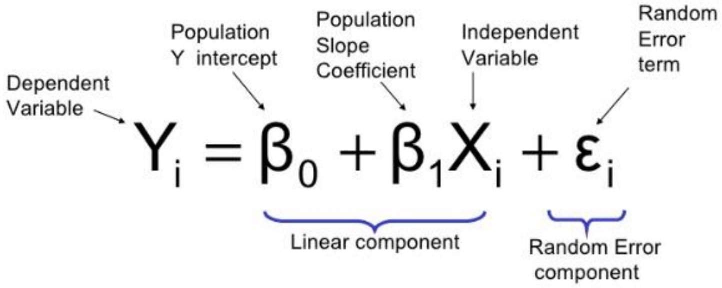
Assignment No. 1

- **Title:** Linear regression by using Deep Neural network
- **Objective:** Study and understand neural network by using Linear Regression for predicting house prices.
- **Problem statement:** Implement Boston housing price prediction problem by Linear regression using Deep Neural network. Use Boston House price prediction dataset.
- **Theory:**

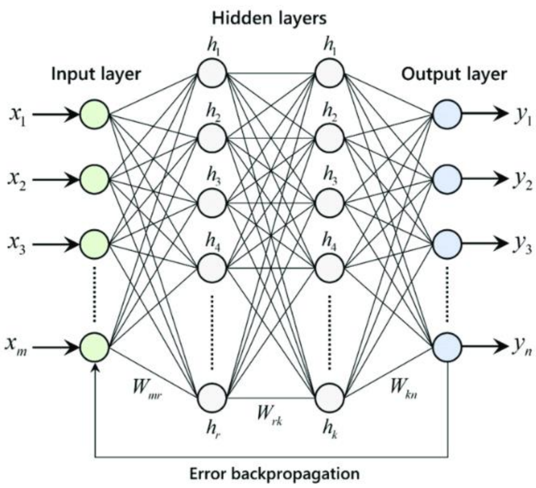
Linear Regression

Linear Regression is a supervised learning technique that involves learning the relationship between the features and the target. The target values are continuous, which means that the values can take any values between an interval. Use-cases of regression include stock market price prediction, house price prediction, sales prediction, and etc.



Neural network

Neural networks are formed when multiple neural layers combine with each other to give out a network, or we can say that there are some layers whose outputs are inputs for other layers.



The purpose of using Artificial Neural Networks for Regression over Linear Regression is that the linear regression can only learn the linear relationship between the features and target and therefore cannot learn the complex non-linear relationship. In order to learn the complex non-linear relationship between the features and target, we are in need of other techniques. One of those techniques is to use Artificial Neural Networks. Artificial Neural Networks have the ability to learn the complex relationship between the features and target due to the presence of activation function in each layer.

Artificial Neural Networks are one of the deep learning algorithms that simulate the workings of neurons in the human brain.

- **Code and Output:**

- **Conclusion:** We have successfully solved Boston house price prediction problem by Linear Regression using deep neural network.

In [1]:

```
# Importing the pandas for data processing and numpy for numerical computing
import numpy as np
import pandas as pd
```

In [ ]:

```
# Importing the Boston Housing dataset from the sklearn
from sklearn.datasets import load_boston
boston = load_boston()
```

In [3]:

```
# Converting the data into pandas dataframe
data = pd.DataFrame(boston.data)
```

In [4]:

```
data.head()
```

Out[4]:

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33

In [5]:

```
# Adding the feature names to the dataframe
data.columns = boston.feature_names
```

In [6]:

```
# Adding the target variable to the dataset
data['PRICE'] = boston.target
```

In [7]:

```
# Looking at the data with names and target variable
data.head()
```

Out[7]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	PRICE
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2

In [8]:

```
# Checking the null values in the dataset
data.isnull().sum()
```

Out[8]:

```
CRIM       0
ZN         0
INDUS      0
CHAS       0
NOX        0
RM         0
AGE        0
DIS        0
RAD        0
TAX        0
PTRATIO    0
```

```
PTRATIO    0
B          0
LSTAT      0
PRICE      0
dtype: int64
```

In [9]:

```python
# Checking the statistics of the data
data.describe()
```

Out[9]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000
mean	3.613524	11.363636	11.136779	0.069170	0.554695	6.284634	68.574901	3.795043	9.549407	408.237154	18.455534
std	8.601545	23.322453	6.860353	0.253994	0.115878	0.702617	28.148861	2.105710	8.707259	168.537116	2.164946
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000	1.129600	1.000000	187.000000	12.600000
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	45.025000	2.100175	4.000000	279.000000	17.400000
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	77.500000	3.207450	5.000000	330.000000	19.050000
75%	3.677083	12.500000	18.100000	0.000000	0.624000	6.623500	94.075000	5.188425	24.000000	666.000000	20.200000
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000	12.126500	24.000000	711.000000	22.000000

In [10]:

```python
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 506 entries, 0 to 505
Data columns (total 14 columns):
 #   Column   Non-Null Count  Dtype  
---  ------   --------------  -----  
 0   CRIM     506 non-null    float64
 1   ZN       506 non-null    float64
 2   INDUS    506 non-null    float64
 3   CHAS     506 non-null    float64
 4   NOX      506 non-null    float64
 5   RM       506 non-null    float64
 6   AGE      506 non-null    float64
 7   DIS      506 non-null    float64
 8   RAD      506 non-null    float64
 9   TAX      506 non-null    float64
 10  PTRATIO  506 non-null    float64
 11  B        506 non-null    float64
 12  LSTAT    506 non-null    float64
 13  PRICE    506 non-null    float64
dtypes: float64(14)
memory usage: 55.5 KB
```

In [11]:

```python
# X = data[['LSTAT','RM','PTRATIO']]
X = data.iloc[:,:-1]
y= data.PRICE
```

In [12]:

```python
# Splitting the data into train and test for building the model
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = 0.2, random_state = 4)
```

In [13]:

```python
# Linear Regression
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
```

In [14]:

```python
# Fitting the model
regressor.fit(X_train,y_train)
```

Out[14]:

```
LinearRegression()
```

In [15]:

```
# Prediction on the test dataset
y_pred = regressor.predict(X_test)
```

In [16]:

```
# Predicting RMSE the Test set results
from sklearn.metrics import mean_squared_error
rmse = (np.sqrt(mean_squared_error(y_test, y_pred)))
print(rmse)
```

```
5.041784121402041
```

In [17]:

```
# Scaling the dataset
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

In [18]:

```
# Creating the neural network model
import keras
from keras.layers import Dense, Activation,Dropout
from keras.models import Sequential

model = Sequential()

model.add(Dense(128,activation  = 'relu',input_dim =13))
model.add(Dense(64,activation  = 'relu'))
model.add(Dense(32,activation  = 'relu'))
model.add(Dense(16,activation  = 'relu'))
model.add(Dense(1))
model.compile(optimizer = 'adam',loss = 'mean_squared_error')
```

In [19]:

