LINEAR REGRESSION From Scratch:

```python
import numpy as np
class LinearRegression():

  def __init__(self,lr,n_iter):
    self.lr=lr            #learning rate
    self.n_iter=n_iter    #number_of_iteration_to_train
    self.weights=None
    self.bias=None

  def fit_model(self,X,y):
    """to train model using gradient descent"""
    n_samples,n_features=X.shape
    print(n_samples,n_features)
    self.weights=np.random.rand(n_features)
    self.bias=0

    for _ in tqdm(range(self.n_iter)):

      # calculate y_predicted
      y_pred=np.dot(X,self.weights)+self.bias

      # Compute Gradients
      delw=(1/n_samples)*np.dot(X.T,(y_pred-y))
      delb=(1/n_samples)*np.sum(y_pred-y)

      # update weights and bias
      self.weights=self.weights-self.lr*delw
      self.bias=self.bias-self.lr*delb
    return

  def predict(self,X):
    return (np.dot(X,self.weights)+self.bias)
```

LOGISTIC REGRESSION From Scratch:

$$\text{Log Loss} = \sum_{(x,y)\in D} -y\log(y') - (1-y)\log(1-y')$$

```
class LogisticRegression():

  def __init__(self,lr,n_iter):
    self.lr=lr
    self.n_iter=n_iter
    self.weights=None
    self.bias=None

  def sigmoid(self,z):
    return 1 / (1 + np.exp(-z))

  def fit_model(self,X,y):
    n_samples,n_features=X.shape

    # weight bias initialization
    self.weights=np.random.rand(n_features)
    self.bias=0

    # start training iterations
    for _ in range(self.n_iter):
      linear_output=np.dot(X,self.weights)+self.bias
      y_pred=self.sigmoid(linear_output)

      # compute gradient
      delw= (1/n_samples)*np.dot(X.T, (y_pred-y))
      delb= (1/n_samples)*np.sum(y_pred-y)

      # update weights and bias
      self.weights=self.weights-self.lr*delw
      self.bias=self.bias-self.lr*delb

  def predict_class(self,X):
    linear_output=np.dot(X,self.weights)+self.bias
    y_pred=self.sigmoid(linear_output)
    y_pred_class=[1 if i>0.5 else 0 for i in y_pred]
    return y_pred_class
```

K-Means Clustering From Scratch:

```
def kmeans(data, K, max_iterations=100, tolerance=1e-4):
    # Randomly initialize centroids
    centroids = random_initialize_centroids(data, K)
    for _ in range(max_iterations):
        # Assignment Step
        clusters = {}
        for point in data:
            nearest_centroid = find_nearest_centroid(point, centroids)
            if nearest_centroid in clusters:
                clusters[nearest_centroid].append(point)
            else:
                clusters[nearest_centroid] = [point]

        # Update Step
        new_centroids = []
        for centroid in centroids:
            new_centroid = calculate_mean(clusters[centroid])
            new_centroids.append(new_centroid)

        # Check for convergence
        if convergence(new_centroids, centroids, tolerance):
            break

        centroids = new_centroids

    return centroids, clusters

def random_initialize_centroids(data, K):
    # Randomly select K data points as initial centroids
    return data[np.random.choice(data.shape[0], K, replace=False)]
def find_nearest_centroid(point, centroids):
    # Calculate distances between the point and all centroids
    distances = [np.linalg.norm(point - centroid) for centroid in
centroids]
    # Return the centroid with the minimum distance
    return centroids[np.argmin(distances)]
def calculate_mean(points):
    # Calculate the mean (average) of a list of points
    return np.mean(points, axis=0)

def convergence(new_centroids, old_centroids, tolerance):
    # Check if centroids have converged (i.e., no significant change)
    return np.max(np.abs(np.array(new_centroids) -
np.array(old_centroids))) < tolerance
```

