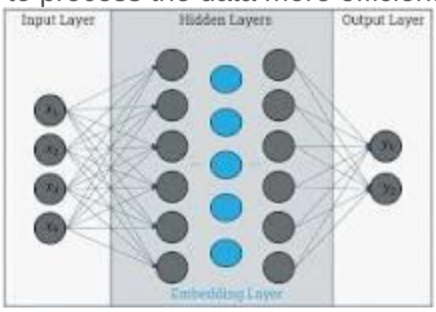
PHASE 2

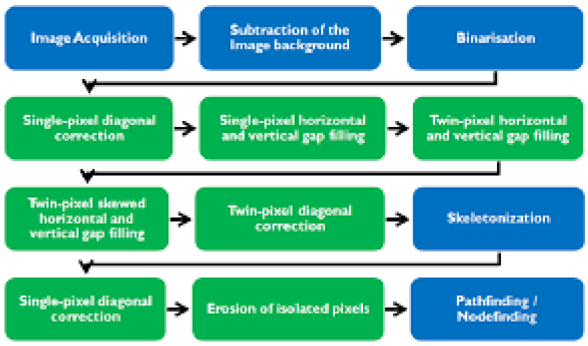
MARKET BASKET ANALYSIS USING NEURAL NETWORKS

Market basket analysis using neural network techniques involves leveraging deep learning models to predict and discover patterns in consumer purchasing behavior. Here's an overview of how neural networks can be applied to market basket analysis:

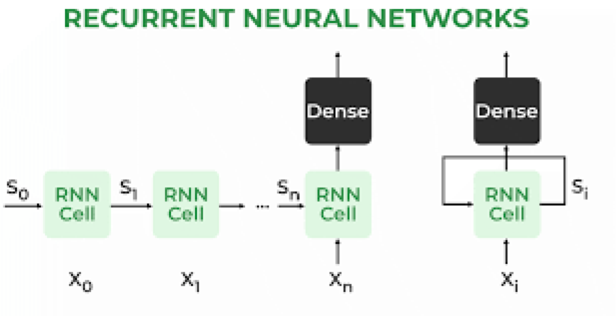
1. **Embedding Layer**: Use an embedding layer to represent products as dense vectors in a continuous space. Each product is mapped to a point in this space, allowing the neural network to learn meaningful representations of items.An embedding layer is a type of hidden layer in a neural network. In one sentence, this layer maps input information from a high-dimensional to a lower-dimensional space, allowing the network to learn more about the relationship between inputs and to process the data more efficiently.



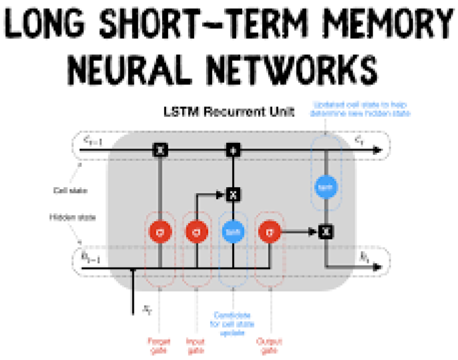
2. **Sequential Data Processing**: Treat the purchase history of a customer as a sequence of interactions (e.g., product purchases). Utilize recurrent neural networks (RNNs) or long short-term memory (LSTM) networks to capture the sequential dependencies in the data.



3. **Recurrent Neural Networks (RNNs)**: RNNs are suitable for processing sequential data, making them effective for analyzing a customer's purchase history and predicting the next item a customer is likely to purchase.



3. **LSTM Networks**: LSTM networks are a type of RNN that can model long-term dependencies, making them well-suited for capturing purchasing patterns over time and predicting future purchases.LSTMs use a series of 'gates' which control how the information in a sequence of data comes into, is stored in and leaves the network. There are three gates in a typical LSTM; forget gate, input gate and output gate. These gates can be thought of as filters and are each their own



neural network.

