

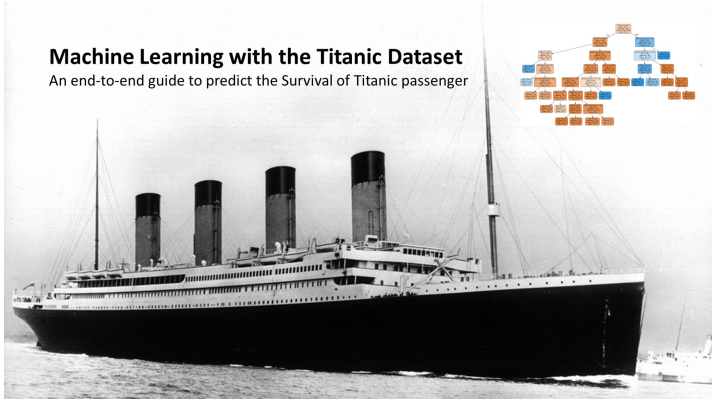

Basic AI

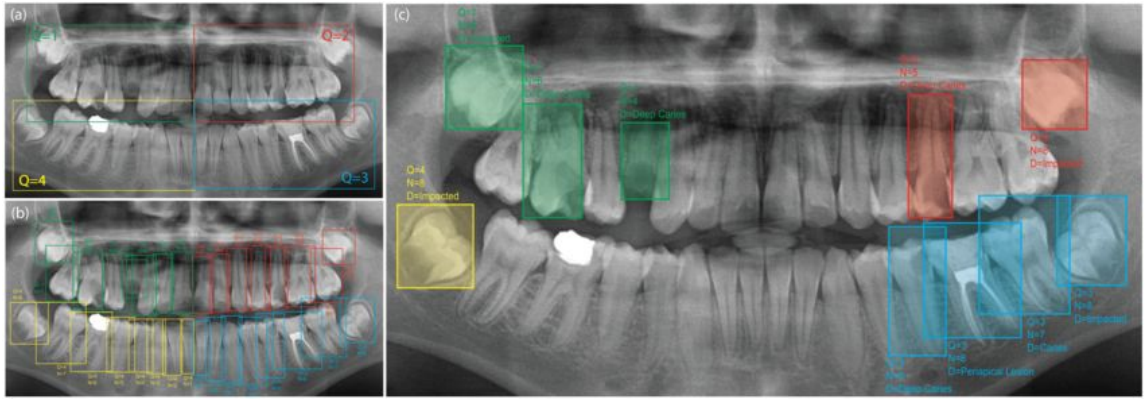
Kao Panboonyuen

https://github.com/kaopanboonyuen/MARS-LEARN

Outlines

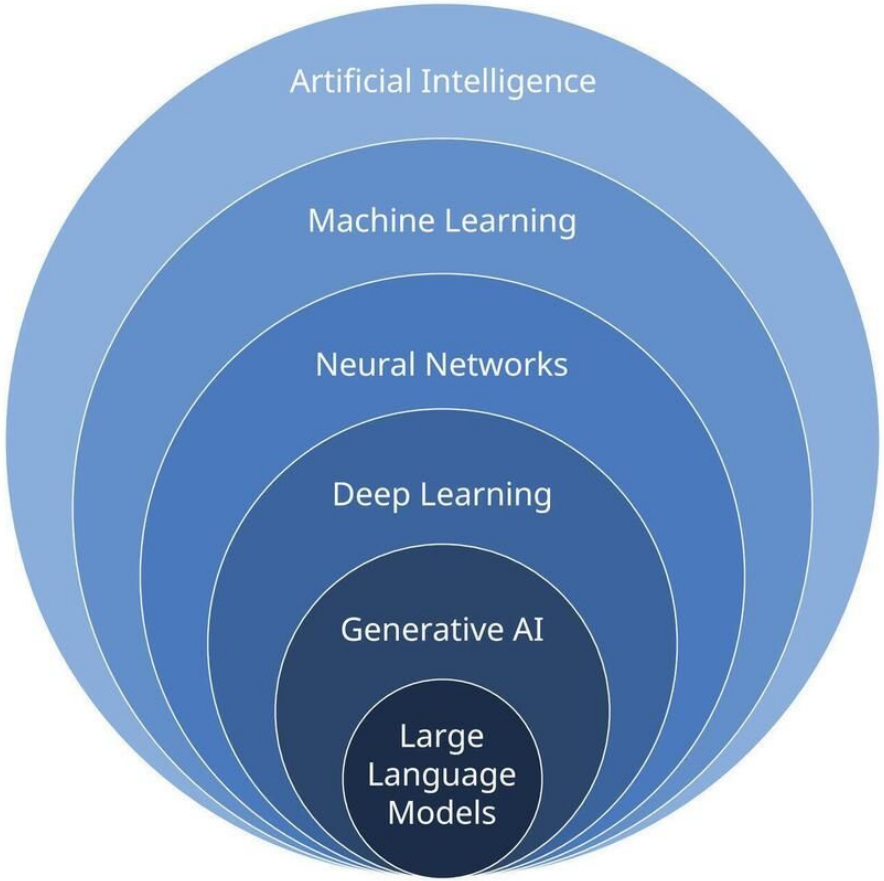
- Basic AI
- Lab: Machine Learning (Titanic)
- Lab: Deep Learning (Dental Radiography)





Overview of AI in Computer Vision

- **Artificial Intelligence (AI)** enables machines to interpret and understand visual data.
- **Computer Vision (CV)** is the field that allows computers to extract, analyze, and process visual information from the world.


Artificial Intelligence
Machine Learning
Neural Networks
Deep Learning
Generative AI
Large Language Models

What is AI?

ANI vs. AGI vs. ASI

Artificial narrow intelligence (ANI)

Designed to perform specific tasks

Artificial general intelligence (AGI)

Can behave in a human-like way across all tasks

Artificial super intelligence (ASI)

Smarter than humans—the stuff of sci-fi

_zapier

TECH

‘Godfathers of AI’ honored with Turing Award, the Nobel Prize of computing

From left to right: Yann LeCun | Photo: Facebook; Geoffrey Hinton | Photo: Google; Yoshua Bengio | Photo: Botler AI

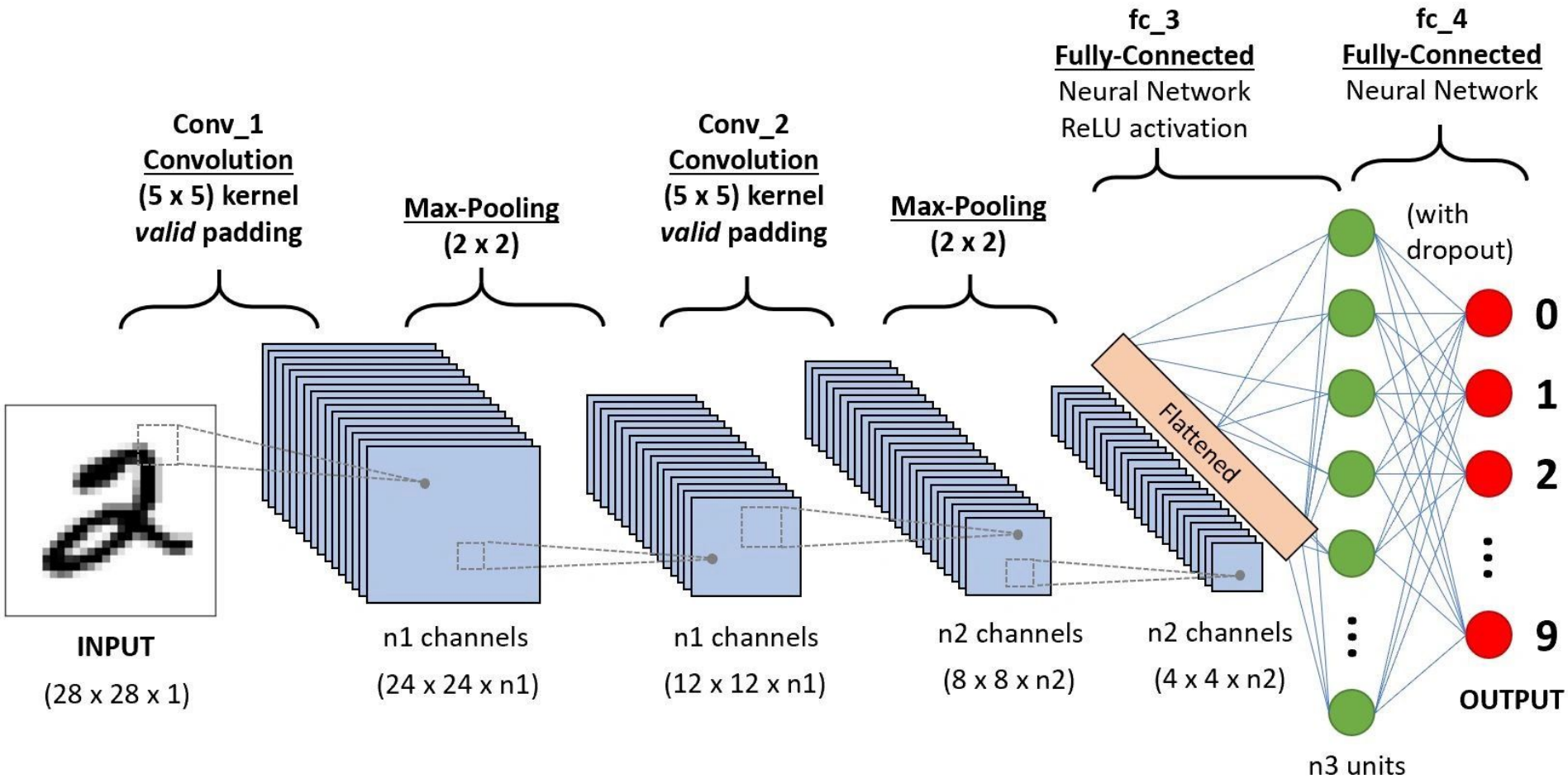
/ Yoshua Bengio, Geoffrey Hinton, and Yann LeCun laid the foundations for modern AI

