REPORT ON MACHINE LEARNING HEURISTIC APPROXIMATION ON KNAPSACK AND TRAVELING SALESMAN PROBLEMS

BY

DANIEL FELIPE LOPEZ MORALES

Master of Science in Computer Science

Graduate College of the

Illinois Institute of Technology

Chicago, Illinois
Spring 2018 - Fall 2019

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

Symbol	Definition
v	Value of an object
x	Integer value
W	Maximum weight in knapsack
N	Number of nodes/points/objects
n	Number of inputs

ABSTRACT

This paper tackles combinatorial optimization problems using various types of artificial neural networks. The focus is on Knapsack 1-0 problem and Traveling Salesman Problem (TSP). We compare Multi-Layered Perceptrons (MLP), Convolutional Neural Networks (CNN), Parallel input MLP for the Knapsack problem. In addition we compare MLPs CNN, Parallel input CNNs, Pointer Networks and Dynamic programming with Pointer Networks. The TSP is trained and tested with a range of 80-20 on 5,10 nodes for MLPs CNN, Parallel input MLP, Parallel CNNs and 5,10,20 for Pointer Networks and Dynamic programming using Pointer Networks.

CHAPTER 1

INTRODUCTION

1.1 The Knapsack Problem

The knapsack problem is an optimization problem where given a knapsack with maximum capacity and a set of objects with weight and value, maximizes the value of the objects which fit the knapsack capacity. The knapsack problem can be stated as the linear programming as such:

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{n} v_i x_i \\ \text{subject to} \quad & \sum_{i=1}^{n} w_i x_i \leq W, \\ & x_j \in \{0,1\}, \quad \forall i \in N \end{aligned} \tag{1.1}$$

Where v, is the value of the object, w is the weight of the object and W is the size of the Knapsack x is a binary variable of 0 or 1 which dictates if the object is chosen or not. Therefore we maximize the value for a positive integer of objects.

1.2 Data Generation The common parameters of a 1-0 Knapsack are sack size, the number of objects and possible sizes and values to those objects. The optimum number of objects was selected by the brute force solution of permuting the sets and getting the minimum value of the sets. The weights have been pseudo random using random.randint() which uses a discrete uniform distribution.

We generated 29000 simulations with the following parameters: First, modified the number of objects available N and set the Knapsack size W constant 50 units of weight(Figure 1.1). The values and the weight were randomized using N was set from 10 to 300 and the number of items is increased by 10. Each data point in the graph is an average, maximum and minimum for 1000 instances of the same inputs.

Algorithm 1 Dynamic Knapsack

```
1:  FUNCTION knapsack(values, weights, capacity):
2:      counter <- 0
3:      for j <- 0 until the number of values:   # j is the column in the table
4:          value <-  value of j                 # set the values of the object
5:          weight <-  weight of j
6:          for i in range(1,  capacity + 1):    # i is the row in the table
7:
8:              IF weight > i:       #If the object is over the capacity remove it
9:                   table[i, j] <-  table[i, j - 1]
10:             ENDIF
11:             ELSE:                #Recur over accepted items and fill the table
12:                  table[i, j] <- max(table[i, j - 1],table[i-weight,j-1]+ value)
13:             ENDIF
14:         ENDFOR
15:     ENDFOR
16:     RETURN  table[-1][-1]         #Get optimum value from the table
17:  ENDFUNCTION
```

Time Complexity: O(nW) we check through all the n items until we have filled the knapsack capacity W.

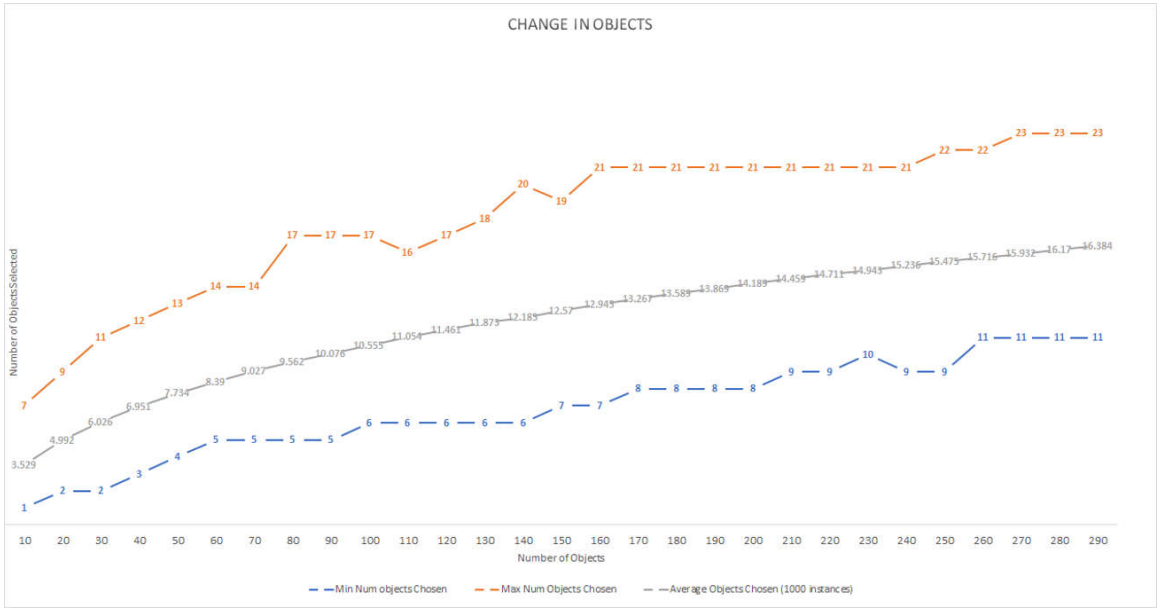

Figure 1.1. The knapsack items not chosen are Blue and the chosen items have Red stars.

