



Prof.Said El-Khamy

Antenna

Name : Tarek Reda Mohamed Ahmed

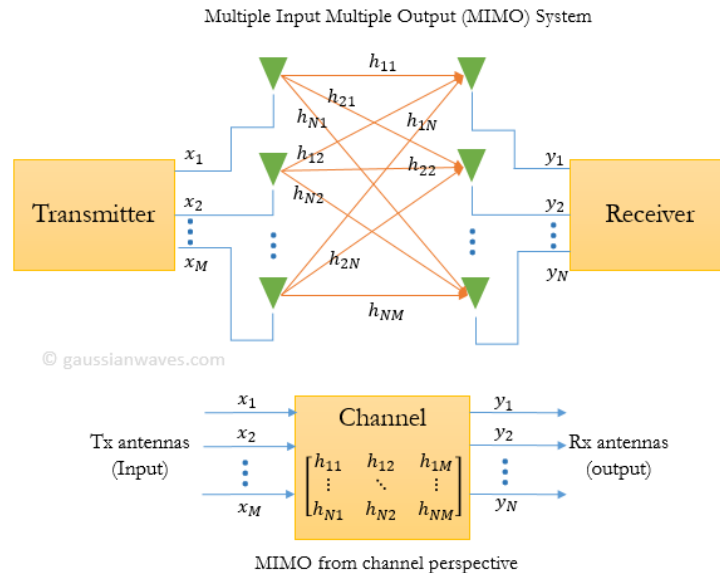
I.D: 147

Section: 5

Topic: MIMO introduction and Deep learning Applications.

General introduction to MIMO

Multiple Input Multiple Output technology uses multiple antennas to make use of reflected signals to provide gains in channel robustness and throughput.



Multiple-input multiple-output, or MIMO, is a radio communications technology or RF technology that is being mentioned and used in many new technologies these days. Wi-Fi, LTE; Long Term Evolution, and many other radio, wireless and RF technologies are using the new MIMO wireless technology to provide increased link capacity and spectral efficiency combined with improved link reliability using what were previously seen as interference paths.

MIMO technology has been developed over many years. Not only did the basic MIMO concepts need to be formulated, but in addition to this, new technologies needed to be developed to enable MIMO to be fully implemented. New levels of processing were needed to allow some of the features of spatial multiplexing as well as to utilise some of the gains of spatial diversity. Up until the 1990s, spatial diversity was often limited to systems that switched between two antennas or combined the signals to provide the best signal. Also various forms of beam switching were implemented, but in view of the levels of processing involved and the degrees of processing available, the systems were generally relatively limited. However with the additional levels of processing power that started to become available, it was possible to utilise both spatial diversity and full spatial multiplexing. The initial work on MIMO systems focused on basic spatial diversity - here the MIMO system was used to limit the degradation caused by multipath propagation. However this was only the first step as system then started to utilise the multipath propagation to advantage, turning the additional signal paths into what might effectively be

considered as additional channels to carry additional data. Two researchers: Arogyaswami Paulraj and Thomas Kailath were first to propose the use of spatial multiplexing using MIMO in 1993 and in the following year their US patent was granted.

A channel may be affected by fading and this will impact the signal to noise ratio. In turn this will impact the error rate, assuming digital data is being transmitted. The principle of diversity is to provide the receiver with multiple versions of the same signal. If these can be made to be affected in different ways by the signal path, the probability that they will all be affected at the same time is considerably reduced. Accordingly, diversity helps to stabilise a link and improves performance, reducing error rate.

Several different diversity modes are available and provide a number of advantages:

- Time diversity: Using time diversity, a message may be transmitted at different times, e.g. using different timeslots and channel coding.
- Frequency diversity: This form of diversity uses different frequencies. It may be in the form of using different channels, or technologies such as spread spectrum / OFDM.
- Space diversity (used in MIMO): Space diversity used in the broadest sense of the definition is used as the basis for MIMO. It uses antennas located in different positions to take advantage of the different radio paths that exist in a typical terrestrial environment.