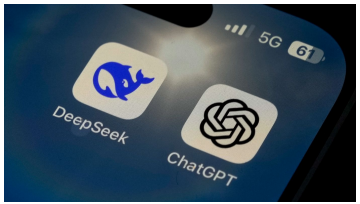
by **James Vincent**

Mar 27, 2019, 5:02 PM GMT+7

0 Comments

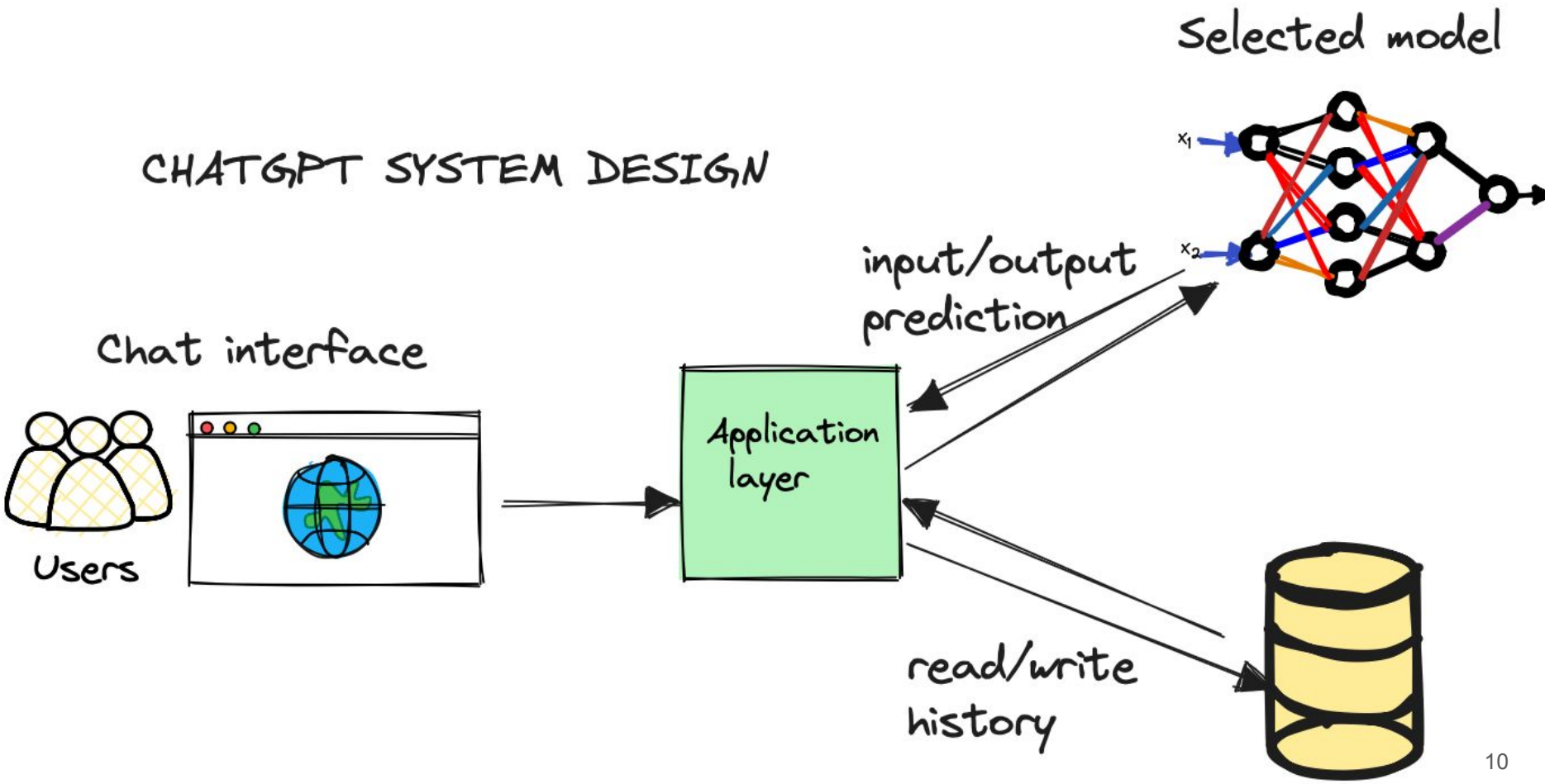

"If your are interested in AGI, don't work on LLMs"
The Next AI Revolution
Yann LeCun

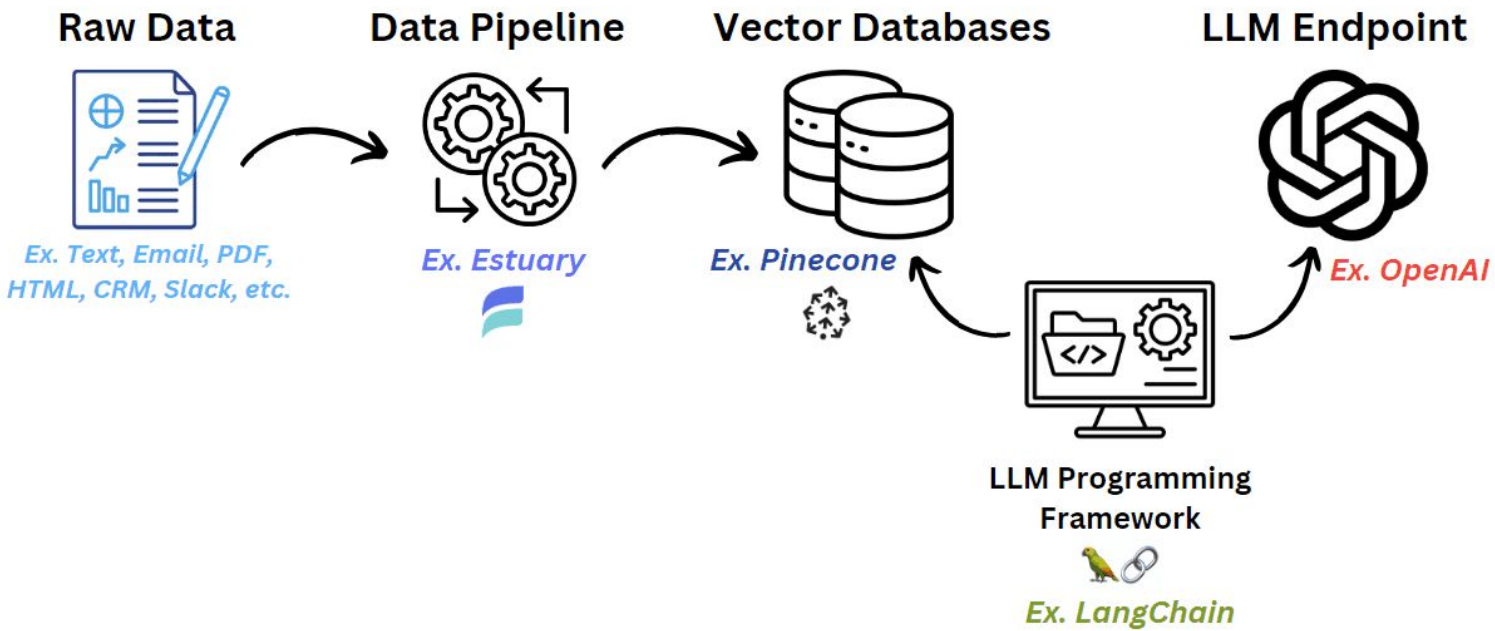

Conv_1
Convolution
(5 x 5) kernel
valid padding
Max-Pooling
(2 x 2)
Conv_2
Convolution
(5 x 5) kernel
valid padding
Max-Pooling
(2 x 2)
fc_3
Fully-Connected
Neural Network
ReLU activation
fc_4
Fully-Connected
Neural Network
(with dropout)
Flattened
INPUT
(28 x 28 x 1)
n1 channels
(24 x 24 x n1)
n1 channels
(12 x 12 x n1)
n2 channels
(8 x 8 x n2)
n2 channels
(4 x 4 x n2)
n3 units
0
1
2
9
OUTPUT

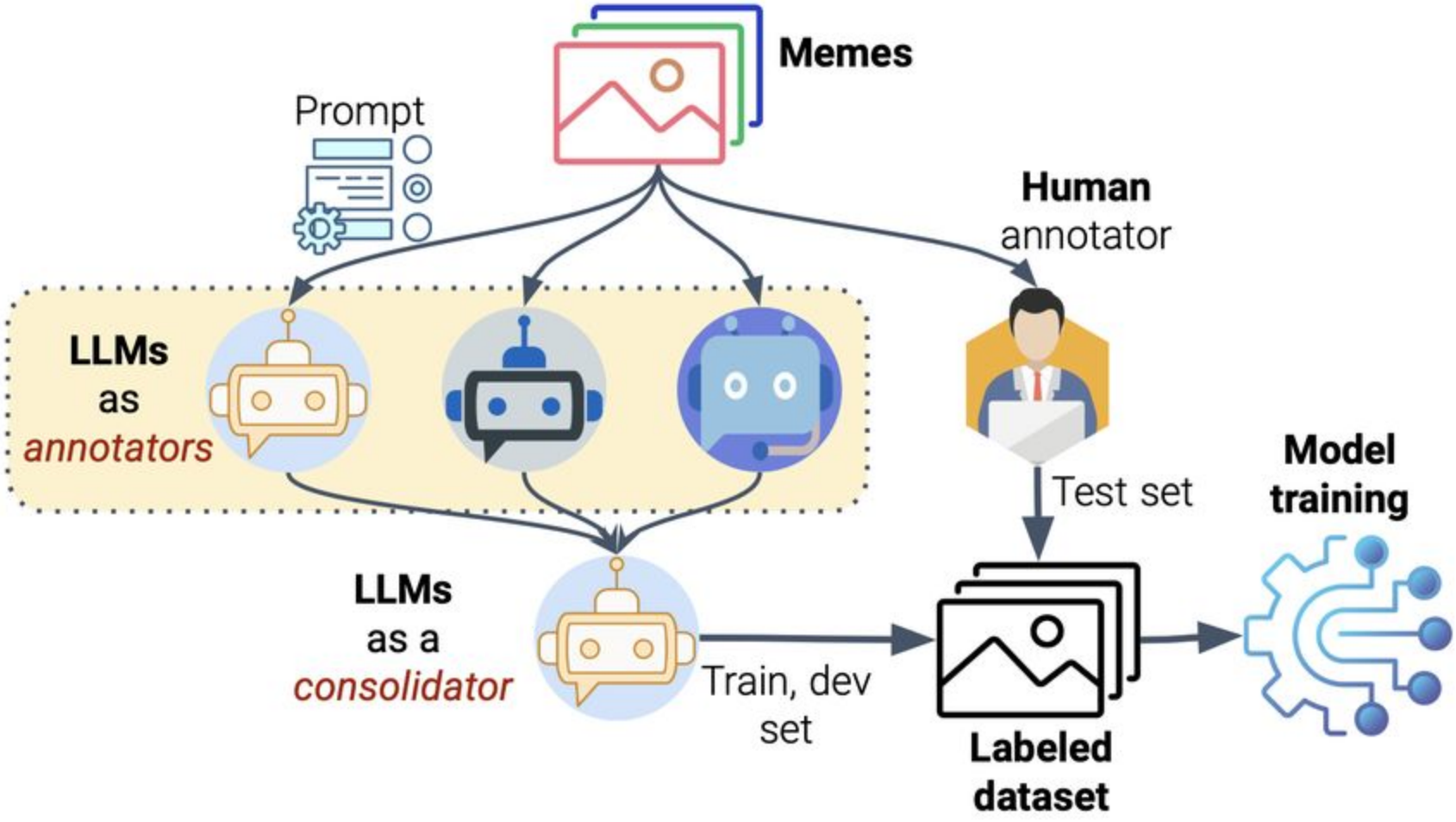


PaLM 2 / BERT	Google
DALL•E / GPT- 4	OpenAI
LLaMA	Meta
Claude	ANTHROP\C
Dolly	databricks
RedPajama	TOGETHER
MPT- 7B	mosaicML

