## PART 1: TRAIN AND VALIDATE YOUR OWN N-LAYER NEURAL NETWORK ON THE APPAREL DATASET TO PREDICT THE CLASS LABEL OF A GIVEN APPAREL.

**Implementation Details:**

- The neural network is modeled in a class called ‘Model’. It has all the functions required to train and test the model which are described later.
- Each layer has its own class called ‘Layer’ containing various parameters, which are listed below.
- Cross entropy is used as a measure of cost because it suits well for classification problems.
- The neural network is flexible enough to have as many hidden layers as we want and apart from every layer can hold any desired number of neurons.
- The activation functions used are sigmoid, softmax, relu and tanh.
- Best results are obtained with 1 hidden layers with size 1000, learning rate 0.001, number of epochs 15, hidden layer activation function sigmoid, output layer activation function softmax and batch size of 100 samples.

The class ‘Layer’ has the following functions and class variables:

- **Class Variables:**
  - **num_neurons:** number of neurons present in that particular layer.
  - **activation:** type of activation used in the layer.
  - **input[ ]:** list of the neuron value before applying activation function.
  - **output[ ]:** list of the neuron value before applying activation function.
  - **delta:** used at the time of back propagation.

- **Functions:**
  - **__init__(num_of_neurons, activation_type, batch_size):** initializes a layer using the three parameters: number of neurons, activation type and batch size.

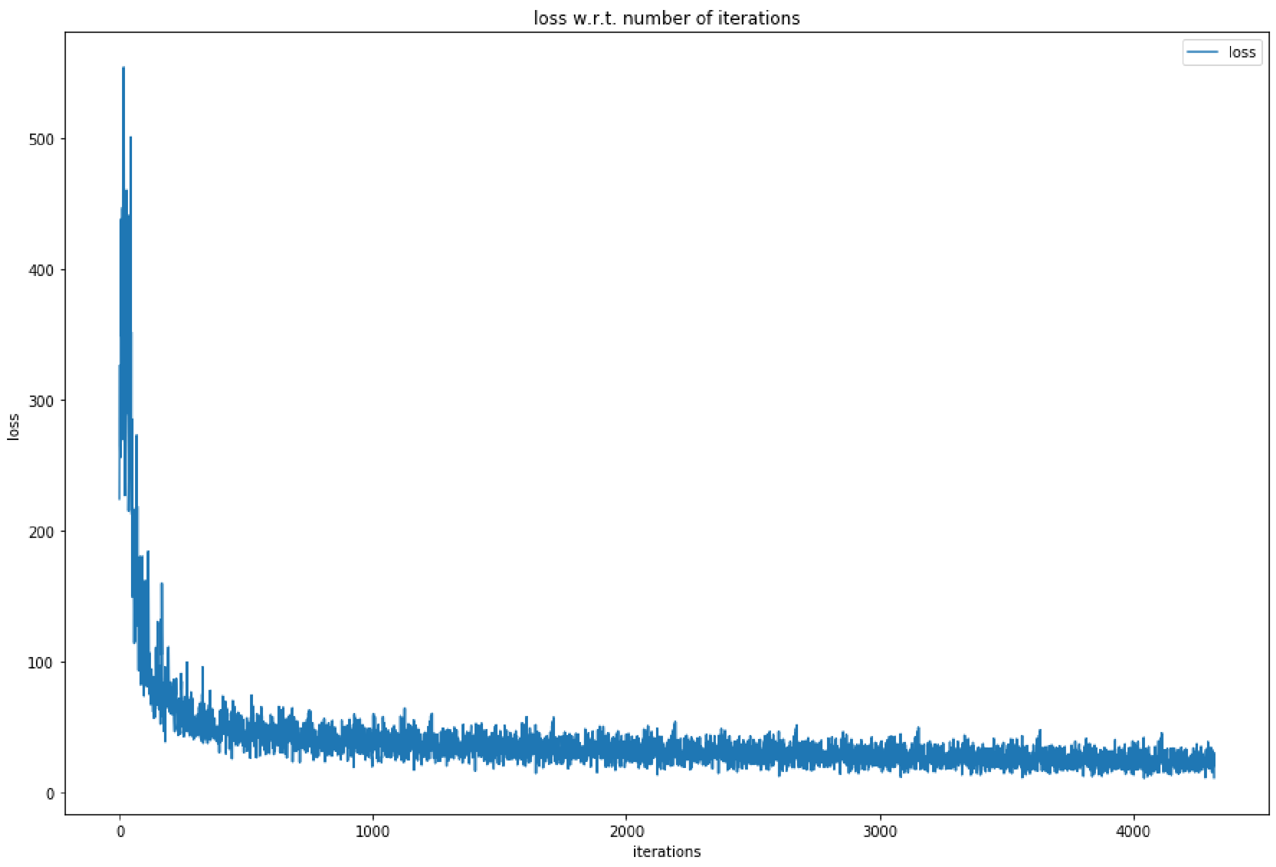
The class ‘Model’ has the following functions and class variables:

- **Class Variables:**
  - **layers[ ]:** a list of layers object
  - **weights{ }:** a dictionary where all the weights of the model are stored.
  - **weight_labels[ ]:** a list of labels through which the weight can be accessed at the time of updation.
  - **learn_rate:** learning rate
  - **batch:** size of the batch
  - **loss:** loss used

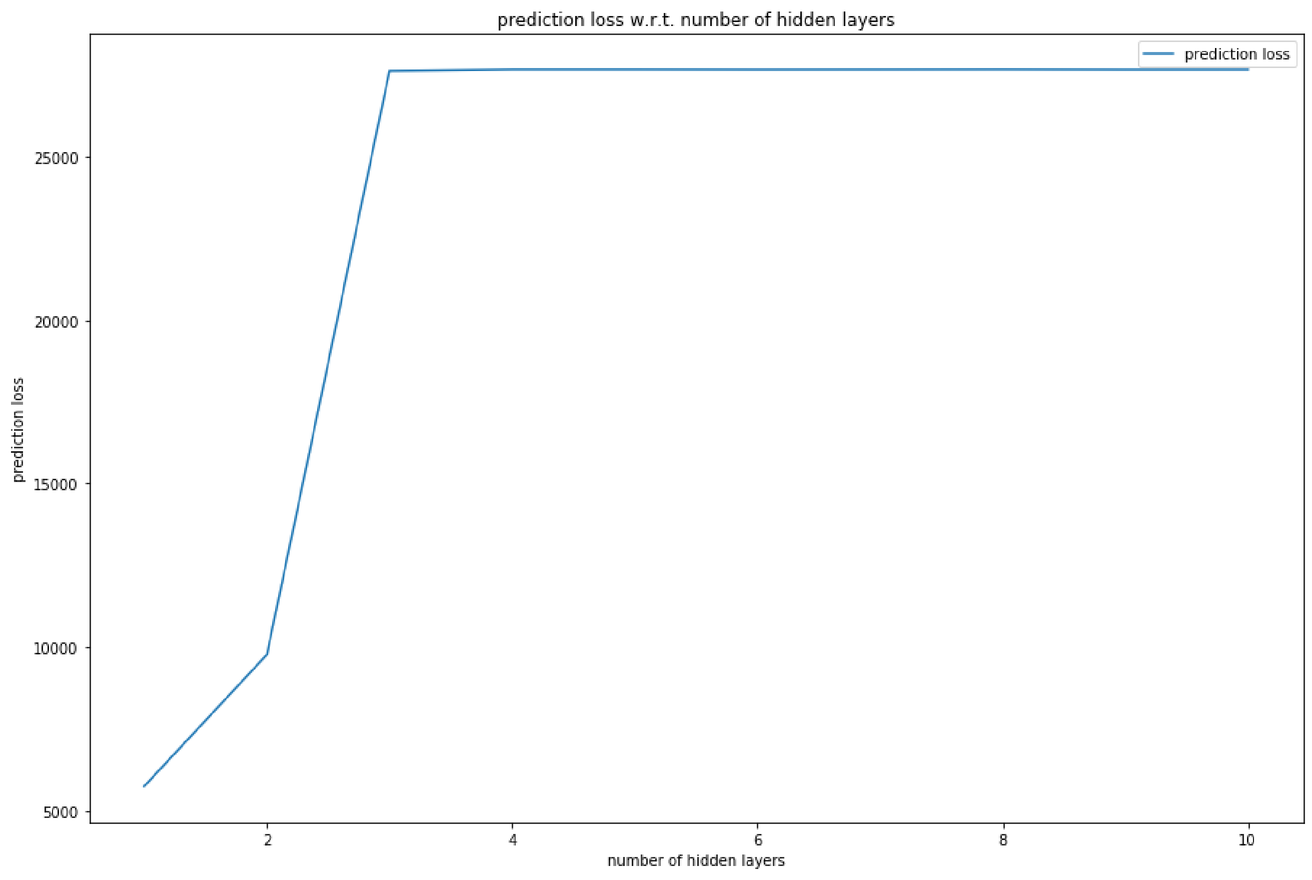
- **Functions:**
  - **__init__(batch_size):** takes the batch size for the model and initializes it.
  - **initialize_weights(input_layer_length):** initializes random weights.
  - **apply_activation(array, type_of_activation):** takes array and the type of activation which needs to be applied and returns the array after applying the activation.
  - **derivative_of_activation(array, type_of_activation):** takes array and the type of activation whose derivative needs to be applied and returns the array after applying the derivative.
  - **add(num_of_neurons, activation_type):** creates a new layer whose neurons equals with the num_of_neurons and activation function used with this layer is set to activation_type.
  - **compile(loss, learn_rate):** sets the class variables: loss and learn_rate
  - **forward_propagate(input_sample):** performs forward propagation.
  - **back_propagation(input_sample, labels):** performs backward propagation.
  - **predict(sample):** applys forward propagation and provides predictions wrt the provided sample.
  - **accracy(true, predicted):** returns the accuracy measure.
  - **convert_encoding(labels, flag):** one hot encodes all the labels and vice versa.
  - **fit(train_sample, train_labels, n_epochs):** performs forward and back propagation on the basis of number of epochs and batch_size.