Second, we modified the Knapsack size W and set the number of objects available N constant(Figure 1.2). The size of the sack was constant at 50 weight units. The values and the weight were randomized using N was set from 10 to 300 and the number of items is increased by 10. Each data point in the graph is an average, maximum and minimum for 1000 instances of the same inputs. This process also reflects a similar pattern as Figure 1.1.

Finally, we modified both the objects and the sack size we get an almost initial linear relationship as seen in Figure 1.3. These data allows us to have a minimum and maximum bound while having a sense of average knapsack growth. These bounds could be used for Branch and Bound dynamic programming and for better approximations with other methods.

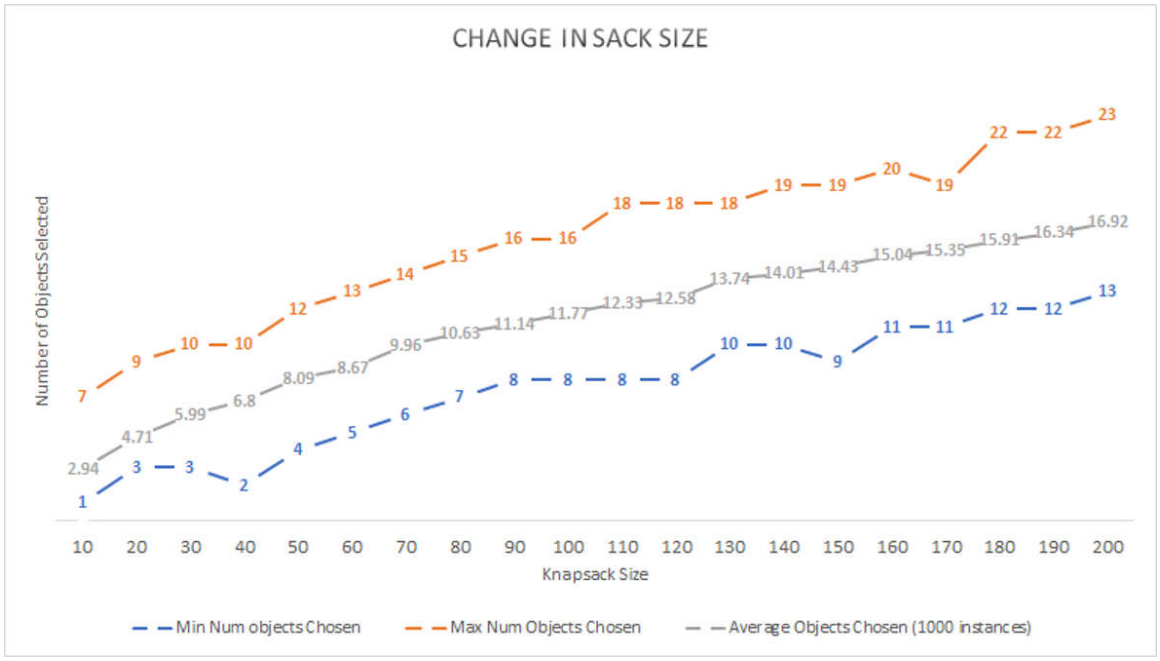

Figure 1.2. Trend given the modification only in the amount of objects.

