```
import numpy as np
import tensorflow as tf
from tensorflow import keras
import matplotlib.pyplot as plt
```

All the parameters required for the network can be tweaked below

```
### Hyperparameters, change accordingly
Channel_uses = 4
Epochs = 2500
Noise_variance = 1e-4
Pert_variance = 1e-4
Batch_size = 1024
# init_losses_vec = np.ones(128)
```

TX loss function

The policy function for the transmitter is similar to that of a cross-entropy between the noisy loss feedback (l) and the J(w,θ) function value

Loss = -$\sum_{i=1}^{n}(l_i$ * J$(w_i$,θ))

```
def tx_loss(y_true, y_pred):
#     loss = - y_true*keras.backend.log(y_pred)

    return -y_true*y_pred
```

Perturbation

After we get the output from the transmitter network, we then add the perturbation matrix as mentioned in the paper. We write a function for this purpose and then make a custom layer like functionality using the `keras.layers.lambda` functionality

```
def perturbation(d):
    W = tf.keras.backend.random_normal(shape = (2*Channel_uses,),
    mean=0.0,stddev=Pert_variance**0.5,dtype=None,seed=None)
    d = ((1-Pert_variance)**0.5)*d + W
    return d
```

Tx model

```
def Int_layer(y):
    w = y[:,y.shape[-1]//2:] - y[:,:y.shape[-1]//2]
    print(w.shape)
    t = -keras.backend.sum(w*w)
#     t = keras.backend.exp(-t/Pert_variance**2)/(np.pi*Pert_variance**2)**Channel_u
    return t
```

```
# tx layers
tx_input = keras.layers.Input((1,), name='tx_input')
```

```
x = keras.layers.BatchNormalization()(tx_input)
x = keras.layers.Dense(units=10*Channel_uses, activation='elu', name='tx_10')(x)
x = keras.layers.Dense(units=2*Channel_uses, activation='elu', name='tx_out')(x)
xp = keras.layers.Lambda(perturbation, name='Xp')(x)
concat = keras.layers.concatenate([x,xp], axis=1)
policy = keras.layers.Lambda(Int_layer)(concat)
print(concat.shape)
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow/pythor
Instructions for updating:
Colocations handled automatically by placer.
(?, 8)
(?, 16)
```

We define the entire graph but for simplicity sake, we also define a sub-model for getting the internediate layer outputs.

To be even more precise, we add perturbation after we get the Tx layer output. So, to get the perturbation matrix out, we define a full model and another proxy model (which shares weights with the full model) which return without perturbation matrix effects.

We then subtract these two layers to get the value of W (perturbation matrix) for a given batch/sample

(Note that we had to take this roundabout method to get W because Keras can't return two tensors for a said layer)

```
tx_model = keras.models.Model(inputs=tx_input, outputs=concat)
```

```
tx_model.summary()
```

```
_____________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
=============================================================================
tx_input (InputLayer)           (None, 1)            0
_____________________________________________________________________________
batch_normalization_v1 (BatchNo (None, 1)            4           tx_input[0][(
_____________________________________________________________________________
tx_10 (Dense)                   (None, 40)           80          batch_normal:
_____________________________________________________________________________
tx_out (Dense)                  (None, 8)            328         tx_10[0][0]
_____________________________________________________________________________
Xp (Lambda)                     (None, 8)            0           tx_out[0][0]
_____________________________________________________________________________
concatenate (Concatenate)       (None, 16)           0           tx_out[0][0]
                                                                 Xp[0][0]
=============================================================================
Total params: 412
Trainable params: 410
Non-trainable params: 2
_____________________________________________________________________________
```

```
pert_model = keras.models.Model(inputs=tx_input, outputs=policy)
```

```
pert_model.compile(loss=tx_loss, optimizer='sgd')
pert_model.summary()
```

```
Layer (type)                    Output Shape         Param #     Connected to
==========================================================================
tx_input (InputLayer)           (None, 1)            0

batch_normalization_v1 (BatchNo (None, 1)            4           tx_input[0][(

tx_10 (Dense)                   (None, 40)           80          batch_normal:

tx_out (Dense)                  (None, 8)            328         tx_10[0][0]

Xp (Lambda)                     (None, 8)            0           tx_out[0][0]

concatenate (Concatenate)       (None, 16)           0           tx_out[0][0]
                                                                 Xp[0][0]

lambda (Lambda)                 ()                   0           concatenate[(
==========================================================================
Total params: 412
Trainable params: 410
Non-trainable params: 2
```

Rx model

In the said RX model, we are taking the Perturbed input, adding channel effects and then passing on for estimation.

```
rx_input = keras.layers.Input((2*Channel_uses,), name='rx_input')
# channel layer
y = keras.layers.Lambda(lambda x: x+keras.backend.random_normal(
        shape = (2*Channel_uses,), mean=0.0, stddev=Noise_variance**0.5), name='chan

y = keras.layers.Dense(2*Channel_uses, activation='relu', name='rx_2')(y)
y = keras.layers.Dense(10*Channel_uses, activation='relu', name='rx_10')(y)
pred = keras.layers.Dense(1, activation='relu', name='rx_output')(y)


rx_model = keras.models.Model(inputs=rx_input, outputs=pred)
rx_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
rx_input (InputLayer)        (None, 8)                 0
_________________________________________________________________
channel (Lambda)             (None, 8)                 0
_________________________________________________________________
rx_2 (Dense)                 (None, 8)                 72
_________________________________________________________________
rx_10 (Dense)                (None, 40)                360
```

```
rx_model.compile(loss='mse', optimizer='sgd')
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow/pytho
Instructions for updating:
Use tf.cast instead.
```

```
data_numbers = np.random.uniform(0,1,(Batch_size,))
y = tx_model.predict(data_numbers)
print(y.shape)
XP = y[:,y.shape[-1]//2:]
estimated_vector  = np.squeeze(rx_model.predict(XP))
print(estimated_vector.shape, data_numbers.shape)
```

```
(1024, 16)
(1024,) (1024,)
```

```
l = (estimated_vector-data_numbers)**2
l_hat = rx_model.predict(tx_model.predict(data_numbers)[:,2*Channel_uses:])
```

```
pert_model.fit(data_numbers, l_hat, batch_size=Batch_size, epochs=1)
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow/pytho
Instructions for updating:
Use tf.cast instead.
1024/1024 [==============================] - 0s 190us/sample - loss: 0.0217
<tensorflow.python.keras.callbacks.History at 0x7f2c434b1d30>
```