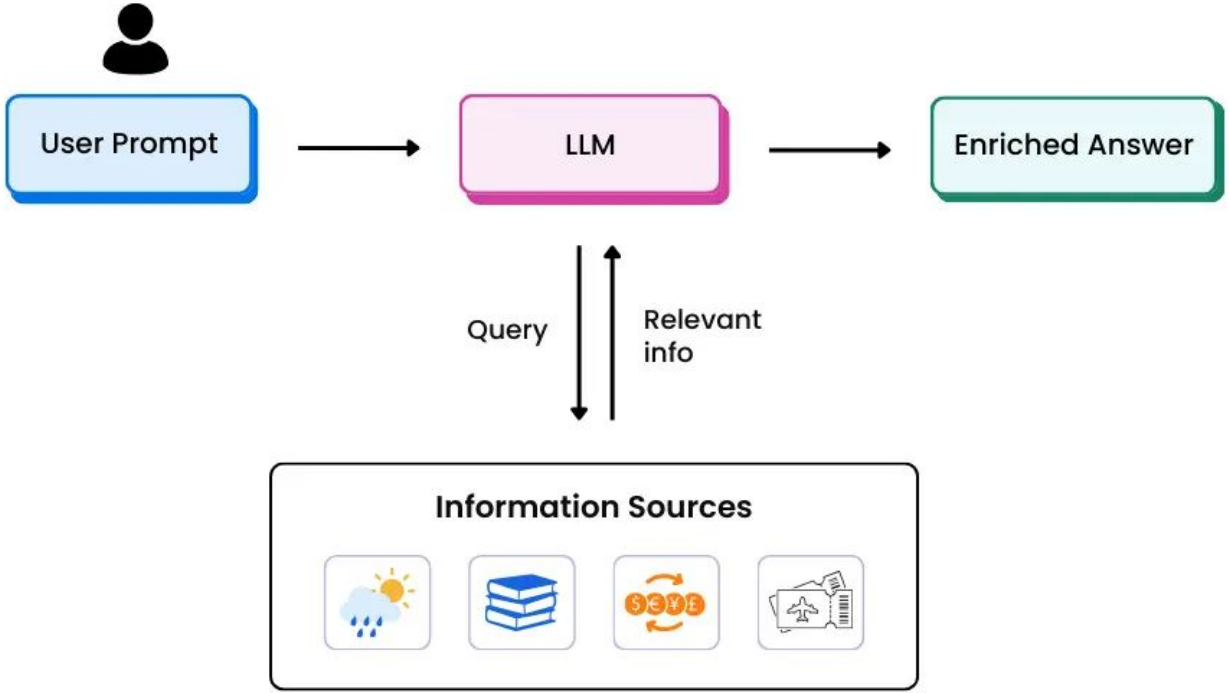
CHATGPT SYSTEM DESIGN

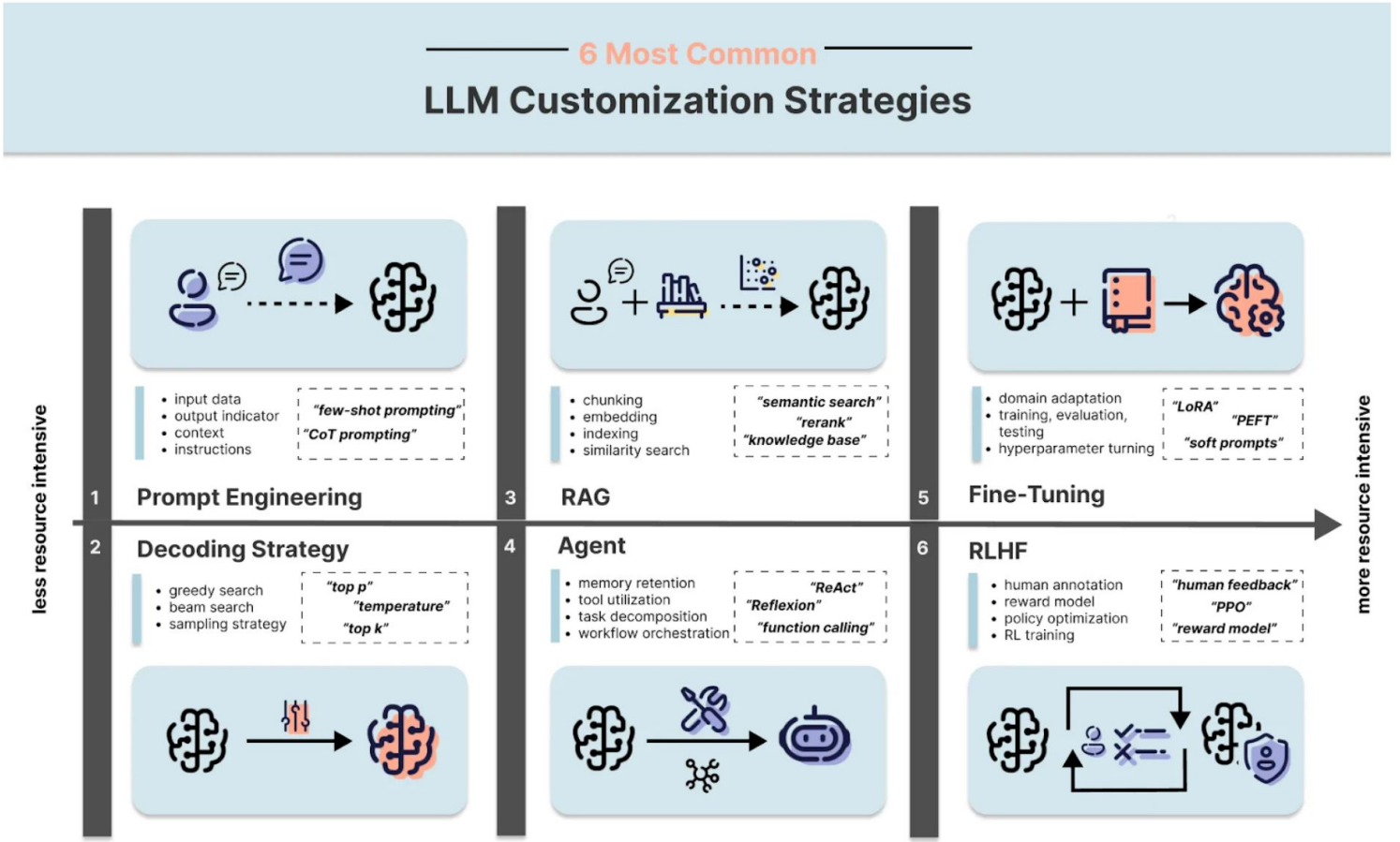



Raw Data
Data Pipeline
Vector Databases
LLM Endpoint
Ex. Text, Email, PDF, HTML, CRM, Slack, etc.
Ex. Estuary
Ex. Pinecone
Ex. OpenAI
LLM Programming Framework
Ex. LangChain


Memes
Prompt
Human
annotator
LLMs
as
annotators
Test set
Model
training
LLMs
as a
consolidator
Train, dev
set
Labeled
dataset

