Linear Algebra and Calculus for Machine Learning

Sashikumaar Ganesan
Department of Computational and Data Sciences
Indian Institute of Science, Bangalore

by Sashikumaar Ganesan
September 2023

Contents

Chapter 1

Vectors and Vector Spaces

1.1 Definition of a Vector

A vector is a mathematical object that has magnitude and direction. It is often represented as an ordered list of numbers. In the context of linear algebra and machine learning, vectors are used to represent data points, features, weights, and many other concepts.

There are different types of vectors, such as:

- **Row Vector:** A single row of numbers. For example, $[1, 2, 3]$.
- **Column Vector:** A single column of numbers. For example, $\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$.

1.2 Vector Operations

Basic vector operations include addition, subtraction, scalar multiplication, and dot product.

Addition: The addition of two vectors of the same size results in a new vector, where each element is the sum of the corresponding elements of the original vectors.

Code Listing 1.1: Python code for vector addition

```
import numpy as np

a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
c = a + b

print(c)
```

Subtraction The subtraction of two vectors of the same size results in a new vector, where each element is the difference of the corresponding elements of the original vectors.

Code Listing 1.2: Python code for vector subtraction

```
c = a - b
print(c)
```

Scalar Multiplication Scalar multiplication is the multiplication of a vector by a scalar (a single number), resulting in a new vector where each element is the product of the corresponding element of the original vector and the scalar.

Code Listing 1.3: Python code for scalar multiplication

```
c = 2 * a
print(c)
```

Dot Product The dot product of two vectors of the same size is a single number, which is the sum of the products of the corresponding elements of the original vectors.

Code Listing 1.4: Python code for dot product

```
c = np.dot(a, b)
print(c)
```

1.3 Vector Spaces

A vector space (or linear space) is a collection of vectors that is closed under vector addition and scalar multiplication and satisfies eight axioms (e.g., associativity of addition, commutativity of addition, existence of an additive identity, etc.).

Examples of vector spaces that are relevant to machine learning include:

- $\mathbb{R}^n$, the set of all n-dimensional vectors of real numbers.
- The set of all functions from $\mathbb{R}$ to $\mathbb{R}$.
- The set of all polynomials of degree at most n.

In machine learning, vector spaces are used to represent data points, features, and model parameters. For example, in linear regression, the features of each data point can be represented as a vector in $\mathbb{R}^n$, and the model parameters can be represented as a vector in the same space.

1.4 Linear Dependency

A set of vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ is linearly dependent if there exist constants $c_1, c_2, \ldots, c_n$, not all zero, such that $c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \ldots + c_n\mathbf{v}_n = \mathbf{0}$.

Example The vectors $\mathbf{v}_1 = (1, 0)$ and $\mathbf{v}_2 = (2, 0)$ are linearly dependent because $2\mathbf{v}_1 - \mathbf{v}_2 = \mathbf{0}$.

1.5 Norms

A norm is a function that assigns a non-negative length or size to each vector in a vector space.

Euclidean Norm The Euclidean norm, or L^2 norm, of a vector $\mathbf{v} = (v_1, v_2, \ldots, v_n)$ is defined as

$$\|\mathbf{v}\| = \sqrt{v_1^2 + v_2^2 + \ldots + v_n^2}.$$

Example The Euclidean norm of the vector $\mathbf{v} = (3, 4)$ is

$$\|\mathbf{v}\| = \sqrt{3^2 + 4^2} = 5.$$

1.6 Applications in Machine Learning

Linear dependency and norms are important concepts in machine learning. For example, checking for linear dependency is essential for feature selection, as linearly dependent features do not add any new information to the model. Norms, on the other hand, are used to measure the magnitude of vectors, which is crucial for optimization algorithms, regularization, and calculating distances between points in the feature space.

1.7 Python, NumPy, and Matplotlib Examples

NumPy is a powerful library for numerical computing in Python, and Matplotlib is a plotting library for Python.

Checking for Linear Dependency Below is an example of how to use NumPy to check for linear dependency between vectors.

```
import numpy as np

v1 = np.array([1, 0])
v2 = np.array([2, 0])

# Check for linear dependency by computing
# the rank of the matrix formed by the vectors
matrix = np.vstack([v1, v2])
rank = np.linalg.matrix_rank(matrix)
is_linearly_dependent = rank < min(matrix.shape)

print(is_linearly_dependent)
```

This will output:

```
True
```

Computing the Euclidean Norm Below is an example of how to use NumPy to compute the Euclidean norm of a vector.

```
v = np.array([3, 4])
euclidean_norm = np.linalg.norm(v)
print(euclidean_norm)
```

This will output:

```
5.0
```

Visualizing Vectors Below is an example of how to use Matplotlib to visualize vectors.

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.quiver(0, 0, v1[0], v1[1], angles='xy',
    scale_units='xy', scale=1, color='r', label='v1')
ax.quiver(0, 0, v2[0], v2[1], angles='xy',
    scale_units='xy', scale=1, color='b', label='v2')
ax.set_xlim(-1, 3)
ax.set_ylim(-1, 3)
ax.set_aspect('equal')
plt.grid(True)
plt.legend()
plt.show()
```

This will display a plot of the vectors $\mathbf{v}_1$ and $\mathbf{v}_2$.

1.8 Machine Learning Application: Linear Regression

Linear regression is a linear approach to modeling the relationship between a dependent variable and one or more independent variables. The case of one independent variable is called simple linear regression; for more than one, the process is called multiple linear regression.

The formula for linear regression is:

$$y = X\beta + \epsilon$$

Where: - y is the dependent variable (output), - X is the matrix of independent variables (input features), - β is the vector of model parameters (weights), - ϵ is the error term.

The goal of linear regression is to find the best-fitting line through the data points. This is achieved by minimizing the sum of the squared residuals, which is the difference between the actual and predicted values.

Code Listing 1.5: Python code for linear regression using NumPy

```
import numpy as np

# Generate some example data
np.random.seed(0)
X = 2 * np.random.rand(100, 1)
y = 3 + 4 * X + np.random.randn(100, 1)

# Add a column of ones to X to account for
# the bias term
X_b = np.c_[np.ones((100, 1)), X]

# Compute the optimal parameters using the
# normal equation
theta_best = np.linalg.inv(X_b.T.dot(X_b))
                .dot(X_b.T).dot(y)

print(theta_best)
```

In this example, the true model parameters are $\beta = [3, 4]$, and the computed parameters should be close to these values. This code generates some example data, adds a column of ones to X to account for the bias term, and computes the optimal parameters using the normal equation.

1.9 Visualization of Vectors using Matplotlib

Visualizing vectors is crucial for understanding and interpreting data in machine learning and data science. Python's matplotlib library can be used to create visualizations of vectors.

Code Listing 1.6: Python code for vector visualization using Matplotlib

