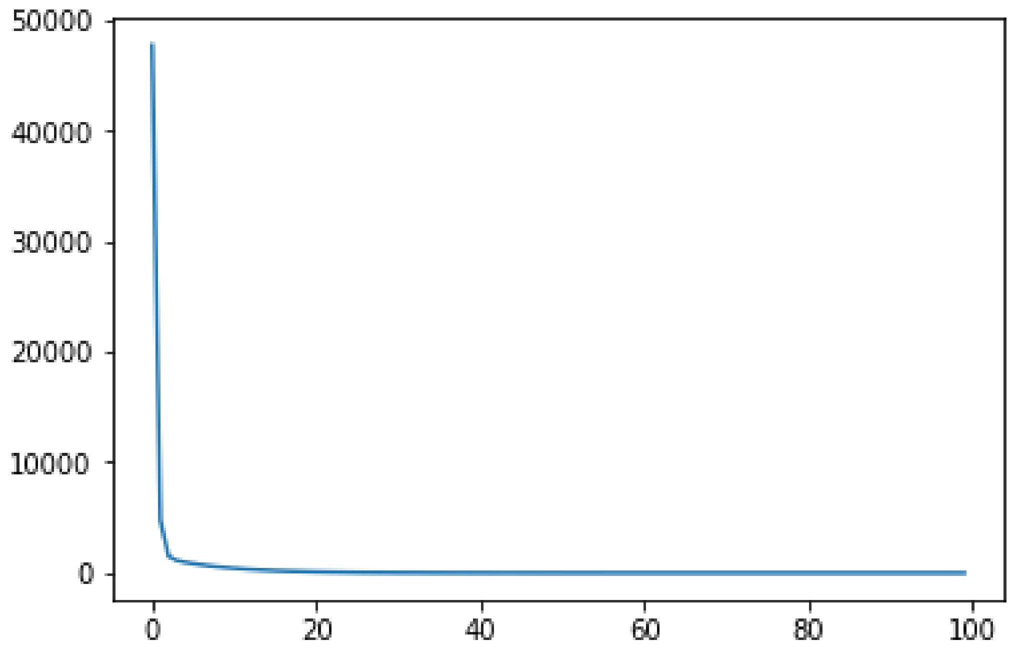
In [63]:
```
## Plotting the Outputs vs Inputs
plt.plot(net[0],net[1],'r')


## Caluclating Predictions from the network
plt.plot(net[0], feed_forward(net[0],tr[1],tr[2]),'b')

plt.show()


# Plotting the Error
plt.plot(tr[0])
plt.show()
```





0.2.2 Generating Test Data

```
In [27]: a=np.random.randint(low=1, high=1000, size=2)
         x = np.reshape(np.linspace(a[0],a[1], n_inputs),(n_inputs,1))
         y = 7*x + 3 + np.random.randn(1)
```

0.2.3 Plot For Test Data

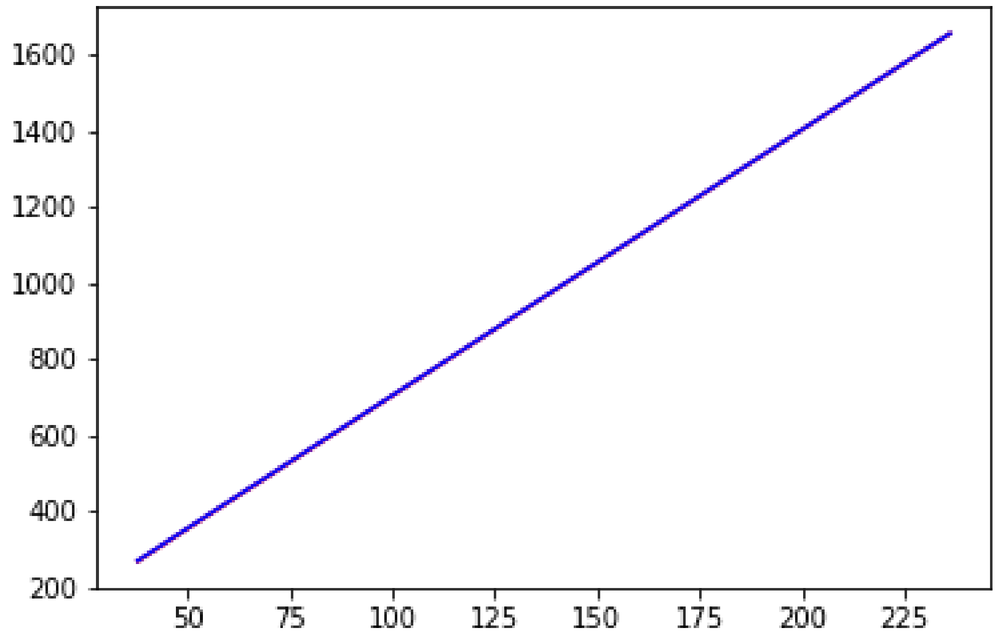
Red: Ground Truth Values
Blue: Predicted Values

