PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013)
100-ft Ring Road, Bengaluru – 560 085, Karnataka, India

Report on

'Constraint Aware Deep Learning for Resource Allocation in D2D Communication'

Submitted by

K Venkat Ramnan (PES1201801319)

Aug - Dec 2021

Carried under the guidance of

Prof Neelesh B Mehta,

Next Generation Wireless Labs, Department of ECE,

Indian Institute of Science(IISc)

ABSTRACT

Device to Device (D2D) Communication in cellular networks allows direct communication between users without the traffic going through any Base station or core network infrastructure. With the main advantage of providing ultra low latency communication, D2D is a promising technology. In D2D, resource allocation is one of the crucial areas of research, since multiple users share the subchannels which introduces the factor of interference. Thus many algorithms have been developed in pursuit of providing efficient management and assignment of D2D pairs to subchannels. The algorithms which are mainly designed to solve optimization problems give optimal values, but the trade off of such algorithms is the time complexity involved. In order to overcome this issue, a deep learning based solution is proposed. Lagrangian based neural networks are developed to solve the optimization problem in a data driven manner. Other solutions including Recurrent Neural Networks are also experimented with. The specific problem of resource allocation, that can be designed as a knapsack problem, forms the dataset upon which the experiments are carried out.

ACKNOWLEDGMENT

I thank **Prof Neelesh B Mehta, Next Generation wireless labs, Dept. of ECE, Indian Institute of Science** for his continuous guidance, teaching, and encouragement throughout the internship period and the project. I thank **Bala Venkata Ramulu Gorantla, PhD student,Next Generation wireless labs, Dept. of ECE, Indian Institute of Science for** continuous guidance, teaching, and encouragement throughout the internship period and the project.

I thank **Dr. Anuradha M., Chairperson, Dept. of ECE, PES University** for all the knowledge and support received from the department.

I would like to thank my parents for their continued support and encouragement throughout the internship period.

TABLE OF CONTENTS

1. Introduction

Device to Device (D2D) refers to the direct communication between two devices (or users) without the need of data traffic passing through Base Station(BS) or Evolved NodeB (eNodeB). Devices in close proximity can directly communicate with each other. In traditional cellular networks, all traffic must go through the BS even if the devices are very close to each other. In today's world where 5G is evolving to promise ultra low latency and high throughput communication, D2D communications are capable of greatly increasing the spectral efficiency of the network. This direct communication promises energy efficiency, improvement in throughput and delay. Potential application scenarios of D2D include, among others, proximity-based services where devices detect their proximity and subsequently trigger different services (such as social applications triggered by user proximity, advertisements, local exchange of information, smart communication between vehicles, etc.). Other applications include public safety support, where devices provide at least local connectivity even in case of damage to the radio infrastructure [1]. Most resource allocation problems can be formulated as an optimization problem and mathematical programming based algorithms to solve the problem to a certain extent of optimality. In the case of D2D communication, the subchannel allocation problem can be designed to maximize the sum rate of D2D transmitters while ensuring the minimum rate of Quality of service (QOS) for Cellular users(CU). The optimization problem now resembles the 0-1 Constrained Knapsack Problem. The knapsack problem is NP Hard in nature and cannot be solved in polynomial time complexity. To solve such problems, specially designed algorithms like Dynamic Programming and Branch and Bound (B&B) algorithms are put to use. Many algorithms have evolved for efficient resource allocation. However in the real world scenario, this complexity poses a problem in terms of latency as it takes time to solve the knapsack problem. A more flexible framework for wireless resource allocation is thus required, driving a shift from the conventional solutions.

Recently, there is a paradigm shift in problem solving due to the introduction and success of deep learning in many interdisciplinary fields. With the availability of huge amounts of data and systems with high computational power, machine learning and deep learning have taken off. Machine Learning solutions have been proposed for many wireless applications ranging from PHY layer

design, resource allocation, networking, and edge computing. Neural networks are able to learn efficient representations of data and understand the insights hidden inside. It enables a powerful data-driven approach to many problems that are traditionally deemed hard due to, e.g., lack of accurate models or prohibitively high computational complexity.

Many of the current deep learning neural networks architectures are able to fit data in order to predict the target given the features. In common settings, neural networks are trained end to end from the data without knowledge of the task. Many problem statements, especially optimization ones, have constraints that need to be satisfied. Incorporation of these constraints during the training of the model will regularize the output space resulting in predictions that satisfy the constraints.

In this project, vanilla neural networks are first experimented with. It is followed by usage of Recurrent neural networks(RNN) with Gated Recurrent Units(GRU) and Long short Term memory(LSTM) units. Then the focus is targeted towards building neural networks that are constraint aware. The concept of Lagrangian duality is introduced in the neural network in order to incorporate the constraints during training.

2. System model

For the system model[2], N subchannels, N Cellular units(CU) and M D2D pairs in a cell are considered. Let $D = \{1,2,...,M\}$ be the set of D2D pairs and $S = \{1,2,...,N\}$ be the set of orthogonal uplink channels.Let CU i be allocated to subchannel i. The D2D pairs share the uplink subchannels with the CUs.

The transmit power of the CU is P_c and that of the DTx of a D2D pair is P_d. The channel power gain from CU i to the DRx of D2D pair j on subchannel i is $g_{ji}(i)$. The uplink channel power gain from CU i to the BS on subchannel i is $h_{bi}(i)$. The channel power gain between the DTx and DRx of D2D pair j on subchannel i is $h_{jj}(i)$, and from the DTx of D2D pair j to the BS is $g_{bj}(i)$. The channel power gain from the DTx of D2D pair k to the DRx of D2D pair j on subchannel i is $g^d_{jk}(i)$.

2.1 Interference Model

In the system model there are two interferences: Inter D2D Interference and Inter Cell Interference.

Inter D2D Interference: The interference power is defined by $I_{jk}(i)$ from the transmitter of D2D pair k to the D2D pair j. Thus $I_{jk}(i) = P_d g^d_{jk}(i)$. Only the statistics of $I_{jk}(i)$ is known since high coordination is required to obtain the values of the channel gain.

Inter Cell Interference : Let $I_{BS}(i)$ be the interference power from neighbouring cell users to the BS.

Let $I_j(i)$ be the interference power to the DRx of D2D pair j. Again, only the statistics of both these powers are known.

Let x_{ij} be the assignment variable, it is 1 if the subchannel i has been assigned to D2D pair j, else it is 0. Thus the Signal to interference noise ratio can be formulated as:

$$SINR_j^d(i) = \frac{P_d h_{jj}(i)}{P_c g_{ji}(i) + I_j^d(i) + \sigma^2}$$

Here $I^d_j(i)$ is the sum of both the interferences. σ^2 is the noise power.

Number of D2D assignments: Although assigning multiple D2D pairs to a subchannel can improve spatial reuse,it will lead to more interference. Thus, at most K D2D pairs are allowed to share a subchannel, where K is a system parameter. Thus we have:

$$\sum_{j=1}^{M} x_{ij} \leq K$$

2.2 QOS requirements

Let $C_{ij} = log_2(1 + \Psi_{\delta ij})$ be defined as the rate of D2D pair j on subchannel i.

We know that

$$SINR_j^c(i) = \frac{P_c h_{bi}(i)}{\sum P_d g_{bi}(i) + I_{BS}(i) + \sigma^2}$$

We require that it must be able to transmit at a minimum rate $R^{(i)}_{min}$ with an outage probability that is at most ϵ_c, which is a system parameter.

Thus,

$$P_r(log_2(1 + SINR^c_i(i)) \geq R^{(i)}_{min}) \geq 1 - \epsilon_c.$$

Substituting $SINR^c_i(i)$ and rearranging terms, we get

$$\sum_{j=1}^{M} x_{ij} w_{ij} \leq b_i$$

$w_{ij} = P_d g_{bj}(i)$ is the interference power at BS on subchannel i due to the DTx of D2D pair j.

$bi = P_c h_{bi}(i)/2\, R^{(i)}_{min} - 1 - \sigma^2 - F^{-1}_{BS}(1 - c)$, and $F^{-1}_{BS}(\cdot)$ is the inverse of the CDF of $I_{BS}(i)$.

2.3 Subchannel Resource allocation problem statement

The final problem can be formulated as maximizing the sum of D2D rates as below:

$$\text{P}: \max_{x_{ij}, \forall i \epsilon S, j \epsilon D} \left\{ \sum_{i=1}^{N} \sum_{j=1}^{M} x_{ij} C_{ij} \right\}$$

$$\text{subject to} \quad \sum_{i=1}^{N} x_{ij} \leq 1, \ \forall j \in D,$$

$$\sum_{j=1}^{M} x_{ij} w_{ij} \leq b_j, \ \forall i \in S,$$

$$\sum_{j=1}^{M} x_{ij} \leq K, \ \forall i \in S,$$

$$x_{ij} \in \{0, 1\}, \ \forall i \in S, \forall j \in D$$

The given problem is similar in terms of the 0-1 Knapsack problem which is defined as:

$$max \sum_{i=1}^{N} v_i x_i$$

$$\text{subject to : } \sum_{i=1}^{M} w_i x_i \leq C$$

$$x_{ij} \in \{0, 1\}$$

where v is the values

w is the weight

C is the capacity

Thus the given problem in *NP-Hard* in nature.