4. **Multi-Layer Perceptrons (MLPs)**: Use feedforward neural networks, or MLPs, to predict item associations and recommendations based on the embeddings of items. MLPs can learn non-linear relationships and patterns in the market basket data.

```
Methods:
    __init__()
    train(X, y, iterations, reset)
    predict(X)
    initialize_theta_weights()
    backpropagation(X, Y)
    feedforward(X)
    unroll_weights(rolled_data)
    roll_weights(unrolled_data)
    sigmoid(z)
    relu(z)
    sigmoid_derivative(z)
    relu_derivative(z)
"""
import numpy as np

class Mlp():
    '''
```

```
    fully-connected Multi-Layer Perceptron (MLP)
    '''

    def __init__(self, size_layers, act_funct='sigmoid', reg_lambda=0,
bias_flag=True):
        '''
        Constructor method. Defines the characteristics of the MLP

        Arguments:
            size_layers : List with the number of Units for:
                [Input, Hidden1, Hidden2, ... HiddenN, Output] Layers.
            act_funtc   : Activation function for all the Units in the MLP
                default = 'sigmoid'
            reg_lambda: Value of the regularization parameter Lambda
                default = 0, i.e. no regularization
            bias: Indicates is the bias element is added for each layer, but the
output
        '''
        self.size_layers = size_layers
        self.n_layers    = len(size_layers)
        self.act_f       = act_funct
        self.lambda_r    = reg_lambda
        self.bias_flag   = bias_flag

        # Ramdomly initialize theta (MLP weights)
        self.initialize_theta_weights()

    def train(self, X, Y, iterations=400, reset=False):
        '''
        Given X (feature matrix) and y (class vector)
        Updates the Theta Weights by running Backpropagation N tines
        Arguments:
            X          : Feature matrix [n_examples, n_features]
            Y          : Sparse class matrix [n_examples, classes]
            iterations : Number of times Backpropagation is performed
                default = 400
            reset      : If set, initialize Theta Weights before training
                default = False
        '''
        n_examples = Y.shape[0]
#        self.labels = np.unique(y)
#        Y = np.zeros((n_examples, len(self.labels)))
#        for ix_label in range(len(self.labels)):
#            # Find examples with with a Label = lables(ix_label)
#           ix_tmp = np.where(y == self.labels[ix_label])[0]
#            Y[ix_tmp, ix_label] = 1

        if reset:
            self.initialize_theta_weights()
        for iteration in range(iterations):
```