Retrieval-Augmented Generation (RAG) in LLMs

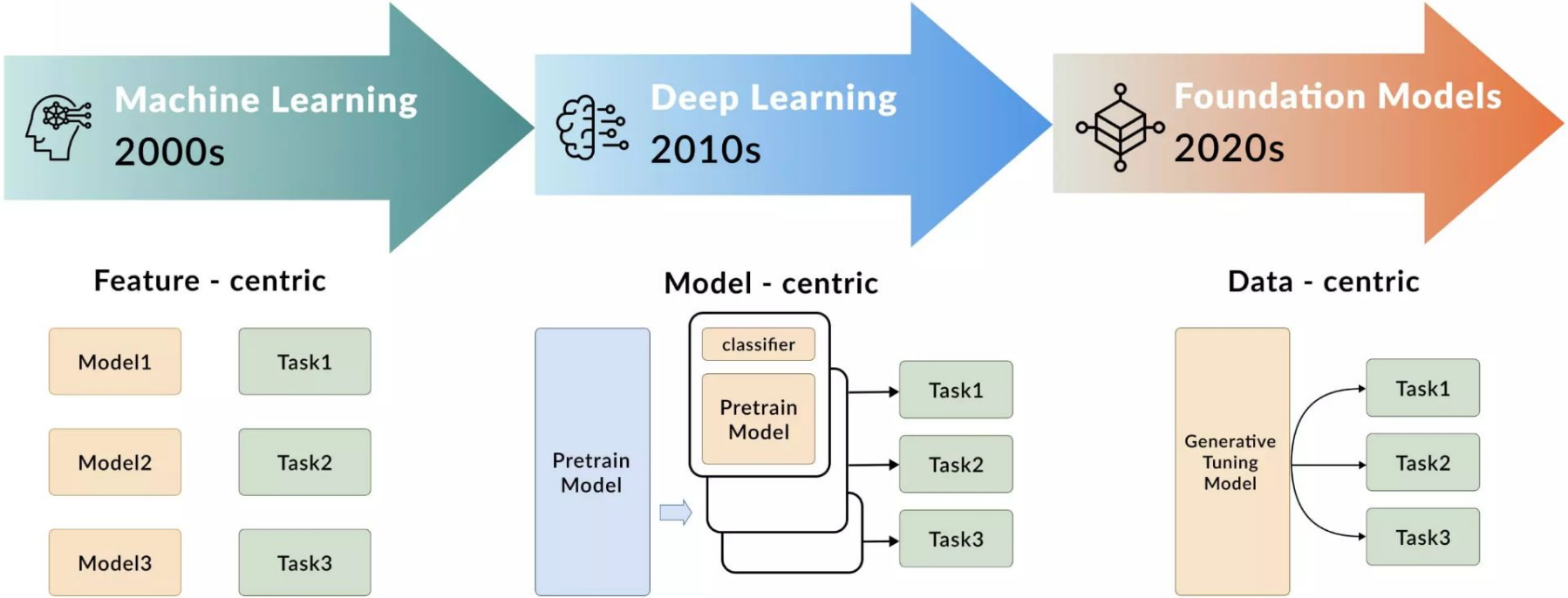




6 Common LLM Customization Strategies (unless otherwise noted, all images are by the author)

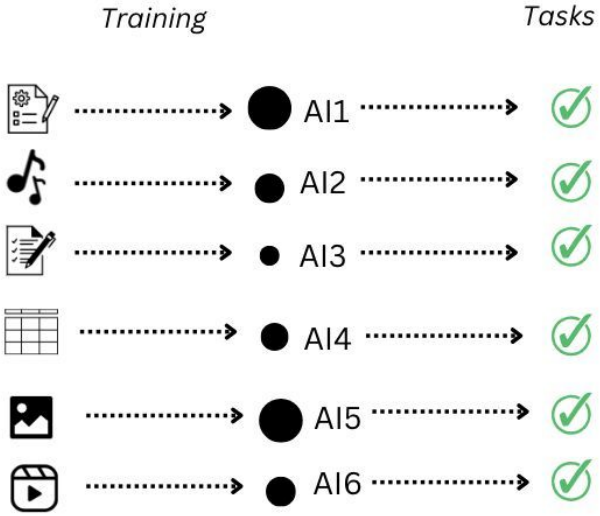
A New Era of AI: Foundation Models

Step function improvements over legacy AI technologies



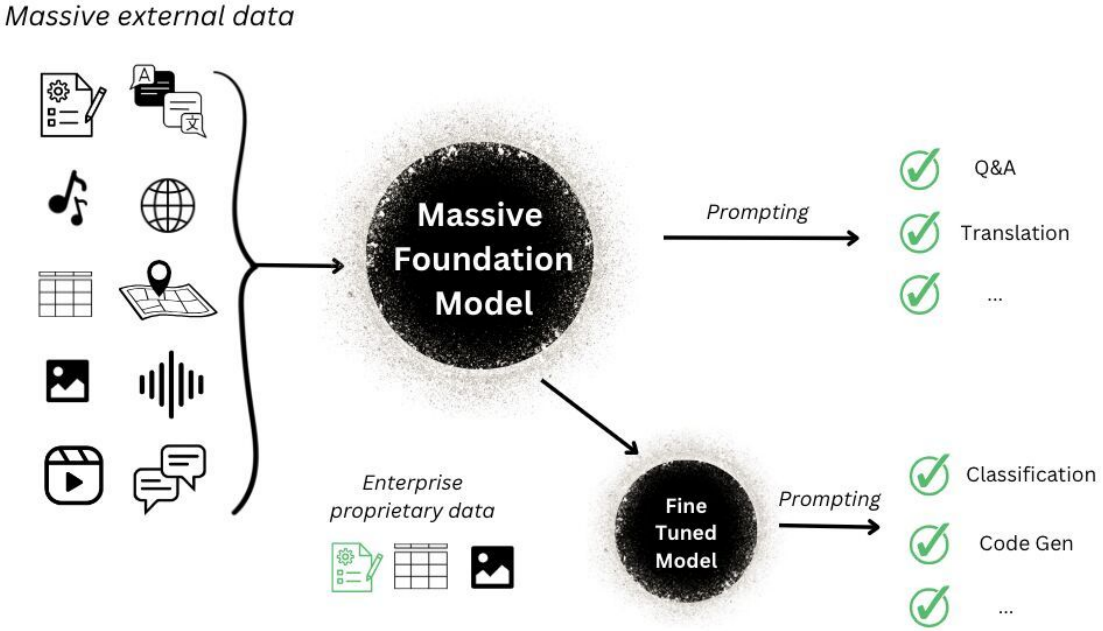
DATAFOREST

Traditional ML

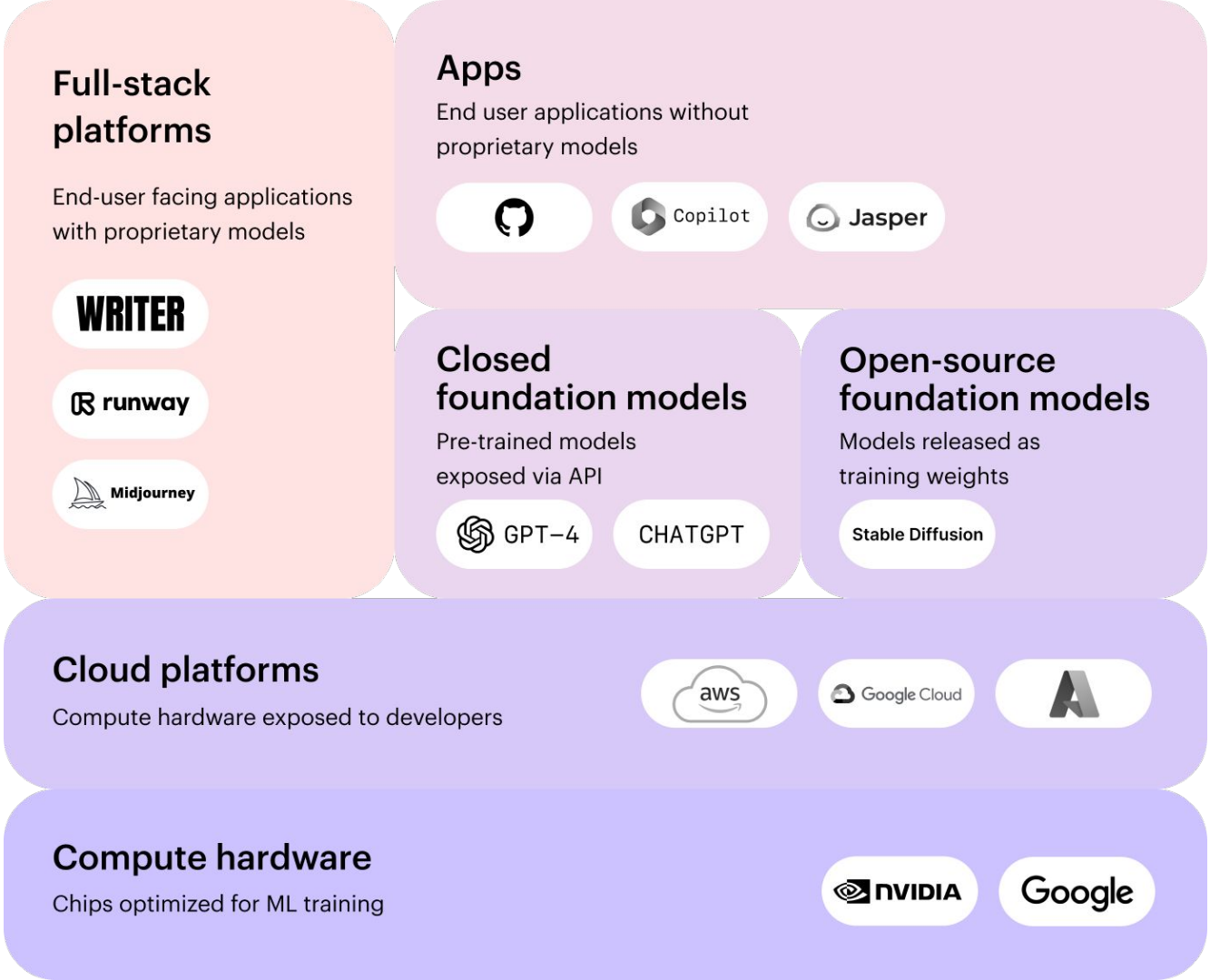


- Individual siloed models
- Require task-specific training
- Lots of human supervised training

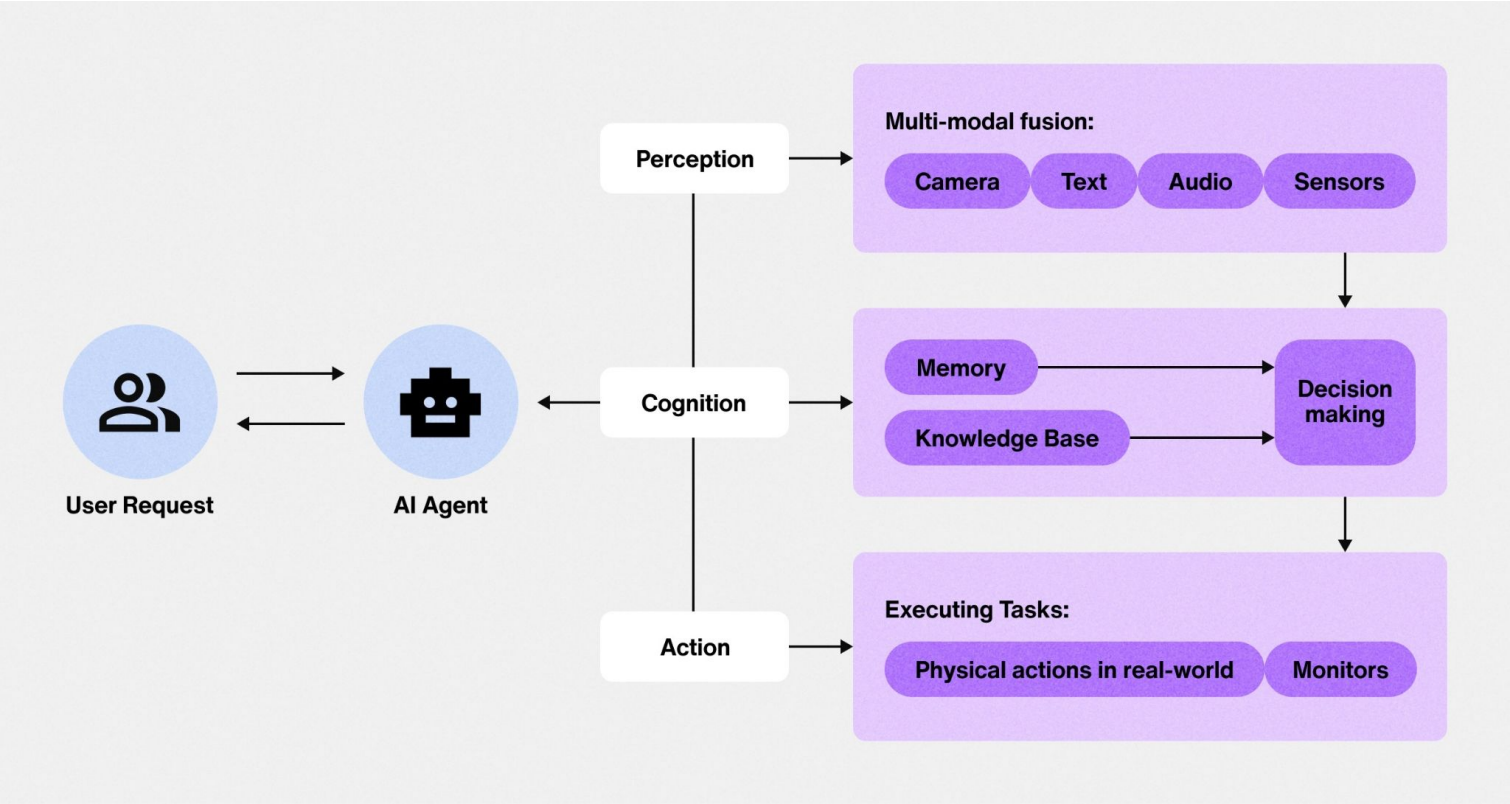
Foundation Models



- Massive multi-tasking model
- Adaptable with little or no training
- Pre-trained unsupervised learning


Full-stack platforms
End-user facing applications with proprietary models
WRITER
runway
Midjourney
Apps
End user applications without proprietary models
Copilot
Jasper
Closed foundation models
Pre-trained models exposed via API
GPT-4
CHATGPT
Open-source foundation models
Models released as training weights
Stable Diffusion
Cloud platforms
Compute hardware exposed to developers
aws
Google Cloud
Compute hardware
Chips optimized for ML training
NVIDIA
Google

