First Assignment

Sajjad Zangiabadi 99222046

Question1

Both perceptron and logistic regression tackle binary classification problems. They aim to categorize an input as belonging to one of two classes based on its features.

A perceptron makes decisions based on a weighted sum of the input features. If this sum is positive, the input is on the positive side of a decision boundary and gets classified as class +1. Conversely, a negative sum indicates the negative side and a class -1 assignment. This approach essentially measures the input's distance from the decision line.

Logistic regression employs the sigmoid function to estimate the probability of an input belonging to a specific class. It applies the sigmoid function to the weighted sum of features (like perceptron), but the output is a probability value between 0 and 1. This probability can then be converted into a class label using a threshold, often set at 0.5.

To make a perceptron equivalent to a logistic regression classifier by introducing a non-linear activation function like the sigmoid. This function transforms the perceptron's output from a simple distance measure to a probability-like value between 0 and 1.

Question2

In the subject of machine learning, the selection of an appropriate loss function is crucial for guiding models towards achieving optimal predictive performance. Two frequently employed loss functions are Cross-Entropy and Mean Squared Error (MSE)

In classification, where models predict class probabilities (e.g., spam or not spam), Cross-Entropy loss measures the divergence between these predictions and the actual binary outcomes. It penalizes confidently wrong predictions more heavily, driving the model to assign high probabilities to the correct class, which is crucial for accurate classification.

$$\text{CrossEntropy} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{c=1}^{C} y_c^{(i)} * \log\left(\hat{y}_c^{(i)}\right)$$

Mean Squared Error (MSE) is a loss function used in regression tasks to measure the average squared difference between the predicted continuous values and the actual values.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

Question3

Random Initialization: The most basic method, assigning weights with random values drawn from a uniform or normal distribution. While computationally efficient, it can result in slow learning or saturation of activation functions, particularly in deep networks.

Xavier Initialization (Glorot Initialization): This scientific approach draws weights from a distribution with standard deviation scaled by 1 / sqrt(fan_in), where fan_in is the number of incoming connections to a neuron. It prevents vanishing/exploding gradients and works well for tanh/sigmoid activations in shallow or moderately deep networks.

He Initialization: Specifically designed to address the issue of ReLU neurons not activating for negative inputs, He initialization employs a normal distribution with a standard deviation scaled by sqrt(2 / fan_in). This ensures that the variance of activations is maintained across layers, preventing ReLU units from becoming dormant and promoting faster learning in deep networks.

The selection of an optimal initialization strategy is scenario dependent. Here's a practical guide for MLP practitioners:

- Random Initialization: A viable baseline for shallow networks or resource-constrained scenarios. However, it may necessitate meticulous hyperparameter tuning.
- Xavier Initialization: The preferred choice for activation functions like tanh or sigmoid, functioning well in both shallow and moderately deep networks.
- He Initialization: Ideal for scenarios involving ReLU activations, especially in deep networks, by preventing ReLU unit dormancy and promoting faster learning.

Question4

ReLU: Efficiency and Gradient Flow

- Computational Efficiency: ReLU excels in computational efficiency due to its simple mathematical operations compared to the more complex calculations required by the sigmoid function. This efficiency translates to faster training times, particularly in large neural networks.
- Mitigating the Vanishing Gradient Problem: ReLU addresses the vanishing gradient problem that can hinder the training of deep networks. Unlike sigmoid activations, which saturate at the extremes, ReLU exhibits non-saturation in the positive region, allowing gradients to flow more freely during backpropagation.

Challenges of ReLU

- Dead Neurons: During training, ReLU neurons can become inactive ("dead") if the weighted sum of their inputs consistently falls below zero. This prevents the neuron from updating its weights and essentially renders it non-functional, potentially hindering learning, especially in large networks.
- Unbounded Activations: ReLU activations are unbounded, which can lead to numerical instability issues during training, particularly in networks with high learning rates. This instability can cause the training process to diverge.

Sigmoid: Interpretation and Bounded Outputs

- Smooth Outputs: Sigmoid produces smooth, continuous outputs between 0 and 1. This characteristic makes it well-suited for tasks involving probabilities or binary classification where clear class boundaries are essential.
- Bounded Activation Range: Sigmoid acts as a gatekeeper, squashing input values into a finite range between 0 and 1. This bounded output can be beneficial in specific scenarios to prevent numerical overflow issues.