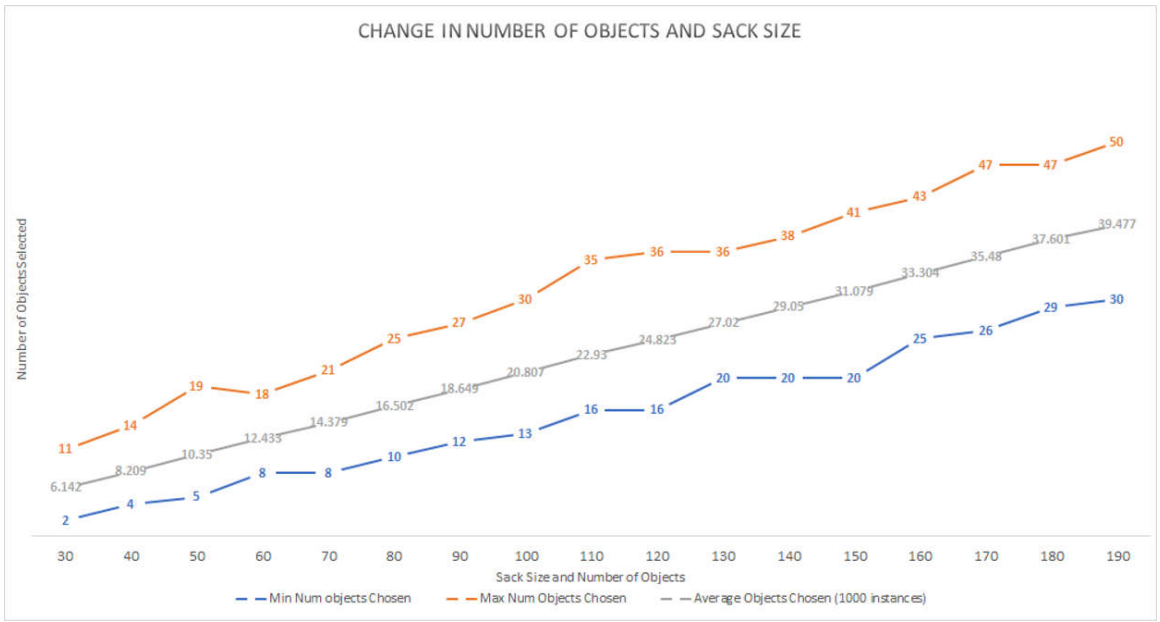

Figure 1.3. Trend given the modification only in the amount of objects.

1.2.1 Multi-Layer Perceptrons on Knapsack. Artificial Neural Networks are able to approximate complex models. To do this the neural networks are based on different layers. The three main layers are the input, hidden and output layers. The input layers consist of the input of the data that is being fed into the neural network. The hidden layer consist in a sum of the inputs (Eq. 1.2):

$$S_j = \sum_{i=1}^{n} \omega_{i,j} I_i + \beta_j \tag{1.2}$$

Where S_j is the sum for the j'th neuron, n is the number of inputs of the previous layer, I is the output of the previous layer (or the data in the input layer), ω is the weight of the connection and β is the bias value And an activation function such as the which we are using called Sigmoid:

$$f_j(x) = \frac{1}{1 + e^{-S_j}} \tag{1.3}$$

1.2.1.1 Training. The input layer will contain the vector of object values, however, since the objects contain weight and value, the vector is split in two and concatenated. Making it a 2N vector. This way each weight and value vector corresponds to its object(Hellstrom & Kanal,1992), therefore, for every N objects we get a 2N vector as input. The output to train for is made by knapsack optimum solution. The data, of 10000 of 10 - 60 objects and 50 as Knapsack size, was split on a 80% training 10% validation 10% test data. The neural network is made with Tensorflow Keras using 4 layers with ReLu activation, Adam as the optimizer with 0.0001 learning rate, and binary_crossentropy as the loss function. The model acquired a 86% accuracy.

1.2.1.2 Dynamic programming using Neural Network upper bound. As the upper bound is added, there is not much change when modifying the number of objects or the sack size. The accuracy maintains the relatively the same. The values are created by creating 1000 problems per each point and calculating the total value after value removal using a heuristic. And dividing by the dynamic knapsack total

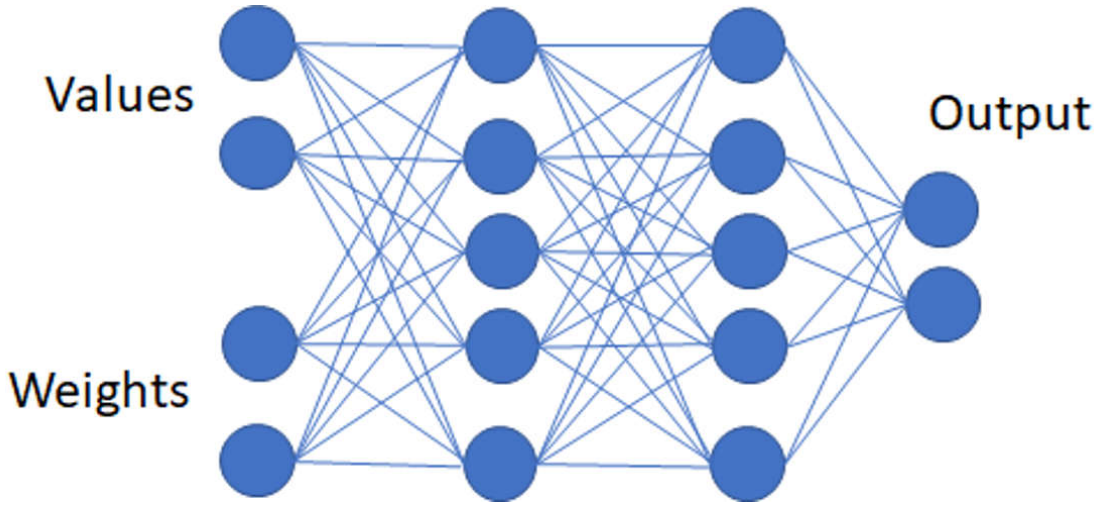

Figure 1.4. Neural Network Model used

value which can be seen in the Table 1.1.