Density Based Spatial Clustering of Application with Noise(DB-SCAN):

```
import numpy as np

class DBSCAN:
    def __init__(self, epsilon, min_points):
        self.epsilon = epsilon
        self.min_points = min_points
        self.visited = set()

    def fit(self, data):
        self.data = data
        self.clusters = []

        for point in self.data:
            if point not in self.visited:
                self.visited.add(point)
                neighbors = self.range_query(point)

                if len(neighbors) < self.min_points:
                    continue

                cluster = self.expand_cluster(point, neighbors)
                self.clusters.append(cluster)
        return self.clusters
    def range_query(self, point):
        neighbors = []
        for q in self.data:
            if np.linalg.norm(point - q) <= self.epsilon:
                neighbors.append(q)
        return neighbors
    def expand_cluster(self, point, neighbors):
        cluster = [point]

        for neighbor in neighbors:
            if neighbor not in self.visited:
                self.visited.add(neighbor)
                new_neighbors = self.range_query(neighbor)
                if len(new_neighbors) >= self.min_points:
                    neighbors.extend(new_neighbors)
            if neighbor not in [p for c in self.clusters for p in c]:
                cluster.append(neighbor)

        return cluster
```

Gradient Boost for Regression:

1. Initialize $f_0(x) = \arg\min_\gamma \sum_{i=1}^{N} L(y_i, \gamma)$.
2. For $m = 1$ to M:
   (a) For $i = 1, 2, \ldots, N$ compute
   $$r_{im} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_{m-1}}.$$
   (b) Fit a regression tree to the targets r_{im} giving terminal regions $R_{jm},\ j = 1, 2, \ldots, J_m$.
   (c) For $j = 1, 2, \ldots, J_m$ compute
   $$\gamma_{jm} = \arg\min_\gamma \sum_{x_i \in R_{jm}} L\left(y_i, f_{m-1}(x_i) + \gamma\right).$$
   (d) Update $f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$.
3. Output $\hat{f}(x) = f_M(x)$.

```python
import numpy as np

class GradientBoostingRegression:
    def __init__(self, num_iterations=100, learning_rate=0.1):
        self.num_iterations = num_iterations
        self.learning_rate = learning_rate
        self.models = []

    def fit(self, X, y):
        F = np.mean(y)
        for _ in range(self.num_iterations):
            residuals = -(y - F)

            # Train a linear regressor on the residuals (e.g., using least squares)
            weak_learner = self.train_linear_regressor(X, residuals)

            prediction = weak_learner.predict(X)
            F += self.learning_rate * prediction
            self.models.append(weak_learner)

    def train_linear_regressor(self, X, residuals):
        # Implement linear regression (e.g., using numpy or scikit-learn)
        # Return the linear regressor (e.g., coefficients)
        pass  # Replace with your linear regression training code

    def predict(self, X):
        predictions = np.mean(X)*np.ones(X.shape[0])
        for model in self.models:
            predictions += self.learning_rate * model.predict(X)
        return predictions
```

Ada Boost for Decision Trees:

```
import numpy as np

class DecisionStump:
    def __init__(self):
        self.feature_index = None
        self.threshold = None
        self.alpha = None

    def fit(self, X, y, sample_weights):
        num_samples, num_features = X.shape
        min_error = float('inf')

        for feature_index in range(num_features):
            unique_thresholds = np.unique(X[:, feature_index])
            for threshold in unique_thresholds:
                y_pred = np.ones(num_samples)
                y_pred[X[:, feature_index] < threshold] = -1

                error = np.sum(sample_weights[y_pred != y])

                if error < min_error:
                    min_error = error
                    self.feature_index = feature_index
                    self.threshold = threshold

        # Calculate alpha (classifier weight)
        self.alpha = 0.5 * np.log((1 - min_error) / (min_error + 1e-
10))

    def predict(self, X):
        num_samples = X.shape[0]
        y_pred = np.ones(num_samples)
        y_pred[X[:, self.feature_index] < self.threshold] = -1
        return y_pred

class AdaBoost:
    def __init__(self, num_iterations=50):
        self.num_iterations = num_iterations
        self.classifiers = []
        self.alphas = []

    def fit(self, X, y):
        num_samples = X.shape[0]
        sample_weights = np.ones(num_samples) / num_samples
```

