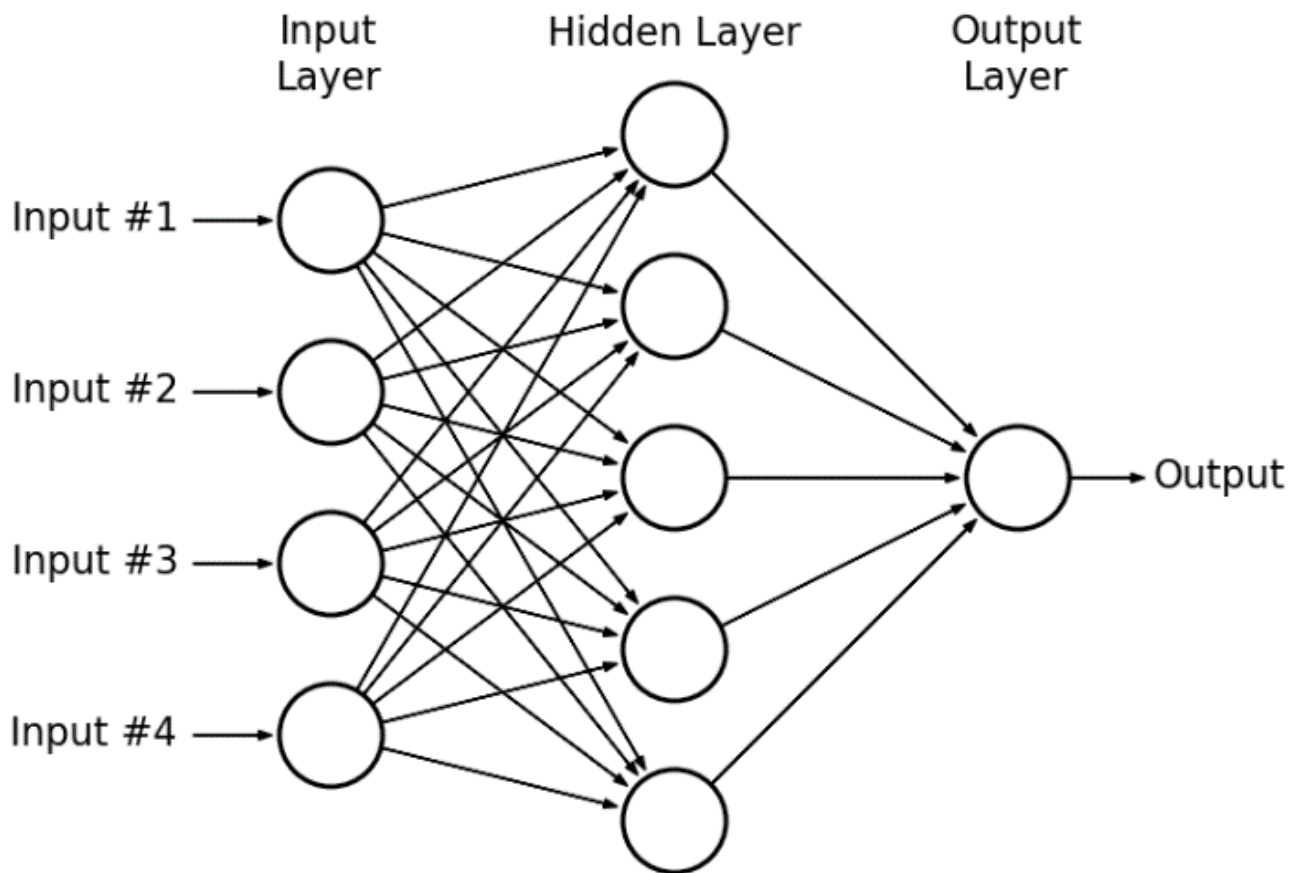


What is Neural Network

They are multi layer perceptrons. By stacking many linear units we get neural network.



Why are Neural Networks popular

Neural Networks are remarkably good at figuring out functions from X to Y.