```
import numpy as np
import matplotlib.pyplot as plt

# Define the vectors
a = np.array([2, 3])
b = np.array([4, 1])

# Create the plot
plt.figure(figsize=(5,5))
plt.axvline(x=0, color='grey', lw=1)
plt.axhline(y=0, color='grey', lw=1)
plt.grid(True, which='both',
    linestyle='--', linewidth=0.5)
plt.xlim(-1, 5)
plt.ylim(-1, 5)
plt.quiver(0, 0, a[0], a[1], angles='xy',
    scale_units='xy', scale=1, color='r',
        label='a')
plt.quiver(0, 0, b[0], b[1], angles='xy',
    scale_units='xy', scale=1, color='b',
        label='b')
plt.legend()
plt.show()
```

In this example, two vectors $a = [2, 3]$ and $b = [4, 1]$ are defined and plotted on a 2D plane. The "quiver" function from matplotlib is used to plot the vectors. The resulting plot will

show the two vectors in red and blue, originating from the origin (0,0).

1.10 Basis and Dimension

A set of vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ in a vector space V is called a basis for V if the set is linearly independent and spans V. The dimension of a vector space V, denoted by $\dim(V)$, is the number of vectors in a basis for V.

1.10.1 Example

The vectors $\mathbf{v}_1 = (1, 0)$ and $\mathbf{v}_2 = (0, 1)$ form a basis for $\mathbb{R}^2$ because they are linearly independent and span $\mathbb{R}^2$. Therefore, $\dim(\mathbb{R}^2) = 2$.

1.11 Orthogonal and Orthonormal Bases

Two vectors $\mathbf{u}$ and $\mathbf{v}$ are orthogonal if their dot product is zero, i.e., $\mathbf{u} \cdot \mathbf{v} = 0$. A set of vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ is called an orthogonal set if every pair of vectors in the set is orthogonal. If, in addition, all the vectors in the set have a magnitude of 1, then the set is called an orthonormal set. An orthogonal (or orthonormal) basis for a vector space V is a basis for V that is also an orthogonal (or orthonormal) set.

1.11.1 Example

The vectors $\mathbf{v}_1 = (1, 0)$ and $\mathbf{v}_2 = (0, 1)$ form an orthonormal basis for $\mathbb{R}^2$ because they are orthogonal and have a magnitude of 1.

1.12 Gram-Schmidt Process

The Gram-Schmidt process is a method for finding an orthogonal basis for a vector space V from an arbitrary basis for V. The process involves taking the vectors in the original basis and orthogonalizing them one by one by subtracting their projections onto the vectors that have already been orthogonalized.

1.12.1 Example

Consider the vectors $\mathbf{v}_1 = (1, 1)$ and $\mathbf{v}_2 = (1, -1)$ in $\mathbb{R}^2$. We can orthogonalize these vectors using the Gram-Schmidt process as follows:

1. Start with $\mathbf{v}_1 = (1, 1)$.
2. Subtract the projection of $\mathbf{v}_2$ onto $\mathbf{v}_1$ from $\mathbf{v}_2$:

$$\mathbf{u}_2 = \mathbf{v}_2 - \frac{\mathbf{v}_2 \cdot \mathbf{v}_1}{\mathbf{v}_1 \cdot \mathbf{v}_1}\mathbf{v}_1 = (1, -1) - \frac{(1, -1) \cdot (1, 1)}{(1, 1) \cdot (1, 1)}(1, 1) = (1, -1) - (0, 0) = (1, -1).$$

The vectors $\mathbf{v}_1 = (1, 1)$ and $\mathbf{u}_2 = (1, -1)$ form an orthogonal basis for $\mathbb{R}^2$.

1.13 Applications in Machine Learning

Orthogonal and orthonormal bases are important in machine learning because they can be used to simplify computations and reduce numerical errors. For example, principal component analysis (PCA) is a dimensionality reduction technique that involves finding an orthonormal basis for the data. This new basis, formed by the principal components, captures the most variance in the data while minimizing the reconstruction error.

1.14 Python and NumPy Examples

NumPy is a powerful library for numerical computing in Python. Below is an example of how to use NumPy to compute an orthonormal basis for a vector space using the Gram-Schmidt process.

```
import numpy as np

def gram_schmidt(vectors):
    basis = []
    for v in vectors:
        w = v - np.sum(np.dot(v, b) * b for b in basis)
        if (w > 1e-10).any():
            basis.append(w/np.linalg.norm(w))
    return np.array(basis)

vectors = np.array([[1, 1], [1, -1]])
orthonormal_basis = gram_schmidt(vectors)
print(orthonormal_basis)
```

This will output:

```
[[ 0.70710678  0.70710678]
 [ 0.70710678 -0.70710678]]
```

The output is the orthonormal basis for the vector space spanned by the input vectors.

Chapter 2

Matrices

2.1 Definition of a Matrix

A matrix is a rectangular array of numbers arranged in rows and columns. It is usually denoted by a capital letter (e.g., A, B, C). The entry in the i-th row and j-th column of matrix A is denoted as a_{ij}. For example, a matrix A of size 2×3 is represented as:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix}$$

2.2 Matrix Operations

Addition and Subtraction Matrix addition and subtraction are performed element-wise. For matrices A and B of the same size, the sum $A + B$ and the difference $A - B$ are computed by adding or subtracting the corresponding elements of A and B.

$$A + B = \begin{pmatrix} a_{11} + b_{11} & a_{12} + b_{12} \\ a_{21} + b_{21} & a_{22} + b_{22} \end{pmatrix}$$

$$A - B = \begin{pmatrix} a_{11} - b_{11} & a_{12} - b_{12} \\ a_{21} - b_{21} & a_{22} - b_{22} \end{pmatrix}$$

Code Listing 2.1: Python example

```
import numpy as np

A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

C = A + B  # Matrix Addition
D = A - B  # Matrix Subtraction

print("C=", C)
print("D=", D)
```

Multiplication Matrix multiplication is performed by taking the dot product of the rows of the first matrix and the columns of the second matrix. For example, for matrices A of size 2×3 and B of size 3×2, the product AB is computed as follows:

$$AB = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} \end{pmatrix}$$

Code Listing 2.2: Python example

```
E = np.dot(A, B.T)  # Matrix Multiplication

print("E=", E)
```

Transpose The transpose of a matrix A, denoted as A^T, is obtained by flipping the rows and columns of A.

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$$

$$A^T = \begin{pmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \end{pmatrix}$$

Code Listing 2.3: Python example

```
F = A.T  # Matrix Transpose

print("F=", F)
```

2.3 Special Types of Matrices

Square Matrix A square matrix is a matrix with the same number of rows and columns.

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$$

Diagonal Matrix A diagonal matrix is a square matrix in which all the elements outside the main diagonal are zero.

$$D = \begin{pmatrix} d_{11} & 0 & 0 \\ 0 & d_{22} & 0 \\ 0 & 0 & d_{33} \end{pmatrix}$$

Identity Matrix The identity matrix, denoted as I, is a special type of diagonal matrix in which all the elements of the main diagonal are ones and all other elements are zeros.

$$I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Zero Matrix A zero matrix is a matrix in which all elements are zeros.

$$O = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Code Listing 2.4: Python example

```
import matplotlib.pyplot as plt

# Define the matrices
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

# Compute the operations
C = A + B
D = A - B
E = np.dot(A, B.T)
F = A.T

# Create a figure with subplots
fig, axs = plt.subplots(2, 2)

# Plot the matrices
cax1 = axs[0, 0].matshow(A, cmap='viridis')
fig.colorbar(cax1, ax=axs[0, 0])
axs[0, 0].set_title('A')

cax2 = axs[0, 1].matshow(B, cmap='viridis')
fig.colorbar(cax2, ax=axs[0, 1])
axs[0, 1].set_title('B')

cax3 = axs[1, 0].matshow(C, cmap='viridis')
fig.colorbar(cax3, ax=axs[1, 0])
axs[1, 0].set_title('A + B')

cax4 = axs[1, 1].matshow(D, cmap='viridis')
fig.colorbar(cax4, ax=axs[1, 1])
axs[1, 1].set_title('A - B')

plt.tight_layout()
plt.show()
```

In this example, the matrices A and B are visualized along with their sum $A + B$ and difference $A - B$ using Matplotlib.

2.4 Applications in Machine Learning

Matrices are fundamental in the field of machine learning and are used for various applications such as:

- Representing datasets: Data is often represented as a matrix, where each row is a sample and each column is a feature. For example, a dataset with m samples and n features can be represented as an $m \times n$ matrix.
- Performing transformations: Matrices are used to perform linear transformations on the data, such as scaling, rotation, and translation. These transformations are essential for data preprocessing and feature extraction.
- Solving systems of equations: Many machine learning algorithms, such as linear regression, involve solving systems of linear equations, which can be efficiently solved using matrices. For example, the normal equation used to solve linear regression is $X^T X\theta = X^T y$, where X is the design matrix, θ is the parameter vector, and y is the target vector.

Code Listing 2.5: Python example for ML application