```
model.fit(X_train, y_train, epochs = 100)
```

```
Epoch 1/100
13/13 [==============================] - 1s 7ms/step - loss: 553.8900
Epoch 2/100
13/13 [==============================] - 0s 3ms/step - loss: 477.5840
Epoch 3/100
13/13 [==============================] - 0s 2ms/step - loss: 313.0531
Epoch 4/100
13/13 [==============================] - 0s 2ms/step - loss: 106.6664
Epoch 5/100
13/13 [==============================] - 0s 2ms/step - loss: 61.2741
Epoch 6/100
13/13 [==============================] - 0s 2ms/step - loss: 34.8987
Epoch 7/100
13/13 [==============================] - 0s 2ms/step - loss: 25.9169
Epoch 8/100
13/13 [==============================] - 0s 2ms/step - loss: 22.7645
Epoch 9/100
13/13 [==============================] - 0s 3ms/step - loss: 20.5753
Epoch 10/100
13/13 [==============================] - 0s 3ms/step - loss: 18.4250
Epoch 11/100
13/13 [==============================] - 0s 2ms/step - loss: 17.2400
Epoch 12/100
13/13 [==============================] - 0s 2ms/step - loss: 16.1215
Epoch 13/100
13/13 [==============================] - 0s 2ms/step - loss: 15.2485
Epoch 14/100
13/13 [==============================] - 0s 3ms/step - loss: 14.6573
Epoch 15/100
13/13 [==============================] - 0s 2ms/step - loss: 14.2568
Epoch 16/100
13/13 [==============================] - 0s 3ms/step - loss: 13.6369
Epoch 17/100
13/13 [==============================] - 0s 2ms/step - loss: 13.1030
Epoch 18/100
13/13 [==============================] - 0s 2ms/step - loss: 12.7586
Epoch 19/100
13/13 [==============================] - 0s 2ms/step - loss: 12.4448
Epoch 20/100
13/13 [==============================] - 0s 3ms/step - loss: 12.1456
Epoch 21/100
```

```
13/13 [==============================] - 0s 2ms/step - loss: 11.9045
Epoch 22/100
13/13 [==============================] - 0s 2ms/step - loss: 11.4259
Epoch 23/100
13/13 [==============================] - 0s 2ms/step - loss: 11.1804
Epoch 24/100
13/13 [==============================] - 0s 2ms/step - loss: 10.8369
Epoch 25/100
13/13 [==============================] - 0s 3ms/step - loss: 10.7921
Epoch 26/100
13/13 [==============================] - 0s 2ms/step - loss: 10.3413
Epoch 27/100
13/13 [==============================] - 0s 3ms/step - loss: 10.2014
Epoch 28/100
13/13 [==============================] - 0s 2ms/step - loss: 9.9532
Epoch 29/100
13/13 [==============================] - 0s 2ms/step - loss: 9.8095
Epoch 30/100
13/13 [==============================] - 0s 3ms/step - loss: 9.7494
Epoch 31/100
13/13 [==============================] - 0s 4ms/step - loss: 9.6172
Epoch 32/100
13/13 [==============================] - 0s 3ms/step - loss: 9.2688
Epoch 33/100
13/13 [==============================] - 0s 3ms/step - loss: 9.2943
Epoch 34/100
13/13 [==============================] - 0s 2ms/step - loss: 8.6004
Epoch 35/100
13/13 [==============================] - 0s 3ms/step - loss: 8.5056
Epoch 36/100
13/13 [==============================] - 0s 4ms/step - loss: 8.3584
Epoch 37/100
13/13 [==============================] - 0s 3ms/step - loss: 8.4113
Epoch 38/100
13/13 [==============================] - 0s 3ms/step - loss: 8.0936
Epoch 39/100
13/13 [==============================] - 0s 5ms/step - loss: 7.8241
Epoch 40/100
13/13 [==============================] - 0s 4ms/step - loss: 7.7833
Epoch 41/100
13/13 [==============================] - 0s 3ms/step - loss: 7.7164
Epoch 42/100
13/13 [==============================] - 0s 3ms/step - loss: 7.3848
Epoch 43/100
13/13 [==============================] - 0s 3ms/step - loss: 7.1203
Epoch 44/100
13/13 [==============================] - 0s 3ms/step - loss: 7.0398
Epoch 45/100
13/13 [==============================] - 0s 3ms/step - loss: 6.9539
Epoch 46/100
13/13 [==============================] - 0s 2ms/step - loss: 6.9556
Epoch 47/100
13/13 [==============================] - 0s 3ms/step - loss: 6.8988
Epoch 48/100
13/13 [==============================] - 0s 3ms/step - loss: 6.7077
Epoch 49/100
13/13 [==============================] - 0s 2ms/step - loss: 6.1695
Epoch 50/100
13/13 [==============================] - 0s 2ms/step - loss: 6.2582
Epoch 51/100
13/13 [==============================] - 0s 3ms/step - loss: 6.0867
Epoch 52/100
13/13 [==============================] - 0s 2ms/step - loss: 5.9881
Epoch 53/100
13/13 [==============================] - 0s 2ms/step - loss: 6.5553
Epoch 54/100
13/13 [==============================] - 0s 2ms/step - loss: 5.8091
Epoch 55/100
13/13 [==============================] - 0s 2ms/step - loss: 6.8235
Epoch 56/100
13/13 [==============================] - 0s 2ms/step - loss: 6.1424
Epoch 57/100
13/13 [==============================] - 0s 2ms/step - loss: 5.5099
Epoch 58/100
13/13 [==============================] - 0s 2ms/step - loss: 5.2211
Epoch 59/100
13/13 [==============================] - 0s 2ms/step - loss: 5.4964
Epoch 60/100
13/13 [==============================] - 0s 2ms/step - loss: 5.0268
Epoch 61/100
13/13 [==============================] - 0s 3ms/step - loss: 5.3282
Epoch 62/100
13/13 [==============================] - 0s 2ms/step - loss: 5.0867
```

```
Epoch 63/100
13/13 [==============================] - 0s 3ms/step - loss: 4.9664
Epoch 64/100
13/13 [==============================] - 0s 2ms/step - loss: 4.8192
Epoch 65/100
13/13 [==============================] - 0s 2ms/step - loss: 4.7862
Epoch 66/100
13/13 [==============================] - 0s 2ms/step - loss: 4.6972
Epoch 67/100
13/13 [==============================] - 0s 2ms/step - loss: 4.7165
Epoch 68/100
13/13 [==============================] - 0s 2ms/step - loss: 4.5970
Epoch 69/100
13/13 [==============================] - 0s 2ms/step - loss: 4.6533
Epoch 70/100
13/13 [==============================] - 0s 2ms/step - loss: 4.7719
Epoch 71/100
13/13 [==============================] - 0s 3ms/step - loss: 4.4936
Epoch 72/100
13/13 [==============================] - 0s 3ms/step - loss: 4.2616
Epoch 73/100
13/13 [==============================] - 0s 2ms/step - loss: 4.3233
Epoch 74/100
13/13 [==============================] - 0s 2ms/step - loss: 4.2909
Epoch 75/100
13/13 [==============================] - 0s 2ms/step - loss: 4.3337
Epoch 76/100
13/13 [==============================] - 0s 2ms/step - loss: 4.1199
Epoch 77/100
13/13 [==============================] - 0s 3ms/step - loss: 4.1012
Epoch 78/100
13/13 [==============================] - 0s 2ms/step - loss: 4.0924
Epoch 79/100
13/13 [==============================] - 0s 2ms/step - loss: 4.0687
Epoch 80/100
13/13 [==============================] - 0s 3ms/step - loss: 3.9866
Epoch 81/100
13/13 [==============================] - 0s 3ms/step - loss: 4.3584
Epoch 82/100
13/13 [==============================] - 0s 3ms/step - loss: 4.2379
Epoch 83/100
13/13 [==============================] - 0s 3ms/step - loss: 4.1398
Epoch 84/100
13/13 [==============================] - 0s 3ms/step - loss: 3.8676
Epoch 85/100
13/13 [==============================] - 0s 3ms/step - loss: 3.9280
Epoch 86/100
13/13 [==============================] - 0s 2ms/step - loss: 3.8878
Epoch 87/100
13/13 [==============================] - 0s 3ms/step - loss: 3.8788
Epoch 88/100
13/13 [==============================] - 0s 3ms/step - loss: 3.8821
Epoch 89/100
13/13 [==============================] - 0s 2ms/step - loss: 3.8107
Epoch 90/100
13/13 [==============================] - 0s 2ms/step - loss: 4.0224
Epoch 91/100
13/13 [==============================] - 0s 3ms/step - loss: 3.6596
Epoch 92/100
13/13 [==============================] - 0s 3ms/step - loss: 3.6219
Epoch 93/100
13/13 [==============================] - 0s 3ms/step - loss: 3.5863
Epoch 94/100
13/13 [==============================] - 0s 3ms/step - loss: 3.8640
Epoch 95/100
13/13 [==============================] - 0s 3ms/step - loss: 3.8445
Epoch 96/100
13/13 [==============================] - 0s 3ms/step - loss: 3.3634
Epoch 97/100
13/13 [==============================] - 0s 3ms/step - loss: 3.3942
Epoch 98/100
13/13 [==============================] - 0s 2ms/step - loss: 3.5340
Epoch 99/100
13/13 [==============================] - 0s 2ms/step - loss: 3.4620
Epoch 100/100
13/13 [==============================] - 0s 2ms/step - loss: 3.4567
```

Out[19]:

```
<keras.callbacks.History at 0x7f66ebcdc040>
```

In [20]:

```
y_pred = model.predict(X_test)
```

```
4/4 [==============================] - 0s 4ms/step
```

In [27]:

```
# Predicting RMSE the Test set results
from sklearn.metrics import mean_squared_error
rmse = (np.sqrt(mean_squared_error(y_test, y_pred)))
print(rmse)
```

```
3.1410293739926494
```

Assignment No. 2

- **Title:** Classification using Deep neural network

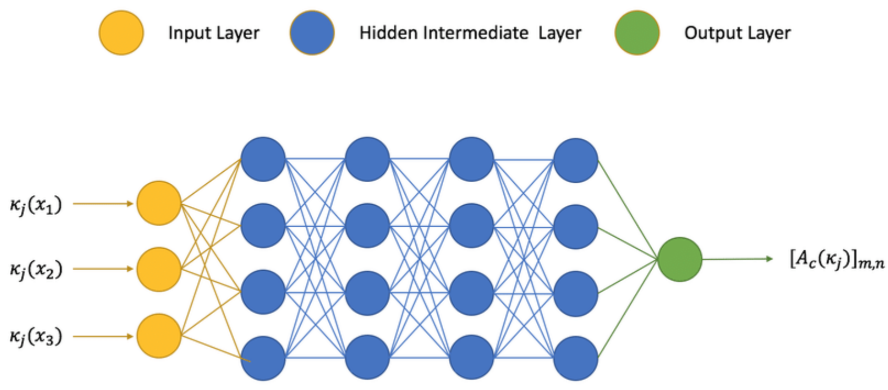
- **Objective:** To study and understand how to solve classification problem using deep neural network.

- **Problem statement:** Binary classification using Deep Neural Networks Example: Classify movie reviews into positive" reviews and "negative" reviews, just based on the text content of the reviews. Use IMDB dataset.

- **Theory:**

Deep learning (DL) is a subfield of machine learning based on learning multiple levels of representations by making a hierarchy of features where the higher levels are defined from the lower levels and the same lower level features can help in defining many higher level features. DL structure extends the traditional neural networks by adding more hidden layers to the network architecture between the input and output layers to model more complex and nonlinear relationships.

Deep Neural Network is another DL architecture that is widely used for classification or regression with success in many areas. It's a typical feedforward network which the input flows from the input layer to the output layer through number of hidden layers which are more than two layers.



The main purpose of a neural network is to receive a set of inputs, perform progressively complex calculations on them, and give output to solve real world problems like classification.

The IMDB sentiment classification dataset consists of 50,000 movie reviews from IMDB users that are labeled as either positive (1) or negative (0). The reviews are preprocessed and each one is encoded as a sequence of word indexes in the form of integers. The words within the reviews are indexed by their overall frequency within the dataset.

- **Code and Output:**

- **Conclusion:** We have successfully classified movie reviews using deep neural networks.

In [34]:

```
from keras.datasets import imdb

# Load the data, keeping only 10,000 of the most frequently occuring words
(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words = 10000)
```

In [35]:

```
# Here is a list of maximum indexes in every review
print(type([max(sequence) for sequence in train_data]))

# Find the maximum of all max indexes
max([max(sequence) for sequence in train_data])
```

```
<class 'list'>
```

Out[35]:

```
9999
```

In [36]:

```
# step 1: load the dictionary mappings from word to integer index
word_index = imdb.get_word_index()

# step 2: reverse word index to map integer indexes to their respective words
reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])

# Step 3: decode the review, mapping integer indices to words
#
# indices are off by 3 because 0, 1, and 2 are reserverd indices for "padding", "Start of sequence" and "
unknown"
decoded_review = ' '.join([reverse_word_index.get(i-3, '?') for i in train_data[0]])

decoded_review
```

Out[36]:

```
"? this film was just brilliant casting location scenery story direction everyone's really suited the par
t they played and you could just imagine being there robert ? is an amazing actor and now the same being
director ? father came from the same scottish island as myself so i loved the fact there was a real conne
ction with this film the witty remarks throughout the film were great it was just brilliant so much that
i bought the film as soon as it was released for ? and would recommend it to everyone to watch and the fl
y fishing was amazing really cried at the end it was so sad and you know what they say if you cry at a fil
m it must have been good and this definitely was also ? to the two little boy's that played the ? of norm
an and paul they were just brilliant children are often left out of the ? list i think because the stars
that play them all grown up are such a big profile for the whole film but these children are amazing and s
hould be praised for what they have done don't you think the whole story was so lovely because it was tru
e and was someone's life after all that was shared with us all"
```

In [37]:

```
# Vectorize input data

import numpy as np

def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))    # Creates an all zero matrix of shape (len(sequen
ces),10K)
    for i,sequence in enumerate(sequences):
        results[i,sequence] = 1                        # Sets specific indices of results[i] to 1s
    return results

# Vectorize training Data
X_train = vectorize_sequences(train_data)

# Vectorize testing Data
X_test = vectorize_sequences(test_data)
```

In [38]:

```
X_train[0]
```

Out[38]:

```
array([0., 1., 1., ..., 0., 0., 0.])
```

In [39]:

```
X_train.shape
```

Out[39]:

```
(25000, 10000)
```

In [40]:

```
# Vectorize labels
y_train = np.asarray(train_labels).astype('float32')
y_test  = np.asarray(test_labels).astype('float32')
```

In [41]:

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

In [42]:

```
from keras import optimizers
from keras import losses
from keras import metrics

model.compile(optimizer=optimizers.RMSprop(lr=0.001), loss = losses.binary_crossentropy, metrics = [metr
ics.binary_accuracy])
```

```
/usr/local/lib/python3.8/dist-packages/keras/optimizers/optimizer_v2/rmsprop.py:135: UserWarning: The `lr
` argument is deprecated, use `learning_rate` instead.
  super(RMSprop, self).__init__(name, **kwargs)
```

In [43]:

```
# Input for Validation
X_val = X_train[:10000]
partial_X_train = X_train[10000:]