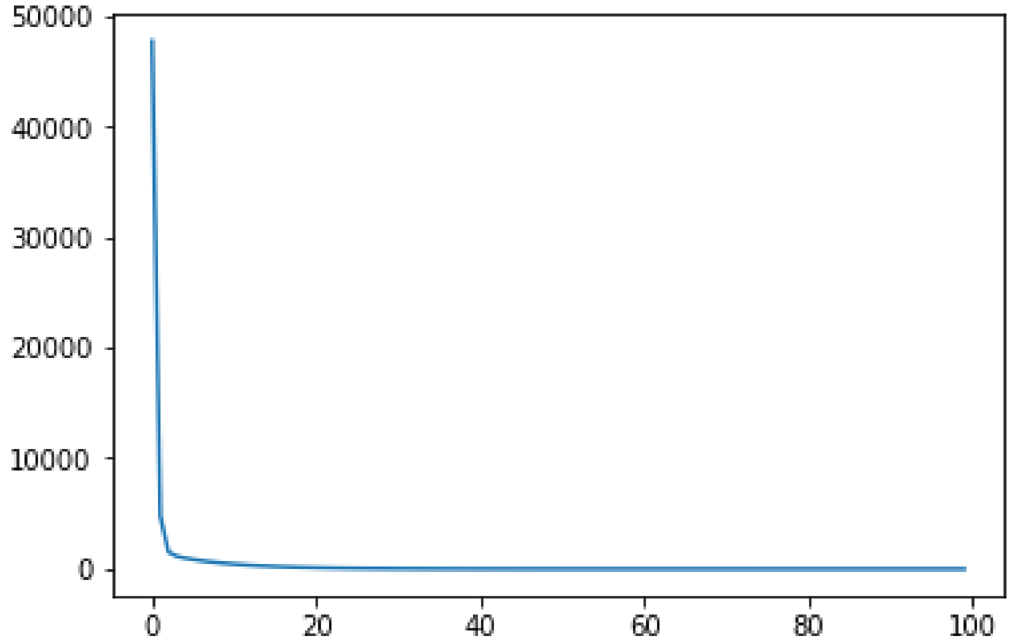
```
In [64]: ## Plotting the Outputs vs Inputs
         plt.plot(x,y,'r')

         ## Caluclating Predictions from the network
         plt.plot(x,feed_forward(x,tr[1],tr[2]),'b')

         plt.show()


         # Plotting the Error
         plt.plot(tr[0])
         plt.show()
```





0.2.4 Important Inferences

1. For n_inputs or number of training sample = 100 ,the training data start to converge from 600 epochs onwards, and totaly converge by 700 epochs.

2. A learning rate or step size of 0.001 is found to be ideal
3. Another hyper paramter found is the number of inputs.If we change the inputs from 100 to 1000, the training converged in just 100 epochs with the same learning rate. Therefore more training data helped the function converge faster.

0.3 (b) Defining the Main Function and Implementing for nD input

In [73]:
```
# Importing the Libraries
import numpy as np
from matplotlib import pyplot as plt
from math import exp

np.random.seed(1)
# Initializing the Network
n_inputs = 100
dimensions= 5
net=initialize_network(n_inputs,dimensions)

## Hyper Parameter Initialization
epochs= 1000    ## Step Size
lr = 0.001
np.random.seed(1)

## Train
tr = train (net,epochs,lr)
```