Table 1.1. Heuristic values on Simple Neural Network

Loss	Accuracy	Heuristic	Real Value	Ratio	Time	Number of Items
0.2954	0.8758	141166	176898	0.7980	4.557836	10
0.4587	0.7803	119173	275310	0.4329	11.58403	20
0.4797	0.7815	97745	320458	0.4329	19.17868	30
0.4661	0.7942	105528	355489	0.2969	28.52071	40
0.5047	0.7817	26184	414311	0.0632	41.73136	50
0.4855	0.7957	22693	463741	0.0489	56.59059	60

CHAPTER 2

TRAVELING SALESMAN PROBLEM

2.1 Traveling Salesman Problem Introduction Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city once and returns to the original city. It is an NP-hard problem in combinatorial optimization, important in operations research and theoretical computer science. TSP can be asymmetric or directed:

$$\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{n} && c_{ij}x_{i,j} \\
\text{subject to} \quad & \sum_{i=1}^{n} && x_{ij} = 1, i = 0, 1, ..., n-1 \\
& \sum_{i=1}^{n} && x_{ij} = 1, j = 0, 1, ..., n-1 \\
& \sum_{i}\sum_{i} && x_{i,j} \leq |S| - 1, S \subset V, 2 \leq |S| \leq n-2 \\
& && x_{ij} \in \{0, 1\}, \forall i, j \in E
\end{aligned} \tag{2.1}$$

It can also be symmetric or undirected:

$$\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{n} && c_{ij}x_{i,j} \\
\text{subject to} \quad & \sum_{i<k}^{n} && x_{ik} + \sum_{j>k}^{n} x_{jk} = 2, k \in V1 \\
& \sum_{i}\sum_{i} && x_{i,j} \leq |S| - 1, S \subset V, 3 \leq |S| \leq n-3 \\
& && x_{ij} \in \{0, 1\}, \forall i, j \in E
\end{aligned} \tag{2.2}$$

Where x is a binary variable of 0 or 1 which dictates if the object is chosen or not. The S is the set of cities/points/nodes, V is vertices and E are edges.

2.1.1 Data Preparation. The adjacency matrix was created with a uniform

random variables from 0 to 100, and to normalize the matrices we divided by 100. The costs were divided by 1000 given the maximum value of the costs was 992. In the case of paths, when using softmax, the output was divided by n the length of the path. Also the accuracy in the neural networks was measured in different ways depending on the output. If the output is a single number, mean squared error was used. If the output was a path, the error of the cost of the path was used for the mean squared error.

Algorithm 2 Dynamic TSP solver

```
matrix <- adjacency matrix
best_tour <- []
current <- 0
cs <- set(range(1, length(matrix))) #Generate node ordered set
while True:
    l, lc <- rec_tsp_solve(current, cs)   #Use auxiliary function
    IF lc = 0:
        Break
    ENDIF
    best_tour.append(lc)            #add optimum node
    c <- lc
    cs <- cs - set([lc])            #remove optimum nodes from input set
ENDWHILE
best_tour <- tuple(reversed(best_tour)) #return optimum path
RETURN best_tour


FUNCTION tsp_dp_solve():
    FUNCTION memoize(f):
        memo_dict <- {}                         #get mapping
        FUNCTION memo_func(*args):
            IF args not in memo_dict:           #if node path not in map
                memo_dict[args] <- f(*args)   #add the node to the path and cost
            ENDIF
            RETURN memo_dict[args]              #return the mapping
        ENDFUNCTION
        RETURN memo_func
    ENDFUNCTION
    GOTO memoize using function(rec_tsp_solve) #Send parameters and result
    FUNCTION rec_tsp_solve(current, ts):      #check the cost of possible nodes
        IF ts:
            vals <- []
            for lc in ts:
                aval <- matrix[lc][current]                #Add current cost and
                        +rec_tsp_solve(lc,ts-set([lc]))[0] #recur next nodes
                vals.append((aval, lc))
            ENDFOR
            val <- min(vals)                #get minimum values from costs
            RETURN val
        ELSE:                               #if there are no more
            RETURN (matrix[0][c], 0)        #nodes return last node 0
        ENDIF
    ENDFUNCTION
ENDFUNCTION
```

Given we consider a set of n objects, every time we select an initial number, we will not use it again, therefore we have to consider a subset of n-1. After the next selection we have to check n-2. Each this recursive operation takes $O(n * 2^n)$ and each operation in the recursion takes linear time, therefore, the time complexity is $O(n^2 * 2^n)$.