```
from sklearn.datasets import make_regression
from sklearn.linear_model import LinearRegression

# Generate a random regression problem
X, y = make_regression(n_samples=100,
       n_features=1, noise=20, random_state=0)

# Fit a linear regression model
model = LinearRegression()
model.fit(X, y)

# Get the parameters of the model
intercept = model.intercept_
coef = model.coef_

print("Intercept:", intercept)
print("Coefficient:", coef)

# Plot the data and the model
plt.scatter(X, y, color='b')
plt.plot(X, model.predict(X), color='r')
plt.xlabel('X')
plt.ylabel('y')
plt.title('Linear Regression')
plt.show()
```

In this example, a random regression problem is generated using the "make_regression" function from Scikitlearn. A linear regression model is then fit to the data using the "LinearRegression" class from Scikitlearn. The parameters of the model, the intercept, and the coefficient, are then retrieved and printed. Finally, the data and the model are plotted using Matplotlib.

2.5 Definition of a Linear Transformation

A linear transformation (or linear map) is a function $T : V \to W$ between two vector spaces V and W that satisfies two properties:

1. Additivity: $T(\mathbf{u} + \mathbf{v}) = T(\mathbf{u}) + T(\mathbf{v})$ for all $\mathbf{u}, \mathbf{v} \in V$.
2. Scalar multiplication: $T(c\mathbf{v}) = cT(\mathbf{v})$ for all scalars c and all $\mathbf{v} \in V$.

2.6 Matrix Representation of Linear Transformations

Every linear transformation $T : \mathbb{R}^n \to \mathbb{R}^m$ can be represented by a matrix A, such that $T(\mathbf{x}) = A\mathbf{x}$ for all $\mathbf{x} \in \mathbb{R}^n$. The matrix A is called the matrix of the transformation T.

```python
import numpy as np

A = np.array([[2, 1], [1, 2]])
x = np.array([1, 2])

# Linear transformation
T_x = np.dot(A, x)
print(T_x)
```

2.7 Kernel and Range of a Linear Transformation

The kernel (or null space) of a linear transformation $T : V \to W$ is the set of all vectors $\mathbf{v} \in V$ such that $T(\mathbf{v}) = \mathbf{0}$, where $\mathbf{0}$ is the zero vector in W. It is denoted as $\ker(T)$ or $\text{null}(T)$.

The range (or image) of a linear transformation $T : V \to W$ is the set of all vectors $T(\mathbf{v})$ for all $\mathbf{v} \in V$. It is denoted as $\text{range}(T)$ or $\text{image}(T)$.

2.8 Applications in Machine Learning

Linear transformations and their properties are fundamental in many areas of machine learning:

- Principal Component Analysis (PCA): PCA is a dimensionality reduction technique that uses eigenvectors and eigenvalues of the covariance matrix to project data onto a lower-dimensional subspace. This involves computing the kernel and range of linear transformations.
- Neural Networks: The operation of each layer in a neural network involves a linear transformation (matrix multiplication) followed by a non-linear activation function.
- Support Vector Machines (SVM): The optimization problem in SVM involves finding the maximum margin hyperplane, which can be formulated as a problem involving linear transformations and their properties.

2.9 Eigenvalues and Eigenvectors

For a square matrix A, a non-zero vector $\mathbf{v}$ is an eigenvector of A with eigenvalue λ if

$$A\mathbf{v} = \lambda\mathbf{v}.$$

In this equation, $\mathbf{v}$ is an eigenvector of A, and λ is a scalar called the eigenvalue corresponding to $\mathbf{v}$. Eigenvalues and eigenvectors play a fundamental role in linear algebra and are widely used in various fields including machine learning, physics, engineering, and computer science.

2.10 Computing Eigenvalues and Eigenvectors

To find the eigenvalues of a matrix A, we need to solve the characteristic equation:

$$\det(A - \lambda I) = 0,$$

where I is the identity matrix of the same size as A. The solutions λ to this equation are the eigenvalues of A. Once the eigenvalues are found, the corresponding eigenvectors can be found by substituting each eigenvalue into the equation $(A - \lambda I)\mathbf{v} = \mathbf{0}$ and solving for $\mathbf{v}$.

Example Consider the matrix

$$A = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}.$$

The characteristic equation is

$$\det\left(\begin{pmatrix} 2-\lambda & 1 \\ 1 & 2-\lambda \end{pmatrix}\right) = 0.$$

Solving this equation yields the eigenvalues $\lambda = 1$ and $\lambda = 3$. The corresponding eigenvectors can be found by substituting each eigenvalue into the equation $(A - \lambda I)\mathbf{v} = \mathbf{0}$.

2.11 Applications of Eigenvalues and Eigenvectors in Machine Learning

Eigenvalues and eigenvectors have several applications in machine learning:

Principal Component Analysis (PCA) PCA is a dimensionality reduction technique that is widely used in machine learning. It involves finding the eigenvectors of the covariance matrix of the data, which correspond to the directions of maximum variance. The data can then be projected onto a lower-dimensional subspace spanned by a subset of the eigenvectors.

Spectral Clustering Spectral clustering is a technique used for clustering data into groups. It involves constructing a similarity graph of the data and finding the eigenvectors of the Laplacian matrix of the graph. The eigenvectors corresponding to the smallest eigenvalues are used to cluster the data.

Latent Semantic Analysis (LSA) LSA is a technique used in natural language processing for extracting the relationships between documents and terms. It involves computing the singular value decomposition (SVD) of the term-document matrix and using the eigenvectors corresponding to the largest singular values to represent the documents and terms in a lower-dimensional space.

2.12 NumPy and Matplotlib Example

NumPy is a library for the Python programming language that provides support for arrays (including matrices) and mathematical functions. Matplotlib is a plotting library for Python. Here is an example of how to compute the eigenvalues and eigenvectors of a matrix using NumPy and visualize them using Matplotlib:

```
import numpy as np
import matplotlib.pyplot as plt

# Define a matrix
A = np.array([[2, 1],
              [1, 2]])

# Compute the eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(A)

print("Eigenvalues:", eigenvalues)
print("Eigenvectors:\n", eigenvectors)