0.3.1 Plot For Training Data

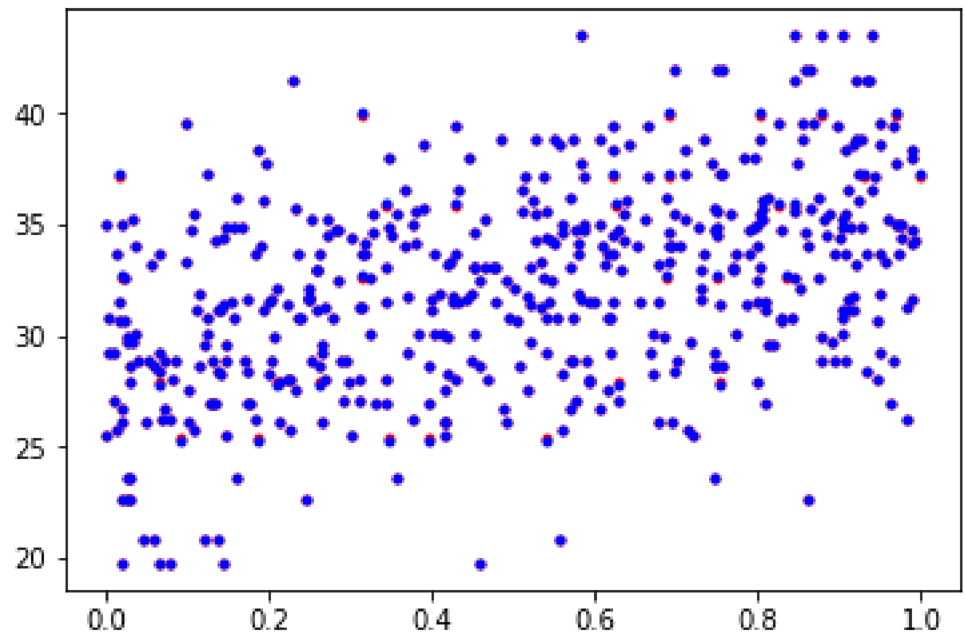
Red: Ground Truth Values
Blue: Predicted Values

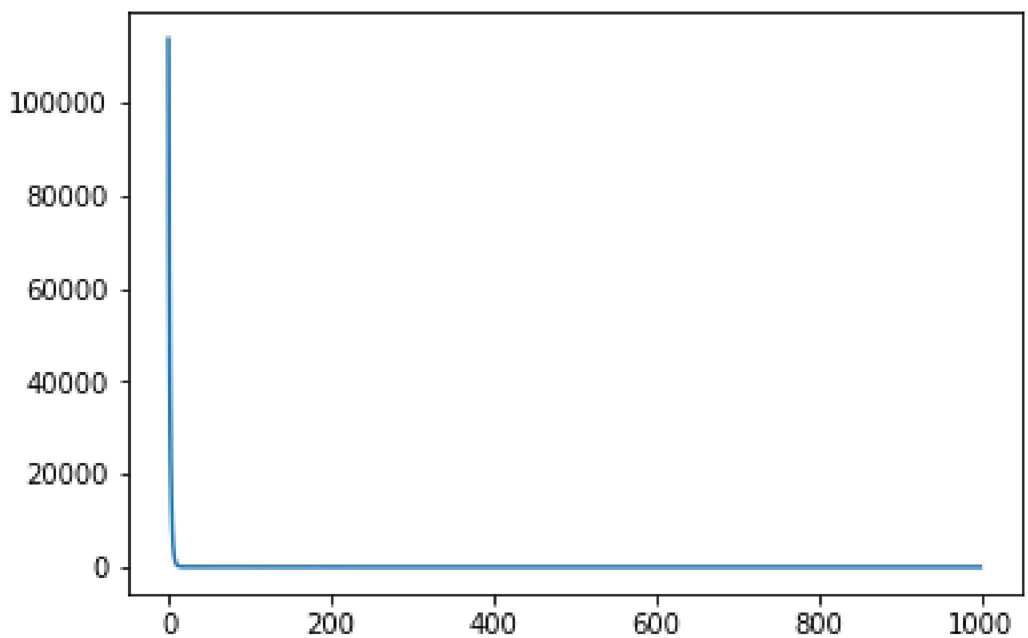
In [74]:
```
## Plotting the Outputs vs Inputs
plt.plot(net[0],net[1],'r.')


## Caluclating Predictions from the network
plt.plot(net[0], feed_forward(net[0],tr[1],tr[2]),'b.')

plt.show()


# Plotting the Error
plt.plot(tr[0])
plt.show()
```





0.3.2 Generating Test Data

In [75]:
```
x= np.random.rand(n_inputs,dimensions) ## Randomly Genrated Inputs
y= 7*x + 3 + np.random.randn(1)
```

```
y = np.sum(y,axis=1,keepdims= True)
```