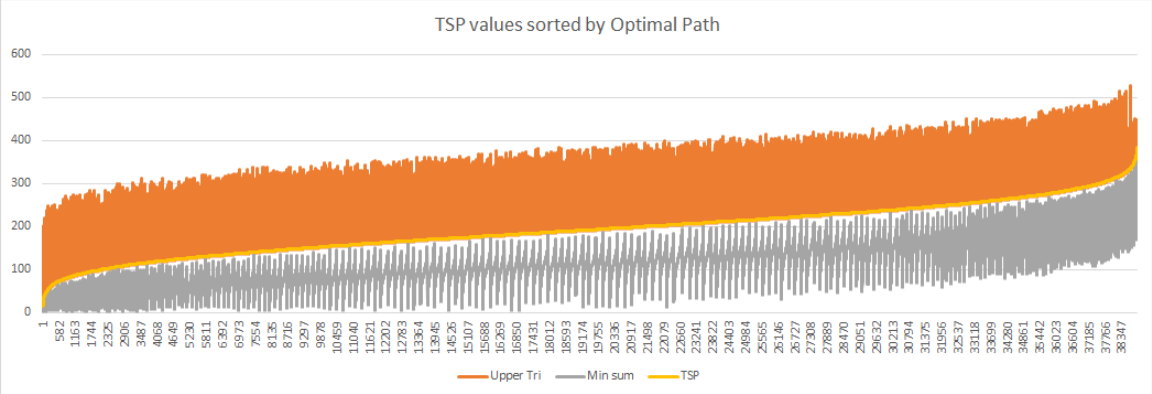

Figure 2.1. TSP costs sorted by optimum value

2.1.2 Data Analysis. 40000 instances of undirected adjacency matrices were created, where 0's are set as a large value of 999 to deter the tsp solver from allocating the diagonal or non existent paths. The matrices we used the dynamic programming TSP solver is found in python's tsp library (https://pypi.org/project/tsp/) . The optimum values gave us important insights. First, the sum of the minimum values of each column gives a lower boundary for the TSP and the sum of the upper triangle of the adjacency matrix gives a upper bound to the TSP which can be seen in Figure 2.1. Second, doing eigenvalue analysis of the TSP we fund that the sum of the 3 lowest eigenvalues on a 5x5 matrix predicts if the adjacency matrix has a solution. It can be seen in Figure 2.2 that the sum of the 3 lowest eigenvalues varies while there is a path for TSP, which is where the cost is less than 999. Once there is no path the sum of the 3 lowest eigenvalues drop significantly. However, it is hard to find a pattern once the every TSP as a path as we can see in Figure 2.3.

2.1.3 Dynamic programming with upper branch and bound. For this case, the data is created by generating an undirected graph with uniform distribution on the upper triangle mirrored by the lower triangle. We create the data by groups of 100000 data points of 5 to 50 cities with increments by 5. The upper bound is made by the sum of the elements of the upper triangular. Therefore, whenever a branch in

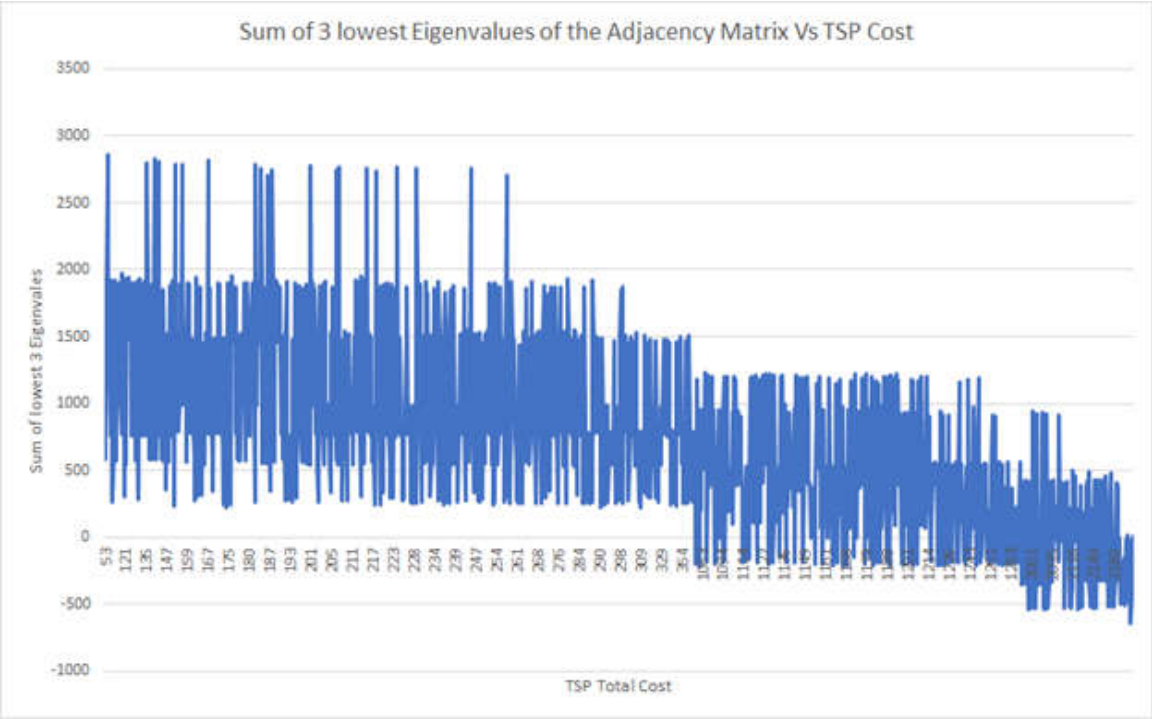

Figure 2.2. TSP costs on path and no path on small adjacency matrices

the dynamic program was higher than the sum of the upper triangular it is pruned. We can see how this upper triangular branch and bound helps in the time of finding the optimum TSP in Table 2.1.