# Labels for validation
y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

In [44]:

```
history = model.fit(partial_X_train, partial_y_train, epochs=20, batch_size=512, validation_data=(X_val,
y_val))
```

```
Epoch 1/20
30/30 [==============================] - 2s 56ms/step - loss: 0.4878 - binary_accuracy: 0.7969 - val_loss
: 0.3705 - val_binary_accuracy: 0.8656
Epoch 2/20
30/30 [==============================] - 1s 45ms/step - loss: 0.2905 - binary_accuracy: 0.9047 - val_loss
: 0.3232 - val_binary_accuracy: 0.8687
Epoch 3/20
30/30 [==============================] - 2s 54ms/step - loss: 0.2151 - binary_accuracy: 0.9300 - val_loss
: 0.2884 - val_binary_accuracy: 0.8847
Epoch 4/20
30/30 [==============================] - 1s 42ms/step - loss: 0.1678 - binary_accuracy: 0.9447 - val_loss
: 0.2873 - val_binary_accuracy: 0.8839
Epoch 5/20
30/30 [==============================] - 2s 61ms/step - loss: 0.1391 - binary_accuracy: 0.9546 - val_loss
: 0.2910 - val_binary_accuracy: 0.8835
Epoch 6/20
30/30 [==============================] - 2s 63ms/step - loss: 0.1157 - binary_accuracy: 0.9634 - val_loss
: 0.2987 - val_binary_accuracy: 0.8840
Epoch 7/20
30/30 [==============================] - 1s 41ms/step - loss: 0.0975 - binary_accuracy: 0.9701 - val_loss
: 0.3287 - val_binary_accuracy: 0.8774
Epoch 8/20
30/30 [==============================] - 1s 34ms/step - loss: 0.0750 - binary_accuracy: 0.9795 - val_loss
: 0.3497 - val_binary_accuracy: 0.8790
Epoch 9/20
30/30 [==============================] - 1s 35ms/step - loss: 0.0700 - binary_accuracy: 0.9807 - val_loss
: 0.3777 - val_binary_accuracy: 0.8768
Epoch 10/20
30/30 [==============================] - 1s 34ms/step - loss: 0.0592 - binary_accuracy: 0.9826 - val_loss
: 0.3836 - val_binary_accuracy: 0.8769
Epoch 11/20
30/30 [==============================] - 1s 35ms/step - loss: 0.0467 - binary_accuracy: 0.9881 - val_loss
: 0.4225 - val_binary_accuracy: 0.8697
Epoch 12/20
```

```
30/30 [==============================] - 1s 35ms/step - loss: 0.0396 - binary_accuracy: 0.9909 - val_loss
: 0.4334 - val_binary_accuracy: 0.8740
Epoch 13/20
30/30 [==============================] - 1s 33ms/step - loss: 0.0306 - binary_accuracy: 0.9935 - val_loss
: 0.4850 - val_binary_accuracy: 0.8631
Epoch 14/20
30/30 [==============================] - 1s 33ms/step - loss: 0.0266 - binary_accuracy: 0.9948 - val_loss
: 0.4939 - val_binary_accuracy: 0.8720
Epoch 15/20
30/30 [==============================] - 1s 35ms/step - loss: 0.0208 - binary_accuracy: 0.9965 - val_loss
: 0.5268 - val_binary_accuracy: 0.8700
Epoch 16/20
30/30 [==============================] - 1s 34ms/step - loss: 0.0172 - binary_accuracy: 0.9977 - val_loss
: 0.5638 - val_binary_accuracy: 0.8683
Epoch 17/20
30/30 [==============================] - 1s 48ms/step - loss: 0.0141 - binary_accuracy: 0.9980 - val_loss
: 0.5940 - val_binary_accuracy: 0.8655
Epoch 18/20
30/30 [==============================] - 2s 52ms/step - loss: 0.0117 - binary_accuracy: 0.9983 - val_loss
: 0.6302 - val_binary_accuracy: 0.8639
Epoch 19/20
30/30 [==============================] - 1s 40ms/step - loss: 0.0102 - binary_accuracy: 0.9985 - val_loss
: 0.6597 - val_binary_accuracy: 0.8657
Epoch 20/20
30/30 [==============================] - 1s 34ms/step - loss: 0.0080 - binary_accuracy: 0.9989 - val_loss
: 0.6970 - val_binary_accuracy: 0.8651
```

In [45]:

```
history_dict = history.history
history_dict.keys()
```

Out[45]:

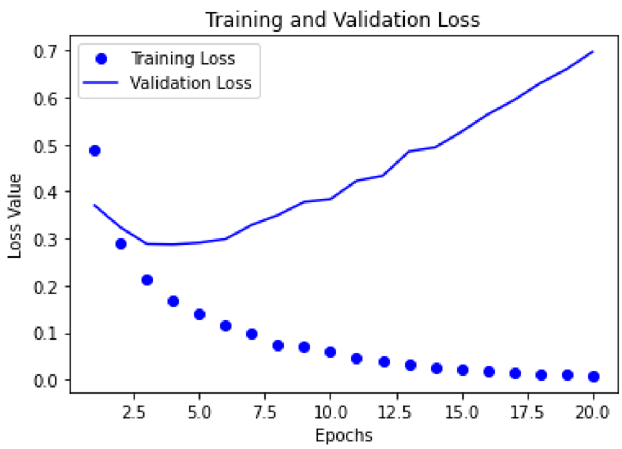
```
dict_keys(['loss', 'binary_accuracy', 'val_loss', 'val_binary_accuracy'])
```

In [46]:

```
import matplotlib.pyplot as plt
%matplotlib inline
```

In [47]:

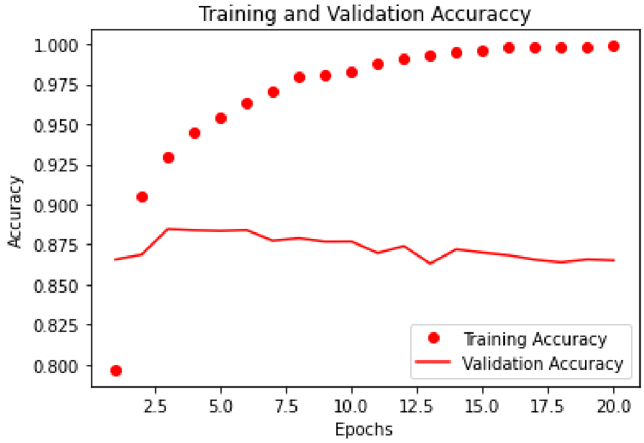
```
# Plotting losses
loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss']

epochs = range(1, len(loss_values) + 1)

plt.plot(epochs, loss_values, 'bo', label="Training Loss")
plt.plot(epochs, val_loss_values, 'b', label="Validation Loss")

plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss Value')
plt.legend()

plt.show()
```



In [48]:

```
# Training and Validation Accuracy

acc_values = history_dict['binary_accuracy']
```

```python
val_acc_values = history_dict['val_binary_accuracy']

epochs = range(1, len(loss_values) + 1)

plt.plot(epochs, acc_values, 'ro', label="Training Accuracy")
plt.plot(epochs, val_acc_values, 'r', label="Validation Accuracy")

plt.title('Training and Validation Accuraccy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```



In [49]:

```python
model.fit(partial_X_train, partial_y_train, epochs=3, batch_size=512, validation_data=(X_val, y_val))
```

```
Epoch 1/3
30/30 [==============================] - 2s 61ms/step - loss: 0.0040 - binary_accuracy: 0.9999 - val_loss
: 0.7310 - val_binary_accuracy: 0.8648
Epoch 2/3
30/30 [==============================] - 1s 33ms/step - loss: 0.0061 - binary_accuracy: 0.9988 - val_loss
: 0.7557 - val_binary_accuracy: 0.8636
Epoch 3/3
30/30 [==============================] - 1s 34ms/step - loss: 0.0039 - binary_accuracy: 0.9995 - val_loss
: 0.8755 - val_binary_accuracy: 0.8475
```

Out[49]:

```
<keras.callbacks.History at 0x7f1cc77b7880>
```

In [50]:

```python
# Making Predictions for testing data
np.set_printoptions(suppress=True)
result = model.predict(X_test)
```

```
782/782 [==============================] - 2s 2ms/step
```

In [51]:

```python
result
```

Out[51]:

```
array([[0.00190504],
       [1.        ],
       [0.09390965],
       ...,
       [0.00027408],
       [0.00282606],
       [0.3104798 ]], dtype=float32)
```

In [52]:

```python
y_pred = np.zeros(len(result))
for i, score in enumerate(result):
    y_pred[i] = 1 if score > 0.5 else 0
```

In [53]:

```python
from sklearn.metrics import mean_absolute_error
mae = mean_absolute_error(y_pred, y_test)
```

In [54]:

```
# error
mae
```

Out[54]:

```
0.16668
```

Assignment No. 3

- **Title:** Convolutional neural network (CNN)

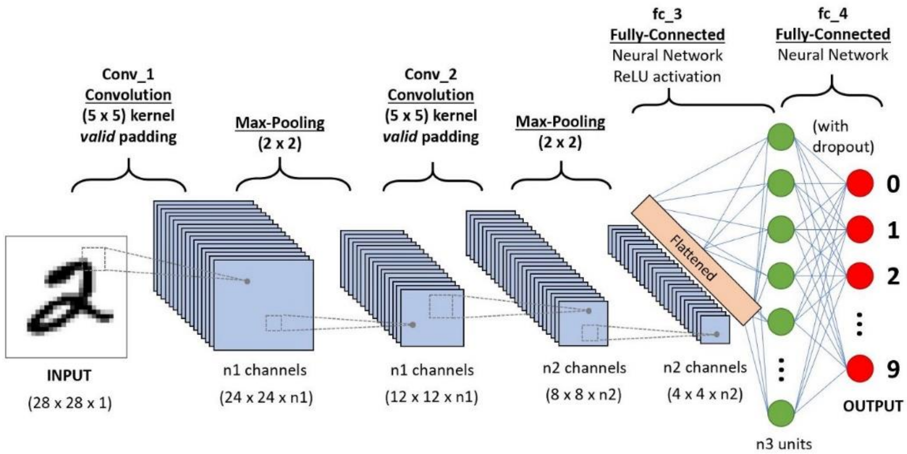
- **Objective:** Study and understand Conolutional Neural Network by creating a classifier.

- **Problem statement:** Use MNIST Fashion Dataset and create a classifier to classify fashion clothing into categories.

- **Theory:**

  Convolutional neural networks are distinguished from other neural networks by their superior performance with image, speech, or audio signal inputs. They have three main types of layers, which are:

  - Convolutional layer
  - Pooling layer
  - Fully-connected (FC) layer



The convolutional layer is the core building block of a CNN, and it is where the majority of computation occurs. It requires a few components, which are input data, a filter, and a feature map. Let's assume that the input will be a color image, which is made up of a matrix of pixels in 3D. This means that the input will have three dimensions—a height, width, and depth—which correspond to RGB in an image. We also have a feature detector, also known as a kernel or a filter, which will move across the receptive fields of the image, checking if the feature is present. This process is known as a convolution. After each convolution operation, a CNN applies a Rectified Linear Unit (ReLU) transformation to the feature map, introducing nonlinearity to the model.

Pooling layers, also known as down sampling, conducts dimensionality reduction, reducing the number of parameters in the input. There are two main types of pooling:

- **Max pooling:** As the filter moves across the input, it selects the pixel with the maximum value to send to the output array. As an aside, this approach tends to be used more often compared to average pooling.

- **Average pooling:** As the filter moves across the input, it calculates the average value within the receptive field to send to the output array.

The name of the full-connected layer aptly describes itself. The pixel values of the input image are not directly connected to the output layer in partially connected layers. However, in the fully-connected layer, each node in the output layer connects directly to a node in the previous layer. This layer performs the task of classification based on the features extracted through the previous layers and their different filters. While convolutional and pooling layers tend to use ReLu functions, FC layers usually leverage a softmax activation function to classify inputs appropriately, producing a probability from 0 to 1.

- **Code and Output:**

- **Conclusion:** We have successfully created a classifier using convoluted neural network.

In [25]:

```
from __future__ import absolute_import, division, print_function

# TensorFlow and tf.keras
import tensorflow as tf
from tensorflow import keras

# Helper libraries
import numpy as np
import matplotlib.pyplot as plt
```

In [26]:

```
fashion_mnist = keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
```

In [27]:

```
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

In [28]:

```
train_images.shape
```

Out[28]:

```
(60000, 28, 28)
```

In [29]:

```
len(train_labels)
```

Out[29]:

```
60000
```

In [30]:

```
train_labels
```

Out[30]:

```
array([9, 0, 0, ..., 3, 0, 5], dtype=uint8)
```

In [31]:

```
test_images.shape
```

Out[31]:

```
(10000, 28, 28)
```

In [32]:

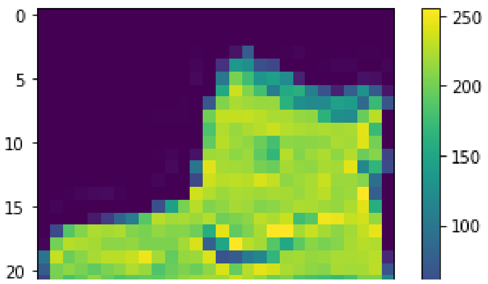
```
len(test_labels)
```

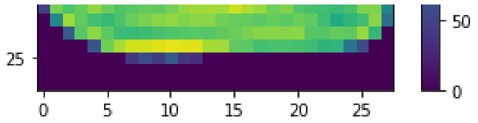
Out[32]:

```
10000
```

In [33]:

```
plt.figure()
plt.imshow(train_images[0])
plt.colorbar()
plt.grid(False)
plt.show()
```





In [34]:

```
train_images = train_images / 255.0

test_images = test_images / 255.0
```

In [35]:

```
plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])
plt.show()
```

In [36]:

```
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(10, activation=tf.nn.softmax)
])
```

In [37]:

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

In [38]:

```
model.fit(train_images, train_labels, epochs=5)
```

```
Epoch 1/5
1875/1875 [==============================] - 7s 4ms/step - loss: 0.4971 - accuracy: 0.8257
Epoch 2/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.3751 - accuracy: 0.8648
Epoch 3/5
```

```
1875/1875 [==============================] - 7s 4ms/step - loss: 0.3338 - accuracy: 0.8780
Epoch 4/5
1875/1875 [==============================] - 7s 3ms/step - loss: 0.3104 - accuracy: 0.8856
Epoch 5/5
1875/1875 [==============================] - 8s 4ms/step - loss: 0.2946 - accuracy: 0.8906
```

Out[38]:

```
<keras.callbacks.History at 0x7f07800931f0>
```

In [39]:

```
test_loss, test_acc = model.evaluate(test_images, test_labels)

print('Test accuracy:', test_acc)
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.3542 - accuracy: 0.8736
Test accuracy: 0.8736000061035156
```

In [40]:

```
predictions = model.predict(test_images)
```

```
313/313 [==============================] - 1s 2ms/step
```

In [41]:

```
predictions[0]
```

Out[41]:

```
array([4.9921914e-06, 3.3841974e-07, 1.8965457e-07, 1.1019566e-09,
       1.3401749e-06, 4.8021390e-03, 1.4439345e-06, 5.2010346e-02,
       2.6271462e-05, 9.4315302e-01], dtype=float32)
```

In [42]:

```
np.argmax(predictions[0])
```

Out[42]:

```
9
```

In [43]:

```
test_labels[0]
```

Out[43]:

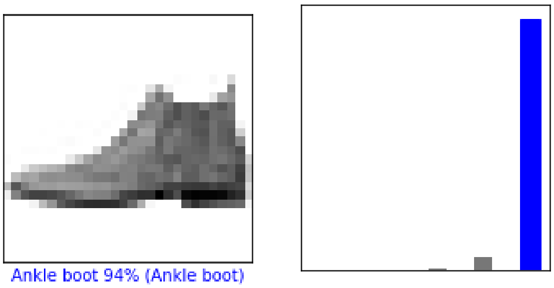
```
9
```

In [44]:

```
def plot_image(i, predictions_array, true_label, img):
  predictions_array, true_label, img = predictions_array[i], true_label[i], img[i]
  plt.grid(False)
  plt.xticks([])
  plt.yticks([])

  plt.imshow(img, cmap=plt.cm.binary)

  predicted_label = np.argmax(predictions_array)
  if predicted_label == true_label:
    color = 'blue'
  else:
    color = 'red'

  plt.xlabel("{} {:2.0f}% ({})".format(class_names[predicted_label],
                                100*np.max(predictions_array),
                                class_names[true_label]),
                                color=color)

def plot_value_array(i, predictions_array, true_label):
  predictions_array, true_label = predictions_array[i], true_label[i]
  plt.grid(False)
  plt.xticks([])
  plt.yticks([])
  thisplot = plt.bar(range(10), predictions_array, color="#777777")
  plt.ylim([0, 1])
  predicted_label = np.argmax(predictions_array)

  thisplot[predicted_label].set_color('red')
  thisplot[true_label].set_color('blue')
```

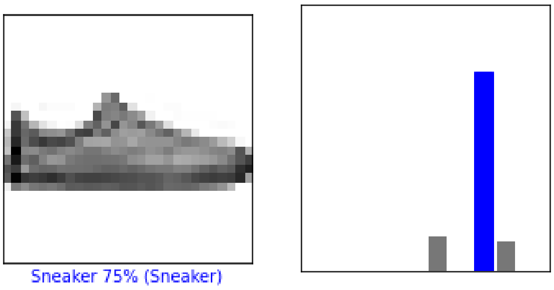
In [45]:

```
i = 0
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions, test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions,  test_labels)
plt.show()
```



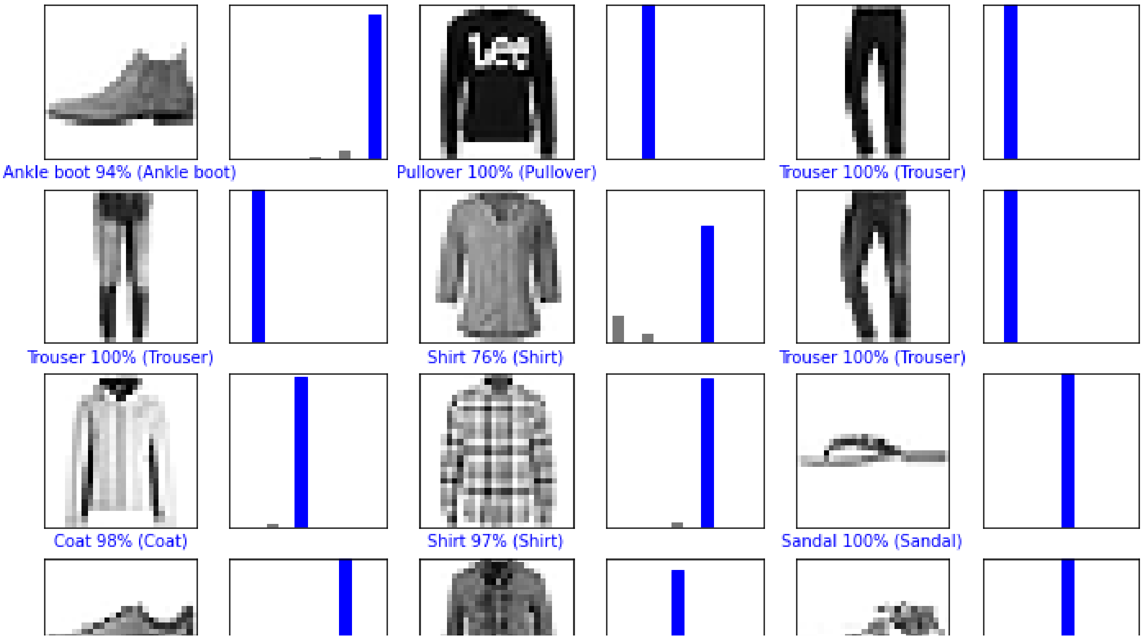
In [46]:

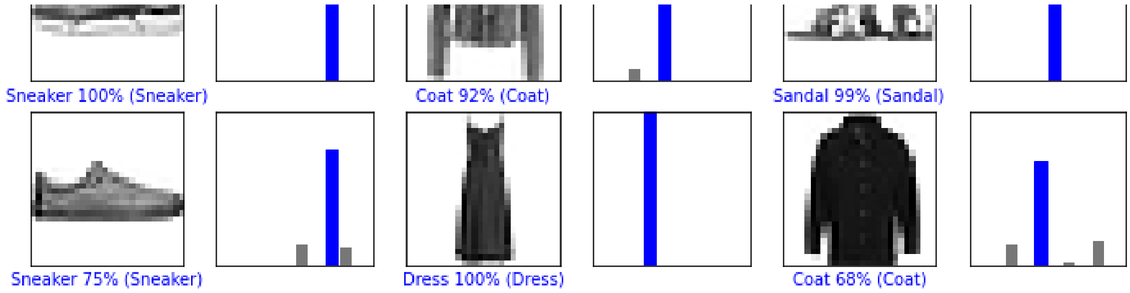
```
i = 12
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions, test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions,  test_labels)
plt.show()
```



In [47]:

```
# Plot the first X test images, their predicted label, and the true label
# Color correct predictions in blue, incorrect predictions in red
num_rows = 5
num_cols = 3
num_images = num_rows*num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
for i in range(num_images):
  plt.subplot(num_rows, 2*num_cols, 2*i+1)
  plot_image(i, predictions, test_labels, test_images)
  plt.subplot(num_rows, 2*num_cols, 2*i+2)
  plot_value_array(i, predictions, test_labels)
plt.show()
```





In [48]:

```
# Grab an image from the test dataset
img = test_images[0]

print(img.shape)
```

```
(28, 28)
```

In [49]:

```
# Add the image to a batch where it's the only member.
img = (np.expand_dims(img,0))

print(img.shape)
```

```
(1, 28, 28)
```

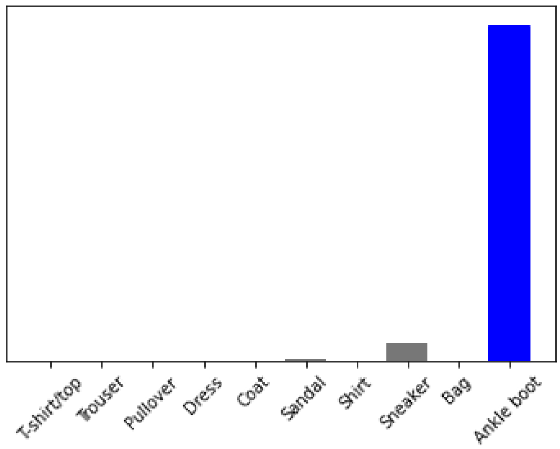
In [50]:

```
predictions_single = model.predict(img)

print(predictions_single)
```

```
1/1 [==============================] - 0s 25ms/step
[[4.9921905e-06 3.3841968e-07 1.8965453e-07 1.1019544e-09 1.3401745e-06
  4.8021362e-03 1.4439315e-06 5.2010350e-02 2.6271480e-05 9.4315284e-01]]
```

