Deep Neural Networks for Linear Sum Assignment Problems

EE5140: Project Review

Bharati. K (EE20D400)
Nitin. P. Shankar (EE20D425)
Tejaram Sangadi (EE20D426)

Indian Institute of Technology, Madras

December 16, 2020

Presentation outline

Table of Contents

An Illustrative Introduction

Many at times, we come across situations in real life where we want to distribute a shared resource or an entity among a set of individuals and ensure that their requirements are met! How is this relevant to modern day wireless communication systems?

Figure: Resource allocation

Fig. 1 is adapted from
https://www.fm-magazine.com/news/2019/nov/resource-allocation-best-practices-201922378.html

Relevance to wireless communication systems

1. Spike in number of wireless users, range of devices and applications used.
2. High Internet speed, uninterrupted web access, seamless cellular communication - needs of the consumer receiving the service.
3. What is being shared? - The radio resources
4. One of the key design challenges in the next generation of wireless communication systems is to allocate radio resources in an optimal fashion.

Table of Contents

Radio resource allocation in wireless communication systems

1. Emerging 5G networks need to support broadband traffic (eMBB) and Ultra-Reliable Low-Latency Communication (URLLC) traffic, thus demanding fast-executing scheduling routines[1].
2. Multi-objective performance optimisation - maximise throughput, overall fairness, for example, PF scheduling[2].
3. Widely device to device (D2D) communications - optimal channel assignment and power control[3].
4. **Operational intelligence** Use ML/DL algorithms to allocate resources efficiently to achieve performance close to the optimum[4].
5. **Environmental intelligence** Intelligent adaptive wireless channel sensing mechanisms have paved to enable resilient, robust and reliable D2D communications[4].

Prior works related to LSAP

1. Kuhn has proposed the Hungarian algorithm by combining concepts in graph theory and the duality of linear programming, which is one of the first algorithms for LSAPs[5].
2. The parallelizable **auction** algorithm was proposed by Bertsekas[6].
3. Near-optimal solutions for the LSAP was derived using heuristic algorithms like greedy randomized adaptive algorithm and the deep greedy switching algorithm[7].
4. ML/DL techniques are now used to solve wireless resource management problems, and the authors of [8] have attempted to solve LSAP using DL techniques.

Table of Contents

The LSAP Problem

1. The LSAP is a classical combinatorial optimization problem that deals with assigning n jobs to n people subject to the following constraints.
2. Let c_{ij} be the cost of assigning job i to person j and $x_{ij} = 1$ stand for job i is assigned to person j. The cost matrix C is defined as $C = \{c_{ij}\}$ and the decision matrix X is defined as $X = \{x_{ij}\}$ where $i, j = 1, 2, ...n$.
3. The LSAP can be formulated as follows: [8]

$$
\begin{aligned}
\text{minimise} \quad & \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \\
\text{subject to} \quad & \sum_{i=1}^{n} x_{ij} = 1, \; j = 1, 2...n \\
& \sum_{j=1}^{n} x_{ij} = 1, \; i = 1, 2...n \\
& x_{ij} \in \{0, 1\}, \; i, j = 1, 2, ...n
\end{aligned} \tag{1}
$$

Towards Solving LSAP

The optimal solution to the LSAP is the decision matrix X that reduces the overall cost for assigning every job to every user, adhering to constraints. For every *nxn* cost matrix C (fed as input), the matrix X is produced.

How does one solve this?

1. Convex optimisation (CVX) tools.
2. The focus of the paper - Deep Neural Networks - the original assignment problem is broken down into sub-assignment problems.

Question: Sub-assignment problems would not have the same objective function as the original assignment. In a constrained optimisation setup, how do the constraints change?

The Sub-Assignment Problem

In this system model, the j^{th} one solves an assignment problem on how to assign one of n jobs to people j.

$$\begin{aligned} \text{input} \quad & vec(C) = [c_{11}, c_{12}.....c_{nn}] \\ \text{output} \quad & X_j = [x_{1j}, x_{2j}, ...x_{nj}] \\ \text{subject to} \quad & \sum_{i=1}^{n} x_{ij} = 1,\ j = 1, 2...n \\ & x_{ij} \in \{0, 1\},\ i, j = 1, 2, ...n \end{aligned} \tag{2}$$

It can not be guaranteed that i_{th} job can be assigned to exactly one user at a given time. There may exist cases where one job may be assigned to different people simultaneously. This is termed as a *collision*.