Training

Training this entire network is done as discussed in the paper -

1. Generate a batch of numbers sampled from Uniform random variable from [0,1]
2. Pass the numbers through Tx and then Rx
3. Get a loss vectors for the said batch of numbers
4. Train the Rx network on MSE with SGD
5. Feed back the loss vector to Tx using the same pair of Tx and Rx to incorporate noise into the loss vector
6. Use policy function, the loss vector and train the Tx for the same batch of numbers

7. Back to step 1

```
for i in range(Epochs):
    data_numbers = np.random.uniform(0,1,(Batch_size,))
    y = tx_model.predict(data_numbers)
    XP = y[:,y.shape[-1]//2:]
    estimated_vector= np.squeeze(rx_model.predict(XP))
    l = (estimated_vector-data_numbers)**2
    l_hat = rx_model.predict(tx_model.predict(data_numbers)[:,2*Channel_uses:])
    pert_model.fit(data_numbers, l_hat, batch_size=Batch_size, epochs=1, verbose=0)
#     print("Tx-done")
    rx_model.fit(XP, data_numbers, batch_size=Batch_size, epochs=1)
#     print("Rx-done")
```

Removing the cell output for loss printing for PDF purposes. The loss must converge to approximately 3*1e-4

Prediction phase

Note that the network is predicting numbers with a quite low error margin (+- 1e-2) This is in case of continous numbers Say we feed numbers sampled from PAM (discrete numbers) and set our prediction rules as a floor or ceiling function, this model easily achieves 95% accuracy

This is all achieved even though there is a noisy feedback of losses from Tx to Rx

```
data_numbers = np.random.uniform(0,1,(10,))
y = tx_model.predict(data_numbers)
XP = y[:,y.shape[-1]//2:]
estimated_vector= np.squeeze(rx_model.predict(XP))
print(data_numbers)
print(estimated_vector)
# l = (estimated_vector-data_numbers)**2
# l_hat = rx_model.predict(tx_model.predict(data_numbers)[:,2*Channel_uses:])
# pert_model.fit(data_numbers, l_hat, batch_size=Batch_size, epochs=1)
# print("Tx-done")
# rx_model.fit(XP, data_numbers, batch_size=Batch_size, epochs=1)
# print("Rx-done")
```

```
[0.43114407 0.0724069  0.71251768 0.62851457 0.3649974  0.80804718
 0.00472749 0.22596115 0.67318013 0.64432473]
[0.43335813 0.08210276 0.734823   0.6464252  0.34259534 0.8232196
 0.04335603 0.20517816 0.6946332  0.6636828 ]
```

Post implimentation tid-bits

Please note that we had to make some chnages from the original discussed implimentation and theory to attain some numerical stability and to dodge NaN losses

1. In the J(w,θ) function, we have a part involving exp(|w|) and some constants. Where as the loss involved L_i * log(J(w,θ)). This causes numerical instability in case the J function goes negative or is very very small due to exp() and then log. To prevent this, we ignored the constants (as they dont affect gradient terms while differentiating) and removed the exp() and log() terms all-together

2. Author assumed two pairs of Tx-Rx with shared weights. We used one for both purposes as it is symmetric

Modulation example

As we mentioned before, here is an 8 PAM example. We generate real numbers between 0 and 1 for 8 samples and pass through the network and decode accordingly and check accuracy. Since this is a fairly small example, it attains 100% accuracy

```
# Generate 8-PAM data
modulation_data = np.linspace(0,1,8)
modulation_inds = np.linspace(0,7,8)
print(modulation_data)
modulation_inds = modulation_inds.astype(int)
print(modulation_inds)
decision_regions = []
for i in range(len(modulation_data) -1):
  decision_regions.append((modulation_data[i]+modulation_data[i+1])/2.0)

print(decision_regions)
```

```
[0.         0.14285714 0.28571429 0.42857143 0.57142857 0.71428571
 0.85714286 1.        ]
[0 1 2 3 4 5 6 7]
[0.07142857142857142, 0.21428571428571427, 0.3571428571428571, 0.5, 0.642857
```

```
y = tx_model.predict(modulation_data)
XP = y[:,y.shape[-1]//2:]
estimated_vector= np.squeeze(rx_model.predict(XP))
print(modulation_data)
print(estimated_vector)

est_sig = []
for i in estimated_vector:
  index = (np.abs(modulation_data-i)).argmin()
  est_sig.append(modulation_inds[index])

est_sig = np.array(est_sig)
print(est_sig)
```

```
[0.         0.14285714 0.28571429 0.42857143 0.57142857 0.71428571
 0.85714286 1.        ]
[0.05045348 0.14988288 0.27580988 0.44272137 0.5882717  0.7469301
 0.8735622  0.97393775]
[0 1 2 3 4 5 6 7]
```

```
from sklearn.metrics import accuracy_score

print(accuracy_score(est_sig, modulation_inds))
```

```
1.0
```