MODELLING IN OPERATIONS MANAGEMENT

NEWSVENDOR INVENTORY OPTIMIZATION USING MACHINE LEARNING

Presented by : Group 10

Parth Tyagi 20BM6JP03
Anuj Maingi 20BM6JP16
Rahul Sharma 20BM6JP17
Prasun Kumar 20BM6JP26
Satyam Anand 20BM6JP35
Sisir Kumar Das 20BM6JP59
Satyam Neelmani 20BM6JP62

1. Introduction:

Inventory optimization is the practice of having the right inventory to meet your target service levels while tying up a minimum amount of capital in inventory. To achieve this, we need to account for the demand volatility.

Optimizing your inventory means that you will determine exactly how much you need to order of every single SKU and when you need to order it to always be able to serve your customers. Inventory optimization takes seasonality and campaigns into account as well as supplier lead times and schedules. This way you will always have the right products in the right warehouse without tying up too much capital in inventory.

These demand forecasting arise mainly in two types:

1. Repeated decision making: When we have to stock up the inventory at a regular time interval. Example: Stocking up newspaper for a daily.
2. Non-repeated decision: When we have to make a decision only once or after an indefinite time gap. Example: hiring secretory

Our focus through this project will stay on the repeated decision making using Multi Feature Newsvendor Model.

The standard objective in the newsvendor model is the expected profit maximization. This arises from the issue of making decision again and again for a regularly changing inventory.

To understand this, we can imagine a newsvendor who has to stock up newspaper every day and sell them during the day. The issue he faces is of stocking appropriate quantity to buy and stock newspaper every day.

If he stocks too many that will result in throwing away extra collected amount. If he stocks too few that would mean lost opportunity.

This model is usually applied to perishable goods that have a limited

selling season; they include fresh produce, newspapers, airline tickets, and fashion goods. But due to the simplicity of the model it is sometimes applied on non-perishable items as well like in the stock market.

The overall problem concludes to estimating the total demand for his product. Since this demand is variable, the problem assumes that the company purchases the goods at the beginning of a time period and sells them during the period. At the end of the period, unsold goods must be discarded, incurring a holding cost.

In addition, if it runs out of the goods in the middle of the period, it incurs a shortage cost, losing potential profit. Therefore, the company wants to choose the order quantity that minimizes the expected sum of the two costs described.

The optimal order quantity for the newsvendor problem can be obtained by solving the following optimization problem:

$$\min_y C(y) = E_d\left[c_p(d-y)^+ + c_h(y-d)^+\right]$$

where d is the random demand, y is the order quantity, c_p and c_h are the per-unit shortage and holding costs (respectively), and $(a)^+ := \max\{0, a\}$.

To apply the above function we need to have the demand distribution known or can be estimated using the data. If *F(.)* is the cumulative density function of the demand distribution and $F^{-1}(.)$ is its inverse, then the optimal solution can be obtained as:

$$y^* = F^{-1}\left(\frac{c_p}{c_p + c_h}\right) = F^{-1}(\alpha)$$

where $\alpha = \frac{c_p}{c_p + c_h}$.

2. Data Description:

We have taken 13,170 data records from Food Market Shop data records. It consists of specifically 3 features i.e. department, day of the week and month of the year. Total explanatory variables(X) is 42 which consists of 23 departments, 7 days of week and 12 months of 2 years data.

There is further scope to choose more data from different product and large number of records. To keep this with limited computation scope, we choose above data. Also there could be more feature

consideration like temperature, other environmental factors, festival seasons etc. We have lot of categorical value in the data which is processed with machine learning one-hot encoding. Finally we have 43 columns and 13,170 rows of data.

Now we divided our data as 75% Train data, 12.5% Validation data and 12.5% Test data. Just to be more precise we train our model with 75% of data and tune with hyper-parameter by 12.5% of data and finally test the model with rest 12.5% of data. Also we visualize the data to understand primary trend and distribution, which we will discuss in next section.

Response variable (Y) is demand. Glance of data is given below:

```
In [190]: data.head()
```

Out[190]:

	0	1	2	3	4	5	6	7	8	9	...	34	35	36	37	38	39	40	41	42	Demand
0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	48.0
1	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	189.0
2	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	314.0
3	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	200.0
4	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	133.0

Dataset: http://pentaho.dlpage.phi-integration.com/mondrian/mysql-foodmart-database

3. Hypothesis Testing:

The Kolmogorov-Smirnov test (also known as the K-S test or one-sample Kolmogorov-Smirnov test) is a nonparametric procedure that determines whether a sample of data comes from a specific distribution, we have tried with normal, exponential and lognormal distribution.