Using the Hungarian Algorithm to find optimal assignment

	W1	W2	W3
J1	30	16	12
J2	14	5	18
J3	17	35	7

Subtract row minima

	W1	W2	W3
J1	18	4	0
J2	9	0	13
J3	10	28	0

Subtract column minima

	W1	W2	W3
J1	9	4	0
J2	0	0	13
J3	1	28	0

Cover all zeros with a minimum number of lines. Number of lines = 2< size of the matrix

	W1	W2	W3
J1	9	4	0
J2	0	0	13
J3	1	28	0

Find the smallest uncovered no. and subtract that from all uncovered numbers and cover all zeroes

No. of lines = 3

	W1	W2	W3
J1	8	3	0
J2	0	0	13
J3	0	27	0

THE HUNGARIAN ALGORITHM

OPTIMAL ASSIGNMENT

	W1	W2	W3
J1	30	16	12
J2	14	5	18
J3	17	35	7

Figure: The Hungarian Algorithm

Table of Contents

The System Architecture

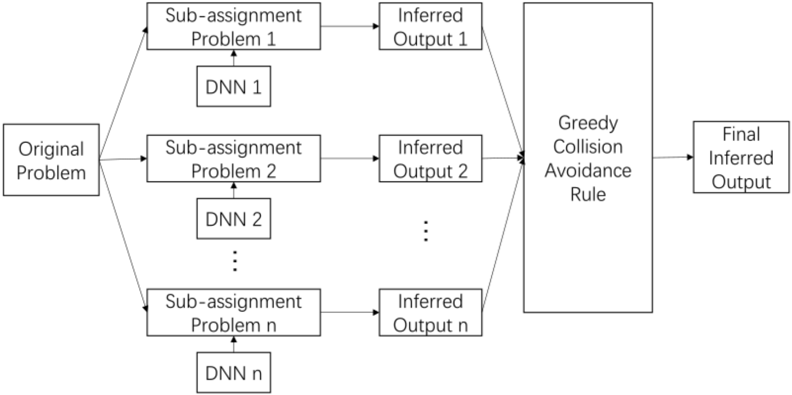

Figure: System model (taken from [8])

Greedy Collision Avoidance Rule

1. According to [8], the Greedy Collision (GC) Avoidance Rule is stated as follows: **If job i is assigned to persons j_1 and j_2 simultaneously, we assign job i to person j_1 when $c_{ij_1} < c_{ij_2}$.**
2. **Modified GC for 1 collision**: Let us assume that in the optimal Hungarian assignment, row k, $1 \leq k \leq n$ has no assignment and row $p \neq k$, $1 \leq p \leq n$ has entry 1 for columns j_1, j_2, $1 \leq j_1, j_2 \leq n$. If $(c_{p,j_1} + c_{k,j_2}) < (c_{p,j_2} + c_{k,j_1})$, then job p is assigned to j_1 and job k is assigned to j_2. If not so, then then job k is assigned to j_1 and job p is assigned to j_2.

```
Cost matrix 0th:
[[81  2 14 99]
 [ 4 48 74 58]
 [34 77 59 34]
 [97 59 70 50]]
```

```
Before Greedy:
 [[0 1 1 0]
 [1 0 0 0]
 [0 0 0 0]
 [0 0 0 1]]
```

```
After Greedy:
 [[0 1 0 0]
 [1 0 0 0]
 [0 0 1 0]
 [0 0 0 1]]
```

Figure: Greedy Collision Avoidance Rule

Data Generation

1. The cost matrix is generated by following the discrete uniform distribution with values ranging from [1,100).
2. The **linear_sum_assignment** module creates test/training samples using the Hungarian Algorithm.
3. The cost matrix is taken as an input and the optimal solution (X matrix) is the output for the assignment problem.
4. The X matrix is divided into columns (for each person j) in order to feed the N neural networks.
5. The generated samples are split into training and testing data.

