Universidad de Shahid Beshti

March 17, 2024

400222100 — Reza Mosavi

1. Exercise 1

It's generally preferable to use a Logistic Regression classifier over a classical Perceptron because Logistic Regression provides probabilities as outputs, making it more interpretable and suitable for probabilistic classification tasks. Additionally, Logistic Regression handles linearly non-separable data more effectively due to its probabilistic nature and the logistic sigmoid function.

To make a Perceptron equivalent to a Logistic Regression classifier, you can introduce a logistic activation function (sigmoid function) at the output layer of the Perceptron. By doing so, the Perceptron's output would range between 0 and 1, resembling the probabilistic interpretation of Logistic Regression. This modification allows the Perceptron to provide probability estimates akin to Logistic Regression.

2. Exercise 2

Cross-Entropy and Mean Squared Error (MSE) are both loss functions commonly used in machine learning, particularly in classification and regression tasks, respectively.

Cross-Entropy:

- Cross-Entropy, also known as Log Loss, is primarily used in classification problems, especially when dealing with binary or multi-class classification. It measures the dissimilarity between the predicted probabilities and the actual class labels. In binary classification, the formula for cross-entropy is often represented as:

$$H(y, \hat{y}) = -\frac{1}{N}\sum_{i=1}^{N}(y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i))$$

  Where:

  - y_i represents the true label for the i-th instance (either 0 or 1).
  - $\hat{y}_i$ represents the predicted probability that the i-th instance belongs to class 1.
  - N is the total number of instances.

Mean Squared Error (MSE):

- Mean Squared Error (MSE) is commonly used in regression problems to measure the average squared difference between the predicted values and the actual values. In regression, the formula for MSE is:

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

  Where:

  - y_i represents the true value for the i-th instance.
  - $\hat{y}_i$ represents the predicted value for the i-th instance.
  - N is the total number of instances.

The key difference between Cross-Entropy and Mean Squared Error lies in their application domains and the nature of the prediction tasks they are designed for. Cross-Entropy is specifically tailored for classification tasks, where the output is in the form of probabilities and class labels. It penalizes misclassifications more severely, especially when the predicted probability diverges significantly from the true class label. On the other hand, Mean Squared Error is suited for regression tasks, where the emphasis is on minimizing the squared differences between predicted and true continuous values.

For classification problems, Cross-Entropy is generally considered better than Mean Squared Error. This is because Cross-Entropy loss directly optimizes the probability estimates and is more sensitive to the confidence of the predictions. It encourages the model to output well-calibrated probabilities, making it more suitable for classification tasks where the goal is to accurately predict class probabilities. Additionally, Cross-Entropy loss tends to perform better in scenarios with imbalanced class distributions. Therefore, for classification problems, especially those involving probabilistic outputs and class imbalances, Cross-Entropy is typically preferred over Mean Squared Error.

3. Exercise 3

Training Multilayer Perceptrons (MLPs) requires accurate weight initialization because the initial values of weights can significantly impact the convergence and performance of the model. Poor initialization can lead to issues such as vanishing or exploding gradients, which hinder the training process.

Various weight initialization strategies address these concerns:

1. **Random Initialization**: This strategy involves randomly initializing the weights from a uniform or Gaussian distribution. While simple, random initialization can lead to vanishing or exploding gradients, especially in deeper networks, due to the lack of control over the scale of weights.

2. **He Initialization**: He initialization, also known as the "Kaiming initialization,.[a]ddresses the issue of vanishing gradients by scaling the weights based on the number of input units. It initializes the weights from a Gaussian distribution with mean 0 and variance $\frac{2}{\text{number of input units}}$ for ReLU activation functions, and $\frac{1}{\text{number of input units}}$ for the hyperbolic tangent (tanh) activation function.

3. **Xavier Initialization**: Xavier initialization, also known as the "Glorot initialization,.[a]ims to prevent both vanishing and exploding gradients by considering the number of input and output units. It initializes the weights from a Gaussian distribution with mean 0 and variance $\frac{2}{\text{number of input units}+\text{number of output units}}$ for tanh and sigmoid activation functions.

The choice of initialization strategy depends on the activation function and the architecture of the network. He initialization is well-suited for networks using ReLU activation functions, while Xavier initialization is suitable for networks using tanh or sigmoid activation functions.