In general all input features are connected to hidden units and NN's are capable of drawing hidden features out of them.

The above neural network is called 2 layer neural network, We don't count Input layer.

Computation of NN

Computation of NN is done by forward propagation for computing outputs(price in above NN) and Backward pass for computing gradients.

What if I remove hidden layer in 2 layer network?

Forward propagation:

$$\mathbf{Z} = \mathbf{W}^T \mathbf{x} + \mathbf{b}$$

Here Z is the weighted sum of inputs with the inclusion of bias

$$\hat{\mathbf{y}} = \mathbf{a}(\text{For simplification}) = \text{activation_function}(\mathbf{z})$$

Predicted Output is activation function applied on weighted sum(Z)

Activation Functions: The following activation functions helps in transforming linear inputs to nonlinear outputs. If we apply linear activation function we will get linear seperable line for classifying the outputs.

1. *Sigmoid:*

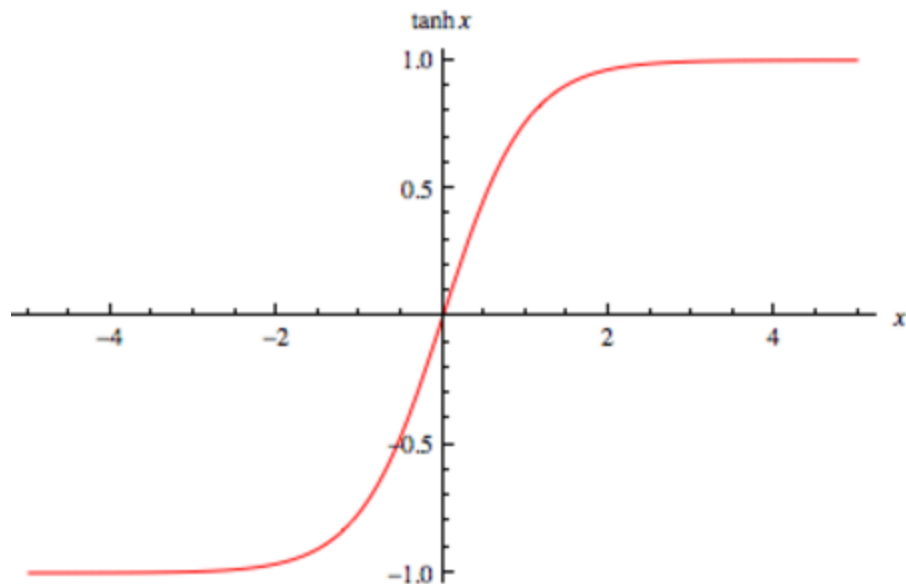
$$\text{Sigmoid}(x) = 1/(1+\exp(-x))$$



The main reason why we use sigmoid function is because it exists between (0 to 1). Therefore, it is especially used for models where we have to predict the probability as an output. Since probability of anything exists only between the range of 0 and 1, sigmoid is the right choice.

1. *Tanh:*

$$\text{Tanh}(x) = (\exp(x) - \exp(-x)) / (\exp(x) + \exp(-x))$$



The advantage is that the negative inputs will be mapped strongly negative and the zero inputs will be mapped near zero in the tanh graph.

1. *Relu:*

$$\text{Relu}(x) = \max(0, x)$$



The ReLU is the most used activation function in the world right now. Since, it is used in almost all the convolutional neural networks or deep learning.

1. *Softmax:*

$$\text{Softmax}(y_i) = \exp(y_i) / \sum \exp(y_j)$$



In general we use softmax activation function when we have multiple output units. For example for predicting hand written digits we have 10 possibilities. We have 10 output units, for getting the 10 probabilities of a given digit we use softmax.

Activation functions can be different for hidden and output layers.

What is categorical variable? What is numeric variable?

Loss Functions

Regression: When actual Y values are numeric. Eg: Price of house as output variable, range of price of a house can vary within certain range.

