# Calculus For Data Science

## Functions & Variables

### What is a Function?

- A **function** is a rule that maps input(s) to output(s).
- Formally: $f: X \to Y$ where each $x \in X$ gives one output $y \in Y$.
- Example: $f(x) = 2x + 3$ → when $x = 2$, output is $y = 7$.

In ML, the **model itself is a function**:

- Example: `y_pred = f(features)`.
- A neural network is just a **complex function** learned from data.

### Independent vs Dependent Variables

- **Independent variable (x):** input feature we control/measure.
- **Dependent variable (y):** output that depends on x.

**Examples in Data Science**:

- Predicting house prices:
  - Independent variables: size, location, number of rooms.
  - Dependent variable: price.
- Predicting exam score:
  - Independent variable: study hours.
  - Dependent variable: score.

### Python Example

```
def house_price(size):
   return 50000 + (3000 * size)  # simple linear relation

print(house_price(10))  # 80,000
```

## Types of Graphs & Their Interpretations

### Common Graph Shapes

1. **Linear (y = mx + c):** straight line, proportional growth.
   - ML: Linear regression.
2. **Quadratic ($y = x^2$):** U-shape parabola.
   - ML: Loss functions are often quadratic.
3. **Exponential (y = e^x):** rapid growth/decay.
   - ML: Learning rate decay, population models.
4. **Sigmoid:**
   - Formula: `1 / (1 + e^-x)`
   - S-shaped curve between 0 and 1.
   - ML: Used in logistic regression & NN activation functions.
5. **Logarithmic:**
   - Formula: y = log(x).
   - Slow growth → useful in information theory (entropy).

### Python Plot Example

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

x = np.linspace(-5, 5, 100)
plt.plot(x, np.exp(x), label="Exponential")
plt.plot(x, 1/(1+np.exp(-x)), label="Sigmoid")
```

plt.legend(); plt.show()

## Limits, Continuity & Chain Rule

### Limits

- A **limit** finds the value a function approaches as input gets closer to some point.
- Example:
  `lim (x→2) (x² + 1) = 5`.

### Continuity

- A function is continuous if there are **no jumps, holes, or asymptotes**.
- Example: `f(x) = 1/x` is discontinuous at x = 0.

### Chain Rule Basics

- If y = f(g(x)), then:
  `dy/dx = f'(g(x)) * g'(x)`.
- Example: $y = (x^2 + 1)^3$
  - Outer function: $u^3$
  - Inner function: $u = x^2+1$
  - Derivative: $3(x^2+1)^2 * (2x) = 6x(x^2+1)^2$.

**In ML**: Chain rule is the backbone of **backpropagation**.

### Visualizing Discontinuity

```
x = np.linspace(-2, 2, 100)
y = 1/x
plt.plot(x, y); plt.axvline(0, color="r", linestyle="--")
plt.show()
```

# Derivatives & Chain Rule Applications

## Basic Derivative Rules

- Power rule: $d/dx\ (x^n) = n \cdot x^{(n-1)}$
- Sum rule: $d/dx\ (f+g) = f' + g'$
- Product rule: $(uv)' = u'v + uv'$
- Quotient rule: $(u/v)' = (u'v - uv')/v^2$

## Chain Rule in Action (ML)

- Neural network = composition of functions.
- Example:
  `y = σ(w·x + b)` → σ = sigmoid.
- Derivative requires applying **chain rule** for weight updates.

# Loss & Cost Functions in ML

## Definitions

- **Loss:** error for a single data point.
- **Cost:** average loss across dataset.

## Common Functions

1. **MSE (Mean Squared Error):**
   - Formula: $(1/n)\ \Sigma\ (y\_pred - y\_true)^2$
   - Penalizes large errors heavily.
2. **MAE (Mean Absolute Error):**
   - Formula: $(1/n)\ \Sigma\ |y\_pred - y\_true|$

- Less sensitive to outliers.

### Graphs

- MSE → smooth parabola.
- MAE → "V" shape.

### Python Example

```
import numpy as np
y_true = np.array([3, -0.5, 2])
y_pred = np.array([2.5, 0.0, 2])

mse = np.mean((y_true - y_pred)**2)
mae = np.mean(np.abs(y_true - y_pred))
print("MSE:", mse, "MAE:", mae)
```

## Gradient Descent

### Intuition

- We want to find **minimum of cost function**.
- Idea: take steps proportional to slope (gradient).
- Update rule:
  `θ_new = θ_old - α * ∇J(θ)`
    - $\alpha$ = learning rate.
    - $\nabla J$ = gradient (slope).

### Example

- Cost: $J(\theta) = \theta^2$
- Gradient: $dJ/d\theta = 2\theta$
- Update: $\theta = \theta - \alpha \cdot 2\theta$

## Multivariable Calculus & ML Integration

### Partial Derivatives

- For $f(x,y) = x^2 + y^2$:
    - $\partial f/\partial x = 2x$
    - $\partial f/\partial y = 2y$

### Gradient Vector

- $\nabla f(x,y) = [2x, 2y]$
- Points in **steepest ascent** direction.

### ML Connection

- Each weight in a neural network has its own gradient.
- Backpropagation computes gradient vector for all weights.
- Gradient Descent uses these to minimize cost function.

# NumPy & Pandas

## NumPy (Numerical Python)

### 1. Introduction

- NumPy = **Numerical Python**
- Provides support for **multi-dimensional arrays (ndarrays)**.
- Faster than Python lists because it uses **C-based implementations**.

# import numpy as np

## 2. Creating Arrays

### 1D Array (Vector)

