Network Analysis

Individual Assignment 1

Hans Alberto Franke

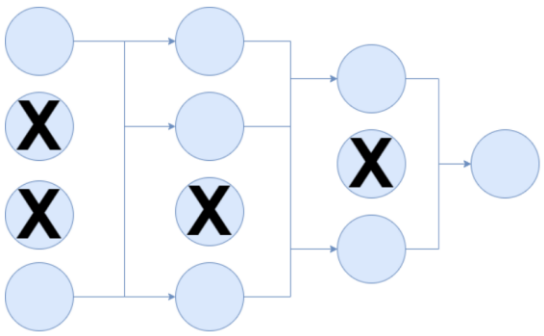
Student ID: 6987680

Feb/21

- Describe the principles of overfitting and how dropout can reduce this (Question 1, 5 points)

Overfitting means that one is being too sensitive to the used training data. In other words, the models prediction is "too good" on the training set, but not as good with real data or even the test set. Overfitting should be avoided as much as underfitting as your model will perform worse than it could perform theoretically. Fortunately, certain techniques – called regularizes – can be used to reduce the impact of overfitting. **Dropout** is one of them.

Using **Dropout**, the training process essentially drops out neurons in a neural network. They are temporarily removed from the network, which can be visualized as follows (image bellow). This removal neurons and synapses during training is performed at random, with a parameter that is tunable (p.e % of neurons dropped).



Note that the connections or synapses are removed as well, and that hence no data flows through these neurons anymore.

Source: https://www.machinecurve.com/index.php/2019/12/16/what-is-dropout-reduce-overfitting-in-your-neural-networks/

As (Srivastava et al., 2014) defines, evaluating the gradient is done with respect to the error, but also with *respect to what all other units are doing*. This means that neurons may fix the mistakes of other neurons by changing their weights. This behavior may lead to co-adaptations that may not generalize to new data, resulting in overfitting. **Dropout**, then, prevents these co-adaptations by making the presence of other hidden [neurons] unreliable. That way neurons cannot depend on other units to fix their mistakes,

which minimizes the number of co-adaptations that do not generalize to unseen data and therefore reduces overfitting.

Dropout prevents **overfitting** due to a layer's "over-reliance" on a few of its inputs, once these inputs aren't always present during training (i.e. they are dropped at random), the layer learns to use all of its inputs which causes to an improvement of the model and better generalizability.

○ Write a short (~500 word) summary of the experimental approach and results. (Question 2, 10 points)

The article compares a neural network with biological functions of humans and monkeys in task of easily recognize objects in scenes. This behavior is known as network of hierarchically connected brain areas. The authors looks to use computational techniques to identify a neural network that matches human performance. First, they construct a scenario considering of natural categories, the objects are putted on random natural scenes to ensure the background is uncorrelated with object.

Using a technique called multiple electrode arrays, they gathered reactions from 168 IT neurons to each image, then using high-throughput computational methods to assess a huge number of candidate neural network model on these images, estimating object categorization performance x IT neural predictive for each model.

The modelling of CNNs attempt to approximate the general retinotopic organization of the ventral stream utilizing spatial convolution, like how our brain work, with estimations in any one region of the visual field identical to those somewhere else. Each of this convolutional layer is made of simple and neuronally basic operations, including *linear filtering, thresholding, pooling, and normalization*. So these layers can be stacked hierarchically to construct deep neural networks.

The numbers of layers in Network range from one to three, and filter weights for each layer were picked randomly from uniform distributions whose limits were model parameters. Models were chosen for evaluation by one of three procedures: *(i) random sampling of the uniform distribution over parameter space. (ii) optimization for performance on the high-variation eight-way categorization task and (iii) optimization directly for IT neural predictivity*. They evaluate the models performance using SVM, through cross-validating to test accuracy

They evaluate the performance of the network in different difficult scenarios, from simple image position to high(p.e 180° rotations on all axes, 2.5× dilation, and full-frame translations..). The comparison was made with human performance and other networks, to proper evaluate performance and find clues where the model can be improved.

After the comparison, they look to expand network performance with a combination of Deep NN which try to mimic the human brain, as the specialized subregions. To achieve that, they proposed hierarchical modular optimization (HMO) procedure that embodies a simple hypothesis for how high-performing combinations of functionally specialized hierarchical architectures can be efficiently discovered and hierarchically combined, without needing to prespecify the subtasks ahead of time.