For regression problems: For regression problems we generally use RMSE as loss function.

Classification(binary): When the given y takes only two values. i.e 0 or 1 Eg: Whether the person will buy the house and each class is mutually exclusive.

For binary Classification problems: For binary classification problem we generally use **binary cross entropy** as loss function.

$$BCE = -\frac{1}{N} \sum_{i=0}^N y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i)$$

Neural Network representation:

3 input cells , 4 hidden cells and one output cell

$W^{[i]}$ --- represent the weight matrix at the i^{th} layer

$b^{[i]}$ --- represent the bias vector at the i^{th} layer

Shape of $W^{[1]}$ is (3,4) then shape of $W^{[1]T}$ is (4,3)

Shape of $b^{[1]}$ is (4,1)

Shape of $W^{[2]}$ is (4,1) then shape of $W^{[2]T}$ is (1,4)

Shape of $b^{[2]}$ is (1,1)

For single input there are three features and the shape of x is (3,1) i.e [x_1] , $[x_2]$, $[x_3]$]

Forward Propagation of 2 Layer NN for single input

- $Z^{[1]} = W^{[1]T} * x + b^{[1]}$ -----shape of $Z^{[1]}$ is (4,1)
- $a^{[1]} = \text{sigmoid}(Z^{[1]})$ -----shape of $a^{[1]}$ is also (4,1)
- $Z^{[2]} = W^{[2]T} * a^{[1]} + b^{[2]}$ -----shape of $Z^{[2]}$ is (1,1)
- $a^{[2]} = \text{sigmoid}(Z^{[2]})$ -----shape of $a^{[2]}$ is also (1,1)
- $\hat{y} = a^{[2]}$

For multiple inputs:

We stack the features of each input vertically, let us say if we have N inputs.

$X = [[x_1^{(1)}, x_1^{(2)} \dots x_1^{(n)}] , [x_2^{(1)}, x_2^{(2)} \dots x_2^{(n)}] , [x_3^{(1)}, x_3^{(2)} \dots x_3^{(n)}]]$

shape of x is (3,n)

the above equations will be same as follows:

- $Z^{[1]} = W^{[1]T} * X + b^{[1]}$ -----shape of $Z^{[1]}$ is (4,n)
- $a^{[1]} = \text{sigmoid}(Z^{[1]})$ -----shape of $a^{[1]}$ is also (4,n)
- $Z^{[2]} = W^{[2]T} * a^{[1]} + b^{[2]}$ -----shape of $Z^{[2]}$ is (1,n)
- $a^{[2]} = \text{sigmoid}(Z^{[2]})$ -----shape of $a^{[2]}$ is also (1,n)

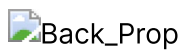
We got predicted value for each input. **This concept of stacking multiple column vectors of inputs is called vectorization.**

Vectorization is faster than writing a for loop over all inputs.....

Derivative of sigmoid activation function

-----IN CLASS BOARD-----

Backward Propagation (Without hidden layer)



BCE Loss: $-y \log a - (1-y) \log(1-a)$ where $a = y^{\wedge}$

$$dL(a, y) / da = da = -y/a + (1-y)/(1-a)$$

$$dL(a, y) / dz = dz = a - y$$

$$dL(a, y) / dw = dw = (a - y) * x$$

$$dL(a, y) / db = db = (a - y)$$

Finally updating w and b

$$w = w - \text{learning_rate} * dw$$

$$b = b - \text{learning_rate} * db$$

Homework: Backward Propagation (With hidden layer)

2NN

- $da^{[2]} =$
- $dz^{[2]} =$
- $dw^{[2]} =$
- $db^{[2]} =$
- $W^{[2]} = W^{[2]} - \text{learning_rate} * dw^{[2]}$
- $b^{[2]} = b^{[2]} - \text{learning_rate} * db^{[2]}$
- $da^{[1]} =$
- $dz^{[1]} =$
- $dw^{[1]} =$
- $db^{[1]} =$
- $W^{[1]} = W^{[1]} - \text{learning_rate} * dw^{[1]}$
- $b^{[1]} = b^{[1]} - \text{learning_rate} * db^{[1]}$

