Devanagari Handwritten Character Classifier

-DEEPANSH NAGARIA

Dataset:

Each row corresponds to an image of size 32*32, with first column as output (46 classes).So, 1st column for Output and then 1024 columns for image, So, Total Columns = 1025.

Problem statement**:**

Write a program implementing a general neural network architecture. Implement the backpropagation algorithm to train the network. You should train the network using mini Batch Gradient Descent (BGD) with adaptive learning rate (η= η0/ (t) ^1/2).

Specifications:

Implementation should be generic enough for all the so that it works with different architectures, Assume a fully connected architecture .i.e., each unit in a hidden layer is connected to every unit in the next layer. Should implement the algorithm from first principles and not use any existing python/R modules. Use Mean Squared Error (MSE) as the loss function. Use sigmoid as the activation function for the units in output layer. Initiate all the weights not to zero but randomly close to zero. Divide all X by 255 or use normalization to pre-process the data.

Solution Proposed**:**

The solution I am proposing is quite strait forward trying to meet as many specifications as possible mentioned along with the problem statement. The explanation of the code I am submitting follows:

The principle of the backpropagation approach is to model a given function by modifying internal weightings of input signals to produce an expected output signal. The system is trained using a supervised learning method, where the error between the system's output and a known expected output is presented to the system and used to modify its internal state. Technically, the backpropagation algorithm is a method for training the weights in a multilayer feed-forward neural network. As such, it requires a network structure to be defined of one or more layers where one layer is fully connected to the next layer. I am dividing the code into segments as follows:

1) Initializing network: Function named **initialize_network ()** creates a new neural network ready for training. It accepts three parameters, the number of inputs, the number of neurons to have in the hidden layer and the number of outputs.
   You can see that for the hidden layer we create **n_hidden** neurons and each neuron in the hidden layer has **n_inputs + 1** weights, one for each input column in a dataset and an additional one for the bias. You can also see that the output layer that connects to the hidden layer has **n_outputs** neurons, each with **n_hidden + 1** weights. This means that each neuron in the output layer connects to (has a weight for) each neuron in the hidden layer.

2) Forward Propagate**:** It is the technique we will need to generate predictions during training that will need to be corrected, and it is the method we will need after the network is trained to make predictions on new data. We can break forward propagation down into three parts:

   - **Neuron Activation**: Neuron activation is calculated as the weighted sum of the inputs. Much like linear regression. (**activation = sum(weight_i * input_i) + bias**). The logic is implemented in **activate()** function.

   - **Neuron Transfer**: Once a neuron is activated, we need to transfer the activation to see what the neuron output actually is. We can transfer an activation function using the sigmoid function as: **output = 1 / (1 + e^(-activation)).** This logic is specified in the **transfer()** function.

- **Forward Propagation:** We work through each layer of our network calculating the outputs for each neuron. All of the outputs from one layer become inputs to the neurons on the next layer. The function named **forward_propagate()** implements the forward propagation for a row of data from our dataset with our neural network.
  You can see that a neuron's output value is stored in the neuron with the name '**output**'. You can also see that we collect the outputs for a layer in an array named **new_inputs** that becomes the array **inputs** and is used as inputs for the following layer.

3) Back Propagate Error: Error is calculated between the expected outputs and the outputs forward propagated from the network. These errors are then propagated backward through the network from the output layer to the hidden layer, assigning blame for the error and updating weights as they go. This part is broken down into two sections:

- **Transfer Derivative:** Given an output value from a neuron, we need to calculate it's slope. We are using the sigmoid transfer function, the derivative of which can be calculated as**: derivative = output * (1.0 - output)**. Same is implemented in the **transfer_derivative()** function.

- **Error Backpropagation:** The first step is to calculate the error for each output neuron, this will give us our error signal (input) to propagate backwards through the network. The error for a given neuron can be calculated as : **error = (expected - output) * transfer_derivative (output)** . The error signal for a neuron in the hidden layer is calculated as the weighted error of each neuron in the output layer. Think of the error traveling back along the weights of the output layer to the neurons in the hidden layer. The back-propagated error signal is accumulated and then used to determine the error for the neuron in the hidden layer.
  The function named **backward_propagate_error()** implements this procedure. You can see that the error signal calculated for each neuron is stored with the name 'delta'. You can see that the layers of the network are iterated in reverse order, starting at the output and working backwards.

This ensures that the neurons in the output layer have ‘delta’ values calculated first that neurons in the hidden layer can use in the subsequent iteration. I chose the name ‘delta’ to reflect the change the error implies on the neuron (e.g. the weight delta).

You can see that the error signal for neurons in the hidden layer is accumulated from neurons in the output layer where the hidden neuron number **j** is also the index of the neuron’s weight in the output layer **neuron[‘weights’][j]**.

4} Train Network**:** The network is trained using stochastic gradient descent. This involves multiple iterations of exposing a training dataset to the network and for each row of data forward propagating the inputs, back propagating the error and updating the network weights. This part is broken down into two sections:

- **Update Weights**: Once errors are calculated for each neuron in the network via the back propagation method above, they can be used to update weights. Network weights are updated as follows: **weight = weight + learning_rate * error * input.** Function named **update_weights()** that updates the weights.

- **Train network**: This involves first looping for a fixed number of epochs and within each epoch updating the network for each row in the training dataset. You can also see that the sum squared error between the expected output and the network output is accumulated each epoch and printed. This is helpful to create a trace of how much the network is learning and improving each epoch. This logic is implemented in the **train_network**() function.

5) Predict**:** Function named **predict()** implements this procedure. It returns the index in the network output that has the largest probability. It assumes that class values have been converted to integers starting at 0.

Extensions:

This section lists extensions to the code that I wish to explore:

- **Tuning Algorithm Parameters**: Try larger or smaller networks trained for longer or shorter.
- **Additional Methods**. Experimenting with different weight initialization techniques (such as small random numbers) and different transfer functions (such as tanh).
- **More Layers**. Add support for more hidden layers, trained in just the same way as the one hidden layer used in this tutorial.
- **Batch Gradient Descent**. Change the training procedure from online to batch gradient descent and update the weights only at the end of each epoch.(This is the one I missed from the specifications(read the problem in a hurry)).

Why me? :

I am very enthusiastic towards working with your company as an intern as it would definitely be a great learning experience for me. Moreover my interest in the topic shall be my motivation to work and shall bring out the best in me. I on my side am very much looking forward to work with you and share a long bond with your company.