Agentic AI

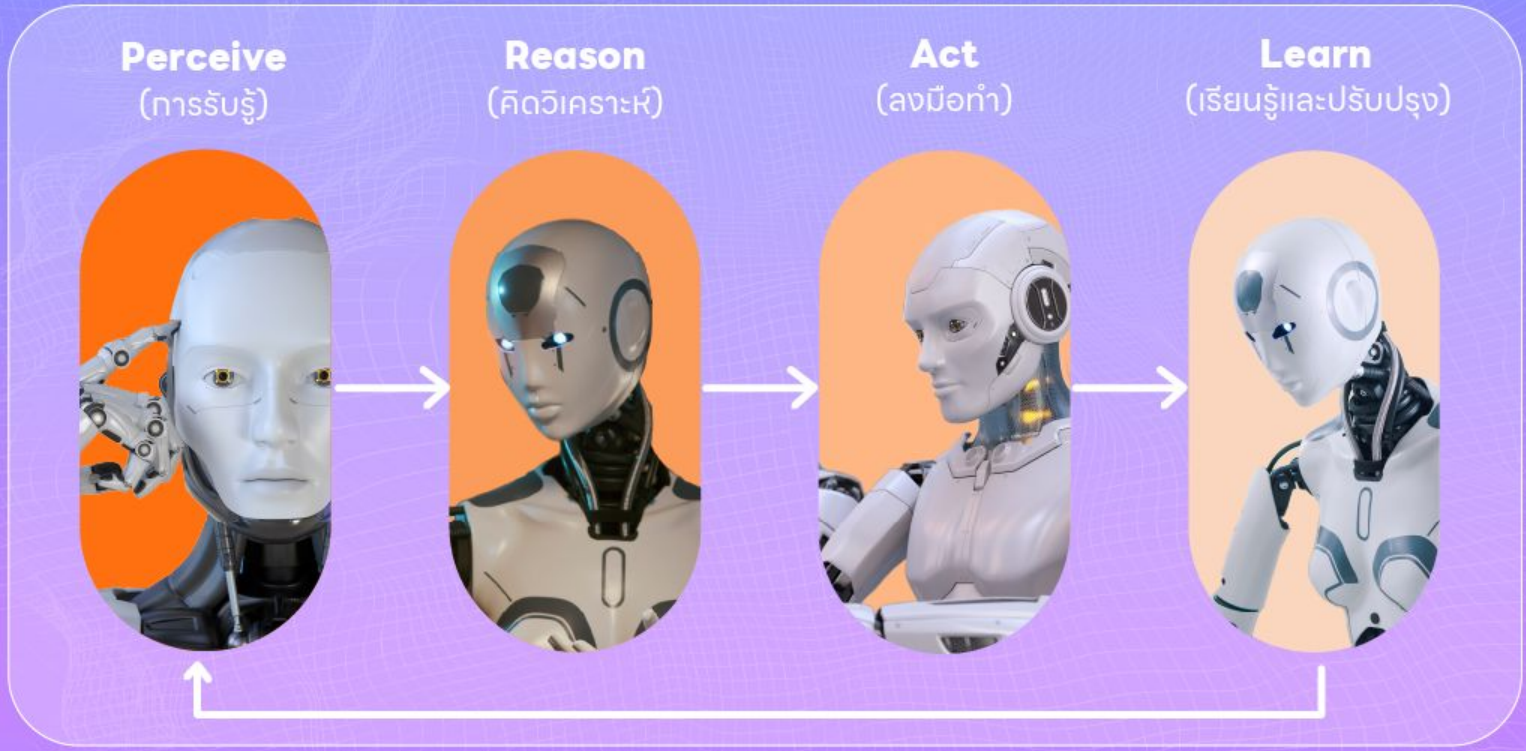

Perception
Multi-modal fusion:
Camera
Text
Audio
Sensors
User Request
AI Agent
Cognition
Memory
Knowledge Base
Decision making
Action
Executing Tasks:
Physical actions in real-world
Monitors

Generative AI

หลักการทำงานของ Agentic AI

4 ขั้นตอนการทำงานของ Agentic AI

9Expert

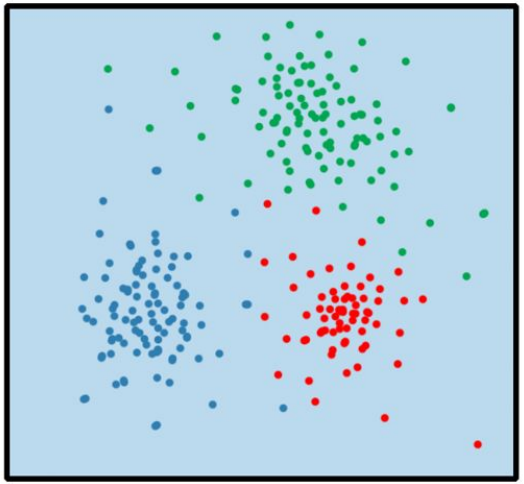


บทความโดย: 9Expert Training

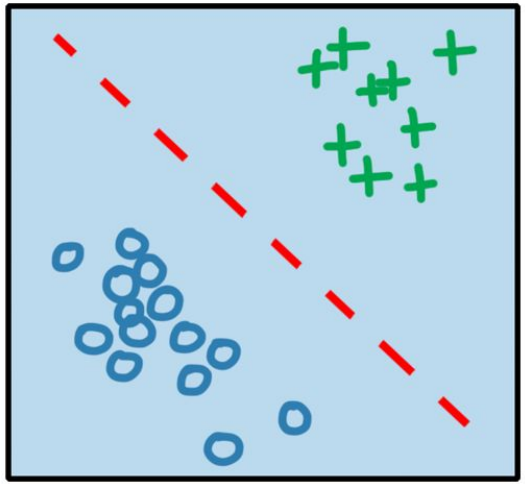
Never Stop Learning

machine learning

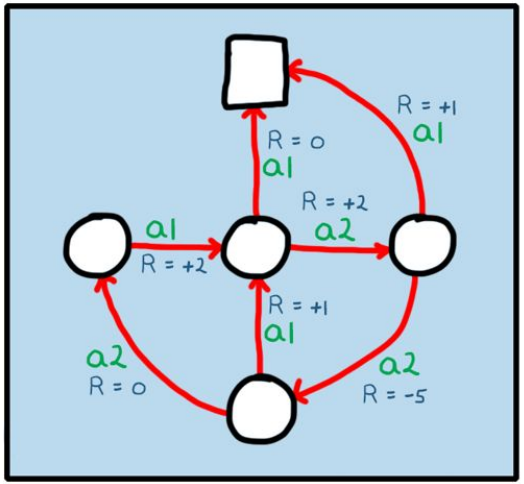
unsupervised learning

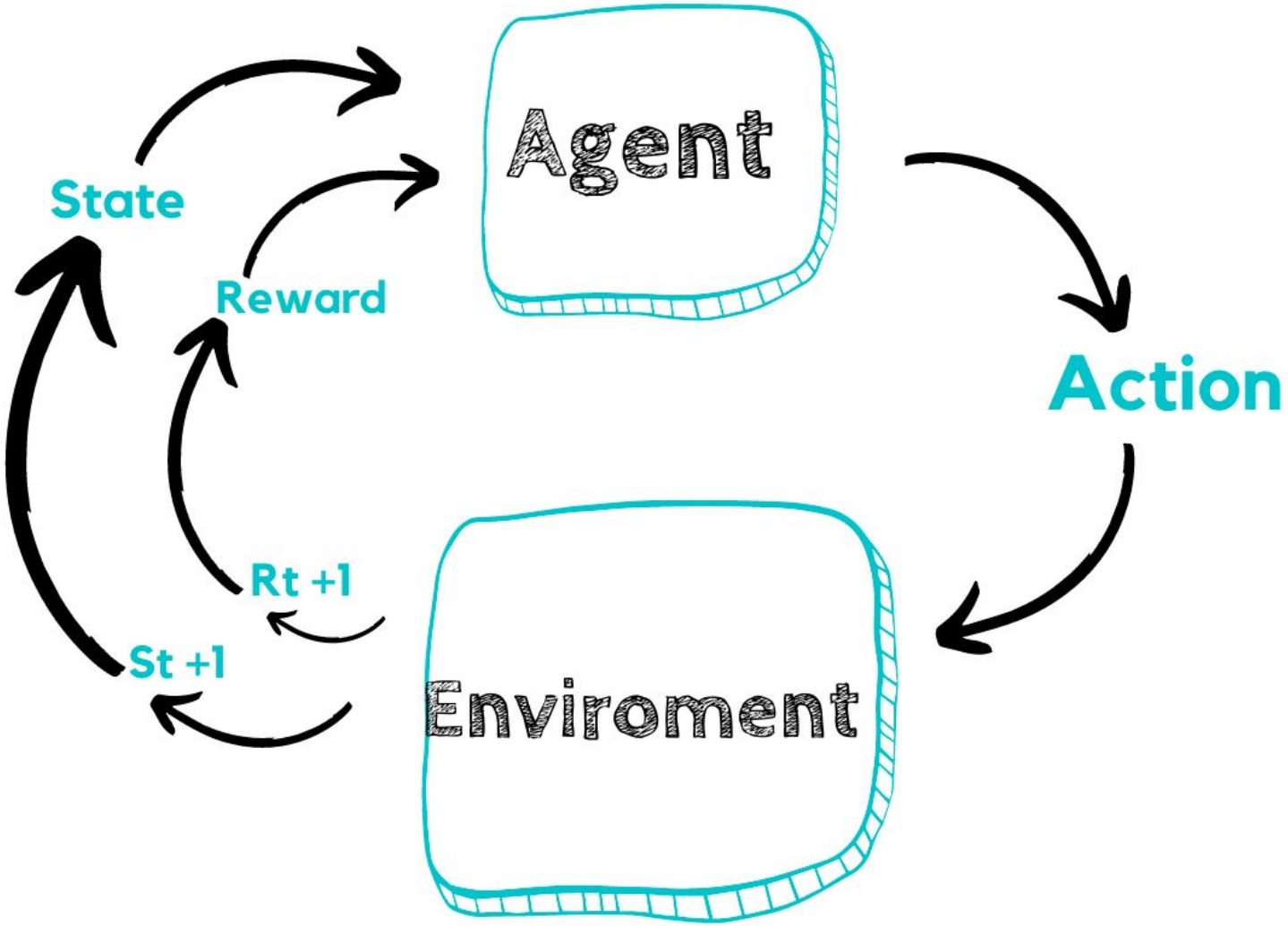


supervised learning



reinforcement learning

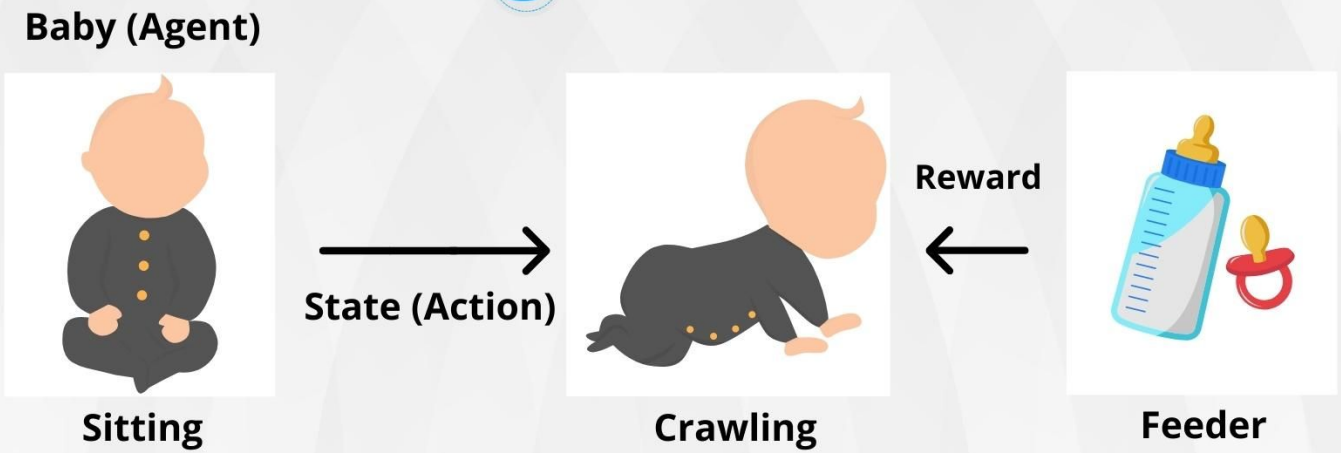



Agent
State
Reward
Action
Rt +1
St +1
Enviroment

REINFORCEMENT LEARNING

Reinforcement learning is a machine learning paradigm that focuses on how agents learn to interact with an environment to maximize cumulative rewards.