Algorithmically, HMO is similar to an adaptive boosting procedure interleaved with hyperparameter optimization.

To acquire more information, an exploratory investigation of the parameters of the learned HMO model, surveying each parameter both for how diverse it was between model mixture components and how sensitively it was tuned. Two classes of model parameters were especially sensitive and diverse: *(i) filter statistics, including filter mean and spread, and (ii) the exponent trading off between max-pooling and average-pooling*. Evaluating a performance-optimized model to these data would provide a strong test both of its ability to predict the internal structure of the ventral stream, as well as to go beyond the direct consequences of category selectivity, these evidences suggest that they can identify intermediate structure in hierarchical models.

Their results can be added in neuroscience, because is a common thoughtful in it, the tuning curves of neurons in lower cortical areas will be a necessary precursor to explaining higher visual cortex, the results indicate that it is useful to complement this bottom-up approach with a top-down perspective characterizing IT as the product of an evolutionary/developmental process that selected for high performance on recognition on tasks like those used in their optimization.

- Play around with these settings and see how they affect your ability to learn classification of different data sets. Write down what you found and how you interpret the effects of these settings. This question is intentionally open to allow you to explore the process. (Question 3, 8 points)

The website is a visual representation of a formal neural network. First, as question states we can select only **classification** problems. Besides classification one can select different parameters, just like one would be able to choose them in Keras to create a network. The tensorflow playground gives a visual example which makes it possible to see the impacts of changing parameters in real time and aids with a better understanding of the meaning of each of these parameters and how they can impact the performance (speed/epochs) and accuracy (loss function) of a model.

Below, I define each of the mentioned parameters:

- **Data sets:** summary of how data are distributed, and what kind of problem the network will try to solve. This is important because many real world problems may be similar to this kind of classifications, so how to predict correct in each of these distributions. The feature selection depends on the kind of data given. They are categorized as:
  - Circle
  - Exclusive Or
  - Gaussian
  - Spiral
- **Parameters**:
  - Learning rate

        - The amount that the weights are updated during **training** is referred to as the step size
        - It is important parameter because if it too large it can be jumping around local optimal or NOT converge at all.
    - Activation function
        - is a **function** that is added into an artificial **neural network** in order to help the network **learn** complex patterns in the data. When comparing with a neuron-based model that is in our brains, the **activation function** is at the end deciding what is to be fired to the next neuron
        - **Funcions**: Rectified Linear Units (ReLU), Linear Activation (Linear), a hyperbolic activation function (Tanh) and Sigmoid. They have different purposes to simulate relations, for example **ReLU** try to add non-linearity, **Sigmoid** (sometimes called logistic) transforms the input in values between 0 and 1. **Tanh**, performs well when your classification have only 2 classes. **Linear**, as the name suggest build linear relation between nodes.
    - Regularization:
        - From the formula of **L1 and L2 regularization**, **L1 regularization** adds the penalty term in cost function by adding the absolute value of weight(Wj) parameters, while **L2 regularization** adds the squared value of weights(Wj) in the cost function
        - **Regularization**, significantly reduces the variance of the model, without substantial increase in its bias. ... As the value of **lambda** rises, it reduces the value of coefficients and thus reducing the variance. It is user to prevent **overfitting.**
    - Regularization Rate
        - Model developers tune the overall impact of the regularization term by multiplying its value by a scalar known as **lambda** (also called the **regularization rate**).
    - Ratio of training and test
        - Split between train and test , or size of each one in the model. P.e 80% training and 20% from test
    - Input Features
        - It provides 7 features or inputs– X1, X2, Squares of X1X2, Product of X1X2 and sin of X1X2. It plays a major role in feature engineering.
    - Batch size
        - How many rows per batch to be used in training iteration (update the weights).
- **Modeling**:
    - Number of hidden layers
        - It defines the network complexity modelling different patterns and relations between the data.
    - Number of neurons in layers
        - Number of nodes, that is used to simulate a neuron that fires or not depending on the activation function and the input of previous layer.
- **Extra**:

    - Noise => to simulate real input problems (f.e. wrong inputs, outliers, ..), data patterns become more irregular as the noise increase
- **Output**:
    - Loss: in train and test (accuracy performance, this loss value is used in backpropagation to update the weights of layers/filters)
    - Visual plot with predict values (and if you click the real values ("show test data")

The colors of the plot represent positive (blue) or negative (orange) impact. In the hidden layers, the lines are colored by the weights of the connections between neurons. Blue shows a positive weight, which means the network is using that output of the neuron as given. An orange line shows that the network is assigning a negative weight. One can look at lines to understand how backpropagation is updating the weights of the network based on the loss function. If one puts the cursor over a node, it shows the impact of that node on the final output. The size of the line visually represents the weights.