3. Deep Learning

3.1 Neural Network for Binary Classification: Understanding the mathematics

Artificial Neural networks are computational models that mimic biological neurons. They are composed of a large number of connected nodes which perform simple mathematical operations. Each node's output is determined by this operation, as well as a set of parameters that are specific to that node. By connecting these nodes together and carefully setting their parameters, very complex functions can be learned and calculated.

3.1.1 Rosenblatt's Perceptron explained

The perceptron is the simplest neural network with one input layer and one output layer. A perceptron is used to implement linearly separable functions by taking in real/boolean inputs and associating them with weights and biases to obtain the function. A perceptron has a single neuron as shown in figure 1.

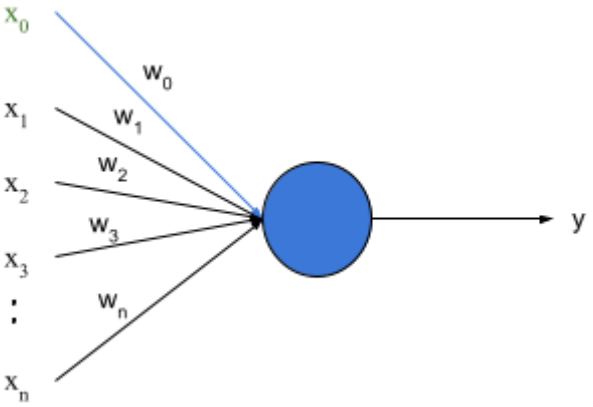

Fig 1. Single Perceptron

The equation of the perceptron is defined as

$$z = 1,\ if\ W.X + b > 0\ or \sum_{i=1}^{N} w_i x_i + b > 0$$

$$0,\ otherwise$$

Here W is the weight matrix, X is the input matrix and b is the bias (w_0), usually set to 1. The function z is passed to a non linear activation function given by

$$\widehat{y} = g(z)$$

3.1.2 Multiple layers

Given multiple layers with multiple neurons, the term x will become the matrix a, corresponding to the activation of each layer. The x vector is thus the activation for layer 0, i.e, the input layer. Thus for a given layer l, each neuron performs the following:

$$z_j^{[l]} = w_j^{[l]}.a^{[l-1]} + b_j$$

$$a_j^{[l]} = g^{[l]}(z_j^{[l]})$$

Thus we stack all the weights w and the biases transposed horizontally to build the matrix W and b. All vertical vectors x,a and z are combined to get X, A and Z. Thus

$$Z^{[l]} = W^{[l]}.A^{[l]} + b^{[l]}$$

$$A^{[l]} = g^{[l]}(Z^{[l]})$$

3.1.3 Activation functions

As mentioned before, the role of the activation function is to introduce non linearity in the neural network thus allowing it to be more flexible in learning complex functions. Usually a differentiable activation function is always preferred since neural networks are optimized using gradient descent. Similarly they are required to be computationally cheap and play an important role on the speed of the learning process. An activation function in a neural network is usually defined by

$$\widehat{y} = \phi(W.A)$$

Therefore, a neuron really computes two functions within the node, the summation and activation function. The most important activation functions are:

- Identity or linear function

$$\phi(v) = v$$

- Sigmoid function

$$\phi(v) = \frac{1}{1+e^{-v}}$$

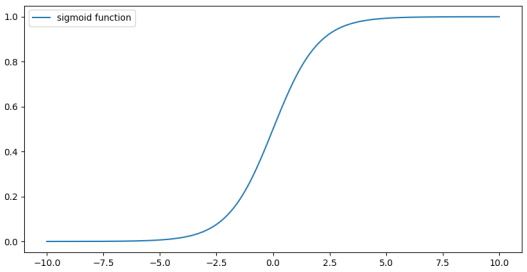

Fig 2. Sigmoid Function

- tanh function

$$\phi(v) = \frac{e^{2v}-1}{e^{2v}+1}$$

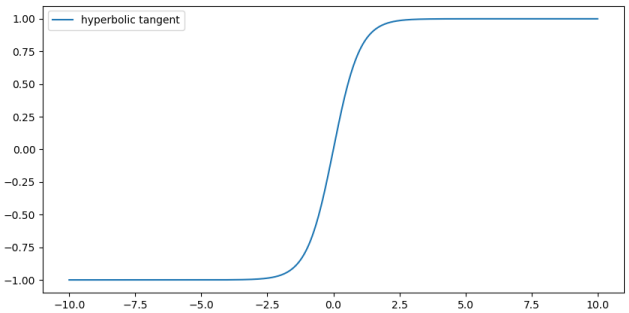

Fig 3. Tanh function

- ReLU function

$$\phi(v) = max\{v, 0\}$$

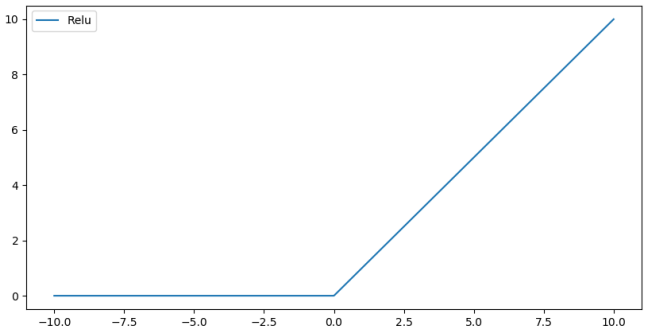

Fig 4. ReLU function

The ReLU and tanh activation functions are commonly used for hidden layers because of the ease in training multilayered neural networks with these activation functions. The sigmoid function is used as the activation function of the output layer when dealing with binary classification. Thus the activation function design of a 3 input 3 output binary classification neural network for binary classification is as below

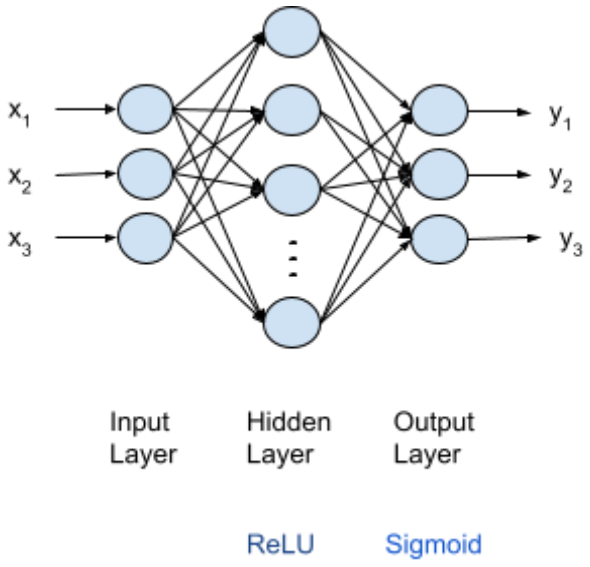

Fig 5. Simple NN classification model

3.1.5 Loss function for binary classification

In the context of an optimization algorithm, the function used to evaluate a candidate solution is

referred to as the objective function.Thus we maximize or minimize the objective function, i.e we are searching for a candidate solution that has the highest or lowest score respectively. Typically, with neural networks, we seek to minimize the error. As such, the objective function is often referred to as a cost function or a loss function and the value calculated by the loss function is referred to as simply loss. The most commonly used loss functions are cross entropy, mean squared error, mean absolute error, and Hinge loss.

For classification, the cross entropy function is commonly used as the loss function.This function comes from information theory where the goal is to measure the difference between two averages of the number of bits of distribution of information.

Each predicted probability is compared to the actual class output value (0 or 1) and a score is calculated that penalizes the probability based on the distance from the expected value. The penalty is logarithmic, offering a small score for small differences (0.1 or 0.2) and enormous score for a large difference (0.9 or 1.0). Cross-entropy loss is minimized, where smaller values represent a better model than larger values. A model that predicts perfect probabilities has a cross entropy or log loss of 0.0. Cross-entropy for a binary or two class prediction problem is actually calculated as the average cross entropy across all examples.