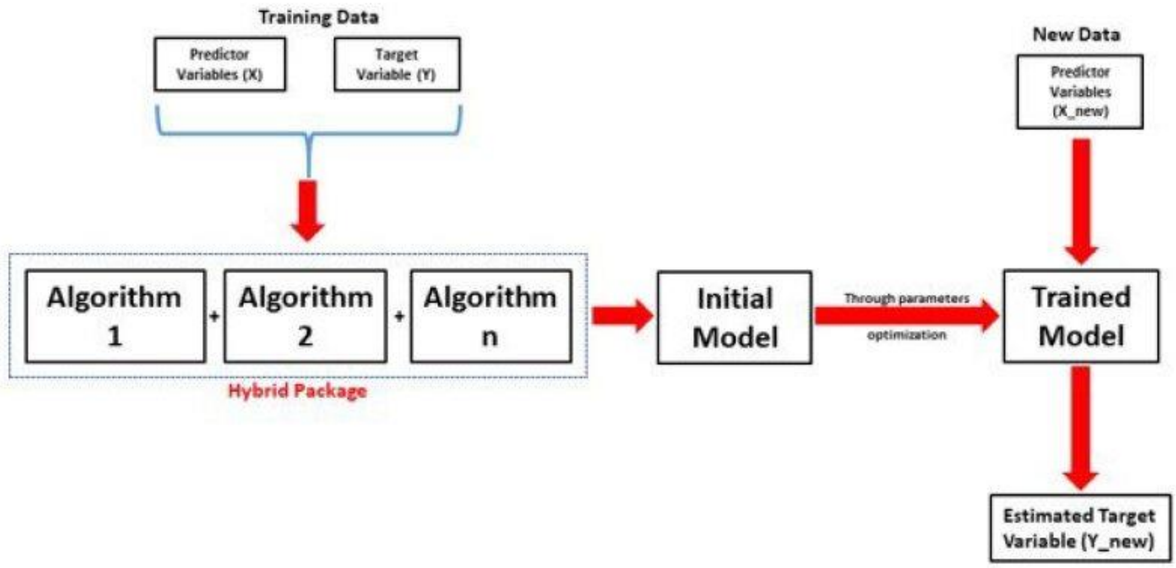
```
            self.gradients = self.backpropagation(X, Y)
            self.gradients_vector = self.unroll_weights(self.gradients)
            self.theta_vector = self.unroll_weights(self.theta_weights)
            self.theta_vector = self.theta_vector - self.gradients_vector
            self.theta_weights = self.roll_weights(self.theta_vector)

    def predict(self, X):
        '''
        Given X (feature matrix), y_hay is computed
        Arguments:
            X       : Feature matrix [n_examples, n_features]
        Output:
            y_hat  : Computed Vector Class for X
        '''
        A , Z = self.feedforward(X)
        Y_hat = A[-1]
        return Y_hat

    def initialize_theta_weights(self):
        '''
        Initialize theta_weights, initialization method depends
        on the Activation Function and the Number of Units in the current layer
        and the next layer.
        The weights for each layer as of the size [next_layer, current_layer + 1]
        '''
        self.theta_weights = []
        size_next_layers = self.size_layers.copy()
        size_next_layers.pop(0)
        for size_layer, size_next_layer in zip(self.size_layers, size_next_layers):
            if self.act_f == 'sigmoid':
                # Method presented "Understanding the difficulty of training deep
feedforward neurla networks"
                # Xavier Glorot and Youshua Bengio, 2010
                epsilon = 4.0 * np.sqrt(6) / np.sqrt(size_layer + size_next_layer)
                # Weigts from a uniform distribution [-epsilon, epsion]
                if self.bias_flag:
                    theta_tmp = epsilon * ( (np.random.rand(size_next_layer,
size_layer + 1) * 2.0 ) - 1)
                else:
                    theta_tmp = epsilon * ( (np.random.rand(size_next_layer,
size_layer) * 2.0 ) - 1)
            elif self.act_f == 'relu':
                # Method presented in "Delving Deep into Rectifiers: Surpassing
Human-Level Performance on ImageNet Classfication"
                # He et Al. 2015
                epsilon = np.sqrt(2.0 / (size_layer * size_next_layer) )
                # Weigts from Normal distribution mean = 0, std = epsion
                if self.bias_flag:
                    theta_tmp = epsilon * (np.random.randn(size_next_layer,
size_layer + 1 ))
```

```
            else:
                theta_tmp = epsilon * (np.random.randn(size_next_layer,
size_layer))
            self.theta_weights.append(theta_tmp)
        return self.theta_weights

    def backpropagation(self, X, Y):
        '''
        Implementation of the Backpropagation algorithm with regularization
        '''
        if self.act_f == 'sigmoid':
            g_dz = lambda x: self.sigmoid_derivative(x)
        elif self.act_f == 'relu':
            g_dz = lambda x: self.relu_derivative(x)

        n_examples = X.shape[0]
        # Feedforward
        A, Z = self.feedforward(X)

        # Backpropagation
        deltas = [None] * self.n_layers
        deltas[-1] = A[-1] - Y
        # For the second last layer to the second one
        for ix_layer in np.arange(self.n_layers - 1 - 1 , 0 , -1):
            theta_tmp = self.theta_weights[ix_layer]
            if self.bias_flag:
                # Removing weights for bias
                theta_tmp = np.delete(theta_tmp, np.s_[0], 1)
            deltas[ix_layer] = (np.matmul(theta_tmp.transpose(), deltas[ix_layer +
1].transpose() ) ).transpose() * g_dz(Z[ix_layer])

        # Compute gradients
        gradients = [None] * (self.n_layers - 1)
        for ix_layer in range(self.n_layers - 1):
            grads_tmp = np.matmul(deltas[ix_layer + 1].transpose() , A[ix_layer])
            grads_tmp = grads_tmp / n_examples
            if self.bias_flag:
                # Regularize weights, except for bias weigths
                grads_tmp[:, 1:] = grads_tmp[:, 1:] + (self.lambda_r / n_examples) *
self.theta_weights[ix_layer][:,1:]
            else:
                # Regularize ALL weights
                grads_tmp = grads_tmp + (self.lambda_r / n_examples) *
self.theta_weights[ix_layer]
            gradients[ix_layer] = grads_tmp;
        return gradients

    def feedforward(self, X):
        '''
        Implementation of the Feedforward
```

```
        '''
        if self.act_f == 'sigmoid':
            g = lambda x: self.sigmoid(x)
        elif self.act_f == 'relu':
            g = lambda x: self.relu(x)

        A = [None] * self.n_layers
        Z = [None] * self.n_layers
        input_layer = X

        for ix_layer in range(self.n_layers - 1):
            n_examples = input_layer.shape[0]
            if self.bias_flag:
                # Add bias element to every example in input_layer
                input_layer = np.concatenate((np.ones([n_examples ,1])
,input_layer), axis=1)
            A[ix_layer] = input_layer
            # Multiplying input_layer by theta_weights for this layer
            Z[ix_layer + 1] = np.matmul(input_layer,
self.theta_weights[ix_layer].transpose() )
            # Activation Function
            output_layer = g(Z[ix_layer + 1])
            # Current output_layer will be next input_layer
            input_layer = output_layer

        A[self.n_layers - 1] = output_layer
        return A, Z


    def unroll_weights(self, rolled_data):
        '''
        Unroll a list of matrices to a single vector
        Each matrix represents the Weights (or Gradients) from one layer to the
next
        '''
        unrolled_array = np.array([])
        for one_layer in rolled_data:
            unrolled_array = np.concatenate((unrolled_array,
one_layer.flatten("F")) )
        return unrolled_array

    def roll_weights(self, unrolled_data):
        '''
        Unrolls a single vector to a list of matrices
        Each matrix represents the Weights (or Gradients) from one layer to the
next
        '''
        size_next_layers = self.size_layers.copy()
        size_next_layers.pop(0)
        rolled_list = []
```

```
        if self.bias_flag:
            extra_item = 1
        else:
            extra_item = 0
        for size_layer, size_next_layer in zip(self.size_layers, size_next_layers):
            n_weights = (size_next_layer * (size_layer + extra_item))
            data_tmp = unrolled_data[0 : n_weights]
            data_tmp = data_tmp.reshape(size_next_layer, (size_layer +
extra_item), order = 'F')
            rolled_list.append(data_tmp)
            unrolled_data = np.delete(unrolled_data, np.s_[0:n_weights])
        return rolled_list

    def sigmoid(self, z):
        '''
        Sigmoid function
        z can be an numpy array or scalar
        '''
        result = 1.0 / (1.0 + np.exp(-z))
        return result

    def relu(self, z):
        '''
        Rectified Linear function
        z can be an numpy array or scalar
        '''
        if np.isscalar(z):
            result = np.max((z, 0))
        else:
            zero_aux = np.zeros(z.shape)
            meta_z = np.stack((z , zero_aux), axis = -1)
            result = np.max(meta_z, axis = -1)
        return result

    def sigmoid_derivative(self, z):
        '''
        Derivative for Sigmoid function
        z can be an numpy array or scalar
        '''
        result = self.sigmoid(z) * (1 - self.sigmoid(z))
        return result

    def relu_derivative(self, z):
        '''
        Derivative for Rectified Linear function
        z can be an numpy array or scalar
        '''
        result = 1 * (z > 0)
        return result
```