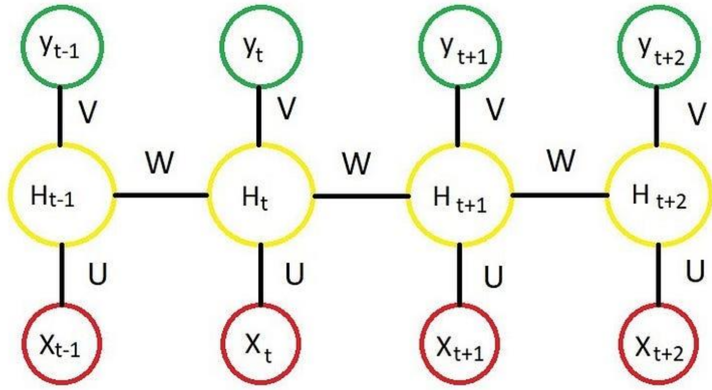
```
        for _ in range(self.num_iterations):
            classifier = DecisionStump()
            classifier.fit(X, y, sample_weights)

            y_pred = classifier.predict(X)
            weighted_error = np.sum(sample_weights[y_pred != y]) /
np.sum(sample_weights)

            # Calculate classifier weight (alpha)
            alpha = 0.5 * np.log((1 - weighted_error) / (weighted_error
+ 1e-10))
            self.alphas.append(alpha)

            # Update sample weights
            sample_weights *= np.exp(-alpha * y * y_pred)
            sample_weights /= np.sum(sample_weights)

            self.classifiers.append(classifier)

    def predict(self, X):
        num_samples = X.shape[0]
        final_predictions = np.zeros(num_samples)

        for alpha, classifier in zip(self.alphas, self.classifiers):
            final_predictions += alpha * classifier.predict(X)

        return np.sign(final_predictions)

# Example usage:
if __name__ == "__main__":
    # Generate synthetic data for binary classification
    np.random.seed(0)
    X = np.random.rand(100, 2)
    y = np.where(X[:, 0] + X[:, 1] > 1, 1, -1)

    # Train the AdaBoost classifier with Decision Stumps as base
learners
    adaboost = AdaBoost(num_iterations=50)
    adaboost.fit(X, y)

    # Make predictions
    X_test = np.array([[0.7, 0.3], [0.4, 0.6]])
    y_pred = adaboost.predict(X_test)
    print("Predicted:", y_pred)
```

Neural Network for binary Clf:

```
import numpy as np

# Define the sigmoid activation function and its derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size,
learning_rate=0.1):
        # Initialize network architecture and hyperparameters
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.learning_rate = learning_rate

        # Initialize weights and biases with random values
        self.weights_input_hidden = np.random.randn(self.input_size,
self.hidden_size)
        self.bias_hidden = np.zeros((1, self.hidden_size))
        self.weights_hidden_output = np.random.randn(self.hidden_size,
self.output_size)
        self.bias_output = np.zeros((1, self.output_size))

    def forward(self, X):
        # Forward propagation through the network
        self.hidden_input = np.dot(X, self.weights_input_hidden) +
self.bias_hidden
        self.hidden_output = sigmoid(self.hidden_input)
        self.output_input = np.dot(self.hidden_output,
self.weights_hidden_output) + self.bias_output
        self.predicted_output = sigmoid(self.output_input)
        return self.predicted_output
    def backward(self, X, y):
        # Backpropagation and weight updates
        error = y - self.predicted_output

        # Calculate gradients
        delta_output = error *
sigmoid_derivative(self.predicted_output)
        d_weights_hidden_output = np.dot(self.hidden_output.T,
delta_output)
```

```
        delta_hidden = np.dot(delta_output,
self.weights_hidden_output.T) * sigmoid_derivative(self.hidden_output)
        d_weights_input_hidden = np.dot(X.T, delta_hidden)

        # Update weights and biases
        self.weights_hidden_output += self.learning_rate *
d_weights_hidden_output
        self.bias_output += self.learning_rate * np.sum(delta_output,
axis=0)
        self.weights_input_hidden += self.learning_rate *
d_weights_input_hidden
        self.bias_hidden += self.learning_rate * np.sum(delta_hidden,
axis=0)
     def train(self, X, y, epochs):
        for epoch in range(epochs):
            # Forward and backward pass for each data point
            for i in range(len(X)):
                input_data = X[i].reshape(1, -1)
                target_output = y[i].reshape(1, -1)
                predicted_output = self.forward(input_data)
                self.backward(input_data, target_output)

            # Calculate and print the mean squared error for this epoch
            mse = np.mean(np.square(y - self.predict(X)))
            print(f"Epoch {epoch + 1}/{epochs}, Mean Squared Error:
{mse:.4f}")
# Example usage:
if __name__ == "__main__":
    # Generate synthetic data for binary classification
    np.random.seed(0)
    X = np.random.rand(100, 2)
    y = np.where(X[:, 0] + X[:, 1] > 1, 1, 0)

    # Define and train the neural network
    input_size = 2
    hidden_size = 4
    output_size = 1
    learning_rate = 0.1
    epochs = 1000

    nn = NeuralNetwork(input_size, hidden_size, output_size,
learning_rate)
    nn.train(X, y, epochs)
    # Make predictions
    X_test = np.array([[0.7, 0.3], [0.4, 0.6]])
    predictions = nn.forward(X_test)
    print("Predicted:", predictions)
```