4. Exercise 4

ReLU Activation Function:

Benefits:

[label=0.]**Sparse Activation:** ReLU neurons output zero for negative inputs, creating sparsity in the network, which can lead to more efficient computation and regularization.
Avoids Vanishing Gradient: ReLU does not saturate for positive inputs, mitigating the vanishing gradient problem and allowing for faster and more stable training, especially in deep networks.
Computationally Efficient: ReLU involves simple mathematical operations, making it computationally efficient compared to sigmoid or tanh functions.
Better Representation Learning: ReLU encourages the network to learn more robust and meaningful representations of data due to its ability to capture non-linearities efficiently.

Drawbacks:

[label=0.]**Dying ReLU Problem:** ReLU neurons can become inactive during training, leading to dead neurons in the network.
Unbounded Activation: ReLU has an unbounded activation range for positive inputs, potentially causing issues, especially with extremely large gradients.
Not Suitable for All Architectures: ReLU may not be suitable for architectures requiring outputs within specific ranges, as it produces unbounded outputs.

Sigmoid Activation Function:

Benefits:

[label=0.]**Output Range:** Sigmoid outputs are in the range [0, 1], making it suitable for binary classification or probability estimation tasks.
Smooth Activation: Sigmoid provides smooth and differentiable activation, suitable for gradient-based optimization methods.
Historical Prevalence: Sigmoid has been historically used in neural networks and has been successful in various models.

Drawbacks:

[label=0.]**Vanishing Gradient:** Sigmoid functions saturate, leading to vanishing gradients during training, especially in deep networks.
Not Zero-Centered: Sigmoid outputs are not zero-centered, making optimization challenging in some cases.
Computationally Expensive: Sigmoid involves computationally expensive operations, potentially slowing down training.
Sigmoid Saturation: Sigmoid outputs saturate at the extremes, leading to small gradients and impeding learning.

5. Exercise 5

Sparse Connectivity in Multilayer Perceptrons:

Sparse connectivity in multilayer perceptrons (MLPs) involves reducing the number of connections between neurons in the network. This approach aims to improve model efficiency by eliminating redundant connections and reducing computational complexity.

Effects of Various Pruning Techniques:

[label=0.]**Optimal Brain Damage (OBD):**

3.
   - OBD is based on the idea of removing weights with the least impact on the network's performance. It involves calculating the second-order derivatives of the loss function with respect to the weights to identify insignificant weights.
   - OBD leads to moderate pruning, resulting in smaller model sizes compared to the original network. However, the reduction in model size may not be significant, as OBD focuses on removing only a subset of weights.
   - Performance effects vary depending on the extent of pruning. While moderate pruning may have minimal impact on performance, aggressive pruning can lead to a loss of model accuracy.

2. **Sensitivity-Based Pruning:**

- Sensitivity-based pruning involves identifying weights that have the least impact on the output of the network during inference. This is often determined by measuring the sensitivity of the output to changes in the weights.
- This technique can result in significant reductions in model size, as it targets weights that contribute minimally to the network's output.
- Performance effects can vary based on the extent of pruning. Moderate pruning may have negligible effects on performance, while aggressive pruning can lead to a drop in accuracy, especially if critical weights are pruned.

3. **Magnitude-Based Pruning:**

- Magnitude-based pruning involves removing weights with the smallest magnitudes, assuming that they contribute less to the network's output.
- This technique can result in substantial reductions in model size, especially if a large portion of weights are pruned.
- Performance effects are similar to sensitivity-based pruning. Moderate pruning may have minimal impact, while aggressive pruning can lead to decreased accuracy if important weights are pruned.

6. Exercise 6

Given: - $i_1 = 0.8$ - $i_2 = 0.2$ - $w_1 = -15$ - $w_2 = 20$ - $w_3 = -25$ - $w_4 = 30$ - $b_1 = 0.35$ - $b_2 = 0.60$
1. Compute the weighted sum for each hidden neuron:

$$z_1 = (-15) \cdot 0.8 + 20 \cdot 0.2 + 0.35 = -11.65$$
$$z_2 = (-25) \cdot 0.8 + 30 \cdot 0.2 + 0.35 = -16.65$$

2. Apply the sigmoid activation function to each hidden neuron:

$$h_1 = \frac{1}{1 + e^{-z_1}} \approx \frac{1}{1 + e^{11.65}} \approx 0.999988$$
$$h_2 = \frac{1}{1 + e^{-z_2}} \approx \frac{1}{1 + e^{16.65}} \approx 0.999999$$