It is mostly used for evaluating the assumption of univariate normality by taking the observed cumulative distribution of scores and comparing them to the theoretical cumulative distribution for a normally distributed variable.

A **p-value** higher than 0.05 (> 0.05) is not statistically significant and indicates strong evidence for the **null hypothesis**. This means we retain the **null hypothesis** and reject the alternative **hypothesis**.

You should note that you cannot accept the **null hypothesis**, we can only reject the **null** or fail to reject it.

```
demand = train[,44]
normal.data = rnorm(9877, mean(demand), sd(demand))
ks.test(demand, normal.data)
```

Normal Distribution

- P value < 2.2e-16
- Very small p-value => Reject Null Hypothesis
- The demand data does not follow normal distribution

Log Normal Distribution

```
demand = train[,44]
lognormal = rnorm(9877, mean(log(demand)), sd(log(demand)))
ks.test(demand, lognormal)
```

- The demand data does not follow log normal

4. Statistical Methods

i. EQ Method:

This approach involves sorting the demand observations in ascending order and then estimating the $alpha_{th}$ quantile of the demand distribution

i.e., it selects the demand d_j such that j = Ceil of [n * (c_p/c_p+c_h)]. This quantile is then used as the base-stock level. Since they approximate the $alpha_{th}$ quantile, we refer to their method as the empirical quantile (EQ) method.

Importantly, EQ does not assume a particular form of the demand distribution and does not approximate the probability distribution, so it avoids those pitfalls. However, an important shortcoming of this approach is that it does not use the information from features.

ii. Clustering Followed by EQ Method:

We can extend above approach to the MFNV by first clustering the demand observations and then applying their method to each cluster. However, like the classical newsvendor algorithm, this would only allow it to consider categorical features and not continuous features, which are common in supply chain demand data.

Moreover, even if we use this clustering approach, the method cannot utilize any knowledge from other data clusters, which contain valuable information that can be useful for all other clusters. Finally, when there is volatility among the training data, the estimated quantile may not be sufficiently accurate, and the accuracy of EQ approach tends to be worse.

5. Machine Learning Methods

i. KNN:

In general, we apply several machine learning (ML) methods on a general optimization problem given by:

$$z^*(x) = \underset{z}{\operatorname{argmin}}\, \mathbb{E}\left[c(z,y)|x\right],$$

Where $(x_1; y_1)$ …… $(x_N; y_N)$ are the available data—in particular, x_i is a d-dimensional vector of feature values and y_i is the uncertain quantity of interest, e.g., demand values—and z is the decision variable.

KNN identifies the set of k-nearest historical records to the new observation x such that:

$$\mathcal{N}(x) = \left\{ i = 1, \ldots, n : \sum_{j=1}^{n} \mathbb{I}\{||x - x_i|| \geq ||x - x_j||\} \leq k \right\}$$

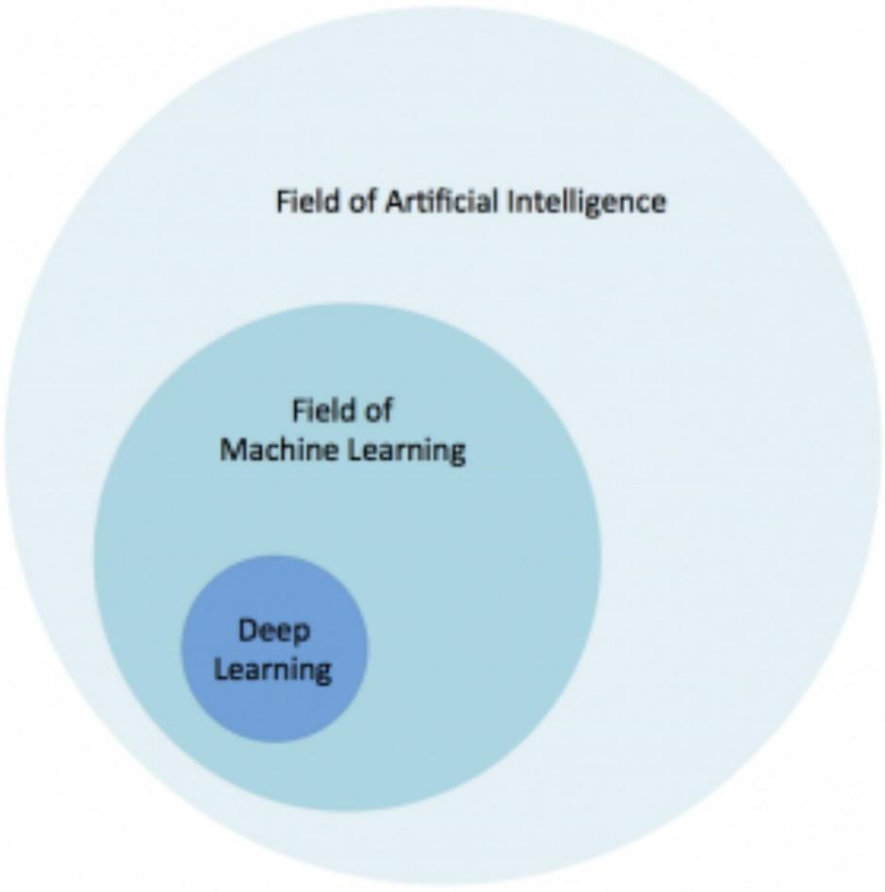
It assigns weight $w_i = 1/k$ for all i in N(x) (and zero otherwise) and call it weighted SAA; For example, if applied to the newsvendor problem, the SAA might take the form:

$$\inf\left\{d_j : \sum_{i=1}^{j} w_i \geq \frac{c_p}{c_p + c_h}\right\}$$

where d_j are the ascending sorted demands.

ii. Deep Learning

Deep learning, or deep neural networks [DNN], is a branch of machine learning that aims to build a model between inputs and outputs. It is a subfield of machine learning which is itself a subfield of artificial intelligence. Deep learning has seen great success in applications like image processing, speech recognition, time series forecasting, and—most relevant to our work—demand prediction. Deep Learning, despite its great success also has some shortcomings, one major criticism of deep learning (in non-vision-based tasks) is that it lacks interpretability—that is, it is hard for a user to discern a relationship between model inputs and outputs. Also, it usually needs careful hyper-parameter tuning, and the training process can take many hours or even days.



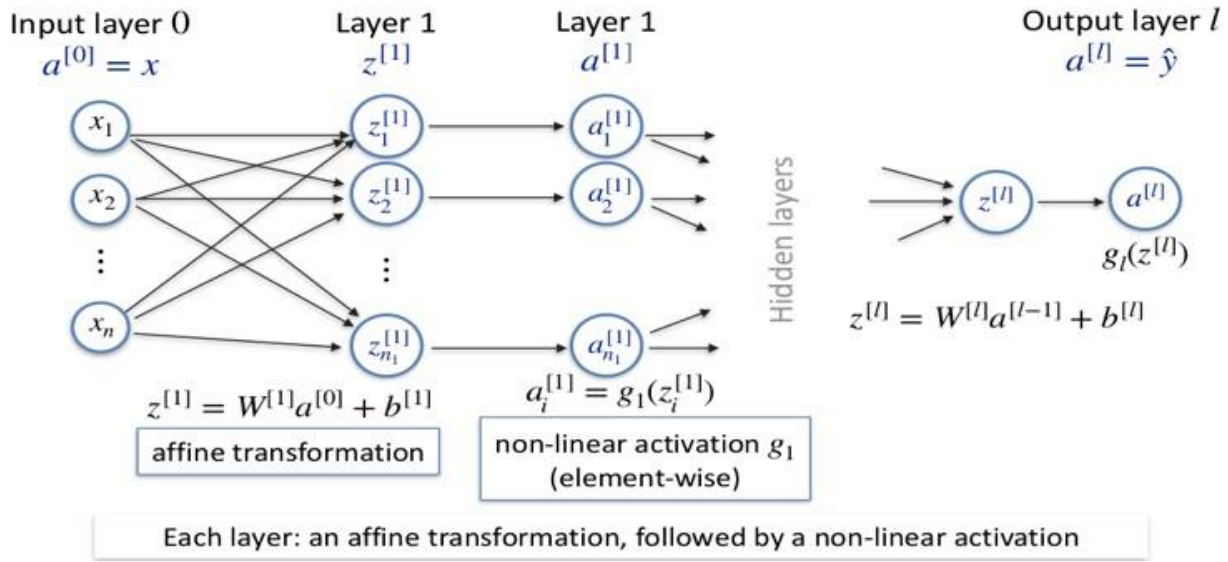
DNN uses a cascade of many layers of linear or nonlinear functions to obtain the output values from inputs. A general view of a DNN is shown in Figure below. The goal is to determine the weights of the network such that a given set of inputs results in a true set of outputs. A loss function is used to measure the closeness of the outputs of the model and the true values. The most common loss functions are the hinge, logistic regression, softmax, and Euclidean loss functions. The goal of the network is to provide a small loss value, i.e., to optimize:

$$min_w \frac{1}{n} \sum_{1}^{n} (l(\theta(x_i; w), y_i) + \lambda R(w)$$

where w is the matrix of the weights, xi is the vector of the inputs from the ith instance, θ(.) is the DNN function, and R(w) is a regularization function with weight λ. The regularization term prevents over-fitting and is typically the l1 or l2 norm of the weights. Over-fitting means that the model learns to do well on the training set but does not extend to the out-of-training samples; this is to be avoided.) DNN learns using the Forward and Backward Propogation steps which are explained below

a. Forward Propagation

Forward propagation refers to the calculation and storage of intermediate variables (including outputs) for a neural network in order from the input layer to the output layer. This is explained in the image below:



In forward propogation each layer does two major operations.

1. Affine Transformation - This refers to the process of multiplying the weights of the layer to the activations of the previous layer plus addition of the bias term. This is shown below:

$$Z^{[i]} = W^{[i]}a^{[i-1]} + b^{[i]}$$

where

$Z^{[i]}$ is the affine transformation of the i^{th} layer.

$W^{[i]}$ is the weights associated with the i^{th} layer.

$a^{[i-1]}$ is the output (activations) of the $(i-1)^{th}$ layer.

$b^{[i]}$ is the bias associated with the i^{th} layer.

2. Activations — This refers to the process of introducing non linearity in the network. This is performed after the affine transformation. The output of the affine transformation is passed through a non linear activation function and its output is what we call the output of the node. The most common activation functions are Sigmoid, Relu, Tanh etc. This is shown below:

$$a^{[i]} = g(Z^{[i]})$$

where

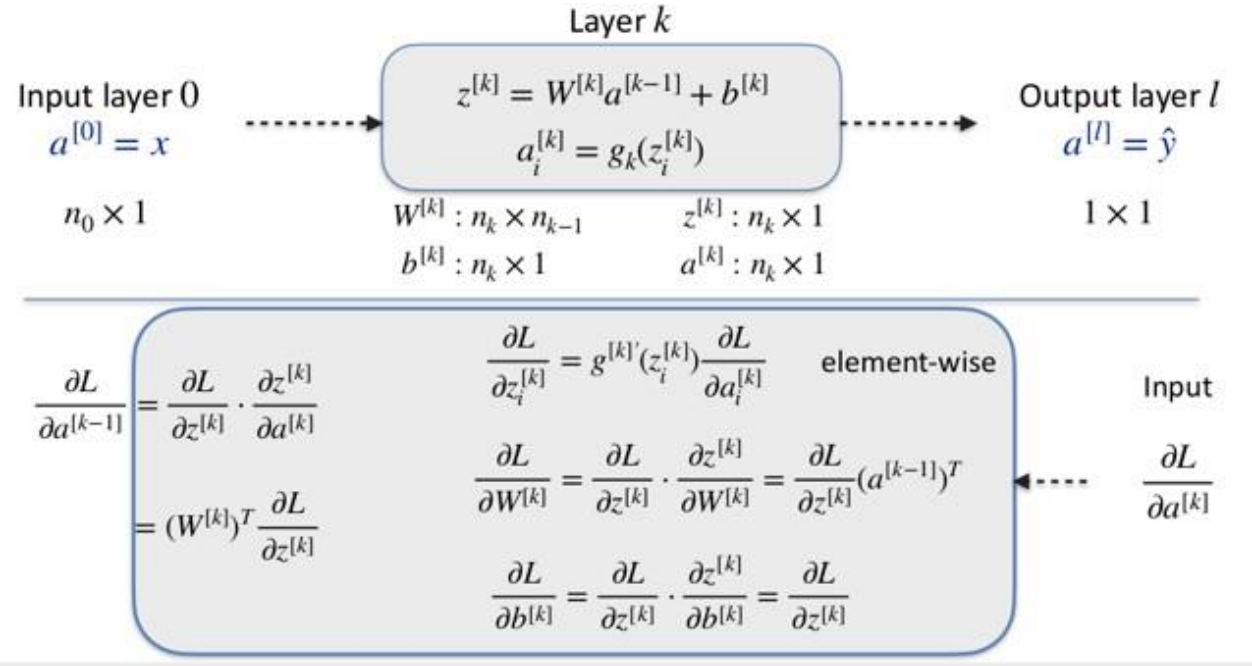
$a^{[i]}$ is the activation of the $i^{[th]}$ layer.