The interface of the website allows one to change the parameters and simulate a neural network. Therefore one can compare how fast a network converges (number of epochs) to a desired loss output.

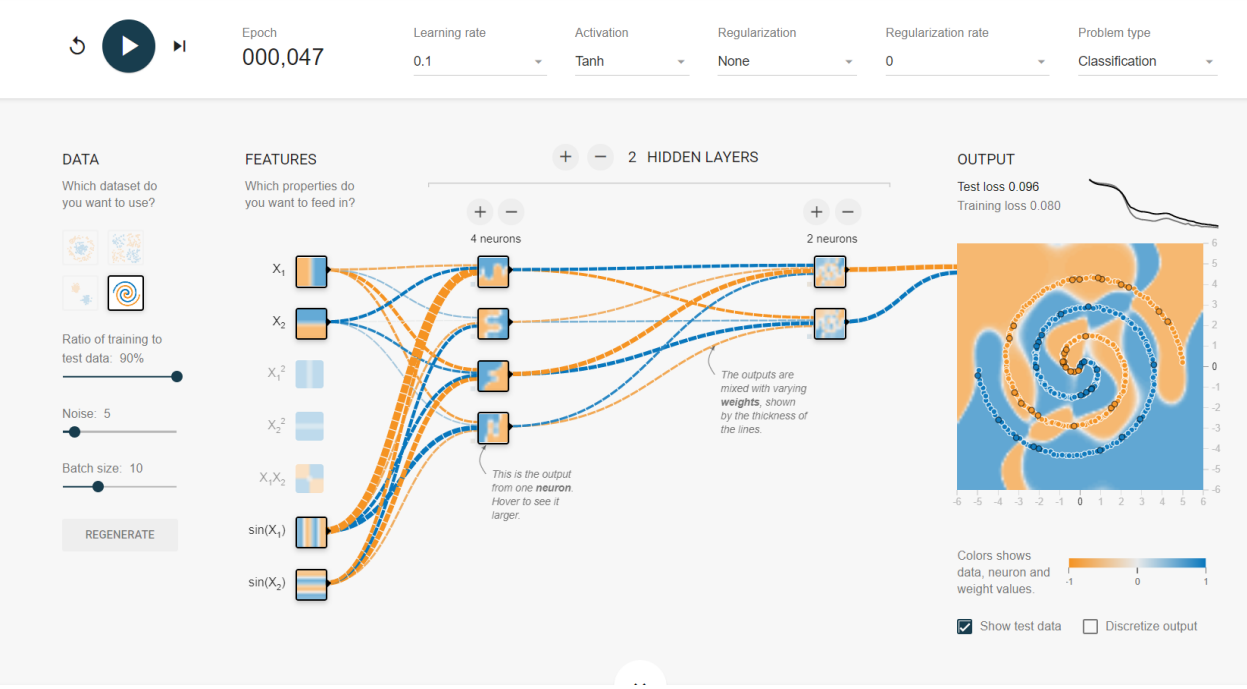
o What is the minimum you need in the network to classify the spiral shape with a test set loss of below 0.1? (Question 4, 7 points)

I'm considering **minimum cost** as achieving a loss threshold (<0.1) with the minimum number of epochs and a model as simple as possible (f.e minimum number of nodes).

Main choices of my model:

- **First step, was to simulate creating new features**. The most suitable features identified to categorize the coordinates to spherical coordinates were the sin(x1) and sin(x2). So, with ~50 epochs it achieved a test loss < 0.1.
- **Activation function**:
    - Tanh, selected because was the fastest to converge
    - Linear, converge FAST but never reach the target
    - Sigmoid, takes a long time to converge (+ 230 epochs)
    - RelU, never reaches the threshold.
- Number of neurons:
    - First layer: 4 (less than that the converge rate almost double)
    - 2nd layer: 2 (if not the converge rate takes almost more 100 epochs, 4 neurons don't affect performance too much so avoided to turn network simpler)
- I do not use a **regularization rate** because the model don't seems overfitted (difference in plot from training to test loss)

- For smaller batch sizes like 1, the learning rat of 0.1 is too high as the model fails to converge and jumps around the global minima. So, if one would like to keep a high **learning rate**(0.1), one needs the batch size to be high(10) as well. This usually gives a slow yet smoother convergence.
- **Ratio of training** to test data: 90% it converge fast and almost same overfitting than with 70% (difference from training loss to test loss)
- **Noise** => 5 (higher than this the model takes long to converge and overfitting becomes an issue as the loss in the test data almost doubles), if one removes the noise the model converge faster and with less variation between loss in train and test data.