In [51]:

```
plot_value_array(0, predictions_single, test_labels)
_ = plt.xticks(range(10), class_names, rotation=45)
```



In [52]:

```
np.argmax(predictions_single[0])
```

Out[52]:

```
9
```

Assignment No. 4

- **Title:** Recurrent neural network (RNN)

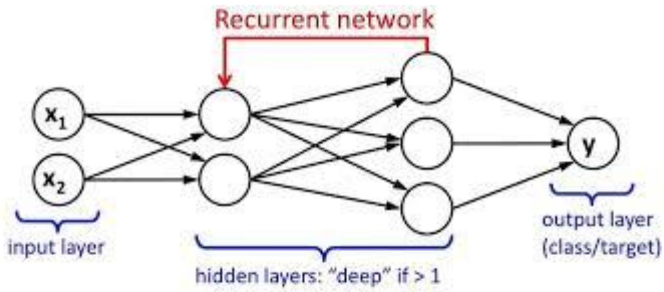
- **Objective:** To study and understand Recurrent Neural Network by doing analysis and designing prediction system.

- **Problem statement:** Use the Google stock prices dataset and design a time series analysis and prediction system using RNN.

- **Theory:**

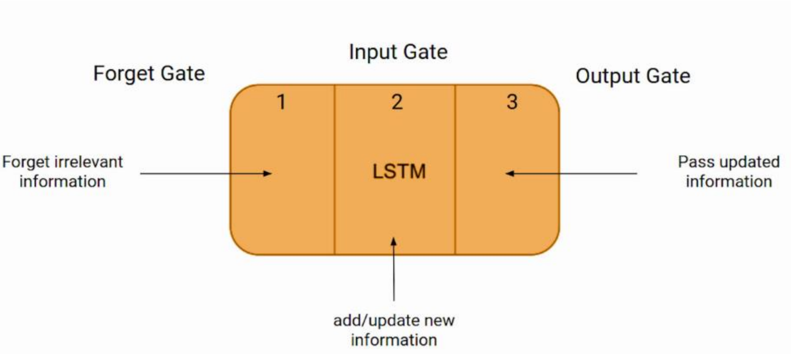
A recurrent neural network (RNN) is a type of artificial neural network which uses sequential data or time series data. These deep learning algorithms are commonly used for ordinal or temporal problems, such as language translation, natural language processing (NLP), speech recognition, and image captioning; they are incorporated into popular applications such as Siri, voice search, and Google Translate. Like feedforward and convolutional neural networks (CNNs), recurrent neural networks utilize training data to learn. They are distinguished by their "memory" as they take information from prior inputs to influence the current input and output. While traditional deep neural networks assume that inputs and outputs are independent of each other, the output of recurrent neural networks depend on the prior elements within the sequence. While future events would also be helpful in determining the output of a given sequence, unidirectional recurrent neural networks cannot account for these events in their predictions.

Let's take an idiom, such as "feeling under the weather", which is commonly used when someone is ill, to aid us in the explanation of RNNs. In order for the idiom to make sense, it needs to be expressed in that specific order. As a result, recurrent networks need to account for the position of each word in the idiom and they use that information to predict the next word in the sequence.



Long short-term memory (LSTM)

This is a popular RNN architecture as a solution to vanishing gradient problem. It addresses the problem of long-term dependencies. That is, if the previous state that is influencing the current prediction is not in the recent past, the RNN model may not be able to accurately predict the current state. LSTMs have "cells" in the hidden layers of the neural network, which have three gates–an input gate, an output gate, and a forget gate. These gates control the flow of information which is needed to predict the output in the network.



- **Code and Output:**

- **Conclusion:** We have successfully implemented a recurrent neural network to create a classifier.

In [1]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import Dense,LSTM,Dropout
```

In [2]:

```
data = pd.read_csv('Google_train_data.csv')
data.head()
```

Out[2]:

	Date	Open	High	Low	Close	Volume
0	1/3/2012	325.25	332.83	324.97	663.59	7,380,500
1	1/4/2012	331.27	333.87	329.08	666.45	5,749,400
2	1/5/2012	329.83	330.75	326.89	657.21	6,590,300
3	1/6/2012	328.34	328.77	323.68	648.24	5,405,900
4	1/9/2012	322.04	322.29	309.46	620.76	11,688,800

In [3]:

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1258 entries, 0 to 1257
Data columns (total 6 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Date    1258 non-null   object
 1   Open    1258 non-null   float64
 2   High    1258 non-null   float64
 3   Low     1258 non-null   float64
 4   Close   1258 non-null   object
 5   Volume  1258 non-null   object
dtypes: float64(3), object(3)
memory usage: 59.1+ KB
```

In [4]:

```
data["Close"]=pd.to_numeric(data.Close,errors='coerce')
data = data.dropna()
trainData = data.iloc[:,4:5].values
```

In [5]:

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1149 entries, 0 to 1257
Data columns (total 6 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Date    1149 non-null   object
 1   Open    1149 non-null   float64
 2   High    1149 non-null   float64
 3   Low     1149 non-null   float64
 4   Close   1149 non-null   float64
 5   Volume  1149 non-null   object
dtypes: float64(4), object(2)
memory usage: 62.8+ KB
```

In [6]:

```
sc = MinMaxScaler(feature_range=(0,1))
trainData = sc.fit_transform(trainData)
trainData.shape
```

Out[6]:

```
(1149, 1)
```

In [7]:

```
X_train = []
y_train = []

for i in range (60,1149): #60 : timestep // 1149 : length of the data
    X_train.append(trainData[i-60:i,0])
    y_train.append(trainData[i,0])

X_train,y_train = np.array(X_train),np.array(y_train)
```

In [8]:

```
X_train = np.reshape(X_train,(X_train.shape[0],X_train.shape[1],1)) #adding the batch_size axis
X_train.shape
```

Out[8]:

```
(1089, 60, 1)
```

In [9]:

```
model = Sequential()

model.add(LSTM(units=100, return_sequences = True, input_shape =(X_train.shape[1],1)))
model.add(Dropout(0.2))

model.add(LSTM(units=100, return_sequences = True))
model.add(Dropout(0.2))

model.add(LSTM(units=100, return_sequences = True))
model.add(Dropout(0.2))

model.add(LSTM(units=100, return_sequences = False))
model.add(Dropout(0.2))

model.add(Dense(units =1))
model.compile(optimizer='adam',loss="mean_squared_error")
```

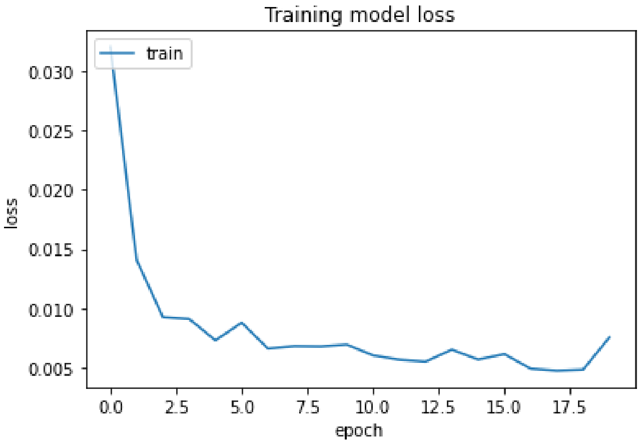
In [10]:

```
hist = model.fit(X_train, y_train, epochs = 20, batch_size = 32, verbose=2)
```