$Z^{[i]}$ is the affine transformation of the i^{th} layer.

g is the activation function.

b. Backward Propagation

Backpropagation refers to the method of calculating the gradient of neural network parameters. In short, the method traverses the network in reverse order, from the output to the input layer, according to the chain rule from calculus. The algorithm stores any intermediate variables (partial derivatives) required while calculating the gradient with respect to some parameters. The calculations of gradients for updation of the weight is done using the chain rule of calculus and some algorithm for learning weights (like the gradient descent) is followed. These steps are shown in the image below:



After calculations of the Gradient of the Cost w.r.t each weight and bias in the network, the weights and biases are then updated.

$$W_{after-updation} = W_{before-updation} - \alpha \nabla L_w$$

where

W represents the weight of the network

α is the learning rate

∇L_w is the gradient of the loss function w.r.t weight W

c. Why Deep Learning:

Traditional OR Methods rely on first estimating the demand distribution and then use it to estimate optimal order quantity. However in deep learning we directly estimate the optimal order quantity. The real value of our approach is that it is

effective for problems with small quantities of historical data, problems with unknown/unfitted probability distributions, or problems with volatile historical data —all cases for which the current traditional approaches fail.

d. Hyperband Algorithm

The objective of the Hyperband Algorithm is to find the best neural network architecture at the same time optimizing run time complexity.

In hyper band algorithm, we create 100 random neural network architecture with 2 or 3 hidden layer chosen randomly and with random number of nodes in these hidden layer (this is discussed in further sections). Then, in order to select the best network among these 100 random neural network architecture we follow the HyperBand algorithm, in which we train each of the 100 networks for five epochs (which is a full pass over the training dataset 5 times), we then obtain the results on the test set (using the newsvendor cost metric discussed further), and we then remove the worst performing 10% of the networks. We then resume training for another 5 epochs on the remaining networks and remove the worst 10%. This procedure is iteratively repeated to obtain the final best network.

e. Stochastic Architecture Creation

This is the initial step in the hyperband algorithm. Here we need to create the neural network architecture stochastically. Here we have two parameters to select randomly.

Number of hidden layer: We need to choose the number of hidden layers randomly. It can be 2 hidden layers or 3 hidden layers.

Number of Nodes in a hidden layer: Once we decide the number of hidden layers , we need to decide the number of nodes that each of the hidden layer should have randomly.

f. Case 1: when hidden layers = 2

For Network with 2 Hidden Layers

$$nn_2 \in [0.5nn_1, 3nn_1]$$
$$nn_3 \in [0.5nn_2, nn_2]$$
$$nn_4 = 1$$

When we have 2 hidden layers, we need to randomly select the nodes from the above mentioned range. Here nni represents the number of nodes to be put in layer i. Also, the number of neurons in the subsequent layers is dependent on the previous layer. This is done in order to make its leaning more efficient.

g. Case 2: when hidden layers = 3

For Network with 3 Hidden Layers

$$nn_2 \in [0.5nn_1, 3nn_1]$$
$$nn_3 \in [0.5nn_2, 2nn_2]$$
$$nn_4 \in [0.5nn_3, nn_3]$$
$$nn_5 = 1$$

When we have 3 hidden layers, we similarly need to randomly select the nodes from the above mentioned range. Here nni represents the number of nodes to be put in layer i. Also, the number of neurons in the subsequent layers is dependent on the previous layer. This is done in order to make its leaning more efficient.

```
def random_nodes(hidden_layers):


    l_bound = 0.5

    if hidden_layers == 2:
        u_bound = [0.,3.,1.]
    elif hidden_layers == 3:
        u_bound = [0.,3.,2.,1.]
    else:
        raise NotImplementedError

    n_nodes = []
    n_nodes.append(NUM_FEATURES)

    for i in range(1, hidden_layers+1):
        n = int(np.round((np.random.rand()*(u_bound[i]-l_bound) + l_bound) * n_nodes[i-1]))
        n_nodes.append(n)

    n_nodes.append(1)

    n_nodes = np.array(n_nodes).astype(np.int)

    print(f'Generating network with {hidden_layers+2} layers, each with {n_nodes} nodes')

    return n_nodes

def random_nn_params():

    hidden_layers = np.random.randint(NN_MIN_LAYER, NN_MAX_LAYER+1)
    n_nodes = random_nodes(hidden_layers)
    lr = np.power(10, np.random.rand()*3. - 5.)
    lambd = np.power(10, np.random.rand()*3. - 5.)

    return hidden_layers, n_nodes, lr, lambd # Lambd is weight decay
```

Above is shown the code to create the stochastic network.

