Feed-Forward Neural Network

A **Feed-Forward Neural Network (FFNN)** consists of layers of neurons where information flows in one direction: from the input layer through hidden layers and finally to the output layer. Each layer processes the input data by applying a weighted sum, followed by an activation function. For this task, we focus on a network with one hidden layer, using **RReLU** as the activation function.

Network structure

1. **Input layer**. The input to the network is a vector $\mathbf{x} \in \mathbb{R}^n$, where n is the number of input features.
2. **Hidden layer**. This layer consists of h neurons.
3. **Output layer**. The output layer contains m neurons, and the network produces an output vector $\mathbf{y} \in \mathbb{R}^m$.

Notation

- $\mathbf{W}^{(1)} \in \mathbb{R}^{h \times n}$ is the weight matrix between the input and hidden layers.
- $\mathbf{b}^{(1)} \in \mathbb{R}^h$ is the bias vector for the hidden layer.
- $\mathbf{W}^{(2)} \in \mathbb{R}^{m \times h}$ is the weight matrix between the hidden and output layers.
- $\mathbf{b}^{(2)} \in \mathbb{R}^m$ is the bias vector for the output layer.
- $\mathbf{x} \in \mathbb{R}^n$ is the input vector.
- $\mathbf{a}^{(1)} \in \mathbb{R}^h$ represents the activations of the hidden layer.
- $\mathbf{a}^{(2)} \in \mathbb{R}^m$ represents the output activations.

Steps

1. Input to Hidden Layer

The input vector $\mathbf{x}$ is multiplied by the weight matrix $\mathbf{W}^{(1)}$ and added to the bias vector $\mathbf{b}^{(1)}$. This gives the pre-activation values for the hidden layer:

$$z_i^{(1)} = \sum_{j=1}^{n} W_{ij}^{(1)} x_j + b_i^{(1)}, \quad \text{for each hidden neuron } i = 1, \ldots, h$$

2. Activation with RReLU

The RReLU activation function is applied to the pre-activation values. For each neuron i in the hidden layer, the output is computed as:

$$a_i^{(1)} = \begin{cases} \max(0, z_i^{(1)}) & \text{if } z_i^{(1)} \geq 0 \\ \alpha_i z_i^{(1)} & \text{if } z_i^{(1)} < 0 \end{cases}$$

Here, α_i is a randomly chosen value from a uniform distribution $\alpha_i \sim \text{Uniform}(l, u)$, where l and u are the lower and upper bounds of the distribution.

3. Hidden to Output Layer

The activations from the hidden layer $\mathbf{a}^{(1)}$ are then multiplied by the weight matrix $\mathbf{W}^{(2)}$ and added to the bias vector $\mathbf{b}^{(2)}$. This gives the pre-activation for the output layer:

$$z_j^{(2)} = \sum_{i=1}^{h} W_{ji}^{(2)} a_i^{(1)} + b_j^{(2)}, \quad \text{for each output neuron } j = 1, \ldots, m$$

4. Output Layer (Final Prediction)

The final output of the network is calculated from the pre-activation values of the output layer. If no activation function is applied in the output layer:

$$y_j = z_j^{(2)}, \quad \text{for each output neuron } j = 1, \ldots, m$$

By following these steps, the network processes input data and makes predictions through forward propagation.

Model training and loss tracking

The training process, for a regression model, uses **Mean Squared Error (MSE)** as the loss function and **Stochastic Gradient Descent (SGD)** as the optimizer.

Mean Squared Error (MSE)

The MSE loss function measures the average squared differences between actual y_i and predicted $\hat{y}_i$ values:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Stochastic Gradient Descent (SGD)

The model parameters θ are updated using:

$$\theta \leftarrow \theta - \eta \nabla_\theta J(\theta)$$

where:

- η is the learning rate

- $\nabla_\theta J(\theta)$ is the gradient of the loss function with respect to θ.

Learning rate scheduler

The learning rate is adjusted dynamically using `StepLR` scheduler (a powerful tool in PyTorch for adjusting the learning rate during training):

$$\eta_t = \eta_0 \cdot \gamma^{\lfloor \frac{t}{\text{step size}} \rfloor}$$

where:

- η_0 is the initial learning rate
- $\gamma = 0.5$ is the decay factor
- $\lfloor \frac{t}{\text{step size}} \rfloor$ represents the number of completed step intervals.

Training Loop

The model is trained over 200 epochs, performing the following steps in each epoch:

1. **Compute predictions**

$$\hat{Y} = f(X; \theta)$$

2. **Calculate Loss**

$$J(\theta) = \frac{1}{n} \sum (y_i - \hat{y}_i)^2$$

3. **Compute gradients**

$$\nabla_\theta J(\theta) = \frac{2}{n} \sum (y_i - \hat{y}_i) \cdot \frac{\partial \hat{y}_i}{\partial \theta}$$

4. **Update parameters**

$$\theta \leftarrow \theta - \eta \nabla_\theta J(\theta)$$

5. **Adjust learning rate**

$$\eta_t = \eta_0 \cdot \gamma^{\lfloor \frac{t}{\text{step size}} \rfloor}$$

This approach ensures a stable training process by gradually decreasing the learning rate while optimizing model weights using gradient descent.