Using DNNs for the Sub-assignment

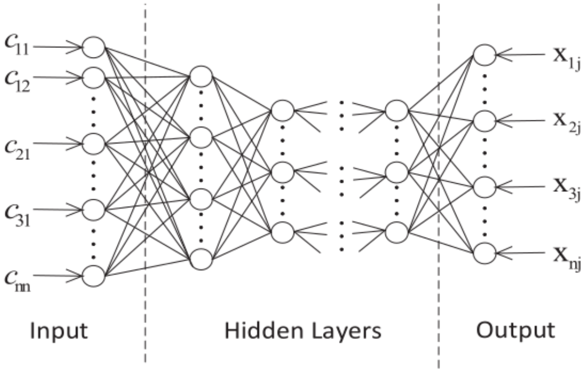

Figure: DNN structure for $j^{th} sub - assignment$

The Training Process

1. The cost matrix C is vectorized and fed as an input to the DNN.
2. The Hungarian algorithm is used to train the weights.
3. The error in the deduced solutions is minimised by updating the weights of the neurons.

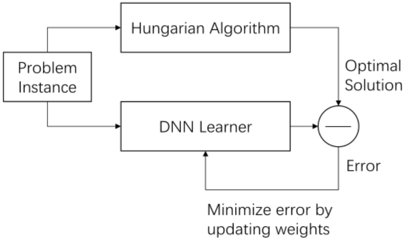

Figure: Resource allocation

Table of Contents

The Loss function - Cross Entropy

1. Based on the concepts from information theory.
2. Mostly used with softmax activation since it produces probabilities as output.
3. Used as a Loss function by comparing a target probability distribution with a predicted probability distribution.
4. Formula

$$L_j = -\frac{1}{M}\sum_{m=1}^{M}\sum_{i=1}^{n} X_{ij}^{(m)} log(y_{ij}^{(m)}) \tag{3}$$

where,
M = Batch size;
n = Problem size;
X = Target distribution;
y = Predicted distribution.
j denotes a particular sub-assignment problem.

L2 Regularization

1. The regularization loss is a function of all the weights in the j^{th} sub assignment problem.
2. It is mainly used to prevent over-fitting of the training data which can occur when the Neural Network becomes complex.
3. Formula

$$L_2 = \frac{\lambda}{2M} \sum_{\omega \epsilon \Omega_j} \omega^2 \tag{4}$$

where,
λ = Regularization parameter.
M = batch size;
ω = Weights of the Neural network.

4. The following term is added to the loss function in order to make the correction.

Adam Optimizer

Adam is an optimization algorithm that can be used instead of the classical stochastic gradient descent procedure to update network weights iterative based in training data.

Comes in the category of Adaptive Learning Rate Algorithms.

Builds upon the advantages of:

- AdaGrad
- RMSProp

Adam takes 3 hyperparameters:

1. Learning rate,
2. Decay rate of 1st-order moment
3. Decay rate of 2nd-order moment. [9]

Activation Functions Used

1. Sigmoid : The function can take any real value and map it value in between to 0 and 1. [10]. Logistic Sigmoid Function

$$S(x) = \frac{1}{1 + e^{-x}}$$

2. Relu : The function returns the input if it is greater than zero, else returns zero. [11]

$$f(x) = max(0, x)$$

3. Softmax : The function transforms input vector of real values into values which sum upto 1. [12]

$$\sigma(\overrightarrow{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_i}}$$

Figures 7 and 8 are taken from [8].