Introduction to keras

- Keras is a minimalist Python library for deep learning that can run on top of TensorFlow.
- It was developed to make implementing deep learning models as fast and easy as possible for research and development.
- The main type of model used in keras is called a Sequence which is a linear stack of layers.
- You create a sequence and add layers to it in the order that you wish for the computation to be performed.
- Once defined, you compile the model which makes use of the underlying framework to optimize the computation to be performed by your model. In this you can specify the loss function and the **optimizer** to be used.
- Once compiled, the model must be fit to data.
- Once trained, you can use your model to make predictions on new data.
- We can summarize the construction of deep learning models in Keras as follows:
 1. Define your model. Create a sequence and add layers.
 2. Compile your model. Specify loss functions and optimizers.
 3. Fit your model. Execute the model using data.
 4. Make predictions. Use the model to generate predictions on new data.



Keras

Click here to see Keras Documentation (<https://keras.io/>)

Creating your first Deep Neural Network using keras (Hands on)

Competition Link (<https://www.kaggle.com/c/diabetes-classification>)

Import dependencies

```
In [1]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense
from keras import optimizers
```

Using TensorFlow backend.

Read the data

```
In [2]: train=pd.read_csv('../input/train.csv')
test=pd.read_csv('../input/test.csv')
subm=pd.read_csv('../input/sample_submission.csv')
```

Stats on Data

In [3]:

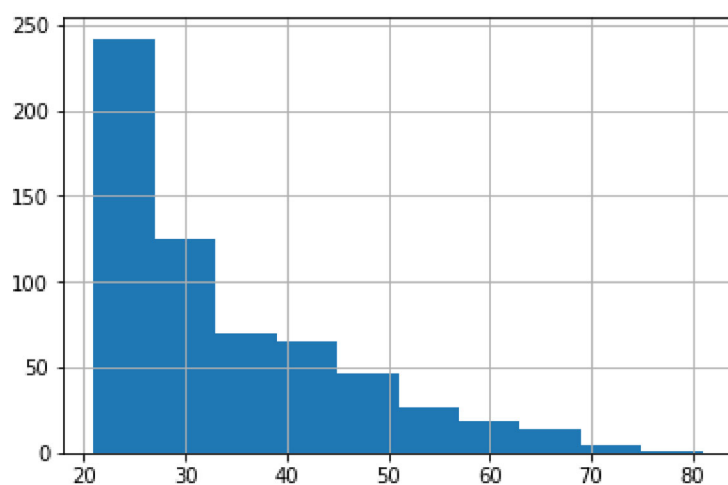
```
train.diabetes.value_counts()
```

Out[3]:

```
0    400
1    214
Name: diabetes, dtype: int64
```

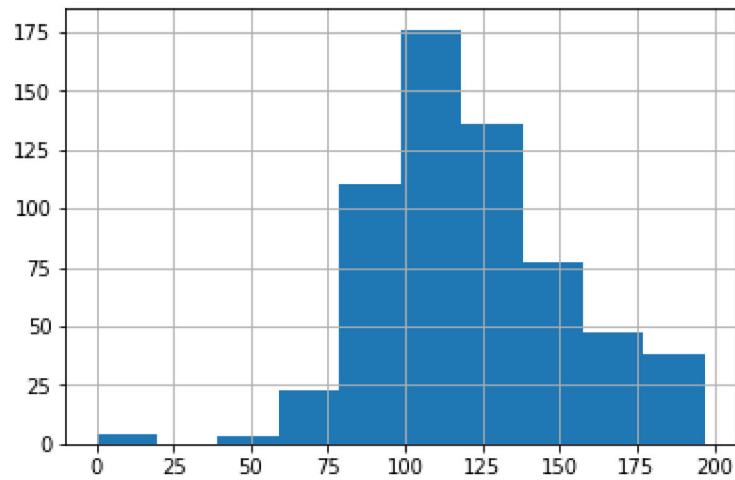
In [4]:

```
train.age.hist()
plt.show()
```