After training the model and testing it on the validation set following results were observed:

- **Accuracy:** 0.876583333333
- With increase in the number of iterations the loss is decresing which proves the correctness of the implementation.



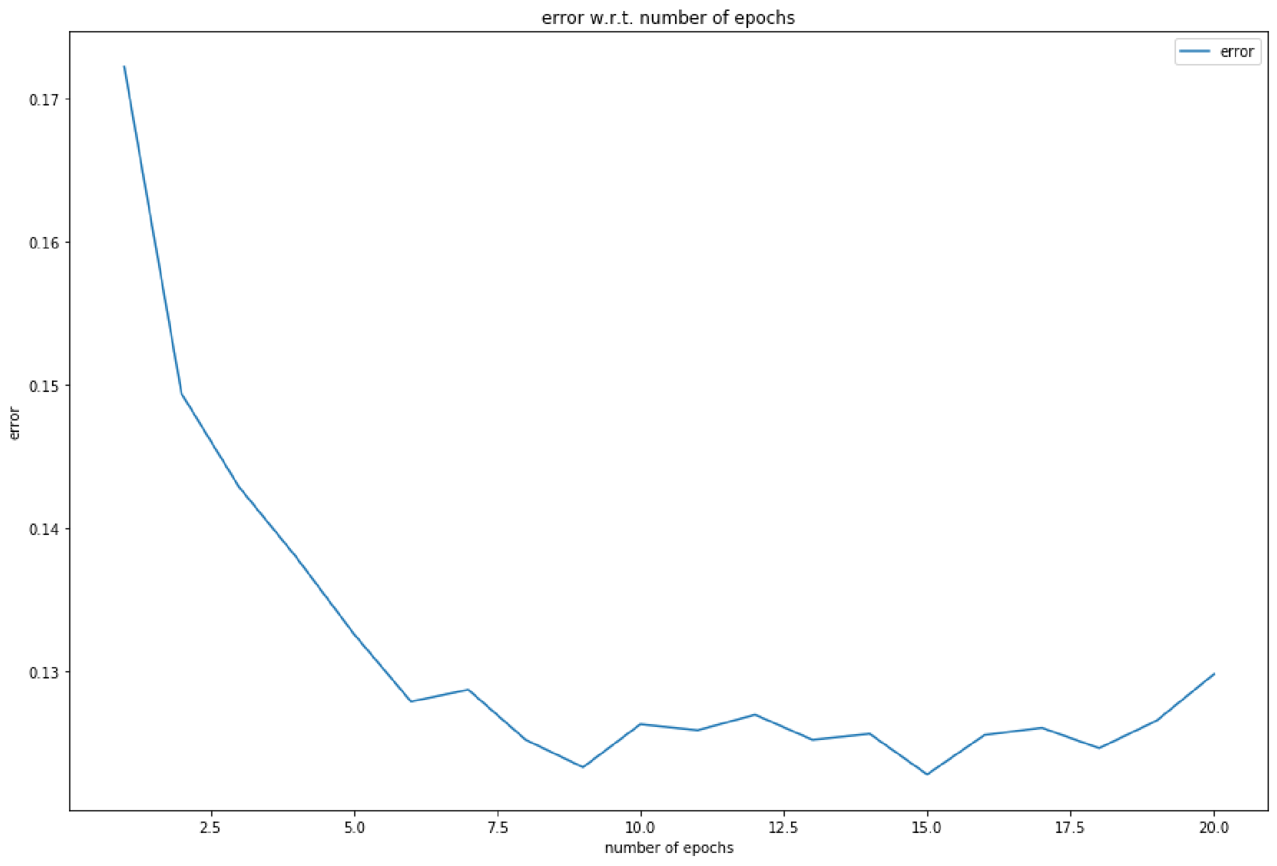
## PLOTTING PREDICTION LOSS AS A FUNCTION OF NUMBER OF HIDDEN LAYERS



## OBSERVATION:

- Increasing the number of hidden layers might improve the accuracy or might not, it really depends on the complexity of the problem that you are trying to solve.
- In our case it increases, the reasons for this increase might be because of overfitting or the training data is not enough to have multiple hidden layers.

## PLOTTING ERROR AS A FUNCTION OF NUMBER OF EPOCHS:



## OBSERVATIONS:

- As the number of epochs increases, the error reduces.
- The reason is that our gradient descent algorithm reaches more close to the minima as we increase number of epochs.
- But after some point it starts to increase due to overfitting.

## PART 2: HOUSE PRICE PREDICTION DATASET. WRITE A REPORT ON HOW YOU WOULD MODIFY THE ABOVE IMPLEMENTED NEURAL NETWORK FOR SUCH TASK.

The neural network designed in question 1 is designed for predicting class labels which is a classification problem. But this problem requires us to predict the house price which is a regression problem.

In our implementation we took the loss function as cross entropy which is a good fit for classification problems, but in the house prediction problem, we need some other loss function which may be mean square error or mean absolute error or mean percentage error. Choosing mean square error for this task is a good choice.
Changing the loss function from cross entropy to mean square error will cause change in the calculation of gradients. Currently we are calculation gradients of cross entropy with respect to weights and bias in the backward function, they need to be updated for mean square error as cost.