DatabaseTown



Algorithms and Approaches in Reinforcement Learning

- Q-learning
- Deep Q-networks (DQN)
- Policy Gradients Methods
- Proximal Policy Optimization (PPO)

environment

agent

actions

rewards

observations

What is Model Distillation?

Created by genuine impact

Model distillation is the process of transferring knowledge from a large teacher model to a smaller student model to improve efficiency while maintaining performance.

- "Runs faster on smaller devices."
- "Consumes less energy."
- "Maintains good accuracy."

Big Teacher Model

A large and powerful AI model that is highly accurate but slow and resource-intensive.

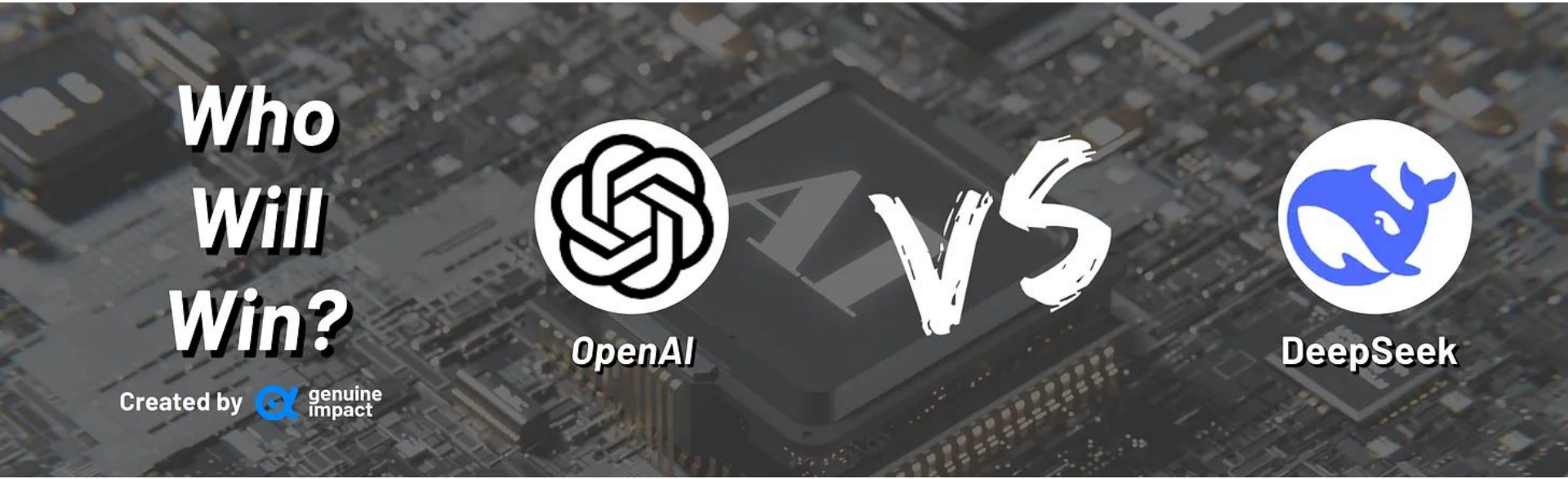
Knowledge Transfer

The teacher passes its knowledge (important patterns and decisions) to a smaller model.

Small Student Model

A lightweight and fast model that learns from the teacher's knowledge, becoming almost as good but much more efficient.

More charts: genuineimpact.substack.com


Who
Will
Win?
OpenAI
VS
DeepSeek
Created by genuine impact

DeepSeek vs OpenAI

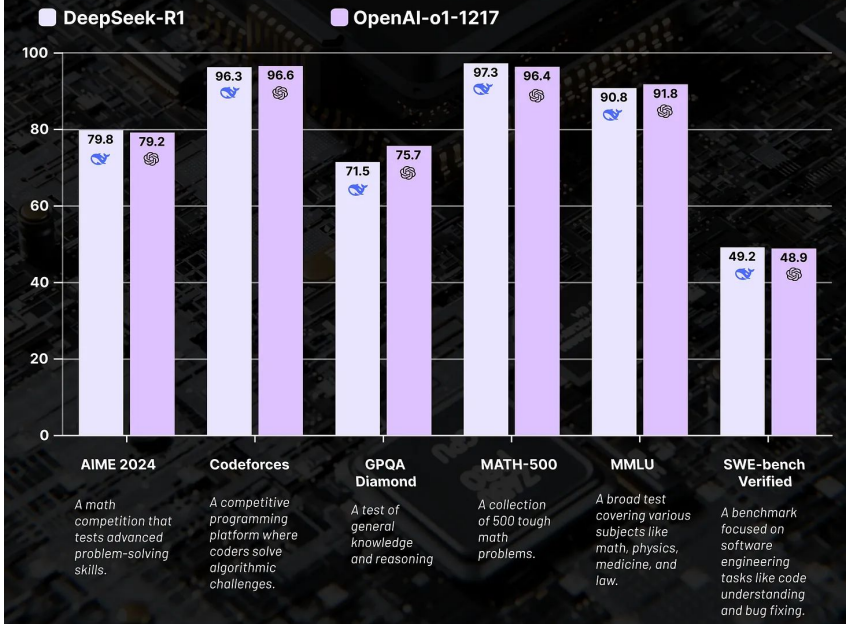
Created by genuine impact

DeepSeek-R1:

- *A reasoning model developed by the Chinese artificial intelligence company DeepSeek, designed to handle complex tasks in mathematics, programming, and natural language reasoning.*
- *Release Date: January 20, 2025.*

OpenAI-o1-1217:

- *A version of OpenAI's o1 model, focusing on enhancing AI's reasoning capabilities.*
- *Release Date: The preview version of o1 was released on September 12, 2024, with the official version launched on December 5, 2024.*



Source: DeepSeek Official Website

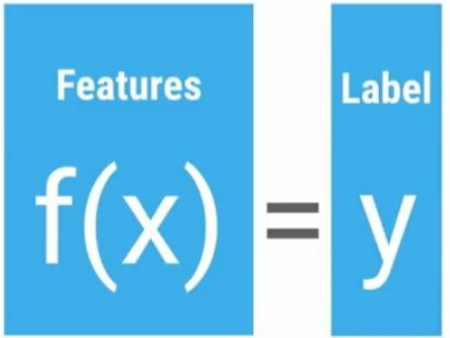
More charts: genuineimpact.substack.com

Model Serving: Paid Options for LLM Integration

Well-Known Examples of Agent Frameworks

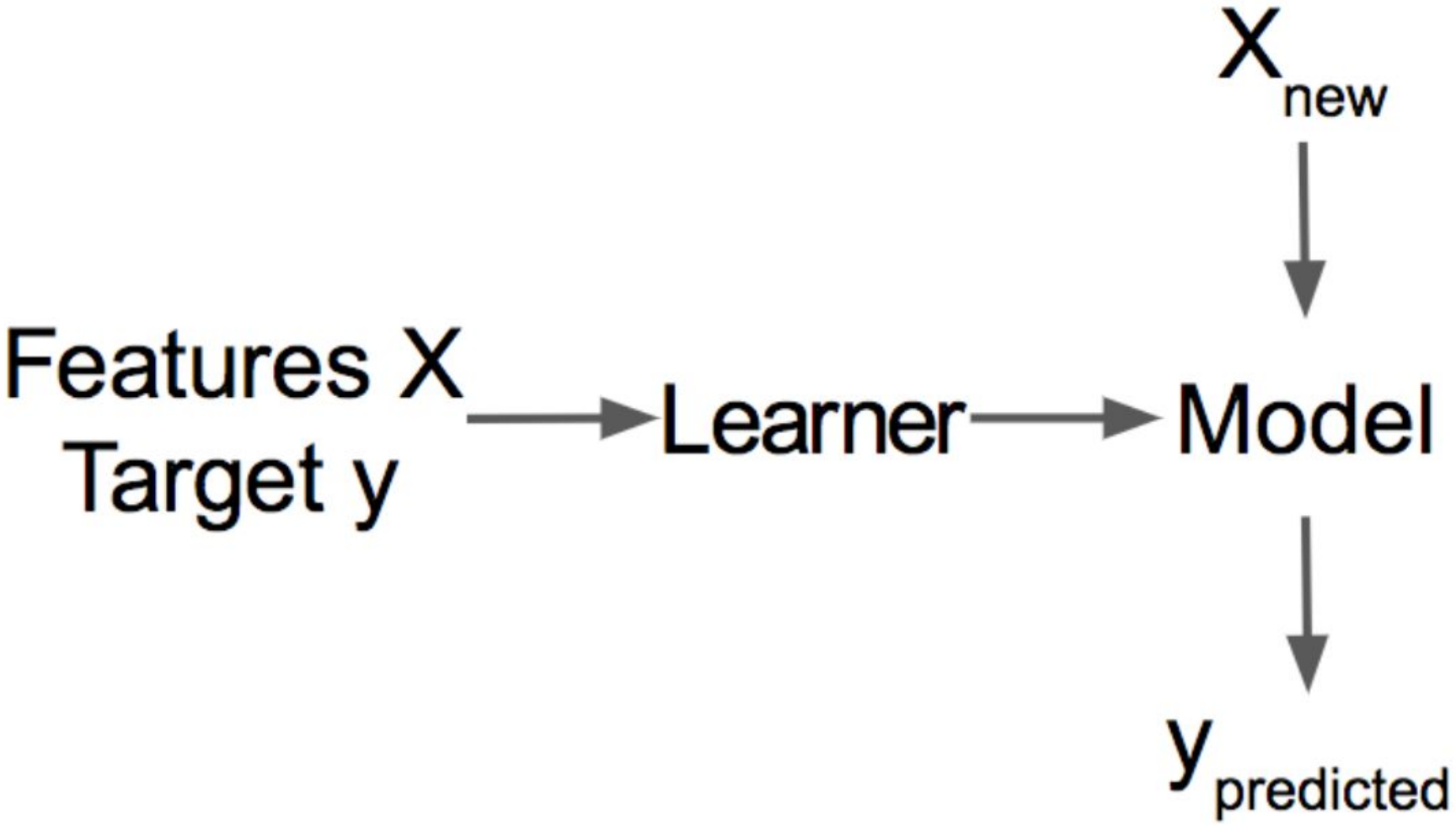
Machine Learning Basics

- **Types:**
    - Supervised (Regression, Classification)
    - Unsupervised (Clustering, Dimensionality Reduction)
    - Reinforcement Learning



1. Define the Problem

- **Goal**: Determine the task (e.g., classification, regression).
- **Input Data**: $\mathbf{X}$ (features) and target $\mathbf{y}$ (labels).


