Backpropagation report - Tom Marini

The goal of this report is to implement a basic backpropagation algorithm.

In [ ]:
```
import numpy as np
```

Backpropagation algorithm is an algorithm used to train a neural network. It actualizes the weights of the network during the training phase. The idea is to modify the weights in order to fit training data by analyzing the error made by the neural network.

A neural network is made of neurons that have weights and a bias. Bias is often accounted as a weight, I decide to keep it distinct from weight in this report. Weights are linking neurons to each other. The neural network I want to implement is a network only made of fully connected layers. This means that the network is split into several layers, each containing several neurons. Since it is fully connected, each neuron from a layer is conntected to all the neurons of the previous and the next layers.

Therefore, I start by implementing a Layer and a Neural Network class that fits this description. A Neural Network oject is made of several Layer and a Layer is a list of neurons that have weights and biases. Note that an activation function can also be provided.

Then, a training phase consisit of forwarding an input through the network and backpropagate the prediction output error of the network until the first layer. A gradient descent is used to update weights. Translating this idea into code, the Neural Network class has a train function that makes the forwarding and backward phases of training data. The Layer class is doted of a forward function that computes the ouput for each layer and a backward that computes the error made by the each neuron. These steps can be repeated several epochs to improve the network.

In [ ]:
```
class Layer: # Fully connected layer (or dense layer)
    """
    A fully connected layer in a neural network.
    A layer has a number of neurons equal to the output_size.
    """
    def __init__(self, input_size, output_size, activation_function=lambda x: x, activation_derivative=lambda x: 1):
        """
        input_size: the number of neurons in the previous layer
        output_size: the number of neurons in the current layer
        """
        self.weights = np.random.randn(input_size, output_size)
```

```
        self.biases = np.random.randn(output_size)
        self.activation_function = activation_function
        self.activation_derivative = activation_derivative

        self.neuron_values = None # the weighted sum of the inputs to the neuron (including the bias). Size is the same as the
        self.output = None # the output of the neuron after applying the activation function. Size is the same as the number o
        self.error = None # partial derivative of the cost function with respect to the neuron values. Size is the same as the
        self.bias_error = None # partial derivative of the cost function with respect to the biases. Size is the same as the n
        self.weight_error = None # partial derivative of the cost function with respect to the weights. Size is the same as th

    def forward(self, input):
        """
        Computes the output of the layer given the input.
        """
        self.neuron_values = np.dot(input, self.weights) + self.biases
        self.output = self.activation_function(self.neuron_values)

    def backward(self, next_layer):
        """
        Computes the error of the layer given the error of the next layer.
        """
        self.error = np.dot(next_layer.error, next_layer.weights.T) * self.activation_derivative(self.neuron_values) # not sur

    def print(self):
        print(f'Weights: {self.weights}')
        print(f'Biases: {self.biases}')
        print(f'Activations: {self.neuron_values}')
        print('')
```

In [ ]:

```
class NeuralNetwork:
    """
    A neural network is a collection of layers.
    """
    def __init__(self):
        self.layers = []

    def add_layer(self, input_size, output_size, activation_function=lambda x: x, activation_derivative=lambda x: 1):
        self.layers.append(Layer(input_size, output_size, activation_function, activation_derivative))

    def forward_propagation(self, x):
```

```
        """
        Compute the output of the network given the input x.
        """
        input = x
        for layer in self.layers:
            layer.forward(input)
            input = layer.output

    def backpropagation(self, x, y, learning_rate):
        """
        Updates the weights and biases of the network given the input x and the target y.
        x: input
        y: target
        """
        # Output layer.
        # Compute the error of the output layer (assuming mean squared error). It depends on the cost function
        self.layers[-1].error = y - self.layers[-1].output

        # Hidden layers. Compute the errors of the hidden layers.
        for i in range(len(self.layers) - 2, -1, -1):
            self.layers[i].backward(self.layers[i+1])

        # Compute the cost variation with respect to the weights and biases
        for i in range(len(self.layers)):
            # Compute the weights errors
            layer = self.layers[i]
            if i == 0: # input layer
                prev_layer_output = x
            else:
                prev_layer_output = self.layers[i-1].output
            layer.weight_error = np.outer(prev_layer_output, layer.error)
            # Compute the biases errors
            layer.bias_error = layer.error

        # Update the weights and biases
        # This is the stochastic gradient descent because we update the weights and biases after each training example
        for layer in self.layers:
            layer.weights += learning_rate * layer.weight_error
            layer.biases += learning_rate * layer.bias_error

    def train(self, x_train, y_train, learning_rate, nb_epochs=1):
```

```
        """
        Trains the network on the training data.
        """
        for epoch in range(nb_epochs):
            self.train_epoch(x_train, y_train, learning_rate)

    def train_epoch(self, x_train, y_train, learning_rate):
        """
        Trains the network on the training data for one epoch.
        """
        for x, y in zip(x_train, y_train): # x is an input, y is a target
            self.forward_propagation(x)
            self.backpropagation(x, y, learning_rate)

    def compute_loss(self, y_true, y_pred):
        return np.mean((y_true - y_pred) ** 2)

    def predict(self, x):
        """
        Applies the network to the input x.
        """
        self.forward_propagation(x)
        return self.layers[-1].output

    def print_network(self):
        for i, layer in enumerate(self.layers):
            print(f'Layer {i}')
            layer.print()
```