The function random_nodes() is used to create the range of nodes to be selected at each level of layer. The function random_nn_params() is used to randomly generate the number of layers and based on that generate the number of nodes in each layer using the function random_nodes(). Python's PyTorch framework has been used here to create the functions.

h. Loss Functions

Next choice that we have to make is of the loss function.

In the original problem the cost function used is mentioned below:

$$min[c_h(y-d)^+ + c_p(d-y)^+]$$

Now we consider two loss function.

One is the same as the original cost. We would like the model to learn to minimize this cost itself during training procedure. The equation of the cost function is shown below:

$$E_i = \begin{cases} \left|c_p(d_i - y_i)\right|, & \text{if } y_i < d_i, \\ \left|c_h(y_i - d_i)\right|, & \text{if } d_i \leq y_i. \end{cases}$$

Also we consider a variation of it. In there we used square loss in it , as squared loss generally gives us a convex solution of which global minima can be found. Also it will be continuous and differentiable at all points.

The equation of the same is shown below:

$$E_i = \begin{cases} \frac{1}{2}\left|\left|c_p(d_i - y_i)\right|\right|_2^2, & \text{if } y_i < d_i, \\ \frac{1}{2}\left|\left|c_h(y_i - d_i)\right|\right|_2^2, & \text{if } d_i \leq y_i. \end{cases}$$

The code used to create the above loss functions is shown below:

```
class EuclideanLoss(nn.Module):

    def __init__(self, c_p, c_h):
        super().__init__()
        self.c_p = c_p
        self.c_h = c_h

    def forward(self, y, d):
        '''
        y:  prediction, size = (n_product, n_obs)
        d:  actual sales, size = (n_product, n_obs)
        '''

        diff = torch.add(y, -d)
        diff = diff.to(device)
        diff = torch.add(torch.mul(torch.max(diff, torch.zeros(1).to(device)), self.c_h),\
                         torch.mul(torch.max(-diff, torch.zeros(1).to(device)), self.c_p))
        diff = torch.norm(diff)
        diff = torch.sum(diff)
        diff = diff.to(device)
        return diff
```

The above code creates the squared loss using python's pytorch.

```
class CostFunction(nn.Module):

    def __init__(self, c_p, c_h):
        super().__init__()
        self.c_p = c_p
        self.c_h = c_h

    def forward(self, y, d):
        '''
        y:  prediction, size = (n_product, n_obs)
        d:  actual sales, size = (n_product, n_obs)
        '''

        cost = torch.add(y, -d)
        cost = torch.add(torch.mul(torch.max(cost, torch.zeros(1).to(device)), self.c_h),\
                         torch.mul(torch.max(-cost, torch.zeros(1).to(device)), self.c_p))
        cost = torch.sum(cost)
        cost = cost.to(device)

        return cost
```

i. Generating Architecture

Till now we have created functions for our building blocks but we need to physically create them now. To recap, these include, number of layers creation, number of nodes in each layer , weight initialization , do regularization and to leverage an efficient optimizer.

We use all the functions above to create two archictectures

- Deep neural network –L 1 (DNN-L1) :It uses all the above building blocks and used L1 cost function
- Deep neural network –L 2 (DNN-L2): It uses all the above building blocks and used L2 (Euclidean) cost function

The code to do the same is shown below:

```
def generate_fc(n_hidden, n_nodes):
    layers = []

    in_chan = n_nodes[0]
    for i in range(n_hidden + 2):
        out_chan = n_nodes[i]
        layers += [
            nn.Linear(in_chan, out_chan),
            nn.LeakyReLU(),
            nn.Dropout(DROPOUT)
        ]
        in_chan = out_chan

    return nn.Sequential(*layers)
```