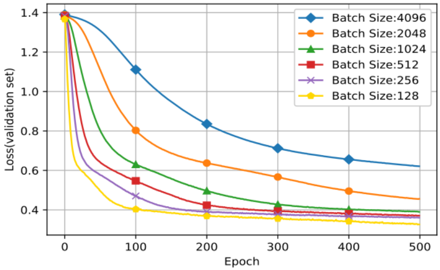

Figure: Batch Size selection

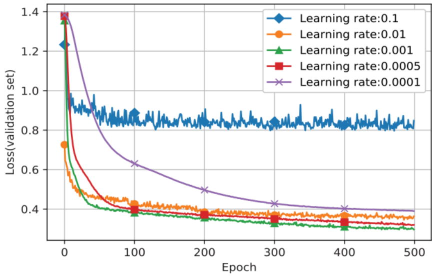

Figure: Learning rate Selection

Resources

The following instruments were used for performing the simulation:

- Google Colab
- Python Packages
  1. Tensorflow v2.3.0
     - Keras v2.4.0
  2. Numpy
  3. Scipy
     - linear_sum_assignment

Workflow

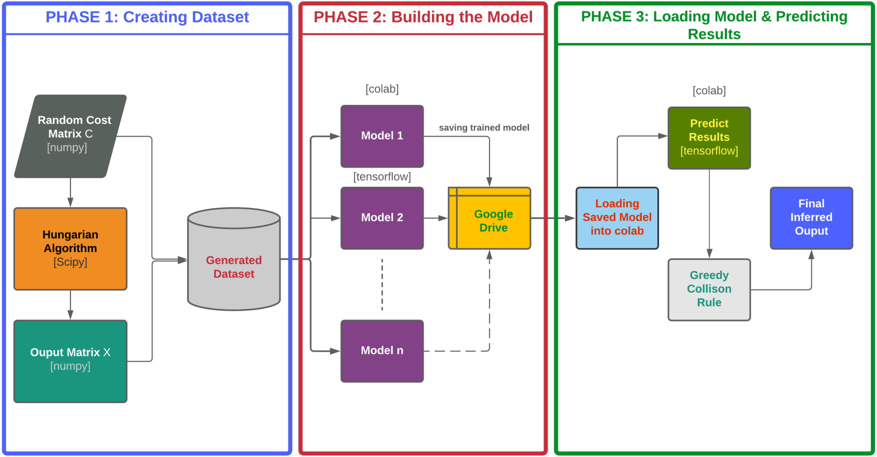

Figure: Phases Involved in Project

Table of Contents

Simulation results

```
# Saved Changes on Dec 1 ; 9.43PM
import timeit


setup_code = '''
from scipy.optimize import linear_sum_assignment
import numpy as np
n     = 4       # No. of Jobs, People
test = 5000    # No, of test samples
C_test = np.random.randint(1,100, size=(test,n,n))
X_test = np.zeros((test,n,n),dtype=int)
'''
statement = """
for ii in range(test):
    row_ind, col_ind = linear_sum_assignment(C_test[ii])
    X_test[ii,row_ind,col_ind] = 1
"""
num = 100
time = timeit.timeit(setup = setup_code, stmt = statement, number = num)
avg = time/num
print("Execution time for {} iterations is: {}".format(num,time))
print("Avg Time:{}".format(avg))
```

```
Execution time for 100 iterations is: 15.652672729000187
Avg Time:0.15652672729000186
```

Figure: Time taken for Hungarian Algorithm(($n = 4$)

Simulation results [contd.]

```
import time
# starting time
start = time.time()
g1= n_model1.predict(test_dataset1)
end = time.time()
# total time taken
print(f"Runtime of the program is {end - start}")
```

```
Runtime of the program is 0.023828983306884766
```

Figure: FNN with no overhead, for 1 model($n = 4$)*

Simulation results [contd.]

```python
import time
# starting time
start = time.time()
g1= n_model1.predict(test_dataset1)
g2= n_model2.predict(test_dataset2)
g3= n_model3.predict(test_dataset3)
g4 = n_model4.predict(test_dataset4)
end = time.time()
# total time taken
print(f"Runtime of the program is {end - start}")
```

```
Runtime of the program is 0.09605026245117188
```

Figure: FNN with no overhead($n = 4$)*

Simulation results [contd.]