One of the core ideas behind MIMO wireless systems space-time signal processing in which time (the natural dimension of digital communication data) is complemented with the spatial dimension inherent in the use of multiple spatially distributed antennas, i.e. the use of multiple antennas located at different points. Accordingly MIMO wireless systems can be viewed as a logical extension to the smart antennas that have been used for many years to improve wireless. It is found between a transmitter and a receiver; the signal can take many paths. Additionally by moving the antennas even a small distance the paths used will change. The variety of paths available occurs as a result of the number of objects that appear to the side or even in the direct path between the transmitter and receiver. Previously these multiple paths only served to introduce interference. By using MIMO, these additional paths can be used to advantage. They can be used to provide additional robustness to the radio link by improving the signal to noise ratio, or by increasing the link data capacity.

- Spatial diversity (Enhance link performance): Spatial diversity used in this narrower sense often refers to transmit and receive diversity. These two methodologies are used to provide improvements in the signal to noise ratio and they are characterized by improving the reliability of the system with respect to the various forms of fading.
- Spatial multiplexing (Enhance link efficiency) : This form of MIMO is used to provide additional data capacity by utilising the different paths to carry additional traffic, i.e. increasing the data throughput capability.

MIMO Formats

The different forms of antenna technology refer to single or multiple inputs and outputs. These are related to the radio link. In this way the input is the transmitter as it transmits into the link or signal path, and the output is the receiver. It is at the output of the wireless link. Therefore the different forms of single / multiple antenna links are defined as below:

- SISO - Single Input Single Output
- SIMO - Single Input Multiple output
- MISO - Multiple Input Single Output
- MIMO - Multiple Input multiple Output

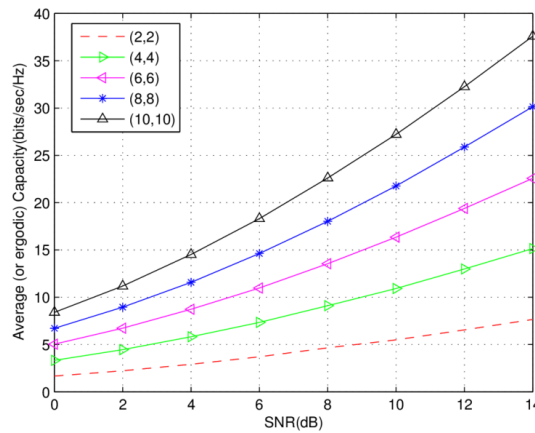
The term MU-MIMO is also used for a multiple user version of MIMO

In order to be able to benefit from MIMO fully it is necessary to utilise coding on the channels to separate the data from the different paths. This requires processing, but provides additional channel robustness / data throughput capacity

MIMO Spatial Multiplexing

The maximum amount of data that can be carried by a radio channel is limited by the physical boundaries defined under Shannon's Law. Shannon's Law and MIMO spatial multiplexing can be Shannon's law defines the maximum rate at which error free data is transmitted over a given bandwidth in the presence of noise. It is usually expressed in the form $C = W \log_2(1 + S/N)$ Where C is the channel capacity in bits per second, W is the bandwidth in Hertz, and S/N is the SNR (Signal to Noise Ratio). From this it can be seen that there is an ultimate limit on the capacity of a channel with a given bandwidth.

However before this point is reached, the capacity is also limited by the signal to noise ratio of the received signal.



To take advantage of the additional throughput offered, MIMO wireless systems utilise a matrix mathematical approach. Data streams t_1, t_2, \dots, t_n can be transmitted from antennas 1, 2, \dots n. Then there are a variety of paths that can be used with each path having different channel properties. To enable the receiver to be able to differentiate between the different data streams it is necessary to use. These can be represented by the properties h_{ij} , travelling from transmit antenna one to receive antenna 2 and so forth. In this way for a three transmit, three receive antenna system a matrix can be set up:

$$\begin{aligned} r_1 &= h_{11} t_1 + h_{21} t_2 + h_{31} t_3 \\ r_2 &= h_{12} t_1 + h_{22} t_2 + h_{32} t_3 \\ r_3 &= h_{13} t_1 + h_{23} t_2 + h_{33} t_3 \end{aligned}$$

Where r_1 = signal received at antenna 1, r_2 is the signal received at antenna 2 and so forth

In matrix format this can be represented as:

$$[R] = [H] \times [T]$$

To recover the transmitted data-stream at the receiver it is necessary to perform a considerable amount of signal processing. First the MIMO system decoder must estimate the individual channel transfer characteristic h_{ij} to determine the channel transfer matrix. Once all of this has been estimated, then the matrix $[H]$ has been produced and the transmitted data streams can be reconstructed by multiplying the received vector with the inverse of the transfer matrix.