# Plot the eigenvectors
origin = [0, 0]
plt.quiver(*origin, eigenvectors[:,0],
  eigenvectors[:,1], color=['r','b'], scale=3)
plt.xlim(-1,1)
plt.ylim(-1,1)
plt.axhline(0, color='black',linewidth=0.5)
plt.axvline(0, color='black',linewidth=0.5)
plt.grid(color = 'gray', linestyle = '--', linewidth = 0.5)
plt.show()
```

In this example, the "np.linalg.eig" function is used to compute the eigenvalues and eigenvectors of the matrix A. The output will be the eigenvalues and eigenvectors of A. The eigenvectors are then plotted using Matplotlib's "quiver" function.

2.13 Introduction to Linear Systems

A system of linear equations is a collection of equations involving one or more variables, where each equation is linear in the variables. For example, the system of equations

$$\begin{aligned} 2x + 3y &= 5, \\ x - 4y &= -2, \end{aligned}$$

is a system of two linear equations in two variables, x and y. A solution to a system of linear equations is a set of values for the variables that satisfies all the equations in the system.

Relationship Between Linear Systems and Matrices A system of linear equations can be represented as a matrix equation $AX = B$, where A is the matrix of coefficients, X is the column matrix of variables, and B is the column matrix of constants. For example, the system of equations above can be written as

$$\begin{pmatrix} 2 & 3 \\ 1 & -4 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 5 \\ -2 \end{pmatrix}.$$

Solving the matrix equation $AX = B$ is equivalent to solving the original system of linear equations.

2.14 Methods for Solving Linear Systems

There are several methods for solving systems of linear equations, including substitution, elimination, and matrix methods.

Substitution Method The substitution method involves solving one of the equations for one variable in terms of the other variables and then substituting this expression into the other equations. This method is most effective for small systems of equations.

Elimination Method The elimination method involves adding or subtracting the equations in order to eliminate one of the variables. This method is most effective for systems of equations with two or three variables.

Matrix Methods Matrix methods involve using matrix operations to solve the matrix equation $AX = B$. There are several matrix methods for solving systems of linear equations, including Gaussian elimination, LU decomposition, and using the inverse of A.

2.15 Existence and Uniqueness of Solutions

The existence and uniqueness of solutions to a system of linear equations depends on the properties of the matrix A.

No Solution If the augmented matrix of a system of linear equations has a row of the form $[0 \cdots 0|b]$, where $b \neq 0$, then the system has no solution. This is because the corresponding equation, $0 = b$, has no solution.

Unique Solution If the matrix A is square and invertible, then the system of equations $AX = B$ has a unique solution, which can be found by multiplying both sides of the equation by A^{-1}:

$$X = A^{-1}B.$$

Infinitely Many Solutions If the matrix A is square and singular, and the augmented matrix of the system of equations has no rows of the form $[0 \cdots 0|b]$, where $b \neq 0$, then the system has infinitely many solutions.

2.16 Applications in Machine Learning

Solving systems of linear equations is a fundamental task in many machine learning algorithms, including linear regression, ridge regression, and support vector machines.

Linear Regression In linear regression, we try to find the best-fitting line for a set of data points. This involves solving the normal equations, which is a system of linear equations of the form $AX = B$, where A is the matrix of predictor variables, B is the column matrix of response variables, and X is the column matrix of regression coefficients.

Ridge Regression Ridge regression is a variation of linear regression that adds a regularization term to the cost function. This involves solving a system of linear equations of the form $AX = B$, where A is a modified version of the matrix of predictor variables, B is the column matrix of response variables, and X is the column matrix of regression coefficients.

Support Vector Machines Support vector machines (SVMs) are used for classification tasks. Training an SVM involves solving a quadratic programming problem, which can be reduced to solving a system of linear equations.

2.17 NumPy Examples

NumPy is a Python library that provides functions for working with arrays and matrices. Below are some examples of how to use NumPy to solve systems of linear equations.

```
import numpy as np

# Define the coefficient matrix A
# and the constant matrix B
A = np.array([[2, 3], [1, -4]])
B = np.array([5, -2])

# Use the numpy.linalg.solve function
# to solve the system of equations
X = np.linalg.solve(A, B)

print(X)
```

In this example, the "numpy.linalg.solve" function is used to solve the system of equations $AX = B$. The result is the column matrix X of variable values that satisfy the system of equations.

2.18 Gaussian Elimination

Gaussian elimination is a method for solving systems of linear equations. It involves three types of elementary row operations: 1. Swapping two rows, 2. Multiplying a row by a non-zero scalar, 3. Adding or subtracting the multiple of one row to another row.

The goal is to transform the augmented matrix of the system into its row echelon form and then into its reduced row echelon form. Once the matrix is in reduced row echelon form, the solutions to the system of equations can be read directly from the matrix.

Example Consider the system of equations:

$$2x + 3y = 8,$$
$$4x - 6y = -12.$$

The augmented matrix of this system is:

$$\left[\begin{array}{cc|c} 2 & 3 & 8 \\ 4 & -6 & -12 \end{array}\right].$$

Applying Gaussian elimination, we get:

$$\left[\begin{array}{cc|c} 1 & -3/2 & -6 \\ 0 & 3 & 12 \end{array}\right] \Rightarrow \left[\begin{array}{cc|c} 1 & -3/2 & -6 \\ 0 & 1 & 4 \end{array}\right] \Rightarrow \left[\begin{array}{cc|c} 1 & 0 & -6+3(4) \\ 0 & 1 & 4 \end{array}\right] = \left[\begin{array}{cc|c} 1 & 0 & 6 \\ 0 & 1 & 4 \end{array}\right].$$

So the solution to the system of equations is $x = 6$ and $y = 4$.

2.19 LU Decomposition

LU decomposition is a method for expressing a matrix A as the product of a lower triangular matrix L and an upper triangular matrix U. That is,

$$A = LU.$$

This decomposition is useful for solving systems of linear equations, computing the determinant, and performing rank determination. LU decomposition is essentially a modified form of Gaussian elimination.

Example Consider the matrix

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}.$$

The LU decomposition of A is

$$L = \begin{pmatrix} 1 & 0 \\ 3 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 1 & 2 \\ 0 & -2 \end{pmatrix}.$$

So $A = LU$.

2.20 Applications in Machine Learning

LU decomposition can be used in various applications in machine learning:

Solving Systems of Linear Equations Systems of linear equations arise frequently in machine learning problems. For example, the normal equations in linear regression form a system of linear equations that can be solved using LU decomposition.

Computing Determinants The determinant of a matrix is used in various machine learning applications, such as computing the inverse of a matrix, which is used in methods like linear regression, logistic regression, and support vector machines.

2.21 NumPy Example

NumPy is a library for the Python programming language that provides support for arrays (including matrices) and mathematical functions. Here is an example of how to perform LU decomposition using NumPy:

```
import numpy as np
from scipy.linalg import lu

# Define a matrix
A = np.array([[1, 2],
              [3, 4]])

# Perform LU decomposition
P, L, U = lu(A)

print("Matrix P:\n", P)
print("Matrix L:\n", L)
print("Matrix U:\n", U)

# Check that A = PLU
print("Product PLU:\n", P @ L @ U)
```

The output of this code will be the matrices P, L, U, and the product PLU, which should be equal to the original matrix A.

2.22 QR Decomposition

The QR decomposition of a matrix is a decomposition of the matrix into an orthogonal matrix and a triangular matrix. A matrix A can be decomposed as $A = QR$, where Q is an orthogonal matrix and R is an upper triangular matrix.

Computing QR Decomposition The QR decomposition can be computed using the Gram-Schmidt process or by using Householder reflections.

Example Consider the matrix

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}.$$

The QR decomposition of A can be computed using NumPy as follows:

```
import numpy as np

# Define a matrix
A = np.array([[1, 2],
              [3, 4]])

# Compute the QR decomposition
Q, R = np.linalg.qr(A)

print("Q:\n", Q)
print("R:\n", R)
```

2.23 Singular Value Decomposition (SVD)

The singular value decomposition of a matrix is a decomposition of the matrix into three matrices: an orthogonal matrix, a diagonal matrix, and another orthogonal matrix. A matrix A can be decomposed as $A = U\Sigma V^T$, where U and V are orthogonal matrices and Σ is a diagonal matrix.

Computing SVD The SVD can be computed using the 'np.linalg.svd' function in NumPy.

Example Consider the matrix A defined above. The SVD of A can be computed using NumPy as follows:

```
# Compute the SVD
U, Sigma, VT = np.linalg.svd(A)

print("U:\n", U)
print("Sigma:\n", Sigma)
print("VT:\n", VT)
```

2.24 Principal Component Analysis (PCA)

PCA is a dimensionality reduction technique that involves finding the eigenvectors of the covariance matrix of the data and projecting the data onto a lower-dimensional subspace spanned by a subset of the eigenvectors.

Computing PCA The PCA can be computed using the 'PCA' class in the 'sklearn.decomposition' module in scikit-learn.

Example Consider a dataset with two features. The PCA can be computed using scikit-learn as follows:

```
from sklearn.decomposition import PCA

# Generate some example data
np.random.seed(0)
X = np.random.randn(100, 2)

# Compute the PCA
pca = PCA(n_components=1)
X_pca = pca.fit_transform(X)
```

```
print("Explained variance ratio:", pca.explained_variance_ratio_)
```

Applications in Machine Learning PCA is widely used in machine learning for: - Data visualization: Reducing the dimensionality of the data to 2 or 3 dimensions for visualization. - Noise reduction: Reducing the dimensionality of the data to remove noise and improve the performance of other machine learning algorithms. - Feature extraction: Extracting new features from the data that can be used as input for other machine learning algorithms.

2.25 PageRank Algorithm

Introduction PageRank is an algorithm used by the Google search engine to rank webpages in its search results. It is named after one of the founders of Google, Larry Page. The algorithm was developed by Larry Page and Sergey Brin while they were graduate students at Stanford University.

The Algorithm The PageRank algorithm is based on the concept of link analysis and operates on the premise that the importance of a webpage can be determined by the number and quality of links to it. The algorithm assigns a rank to each webpage in a hyperlinked set, which represents the probability that a user, who is randomly clicking on links, will arrive at a particular page. The rank of a webpage is denoted by $PR(p)$ and is defined as follows:

$$PR(p) = \frac{1-d}{N} + d\sum_{i=1}^{n} \frac{PR(p_i)}{L(p_i)}$$

Where: - p is the webpage for which we are computing the PageRank. - N is the total number of webpages. - d is a damping factor, usually set to 0.85. - $p_1, p_2, ..., p_n$ are the webpages that link to p. - $L(p_i)$ is the number of outbound links on page p_i.

The Power Iteration Method The power iteration method is a simple and efficient method for finding the eigenvector of a matrix corresponding to its largest eigenvalue. It can be used to compute the PageRank of a set of webpages. The steps of the power iteration method are as follows:

1. Start with a random vector x_0.
2. Compute a new vector $x_{k+1} = Ax_k$.
3. Normalize x_{k+1}.
4. Repeat steps 2 and 3 until x_{k+1} converges.

Where A is the transition matrix of the webpages.

Applications The PageRank algorithm was originally developed for ranking webpages in search results. However, its applications have extended to various other fields such as social networks, citation networks, recommendation systems, and many more.

The PageRank algorithm is a powerful tool for ranking items in a network based on their importance. It has revolutionized the way search engines work and has found applications in many other fields.

Chapter 3

Functions, Limits, and Derivatives

3.1 Functions

A function is a rule that assigns to each element in one set (the domain) exactly one element in another set (the codomain). Functions are often represented by equations, tables, or graphs.
Common Functions in Calculus and Machine Learning - Linear functions: $f(x) = mx + b$ - Quadratic functions: $f(x) = ax^2 + bx + c$ - Exponential functions: $f(x) = a^x$ - Logarithmic functions: $f(x) = \log_a(x)$ - Sigmoid functions: $f(x) = \frac{1}{1+e^{-x}}$

3.2 Limits

The limit of a function at a certain point refers to the value that the function approaches as its input (or variable) approaches that point.

$$\lim_{x \to a} f(x) = L$$

This means that for each $\varepsilon > 0$ there exists a $\delta > 0$ such that if $0 < |x - a| < \delta$ then $|f(x) - L| < \varepsilon$.
Example Compute the limit of the function $f(x) = x^2$ as x approaches 2.

$$\lim_{x \to 2} x^2 = 2^2 = 4$$

3.3 Derivatives

The derivative of a function of a real variable measures the sensitivity to change of the function value (output value) with respect to a change in its argument (input value).

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Example Compute the derivative of the function $f(x) = x^2$.

$$f'(x) = \lim_{h \to 0} \frac{(x+h)^2 - x^2}{h} = \lim_{h \to 0} \frac{x^2 + 2xh + h^2 - x^2}{h} = \lim_{h \to 0} 2x + h = 2x$$

3.4 Applications in Machine Learning

The concept of derivatives is fundamental in machine learning. For example, gradient descent, an optimization algorithm often used for finding the minimum of a function, is based on the derivative of the function.
Example: Gradient Descent Gradient descent is an iterative optimization algorithm for finding the minimum of a function. To implement gradient descent, one needs to compute the derivative of the function.

Here is a Python code example that uses NumPy to implement gradient descent for the function $f(x) = x^2$:

```
import numpy as np

def gradient_descent(gradient, start,
    learn_rate, n_iter):
    vector = start
    for _ in range(n_iter):
        diff = -learn_rate * gradient(vector)
        vector += diff
    return vector

def f(x):
    return x ** 2

def f_prime(x):
    return 2 * x

start = 10
learn_rate = 0.1
n_iter = 100

minima = gradient_descent(f_prime,
    start, learn_rate, n_iter)
print(minima)
```

This will output a value close to 0, which is the minimum of the function $f(x) = x^2$.

3.5 Plotting Functions

Matplotlib is a plotting library for Python. Here is an example of how to use Matplotlib to plot the function $f(x) = x^2$ and its derivative $f'(x) = 2x$.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-10, 10, 100)
y = x ** 2
y_prime = 2 * x

plt.plot(x, y, label='f(x) = x^2')
plt.plot(x, y_prime, label="f'(x) = 2x")
plt.legend()
plt.xlabel('x')
plt.ylabel('y')
plt.title('Plot of f(x) and f\'(x)')
plt.grid(True)
plt.show()
```

This will display a plot of the function $f(x) = x^2$ and its derivative $f'(x) = 2x$.

3.6 Integration

Integration is one of the two main operations in calculus, the other being differentiation. The integral of a function can be thought of as the area under the curve of the function. More formally, the integral of a function $f(x)$ from a to b is the limit of the Riemann sums as the number of subintervals approaches infinity.

Properties of Integrals Some important properties of integrals are:

- Linearity: $\int [f(x) + g(x)]\,dx = \int f(x)\,dx + \int g(x)\,dx$
- Constant Multiple: $\int cf(x)\,dx = c\int f(x)\,dx$
- Power Rule: $\int x^n\,dx = \frac{x^{n+1}}{n+1} + C$, for $n \neq -1$

Computing Integrals To compute the integral of a function, we can use various techniques such as substitution, integration by parts, or using standard integral tables.

Example Compute the integral of $f(x) = x^2$ from 0 to 1.

Solution:

$$\begin{aligned}\int_0^1 x^2\,dx &= \left[\frac{x^3}{3}\right]_0^1 \\ &= \frac{1^3}{3} - \frac{0^3}{3} \\ &= \frac{1}{3}\end{aligned}$$

3.7 Definite Integrals

The definite integral of a function $f(x)$ from a to b is the signed area between the curve of the function and the x-axis from $x = a$ to $x = b$. It is denoted by $\int_a^b f(x)\,dx$ and is computed using the fundamental theorem of calculus:

$$\int_a^b f(x)\,dx = F(b) - F(a)$$

where $F(x)$ is an antiderivative of $f(x)$.