We can note that the bigger the network the better the Branch and Bound dynamic programming does compared to the not bound dynamic programming.

2.1.4 Low Accuracy Test Networks. Here, we examine several suggested methods of neural networks which did not result in accuracy over 50%.

2.1.4.1 Fully connected MLPs. First, a MLP with n^2 inputs, 4 hidden layers with ReLu activation as: $2n^2, 4n^2, 4n^2, 2n^2$ and output layer of 1 sigmoid, using mean squared error (MSE) as the loss function and Adam as its optimizer. The output is compared directly to the TSP value. The second is the same as the first with a modification of the output layer. The output layer is has n neurons, with softmax we

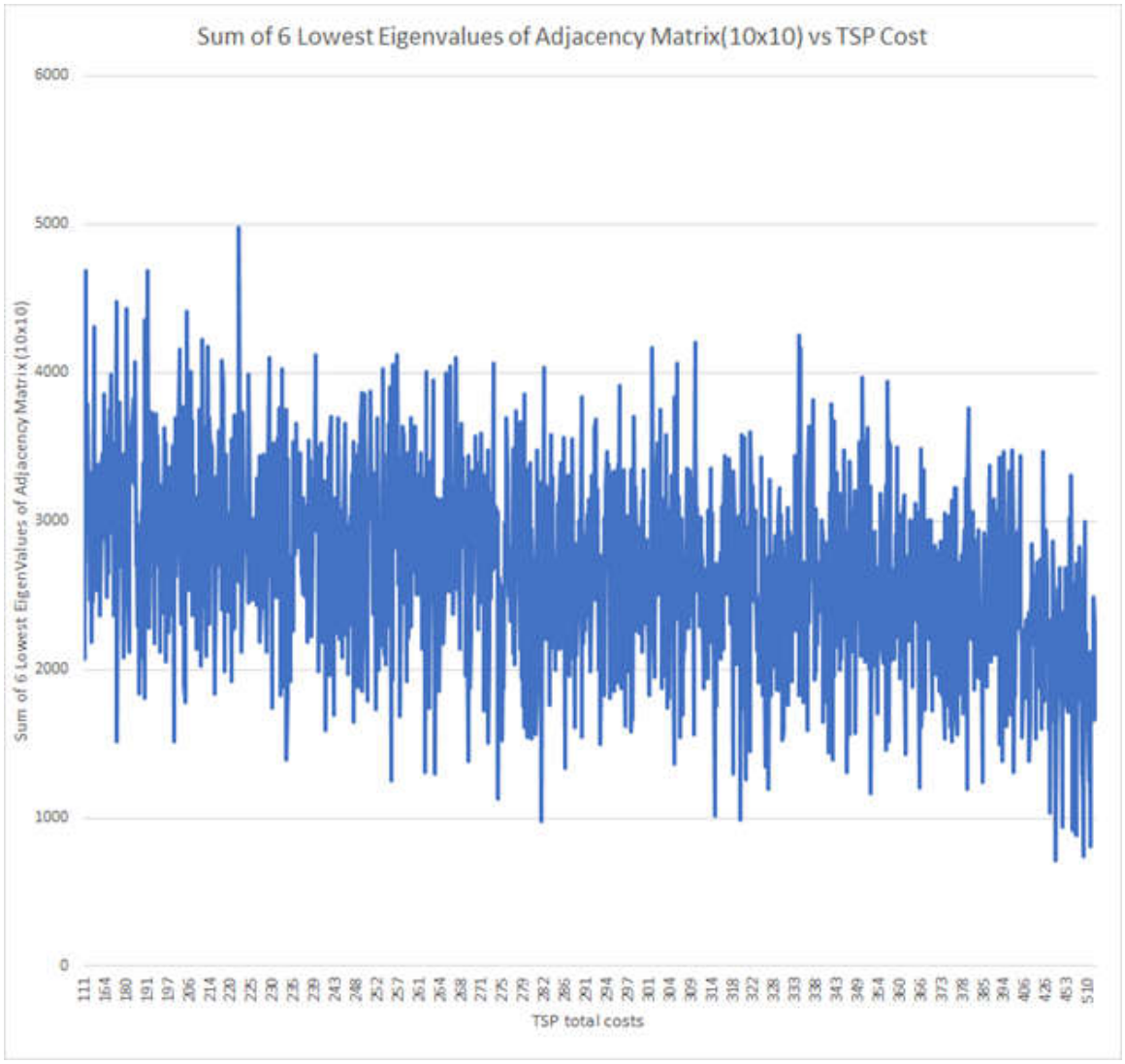

Figure 2.3. TSP costs on large adjacency matrices

Table 2.1. Times of different dynamic programs

‘ N	Naive	Dynamic Programming	Branch and Bound Dynamic Programming
2	3.7534e-06	3.5177e-05	3.3453e-05
3	1.0361e-05	5.5103e-05	5.5054e-05
4	3.8005e-05	1.0624e-04	1.1638e-04
5	1.9681e-04	2.6569e-04	2.5511e-04
6	1.2557e-03	8.4294e-04	7.7096e-04
7	9.4897e-03	1.9423e-03	1.8993e-03
8	9.9703e-02	5.1774e-03	5.3715e-03
9	8.7732e-01	1.3077e-02	1.2818e-02
10	9.4132e+00	3.9286e-02	3.3693e-02
11	1.0340e+02	9.3425e-02	1.2762e-01
12	1.3519e+03	2.0254e-01	1.9764e-01
13	1.7062e+04	5.4397e-01	4.9735e-01

test the difference between the path and the output.

