01/05/2025

Training Day-55

Cost Function and **Gradient Descent**:

Cost Function

A **Cost Function** measures the error or difference between the predicted values and actual values in a model. It quantifies the model's performance and helps in optimizing it. The objective is to minimize the cost function during the training phase to improve the model's accuracy.

Types of Cost Functions:

1. **Mean Squared Error (MSE):**

MSE=1n∑i=1n(yi−y^i)2MSE = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2

Where:

- yiy_i: Actual output
- y^i\hat{y}_i: Predicted output
- nn: Number of data points

2. **Log Loss (Cross-Entropy Loss):** Commonly used for classification tasks.

LogLoss=−1n∑i=1n[yilog (y^i)+(1−yi)log (1−y^i)]Log Loss = -\frac{1}{n} \sum_{i=1}^{n} \left[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) \right]

3. **Hinge Loss:** Used for SVM models in classification.

Gradient Descent

Gradient Descent is an optimization algorithm used to minimize the cost function. It updates the model's parameters iteratively by moving in the direction of the negative gradient of the cost function.

Algorithm Steps:

1. **Initialize** the parameters (e.g., weights and biases) randomly or with zeros.
2. Compute the **gradient** of the cost function with respect to each parameter.
3. Update the parameters using the formula:

θ=θ−α∂J(θ)∂θ\theta = \theta - \alpha \frac{\partial J(\theta)}{\partial \theta}

Where:

- θ\theta: Model parameter

- α\alpha: Learning rate
- ∂J(θ)∂θ\frac{\partial J(\theta)}{\partial \theta}: Gradient of the cost function

4. Repeat until the cost function converges to a minimum.

Variants of Gradient Descent:

1. **Batch Gradient Descent:** Uses the entire dataset for each iteration.
2. **Stochastic Gradient Descent (SGD):** Updates parameters using a single data point at each iteration.
3. **Mini-Batch Gradient Descent:** Combines aspects of batch and SGD by using small batches of data.

This explanation is concise and suitable for understanding the concepts of cost function and gradient descent in the context of machine learning【6†source】【7†source】. Let me know if you need more details or practical examples!

02/05/2025

Training Day-56

Linear Algebra Basics and the Vanilla Implementation of Neural Networks:

Linear Algebra Basics

Linear algebra is fundamental in machine learning and neural networks as it provides the mathematical framework for operations on data.

Key Concepts

1. Scalars: A single number (e.g., x=5x = 5).
2. Vectors: A 1D array of numbers (e.g., v=[v1,v2,…,vn]\mathbf{v} = [v_1, v_2, \ldots, v_n]).
3. Matrices: A 2D array of numbers (e.g., A=[a11a12a21a22]\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}).
4. Tensors: A generalization of vectors and matrices to higher dimensions.

Common Operations

1. Addition: Adding matrices or vectors element-wise.

A+B=[a11+b11a12+b12a21+b21a22+b22]\mathbf{A} + \mathbf{B} = \begin{bmatrix} a_{11} + b_{11} & a_{12} + b_{12} \\ a_{21} + b_{21} & a_{22} + b_{22} \end{bmatrix}

2. Dot Product: For vectors u\mathbf{u} and v\mathbf{v}:

u·v=∑iuivi\mathbf{u} \cdot \mathbf{v} = \sum_{i} u_i v_i

3. Matrix Multiplication: The dot product of rows of the first matrix with columns of the second matrix.

C=A·B\mathbf{C} = \mathbf{A} \cdot \mathbf{B}

4. Transpose: Flipping a matrix over its diagonal.

AT=[a11a21a12a22]\mathbf{A}^T = \begin{bmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \end{bmatrix}

5. Inverse: If A·A−1=I\mathbf{A} \cdot \mathbf{A}^{-1} = \mathbf{I}, then A−1\mathbf{A}^{-1} is the inverse of A\mathbf{A}.

Applications in ML

- Representing datasets (matrices where rows are samples and columns are features).
- Performing transformations (rotation, scaling).
- Solving systems of linear equations.

Vanilla Implementation of Neural Networks

A "vanilla" neural network refers to a simple, fully connected feedforward neural network.

Components

1. Input Layer: The data features.
2. Hidden Layers: Layers between input and output, containing neurons that perform intermediate computations.
3. Output Layer: Provides the final prediction or classification.

Forward Propagation

1. Input to Hidden Layer:

z1=W1·x+b1\mathbf{z}_1 = \mathbf{W}_1 \cdot \mathbf{x} + \mathbf{b}_1

   - W1\mathbf{W}_1: Weight matrix for the hidden layer.
   - b1\mathbf{b}_1: Bias vector for the hidden layer.
   - x\mathbf{x}: Input vector.
   - z1\mathbf{z}_1: Linear transformation result.

2. Activation Function:

a1=f(z1)\mathbf{a}_1 = f(\mathbf{z}_1)

Common activations:

   - Sigmoid: f(x)=11+e−xf(x) = \frac{1}{1 + e^{-x}}
   - ReLU: f(x)=max (0,x)f(x) = \max(0, x)

3. Hidden to Output Layer:

y=f(W2·a1+b2)\mathbf{y} = f(\mathbf{W}_2 \cdot \mathbf{a}_1 + \mathbf{b}_2)

   - W2\mathbf{W}_2: Weight matrix for the output layer.
   - b2\mathbf{b}_2: Bias vector for the output layer.

Backpropagation

1. Compute the loss using a cost function (e.g., Mean Squared Error, Cross-Entropy Loss).
2. Calculate gradients using the chain rule.
3. Update weights and biases using gradient descent: W=W−α·∂J∂W\mathbf{W} = \mathbf{W} - \alpha \cdot \frac{\partial J}{\partial \mathbf{W}}

   - α\alpha: Learning rate.
   - JJ: Cost function.

Python Implementation

```
import numpy as np

# Activation function (Sigmoid)
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Derivative of sigmoid
def sigmoid_derivative(x):
    return x * (1 - x)

# Input dataset
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])

# Initialize weights and biases
np.random.seed(0)
weights1 = np.random.rand(2, 2)
weights2 = np.random.rand(2, 1)
bias1 = np.random.rand(1, 2)
bias2 = np.random.rand(1, 1)

# Training process
for epoch in range(10000):
    # Forward propagation
    z1 = np.dot(X, weights1) + bias1
    a1 = sigmoid(z1)
    z2 = np.dot(a1, weights2) + bias2
    output = sigmoid(z2)
```

```
    # Backpropagation
    loss = y - output
    d_output = loss * sigmoid_derivative(output)
    d_hidden = d_output.dot(weights2.T) * sigmoid_derivative(a1)

    # Update weights and biases
    weights2 += a1.T.dot(d_output)
    weights1 += X.T.dot(d_hidden)
    bias2 += np.sum(d_output, axis=0, keepdims=True)
    bias1 += np.sum(d_hidden, axis=0, keepdims=True)

print("Trained Output:", output)
```

This provides a foundation for understanding and implementing basic neural networks.