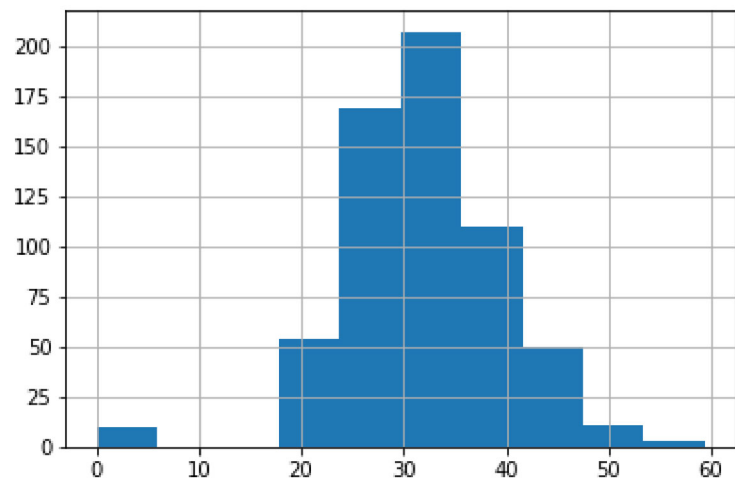
In [5]:

```
train.glucose_concentration.hist()  
plt.show()
```



In [6]:

```
train.bmi.hist()  
plt.show()
```



Hyper parameters

Parameters which are external to the model. Weights and bias are the parameters which are internal to the model generation.

- No of Hidden Units.
- Learning rate.
- Activation function.
- No of epochs

```
In [7]: hidden_units=100
        learning_rate=0.01
        hidden_layer_act='tanh'
        output_layer_act='sigmoid'
        no_epochs=100
```

Model creation

```
In [8]: model = Sequential()
```

Layers Addition

```
In [9]: model.add(Dense(hidden_units, input_dim=8, activation=hidden_layer_act))
        model.add(Dense(hidden_units, activation=hidden_layer_act))
        model.add(Dense(1, activation=output_layer_act))
```

Compilation

In [10]:

```
sgd=optimizers.SGD(lr=learning_rate)
model.compile(loss='binary_crossentropy',optimizer=sgd, metrics=['acc'])
```

Fitting the model

In [11]:

```
train.head()
```

Out[11]:

	p_id	no_times_pregnant	glucose_concentration	blood_pressure	skin_fold_thickness	serum_i
0	316	2	112	68	22	94
1	25	11	143	94	33	146
2	710	2	93	64	32	160
3	658	1	120	80	48	200
4	542	3	128	72	25	190

In [12]:

```
train_x=train.iloc[:,1:9]
train_x.head()
```

Out[12]:

	no_times_pregnant	glucose_concentration	blood_pressure	skin_fold_thickness	serum_insulin
0	2	112	68	22	94
1	11	143	94	33	146
2	2	93	64	32	160
3	1	120	80	48	200
4	3	128	72	25	190

In [13]:

```
train_y=train.iloc[:,9]
train_y.head()
```

Out[13]:

```
0    0
1    1
2    1
3    0
4    1
Name: diabetes, dtype: int64
```

