Back propagation Algorithm

K S Ananth

Task 1

1 Overview:

Back propagation is a widely used algorithm in training feed forward neural networks for supervised learning tasks. It is a method to update the weights in the neural network by propagating the loss back into the network to know how much each node contributed to the loss. This information is then used to update the weights of the network, typically using a simple update rule: weight = weight - learning_rate * gradient.

Here's a high-level overview of the back propagation algorithm:

1. Forward Propagation: Each input passes through the network, layer by layer, until it reaches the output layer. The network's output is then compared with the desired output, and the difference forms the total error.
2. Computing Output Error: The first step in the actual back propagation algorithm is to compute the error derivative for the neurons in the output layer. This is typically done using a loss function that compares the network's output with the desired output.
3. Backward Propagation of Errors: Once we have the error at the output, we propagate it back into the network. This is done by moving from the last hidden layer to the input layer, and at each layer, an error derivative is computed for each neuron.
4. Updating Weights and Biases: After the errors are computed, they are used to update the weights and biases. This is where the learning actually happens. The weights are updated in the direction that most decreases the error. The size of the update is determined by the learning rate, a parameter that you set before training begins.
5. Iterate Until Convergence: Steps 1-4 are repeated for each input in the training set, typically many times, until the network's performance on the training data is satisfactory.

2 Implementation:

For the implementation of the back propagation algorithm I have used numpy, math, random and digraph from graphviz libraries. The graphviz library is used for better visualization of the neural network. In order to implement the algorithm the code has 4 classes which are Variable class, Neuron class, Layer class and Network class. The code for the classes are given below along with their explanations. The Variable class is a wrapper around any variable having data where we define the operations it can perform and also define special functions such as back propagate and other activation functions. The neuron class defines a single neuron in a neural structure which contains variables, weights and a bias. The Layer class defines a layer containing neurons and similarly the Network class is a combination of layers.

2.1 Variable Class

Listing 1: Variable Class

```
class Variable:

    def __init__(self, value, relation=(), operator='', name=''):
        self.value = value
        self.grad = 0.0
        self.previous = set(relation)
        self.operator = operator
        self.name = name
        self.back = lambda: None        # define an empty function for now

    def __repr__(self):
        if(self.name != ''):
            return f"Variable(value={self.value}, name='{self.name}')"
        return f"Variable(value={self.value})"

    def __add__(self, next):
        next = next if isinstance(next, Variable) else Variable(next)
        out = Variable(self.value + next.value, (self, next), '+')

        def back():
            self.grad += 1.0 * out.grad
            next.grad += 1.0 * out.grad
        out.back = back
        return out

    def __sub__(self, next):
        return self + (-next)

    def __rsub__(self, next):
        return next + (-self)

    def __neg__(self):
        return self * -1

    def __radd__(self, next):
        return self + next

    def __truediv__(self, next):
        return self * next**-1

    def __rtruediv(self, next):
        return next * self**-1

    def __pow__(self, next):
        output = Variable(self.value ** next, (self, ), f'**{next}')

        def back():
            self.grad += next * (self.value ** (next - 1)) * output.grad
        output.back = back

        return output

    def __mul__(self, next):
        next = next if isinstance(next, Variable) else Variable(next)
        out = Variable(self.value * next.value, (self, next), '*')
```

```
56
57          def back():
58              self.grad += next.value * out.grad
59              next.grad += self.value * out.grad
60          out.back = back
61
62          return out
63
64      def log(self):
65          epsilon = 1e-9
66          z = self.value
67          out = Variable(math.log(z + epsilon), (self, ), 'log')
68
69          def back():
70              self.grad += (1/(self.value + epsilon)) * out.grad
71
72          out.back = back
73
74          return out
75
76      def __rmul__(self, next):
77          return self * next
78
79      def sigmoid(self):
80          z = self.value
81          temp = (1/(1+math.exp(-1*z)))
82          output = Variable(temp, (self, ), 'sig')
83
84          def back():
85              self.grad += temp * (1 - temp) * output.grad
86          output.back = back
87
88          return output
89
90      def tanh(self):
91          z = self.value
92          temp = (math.exp(2*z) - 1)/(math.exp(2*z) + 1)
93          output = Variable(temp, (self, ), 'tanh')
94
95          def back():
96              self.grad += (1 - temp**2) * output.grad
97
98          output.back = back
99
100         return output
101
102     def exp(self):
103         z = self.value
104         out = Variable(math.exp(z), (self, ), 'exp')
105
106         def back():
107             self.grad += out.value * out.grad
108
109         out.back = back
110
111         return out
112
113     def backward(self):
114         list_in_order = []
115         visited = set()
116         def topological_sort(node):
117             if node not in visited:
118                 visited.add(node)
```

```
                for temp in node.previous:
                    topological_sort(temp)
                list_in_order.append(node)

        topological_sort(self)

        self.grad = 1.0
        list_in_order.reverse()
        for node in list_in_order:
            node.back()
```