Sigmoid's Limitations

- Vanishing Gradients: Similar to ReLU, sigmoid activations suffer from the vanishing gradient problem. Saturation at the extremes hinders gradient flow during backpropagation, making it challenging to train deep neural networks with sigmoid activation functions.
- Computational Cost: Sigmoid's calculations, particularly the exponentiation involved, are computationally expensive. This can significantly slow down the training process, especially in large networks.

Question5

Multilayer perceptrons (MLPs) have established themselves as powerful tools for various machine learning tasks. However, their traditional architecture, characterized by dense connectivity between neurons in adjacent layers, can lead to computational inefficiencies and overfitting. Sparse connectivity offers a promising alternative, strategically limiting connections within the network to achieve several advantages.

Benefits of Sparse Connectivity:

- Reduced Model Complexity: By minimizing the number of connections, sparse connectivity leads to a smaller model size. This translates to faster training times, lower memory requirements during inference, and potentially reduced deployment costs on resource-constrained devices.
- Improved Generalizability: Fewer parameters inherent to sparse architectures can help mitigate overfitting. The model is less likely to memorize training data peculiarities and exhibits better performance on unseen examples.
- Enhanced Interpretability: A sparser network simplifies the relationships between neurons, making it easier to understand how the model arrives at its predictions. This increased interpretability facilitates debugging, analysis, and potential model improvements.

Pruning Techniques for Sparse Connectivity:

- Optimal Brain Damage (OBD): OBD meticulously analyzes connections and removes those with the least impact on the network's overall performance. This approach offers significant model size reduction but can be computationally expensive due to its reliance on calculating higher-order derivatives.
- Sensitivity-Based Pruning: This technique identifies connections with minimal influence on the network's output by measuring the sensitivity of the output to changes in parameters. Connections with low sensitivity are then pruned. Sensitivity-based pruning offers a good balance between effectiveness and computational efficiency.
- Magnitude-Based Pruning: This approach prioritizes connections based on their absolute values (magnitudes). Connections with smaller magnitudes are assumed to be less significant and are pruned. While computationally efficient, magnitude-based pruning may overlook crucial connections with surprisingly low values.

Questoin6

Output of h1: sigmoid (z1 = 0.05 * 0.15 + 0.10 * 0.25 + 0.35 = 0.3825) = 0.5945

Output of h2: sigmoid (z2 = 0.05 * 0.20 + 0.10 * 0.30 + 0.35 = 0.3900) = 0.5963

Output of o1: sigmoid (z3 = 0.5945* 0.40 + 0.5963 * 0.50 + 0.60 =1.1359) = 0.7569

Output of o2: sigmoid (z4 = 0.5945* 0.45 + 0.5963 * 0.55 + 0.60 = 1.1955) = 0.7677

Error: (0.01 - 0.7569) ** 2 + (0.99 - 0.7677) ** 2 = 0.6072

derivative of loss w.r.t (with respect to) to z3: - 2 * (0.01 – 0.7569) * 0.7569 * (1 – 0.7569) = 0.2749

derivative of loss w.r.t (with respect to) to z4: -2 * (0.99 – 0.7677) * 0.7677* (1 – 0.7677) = -0.0793 dL with respect to w5: 0.2749 * 0.5945= 0.1634

So: w5 = 0.4 - 0.3 * 0.1634 = 0.3510

dL w.r.t w7: 0.2749 * 0.5963 = 0.1639

So: w7 = 0.50 - 0.3 * 0.1639 = 0.4508

dL w.r.t w6: -0.0793* 0.5945= -0.0471

So: w6 = 0.45 + 0.3 * 0.0471 = 0.4641

dL w.r.t w8: -0.0793* 0.5963 = -0.0473

So: w8 = 0.55 + 0.3 * 0.0473= 0.5642

derivative of loss with respect to z1: (0.2749 * 0.40 + -0.0793 * 0.45) * 0.5945 * (1 - 0.5945) = 0.0179

derivative of loss with respect to: (0.2749 * 0.50 + -0.0793 * 0.55) * 0.5963 * (1- 0.5963) = 0.0226

dL w.r.t w1: 0.0179 * 0.05 = 0.0008

So: w1 = 0.15 - 0.3 * 0.0008 = 0.1498

dL w.r.t w3: 0.0179 * 0.10 = 0.0018

So: w3 = 0.25 - 0.3 * 0.0018 = 0.2495

dL w.r.t w2: 0.0226 * 0.05 = 0.0011

So: w2 = 0.20 - 0.3 * 0.0011 = 0.1997

dL w.r.t w4: 0.0226 * 0.10 = 0.0023

So: w4 = 0.30 - 0.3 * 0.0023 = 0.2993