The cross entropy function is given as

$$L = -\frac{1}{m}\sum_{i=1}^{m} y_i . log(\widehat{y}_i)$$

The binary cross entropy function for a given data set is given by

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\widehat{y}_i, y_i)$$

$$L(\widehat{y}_i, y_i) = -(ylog(\widehat{y}) + (1-y)log(1-\widehat{y}))$$

3.1.6 Optimization and Learning in Neural network : Gradient Descent and Backpropagation

A deep learning neural network learns to map a set of inputs to a set of outputs from training data.The problem of learning is set as an optimization problem and an algorithm is used to navigate the space of possible sets of weights the model may use in order to make good or good enough predictions.Typically, a neural network model is trained using the stochastic gradient descent

optimization algorithm and weights are updated using the backpropagation of the error algorithm. Here the gradient pertains to the error gradient. The error function for N data points is denoted as

$$E = \frac{1}{N}\sum_{k=1}^{N}\sum_{i}(y_i^k - f(\sum_j w_{ji}x_j^k))^2$$

$$\text{Let } \hat{y} = f(\sum_j w_{ji}x_j^k)$$

Here

y = true labels

k = all data points

i = output units of the network

j= number of features

x= feature

w = weights

f is a function of the predicted values

Thus the objective here would be to minimize the above function. To find the minimum of a function, its derivative is taken. When we take a single neuron with a single output unit,i.e i = 1, the derivative with respect to weight is given as

$$\frac{\partial E}{\partial w_j} = (y - \hat{y})\frac{\partial \hat{y}}{\partial w_j}$$

$$\hat{y} = f(\sum_j x_j w_j) = f(h)$$

$$\frac{\partial E}{\partial w_j} = (y - \hat{y})f'(h)\frac{\partial}{\partial w_j}\sum_j x_j w_j$$

$$\frac{\partial E}{\partial w_j} = (y - \hat{y})f'(h)x_j$$

When we multiply using a learning rate η we end up with the gradient of the squared error with respect to weight as

$$\Delta w_j = \eta\,(y - \hat{y})f'(h)x_j$$

Let δ be the error term defined as

$$\delta = (y - \hat{y})f'(h)$$

Thus the general equation is now

$$\Delta w_{ji} = \eta\,\delta x_{ji}$$

Thus for each epoch the update function would now be

$$w_{ji}^{t} = w_{ji}^{t-1} + \Delta w_{ji}$$

This whole process is repeated after each epoch during training.

The forward propagation leads to the production of predicted values. As explained in the previous section, the weights must be updated using gradient descent in order to minimize the loss function,

thus essentially learning during the process. Thus forward propagation is filled by backpropagation.The error propagates through backpropagation. Thus backpropagation aims to minimize the loss by adjusting the weights by performing gradient descent based optimization. Consider an error δ attributed to an output unit *k* and a hidden input *i* of a hidden layer *h*.

$$\delta_i^h = \sum W_{ik} \delta_k^0 f'(h_j)$$

Thus the gradient descent step is

$$\Delta w_{ji} = \eta \delta_j^h x_{ji}$$

With the weights updated, the steps are repeated until the prediction becomes as close as to the ground truth as given in the figure below.

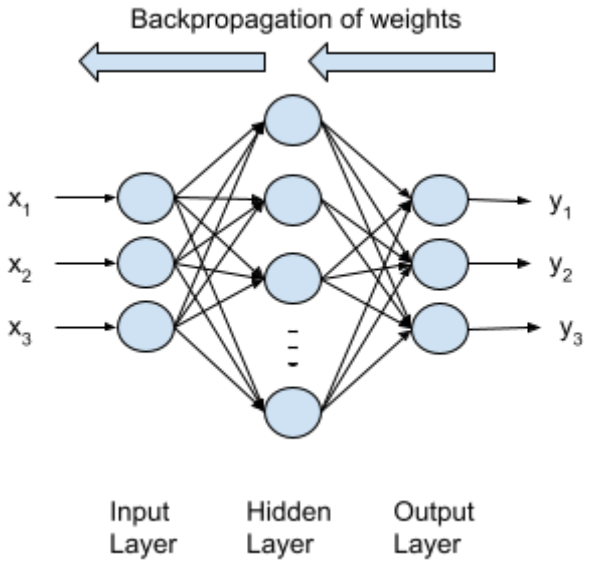

Fig 6. Backpropagation of weights

\3.1.7 Optimizers

As explained in the previous section, optimization forms an essential part of the neural network. Thus selecting the right optimizers matters. The most commonly used ones include Stochastic Gradient Descent(SGD), Root mean square(RMS) Propagation and Adaptive moment estimation(ADAM).

The basic vanilla gradient descent for update in parameter θ in given by

$$\theta = \theta - \eta \nabla_{\theta} J(\theta)$$

Stochastic gradient descent(SGD) performs parameter update for every training sample x^i and y^i

$$\theta = \theta - \eta \nabla_{\theta} J(\theta; x^i; y^i)$$

SGD is relatively faster compared to ordinary gradient descent. But it has problems with being stuck in local minima.ADAM and RMS Prop are on average better optimizers compared to SGD.This is due to their ability to reach the global minima faster and in a more reliable manner.

3.1.8 Experiments on the Problem statement

The Vanilla RNN is used. The dataset can be explained as

Features :

1.Rates (V1 ….. V30) : 30 values

2.Weights (w1 ….. w30) : 30 Values

3.C : 1

Target :

X1 …. X30 : 30 Values

Number of columns :

Features : 61

Target : 30

Number of instances (rows) = 9210

Thus,

If X matrix denotes the features, the shape of matrix is (9210 x 61)

If y matrix denotes the target, the shape is (9210 x 30)

Training Data : 80% of total dataset

Validation/Testing Data: 20% of total dataset

Here we are trying to predict Xn one by one. We are not predicting all 30 values at once. Since doing so gives very bad results. In the given case, we use two models,

Model 1:

Layer (type)	Output Shape	Param #	
input_7 (InputLayer)	[(None, 61)]	0	Input
dense_30 (Dense)	(None, 128)	7936	Activation: Relu
dense_31 (Dense)	(None, 64)	8256	Activation: Relu
dense_32 (Dense)	(None, 32)	2080	Activation: Relu
dense_33 (Dense)	(None, 16)	528	Activation: Relu
dense_34 (Dense)	(None, 1)	17	Output : Sigmoid Layer

Optimizer Used : SGD

Learning Rate : 0.0001

Loss : Binary Crossentropy

Result Metric : Accuracy

Epochs Trained : 1000

Batch Size : 32

Model 2:

Layer (type)	Output Shape	Param #	
input_7 (InputLayer)	[(None, 61)]	0	Input
dense_5 (Dense)	(None, 128)	7936	Activation: Relu

dropout_4 (Dropout)	(None, 128)	0	
dense_6 (Dense)	(None, 64)	8256	Activation: Relu
dropout_5 (Dropout)	(None, 64)	0	
dense_7 (Dense)	(None, 32)	2080	Activation: Relu
dropout_6 (Dropout)	(None, 32)	0	
dense_8 (Dense)	(None, 16)	528	Activation: Relu
dropout_7 (Dropout)	(None, 16)	0	
dense_9 (Dense)	(None, 1)	17	Output : Sigmoid Layer

Optimizer Used : SGD

Learning Rate : 0.0001

Loss : Binary Crossentropy

Result Metric : Accuracy

Epochs Trained : 1000

Batch Size : 32

The results of the model 1 :

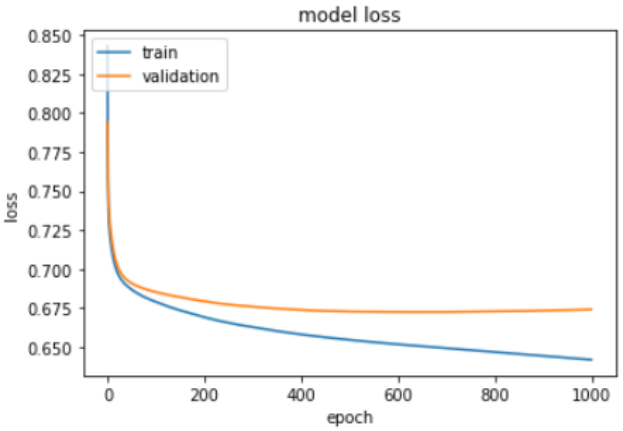

Fig 7. Model 1 Loss

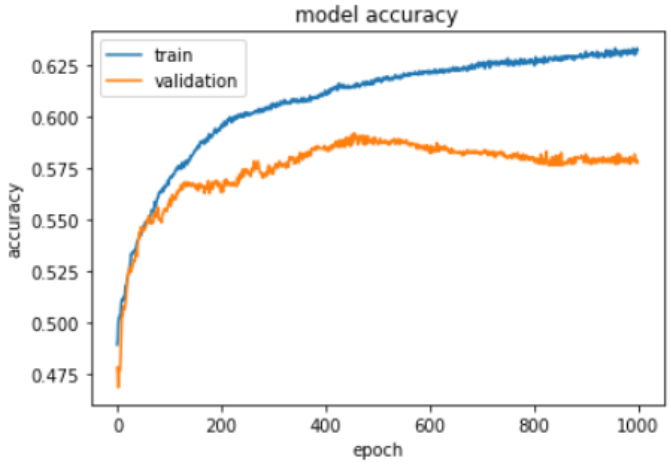

Fig 8. Model 1 Accuracy

The results of model 2 are as below:

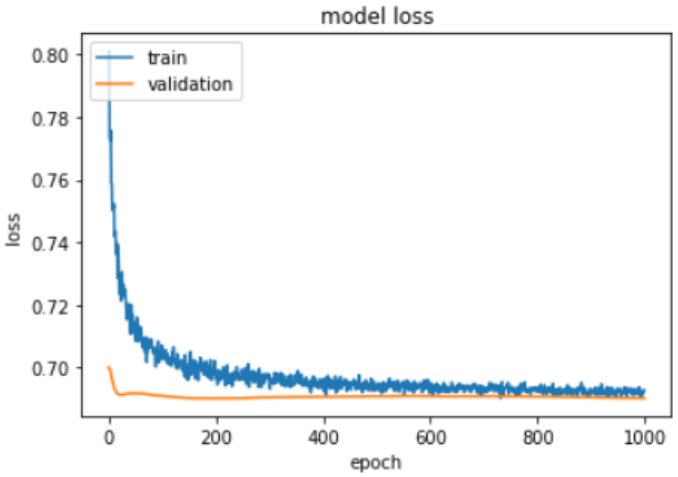

Fig 9. Model 2 loss

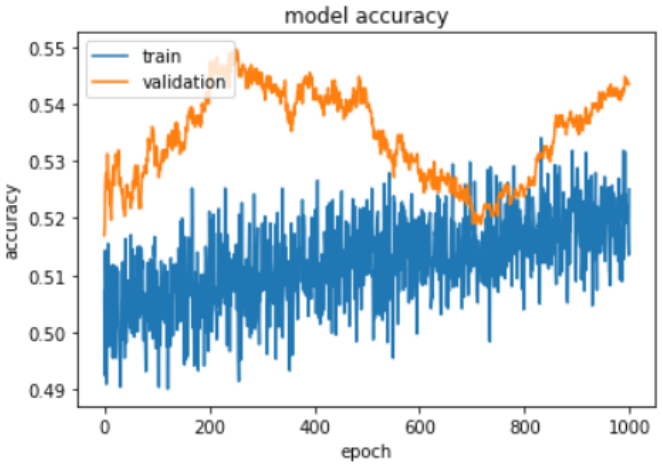

Fig 10. Model 2 accuracy

Thus the above models gave a validation accuracy of 55 to 60 percent. The drawback of this model is the bad accuracy obtained when multilabel classification is performed to learn all 30 parameters together.

3.2 Recurrent Neural Network (RNN)

3.2.1 Why RNNs

Compared to vanilla neural networks the RNNs have a special ability of gaining insights from sequence based data. In RNNs, the output from the previous step is fed as the input to the next step. Ordinary feed forward neural networks are only meant for data points, which are independent of each other. However, if we have data in a sequence such that one data point depends upon the previous data point, we need to modify the neural network to incorporate the dependencies between these data points. RNNs have the concept of 'memory' that helps them store the states or information of previous inputs to generate the next output of the sequence.

Although in the case of our problem statement, the features of values and weights are not related directly to each other, the way they are ordered(taken in consideration) with respect to the capacity can be related. Authors of [3] have used an Encoder Decoder based RNN with attention to try to solve the knapsack problem. Similarly, RNNs have found their use in solving combinatorial problems like Travelling salesman problem(TSPs) using specially designed RNNs called pointer networks[4]. RNNs have thus served as a supervised learning model to try to learn the combinatorial problems.

3.2.2 RNN mathematics

Unlike feedforward neural networks, hidden layers of RNNs have connections back to themselves, allowing the states of the hidden layers at one instant to be used as input to the hidden layers the next time instant. This provides the memory that allows hidden states to capture information about the temporal relation between input sequences and output sequences.

RNNs are called recurrent because they perform the same computation (determined by the weights, biases, and activation functions) for every element in the input sequence. The difference between the outputs for different elements of the input sequence comes from the different hidden states, which are dependent on the current element in the input sequence and the value of the hidden states at the last time step.

The important idea of RNN is parameter sharing. RNNs share the same weights across several time steps. Each member of the output is a function of the previous member of the output. Each member of output is produced using the same update applied to previous outputs.

RNN operates on a sequence that contains vectors x^t with time index t ranging from 1 to τ. Thus RNNs operate on minibatches of such sequences.

Unlike a DNN, let us consider handling individual time steps as given below:

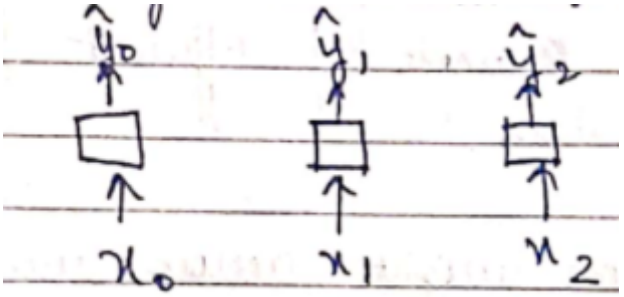

Fig 11. Individual timesteps

Thus the function would thus be:

$$\widehat{y_t} = f(x_t)$$

Suppose the $\widehat{y_2}$ depends on x_0 and x_1 in the later stages, it is necessary to introduce the factor of recurrence. Thus neurons with recurrence would be as

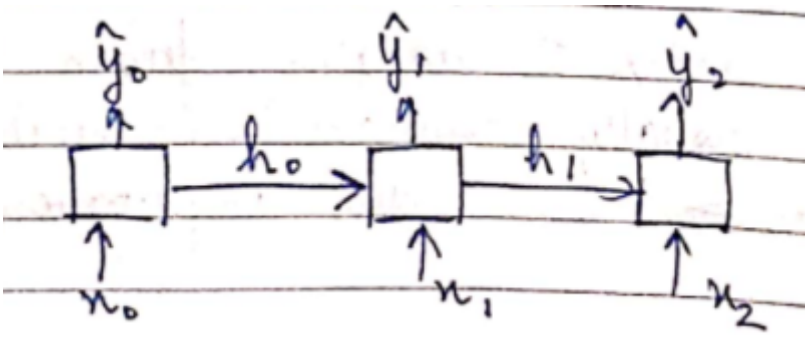

Fig 12. Recurrence between timesteps

Here h_t is the internal memory maintained which is maintained from time step to time step. Since it is a function of input as well as that of the past step memory, it now becomes

$$\widehat{y_t} = f(x_t, h_{t-1})$$

where h_t is the internal memory maintained.

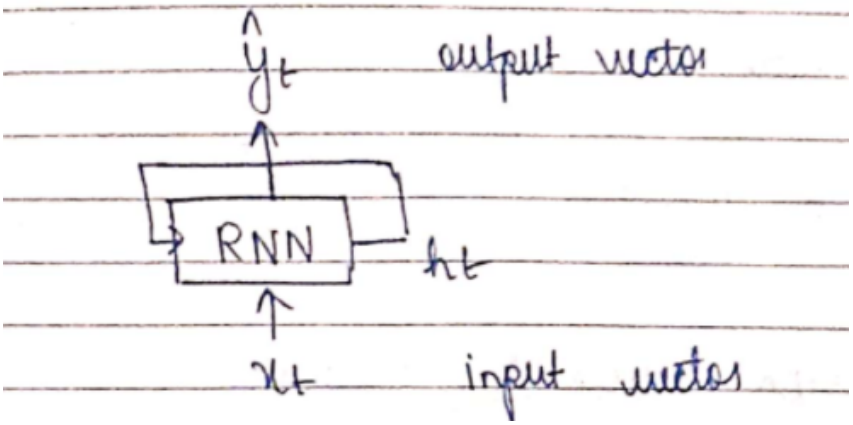

Fig 13. RNN model

RNNs have a state h_t that have is updated each time step as a sequence is processed.Applying the recurrence relation at every time step,

$$h_t = f_w(x_t, h_{t-1})$$

where h_t is the cell state and f_w is a function with weights w.

Thus for a given input vector x_t. The update hidden state is given by:

$$h_t = tanh(W_{hh}^T h_{t-1} + W_{xh}^T x_t)$$

The output vector is thus given as

$$\widehat{y}_t = W_{hy}^T h_t$$

where W_{xh} is the weight matrix and W_{hh} is the hidden state to hidden state matrix.

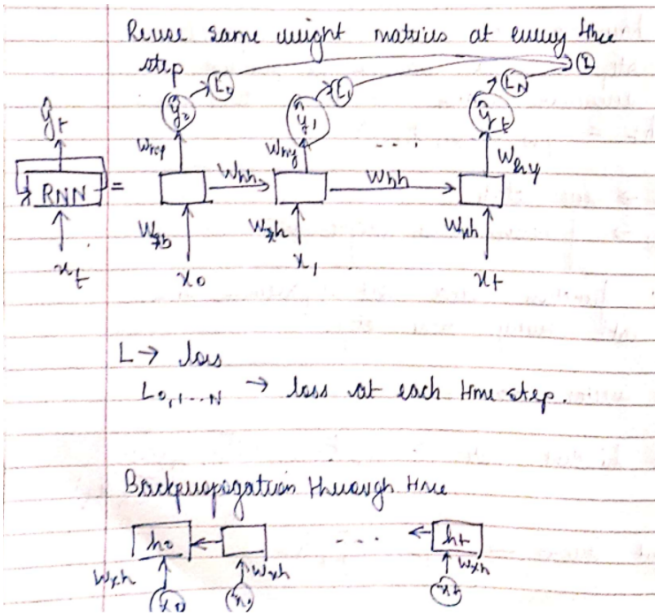

Fig 14. Inside RNN and BPTT

Backpropagation through time (BPTT) is a special type of backpropagation for RNN. Thus we compute gradients wrt h_0 which involves many factors of W_{hh} .