0.3.3 Plot For Test Data

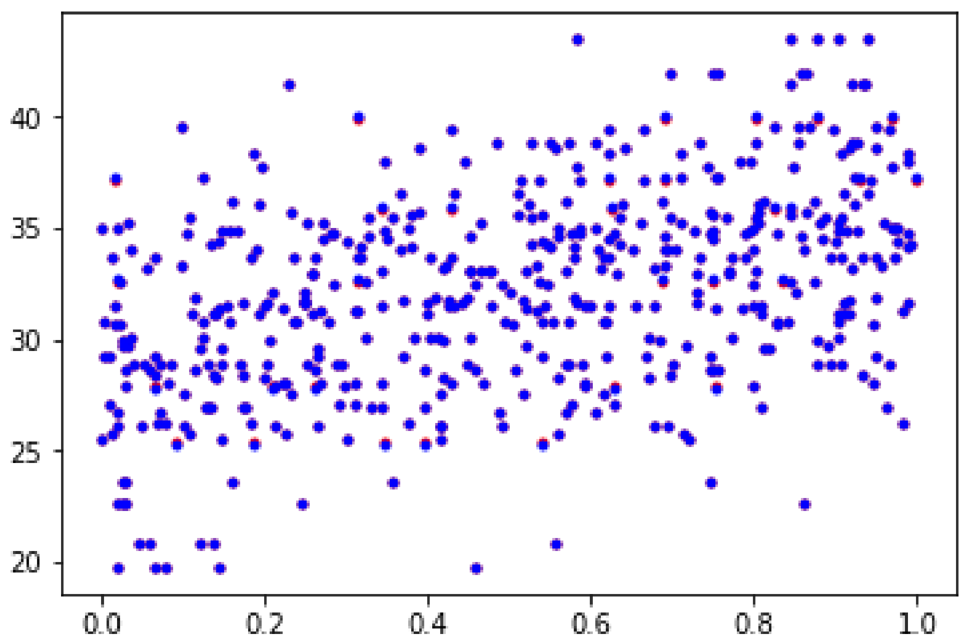
Red: Ground Truth Values
Blue: Predicted Values

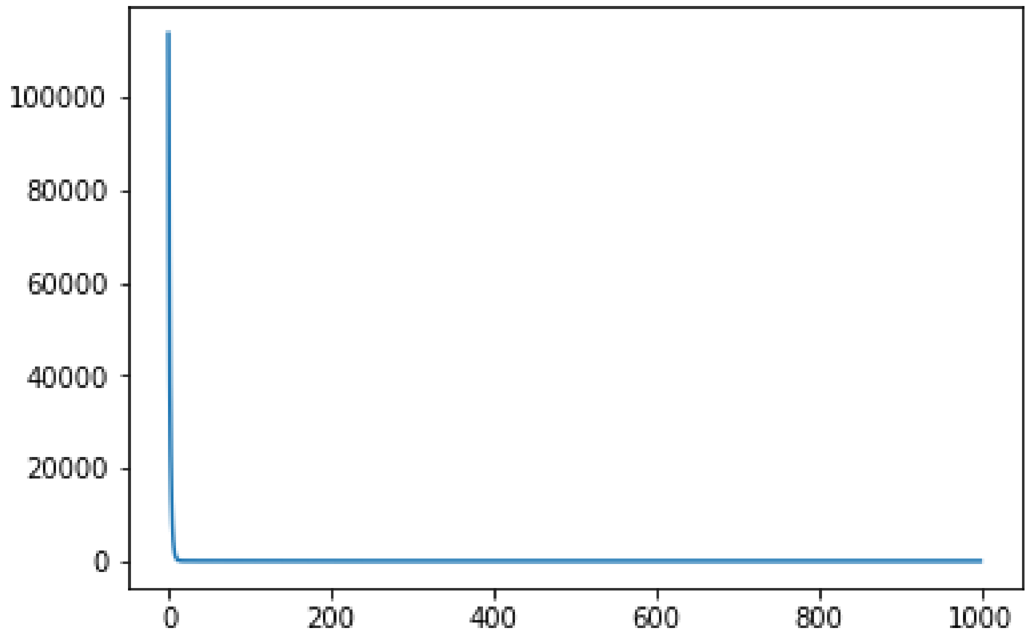
In [76]:
```
## Plotting the Outputs vs Inputs
plt.plot(net[0],y,'r.')

## Caluclating Predictions from the network
plt.plot(x,feed_forward(x,tr[1],tr[2]),'b.')

plt.show()


# Plotting the Error
plt.plot(tr[0])
plt.show()
```





0.3.4 Important Inferences

1. The relation of increased inputs to faster convergence also holds true here
2. As the dimensions increase, more epochs are needed or more training samples are needed to get high accuracy.
3. Learning rate og 0.001 is again found to e ideal. If we decrease the learning rate significantly, the error can aslo shoot up and therefore no convergence. With very low learning rate we need higher epochs.

0.4 (C) Comments on Hyper Parameters

- For the first case of 1D input:
  - there were two hyper paramters: **Learning Rate** and the number of **epochs**
  - Apart from this, **increasing the number of training inputs** also helps reducing the convergence time.
  - Therefore, tuning just three paramters is not too tedious and therefore, getting the right paramters is relatively easy.
- For the second case of nD input:
  - here there is an additional hyperparamter of **Dimension**
  - This adds on to the complexity because, by increasing the dimensions, more data is required or more epochs are required to converge.
  - But still, the number of paramters to tune are 4, which is still reasonable and therefore finding the right parmaters is still not that difficult