Source:

http://playground.tensorflow.org/#activation=tanh&batchSize=10&dataset=spiral&regDataset=reg-plane&learningRate=0.1&regularizationRate=0&noise=5&networkShape=4,2&seed=0.19939&showTestData=true&discretize=false&percTrainData=90&x=true&y=true&xTimesY=false&xSquared=false&ySquared=false&cosX=false&sinX=true&cosY=false&sinY=true&collectStats=false&problem=classification&initZero=false&hideText=false

- Explain the principle of backpropagation of error in plain English in about 500 words. This can be answered with minimal mathematical content and should be IN YOUR OWN WORDS. What is backpropagation trying to achieve, and how does it do so? (Question 5, 8 points)

If one was interested in the final grade in the course Network Analysis it could be shown as the function of the grade of the individual assignment and the group assignment. An algorithm do decide on that needs training data, p.e. grades of the other students. Then one can try to predict one specific grade (y^) with individual inputs for instance if my grade on individual assignment (x1) and my grade on group assignment(x2).

In any ANN (artificial neural network) we have the following items: Neurons and Weights. Neurons store the values that will be calculated to define the Weights, where these weights are the "key" to the functioning of all ANN,. With the weights an ANN can identify for instance, whether an object is round and not square or as introduced in the previous example, what my predicted grade will be. I will illustrate the effect of backpropagation with an example in a simple multiple-linear regression.

Every time one submits an input(Student 1 (x1 = 5, x2 = 6)) and a certain output for this (student 1 y = 6), it regulates the weights (which are the lines that connect the neurons) to try to get as close as possible to the real result y. This relation can be mapped as (f(y) = W1.x1 + W2.x2 + B), where B = intercept or bias. The way that ANN learns its **weights**, or solve this equation, can be expressed by gradient descend. In other words, it minimizes the **loss function** of y for y^. What **backpropagation** does, is propagate this error backwards in the network, therefore the layers weights are adjusted based on their contribution on the overall error, by updating network parameters in small amount in the direction opposite to the error gradient.

In a neural network, its weights are set for its individual elements, called neurons. Inputs are loaded, they are passed through the network of neurons, and the network provides an output for each neuron depending on the initial weights (feedforward). The initial weights are random numbers. Essentially, backpropagation evaluates the expression for the derivative of the cost function as a product of derivatives between each layer from left to right – "backwards" – with the gradient of the weights between each layer being a simple modification of the partial products (the "backwards propagated error"). **Back-propagation** is the essence of neural network training. It is the practice of fine-tuning the weights of a neural net based on the error rate (i.e. loss) obtained in the previous epoch (i.e. iteration). Proper tuning of the weights ensure lower error rates, making the model reliable by increasing its generalizability.

Example: See figure1.

- **Inputs**:
  - Individual assignment = x1
  - Group assignment = x2
- **Hidden Layer** of 2 **nodes** (N1 and N2) and 4 **weights** where **a** = outputs
  - W11 = link from x1 to N1
  - W21 = link from x2 to N1
  - W12 = link from x1 to N2
  - W22 = link from x2 to N2
  - **N1 =>** a1 **=** w11*x1 + w21*x2
    - **F(a1) = a1** (using identity activation function)
  - **N2 =>** a2 **=** w12*x1 + w22*x2
    - **F(a2) = a2**

- **Output** layer (O1) = 1 node => predicted grade (**y^**)
    - W5 = link from N1 to O1
    - W6 = link from N2 to O1
    - **y^ =** a1*w5 + a2*w6
        - F(y^) = y^
- **Error / Cost Function**
    - **Error/Loss: E =** y^- y
- **activation function** that determines the activation value at every node in the neural net. For simplicity, let's choose an identity activation function: **f(a) = a**
- **Formula of partial derivative or *delta*:**
    - delta_0 = w * delta_1 * f`(z) , where *delta = total_loss = E*, and *delta_1* is on the **right** and delta_0 to the **left**, and *f`(z)* is the *derivate* of activation function, as identity *f`(z) = f(z), w* is the weight.