✚ Deep neural networks in MIMO systems

In recent years, the world is witnessing a revolution in deep machine learning. In many fields of engineering, e.g., computer vision, it was shown that computers can be fed with sample pairs of inputs and desired outputs, and “learn” the functions which relates them. These rules can then be used to classify (detect) the unknown outputs of future inputs. We will apply deep machine learning in the classical MIMO detection problem and understand its advantages and disadvantages.

The binary MIMO detection setting is a classical problem in simple hypothesis testing. The maximum likelihood (ML) detector is the optimal detector in the sense of minimum joint probability of error for detecting all the symbols simultaneously. It can be implemented via efficient search algorithms, e.g., the sphere decoder. The difficulty is that its worst case computational complexity is impractical for many applications.

Consequently, several modified search algorithms have been purposed, offering improved complexity performance.

In the context of MIMO detection, a model is known and allows us to generate as many synthetic examples as needed. Therefore we adapt an alternative notion. We interpret “learning” as the idea of choosing a best decoder from a prescribed class of algorithms. Classical detection theory tries to choose the best estimate of the unknowns, whereas machine learning tries to choose the best algorithm to be applied.

In recent years, deep learning methods have been purposed for improving the performance of a decoder for linear codes in fixed channels. And in several applications of deep learning for communication applications have been considered, including decoding signals over fading channels, but the architecture purposed there does not seem to be scalable for higher dimension signals.

✚ Coding steps

we formulate the MIMO detection problem in a machine learning framework. We consider the standard linear MIMO model:

$$y = Hx + w \quad (1)$$

We assume perfect channel state information (CSI) and that the channel H is exactly known. However, we differentiate between two possible cases: ② Fixed Channel (FC): In the FC scenario, H is deterministic and constant (or

a realization of a degenerate distribution which only takes a single value).

Varying Channel (VC): In the VC scenario, we assume H random with a known distribution. Our goal is to detect x , using an algorithm that receives y and H as inputs and estimates \hat{x} .

To find the best detector, we fix a loss function $l(x; \hat{x}(H; y))$ that measures the distance between the true symbols and their estimates. Then, we find \hat{x} by minimizing the loss function we chose over the MIMO model distribution:

$$\min_{\hat{x}} \{E l(x; \hat{x}(H; y))\} \quad (2)$$

- method 1 : The goal in detection is to decrease the probability of error. Therefore, the best loss function in this problem is choosing an unrealistically flexible architecture with unbounded parameterization and no restrictions such that

$$l(x; \hat{x}_{\theta}(H, y)) = \begin{cases} 1 & x \neq \hat{x}_{\theta}(y, H) \\ 0 & \text{else.} \end{cases}$$

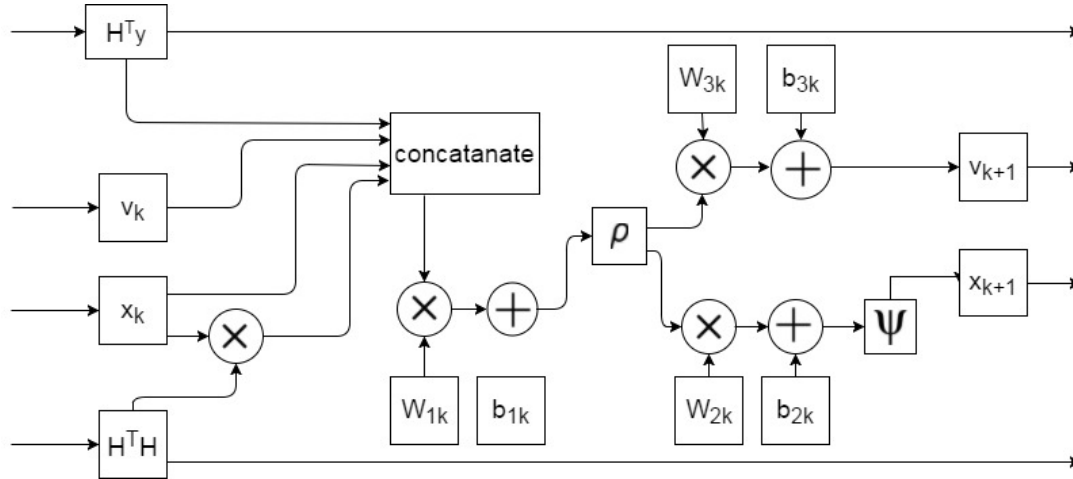
- method 2: On the other extreme, consider the architecture of fixed linear detectors where the parameter is a single fixed matrix to be optimized within $R_{\text{power}}(K \times N)$.