Since in regression problems, we have a single output, we need to change the number of nodes in output layer to one that can be done by simply passing the outputLayerSize parameter as one when we initialize our neural network.
In the house prediction problem, it would be a good choice to use relu activation function for the hidden layers and a linear activation function for the output layer since it is a resression problem and output can be any positive real number. In our implementation, we will need to add another activation function 'linear'. We can also use relu at the output layer since the house price will always be greater than zero.
The number of hidden layers will be more than one for this problem as the output will be a highly non linear function of the inputs, the exact number of hidden layers and number of nodes in each hidden layer can only be found by experimenting the validation results.
In question 1, we only had numerical attributes of pixel values that range from 0 to 255. But for the problem of house price prediction, there are several attributes that are not numerical but categorical. A good way to account for categorical features is to one hot encode them. This will result in increase in number of attributes but is a reliable way to handle categorical attributes. In our implementation, this can be done by reading the numerical and categorical attributes separately, then one hot encoding the categorical attributes and then finally combining all the attributes to form the train and validation data.
Another way to account for categorical values is to use one input for each category and scale the integer values, e.g. (0,...,11) for month to continuous values in the range of other input variables. However, this approach would assume that you have some hierarchy in the categories, let's say February is 'better' or 'higher' than January.
The optimal number of epochs and batch size can be decided by graph plotting as we did in question one. We will need to choose these parameters that result in minimum loss.