X new
Features X
Target y
Learner
Model
y predicted

2. Model Selection

- Choose the type of model (e.g., linear regression, neural networks, etc.).
- **Example (Linear Regression)**:

$$\hat{y} = \mathbf{X}\mathbf{w} + b$$

where:

- $\hat{y}$ is the predicted output,
- $\mathbf{w}$ is the weight vector,
- b is the bias.

Linear regression equation

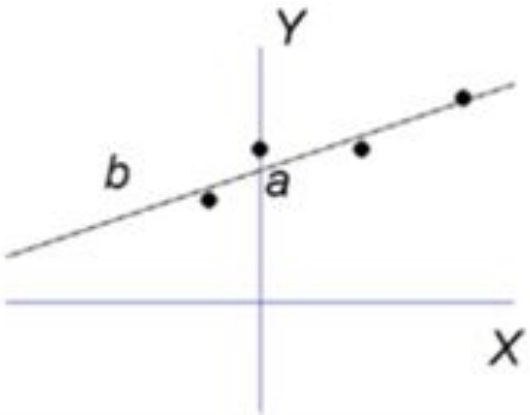
(without error)

$$\hat{Y} = bX + a$$

predicted values of *Y*

b = slope = rate of predicted ↑/↓ for Y scores for each unit increase in *X*

Y-intercept = level of *Y* when X is 0



3. Initialize Parameters

- For most models, initialize parameters (weights and bias):
  - **Linear Regression**: Initialize $\mathbf{w}$ and b randomly (or with small values).
  - **Neural Network**: Initialize weights $\mathbf{W}$ and biases $\mathbf{b}$ for each layer.

4. Define the Loss Function

- **Loss function**: Measures how well the model's predictions match the actual values.
  - For **Regression** (e.g., Mean Squared Error):

$$L(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

  - For **Classification** (e.g., Cross-Entropy Loss):

$$L(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{n} y_i \log(\hat{y}_i)$$

5. Train the Model (Optimization)

- **Gradient Descent**: Update parameters based on the gradient of the loss function.
  - **Gradient Update Rule**:

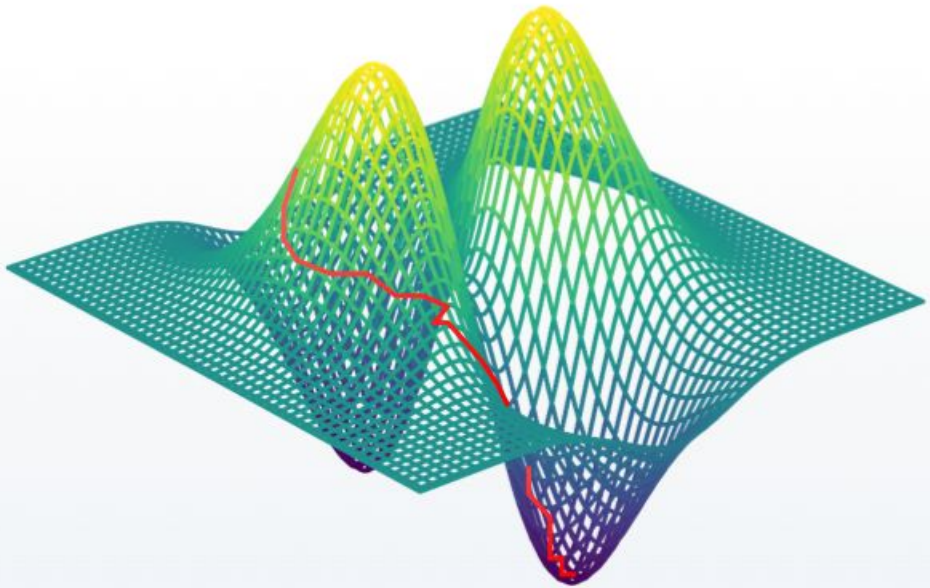
$$\mathbf{w} := \mathbf{w} - \eta \nabla_{\mathbf{w}} L(\mathbf{X}, \mathbf{y})$$

    where:

    - η is the learning rate,
    - $\nabla_{\mathbf{w}} L$ is the gradient of the loss with respect to $\mathbf{w}$.
  - For a **Neural Network**, backpropagation computes the gradient for each layer.

Gradient Descent

THE WORKING HORSE FOR DEEP LEARNING OPTIMIZATION



GRADIENT DESCENT

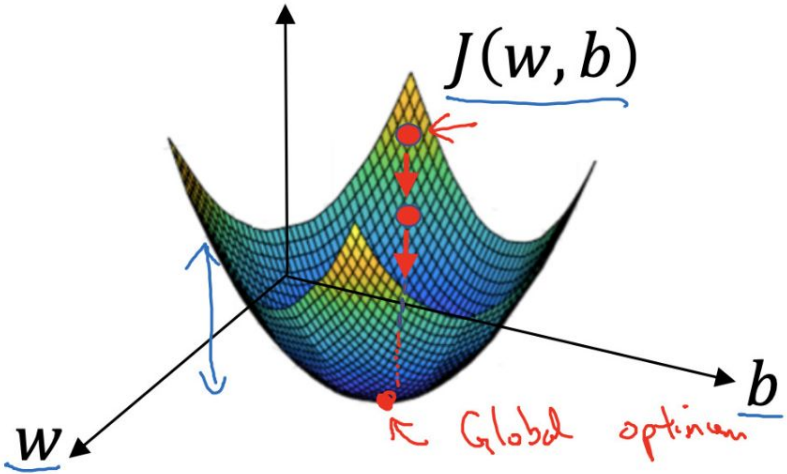
IN MACHINE LEARNING

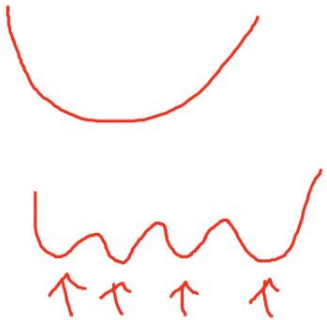
Gradient Descent

Recap: $\hat{y} = \sigma(w^T x + b)$, $\sigma(z) = \frac{1}{1+e^{-z}}$

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})$$

Want to find w, b that minimize $J(w,b)$





6. Evaluate the Model

- After training, evaluate model performance on a validation dataset.
- Calculate the loss on the validation set:

$$L_{val} = L(\mathbf{X}_{val}, \mathbf{y}_{val})$$

7. Fine-Tuning the Model

- **Fine-tuning** involves adjusting model hyperparameters or layers based on performance.

Steps for Fine-Tuning:

1. **Freeze Layers**: In the case of transfer learning, freeze early layers of the model and only fine-tune later layers.

$$\mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_k \quad (\text{freeze layers 1 to k})$$

2. **Adjust Learning Rate**: Lower the learning rate for fine-tuning.

$$\eta' = \eta / 10$$

3. **Regularization**: Add L2 regularization (weight decay) to prevent overfitting:

$$L_{reg}(\mathbf{w}) = L(\mathbf{y}, \hat{\mathbf{y}}) + \lambda \|\mathbf{w}\|^2$$

   where λ is the regularization coefficient.

4. **Early Stopping**: Stop training once the validation loss no longer improves.

8. Hyperparameter Tuning

- Optimize hyperparameters (e.g., learning rate, batch size, regularization strength) using techniques like grid search or random search.

---

9. Final Model

- Once the model is fine-tuned, retrain on the full dataset and evaluate its performance.

---

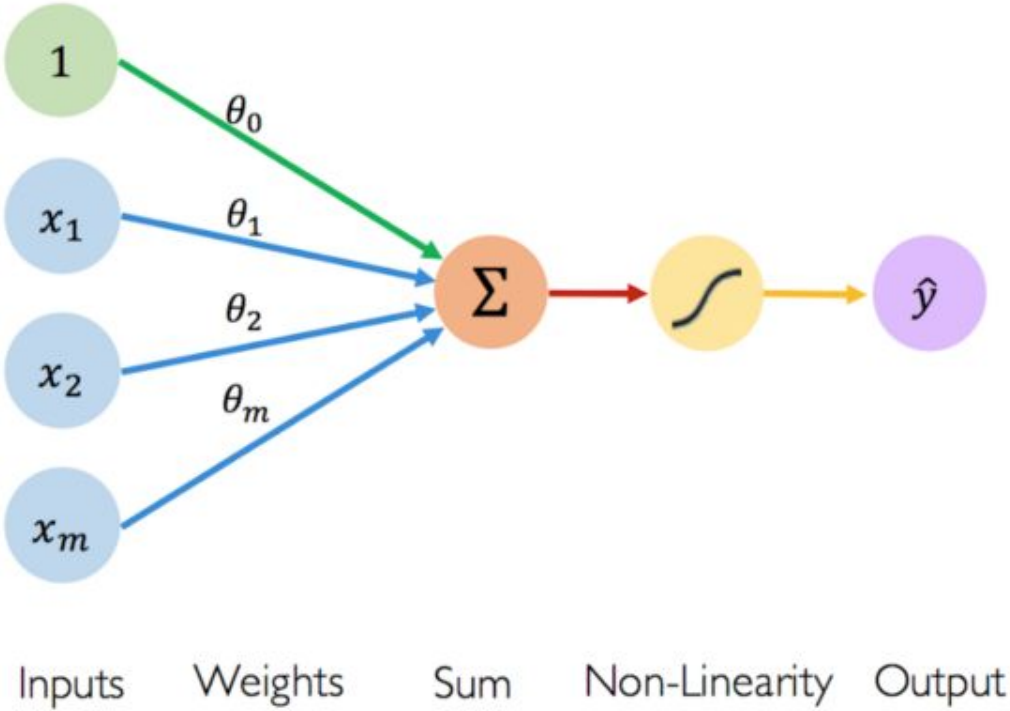
10. Deployment

- Deploy the trained and fine-tuned model for inference or use in a production system.

Summary of Mathematical Steps for Fine-Tuning:

1. **Loss Function**: Choose an appropriate loss function (e.g., MSE for regression, cross-entropy for classification).
2. **Gradient Descent**: Update weights based on the gradient of the loss.
3. **Fine-Tuning**: Freeze layers, adjust the learning rate, and apply regularization.
4. **Evaluate**: Use a validation set to fine-tune hyperparameters and monitor performance.

Understanding the Model

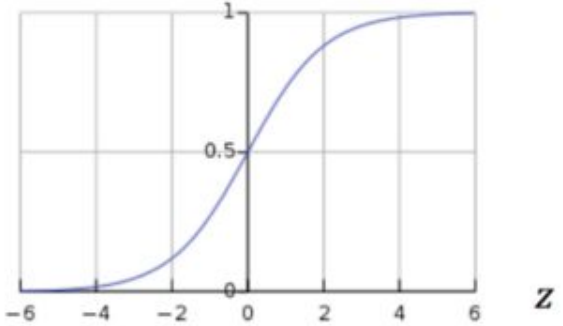


Activation Functions

$$\hat{y} = g\left(\theta_0 + \boldsymbol{X}^T\boldsymbol{\theta}\right)$$

- Example: sigmoid function

$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$



Imagine we are trying to predict something based on certain features. For example, we might want to predict a student's exam score based on the number of hours studied, their sleep hours, and their attendance in class.