1. Model prediction (In Sequence) and Greedy Collision Rule

```
import time
# starting time
start = time.time()
X_cap = np.zeros((test,n,n),dtype=int)
job = np.zeros((test,n),dtype=int)
p= [0, 1, 2, 3] # Persons {0..j}
count = np.zeros((test,1),dtype=int)
for s in range(test):
    job[s,p[0]] = np.argmax(g1[s])
    job[s,p[1]] = np.argmax(g2[s])
    job[s,p[2]] = np.argmax(g3[s])
    job[s,p[3]] = np.argmax(g4[s])
for s in range(test):
  for i in range(n-1):
    for j in range(i+1,n):
      if (job[s,p[i]]==job[s,p[j]]):
        count[s]+=1
for s in range(test):
  for i in range(n):
    X_cap[s,job[s,p[i]],p[i]] = 1
for s in range(test):
  if count[s]==1:
    for i in range(n-1):
      for j in range(i+1,n):
        #Greedy Collison Rule
        if (job[s,p[i]]==job[s,p[j]]):
            if (C_test[s,job[s,p[i]],p[i]] + C_test[s,r[s],p[j]] < C_test[s,job[s,p[j]],p[j]] + C_test[s,r[s],p[i]]):
                X_cap[s,job[s,p[i]],p[i]] = 1
                X_cap[s,job[s,p[j]],p[j]] = 0
                X_cap[s,r[s],p[j]]        = 1
                X_cap[s,r[s],p[i]]        = 0
            else:
                X_cap[s,job[s,p[j]],p[j]] = 1
                X_cap[s,job[s,p[i]],p[i]] = 0
                X_cap[s,r[s],p[i]]        = 1
                X_cap[s,r[s],p[j]]        = 0
end = time.time()
# total time taken
print(f"Runtime of the program is {end - start}")
```

```
Runtime of the program is 0.1512448787689209
```

Figure: FNN with overhead of Greedy Collision $(n = 4)$*

```
import time
# starting time
start = time.time()
g1= n_model1.predict(test_dataset1)
end = time.time()
# total time taken
print(f"Runtime of the program is {end - start}")
```

```
Runtime of the program is 0.03426003456115723
```

Figure: CNN with no overhead, for 1 model($n = 4$)*

```python
import time
# starting time
start = time.time()
g1= n_model1.predict(test_dataset1)
g2= n_model2.predict(test_dataset2)
g3= n_model3.predict(test_dataset3)
g4 = n_model4.predict(test_dataset4)
end = time.time()
# total time taken
print(f"Runtime of the program is {end - start}")
```

```
Runtime of the program is 0.09264516830444336
```

Figure: CNN with no overhead ($n = 4$)*

Simulation results [contd.]

Time taken for CNN

```
import time
# starting time
start = time.time()
X_cap = np.zeros((test,n,n),dtype=int)
job = np.zeros((test,n),dtype=int)
p= [0, 1, 2, 3] # Persons {0..j}
count = np.zeros((test,1),dtype=int)
for s in range(test):
    job[s,p[0]] = np.argmax(g1[s])
    job[s,p[1]] = np.argmax(g2[s])
    job[s,p[2]] = np.argmax(g3[s])
    job[s,p[3]] = np.argmax(g4[s])
for s in range(test):
  for i in range(n-1):
    for j in range(i+1,n):
      if (job[s,p[i]]==job[s,p[j]]):
        count[s]+=1
for s in range(test):
  for i in range(n):
    X_cap[s,job[s,p[i]],p[i]] = 1
for s in range(test):
  if count[s]==1:
    for i in range(n-1):
      for j in range(i+1,n):
        #Greedy Collison Rule
        if (job[s,p[i]]==job[s,p[j]]):
            if (C_test[s,job[s,p[i]],p[i]] + C_test[s,r[s],p[j]] < C_test[s,job[s,p[j]],p[j]] + C_test[s,r[s],p[i]]):
                X_cap[s,job[s,p[i]],p[i]] = 1
                X_cap[s,job[s,p[j]],p[j]] = 0
                X_cap[s,r[s],p[j]]        = 1
                X_cap[s,r[s],p[i]]        = 0
            else:
                X_cap[s,job[s,p[j]],p[j]] = 1
                X_cap[s,job[s,p[i]],p[i]] = 0
                X_cap[s,r[s],p[i]]        = 1
                X_cap[s,r[s],p[j]]        = 0
end = time.time()
# total time taken
print(f"Runtime of the program is {end - start}")
```

```
Runtime of the program is 0.1038973331451416
```