Example data:

- W5 = W5 – alpha * y^ * delta(y^)
- W1 = W1 – alpha * a1* delta(a1)
- Or, Wi = Wi – alpha * ai * delta(ai), where ai is always the result in the node on the **right** of the Wi. Alpha is the learning rate

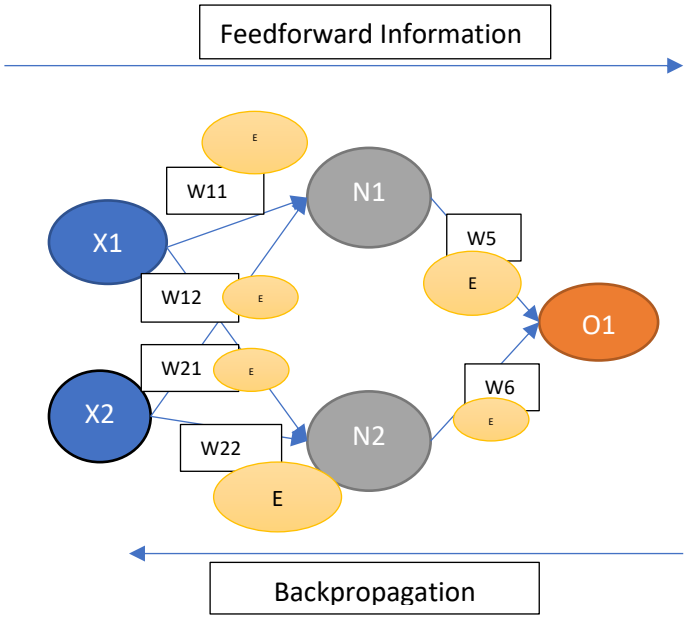
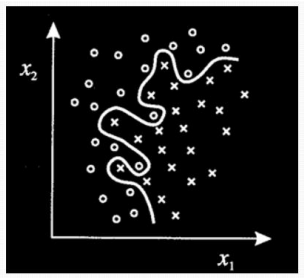


Figure1. **Backpropagation**: The sizes of yellow circles represents the contribution of each weight(W) based on the overall error

After one epoch is finished and the weights are updated there is another round of calibration, evaluation and backpropagation. This is done until the model "converges". The end of the process could for instance to minimize loss function until some threshold is reached or a predefined number of epochs. Each round is defined or by one row, or by a batch of rows (batch gradient).

These assumptions are made trying to simulate neuron of human brain, or in other words, adjusting weights so that a certain neuron fires or does not fire depending on the added value that it has to the network. This weights in the network are used to predict the new value, in our example this would be, the final grade.

The main advantage of using Backpropagation is that it works with multiple layers and solves "nonlinearly separable" problems that many other algorithms do not solve. A "nonlinearly separable" problem is one where we cannot separate 2 distinct classes on the two-dimensional Cartesian axis just by tracing a line.



Another important feature is that Backpropagation is feedforward, that is, the connection between neurons is not cyclic, going from start to finish you will not find a cycle.

BONUS QUESTION:

- Describe the process of backpropagation in mathematical terms. Here, explain (in English, in about 500 words) what each equation you give does, and relate this to the answers given in Question 5. You are welcome to express equations in your own R or python code rather than using equation layout, but you need to make clear you understand what each line is doing. (Question 6, 5 points).

```
import numpy as np

# define the sigmoid function (activation function)
# Backpropagation is actually a major motivating factor in the historical use of sigmoid activatio
n functions due to its convenient derivative:
def sigmoid(x, derivative=False):

    if (derivative == True):
        return sigmoid(x,derivative=False) * (1 - sigmoid(x,derivative=False))
    else:
        return 1 / (1 + np.exp(-x))


# learning rate = by which rate the parameters/weights are updated
alpha = .1

# number of nodes in the hidden layer
```

```
num_hidden = 2

# inputs (grade of individual assignment and grade of group assignment)
# Generate random inputs from Student 1 to student n (number of rows)
number_students = 15
X = np.random.randint(10, size=(number_students, 2))

# outputs
# y.T is the transpose of y, making this a column vector
y = np.random.randint(10, size=(number_students, 1)).T

# initialize weights randomly with mean 0 and range [-1, 1]
# the +1 in the 1st dimension of the weight matrices is for the bias weight
hidden_weights = 2*np.random.random((X.shape[1] + 1, num_hidden)) - 1
output_weights = 2*np.random.random((num_hidden + 1, y.shape[1])) - 1

# number of iterations of gradient descent  / number of epochs
num_iterations = 10000

# for each iteration of gradient descent
for i in range(num_iterations):

    # forward phase
    # np.hstack((np.ones(...), X) adds a fixed input of 1 for the bias weight
    input_layer_outputs = np.hstack((np.ones((X.shape[0], 1)), X))
    #use activation function
    hidden_layer_outputs = np.hstack((np.ones((X.shape[0], 1)), sigmoid(np.dot(input_layer_outputs
, hidden_weights))))
    #dot product of hidden layers * weights
    output_layer_outputs = np.dot(hidden_layer_outputs, output_weights)

    # backward phase
    # output layer error term
    output_error = output_layer_outputs - y
    # hidden layer error term
    # [:, 1:] removes the bias term from the backpropagation
    hidden_error = hidden_layer_outputs[:, 1:] * (1 - hidden_layer_outputs[:, 1:]) * np.dot(output
_error, output_weights.T[:, 1:])

    # partial derivatives
    hidden_pd = input_layer_outputs[:, :, np.newaxis] * hidden_error[: , np.newaxis, :]
    output_pd = hidden_layer_outputs[:, :, np.newaxis] * output_error[:, np.newaxis, :]

    # average for total gradients
    total_hidden_gradient = np.average(hidden_pd, axis=0)
    total_output_gradient = np.average(output_pd, axis=0)

    # update weights
    hidden_weights += - alpha * total_hidden_gradient
    output_weights += - alpha * total_output_gradient

# print the final outputs of the neural network on the inputs X

#transform output to predict values
y_hat = np.mean(output_layer_outputs, axis=0)
y_hat = y_hat.reshape(1,number_students)


#print("Output After Training: \n{}".format(output_layer_outputs))

print("Matrix of grades-assigments:X1,X2 \n")
```

```
print("Matrix of grades-EXAMs: Exam or Y \n")
print("Number of students:", number_students)
print("\n X1 , X2, EXAM, Y_HAT")
print(np.hstack((X, y.T, y_hat.T)))
#Total error after 1000 iterations
print('Total error',np.sum(output_error))
```

```
#print("Output After Training: \n{}".format(output_layer_outputs))

print("Matrix of grades-assigments:X1,X2 \n")
print("Matrix of grades-EXAMs: Exam or Y \n")
print("Number of students:", number_students)
print("\n X1 , X2, EXAM, Y_HAT")
print(np.hstack((X, y.T, y_hat.T)))
#Total error after 1000 iterations
print('Total error',np.sum(output_error))
```

```
Matrix of grades-assigments:X1,X2

Matrix of grades-EXAMs: Exam or Y

Number of students: 15

 X1 , X2, EXAM, Y_HAT
[[ 4.00000000e+00  5.00000000e+00  6.00000000e+00  5.99999726e+00]
 [ 5.00000000e+00  7.00000000e+00  2.00000000e+00  1.99999818e+00]
 [ 1.00000000e+00  1.00000000e+00  4.00000000e+00  3.99999584e+00]
 [ 3.00000000e+00  0.00000000e+00  8.00000000e+00  7.99999704e+00]
 [ 7.00000000e+00  9.00000000e+00  2.00000000e+00  1.99999732e+00]
 [ 9.00000000e+00  6.00000000e+00  4.00000000e+00  4.00000005e+00]
 [ 7.00000000e+00  7.00000000e+00  2.00000000e+00  1.99999938e+00]
 [ 6.00000000e+00  5.00000000e+00  2.00000000e+00  1.99999870e+00]
 [ 3.00000000e+00  1.00000000e+00  0.00000000e+00 -1.12524817e-06]
 [ 2.00000000e+00  2.00000000e+00  4.00000000e+00  3.99999963e+00]
 [ 8.00000000e+00  9.00000000e+00  4.00000000e+00  4.00000106e+00]
 [ 5.00000000e+00  3.00000000e+00  9.00000000e+00  8.99999460e+00]
 [ 8.00000000e+00  8.00000000e+00  5.00000000e+00  4.99999639e+00]
 [ 3.00000000e+00  6.00000000e+00  7.00000000e+00  6.99999863e+00]
 [ 8.00000000e+00  1.00000000e+00  1.00000000e+00  1.00000159e+00]]
Total error -0.00038192668749112424
```

```
[ ]:
```