2.1.4.2 CNN. Using the unweighted adjacency matrix as input n^2 we convolute using $2 * n^2$ using 156 Conv2D neurons, maxpool of (2,2), 3 times and 2 Dense layers of n and 1 corresponding neurons with ReLu and sigmoid activation functions. The optimization is Adam with MSE loss function which compares the output to the path cost.

2.1.5 Context CNN. This network required a different input preparation. We had the adjacency matrix repeated n times which correspond to a row and column. Then, link the row to the corresponding real value. Therefore the network would be trained by the matrix, a row and column to get the error of the single path value. This network is the convolutional network described above along with a parallel input of each row and column as vectors. Therefore we have $n^2 + 2n$. This idea was to give context about the graph along with the individual row and column. Convolute using $2 * n^2$

using 156 Conv2D neurons , maxpool of (2,2), 3 times and parallel dense 2n neurons. Concatenate the output of the maxpool with the output of the dense network and 2 Dense layers of n and 1 corresponding neurons with ReLu and sigmoid activation functions. The optimization is Adam with MSE loss function which compares the output to the path cost.

Table 2.2. Accuracies using 10 nodes of training data

‘ Network	Un-directed weights	Directed weights	Eigenvalue matrix
MLP	35%	23%	30%
CNN	31%	21%	20%
Context CNN	30%	17%	20%

2.1.6 Pointer Network for TSP Approximation. Pointer networks(Vinyals, Fortunato, & Jaitly,2015) are a variation of the sequence-to-sequence model with attention. Instead of translating one sequence into another, they have variable size points as inputs and yield a succession of attention encoding as a pointer to select an input element. The encoded latent space is then sorted and decoded by an LSTM arranging it into a path. This network is able to learn from the points to minimize the error on the optimum path and that way be trained to predict the optimum path. Please refer to https://arxiv.org/abs/1506.03134 to go into more detail in Pointer Networks.

2.1.6.1 Data Generation. The data used was generated by generating uniformly random variables with a range from 0 to 1 and for each two variables a point is generated. Every point's distance to each other is calculated by euclidean distance and an adjacency matrix is generated. The optimum path is calculated using the dynamic tsp program in the python tsp library. This path is used as labels.

2.1.7 Shortest Path Approximation with a Pointer Network. Using the Pointer Network we have described before, we made a change in the training. Instead

of using TSP we have generated the data to train from points to the shortest path from randomly selected points to another randomly selected point while going through all the other points. By Generating the shortest path we train the network to have input points and output the shortest path. This gave us a 87% accuracy.

2.2 Branch and Bound using Pointer Network Training the Pointer Network with 5 point optimum path cost with 88% accuracy we are able to approximate a cost of the path. We created a Dynamic programming with memoization which creates a map of cost of the getting to nodes(points) from an initial position. By generating trees of points and its costs we are able to get the costs when we arrive a the same node through a seen path. The dynamic programming also has a branch and bound, where after getting the end of a tree a path cost is set as minimum value, if a branch of the node tree is seen to have a higher cost it is pruned.

The memoization technique in the dynamic programming allow us to use the pointer network for a cost prediction. Whenever there are 5 points in a tree that have not been seen, the prediction will get the cost. This cost will be then compared with the minimum value and assessed if it needs to be iterated.

Algorithm 3 Branch and Bound using Pointer Networks

```
matrix <- adjacency matrix
best_tour <- []
current <- 0
cs <- set(range(1, length(matrix))) #Generate node ordered set
while True:
    l, lc <- rec_tsp_solve(current, cs)   #Use auxiliary function
    IF lc = 0:
        Break
    ENDIF
    best_tour.append(lc)            #add optimum node
    c <- lc
    cs <- cs - set([lc])            #remove optimum nodes from input set
ENDWHILE
best_tour <- tuple(reversed(best_tour)) #return optimum path
RETURN best_tour
FUNCTION tsp_ptr_solve(self, d):
    FUNCTION memoize(f):
        memo_dict <- {}                         #get mapping
        FUNCTION memo_func(*args):
            IF args not in memo_dict:           #if node path not in map
                memo_dict[args] <- f(*args)   #add the node to the path and cost
            ENDIF
            RETURN memo_dict[args]              #return the mapping
        ENDFUNCTION
        RETURN memo_func
    ENDFUNCTION
    GOTO memoize using function(rec_tsp_solve) #Send parameters and result
    FUNCTION rec_path_solve(c, ts):
        IF ts:
           #If the last number of cities equal the size of our trained network
            IF (len(ts) = n_cities):
                points <- np.array( points)   #list the points
                indx <- list(ts)
                cost <-  predict_cost(c, points, indx, graph) #predict the cost
                first <- list(ts)[0]
           #Check if the total cost is enough to grant a traversal
                IF ( min_cost > cost + d[first][c]): #check if the c
                   min_cost <- cost + d[first][c]
                ENDIF
            ENDIF
        vals <- []
            for lc in ts:
           #The bound checks if it is worth looking into the branch
                IF (current cost+ rec_path_solve(lc, ts - set([lc <  min_cost):
                   aval <- d[lc][c] + rec_path_solve(lc, ts - set([lc]))[0]
                  #If it is worth, append to possible
                   vals.append((aval, lc)) branch
                ENDIF
            ENDFOR
            IF (vals = []):       #If there are no worth values set a huge cost
                RETURN (999, lc)
            ENDIF
            val <- min(vals)      #Get the minimum value of the worthy costs
            RETURN val            #This value will be set for the meoization
        ELSE:
            #If there are no more nodes return the cost of current to 0
            RETURN (d[0][c], 0)
        ENDIF
    ENDFUNCTION
ENDFUNCTION
```