Figure: CNN with overhead of Greedy Collision ($n = 4$) *

```
Epoch 494/500
44/44 [==============================] - 0s 8ms/step - loss: 0.4409 - accuracy: 0.8466
Epoch 495/500
44/44 [==============================] - 0s 9ms/step - loss: 0.4408 - accuracy: 0.8466
Epoch 496/500
44/44 [==============================] - 0s 9ms/step - loss: 0.4409 - accuracy: 0.8466
Epoch 497/500
44/44 [==============================] - 0s 8ms/step - loss: 0.4401 - accuracy: 0.8464
Epoch 498/500
44/44 [==============================] - 0s 8ms/step - loss: 0.4395 - accuracy: 0.8472
Epoch 499/500
44/44 [==============================] - 0s 8ms/step - loss: 0.4389 - accuracy: 0.8478
Epoch 500/500
44/44 [==============================] - 0s 8ms/step - loss: 0.4388 - accuracy: 0.8485
5/5 [==============================] - 0s 5ms/step - loss: 0.4371 - accuracy: 0.8548
```

Figure: Accuracy for FNN ($n = 4$)

Simulation results [contd.]

```
1407/1407 [==============================] - 2s 2ms/step - loss: 0.1129 - accuracy: 0.9537 - val_loss: 0.2465 - val_accuracy: 0.9096
Epoch 489/500
1407/1407 [==============================] - 2s 2ms/step - loss: 0.1118 - accuracy: 0.9540 - val_loss: 0.2478 - val_accuracy: 0.9122
Epoch 490/500
1407/1407 [==============================] - 2s 2ms/step - loss: 0.1111 - accuracy: 0.9549 - val_loss: 0.2522 - val_accuracy: 0.9110
Epoch 491/500
1407/1407 [==============================] - 2s 2ms/step - loss: 0.1117 - accuracy: 0.9541 - val_loss: 0.2472 - val_accuracy: 0.9108
Epoch 492/500
1407/1407 [==============================] - 2s 2ms/step - loss: 0.1110 - accuracy: 0.9542 - val_loss: 0.2468 - val_accuracy: 0.9106
Epoch 493/500
1407/1407 [==============================] - 2s 2ms/step - loss: 0.1115 - accuracy: 0.9540 - val_loss: 0.2494 - val_accuracy: 0.9084
Epoch 494/500
1407/1407 [==============================] - 2s 2ms/step - loss: 0.1105 - accuracy: 0.9554 - val_loss: 0.2517 - val_accuracy: 0.9110
Epoch 495/500
1407/1407 [==============================] - 2s 2ms/step - loss: 0.1113 - accuracy: 0.9538 - val_loss: 0.2532 - val_accuracy: 0.9084
Epoch 496/500
1407/1407 [==============================] - 2s 2ms/step - loss: 0.1104 - accuracy: 0.9549 - val_loss: 0.2518 - val_accuracy: 0.9074
Epoch 497/500
1407/1407 [==============================] - 2s 2ms/step - loss: 0.1107 - accuracy: 0.9554 - val_loss: 0.2499 - val_accuracy: 0.9120
Epoch 498/500
1407/1407 [==============================] - 2s 2ms/step - loss: 0.1108 - accuracy: 0.9548 - val_loss: 0.2460 - val_accuracy: 0.9134
Epoch 499/500
1407/1407 [==============================] - 2s 2ms/step - loss: 0.1105 - accuracy: 0.9553 - val_loss: 0.2612 - val_accuracy: 0.9066
Epoch 500/500
1407/1407 [==============================] - 2s 2ms/step - loss: 0.1114 - accuracy: 0.9539 - val_loss: 0.2479 - val_accuracy: 0.9110
157/157 - 0s - loss: 0.2479 - accuracy: 0.9110
INFO:tensorflow:Assets written to: saved_model/model3/assets
0.9110000133514404
```

Figure: Accuracy for CNN ($n = 4$)

Simulation results [contd.]