```
x = np.array([12, 14, 23, 45, 105])
print(x)
print(type(x))
```

Output: NumPy array `[12 14 23 45 105]` of type `numpy.ndarray`.

### 2D Array (Matrix)

```
y = np.array([
    [1, 2, 3],
    [7, 5.8, 8],
    [4, 6, 7]
])
print(y)
```

Output: a 3x3 matrix.
Mixed int and float → all converted to `float64`.

### Special Arrays

```
np.zeros((2, 3))     # 2x3 array of zeros
np.ones((3, 3))      # 3x3 array of ones
np.eye(3)            # Identity matrix
np.arange(0, 10, 2)  # [0, 2, 4, 6, 8]
np.linspace(0, 1, 5) # [0. , 0.25, 0.5, 0.75, 1. ]
```

## 3. Array Attributes

```
print(x.shape)   # (5,) → 1D array with 5 elements
print(x.ndim)    # 1 → 1D
print(x.dtype)   # int64
```

## 4. Indexing & Slicing

### 1D Array

```
x[0]     # first element
x[-1]    # last element
x[1:4]   # slice → elements from index 1 to 3
```

### 2D Array

```
print(y[0, 1])   # element at row 0, col 1
print(y[:, 2])   # all rows, 3rd column
print(y[1, :])   # entire 2nd row
```

## 5. Array Operations

### Arithmetic

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

print(a + b)  # [5 7 9]
print(a * b)  # [ 4 10 18] (element-wise)
print(a ** 2) # [1 4 9]
```

### Broadcasting

```
print(a + 10)  # [11 12 13]
```

## 6. Reshaping & Flattening

```
arr = np.arange(1, 13)   # [1, 2, ..., 12]
arr2 = arr.reshape(3, 4) # 3x4 matrix
arr2.flatten()           # back to 1D
```

## 7. Aggregate Functions

```
print(np.sum(arr2))    # sum of all elements
print(np.mean(arr2))   # average
print(np.max(arr2))    # maximum
print(np.min(arr2))    # minimum
print(np.std(arr2))    # standard deviation
```

# Pandas

## 1. Introduction

- Built on top of NumPy.
- Provides two main data structures:

- **Series** (1D labeled array)
- **DataFrame** (2D table: rows + columns)

```
# import pandas as pd
```

### 2. Pandas Series

```
s = pd.Series([10, 20, 30, 40], index=['a', 'b', 'c', 'd'])
print(s)
```

Series = like a column in Excel with labels (index).

Accessing elements:

```
print(s['b'])   # 20
print(s[2])     # 30
```

### 3. Pandas DataFrame

```
data = {
    "Name": ["Alice", "Bob", "Charlie"],
    "Age": [25, 30, 35],
    "Salary": [50000, 60000, 70000]
}
df = pd.DataFrame(data)
print(df)
```

Creates a table with columns Name, Age, Salary.

### 4. DataFrame Operations

```
print(df.head())    # first 5 rows
print(df.tail())    # last 5 rows
print(df.info())    # summary
print(df.describe())# statistics summary
```

### 5. Selecting Data

```
print(df["Name"])        # select column
print(df[["Name","Age"]])# multiple columns
print(df.iloc[0])        # row by index (first row)
```

```
print(df.loc[0,"Salary"])# specific cell
```

## 6. Filtering Data

```
print(df[df["Age"] > 28])  # filter rows
```

## 7. Adding & Removing Columns

```
df["Bonus"] = df["Salary"] * 0.1
df.drop("Bonus", axis=1, inplace=True)
```

## 8. Handling Missing Data

```
df2 = pd.DataFrame({
    "A": [1, 2, np.nan],
    "B": [4, np.nan, 6]
})

print(df2.fillna(0))    # replace NaN with 0
print(df2.dropna())     # drop rows with NaN
```

## 9. GroupBy

```
data = {
    "Department": ["HR", "IT", "IT", "HR", "Finance"],
    "Salary": [40000, 50000, 60000, 45000, 70000]
}
df3 = pd.DataFrame(data)
print(df3.groupby("Department")["Salary"].mean())
```

Groups by department and finds average salary.

## 10. Merging & Concatenation

```
df1 = pd.DataFrame({"ID": [1, 2], "Name": ["A", "B"]})
df2 = pd.DataFrame({"ID": [1, 2], "Marks": [90, 80]})

print(pd.merge(df1, df2, on="ID"))

df3 = pd.DataFrame({"ID": [3, 4], "Name": ["C", "D"]})
print(pd.concat([df1, df3]))
```

**NumPy**: efficient numerical computations using arrays, supports vectorized operations, reshaping, broadcasting, and aggregation.

**Pandas**: built on NumPy, used for structured/tabular data with Series & DataFrames, supports filtering, grouping, merging, and handling missing data.