INFT 5603 Principals of Data Science, Fall 2023 Graduate College, Information Technology, Arkansas Tech University Assignment 12 / Due date: Sunday, November 19th, 2023 before midnight.

1)

Learning in artificial neural networks (ANNs) involves training the network to perform specific tasks by adjusting the strength of connections between its nodes, which are analogous to neurons in the human brain. These connections are represented by weights, and the weight of an edge determines the strength of the connection between nodes. The learning process in ANNs typically follows specific rules or algorithms, such as:

- **Perceptron Learning Rule**: This involves initializing the weights randomly and repeatedly adjusting them based on each training example until a stopping condition is met, such as a maximum number of iterations or a specific error value.
- **Other Learning Rules**: Besides the Perceptron Learning Rule, there are other early learning rules like the Hebbian Learning Rule and the Delta Learning Rule.

Updating weight coefficients in ANNs can be done using different methods:

- **Intuition-Based Weight Update**: The weights are updated based on the error. For instance, if the actual output (y) is less than the desired output (y_hat), the weight is increased to make y increase. Conversely, if y is greater than y_hat, the weight is decreased.
- **Gradient Descent Method**: This method involves updating the parameters in the direction of the maximum descent in the loss function. Stochastic gradient descent (SGD), a variant of this method, updates the weight for every instance or over mini batches of instances.
- **Backpropagation Algorithm**: This involves computing gradients at each layer using the gradients from the following layer. The algorithm starts from the output layer and backpropagates the gradients to all previous layers, using gradient descent to update weights at every epoch until convergence is achieved.

As for representing excitatory and inhibitory weights in ANNs, typically:

- **Excitatory Weights**: These are positive weights that increase the activation of the neuron they feed into.
- **Inhibitory Weights**: These are negative weights that decrease the activation of the neuron they feed into.

These weights modulate the influence one neuron has on another, analogous to how in biological neural networks, excitatory and inhibitory signals regulate neural activity.

2)

First Hidden Layer (Low-Level Features):

- This layer, being the earliest in the network, handles low-level features.
- It typically identifies basic patterns or simple characteristics of the input data. For example, in image processing, this could involve detecting sharp edges with different gradients and orientations.

Second Hidden Layer (Mid-Level Features):

- The middle layer takes the low-level features identified by the first layer and starts to form more abstract representations.
- It combines simple features to model parts or components of the overall structure. In the context of face recognition, this layer might begin to recognize facial parts like eyes, nose, ears, and lips.

Third Hidden Layer (High-Level Features):

- The last hidden layer, being closest to the output layer, captures the most abstract or high-level features.
- This layer integrates the components identified by the middle layer to form a comprehensive representation. Continuing with the face recognition example, this layer would arrange the recognized facial parts into a coherent representation of a human face.

3)

[n [7]:

```
import pandas as pd
from sklearn import preprocessing
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier


url="https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'Class']

irisdata = pd.read_csv(url, names=names)
x = irisdata.iloc[:, 0:4]
y=irisdata.select_dtypes (include=[object])

# Label encoding
labelEncoder=preprocessing.LabelEncoder()
y = y.apply(labelEncoder.fit_transform)

# Feature scaling
scaler = StandardScaler()
scaler.fit(x)
x = scaler.transform(x)



# Create and train neural network
mlp = MLPClassifier(hidden_layer_sizes=(20, 20, 20), max_iter=2000)
mlp.fit(x, y.values.ravel())

# Do two predictions (This is testing part)
predictions1=mlp.predict( [[3.2, 1.9, 4.0, 5.2]] )
print(predictions1)

predictions2 = mlp.predict( [[4.1, 3.8, 2.3, 4.6]] )
print(predictions2)
```

```
[2]
[2]
```