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_3 (Conv2D)            (None, 8, 8, 2)           4
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 8, 8, 4)           12
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 8, 8, 8)           40
_________________________________________________________________
flatten_1 (Flatten)          (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 512)               262656
_________________________________________________________________
dense_3 (Dense)              (None, 8)                 4104
=================================================================
Total params: 266,816
Trainable params: 266,816
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
28125/28125 [==============================] - 164s 6ms/step - loss: 0.9694 - accuracy: 0.6235 - val_loss: 0.8661 - val_accuracy: 0.6434
Epoch 2/10
28125/28125 [==============================] - 163s 6ms/step - loss: 0.8424 - accuracy: 0.6530 - val_loss: 0.8120 - val_accuracy: 0.6642
Epoch 3/10
28125/28125 [==============================] - 173s 6ms/step - loss: 0.7574 - accuracy: 0.6883 - val_loss: 0.7325 - val_accuracy: 0.6986
Epoch 4/10
28125/28125 [==============================] - 167s 6ms/step - loss: 0.6885 - accuracy: 0.7169 - val_loss: 0.6682 - val_accuracy: 0.7250
Epoch 5/10
28125/28125 [==============================] - 163s 6ms/step - loss: 0.6583 - accuracy: 0.7300 - val_loss: 0.6501 - val_accuracy: 0.7336
Epoch 6/10
28125/28125 [==============================] - 165s 6ms/step - loss: 0.6415 - accuracy: 0.7374 - val_loss: 0.6360 - val_accuracy: 0.7391
Epoch 7/10
28125/28125 [==============================] - 163s 6ms/step - loss: 0.6321 - accuracy: 0.7418 - val_loss: 0.6320 - val_accuracy: 0.7408
Epoch 8/10
28125/28125 [==============================] - 165s 6ms/step - loss: 0.6276 - accuracy: 0.7439 - val_loss: 0.6322 - val_accuracy: 0.7411
Epoch 9/10
28125/28125 [==============================] - 163s 6ms/step - loss: 0.6245 - accuracy: 0.7451 - val_loss: 0.6327 - val_accuracy: 0.7406
Epoch 10/10
28125/28125 [==============================] - 163s 6ms/step - loss: 0.6221 - accuracy: 0.7464 - val_loss: 0.6249 - val_accuracy: 0.7429
3125/3125 - 5s - loss: 0.6249 - accuracy: 0.7429
0.7429400086402893
(100000, 8, 8, 1)
```

Figure: Accuracy for CNN ($n = 8$)

Simulation results [contd.]

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 16, 16, 2)         4
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 16, 16, 4)         12
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 16, 16, 4)         20
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 16, 16, 8)         40
_________________________________________________________________
flatten (Flatten)            (None, 2048)              0
_________________________________________________________________
dense (Dense)                (None, 1024)              2098176
_________________________________________________________________
dense_1 (Dense)              (None, 16)                16400
=================================================================
Total params: 2,114,652
Trainable params: 2,114,652
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
28125/28125 [==============================] - 901s 32ms/step - loss: 1.0965 - accuracy: 0.5807 - val_loss: 0.9470 - val_accuracy: 0.6287
Epoch 2/10
28125/28125 [==============================] - 885s 31ms/step - loss: 0.9463 - accuracy: 0.6280 - val_loss: 0.9350 - val_accuracy: 0.6321
Epoch 3/10
28125/28125 [==============================] - 878s 31ms/step - loss: 0.9350 - accuracy: 0.6317 - val_loss: 0.9250 - val_accuracy: 0.6350
Epoch 4/10
28125/28125 [==============================] - 882s 31ms/step - loss: 0.9207 - accuracy: 0.6366 - val_loss: 0.9271 - val_accuracy: 0.6329
Epoch 5/10
28125/28125 [==============================] - 882s 31ms/step - loss: 0.9016 - accuracy: 0.6438 - val_loss: 0.9225 - val_accuracy: 0.6364
Epoch 6/10
28125/28125 [==============================] - 890s 32ms/step - loss: 0.8773 - accuracy: 0.6533 - val_loss: 0.9177 - val_accuracy: 0.6355
Epoch 7/10
28125/28125 [==============================] - 883s 31ms/step - loss: 0.8483 - accuracy: 0.6649 - val_loss: 0.9118 - val_accuracy: 0.6400
Epoch 8/10
28125/28125 [==============================] - 891s 32ms/step - loss: 0.8169 - accuracy: 0.6774 - val_loss: 0.9201 - val_accuracy: 0.6356
Epoch 9/10
28125/28125 [==============================] - 879s 31ms/step - loss: 0.7866 - accuracy: 0.6897 - val_loss: 0.9190 - val_accuracy: 0.6380
Epoch 10/10
28125/28125 [==============================] - 903s 32ms/step - loss: 0.7583 - accuracy: 0.7013 - val_loss: 0.9252 - val_accuracy: 0.6367
3125/3125 - 17s - loss: 0.9252 - accuracy: 0.6367
0.6366900205612183
(100000, 16, 16, 1)
```