$$\hat{x}_{\theta}(y, H) = Ay,$$

These two methods emphasize how fixing an architecture and a loss function determines what will be the optimal detector for the MIMO detection problem. The more expressive we choose \hat{x} to be, the more accurate the final detector can be, on the expense of the computational complexity.

Our chosen deep architecture is based on multiple layers with multivariate linear operations and element-wise non-linear operators. These allow rich decoding functions while resorting to a finite and tractable parameterization. In addition, analytic computation of the expectation in the objective is usually impossible. Instead, we approximate it using an empirical mean of samples drawn from a data set of examples (thus the 'learning' notion). In our case, the data set is composed of synthetically generated samples satisfying equation (1).

Implementing DetNet “a detection network proposed in 2017 for MIMO detection”



This graph represents the construction of each single layer of DetNet

the network was trained using a variant of the stochastic gradient descent method, for optimizing deep networks, named Adam Optimizer. using batch training with 5000 random data samples at each iteration, and trained the network for 50000 iterations. To give a rough idea of the complexity, learning the detectors in our numerical results took 2 days on a standard Intel i7-6700 processor. Each sample was independently generated from (1) according to the statistics of x , H (either in the FC or VC model) and w . With respect to the noise, its variance is unknown and therefore this too was randomly generated so that the SNR will be uniformly distributed on $U(\text{SNR}_{\min}; \text{SNR}_{\max})$. This approach allows us to train the network to detect over a wide range of SNR values.

(disclaimer: this is the process of training the real det net model not my version which we will discuss later)

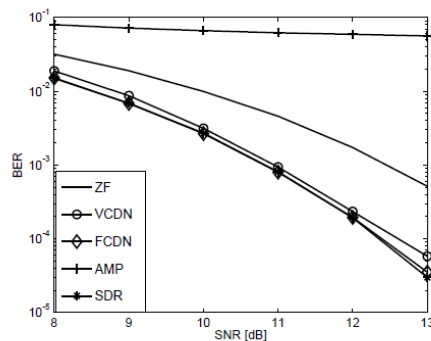


Fig. 3. Comparison of BER performance in the fixed channel case between the detection algorithms. all algorithms were tested on the 0.55-Toeplitz channel.

the performance of the following detection algorithms:

- FCDN: DetNet algorithm described in with 3K layers, z_k of size 8K, and v_k of size 2K. FCDN was trained using the FC model described above, and is specifically designed to handle a specific ill conditioned channel matrix.
- VCDN: Same architecture as the FCDN but the training is on the VC model and is supposed to cope with arbitrary channel matrices.
- ShVCDN : Same as the VCDN algorithm, but with a shallow network architecture using only K layers.
- ZF: This is the classical decorrelator, also known as least squares or zero forcing (ZF) detector .
- AMP: Approximate message passing algorithm from [5]. The algorithm was adjusted to the real-valued case and was implemented with 3K iterations.
- AMP2: Same as the AMP algorithm but with a mis-specified SNR.
- SDR: A decoder based on semidefinite relaxation implemented using a specifically tailored and efficient interior point solver .

Another important feature of DetNet is the ability to trade off complexity and accuracy by adding or removing additional layers.