Let's test the backpropagation algorithm over some fictive data. I will use the XOR dataset to test the algorithm. The XOR dataset is a simple dataset that is not linearly separable. It is a good dataset to test the backpropagation algorithm.

In [ ]:
```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

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

x_train = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_train = np.array([[0], [1], [1], [0]])
```

```
nn = NeuralNetwork()
nn.add_layer(2, 2, activation_function=sigmoid, activation_derivative=sigmoid_derivative)
nn.add_layer(2, 1, activation_function=sigmoid, activation_derivative=sigmoid_derivative)

nn.train(x_train, y_train, 0.1, 1000)

for x, y in zip(x_train, y_train):
    print(f'Input: {x}')
    print(f'Prediction: {np.round(nn.predict(x))}')
    print(f'Target: {y}')
    print('')
```

```
Input: [0 0]
Prediction: [1.]
Target: [0]

Input: [0 1]
Prediction: [1.]
Target: [1]

Input: [1 0]
Prediction: [1.]
Target: [1]

Input: [1 1]
Prediction: [0.]
Target: [0]
```

After 1000 epochs, the network is not able to predict the XOR dataset, some prediction are incorrect! Let's try to increase the number of epochs to see if the network can learn the XOR dataset. I tried to train the network on the XOR problem for different number of epochs. The network is able to predict the XOR dataset after 10000 epochs. We can notice that the loss function is decreasing over epochs. This is a good sign that the network is learning and it is confirmed by the prediction that are now correct.

In [ ]:
```
for i in range(0, 5):
    nn = NeuralNetwork()
    nn.add_layer(2, 2, activation_function=sigmoid, activation_derivative=sigmoid_derivative)
    nn.add_layer(2, 1, activation_function=sigmoid, activation_derivative=sigmoid_derivative)
    nn.train(x_train, y_train, 0.1, 10**i)
```

```
    print(f'Number of training iterations: {10**i}')
    for x, y in zip(x_train, y_train):
        print(f'Input: {x}, Prediction: {np.round(nn.predict(x))}, Target: {y}')

    print(f'Loss: {nn.compute_loss(y_train, nn.predict(x_train))}')
    print('')
```

```
Number of training iterations: 1
Input: [0 0], Prediction: [0.], Target: [0]
Input: [0 1], Prediction: [0.], Target: [1]
Input: [1 0], Prediction: [0.], Target: [1]
Input: [1 1], Prediction: [0.], Target: [0]
Loss: 0.28493118385080163

Number of training iterations: 10
Input: [0 0], Prediction: [0.], Target: [0]
Input: [0 1], Prediction: [0.], Target: [1]
Input: [1 0], Prediction: [0.], Target: [1]
Input: [1 1], Prediction: [0.], Target: [0]
Loss: 0.24883854837943403

Number of training iterations: 100
Input: [0 0], Prediction: [1.], Target: [0]
Input: [0 1], Prediction: [0.], Target: [1]
Input: [1 0], Prediction: [1.], Target: [1]
Input: [1 1], Prediction: [0.], Target: [0]
Loss: 0.24986954545577877

Number of training iterations: 1000
Input: [0 0], Prediction: [0.], Target: [0]
Input: [0 1], Prediction: [1.], Target: [1]
Input: [1 0], Prediction: [1.], Target: [1]
Input: [1 1], Prediction: [0.], Target: [0]
Loss: 0.024982939619820566

Number of training iterations: 10000
Input: [0 0], Prediction: [0.], Target: [0]
Input: [0 1], Prediction: [1.], Target: [1]
Input: [1 0], Prediction: [1.], Target: [1]
Input: [1 1], Prediction: [0.], Target: [0]
Loss: 6.5634352838248186e-06
```