The generate_fc() function generates the architecture.

```
class DeepVendorSimple(nn.Module):

    def __init__(self, model_type = 'simple_fc', n_hidden = 2, n_nodes = [4,3,2,1]):
        super().__init__()

        self.n_features = n_nodes[0]

        if model_type == 'simple_fc':
            self.net = generate_fc(n_hidden, n_nodes)
        else:
            raise NotImplementedError


    def forward(self, x):

        '''
        input   x:      size n_product by n_obs by n_features
        output  y:      size n_product
        '''
        y = self.net(x)
        y = y.squeeze()

        return y
```

The above code uses generate_fc() function to create instance of the class Deep Vender Simple. This creates a random network.

j. Procedure to create DNN-L1:

- Create 100 Feed Forward Neural Network models.
- Loss Function used is L1 (Cost Function)
- Iteratively remove the worst performing 10% models after training the models in steps of 5 epochs.
- Criteria for accessing the performance of the model is Euclidean Loss evaluated on the test set.
- Iterate till the final best model is obtained.

k. Procedure to create DNN-L2:

- Create 100 Feed Forward Neural Network models.
- Loss Function used is L2 (Euclidean Cost Function)
- Iteratively remove the worst performing 10% models after training the models in steps of 5 epochs.
- Criteria for accessing the performance of the model is Euclidean Loss evaluated on the test set.
- Iterate till the final best model is obtained.

I. DNN-L2Multi Level Optimization

The DNN-L2 model is optimized further by going sequence of grid search where our optimizer is Adam as shown below :

```
optimizer = torch.optim.Adam(
    model.parameters(),
    lr=lr,
    weight_decay=weight_decay)
```

Apart from this the learning rate used is shown below:

$$lr_t = lr \times (1 + \gamma \times t)^{-0.75}.$$

The grid search is shown below in which the following parameters are tuned-

$$\gamma \in \{0.001, 0.0008, 0.0009\}$$
$$\lambda \in \{0.01, 0.005, 0.001, 0.0005\}$$
$$\mathbf{lr} \in \{0.001, 0.0007, 0.0005, 0.0001\}$$

This created 48 models.

The minimum cost model parameters are used to create another exhaustive grid which is shown below:

$$\gamma \in \{0.01, 0.005, 0.001, 0.0001, 0.0005, 0.00005\}$$

$$\lambda \in \{0.01, 0.005, 0.001, 0.0005, 0.0001, 0.00005\}$$

$$\text{lr} \in \{0.001, 0.005, 0.0005, 0.0001, 0.00005, 0.00003, 0.00001, 0.000009, 0.000008, 0.000005\}$$

This made a total of 360 models.

The set of parameter giving the lowest cost is shown below:

```
In [29]: DNN_Grid_Search.best_params

         DNN-L2_tuned(lr = 0.000009 , weight_decay = 0.00005 , gamma = 0.00005 )
```

m. Deep Learning Results

Now we display initially the results from the deep learning models and then compare them with the rest of the models.

Algorithm	(cp , ch)				
	(1,1)	(2,1)	(5,1)	(10,1)	(20,1)
DNN-L1	12.273	17.222	25.573	35.743	53.650
DNN-L2	16.848	23.153	38.970	48.400	67.882
DNN-L2-T	14.883	20.455	32.489	45.998	63.877

We see that in all the scenarios DNN-L1 is performing better than all the other algorithms. The next performer is DNN-L2-Tuned followed by DNN-L2.

The reason for such high performance of the deep learning models is:

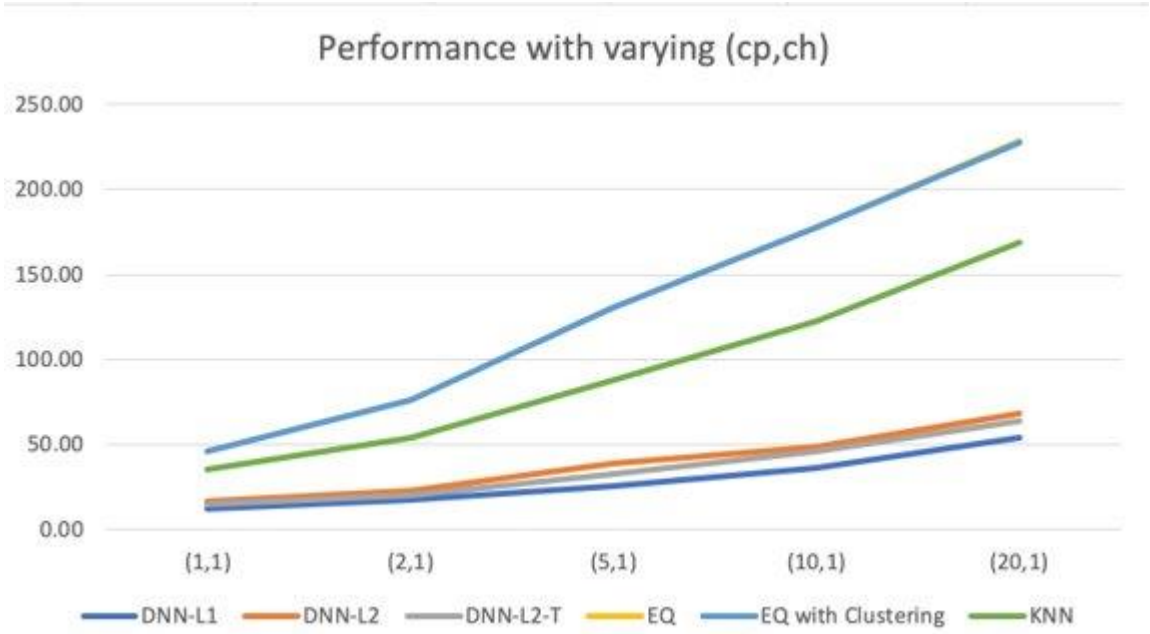
- Performance is powered by high learning capacity of neural networks
- Extremely sophisticated optimizer has been used: Adam
- Neural networks can effectively capture non-linearity
- Multi-level grid search has further improved the performance

6. Final Results

We checked the cost values for the various values of the combination of c_p and c_h. In the table below, we have listed down cost for each algorithm.

Algorithm	(cp , ch)				
	(1,1)	(2,1)	(5,1)	(10,1)	(20,1)
DNN-L1	12.273	17.222	25.573	35.743	53.650
DNN-L2	16.848	23.153	38.970	48.400	67.882
DNN-L2-T	14.883	20.455	32.489	45.998	63.877
EQ	45.776	76.351	130.526	177.590	228.102
EQ with Clustering	45.717	76.322	130.725	177.551	227.331
KNN	35.530	53.710	88.090	122.700	168.880

The best performing algorithm comes out to be DNN-L1 followed by DNN-L2-T and others. All three deep learning models outperformed the traditional ML method and EQ methods. For better understanding of results, lets plot the cost values for each combination of (cp, ch) and for each algorithm.



We observe that as value of c_p increases, the optimal cost keeps on increasing which is quite intuitive the cost incurred will be higher if c_p (shortage cost) increases. We also see that the slope of traditional methods such as EQ and KNN is higher, thus implying that there optimal cost increases at a much higher rate as we increase c_p.

7. Conclusion

In this project, we have attempted to provide a robust solution for Multi-Feature Newsvendor (MFNV) problem. If the demand distribution is known to us then exact solution exists for MFNV. But in real world, we never know the demand distribution. So, we will have to estimate the demand using machine learning and statistical methods. We used KS test to see if our demand follows any standard distribution, but we found that our demand data could not be fit by any of the standard distributions. Thus we moved on to exploring EQ, KNN and Deep Learning methods.

The proposed method does not require the prior information about the demand distribution. Deep Learning methods only require historical data to find the pattern. The reason for outstanding performance by DNN can be attributed to the fact that deep learning networks are vey good at learning the complex relations between taget and features. We have used the

cost function of deep learning same as the actual cost function that needs to be minimized. So it performs much better than other algorithms.

8. Future Scope:

One can add new features to the dataset for better performance of the model. The new features can be weather condition, stock prices, location etc.

In this project we have proposed the solution to MFNV problem by considering a single location. But many big companies use a distribution channel to sell their products.

Fig – Distribution Channel

For solving such problems, it is better to optimize the entire chain at once rather than individually optimizing each location. One can use mult-echelon inventory optimization methods to solve such problems.

9. References

i. Applying Deep Learning to the Newsvendor Problem, Afshin Oroojlooyjadid et.al.
ii. https://medium.com/opex-analytics/machine-learning-for-inventory-optimization-38a9ac86a80a
iii. Deep Learning Forward prop and Backprop Slides Credit – Debapriyo Majumdar, ISI
iv. Foodmart Database - http://pentaho.dlpage.phi-integration.com/mondrian/mysql-foodmart-database