✚ My modifications and DetNet code version:

I took the main concepts of the det net paper presented in

<https://arxiv.org/abs/1706.01151>

Using the same code base but also forking some of the modifications from another deep mimo architecture presented by Deeksha Kaurav “senior undergraduate @ IIT, varanasi” and also presenting my own changes

- Code:

```
import math
import tensorflow as tf
import pylab
import numpy as np
import os
import random
import numpy as np
import tensorflow as tf
from tensorflow.python.util import nest
from tensorflow.python.ops import tensor_array_ops
from tensorflow.python.framework import tensor_shape
```

```

from tensorflow.python.framework import ops
from tensorflow.python.framework import tensor_util
from tensorflow.python.ops import control_flow_ops

DataSet_x=[]
DataSet_F1=[]
DataSet_F2=[]
NrSamples = 1000
NrSamples_ToTest = 100
SNR_dB_range=np.arange(0,22,2)
BER=np.zeros(SNR_dB_range.shape,dtype=np.int)
Sigma2 = 1.0
SNR_min_dB = 0.0
SNR_max_dB = 20.0
K = 2
N = 2*K
L = 3*K
batch_size=10
number_of_batches=NrSamples/batch_size
c=0
kk=0
l=1
for runIdx in range(0,NrSamples):
    # generate a random operating SNR
    SNR_dB = SNR_min_dB + (SNR_max_dB - SNR_min_dB)*np.random.rand()
    SNR = 10.0**(SNR_dB/10.0)
    H = ((0.5*float(SNR)/float(K))**(1/2.0))*np.random.randn(N,K)
    x = 2*np.round(np.random.rand(K,1))-1;
    h_prime=np.transpose(H)
    noise = (Sigma2/2.0)**(1/2.0)*np.random.randn(N,1)
    y = np.matmul(H,x) + noise
    c=c+1
    DataSet_x.append(x)
    DataSet_F1.append(np.matmul(h_prime,y))
    DataSet_F2.append(np.matmul(h_prime,H))
print("training_data")
kk=0
kk2=0
kk3=0
X=tf.placeholder(tf.float64,shape=(batch_size,2,1))
F1=tf.placeholder(tf.float64,shape=(batch_size,2,1))
F2=tf.placeholder(tf.float64,shape=(batch_size,2,2))

#Test data
TestSet_x=[]
TestSet_F1=[]
TestSet_F2=[]
SNR_dB_range=np.arange(0,22,2)
BER=np.zeros(SNR_dB_range.shape,dtype=np.int)
Sigma2 = 1.0
SNR_min_dB = 0.0

```

```

SNR_max_dB = 20.0
K = 2
N = 2*K
L = 3*K
number_of_batches_test=NrSamples_ToTest/batch_size
c=0
for runIdx in range(0,NrSamples_ToTest):
    # generate a random operating SNR
    SNR_dB = SNR_min_dB + (SNR_max_dB - SNR_min_dB)*np.random.rand()
    SNR = 10.0**(SNR_dB/10.0)
    H = ((0.5*float(SNR)/float(K))**(1/2.0))*np.random.randn(N,K)
    x = 2*np.round(np.random.rand(K,1))-1;
    h_prime=np.transpose(H)
    noise = (Sigma2/2.0)**(1/2.0)*np.random.randn(N,1)
    y = np.matmul(H,x) + noise
    c=c+1
    TestSet_x.append(x)
    TestSet_F1.append(np.matmul(h_prime,y))
    TestSet_F2.append(np.matmul(h_prime,H))
kk1=0
kk2=0
kk3=0
with tf.variable_scope("embedding") as scope:
    if kk1 > 0:
        scope.reuse_variables()
    else:
        W1=tf.get_variable("embedding1",initializer=tf.random_normal([L,8*K,5*K],stddev=0.1,dtype=
tf.float64),trainable=True,dtype=tf.float64)

        b1=tf.get_variable("embedding2",initializer=tf.random_normal([L,8*K,1],stddev=0.1,dtype=tf.fl
oat64),trainable=True,dtype=tf.float64)
        kk1=1
with tf.variable_scope("embedding") as scope:
    if kk2 > 0:
        scope.reuse_variables()
    else:
        W2=tf.get_variable("embedding3",initializer=tf.random_normal([L,K,8*K],stddev=0.1,dtype=tf.
float64),trainable=True,dtype=tf.float64)

        b2=tf.get_variable("embedding4",initializer=tf.random_normal([L,K,1],stddev=0.1,dtype=tf.floa
t64),trainable=True,dtype=tf.float64)

with tf.variable_scope("embedding") as scope:
    if kk3 > 0:
        scope.reuse_variables()
    else:
        W3=tf.get_variable("embedding5",initializer=tf.random_normal([L,2*K,8*K],stddev=0.1,dtype=
tf.float64),trainable=True,dtype=tf.float64)