In the worst case scenario for a branch and bound, the branch is never pruned, therefore it has the same complexity of $O((n-z)^2 * 2^{n-z})$. However, we have to add the complexity of the Usage of the Pointer Networks $p = O(zy(t = 0\ to\ 128))$ and the new path cost calculation. $O(z)$. Therefore, the complexity

is $O(pz(n-z)^2 * 2pz^{n-z})$.

Algorithm 4 Path Cost

```
1: FUNCTION path_cost(self, path, graph):
2:     cost <- 0
3:     for i in range(len(path) - 1):
4:         cost += graph[path[i]][path[i + 1]]
5:     ENDFOR
6:     RETURN cost
7: ENDFUNCTION
```

Since the path is z numbers and the accumulation of the costs is linear it is $O(z)$.

Algorithm 5 Usage of Pointer Networks

```
1: FUNCTION predict_cost(self, current, points, index, graph):
2:     inputs_batch <- points[index]                #set the points as inputs
3:     inputs_batch <- [inputs_batch] * 256         #arrange for prediction set
4:     inputs_batch <- np.array(inputs_batch)
5:     predicted_outputs <- np.array(sess.run(decoder_outputs,  #use Ptr-Net
   feed_dict={enc_inputs: inputs_batch})).T[0]
6:     pred_outs <- []
7:     for x in predicted_outputs:          #Get prediction using original points
8:         pred_outs.append(index[x])
9:     ENDFOR
10:    cost <- path_cost(pred_outs, graph) + current_cost    #return total cost
11:    RETURN cost
12: ENDFUNCTION
```

Given that after trained the only prediction is done by the LSTM decoder, we use the LSTM as the complexity. The complexity of an LSTM according to (Sak, Haşim, Senior, Andrew, Beaufays, Françoise, 2014) is :

$$i_t = \sigma(W_{ix}x_t + W_{im}m_{t-1} + W_{ic}c_{t-1} + b_i)$$
$$f_t = \sigma(W_{fx}x_t + W_{mf}m_{t-1} + W_{cf}\,c_{t-1} + b_f)$$
$$c_t = f_t \odot c_{t-1} + i_t \odot g(W_{cx}x_t + W_{cm}m_{t-1} + b_c)$$
$$o_t = \sigma(W_{ox}x_t + W_{om}m_{t-1} + W_{oc}c_t + b_o)$$
$$m_t = o_t \odot h(c_t)$$
$$y_t = W_{ym}m_t + b_y$$

Given with 128 hidden neurons each and the input is of z LSTM cells, therefore it is $O(zy(t = 0\ to\ 128))$. In our case since the $z = 5$ it would $O(5y(t = 0\ to\ 128))$.

Table 2.3. Branch and Bound Dynamic program Accuracies per nodes

Nodes	Accuracy
5	98%
10	89%
15	81%
20	74%

2.3 Conclusions As we have tested the various networks on the Knapsack and Traveling Salesman Problem, it is concluded that the networks are able to identify certain aspects of the adjacency matrices, the eigenvalue matrices and that more data or dividing data to give context does not correspond to better accuracy. We have to also note that the difficulty in most neural networks is due to the randomization factor in the data, since an adjacency matrix can vary, each element except for the diagonal in it is needed to get an optimum path. This leads to the need of comparison between each element in its row and column over multiple elements with a step in its neighbors and do more comparisons. The sequence to sequence networks like the pointer network helps in this task which can be seen in the accuracy we got.

2.4 Future Work Finally we believe there is more to do, something that has yet to optimize the parallel CNN layers and to check the best layer to append both the CNN and row perceptron layers and to add the eigen vectors as a row in the parallel CNN.

BIBLIOGRAPHY

Hellstrom, B. J., & Kanal, L. N. (1992). Knapsack packing networks. *IEEE transactions on neural networks*, *3*(2), 302–307.

Vinyals, O., Fortunato, M., & Jaitly, N. (2015). *Pointer networks.*