In [14]:

```
model.fit(train_x, train_y, epochs=no_epochs, batch_size=len(train), verbose=2)
```

```
Epoch 1/100
- 0s - loss: 0.6600 - acc: 0.6205
Epoch 2/100
- 0s - loss: 0.6502 - acc: 0.6336
Epoch 3/100
- 0s - loss: 0.6435 - acc: 0.6547
Epoch 4/100
- 0s - loss: 0.6387 - acc: 0.6645
Epoch 5/100
- 0s - loss: 0.6350 - acc: 0.6580
Epoch 6/100
- 0s - loss: 0.6321 - acc: 0.6515
Epoch 7/100
- 0s - loss: 0.6296 - acc: 0.6547
Epoch 8/100
- 0s - loss: 0.6273 - acc: 0.6531
Epoch 9/100
- 0s - loss: 0.6253 - acc: 0.6515
Epoch 10/100
- 0s - loss: 0.6234 - acc: 0.6498
Epoch 11/100
- 0s - loss: 0.6215 - acc: 0.6515
Epoch 12/100
- 0s - loss: 0.6197 - acc: 0.6515
Epoch 13/100
- 0s - loss: 0.6179 - acc: 0.6531
Epoch 14/100
- 0s - loss: 0.6161 - acc: 0.6531
Epoch 15/100
- 0s - loss: 0.6142 - acc: 0.6531
Epoch 16/100
- 0s - loss: 0.6124 - acc: 0.6515
Epoch 17/100
- 0s - loss: 0.6108 - acc: 0.6515
Epoch 18/100
- 0s - loss: 0.6093 - acc: 0.6515
Epoch 19/100
- 0s - loss: 0.6079 - acc: 0.6515
Epoch 20/100
- 0s - loss: 0.6065 - acc: 0.6515
```

```
Epoch 21/100
- 0s - loss: 0.6052 - acc: 0.6531
Epoch 22/100
- 0s - loss: 0.6039 - acc: 0.6547
Epoch 23/100
- 0s - loss: 0.6028 - acc: 0.6547
Epoch 24/100
- 0s - loss: 0.6017 - acc: 0.6564
Epoch 25/100
- 0s - loss: 0.6006 - acc: 0.6564
Epoch 26/100
- 0s - loss: 0.5996 - acc: 0.6564
Epoch 27/100
- 0s - loss: 0.5987 - acc: 0.6580
Epoch 28/100
- 0s - loss: 0.5977 - acc: 0.6596
Epoch 29/100
- 0s - loss: 0.5966 - acc: 0.6629
Epoch 30/100
- 0s - loss: 0.5955 - acc: 0.6678
Epoch 31/100
- 0s - loss: 0.5942 - acc: 0.6710
Epoch 32/100
- 0s - loss: 0.5929 - acc: 0.6710
Epoch 33/100
- 0s - loss: 0.5917 - acc: 0.6710
Epoch 34/100
- 0s - loss: 0.5907 - acc: 0.6710
Epoch 35/100
- 0s - loss: 0.5899 - acc: 0.6759
Epoch 36/100
- 0s - loss: 0.5891 - acc: 0.6759
Epoch 37/100
- 0s - loss: 0.5884 - acc: 0.6775
Epoch 38/100
- 0s - loss: 0.5877 - acc: 0.6808
Epoch 39/100
- 0s - loss: 0.5870 - acc: 0.6824
Epoch 40/100
- 0s - loss: 0.5864 - acc: 0.6840
Epoch 41/100
```

```
- 0s - loss: 0.5858 - acc: 0.6840
Epoch 42/100
- 0s - loss: 0.5852 - acc: 0.6857
Epoch 43/100
- 0s - loss: 0.5846 - acc: 0.6857
Epoch 44/100
- 0s - loss: 0.5839 - acc: 0.6889
Epoch 45/100
- 0s - loss: 0.5833 - acc: 0.6906
Epoch 46/100
- 0s - loss: 0.5826 - acc: 0.6906
Epoch 47/100
- 0s - loss: 0.5819 - acc: 0.6922
Epoch 48/100
- 0s - loss: 0.5812 - acc: 0.6987
Epoch 49/100
- 0s - loss: 0.5805 - acc: 0.6954