Recurrent neural networks



At Timestep (t)

$Ht = \sigma (U * Xt + W * Ht\text{-}1)$
$yt = \text{Softmax} (V * Ht)$

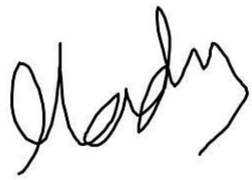
$$J^t(\theta) = -\sum_{j=1}^{|M|} y_{t,j} \log \bar{y}_{t,j}$$

$$J(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{j=1}^{|M|} y_{t,j} \log \bar{y}_{t,j}$$

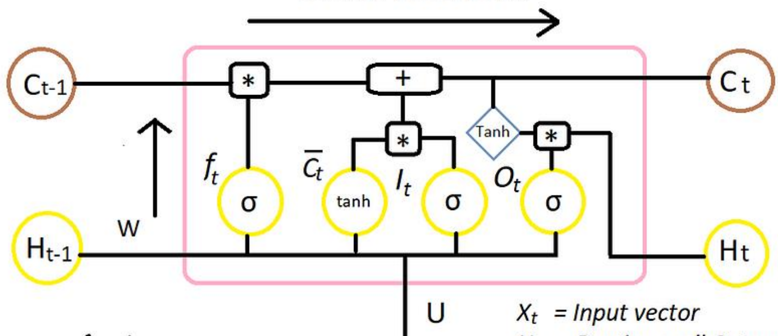
M = vocabulary , J(θ) = Cost function

Cross Entropy Loss

U = Weight vector for Hidden layer
V = Weight vector for Output layer
W = Same weight vector for different Timesteps
X = Word vector for Input word
y = Word vector for Output word



LSTM Network



X_t = Input vector
H_{t-1} = Previous cell Output
C_{t-1} = Previous Cell Memory
Ht = Current cell Output
Ct = Current cell Memory

$\boxed{*}$ = *Element-wise multiplication*
$\boxed{+}$ = *Element-wise addition*

$f_t = \sigma (X_t * U_f + H_{t-1} * W_f)$
$\bar{C}_t = \tanh (X_t * U_c + H_{t-1} * W_c)$
$I_t = \sigma (X_t * U_i + H_{t-1} * W_i)$
$O_t = \sigma (X_t * U_o + H_{t-1} * W_o)$

$C_t = f_t * C_{t-1} + I_t * \bar{C}_t$
$H_t = O_t * \tanh (C_t)$

W, U = weight vectors for forget gate (f), candidate (c) , i/p gate (I) and o/p gate (O)

Note : These are different weights for different gates, for simpicity's sake, I mentioned W and U