Stacked RNNs form the encoder decoder architecture. The encoder processes the input and produces one compact representation called z. This z value is then fed to the decoder to produce output which is predicted at the final hidden state.

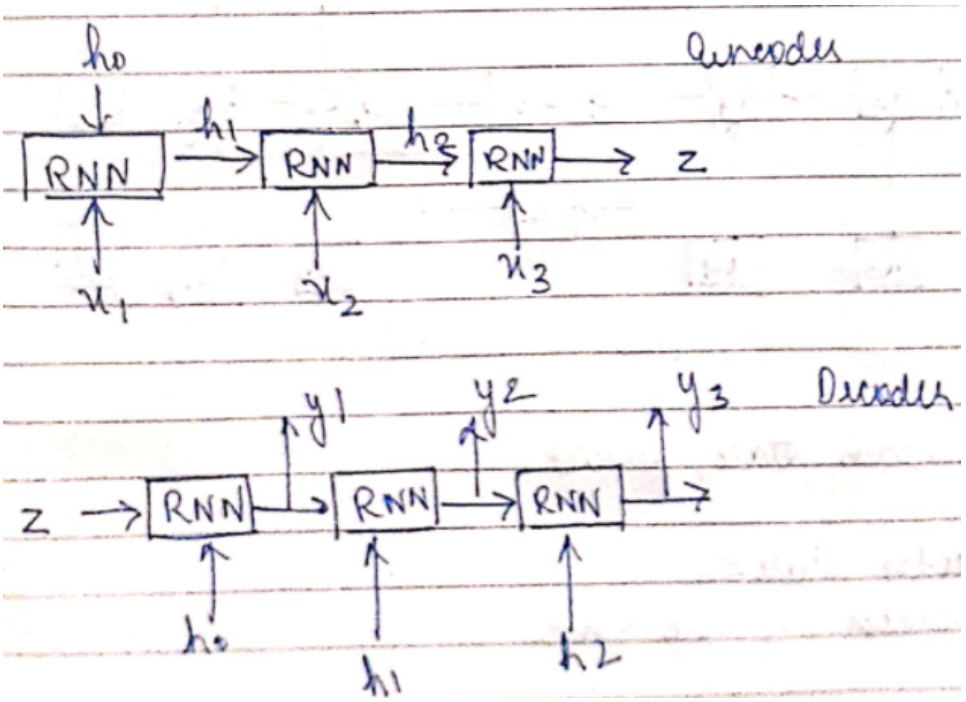

Fig 15. Encoder Decoder

3.2.3 The problem with RNNs : Solution is LSTM

During back propagation, RNN suffers from the vanishing gradient problem. The vanishing gradient is when the gradient shrinks through time. If the gradient becomes too small, it does not contribute to learning much. Thus RNNs can forget what it sees in longer sequences thus having a short term memory.

In order to overcome this, Long short term memory or LSTMs are used. LSTMs have internal mechanisms called gates that regulate the information flow. LSTMs have a similar process as that of LSTM on data passing on information as it propagates forward. The operations of an LSTM allows it to keep or forget information.

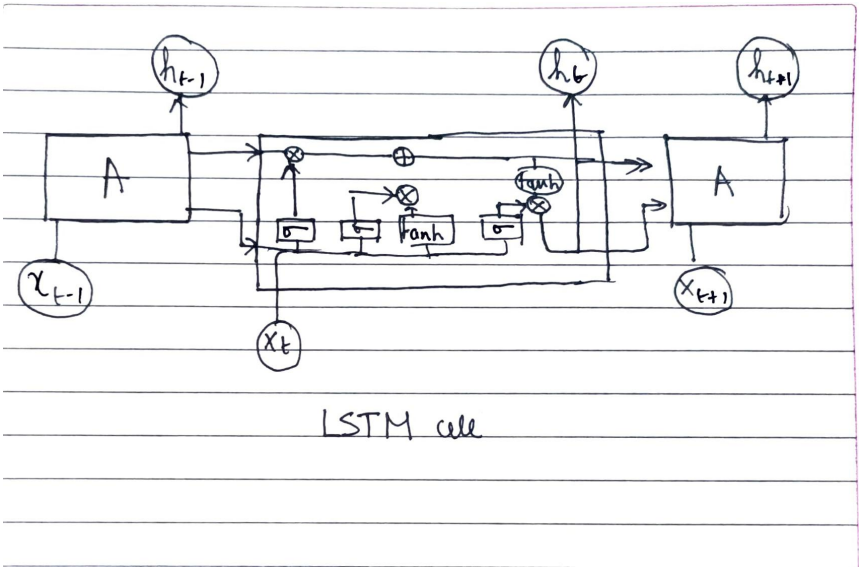

Fig 16. LSTM

As given in the above image, the LSTM cell is made up of tanh and sigmoid functions with pointwise multiplication and addition.

The first step in our LSTM is to decide what information to throw away from the cell state. This decision is made by a sigmoid layer called the "forget gate layer." It looks at h_{t-1} and x_t, and outputs a number between 0 and 1 for each number in the cell state C_{t-1}. A 1 represents "completely keep this" while a 0 represents "completely get rid of this."

The forget function can be implemented as:

$$f_t = \sigma(W_f . [h_{t-1}, x_t])$$

The first sigmoid layer performs the following:

$$i_t = \sigma(W_f . [h_{t-1}, x_t])$$

Next, a tanh layer creates a vector of new candidate values, $\tilde{C_t}$, that could be added to the state.

$$\tilde{C_t} = tanh(W_c . [h_{t-1}, x_t])$$

Thus now the update is given by

$$C_t = f_t * C_{t-1} + i_t * \tilde{C_t}$$

The output will be based on the cell state, but will be a filtered version. A sigmoid layer which decides what parts of the cell state we're going to output is first used. Then, the cell state is run through tanh (to push the values to be between −1 and 1) and multiplied by the output of the sigmoid gate.

$$o_t = \sigma(W_o [h_{t-1}, x_t])$$

$$h_t = o_t * tanh(C_t)$$

3.2.4 Experiments using LSTM based RNN

As mentioned in the previous experiments, we use the same dataset as mentioned above. The basic RNN is built with a single LSTM layer containing 200 cells followed by a time-distributed sigmoid layer. The second model built uses a bidirectional RNN which is a combination of two RNNs - one RNN moves forward, beginning from the start of the data sequence, and the other, moves backward, beginning from the end of the data sequence. The third model is an encoder decoder model which utilizes the Repeat Vector functionality of tensorflow. Unlike the previous models, all the above 3 RNN models predict all 30 values together.

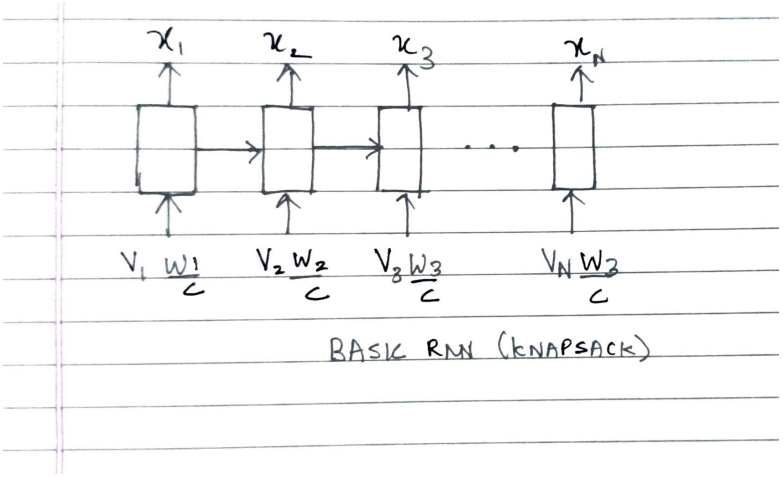

Fig 17. Basic RNN for knapsack

Model 1 (Basic RNN):

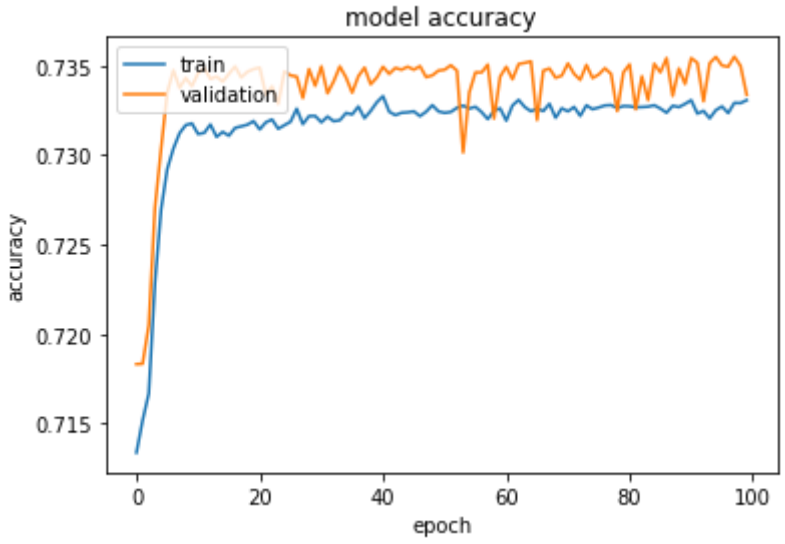

Fig 18. RNN Model 1 accuracy

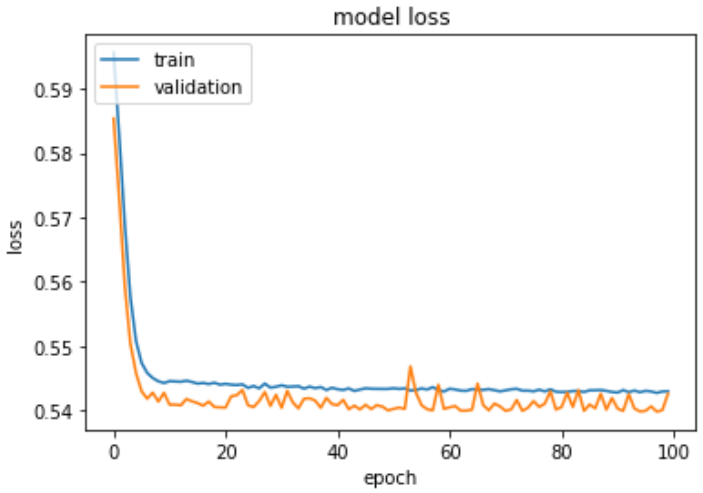

Fig 19. RNN Model 2 loss

Model 2 (Bidirectional RNN):