Epoch 50/100
- 0s - loss: 0.5799 - acc: 0.6971
Epoch 51/100
- 0s - loss: 0.5793 - acc: 0.6971
Epoch 52/100
- 0s - loss: 0.5788 - acc: 0.6987
Epoch 53/100
- 0s - loss: 0.5783 - acc: 0.7020
Epoch 54/100
- 0s - loss: 0.5778 - acc: 0.7036
Epoch 55/100
- 0s - loss: 0.5774 - acc: 0.7068
Epoch 56/100
- 0s - loss: 0.5769 - acc: 0.7150
Epoch 57/100
- 0s - loss: 0.5765 - acc: 0.7150
Epoch 58/100
- 0s - loss: 0.5761 - acc: 0.7166
Epoch 59/100
- 0s - loss: 0.5757 - acc: 0.7166
Epoch 60/100
- 0s - loss: 0.5753 - acc: 0.7150
Epoch 61/100
- 0s - loss: 0.5749 - acc: 0.7150
```

```
Epoch 62/100
- 0s - loss: 0.5746 - acc: 0.7134
Epoch 63/100
- 0s - loss: 0.5742 - acc: 0.7150
Epoch 64/100
- 0s - loss: 0.5738 - acc: 0.7166
Epoch 65/100
- 0s - loss: 0.5735 - acc: 0.7150
Epoch 66/100
- 0s - loss: 0.5731 - acc: 0.7150
Epoch 67/100
- 0s - loss: 0.5727 - acc: 0.7166
Epoch 68/100
- 0s - loss: 0.5723 - acc: 0.7150
Epoch 69/100
- 0s - loss: 0.5719 - acc: 0.7134
Epoch 70/100
- 0s - loss: 0.5716 - acc: 0.7134
Epoch 71/100
- 0s - loss: 0.5713 - acc: 0.7134
Epoch 72/100
- 0s - loss: 0.5710 - acc: 0.7150
Epoch 73/100
- 0s - loss: 0.5707 - acc: 0.7166
Epoch 74/100
- 0s - loss: 0.5704 - acc: 0.7166
Epoch 75/100
- 0s - loss: 0.5701 - acc: 0.7150
Epoch 76/100
- 0s - loss: 0.5698 - acc: 0.7134
Epoch 77/100
- 0s - loss: 0.5695 - acc: 0.7150
Epoch 78/100
- 0s - loss: 0.5692 - acc: 0.7150
Epoch 79/100
- 0s - loss: 0.5689 - acc: 0.7150
Epoch 80/100
- 0s - loss: 0.5687 - acc: 0.7150
Epoch 81/100
- 0s - loss: 0.5684 - acc: 0.7150
Epoch 82/100
```

```
- 0s - loss: 0.5681 - acc: 0.7150
Epoch 83/100
- 0s - loss: 0.5679 - acc: 0.7150
Epoch 84/100
- 0s - loss: 0.5676 - acc: 0.7150
Epoch 85/100
- 0s - loss: 0.5674 - acc: 0.7150
Epoch 86/100
- 0s - loss: 0.5671 - acc: 0.7150
Epoch 87/100
- 0s - loss: 0.5669 - acc: 0.7134
Epoch 88/100
- 0s - loss: 0.5666 - acc: 0.7134
Epoch 89/100
- 0s - loss: 0.5664 - acc: 0.7134
Epoch 90/100
- 0s - loss: 0.5661 - acc: 0.7101
Epoch 91/100
- 0s - loss: 0.5659 - acc: 0.7117
Epoch 92/100
- 0s - loss: 0.5656 - acc: 0.7117
Epoch 93/100
- 0s - loss: 0.5654 - acc: 0.7117
Epoch 94/100
- 0s - loss: 0.5651 - acc: 0.7117
Epoch 95/100
- 0s - loss: 0.5649 - acc: 0.7101
Epoch 96/100
- 0s - loss: 0.5646 - acc: 0.7117
Epoch 97/100
- 0s - loss: 0.5644 - acc: 0.7117
Epoch 98/100
- 0s - loss: 0.5642 - acc: 0.7117
Epoch 99/100
- 0s - loss: 0.5639 - acc: 0.7117
Epoch 100/100
- 0s - loss: 0.5637 - acc: 0.7134
```