Properties of Definite Integrals Some important properties of definite integrals are:

- Linearity: $\int_a^b [f(x) + g(x)]\,dx = \int_a^b f(x)\,dx + \int_a^b g(x)\,dx$
- Constant Multiple: $\int_a^b cf(x)\,dx = c\int_a^b f(x)\,dx$

Computing Definite Integrals To compute a definite integral, we can find an antiderivative of the function and then apply the fundamental theorem of calculus.

Example Compute the definite integral of $f(x) = x^2$ from 0 to 1.

Solution:

$$\begin{aligned}\int_0^1 x^2\,dx &= \left[\frac{x^3}{3}\right]_0^1 \\ &= \frac{1^3}{3} - \frac{0^3}{3} \\ &= \frac{1}{3}\end{aligned}$$

3.8 Applications in Machine Learning

Integration and definite integrals play a crucial role in machine learning, especially in the computation of probabilities and in the optimization of functions. For example, the area under the curve of a probability density function (PDF) gives the probability of a random variable falling within a certain range. This is essential in various machine learning tasks such as classification and regression.

3.9 Python and NumPy Examples

NumPy is a powerful library for numerical computing in Python. Below is an example of how to use NumPy and Matplotlib to compute and visualize the integral of a function.

```
import numpy as np
import matplotlib.pyplot as plt

# Define the function
def f(x):
    return x**2

# Compute the integral
x = np.linspace(0, 1, 100)
y = f(x)
```

```
integral = np.trapz(y, x)

print("Integral:", integral)

# Plot the function and the area under the curve
plt.plot(x, y, 'r')
plt.fill_between(x, y, alpha=0.2)
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Integral of f(x) = x^2')
plt.show()
```

This will output:

```
Integral: 0.33335033840084355
```

And display a plot of the function $f(x) = x^2$ and the area under the curve from 0 to 1.

3.10 Partial Derivatives

A partial derivative of a function of several variables is its derivative with respect to one of those variables, keeping the others constant.
Example The partial derivatives of the function $f(x, y) = x^2 + y^2$ are:

$$\frac{\partial f}{\partial x} = 2x, \quad \frac{\partial f}{\partial y} = 2y.$$

3.11 Gradients

The gradient of a function of several variables is a vector containing all of its partial derivatives.
Example The gradient of the function $f(x, y) = x^2 + y^2$ is:

$$\nabla f = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right] = [2x, 2y].$$

3.12 Optimization

Optimization involves finding the maximum or minimum of a function. In machine learning, we often want to find the minimum of a cost function in order to optimize a model.
Example Consider the cost function $J(\theta) = (1/2)(h_\theta(x) - y)^2$ where $h_\theta(x)$ is a hypothesis function. To minimize this cost function, we can use gradient descent, an optimization algorithm that involves iteratively updating the parameter θ in the direction of the negative gradient of the cost function.

3.13 Applications in Machine Learning

Partial derivatives and gradients are fundamental to optimization algorithms in machine learning, such as gradient descent. Optimization is a crucial step in training machine learning models because it helps to find the optimal parameters of the model that minimize the cost function.

3.14 Python, NumPy, and Matplotlib Examples

Below is an example of how to use Python, NumPy, and Matplotlib to compute partial derivatives and gradients and visualize a function and its gradient.

```
import numpy as np
import matplotlib.pyplot as plt

# Define the function
def f(x, y):
    return x**2 + y**2
```

```
# Define the partial derivatives
def df_dx(x, y):
    return 2*x

def df_dy(x, y):
    return 2*y

# Define the gradient
def gradient(x, y):
    return np.array([df_dx(x, y), df_dy(x, y)])

# Create a grid of points
x = np.linspace(-10, 10, 100)
y = np.linspace(-10, 10, 100)
X, Y = np.meshgrid(x, y)

# Compute the function values
Z = f(X, Y)

# Compute the gradient values
Gx, Gy = np.gradient(Z, axis=(0, 1))

# Plot the function
plt.contourf(X, Y, Z, levels=100, cmap='viridis')
plt.colorbar(label='f(x,-y)')
plt.quiver(X[::5, ::5], Y[::5, ::5], Gx[::5, ::5],
  Gy[::5, ::5], color='white')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Function-and-Gradient')
plt.show()
```

This code will plot the function $f(x, y) = x^2+y^2$ and its gradient. The function is plotted as a contour plot and the gradient is plotted as a quiver plot on top of the function.

3.15 Composite Functions and Chain Rule

A composite function is a function composed of two other functions, i.e., $f(g(x))$. The chain rule is a fundamental tool in calculus for computing the derivative of composite functions.

Chain Rule The chain rule states that if f and g are functions, then the derivative of the composite function $f(g(x))$ with respect to x is

$$\frac{d}{dx}f(g(x)) = f'(g(x))g'(x).$$

Example Consider the composite function $f(g(x)) = \sqrt{1+x^2}$. To compute the derivative of this function with respect to x, we can use the chain rule:

$$\frac{d}{dx}\sqrt{1+x^2} = \frac{1}{2\sqrt{1+x^2}} \cdot 2x = \frac{x}{\sqrt{1+x^2}}.$$

3.16 Role of Gradients in Optimization

In machine learning, we often need to optimize a loss function to train a model. The gradient of the loss function with respect to the model parameters provides the direction of the steepest ascent of the function. To minimize the function, we update the parameters in the opposite direction of the gradient. This process is called gradient descent.

Example Consider a simple linear regression problem where we have a single feature x and a target variable y. The model is $y = mx+b$, where m is the slope and b is the intercept. The loss function is the mean squared error (MSE) between the predicted and actual values of y:

$$L(m, b) = \frac{1}{n}\sum_{i=1}^{n}(y_i - (mx_i + b))^2.$$

The gradients of the loss function with respect to m and b are

$$\frac{\partial L}{\partial m} = -\frac{2}{n}\sum_{i=1}^{n} x_i(y_i - (mx_i + b)),$$

$$\frac{\partial L}{\partial b} = -\frac{2}{n}\sum_{i=1}^{n} (y_i - (mx_i + b)).$$

To minimize the loss function, we update the parameters m and b in the opposite direction of the gradients:

$$m := m - \alpha\frac{\partial L}{\partial m},$$

$$b := b - \alpha\frac{\partial L}{\partial b},$$

where α is the learning rate.

3.17 Introduction to Gradient Descent

Gradient descent is an optimization algorithm used to find the minimum of a function. Given a function defined by a set of parameters, gradient descent starts with an initial set of parameter values and iteratively moves towards a set of parameter values that minimize the function. This iterative minimization is achieved by taking steps in the negative direction of the function's gradient.

Geometric Interpretation The geometric interpretation of gradient descent involves visualizing the function as a surface in a high-dimensional space. The gradient of the function at a particular point is a vector that points in the direction of the steepest ascent at that point. Therefore, to find the minimum of the function, one should move in the direction of the steepest descent, which is the negative of the gradient.

3.18 Variations of Gradient Descent

There are several variations of gradient descent, each with its own advantages and disadvantages.

Batch Gradient Descent In batch gradient descent, the entire dataset is used to compute the gradient of the cost function with respect to the parameters. The parameters are then updated in the direction of the negative gradient.

Stochastic Gradient Descent In stochastic gradient descent (SGD), a single training example is used to compute the gradient and update the parameters. This leads to faster convergence but with more noise in the parameter updates.