```

```
b3=tf.get_variable("embedding6",initializer=tf.random_normal([L,2*K,1],stddev=0.1,dtype=tf.float64),trainable=True,dtype=tf.float64)
```

```
def zk(Hty,xk,HtH,vk,k,name='embedding'): #(f1,x_hat,f2,v_hat,k)
    inp3=tf.matmul(HtH,xk)
    concat=tf.concat([Hty,xk,inp3,vk],0)
    temp=tf.matmul(W1[k-1],concat)+b1[k-1]
    zk=tf.nn.relu(temp)
    return zk
```

```
def psi(x,tt):
    t=tt*tf.ones_like(x)
    relu1=tf.nn.relu((x+t))
    relu2=tf.nn.relu((x-t))
    ab=tf.abs(t)
    out1=tf.div(relu1,ab)
    out2=tf.div(relu2,ab)
    one=tf.ones_like(out1,dtype=tf.float64)
    temp=tf.add(one,out2)
    return tf.subtract(out1,temp)
```

```
def xk(z_hat,k,name='embedding'):
    return psi(tf.matmul(W2[k-1],z_hat)+b2[k-1],tt=0.1)
```

```
def vk(z_hat,k,name='embedding'):
    return tf.matmul(W3[k-1],z_hat)+b3[k-1]
```

```
def x_tilde(HtH,Hty):
    HtH_inv=tf.matrix_inverse(HtH)
    return tf.matmul(HtH_inv,Hty)
```

```
def model(x_f,f1_f,f2_f):
    loss_2=0
    pred=[]
    for dsIdx in range(0,batch_size):
        x = x_f[dsIdx]
        f1 = f1_f[dsIdx] ##hty
        f2 = f2_f[dsIdx] ##hth
        x_hat=tf.zeros((K,1),dtype = tf.float64)
        v_hat=tf.zeros((2*K,1), dtype = tf.float64)
        x_tilde_num=x_tilde(f2,f1)
        temp_loss=0
        for k in range(1,L+1):
            z_hat = zk(f1,x_hat,f2,v_hat,k)
            x_hat=xk(z_hat,k)
            v_hat=vk(z_hat,k)
            num=math.log(k)*tf.abs(x-x_hat)*tf.abs(x-x_hat)
            denom=tf.abs(tf.subtract(x,x_tilde_num))*tf.abs(tf.subtract(x,x_tilde_num))
            loss_k=tf.div(num,denom)
            loss_k=tf.reduce_sum(loss_k)
            pred.append(x_hat)
```

```

        temp_loss=temp_loss+loss_k
    loss_2=loss_2+temp_loss
    ##### Combined Loss of all layers
#####
    return loss_2,pred
print("model")
loss_, out=model(X,F1,F2)
loss=loss_/batch_size
tf.summary.histogram("loss",loss)
loss=tf.clip_by_value(loss,clip_value_min=0,clip_value_max=1000)
train_step=tf.train.AdamOptimizer(0.001).minimize(loss)
print("here1")
saver=tf.train.Saver()
gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=0.333)
sess = tf.Session(config=tf.ConfigProto(gpu_options=gpu_options))
list_of_variables = tf.all_variables()
sess.run(tf.global_variables_initializer())
variable_names= [v.name for v in tf.trainable_variables()]
uninitialized_variables = list(tf.get_variable(name) for name in
sess.run(tf.report_uninitialized_variables(list_of_variables)))
print("here2")
init_op = tf.variables_initializer(list_of_variables)
sess.run(tf.variables_initializer(uninitialized_variables))
print(sess.run(tf.report_uninitialized_variables(list_of_variables)))
t=0
with sess.as_default():
    merged=tf.summary.merge_all()
    train_writer=tf.summary.FileWriter("train_loss_f",sess.graph)
    for epoch in range(100,1000):
        loss_ep=0
        random.shuffle(DataSet_x)
        random.shuffle(DataSet_F1)
        random.shuffle(DataSet_F2)
        print("ep")
        for i in range(0,int(number_of_batches)):
            train_step.run(feed_dict={X: DataSet_x[i*batch_size:i*batch_size+batch_size],
F1:DataSet_F1[i*batch_size:i*batch_size+batch_size],
F2:DataSet_F2[i*batch_size:i*batch_size+batch_size]})
            loss1,summery=sess.run([loss,merged],feed_dict={X:
DataSet_x[i*batch_size:i*batch_size+batch_size],
F1:DataSet_F1[i*batch_size:i*batch_size+batch_size],
F2:DataSet_F2[i*batch_size:i*batch_size+batch_size]})
            loss_ep=loss_ep+loss1
            file=open("loss_f.txt","a")
            file.write("loss"+" "+str(i)+" "+str(loss1)+"\n")
            file.close()
            print("loss"+" "+str(loss1))
            train_writer.add_summary(summary,t)
            t=t+1
        file=open("loss_f.txt","a")
        file.write("loss_ep"+" "+str(loss_ep/number_of_batches)+"\n")