Out[14]:

```
<keras.callbacks.History at 0x7f6372fcc5c0>
```

Predicting the model

```
In [15]: test_x=test.iloc[:,1:]  
         predictions = model.predict(test_x)
```

In [16]:

```
predictions
```

Out[16]:

```
array([[0.32519352],
       [0.1789516 ],
       [0.14183734],
       [0.47569337],
       [0.2092614 ],
       [0.5127485 ],
       [0.5293924 ],
       [0.42303947],
       [0.4265712 ],
       [0.31786793],
       [0.42688912],
       [0.33391085],
       [0.5708902 ],
       [0.23096853],
       [0.5153647 ],
       [0.11459173],
       [0.31273997],
       [0.09854311],
       [0.6493922 ],
       [0.5655404 ],
       [0.26971763],
       [0.27603036],
       [0.39713877],
       [0.45368743],
       [0.46342158],
       [0.17325322],
       [0.12139376],
       [0.56494963],
       [0.29674464],
       [0.22686647],
       [0.31543922],
       [0.21873581],
       [0.27146983],
       [0.12044993],
       [0.36309895],
       [0.21260016],
       [0.3513841 ],
       [0.56871617],
       [0.10023256],
       [0.2525152 ]],
```

[0.17008044],
[0.48271444],
[0.26415515],
[0.5684487],
[0.30454102],
[0.22443233],
[0.5638733],
[0.6186388],
[0.40887114],
[0.09436184],
[0.54153985],
[0.18850315],
[0.4288712],
[0.4990434],
[0.21978141],
[0.37273332],
[0.15427314],
[0.18910368],
[0.33773175],
[0.22330815],
[0.23085932],
[0.16670792],
[0.5153357],
[0.53257006],
[0.43625483],
[0.4782033],
[0.29216114],
[0.2615525],
[0.23257071],
[0.2733693],
[0.6346425],
[0.3793645],
[0.3309497],
[0.10393262],
[0.20699427],
[0.32425094],
[0.31475794],
[0.35728997],
[0.1661055],
[0.19146824],
[0.27904886],

[0.49435005],
[0.35308388],
[0.5346165],
[0.25749615],
[0.49004167],
[0.37203065],
[0.2269389],
[0.09808934],
[0.09966293],
[0.23723662],
[0.3113749],
[0.2760008],
[0.33329624],
[0.5592028],
[0.22092135],
[0.2750802],
[0.4463756],
[0.39119774],
[0.31702927],
[0.6505931],
[0.4623142],
[0.35204563],
[0.10474946],
[0.5565197],
[0.35225472],
[0.27545145],
[0.64125276],
[0.5004096],
[0.3032571],
[0.647756],
[0.320595],
[0.49571234],
[0.45491332],
[0.26812038],
[0.6523489],
[0.6177095],
[0.10529633],
[0.3044368],
[0.51407194],
[0.49724627],
[0.22001687],

```
[0.6363848 ],  
[0.44463426],  
[0.33464143],  
[0.31322625],  
[0.31705213],  
[0.2638835 ],  
[0.21307938],  
[0.30747017],  
[0.23328865],  
[0.2488239 ],  
[0.50459343],  
[0.2572702 ],  
[0.3121979 ],  
[0.5005653 ],  
[0.46742797],  
[0.43466327],  
[0.47940272],  
[0.3388698 ],  
[0.1187799 ],  
[0.27590895],  
[0.30710572],  
[0.14506897],  
[0.56165165],  
[0.17274927],  
[0.3549242 ],  
[0.37421405],  
[0.30124718],  
[0.14527854],  
[0.41243    ],  
[0.1469925 ],  
[0.32247812],  
[0.44214776]], dtype=float32)
```

In [17]:

```
rounded = [int(round(x[0])) for x in predictions]
print(rounded)
```

```
[0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0,
0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0,
1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0,
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, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0,
1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0,
0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Submission

In [18]:

```
subm.diabetes=rounded
subm.to_csv('submission.csv',index=False)
```