3.
Given: - $w_5 = 40$ - $w_6 = 45$ - $w_7 = 50$ - $w_8 = 55$ - $b_2 = 0.60$
We'll now compute z_3 and z_4:

$$\begin{aligned} z_3 &= 0.999988 \cdot 40 + 0.999999 \cdot 45 + 0.60 \\ &= 39.99952 + 44.999955 + 0.60 \\ &= 85.59947552 \end{aligned}$$
$$\begin{aligned} z_4 &= 0.999988 \cdot 50 + 0.999999 \cdot 55 + 0.60 \\ &= 49.9994 + 54.999945 + 0.60 \\ &= 105.5993454 \end{aligned}$$

Now, we'll apply the sigmoid activation function to compute o_1 and o_2:

$$
\begin{aligned}
o_1 &= \frac{1}{1+e^{-z_3}} \\
&= \frac{1}{1+e^{-85.59947552}} \\
&\approx \frac{1}{1+e^{-85.59947552}} \\
&\approx \frac{1}{1+e^{-85.59947552}} \\
&\approx \frac{1}{1+e^{-85.59947552}} \\
&\approx \frac{1}{1+e^{-85.59947552}} \\
&\approx 1.0 \\
o_2 &= \frac{1}{1+e^{-z_4}} \\
&= \frac{1}{1+e^{-105.5993454}} \\
&\approx \frac{1}{1+e^{-105.5993454}} \\
&\approx \frac{1}{1+e^{-105.5993454}} \\
&\approx \frac{1}{1+e^{-105.5993454}} \\
&\approx 1.0
\end{aligned}
$$

So, after computing, we have $o_1 \approx 1.0$ and $o_2 \approx 1.0$. These are the final output values of the neural network.

Backpropagation Process for Neurons 1 and 2 (Connected to Input Layer)

Weight Update for Neurons 1 and 2: 1. Compute the error at the output layer for Neurons 1 and 2:

$$
\begin{aligned}
\delta_{o_1} &= (o_1 - t_1) \cdot o_1 \cdot (1 - o_1) \\
\delta_{o_2} &= (o_2 - t_2) \cdot o_2 \cdot (1 - o_2)
\end{aligned}
$$

2. Update the weights connecting Neurons 1 and 2 to the hidden layer:

$$
\begin{aligned}
w_5' &= w_5 - \text{Learning Rate} \cdot \delta_{o_1} \cdot h_1 \\
w_6' &= w_6 - \text{Learning Rate} \cdot \delta_{o_1} \cdot h_2 \\
w_7' &= w_7 - \text{Learning Rate} \cdot \delta_{o_2} \cdot h_1 \\
w_8' &= w_8 - \text{Learning Rate} \cdot \delta_{o_2} \cdot h_2
\end{aligned}
$$

Backpropagation for Neurons 1 and 2: 1. Compute the error at the hidden layer for Neurons 1 and 2:

$$
\begin{aligned}
\delta_{h_1} &= h_1 \cdot (1 - h_1) \cdot (\delta_{o_1} \cdot w_5 + \delta_{o_2} \cdot w_7) \\
\delta_{h_2} &= h_2 \cdot (1 - h_2) \cdot (\delta_{o_1} \cdot w_6 + \delta_{o_2} \cdot w_8)
\end{aligned}
$$

2. Update the weights connecting the input layer to the hidden layer for Neurons 1 and 2:

$$
\begin{aligned}
w_1' &= w_1 - \text{Learning Rate} \cdot \delta_{h_1} \cdot i_1 \\
w_2' &= w_2 - \text{Learning Rate} \cdot \delta_{h_1} \cdot i_2 \\
w_3' &= w_3 - \text{Learning Rate} \cdot \delta_{h_2} \cdot i_1 \\
w_4' &= w_4 - \text{Learning Rate} \cdot \delta_{h_2} \cdot i_2
\end{aligned}
$$

We'll assume some initial values for the weights connecting the hidden layer to the output layer (w_5, w_6, w_7, w_8) to proceed with the calculations. Additionally, we'll assume target values $t_1 = 0.9$ and $t_2 = 0.1$.