6. **Hybrid Models**: Combine various neural network architectures, such as CNNs (Convolutional Neural Networks) for feature extraction and LSTMs for sequential modeling, to create a hybrid model that can capture both item relationships and sequential patterns.

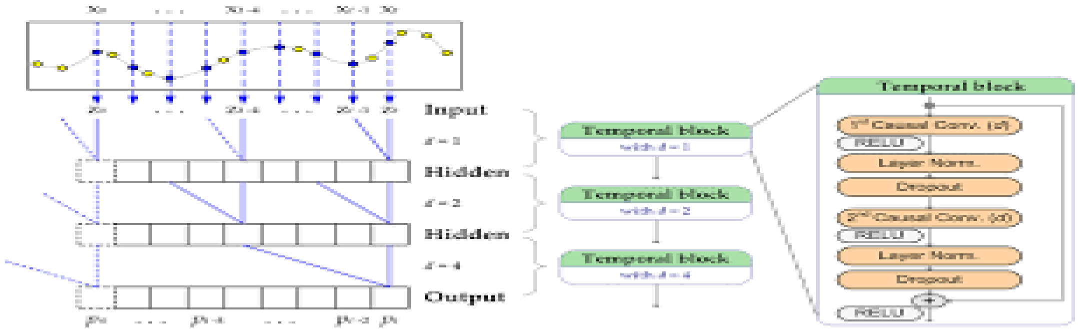


7. **Generative Models**: Explore generative models like Variational Autoencoders (VAEs) or Generative Adversarial Networks (GANs) to generate new product recommendations or simulate potential market baskets.

```
class VAE(keras.Model):
    def __init__(self, encoder, decoder, **kwargs):
        super(VAE, self).__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder

    def train_step(self, data):
        if isinstance(data, tuple):
            data = data[0]
        with tf.GradientTape() as tape:
            z_mean, z_log_var, z = encoder(data)
            reconstruction = decoder(z)
            reconstruction_loss = tf.reduce_mean(
                keras.losses.binary_crossentropy(data, reconstruction)
            )
            reconstruction_loss *= 28 * 28
            kl_loss = 1 + z_log_var - tf.square(z_mean) -
tf.exp(z_log_var)
            kl_loss = tf.reduce_mean(kl_loss)
            kl_loss *= -0.5
            total_loss = reconstruction_loss + kl_loss
        grads = tape.gradient(total_loss, self.trainable_weights)
        self.optimizer.apply_gradients(zip(grads,
self.trainable_weights))
        return {
            "loss": total_loss,
            "reconstruction_loss": reconstruction_loss,
            "kl_loss": kl_loss,
        }
```

:)The above code was used to implement vae in all machine learning algorithms.

8. **Temporal Convolutional Networks (TCNs)**: TCNs are effective for learning temporal patterns in sequences. Use TCNs to model purchase sequences and predict future purchases based on these patterns.



Neural network techniques offer a powerful and flexible framework for market basket analysis, enabling the extraction of intricate patterns and providing valuable insights for retailers to optimize their marketing strategies and improve customer satisfaction.