```
Epoch 1/20
35/35 - 16s - loss: 0.0320 - 16s/epoch - 459ms/step
Epoch 2/20
35/35 - 12s - loss: 0.0141 - 12s/epoch - 335ms/step
Epoch 3/20
35/35 - 9s - loss: 0.0093 - 9s/epoch - 244ms/step
Epoch 4/20
35/35 - 7s - loss: 0.0091 - 7s/epoch - 214ms/step
Epoch 5/20
35/35 - 10s - loss: 0.0073 - 10s/epoch - 285ms/step
Epoch 6/20
35/35 - 9s - loss: 0.0088 - 9s/epoch - 250ms/step
Epoch 7/20
35/35 - 7s - loss: 0.0066 - 7s/epoch - 212ms/step
Epoch 8/20
35/35 - 9s - loss: 0.0068 - 9s/epoch - 243ms/step
Epoch 9/20
35/35 - 9s - loss: 0.0068 - 9s/epoch - 253ms/step
Epoch 10/20
35/35 - 7s - loss: 0.0069 - 7s/epoch - 213ms/step
Epoch 11/20
35/35 - 9s - loss: 0.0060 - 9s/epoch - 247ms/step
Epoch 12/20
35/35 - 10s - loss: 0.0057 - 10s/epoch - 284ms/step
Epoch 13/20
35/35 - 7s - loss: 0.0055 - 7s/epoch - 214ms/step
Epoch 14/20
35/35 - 9s - loss: 0.0065 - 9s/epoch - 252ms/step
Epoch 15/20
35/35 - 8s - loss: 0.0057 - 8s/epoch - 242ms/step
Epoch 16/20
35/35 - 7s - loss: 0.0061 - 7s/epoch - 210ms/step
Epoch 17/20
35/35 - 9s - loss: 0.0049 - 9s/epoch - 243ms/step
Epoch 18/20
35/35 - 10s - loss: 0.0047 - 10s/epoch - 282ms/step
Epoch 19/20
35/35 - 7s - loss: 0.0048 - 7s/epoch - 211ms/step
Epoch 20/20
35/35 - 8s - loss: 0.0076 - 8s/epoch - 243ms/step
```

In [11]:

```
plt.plot(hist.history['loss'])
plt.title('Training model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train'], loc='upper left')
plt.show()
```



In [12]:

```
testData = pd.read_csv('Google_test_data.csv')
testData["Close"]=pd.to_numeric(testData.Close,errors='coerce')
testData = testData.dropna()
testData = testData.iloc[:,4:5]
y_test = testData.iloc[60:,0:].values
#input array for the model
inputClosing = testData.iloc[:,0:].values
inputClosing_scaled = sc.transform(inputClosing)
inputClosing_scaled.shape
X_test = []
length = len(testData)
timestep = 60
for i in range(timestep,length):
    X_test.append(inputClosing_scaled[i-timestep:i,0])
X_test = np.array(X_test)
X_test = np.reshape(X_test,(X_test.shape[0],X_test.shape[1],1))
X_test.shape
```

Out[12]:

```
(192, 60, 1)
```

In [13]:

```
y_pred = model.predict(X_test)
y_pred
```

```
6/6 [==============================] - 2s 72ms/step
```

Out[13]:

```
array([[1.16179  ],
       [1.1631088],
       [1.1724331],
       [1.1868168],
       [1.1983689],
       [1.1982452],
       [1.1870549],
       [1.1716752],
       [1.1618347],
       [1.1591649],
       [1.1531832],
       [1.1432477],
       [1.1342392],
       [1.1257831],
       [1.1235965],
       [1.127025 ],
       [1.142286 ],
       [1.1664964],
       [1.1942145],
       [1.2218447],
       [1.2332776],
       [1.2312684],
       [1.2147205],
       [1.1905136],
```

```
[1.1684806],
[1.1558316],
[1.1526709],
[1.1519624],
[1.1453004],
[1.1353829],
[1.1244267],
[1.1120113],
[1.0936927],
[1.071203 ],
[1.0618066],
[1.0695277],
[1.0879493],
[1.1089689],
[1.1289511],
[1.1389962],
[1.1495905],
[1.1627021],
[1.1780866],
[1.1925853],
[1.2027087],
[1.2047343],
[1.1971886],
[1.1904229],
[1.1890376],
[1.1945436],
[1.2060889],
[1.2140377],
[1.2166203],
[1.2145796],
[1.213667 ],
[1.2109393],
[1.2041996],
[1.2009138],
[1.2086142],
[1.2239051],
[1.2459278],
[1.2723885],
[1.2911754],
[1.2952675],
[1.2863945],
[1.2705956],
[1.259561 ],
[1.2560648],
[1.2588812],
[1.2632502],
[1.2677509],
[1.2704453],
[1.2674361],
[1.2628492],
[1.2588946],
[1.257737 ],
[1.2593839],
[1.2635039],
[1.2732879],
[1.2881215],
[1.3073635],
[1.3259524],
[1.3369976],
[1.3394896],
[1.3432838],
[1.3541006],
[1.3695042],
[1.3831407],
[1.3916621],
[1.3947477],
[1.3971055],
[1.405585 ],
[1.4202806],
[1.4308047],
[1.4314191],
[1.4221445],
[1.4051017],
[1.3847787],
[1.3672425],
[1.3602483],
[1.3645082],
[1.3769275],
[1.3942529],
[1.4098742],
[1.4199001],
[1.4225647],
[1.4208788],
[1.4179636],
```

```
[1.4179636],
[1.4184278],
[1.4228212],
[1.430534 ],
[1.4411852],
[1.452102 ],
[1.4638307],
[1.4781822],
[1.4905659],
[1.5004128],
[1.5090445],
[1.5194818],
[1.5067695],
[1.4701803],
[1.4219117],
[1.3810912],
[1.3563739],
[1.3441346],
[1.3388975],
[1.3368089],
[1.3368678],
[1.331129 ],
[1.3178322],
[1.3110298],
[1.3169495],
[1.3284833],
[1.334781 ],
[1.3362976],
[1.3352803],
[1.3310721],
[1.3234364],
[1.3145592],
[1.302618 ],
[1.2899239],
[1.2758108],
[1.2473798],
[1.2128351],
[1.178672 ],
[1.1512984],
[1.1383212],
[1.142172 ],
[1.1575152],
[1.1768603],
[1.1970001],
[1.2139362],
[1.2271951],
[1.2387699],
[1.248167 ],
[1.2566313],
[1.2659454],
[1.2736806],
[1.2721103],
[1.2606483],
[1.2429825],
[1.2254122],
[1.2154773],
[1.2169083],
[1.2290485],
[1.248789 ],
[1.2669588],
[1.2810075],
[1.2934813],
[1.3050567],
[1.3148568],
[1.3230573],
[1.3298122],
[1.3330575],
[1.3328451],
[1.3266706],
[1.3183285],
[1.3124657],
[1.3083488],
[1.3043047],
[1.3266984],
[1.3688518],
[1.4127072],
[1.4445978],
[1.4592773],
[1.4563084],
[1.4326231],
[1.4018185],
[1.3746797],
[1.363181 ],
[1.3625406],
```

```
       [1.3625406]],
       [1.3645191]], dtype=float32)
```

In [14]:

```
predicted_price = sc.inverse_transform(y_pred)
```

In [15]:

```
plt.plot(y_test, color = 'red', label = 'Actual Stock Price')
plt.plot(predicted_price, color = 'green', label = 'Predicted Stock Price')
plt.title('Google stock price prediction')
plt.xlabel('Time')
plt.ylabel('Stock Price')
plt.legend()
plt.show()
```