Figure: Accuracy for CNN ($n = 16$)

Comparison

	Hungarian Algorithm	CNN	FNN
Time	0.5916	0.0120	0.0040
Accuracy	100%	92.76%	90.80%

Table: Performance Comparison for Different Methods[8]

	Hungarian Algorithm	CNN	FNN
Time	0.1565	0.0342	0.0238
Accuracy	100%	91.10%	85.48%

Table: Performance Comparison for Different Methods in Simulation

Comparison [contd]

	CNN	Random	Accuracy Gain
n=4	92.76%	25%	3.71
n=8	77.8%	12.25%	6.21
n=16	65.7%	6.25%	10.512

Table: Comparison Of CNN With Random Assignment[8]

	CNN	Random	Accuracy Gain
n=4	91.10%	25%	3.644
n=8	74.29%	12.25%	6.06
n=16	63.67%	6.25%	10.187

Table: Comparison Of CNN Accuracy from Simulation

Future Works and Conclusion

1. Scalability aspects.
2. Designing a tighter GC Avoidance rule.
3. Improving the training of the DNNs and finding an optimal DNN architecture.
4. DNNs solve the LSAP in significantly less amount of time with a compromise in accuracy.

References I

J. Ma, A. Aijaz, and M. Beach, "Recent results on proportional fair scheduling for mmwave-based industrial wireless networks," 07 2020.

Hoon Kim and Youngnam Han, "A proportional fair scheduling for multicarrier transmission systems," *IEEE Communications Letters*, vol. 9, no. 3, pp. 210–212, 2005.

G. Yu, L. Xu, D. Feng, R. Yin, G. Y. Li, and Y. Jiang, "Joint mode selection and resource allocation for device-to-device communications," *IEEE Transactions on Communications*, vol. 62, no. 11, pp. 3814–3824, 2014.

S. Dang, O. Amin, B. Shihada, and M.-S. Alouini, "What should 6g be?," 11 2019.

H. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistic Quarterly*, vol. 2, 05 2012.

References II

D. Bertsekas, "The auction algorithm: A distributed relaxation method for the assignment problem," *Annals of Operations Research*, vol. 14, pp. 105–123, 12 1988.

A. Naiem, M. El-Beltagy, P. Ab, and Sweden, "Deep greedy switching: A fast and simple approach for linear assignment problems," 01 2009.

M. Lee, Y. Xiong, G. Yu, and G. Y. Li, "Deep neural networks for linear sum assignment problems," *IEEE Wirel. Commun. Lett.*, vol. 7, no. 6, pp. 962–965, 2018.

D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings* (Y. Bengio and Y. LeCun, eds.), 2015.

References III

DanB, "Rectified linear units (relu) in deep learning — kaggle." https://www.kaggle.com/dansbecker/rectified-linear-units-relu-in-deep-learning, 2017. (Accessed on 12/16/2020).

T. Wood, "Softmax function definition — deepai." https://deepai.org/machine-learning-glossary-and-terms/softmax-layer. (Accessed on 12/16/2020).

V. Martinek, "Cross-entropy for classification." https://towardsdatascience.com/cross-entropy-for-classification-d98e7f974451, 05 2020. (Accessed on 12/16/2020).

The End