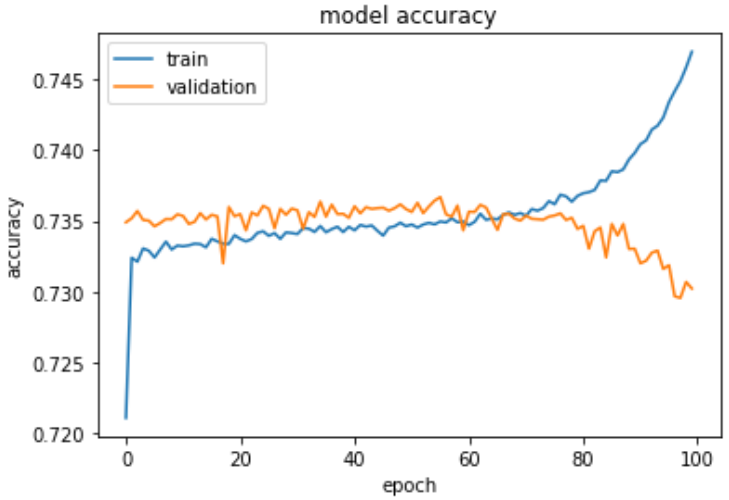

Fig 20. RNN Model 2 accuracy

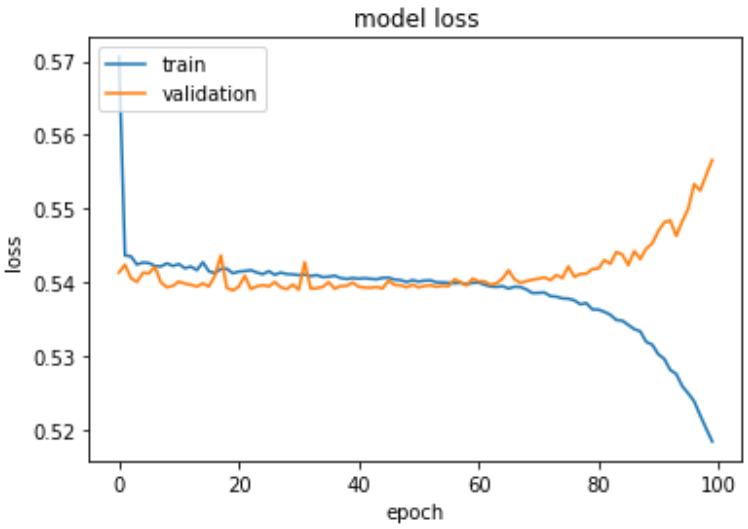

Fig 21. RNN Model 2 loss

Model 3 (Encoder Decoder Model) :

Final Training accuracy: 71.52 %

Final Validation accuracy : 71.23%

Final Training Loss : 0.5944

Final Validation Loss : 0.5999

Thus the RNN models perform better than the vanilla NNs in predicting all the 30 values together. They average around 70 percent accuracy across different models.

4. Constrained Optimization

4.1 Optimization

An optimization problems is of the form

$$minimize\ f(w_1,\ w_2,\ \ldots\ldots\ ,\ w_d)$$

where f: $R^d \rightarrow R$ is the objective function and $w_1, w_2, \ldots, w_d$ are decision variables.The goal of the optimization problem is to compute the values of the variables at which the objective function is either maximized or minimized. It is common to use a minimization form of the objective function in machine learning, and the corresponding objective function is often referred to as a loss function. The optimal solution $w^* = [w^*_1, \ldots w_d^*]^T$ is the solution which has the minimum value.

The most common optimization is the univariate optimization. To find the optimal value, we take the derivative of the optimization function in order to obtain the optimal solution.

$$f(x) = x^2 - 2x + 3$$

$$\frac{df(x)}{dx} = 2x - 2 = 0$$

Thus for the above function x =1 is the optimal value.

4.2 Constraints in optimization

Most optimization problems always carry constraints with them. These constraints put a restriction on the decision variables as to what values can be taken. In general case it can be expressed as

$$minimize\ f_o(w)$$

$$subject\ to\ f_i(w) \leq 0,\ i = 1,\ldots m \text{ and } \quad h_i(w) = 0,\ i = 1,..,m$$

where $f_i(w)$ is the inequality constraint and $h_i(w)$ is the equality constraint.A point w is feasible if it

satisfies all the constraints. Among all the feasible points, we want to find the one for which the objective function, f_0, takes the smallest value.

4.2.1 Linear Programming

An optimization problem is linear if both the inequality and equality constraints are linear.

A Linear optimization problem or Linear programming (LP) is given by:

$$minimize\ c^T w$$

$$such\ that\ a_i^T w + b_i \leq 0,\ i = 1,..,m$$

This is a classical constrained optimization problem that is solved by methods like simplex or graphical methods. An example is as below

$$min\ f(c,w) = 0.40c + 0.45w$$

$$such\ that: 0.1c + 0.15w \geq 0.65$$

$$0.75\,c + 0.7w \geq 4$$

$$c + w \leq 7$$

$$c \geq 0$$

$$w \geq 0$$

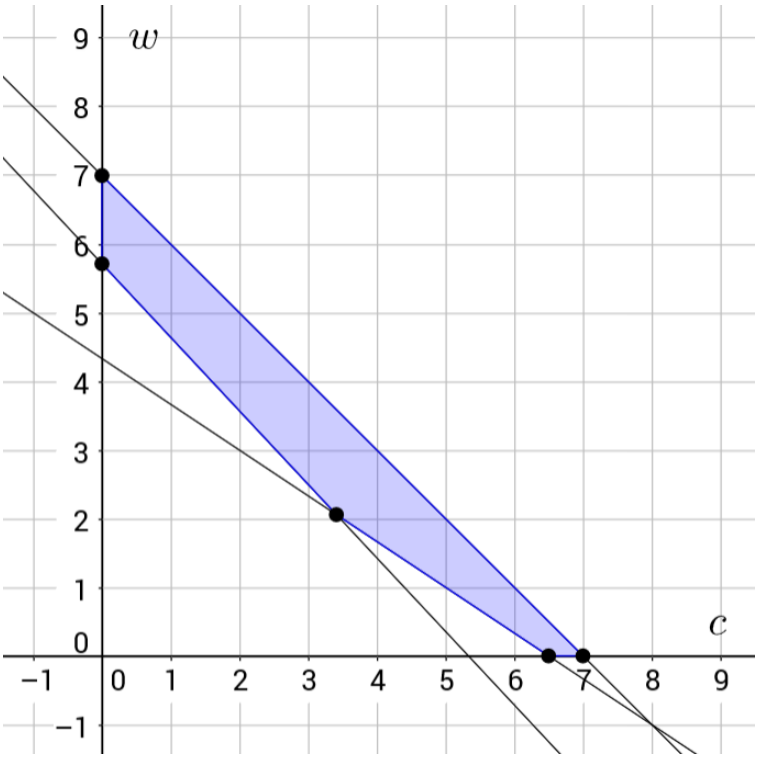

Fig 22. LP example using graphical method

Thus the system of equations to solve would be

$$0.1c + 0.15w = 0.65$$

$$0.75c + 0.7w = 4$$

Thus the optimal solution would be c = 3.4 and w = 2.

4.2.2 Penalty function

Constrained optimization problems can be converted into unconstrained one using penalty

function.The goal of penalty functions is to convert constrained problems into unconstrained problems by introducing an artificial penalty for violating the constraint. Suppose an inequality constraint is x -5 ≤ 0, design a penalty function such that:

it is 0 when x -5 ≤ 0

positive when x-5 > 0

Thus the function can be defined as

$$P(x) = max(0, x - 5)$$

A more severely penalizing function can be defined as:

$$P(x) = max(0, x - 5)^2$$

Now this penalty function must be added to the original objective function and minimize. Thus the total function must be:

$$minimize\ T(x) = f(x) + P(x)$$

Thus

$$minimize\ T(x) = \frac{100}{x} + max(0, x - 5)^2$$

The addition of penalty functions can create severe slope changes in the graph at the boundary, which interferes with typical minimization programs.