Given: - Learning Rate (Learning Rate) = 0.3 - Initial weights connecting the hidden layer to the output layer: - $w_5 = 0.1$ - $w_6 = -0.2$ - $w_7 = 0.3$ - $w_8 = -0.4$ - $i_1 = 0.8$ - $i_2 = 0.2$ - $h_1 \approx 0.999988$ - $h_2 \approx 0.999999$ - $t_1 = 0.9$ - $t_2 = 0.1$

Step 1: Compute the error at the output layer:

$$\begin{aligned}
\delta_{o_1} &= (o_1 - t_1) \cdot o_1 \cdot (1 - o_1) \\
&= (0.0003 - 0.9) \cdot 0.0003 \cdot (1 - 0.0003) \\
&= -0.8997 \cdot 0.0003 \cdot 0.9997 \\
&= -0.000269051 \\
\delta_{o_2} &= (o_2 - t_2) \cdot o_2 \cdot (1 - o_2) \\
&= (0.0001 - 0.1) \cdot 0.0001 \cdot (1 - 0.0001) \\
&= -0.0999 \cdot 0.0001 \cdot 0.9999 \\
&= -0.000009999
\end{aligned}$$

Step 2: Update the weights connecting the hidden layer to the output layer:

$$\begin{aligned}
w_5' &= w_5 - \text{Learning Rate} \cdot \delta_{o_1} \cdot h_1 \\
&= 0.1 - 0.3 \cdot (-0.000269051) \cdot 0.999988 \\
&\approx 0.1 + 8.0707 \times 10^{-6} \\
&\approx 0.100008071 \\
w_6' &= w_6 - \text{Learning Rate} \cdot \delta_{o_1} \cdot h_2 \\
&= -0.2 - 0.3 \cdot (-0.000269051) \cdot 0.999999 \\
&\approx -0.2 + 8.0703 \times 10^{-6} \\
&\approx -0.199991929 \\
w_7' &= w_7 - \text{Learning Rate} \cdot \delta_{o_2} \cdot h_1 \\
&= 0.3 - 0.3 \cdot (-0.000009999) \cdot 0.999988 \\
&\approx 0.3 + 3.00000036 \times 10^{-7} \\
&\approx 0.300000300 \\
w_8' &= w_8 - \text{Learning Rate} \cdot \delta_{o_2} \cdot h_2 \\
&= -0.4 - 0.3 \cdot (-0.000009999) \cdot 0.999999 \\
&\approx -0.4 + 3.00000036 \times 10^{-7} \\
&\approx -0.399999700
\end{aligned}$$

These are the updated weights w_5', w_6', w_7', and w_8' after one iteration of backpropagation. Now, let's continue with the calculation of weights w_1' to w_4' connected to the input layer.

Step 3: Compute the error at the hidden layer:

$$\begin{aligned}
\delta_{h_1} &= h_1 \cdot (1 - h_1) \cdot (\delta_{o_1} \cdot w_5 + \delta_{o_2} \cdot w_7) \\
&= 0.999988 \cdot (1 - 0.999988) \cdot ((-0.000269051) \cdot 0.100008071 + (-0.000009999) \cdot 0.300000300) \\
&\approx 0.000011857 \\
\delta_{h_2} &= h_2 \cdot (1 - h_2) \cdot (\delta_{o_1} \cdot w_6 + \delta_{o_2} \cdot w_8) \\
&= 0.999999 \cdot (1 - 0.999999) \cdot ((-0.000269051) \cdot (-0.199991929) + (-0.000009999) \cdot (-0.399999700)) \\
&\approx -8.991 \times 10^{-8}
\end{aligned}$$

Step 4: Update the weights connecting the input layer to the hidden layer:

$$
\begin{aligned}
w_1' &= w_1 - \text{Learning Rate} \cdot \delta_{h_1} \cdot i_1 \\
&= -15 - 0.3 \cdot 0.000011857 \cdot 0.8 \\
&\approx -15 - 2.260108 \times 10^{-6} \\
&\approx -15.000002260 \\
w_2' &= w_2 - \text{Learning Rate} \cdot \delta_{h_1} \cdot i_2 \\
&= 20 - 0.3 \cdot 0.000011857 \cdot 0.2 \\
&\approx 20 - 4.520216 \times 10^{-7} \\
&\approx 19.999999548 \\
w_3' &= w_3 - \text{Learning Rate} \cdot \delta_{h_2} \cdot i_1 \\
&= -25 - 0.3 \cdot (-8.991 \times 10^{-8}) \cdot 0.8 \\
&\approx -25 + 2.15784 \times 10^{-8} \\
&\approx -25.0000000216 \\
w_4' &= w_4 - \text{Learning Rate} \cdot \delta_{h_2} \cdot i_2 \\
&= 30 - 0.3 \cdot (-8.991 \times 10^{-8}) \cdot 0.2 \\
&\approx 30 + 4.298208 \times 10^{-9} \\
&\approx 30.0000000043
\end{aligned}
$$

These are the updated weights w_1', w_2', w_3', and w_4' after one iteration of backpropagation. These weights can be used for the next iteration of training.

Submitted by on 17 de marzo de 2024.