Mini-batch Gradient Descent In mini-batch gradient descent, a small random sample of the training data, called a mini-batch, is used to compute the gradient and update the parameters. This is a compromise between batch gradient descent and SGD that aims to balance the computational efficiency of batch gradient descent with the faster convergence of SGD.

3.19 Convergence and Optimization

The convergence of gradient descent is affected by several factors, such as the learning rate and the initial guess for the parameters. A learning rate that is too high may cause the algorithm to oscillate or diverge, while a learning rate that is too low may cause the algorithm to converge too slowly.

Optimizing the Learning Rate The learning rate is a hyperparameter that controls the size of the steps taken during the optimization process. It can be optimized using techniques such as grid search or random search.

Other Optimization Techniques There are several other techniques that can be used to optimize the convergence of gradient descent, such as momentum, RMSprop, and Adam. These techniques use different strategies to adapt the learning rate during the optimization process.

3.20 Applications in Machine Learning

Gradient descent and its variations are widely used in machine learning to optimize the parameters of models, such as linear regression, logistic regression, and neural networks.

3.21 Python, NumPy, and Matplotlib Examples

Below is an example of how to use Python, NumPy, and Matplotlib to implement and visualize the gradient descent algorithm.

```
import numpy as np
import matplotlib.pyplot as plt

# Generate some example data
np.random.seed(0)
X = 2 * np.random.rand(100, 1)
y = 3 + 4 * X + np.random.randn(100, 1)

# Add a column of ones to X
X_b = np.c_[np.ones((100, 1)), X]

# Compute the normal equation
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)

# Define the cost function
def compute_cost(theta, X, y):
    m = len(y)
    cost = (1/2*m) * np.sum(np.square(X.dot(theta) - y))
    return cost

# Define the gradient descent function
def gradient_descent(X, y, theta, learning_rate, num_iterations):
    m = len(y)
    cost_history = np.zeros(num_iterations)
    for i in range(num_iterations):
        theta = theta - (1/m) * learning_rate * (X.T.dot(X.dot(theta)
        cost_history[i] = compute_cost(theta, X, y)
    return theta, cost_history

# Set the hyperparameters
learning_rate = 0.1
num_iterations = 1000

# Initialize theta to zero
theta = np.zeros((2,1))

# Run the gradient descent algorithm
theta, cost_history = gradient_descent(X_b, y, theta, learning_rate, n

# Plot the cost function
plt.plot(range(1, num_iterations+1), cost_history, color='b')
plt.rcParams["figure.figsize"] = (10,6)
plt.grid()
plt.xlabel('Number of iterations')
plt.ylabel('Cost')
plt.title('Convergence of gradient descent')
plt.show()
```

In this example, some random data is generated, and then the gradient descent algorithm is used to fit a linear model to the data. The cost function is plotted to show the convergence of the algorithm.

3.22 Applications in Machine Learning

Gradients are fundamental to many optimization algorithms used in machine learning, such as gradient descent and its variants (e.g., stochastic gradient descent, mini-batch gradient descent, and adaptive gradient algorithms like Adam). These algorithms are used to train various models, including linear regression, logistic regression, and neural networks.

3.23 Python, NumPy, and Matplotlib Examples

Below is an example of how to use Python, NumPy, and Matplotlib to implement and visualize gradient descent for a simple linear regression problem.

```
import numpy as np
import matplotlib.pyplot as plt

# Generate synthetic data
np.random.seed(0)
x = 2 * np.random.rand(100, 1)
y = 3 + 4 * x + np.random.randn(100, 1)

# Compute the gradients of the loss function
def compute_gradients(x, y, m, b):
    n = len(x)
    y_pred = m * x + b
    error = y - y_pred
    grad_m = -2/n * np.sum(x * error)
    grad_b = -2/n * np.sum(error)
    return grad_m, grad_b

# Update the parameters
def update_parameters(m, b, grad_m,
     grad_b, learning_rate):
    m -= learning_rate * grad_m
    b -= learning_rate * grad_b
    return m, b

# Gradient descent
def gradient_descent(x, y, m, b,
    learning_rate, n_iterations):
    for i in range(n_iterations):
        grad_m, grad_b =
              compute_gradients(x, y, m, b)
        m, b = update_parameters(m, b,
            grad_m, grad_b, learning_rate)
    return m, b

# Initial parameters
m = np.random.randn()
b = np.random.randn()

# Learning rate and number of iterations
learning_rate = 0.1
n_iterations = 1000

# Run gradient descent
m, b = gradient_descent(x, y, m, b,
       learning_rate, n_iterations)

# Plot the data and the regression line
plt.scatter(x, y)
plt.plot(x, m*x+b, color='red')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

This will generate a plot of the synthetic data and the re-

gression line obtained by gradient descent.

3.24 Automatic Differentiation and Backpropagation

Automatic differentiation is a set of techniques for numerically evaluating the derivative of a function specified by a computer program. Backpropagation is a specific application of automatic differentiation to neural networks.

Backpropagation Backpropagation is a widely used algorithm for training feedforward artificial neural networks. It computes the gradient of the loss function with respect to each weight by applying the chain rule, computing the gradient one layer at a time, iterating backward from the last layer to avoid redundant calculations of intermediate terms in the chain rule.

3.25 Detailed Explanation

The backpropagation algorithm consists of two phases: the forward pass and the backward pass.

Forward Pass In the forward pass, the input is passed through the network, and the output of each layer is computed. This is done by multiplying the input of each layer with the weights and then passing it through an activation function.

Backward Pass In the backward pass, the error is computed by taking the difference between the predicted output and the actual output. This error is then propagated backward through the network to update the weights. The weights are updated in a way to minimize the error.

The weight update is done using the gradient descent algorithm. The gradient of the error with respect to each weight is computed using the chain rule. The weights are then updated by subtracting a fraction of the gradient from the current weights.

3.26 Mathematical Formulation

Let L be the loss function, y be the actual output, and $\hat{y}$ be the predicted output. The error E is then given by:

$$E = L(y, \hat{y})$$

The goal is to minimize this error by updating the weights W. The weights are updated using the gradient descent algorithm:

$$W = W - \alpha \frac{\partial E}{\partial W}$$

Where α is the learning rate.

The gradient of the error with respect to the weights is computed using the chain rule:

$$\frac{\partial E}{\partial W} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial W}$$

3.27 Applications in Machine Learning

Backpropagation is widely used in machine learning for training artificial neural networks. It is used in applications such as image recognition, speech recognition, and natural language processing.

3.28 Python, NumPy, and Matplotlib Example

Below is a simple Python example using NumPy and Matplotlib to implement the backpropagation algorithm.