Thus penalty regularization factor *r* which controls how much the penalty function penalizes is added.

Thus

$$minimize\ T(x) = \frac{100}{x} + r.max(0, x - 5)^2$$

In case of inequality constraints, this value of r must be positive and increased until the solutions converge.

4.2.3 Duality

Optimization problems can be viewed as two forms, primal and dual forms.The primal is the original problem where we optimize parameters given constraints; in the dual, the constraints define the parameters that we optimize.

For an optimization problem

$$minimize\ f_0(x)$$

$$such\ that\ f_i(x)\ \leq\ 0\ and\ h_i(x)\ =\ 0$$

Thus we add Lagrangian as:

$$L(x, \lambda, v)\ =\ f_0(x)\ +\ \sum_{i=1}^{m} \lambda_i f_i(x)\ +\ \sum_{i=1}^{m} v_i h_i(x)$$

using the Lagrange multipliers $\lambda_i \geq 0$ and ν_i associated with the inequality and equality constraints, respectively.

If $f_i(w) < 0$, to get the maximum value, the corresponding λ_i should be 0 (λ_i cannot be negative); if $f_i(w) > 0$, that is, if the constraint is not satisfied, λ_i can be arbitrarily large. If $h_i(w) = 0$, we do not care for the corresponding v_i; if $h_i(w) \neq 0$, v_i can be of the same sign as $h_i(w)$ and arbitrarily large in magnitude. So if the constraints are satisfied, that is, if the primal is feasible, the added constraints will be 0; otherwise they can go up to infinity.

The dual problem is

$$max_{\lambda,v}\, min_x L(x, \lambda, v)$$

The optimal value of the dual, d*, is the best lower bound for the optimal value p* of the primal problem. If d* ≤ p*,weak duality; if d* = p*strong duality.

5. Constraint aware Neural networks

5.1 Why constraint aware

In the latest advancements in neural networks, a lot of focus has been placed on several problems which have constraints that exist over the output label space. These constraints can be a great way of injecting prior knowledge into a deep learning model, thereby improving overall performance. Such constraints arise in many science and engineering domains, where the task amounts to learning optimization problems which must be solved repeatedly and include hard physical and

operational constraints.

Many latest works [5,6,7] explore using penalty based methods and using the lagrangian in order to train neural networks that need to enforce certain constraints. Since our problem is an optimization problem with constraints to be solved, this can be solved using the penalty function as soft constraints applied to the loss function.

In the methods mentioned above, where we do not explicitly the constraints, the predicted results do not satisfy the constraints.

For the above models, even though the accuracy is 60% for vanilla networks and 72% for RNN,

only 20 percent of the predictions of the model in test data are satisfying the constraint. It is usually thought that since the data already has the optimal solutions as the target data, the constraints are satisfied inherently.

5.2 Soft constraint based Loss function for given problem statement

For the given problem statement, we can convert the constraint problem into unconstrained problem

as given below:

$$minimize\ mean(-\sum_i v_i x_i + \lambda max(\frac{\sum_i w_i x_i}{C} - 1, 0))$$

Thus this new function will be the loss function for a given neural network. The knapsack problem is converted into an unconstrained problem. The λ value decides how much the penalizing takes place. Now this loss function can be applied in a neural network where the weights of the neural network are optimized using gradient descent.

Apart from the above function, which is assumed as loss function L_1, two more loss functions can be produced.

$$L_2 : mean \mid \sum_i v_i x_i - \sum_i v_i \widehat{x_i} \mid + \lambda max(\frac{\sum_i w_i x_i}{C} - 1, 0)$$

$$L_3 : mean \left(\sum_i v_i x_i - \sum_i v_i \widehat{x_i}\right)^2 + \lambda max(\frac{\sum_i w_i x_i}{C} - 1, 0)$$

5.3 Experiments

5.3.1 Neural network design

The neural network is designed with one input layer, two hidden layers and one output layer. The first hidden layer contains 128 neurons and the second hidden layer has 64 neurons. The model is kept as simple as possible in order to avoid the effect of overfitting.

The hidden layers are ReLU activated and the output layer is sigmoid layer. The model is trained using the ADAM optimizer. It is trained over 1000 epochs. The learning rate is set to 0.01.The training validation split of data is taken as 70 percent for training and 30 percent for testing.

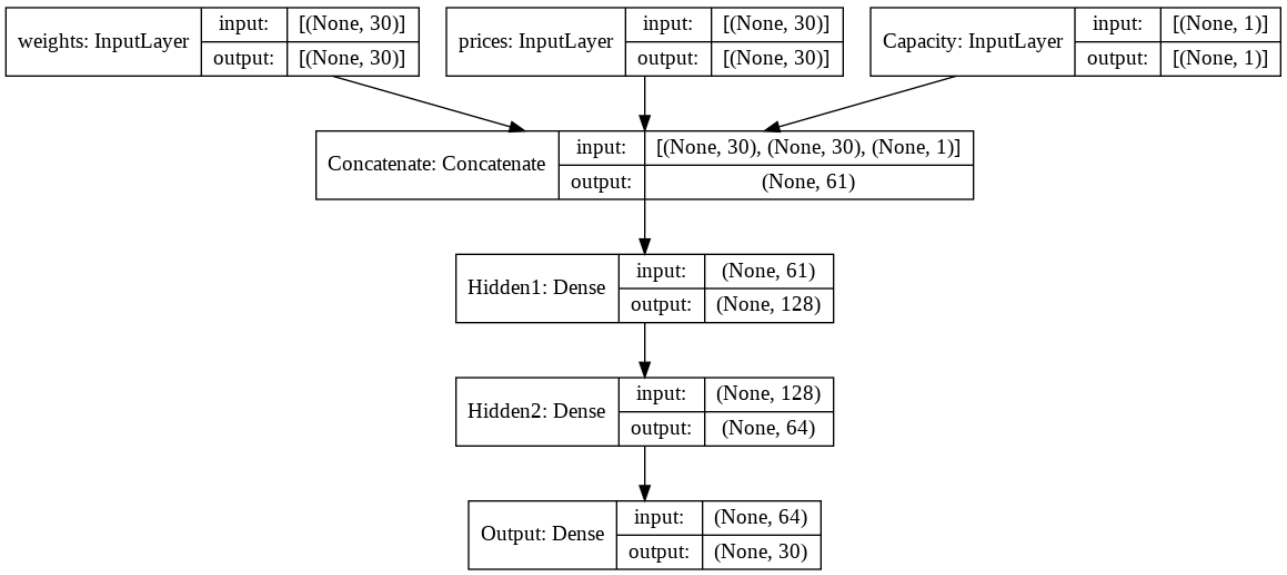

Fig 23. Model architecture

5.3.2 Simple knapsack problem with 5 items

A 10,000 value dataset is built which has 5 items. The prices(v) are selected in random between 1 to 45 using uniform distribution. Similarly the weights(w) are selected between 1 to 45 and the capacity(C) is between 1 to 99. The knapsack is applied using brute force to obtain the optimal set of x values for each of the data. The λ value is taken as 10.

The loss function used in this case is the L_1 loss function. When applied and tried over 1000 epochs, the accuracy obtained is 71%. More importantly is the number of constraints satisfied by this new model compared to the model which directly uses Binary cross entropy as loss function. After training, when predicting on the test data containing 3000 datavalues, the number of data that

satisfies the constraints were noted.

Model	Number of outputs satisfying constraints	Number of outputs not satisfying constraints
Constrained Model	2640	359
Unconstrained model	130	2869

As it is observed, the model with penalty function seems to make sure that the outputs satisfy the constraints better than the unconstrained model.

5.3.3 Training on more complex dataset

A more complex dataset is taken. A 200,000 value dataset is built which has 30 items. The prices(v) are selected in random between 1 to 5 using uniform distribution. The weights(w) and the capacity(C) are selected between 1 x 10^{-14} to 5 x 10^{-14}. Here we use the following error metric to find out the average over dataset of the term $\sum_i v_i \hat{x}_i$, which is described as pred_c. Ideally for the given dataset the values should reach about 72.

The outputs are either zeros or ones as usual.