```

```

file.close()
loss_ep_t=0
for j in range(0,int(number_of_batches_test)):
    loss_t, out_t=sess.run([loss,out],feed_dict={X:
TestSet_x[j*batch_size:j*batch_size+batch_size],
F1:TestSet_F1[j*batch_size:j*batch_size+batch_size],
F2:TestSet_F2[j*batch_size:j*batch_size+batch_size]})
    loss_ep_t=loss_ep_t+loss_t
    file=open(str(epoch)+"-"+str("test_labels_f.txt"),"a")
    for z in range(0,batch_size):
        file.write("test_input"+" "+str(TestSet_x[j*batch_size+z][0][0])+
"+str(TestSet_x[j*batch_size+z][1][0])+"\n")
        file.write("test_label"+" "+str(out_t[z][0][0])+ " "+str(out_t[z][1][0])+"\n")
    file.close()
file1=open("test_loss_f.txt","a")
file1.write("test_loss"+" "+str(loss_ep_t/number_of_batches_test)+"\n")
file1.close()
print("loss_ep")
print(loss_ep/number_of_batches)

```

Training the model and changing batch sizes

I changed the batch sizes and the number of training iterations to much smaller values to be able to train the model and test it on my local machine which affected the loss but still proves the concept

The original values used in the DetNet paper:

NrSamples = 100000

NrSamples_ToTest = 10000

batch_size=1000

for epoch in range(100,4000):

Loss values

```

[Running] python -u "d:\programms eng\software projects\Deep-MIMO-Detection-master\Deep-MIMO-Detection-master\DeepMimo.py"
training_data
model
here1
2019-12-20 00:00:28.745992: I C:\tf_jenkins\home\workspace\rel-win\M\windows\PY36\tensorflow\core\platform\cpu_feature_guard.cc:
WARNING:tensorflow:From d:\programms eng\software projects\Deep-MIMO-Detection-master\Deep-MIMO-Detection-master\DeepMimo.py:173:
Instructions for updating:
Please use tf.global_variables instead.
here2
[]
ep
loss 144.68960825160474
loss 82.44932325723465
loss 8.05493955685488
loss 344.1071546097637
loss 1000.0
loss 8.441334108739742
loss 14.73624838888881
loss 49.62072235431318
loss 1000.0
loss 26.960726717081684
loss_ep
267.90600572444816

```






The loss value in the first iteration was 267.9

```
loss 152.6359334281513
loss 9.080317857216812
loss 18.419877255650412
loss 12.864076272415323
loss 7.057918859733673
loss 9.744055337983536
loss 9.275616666586995
loss 10.690866722774237
loss 8.701728854426625
loss 8.846923131979546
loss_ep
24.731731438691845
```

```
[Done] exited with code=0 in 645.383 seconds
```

The loss value in the last iteration was 24.73

Achieving better values is possible but will take much longer training times

	train_loss_f	12/19/2019 11:46 ...	File folder	
	100-test_labels_f	12/19/2019 11:46 ...	Text Document	1 KB
	DeepMimo	12/20/2019 12:19 ...	Python File	8 KB
	loss_f	12/19/2019 11:56 ...	Text Document	425 KB
	test_loss_f	12/19/2019 11:56 ...	Text Document	27 KB

The code generates txt files documenting all the training and test losses

References :

- <https://www.electronics-notes.com/articles/antennas-propagation/mimo/massive-mimo-large.php>
- <https://silvustechologies.com/why-silvus/technology/introduction-to-mimo>
- <https://arxiv.org/abs/1706.01151>