**Neural Network in C# using back propagation**

The Neural Network project attempts to create a flexible and efficient framework for building and training neural networks. It leverages advanced techniques like parallel processing and GPU acceleration to enhance performance. The class is designed to be extensible, allowing for easy modifications and additions to the network architecture, making it suitable for various machine learning applications.

I used a matrix to store neurons (called INeuron), and another structure called NeuriteTensor to manage connections between these neurons across different layers. You can easily set the number of inputs, outputs, hidden layers, and their width, and it even includes gradient clipping to keep training stable.

For performance, it uses a TaskManager with TaskContainer objects to handle various tasks asynchronously, which makes everything run smoother and faster. During initialization, it sets up all the neurons and layers, and loads GPU kernels for efficient calculations. When training, it uses forward propagation to compute activations and backpropagation to update weights and biases. It supports Mean Squared Error (MSE) for loss calculation, and trains over multiple epochs to minimize this loss. Plus, it’s designed to be extensible, so you can easily add or modify network components, layers, and functions.

**Backpropagation**

Backpropagation is a fundamental algorithm used for training artificial neural networks. It leverages the chain rule from calculus to compute gradients of the loss function with respect to each weight in the network, allowing for efficient updates during training. Here’s a detailed breakdown of how backpropagation works, including the role of the chain rule:

**1. Forward Propagation:**

Before diving into backpropagation, it's essential to understand forward propagation, where inputs pass through the network to generate an output. Here's how it works step-by-step:

1. **Input Layer**: The input data is fed into the input layer.
2. **Hidden Layers**: The data is processed through one or more hidden layers, where each neuron performs a weighted sum of its inputs, adds a bias, and applies an activation function.
3. **Output Layer**: The final output is produced after processing through the output layer neurons.

Mathematically, for a single neuron 𝑗 in layer *l*:

where:

* is the weighted sum.
* is the weight connecting neuron 𝑖 in layer to neuron 𝑗 in layer 𝑙.
* is the activation of neuron 𝑖in layer
* is the bias for neuron 𝑗 in layer *l*.
* is the activation function.

**2. Loss Calculation:**

The network's output is compared to the actual target values using a loss function (e.g., Mean Squared Error, Cross-Entropy). The loss function quantifies the error of the network's prediction.

For instance, with Mean Squared Error (MSE):

where:

* ​ is the actual target value.
* is the predicted value.

**3. Backpropagation:**

Backpropagation aims to minimize the loss function by adjusting the weights and biases in the network. It involves the following steps:

1. **Compute the Gradient of the Loss with Respect to Each Output Neuron:** The gradient of the loss 𝐿*L* with respect to the output of neuron 𝑘*k* in the output layer is given by:

For MSE:

1. **Backpropagate the Error:** Using the chain rule, the error is propagated backward through the network to compute gradients for each weight and bias.

**Chain Rule Application:** For a weight the chain rule helps compute the gradient of the loss with respect to this weight by breaking it down into intermediate steps:

* + **Gradient of the Loss with Respect to the Activation:** This term represents the gradient of the loss with respect to the activation of neuron *j* in layer *l*.
  + **Gradient of the Activation with Respect to the Weighted Sum:** This is the derivative of the activation function.
    1. For instance, for a sigmoid function ​, the derivative
    2. In this project we use LeakyReLU
  + **Gradient of the Weighted Sum with Respect to the Weight:** This term is simply the activation of the neuron from the previous layer.

Combining these, the gradient for the weight is:

where

1. **Update the Weights and Biases:** Using the computed gradients, the weights and biases are updated to minimize the loss. This is typically done using gradient descent or a variant (e.g., stochastic gradient descent).

where is the learning rate.

**4. Iterate Over Training Data:**

The above steps (forward propagation, loss calculation, backpropagation, and parameter update) are repeated for multiple epochs over the entire training dataset until the network's performance converges.