1. Initialization: The __init__ method initializes a Variable with a value, a set of previous variables it depends on, an operator that produced it, a name, and a back-propagation function.
2. Representation: The __repr__ method provides a human-readable representation of the Variable.
3. Arithmetic Operations: The class overloads several arithmetic operators (__add__, __sub__, __neg__, __radd__, __truediv__, __rtruediv__, __pow__, __mul__, __rmul__) to allow for mathematical operations between Variable instances. Each operation creates a new Variable that depends on its operands and defines a back propagation function that computes the gradients with respect to its operands.
4. Mathematical Functions: The class also defines several mathematical functions (log, sigmoid, tanh, exp) as methods. Each function creates a new Variable that depends on self and defines a back propagation function that computes the gradient with respect to self.
5. Backward Propagation: The backward method performs the back propagation algorithm. It first performs a topological sort of the nodes in the computational graph to determine the order in which to compute the gradients. It then sets the gradient of self to 1 and propagates this gradient back through the graph by calling each node's back propagation function in the order determined by the topological sort.
6. This Variable class forms the basis for automatic differentiation, which is a key component of many machine learning algorithms, including neural networks. By defining mathematical operations and functions in this way, we can build up complex computations and then easily compute the derivatives of the output with respect to the inputs.

2.2 Neuron Class:

Listing 2: Neuron Class

```
class Neuron:

    def __init__(self, number_of_inputs):
        self.weight = [Variable(random.uniform(-1, 1)) for _ in range(
            number_of_inputs)]
        self.bias = Variable(random.uniform(-1, 1))

    def __call__(self, input):
        # a = summation (wi*xi) + b

```

```
        # weighted_input = (inputi*weighti for inputi, weighti in zip(input,
            self.weight))
        activation = sum((weighti*inputi for inputi, weighti in zip(input, self.
            weight)), self.bias)
        output = activation.tanh()
        return output

    def definition(self):
        return self.weight + [self.bias]
```

1. Initialization: The __init__ method initializes a Neuron with a specified number of inputs. Each input is associated with a weight, which is a Variable with a random initial value between -1 and 1. The neuron also has a bias, which is a Variable with a random initial value between -1 and 1.
2. Activation: The __call__ method computes the activation of the neuron given some input. It first computes the weighted sum of the inputs plus the bias. This is done using a generator expression that multiplies each input by its corresponding weight and then sums the results. The bias is then added to this sum. The result is passed through activation functions(tanh, sigmoid) to produce the output of the neuron.
3. Definition: The definition method returns a list of all the weights and the bias of the neuron. This could be useful for inspecting the neuron's parameters or for updating them during training.
4. This Neuron class can be used to build up a neural network. Each neuron takes a number of inputs, applies a linear transformation to them (the weighted sum plus the bias), and then applies a non-linear activation function (in this case, tanh). The parameters of the transformation (the weights and the bias) are represented as Variable instances, which means they can be updated using the back propagation algorithm implemented in the Variable class.

2.3 Layer Class:

Listing 3: Layer Class

```
class Layer:

    def __init__(self, number_of_input, number_of_output):
        self.neurons = [Neuron(number_of_input) for _ in range(number_of_output)
            ]

    def __call__(self, input):
        output = [temp(input) for temp in self.neurons]
        if(len(output) == 1):
            return output[0]
        return output

    def definition(self):
        output = []
        for neuron in self.neurons:
            for temp in neuron.definition():
                output.append(temp)
        return output
```

1. Initialization: The __init__ method initializes a Layer with a specified number of inputs and outputs. It creates the specified number of Neuron instances, each of which takes the specified number of inputs.
2. Forward Propagation: The __call__ method computes the output of the layer given some input. It does this by calling each neuron in the layer with the input and collecting the results. If the layer contains only one neuron, it returns the output of that neuron directly. Otherwise, it returns a list of the outputs of all the neurons.
3. Definition: The definition method returns a list of all the Variable instances that define the layer. It does this by calling the definition method of each neuron in the layer and appending the results to a list.
4. This Layer class can be used to build up a neural network. Each layer consists of a number of neurons, each of which takes the same input and produces an output. The outputs of all the neurons in a layer form the output of the layer. The parameters of the neurons (the weights and the biases) are represented as Variable instances, which means they can be updated using the back propagation algorithm implemented in the Variable class.