For the same model as explained in section 5.3.1, when the loss function is taken as L_1 and model is trained over 50 epochs, the following is observed

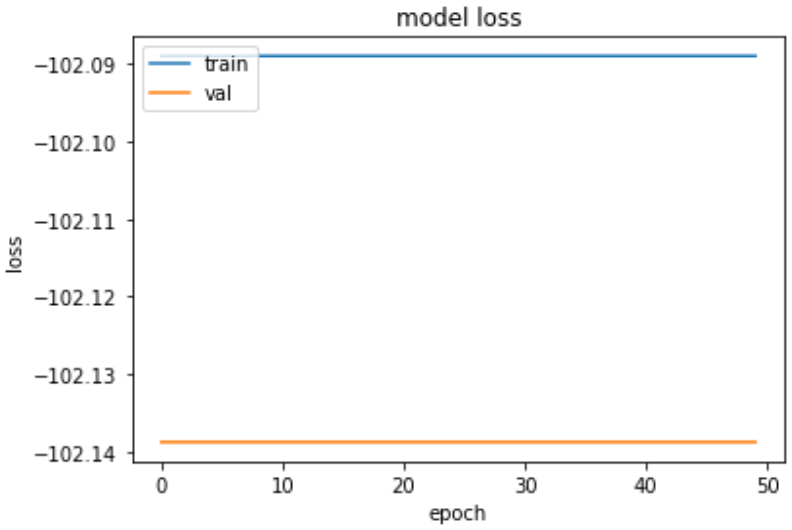

Fig 24. Model loss (L_1)

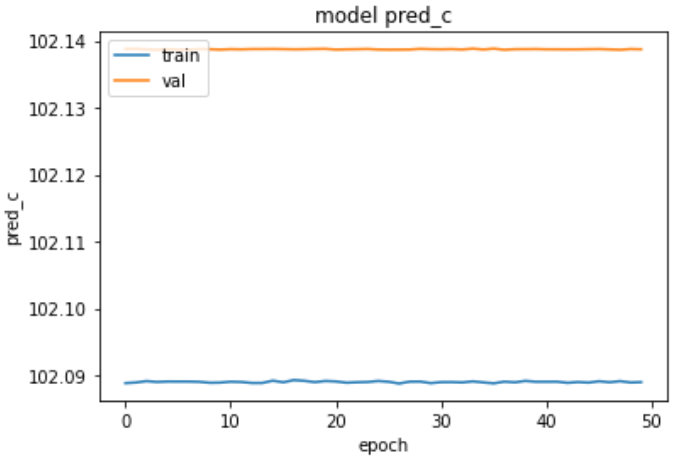

Fig 24. Model pred_c(L_1)

When L_2 or L_3 is taken as loss function,

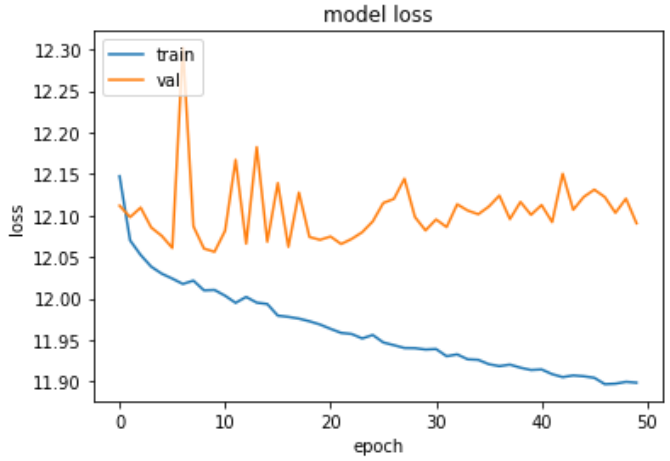

Fig 24. Model loss (L_2)

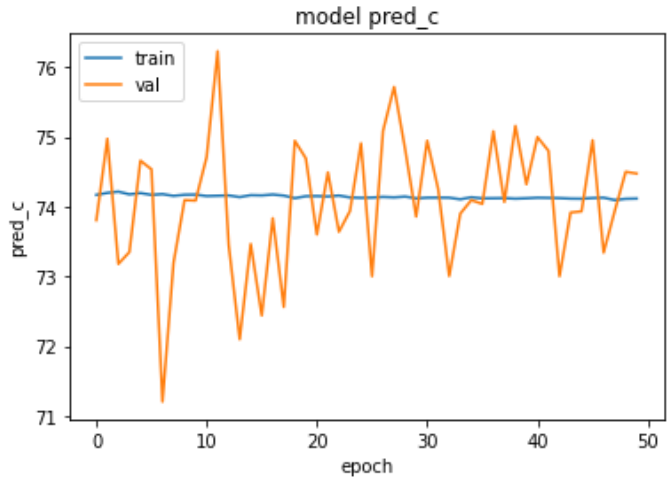

Fig 24. Model pred_c(L_2)

The same model is trained over 500 epochs using the L_1 loss function. The following is observed

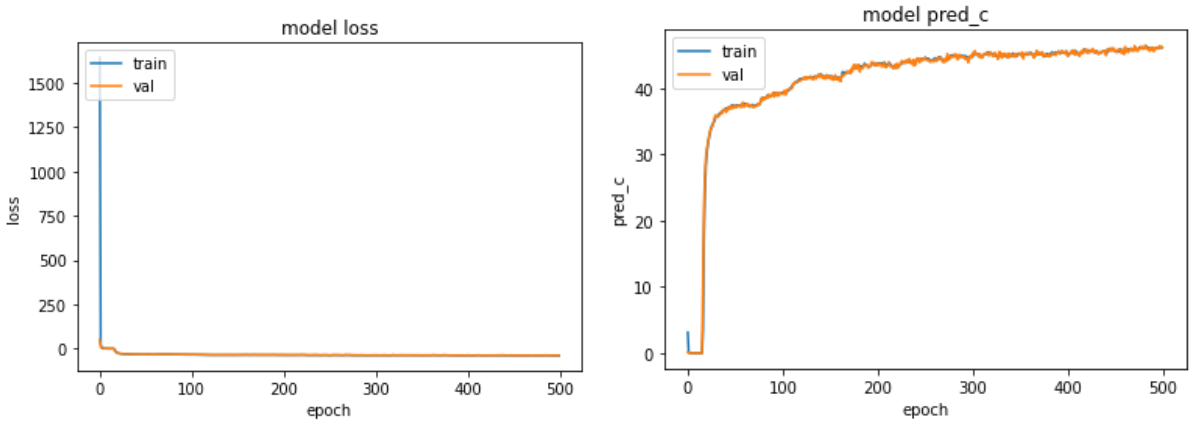

Fig 24. Model loss (L_1) 500 epochs

Although the model seems to perform better with the increase in epochs, the most important factor to look into is the amount of outputs satisfying the constraint. The table below shows the loss functions used, λ values used and how the outputs are predicted. Since sometimes certain λ values make the model to predict only 0s.

Loss function	λ	**output**	**Constraints NOT satisfied (percentage)**
L_1	$> 10^4$	all 0s	0%
L_2 and L_3	$> 10^4$	0s and 1s	90%
L_1	0.1	0s and 1s	73%
L_2 and L_3	100	0s and 1s	42%
L_2 and L_3	0.1	0s and 1s	80%

From the above table, it can be observed that choosing the right lambda value with the right loss function can produce the model that can learn and give out the best sub optimal values. Using better and custom optimizers for weight update of the neural network can result in better output.

6. Future work

Figuring out the reason why these models are failing is the major step to be taken. Some other methods of incorporating the constraint can be through applying them as layers after training. The problem with applying new layers is the need to figure out the BPA differentiation for the new

given layer. Reinforcement learning is another field which is being used widely for such problems. RL techniques have to be studied for this problem statement.

References

[1] M. N. Tehrani, M. Uysal, and H. Yanikomeroglu, “Device-to-device communication in 5G cellular networks: challenges, solutions, and future directions,” IEEE Commun. Mag., vol. 52, no. 5, pp. 86–92, May 2014

[2] B. V. R. Gorantla and N. B. Mehta, "Subchannel Allocation with Low Computational and Signaling Complexity in 5G D2D Networks," *ICC 2021 - IEEE International Conference on Communications*, 2021, pp. 1-6, doi: 10.1109/ICC42927.2021.9500968

[3] H. A. A. Nomer, K. A. Alnowibet, A. Elsayed and A. W. Mohamed, "Neural Knapsack: A Neural Network Based Solver for the Knapsack Problem," in *IEEE Access*, vol. 8, pp. 224200-224210, 2020, doi:10.1109/ACCESS.2020.3044005

[4]Vinyals, Oriol, Meire Fortunato, and Navdeep Jaitly. "Pointer networks." *arXiv preprint arXiv:1506.03134* (2015).

[5] Fioretto, Ferdinando, et al. "Lagrangian duality for constrained deep learning." *arXiv preprint arXiv:2001.09394* (2020).

[6] Nandwani, Yatin, Abhishek Pathak, and Parag Singla. "A primal dual formulation for deep learning with constraints." *Advances in Neural Information Processing Systems*. 2019.

[7] Bayati, Seyedrazieh, and Faramarz Jabbarvaziri. "Learning to Optimize Under Constraints with Unsupervised Deep Neural Networks." *arXiv preprint arXiv:2101.00744* (2021).

Books referred

1. Deep Learning By Ian Goodfellow
2. Neural Networks and Deep Learning by Charu C Aggarwal
3. Linear Algebra and Optimization for Machine Learning by Charu C Aggarwal

Tutorials and websites referred

1. CMU Deep learning youtube videos
2. Stanford ML and AI youtube videos
3. Medium and Towards Data Science Posts

The language used for coding experiments is python. The jupyter notebook environment is used along with Google Colab cloud services. The major important data/numerical analysis libraries used include numpy, pandas and matplotlib. The Deep Learning library used is Tensorflow (Keras).