```
import numpy as np
import matplotlib.pyplot as plt

# Define the sigmoid activation function and its derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

# Define the dataset and the expected output
inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
expected_output = np.array([[0], [1], [1], [0]])

# Define the parameters of the network
inputLayerNeurons, hiddenLayerNeurons, outputLayerNeurons = 2, 2, 1

# Initialize the weights and biases
hidden_weights = np.random.uniform(size=(inputLayerNeurons, hiddenLayerNeurons))
hidden_bias = np.random.uniform(size=(1, hiddenLayerNeurons))
hidden_layer_activation = np.dot(inputs, hidden_weights)
hidden_layer_activation += hidden_bias
hidden_layer_output = sigmoid(hidden_layer_activation)

output_weights = np.random.uniform(size=(hiddenLayerNeurons,outputLayerNeurons))
output_bias = np.random.uniform(size=(1,outputLayerNeurons))
output_layer_activation = np.dot(hidden_layer_output, output_weights)
output_layer_activation += output_bias
output = sigmoid(output_layer_activation)

# Define the learning rate and the number of epochs
learning_rate = 0.1
epochs = 10000
```

```
# Train the network
for epoch in range(epochs):
    # Forward pass
    hidden_layer_activation = np.dot(inputs, hidden_weights)
    hidden_layer_activation += hidden_bias
    hidden_layer_output = sigmoid(hidden_layer_activation)

    output_layer_activation = np.dot(hidden_layer_output, output_weights)
    output_layer_activation += output_bias
    predicted_output = sigmoid(output_layer_activation)

    # Compute the error
    error = expected_output - predicted_output

    # Backward pass
    d_predicted_output = error * sigmoid_derivative(predicted_output)
    error_hidden_layer = d_predicted_output.dot(output_weights.T)
    d_hidden_layer = error_hidden_layer * sigmoid_derivative(hidden_layer_output)

    # Update the weights and biases
    output_weights += hidden_layer_output.T.dot(d_predicted_output) * learning_rate
    output_bias += np.sum(d_predicted_output, axis=0, keepdims=True) * learning_rate
    hidden_weights += inputs.T.dot(d_hidden_layer) * learning_rate
    hidden_bias += np.sum(d_hidden_layer, axis=0, keepdims=True) * learning_rate

# Plot the predicted output
plt.plot(predicted_output)
plt.show()
```

This code will train a neural network with one hidden layer to solve the XOR problem. The predicted output will be plotted using Matplotlib.

Chapter 4

Linear Regression and the Normal Equations

4.1 Introduction to Linear Regression

Linear regression is a method for modeling the relationship between a dependent variable y and one or more independent variables X. The objective of linear regression is to find the best-fitting line, plane, or hyperplane that minimizes the sum of the squared residuals (the differences between the observed and predicted values).

The least squares criterion is a common way to find the best-fitting line. It involves minimizing the sum of the squared residuals, which can be written as:

$$\min_{\beta} \sum_{i=1}^{n} (y_i - (X_i \beta))^2$$

where y_i is the observed value, X_i is the vector of independent variables, and β is the vector of parameters to be estimated.

4.2 The Normal Equations

The normal equations are a way to solve the least squares problem. They can be derived by taking the derivative of the sum of the squared residuals with respect to β and setting it to zero. This yields the following equation:

$$X^T X \beta = X^T y$$

Where X is the matrix of independent variables, y is the vector of dependent variables, and β is the vector of parameters to be estimated.

The solution for β can be found by multiplying both sides by the inverse of $X^T X$:

$$\beta = (X^T X)^{-1} X^T y$$

This is the closed-form solution for β and can be computed directly. However, it involves computing the inverse of a matrix, which can be computationally expensive for large datasets.

The normal equations are related to matrix factorization and the pseudo-inverse. The matrix $X^T X$ is a positive semidefinite matrix, and its inverse can be computed using the Cholesky decomposition or the singular value decomposition (SVD). The pseudo-inverse of X, denoted by X^+, is defined as:

$$X^+ = (X^T X)^{-1} X^T$$

So the solution for β can also be written as:

$$\beta = X^+ y$$

4.3 Computational Considerations

Solving the normal equations involves computing the inverse of a matrix, which has a computational complexity of $O(n^3)$, where n is the number of independent variables. This can be

prohibitive for large datasets or datasets with a large number of independent variables.

Alternatives for large datasets include using gradient descent or stochastic gradient descent, which have a computational complexity of $O(n^2)$ but may require multiple iterations to converge to the solution.

4.4 Applications in Machine Learning

Linear regression is widely used in machine learning for prediction and inference. It can be used to predict the value of a dependent variable based on the values of the independent variables. It can also be used to infer the relationships between the independent variables and the dependent variable, and to select the most important independent variables.

4.5 Python, NumPy, and Matplotlib Examples

NumPy is a powerful library for numerical computing in Python. Matplotlib is a library for creating visualizations in Python. Below is an example of how to use NumPy and Matplotlib to compute the parameters of a linear regression model and plot the results.

```
import numpy as np
import matplotlib.pyplot as plt

# Generate some example data
np.random.seed(0)
X = 2 * np.random.rand(100, 1)
y = 3 + 4 * X + np.random.randn(100, 1)

# Add a column of ones to X
X_b = np.c_[np.ones((100, 1)), X]

# Compute the parameters using the normal equations
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(

# Make predictions
X_new = np.array([[0], [2]])
X_new_b = np.c_[np.ones((2, 1)), X_new]
y_predict = X_new_b.dot(theta_best)

# Plot the results
plt.plot(X_new, y_predict, "r-")
plt.plot(X, y, "b.")
plt.axis([0, 2, 0, 15])
plt.show()
```

This will output a plot of the observed data and the best-fitting line computed using the normal equations.

Chapter 5

Principal Component Analysis (PCA)

5.1 Introduction to PCA

Principal Component Analysis (PCA) is a dimensionality reduction technique that transforms a set of correlated variables into a new set of uncorrelated variables called principal components. The first principal component accounts for the most variance in the data, the second principal component accounts for the second most variance, and so on. The geometric interpretation of PCA is that it finds the directions of maximum variance in the data.

5.2 Computing the Principal Components

To compute the principal components of a dataset, the following steps are performed:

1. Center the data by subtracting the mean of each feature from the data points.
2. Compute the covariance matrix of the centered data.
3. Compute the eigenvalues and eigenvectors of the covariance matrix.
4. Sort the eigenvectors by the magnitude of their corresponding eigenvalues in descending order.
5. Select the top k eigenvectors to form a matrix W.
6. Project the centered data onto W to get the principal components.

5.3 Applications of PCA

PCA has several applications in machine learning:

- Dimensionality Reduction: Reducing the number of features in the data while retaining as much information as possible.
- Data Visualization: Projecting the data into a lower-dimensional space (e.g., 2D or 3D) for visualization.
- Noise Reduction: Removing noise from the data by projecting it onto a lower-dimensional subspace.

5.4 Python, NumPy, and Matplotlib Examples

Below is an example of how to use Python, NumPy, and Matplotlib to compute the principal components of a dataset and visualize the results.

```
import numpy as np
import matplotlib.pyplot as plt

# Generate some example data
np.random.seed(0)
X = np.dot(np.random.rand(2, 2),
       np.random.randn(2, 200)).T

# Center the data
mean = np.mean(X, axis=0)
X_centered = X - mean

# Compute the covariance matrix
covariance_matrix = np.cov(X_centered.T)

# Compute the eigenvalues and eigenvectors
eigenvalues, eigenvectors
    = np.linalg.eig(covariance_matrix)

# Sort the eigenvectors by the
    magnitude of their corresponding eigenvalues
sorted_indices = np.argsort(-eigenvalues)
eigenvectors_sorted
   = eigenvectors[:, sorted_indices]
eigenvalues_sorted = eigenvalues[sorted_indices]

# Select the top k eigenvectors
k = 2
W = eigenvectors_sorted[:, :k]

# Project the data onto W
X_pca = X_centered.dot(W)

# Plot the original data
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.scatter(X[:, 0], X[:, 1])
plt.title('Original Data')

# Plot the projected data
plt.subplot(1, 2, 2)
plt.scatter(X_pca[:, 0], X_pca[:, 1])
plt.title('Projected Data')
plt.show()
```

In this example, the original data is 2D, so there is no need to reduce the dimensionality. However, the same approach can be used for higher-dimensional data. The left plot shows the original data, and the right plot shows the data projected onto the two principal components.