2.4 Network Class:

Listing 4: Network Class

```
class Network:

    def __init__(self, number_of_inputs, number_of_outputs):
        temp = [number_of_inputs] + number_of_outputs
        self.layers = [Layer(temp[i], temp[i+1]) for i in range(len(
            number_of_outputs))]

    def __call__(self, x):
        for layer in self.layers:
            x = layer(x)
        return x

    def definition(self):
        output = []
        for layer in self.layers:
            for temp in layer.definition():
                output.append(temp)
        return output
```

1. Initialization: The __init__ method initializes a Network with a specified number of inputs and a list of numbers of outputs for each layer. It creates a Layer for each number in the list, where each layer takes as input the number of outputs of the previous layer.
2. Forward Propagation: The __call__ method computes the output of the network given some input. It does this by passing the input through each layer in turn, using the output of each layer as the input to the next. The output of the final layer is the output of the network.
3. Definition: The definition method returns a list of all the Variable instances that define the network. It does this by calling the definition method of each layer in the network and appending the results to a list.

4. This Network class can be used to build up a neural network. Each network consists of a number of layers, each of which consists of a number of neurons. The parameters of the neurons (the weights and the biases) are represented as Variable instances, which means they can be updated using the back propagation algorithm implemented in the Variable class.

3 Algorithm:

Listing 5: Network Class

```
# declare the network (this network is only for example purpose)
net = Network(2, [5, 3, 2, 1])

# define any input (this is only for example purpose)
xs = [
    [2.0, 3.0],
    [3.0, 4.0],
    [1.0, 1.0],
    [1.0, 2.0],
]
ys = [0.0, 1.0, 0.0, 1.0]

# train the network
for iteration in range(20):

    # forward pass ----------------------------------------------------------
    y_predictions = [net(temp) for temp in xs]
    loss = sum((y_prediction - y_desired)**2 for y_desired, y_prediction in zip(
        ys, y_predictions)) / (2*len(xs))
    # -----------------------------------------------------------------------

    # backward pass ---------------------------------------------------------
    for temp in net.definition():
        # set the gradient to zero
        temp.grad = 0.0

    # call the backward function on the loss
    loss.backward()
    # -----------------------------------------------------------------------

    # update ----------------------------------------------------------------
    for temp in net.definition():
        temp.value += -0.1 * temp.grad
    # -----------------------------------------------------------------------

    print(f"Iteration number {iteration}: {loss.value}")
```

1. Network Initialization: A Network is initialized with 2 inputs and a list of numbers of outputs for each layer [5, 3, 2, 1]. This creates a network with 4 layers, where the first layer has 5 neurons, the second layer has 3 neurons, the third layer has 2 neurons, and the final layer has 1 neuron.
2. Input Definition: A list of inputs xs and desired outputs ys are defined. These are used to train the network.
3. Training Loop: The network is trained for 20 iterations. In each iteration, the following steps are performed:
   (a) Forward Pass: The network is run on each input to produce a list of predictions.

The loss is then computed as the mean squared error between the predictions and the desired output.

(b) Backward Pass: The gradients of all the Variable instances that define the network are set to zero. Then, the backward method is called on the loss to compute the gradients of the loss with respect to all the Variable instances.

(c) Update: The value of each Variable instance is updated by subtracting a fraction (0.1) of its gradient. This is the gradient descent step that adjusts the parameters of the network to minimize the loss. Print the loss at each iteration.

4. This code demonstrates how to train a neural network using the back propagation algorithm. The network is defined using the Network, Layer, and Neuron classes, and the parameters of the network are represented as Variable instances. The Variable class implements the back propagation algorithm, which allows the network to learn from the training data.

4 Conclusion:

The provided code represents the process of training a neural network. It starts with a forward pass where the network's predictions are computed and the loss is calculated. Then, a backward pass is performed to compute the gradients of the loss with respect to the network's parameters. Finally, an update step adjusts the parameters in the direction that reduces the loss. This process is repeated for a number of iterations, with the goal of minimizing the loss and improving the network's predictions.