The model we use is a **linear model**:

$$\hat{y} = w_0 x_0 + w_1 x_1 + w_2 x_2$$

Where:

- $\hat{y}$ is the predicted exam score (our output).
- x_0, x_1, x_2 are the features:
  - x_0 could represent the hours studied,
  - x_1 could represent the hours slept, and
  - x_2 could represent class attendance.
- w_0, w_1, w_2 are **weights** that the model needs to learn. These weights represent how much each feature contributes to the final prediction.

The Role of Weights:

- The **weight** w_0 is the bias term (or the intercept). This is the starting value when all features x_0, x_1, x_2 are 0. It represents the baseline prediction.
- The **weights** w_1, w_2 are coefficients that determine how much each feature affects the prediction. For example:
  - If w_1 is large, it means that the **hours studied** have a strong effect on the predicted exam score.
  - If w_2 is small, it means that **class attendance** has less impact on the score.

How the Model Finds the Best Weights:

The goal of the model is to **find the best weights** w_0, w_1, w_2 that minimize the difference between the predicted scores $\hat{y}$ and the actual scores y (the true exam scores).

To do this, we:

1. **Start with random weights** (e.g., $w_0 = 0, w_1 = 0, w_2 = 0$).
2. **Calculate the predicted score** using the model formula:

$$\hat{y} = w_0 x_0 + w_1 x_1 + w_2 x_2$$

3. **Compare the predicted score** $\hat{y}$ with the actual score y.
4. **Adjust the weights** based on how much the prediction was off. This process is called **gradient descent**:
    - If the model predicted too high, decrease the weights.
    - If the model predicted too low, increase the weights.
5. **Repeat** this process many times (over many iterations or epochs), gradually improving the weights until the model makes accurate predictions.

Example to Illustrate:

Imagine this small dataset of hours studied, hours slept, and class attendance (let's say we have a few students):

Hours Studied (x_0)	Hours Slept (x_1)	Attendance (x_2)	Actual Score (y)
3	6	80%	75
5	7	90%	85
2	5	70%	65

Let's say our model starts with random weights: $w_0 = 0$, $w_1 = 0$, and $w_2 = 0$.

For the first student:

- $x_0 = 3$, $x_1 = 6$, $x_2 = 80\%$ (converted to a numeric value: 0.8).
- Predicted score $\hat{y} = w_0 \times 3 + w_1 \times 6 + w_2 \times 0.8 = 0$.

The model predicted a score of 0, but the actual score was 75. So, the model needs to adjust the weights to reduce the prediction error. Over many iterations, the model will learn that:

- **Increasing** w_1 (weight for hours studied) can increase the score.
- **Increasing** w_2 (weight for attendance) also improves the score.
- The bias w_0 adjusts the baseline prediction.

Key Takeaway:

- The model's goal is to find the **optimal weights** w_0, w_1, w_2 that **minimize** the prediction error, allowing it to make accurate predictions on new data.

1. Logistic Regression Example

Problem: We want to predict whether a student will pass or fail based on the number of hours they studied.

Let's assume we have a dataset like this:

Hours Studied (x_0)	Pass (1) / Fail (0)
1	0
2	0
3	1
4	1
5	1

We'll use **Logistic Regression** to predict the probability of passing based on the hours studied.

Step-by-Step Logistic Regression:

The logistic regression model is based on the following equation:

$$\hat{y} = \sigma(w_0 + w_1 x_0)$$

Where:

- $\hat{y}$ is the predicted probability of passing (between 0 and 1).
- w_0 is the bias (intercept).
- w_1 is the weight for the feature x_0 (hours studied).
- σ is the **sigmoid function** that maps any real number to a value between 0 and 1:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Initialization:

We start by initializing weights randomly. Let's assume:

- $w_0 = 0$
- $w_1 = 0$

So initially, the prediction for every student will be:

$$\hat{y} = \sigma(0 + 0 \cdot x_0) = \sigma(0) = 0.5$$

This means, initially, the model predicts a 50% chance of passing for every student.

Training (Gradient Descent):

We will adjust w_0 and w_1 using gradient descent to minimize the loss function, which for logistic regression is the **cross-entropy loss**:

$$L = -\sum [y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]$$

Where:

- y is the actual label (0 or 1),
- $\hat{y}$ is the predicted probability.

Gradient Descent Update:

The weights are updated using the gradient of the loss with respect to w_0 and w_1:

$$w_0 := w_0 - \eta \frac{\partial L}{\partial w_0}$$

$$w_1 := w_1 - \eta \frac{\partial L}{\partial w_1}$$

Where η is the learning rate, which controls how large the weight updates are.

After a few iterations of training, the weights w_0 and w_1 will be updated to better predict whether a student passes based on hours studied.

Example Final Model:

After training, the model might learn that:

- $w_0 = -4$
- $w_1 = 1$

So the model equation becomes:

$$\hat{y} = \sigma(-4 + 1 \cdot x_0)$$

For a student who studied for 3 hours ($x_0 = 3$):

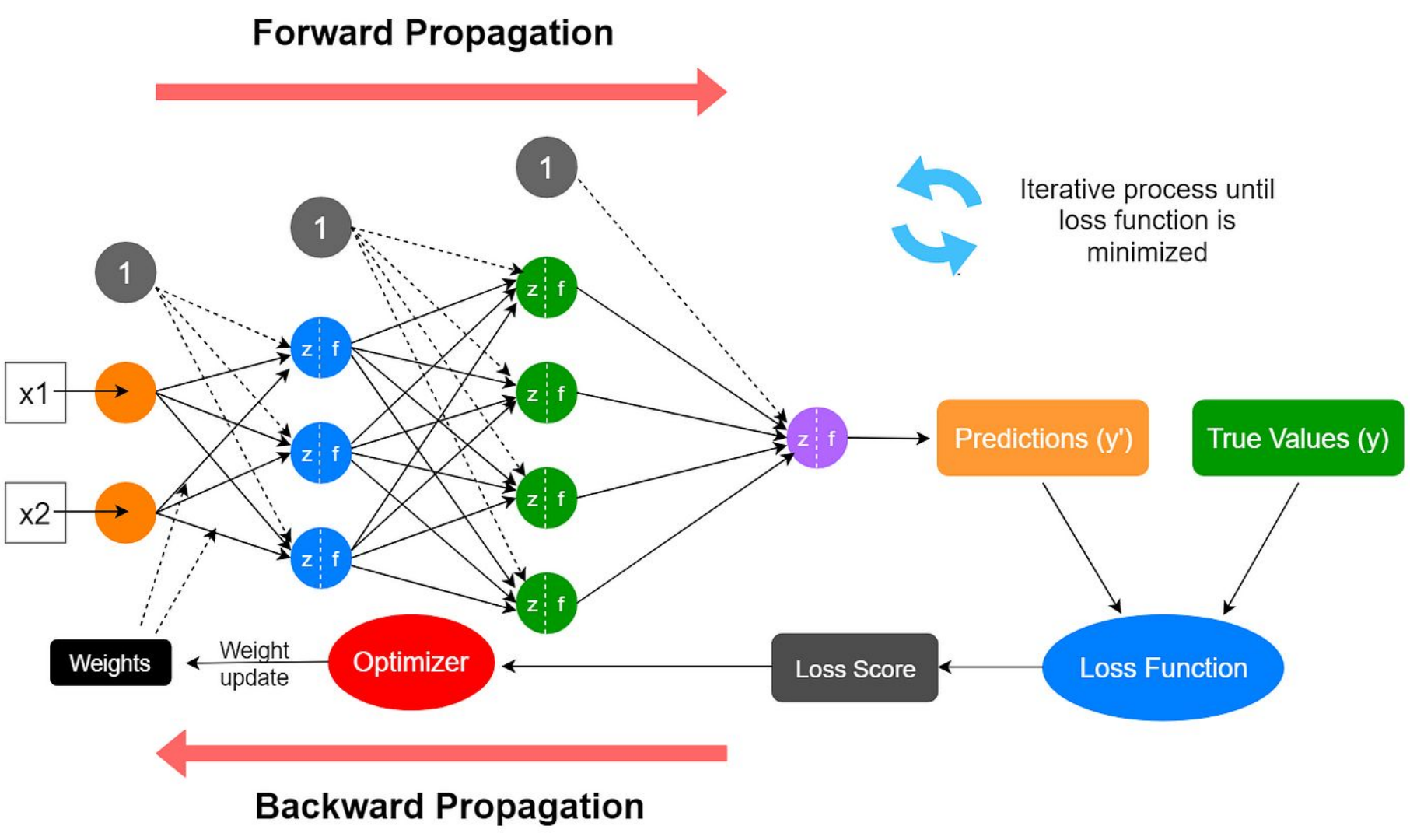
$$\hat{y} = \sigma(-4 + 1 \cdot 3) = \sigma(-1) \approx 0.268$$

This means the model predicts about a **27%** chance of passing for a student who studied for 3 hours.

2. Neural Network Example

Problem: We want to predict the same **Pass/Fail** based on **Hours Studied**, but this time using a **Neural Network**.

Let’s assume a very simple neural network with one hidden layer, and we will use **Sigmoid** activation in both layers.


Forward Propagation
1
1
1
x1
x2
z f
z f
z f
z f
z f
z f
z f
Iterative process until loss function is minimized
Predictions (y')
True Values (y)
Loss Function
Loss Score
Optimizer
Weight update
Weights
Backward Propagation

Step-by-Step Neural Network:

1. **Architecture**:
   - **Input Layer**: 1 node (for hours studied, x_0).
   - **Hidden Layer**: 2 nodes, each using the **sigmoid activation function**.
   - **Output Layer**: 1 node, predicting the probability of passing.
2. **Model**:

$$\text{Hidden Output } 1 = \sigma(w_0 + w_1 x_0)$$

$$\text{Hidden Output } 2 = \sigma(w_2 + w_3 x_0)$$

$$\hat{y} = \sigma(w_4 + w_5 \cdot \text{Hidden Output } 1 + w_6 \cdot \text{Hidden Output } 2)$$

Where:

- $w_0, w_1, w_2, w_3, w_4, w_5, w_6$ are the weights of the network.
- **Sigmoid Activation**: $\sigma(z) = \frac{1}{1+e^{-z}}$.

Training the Neural Network:

Just like with logistic regression, we use **gradient descent** to update the weights. The difference here is that we have more weights (since there are hidden layers) and we need to compute the **gradients** for the loss with respect to each weight in the network.

The loss function will still be **cross-entropy loss**, and the gradients will be computed through **backpropagation**, which propagates the error backward through the network and adjusts the weights to minimize the error.

Example Final Model:

After training, the model might learn the following weights (simplified example):

- $w_0 = -2, w_1 = 1$ for the first hidden layer,
- $w_2 = -1, w_3 = 2$ for the second hidden layer,
- $w_4 = -3, w_5 = 1.5, w_6 = -1.2$ for the output layer.

Now, when we input a value (e.g., hours studied), the network performs multiple calculations:

- First, it calculates the hidden layer outputs.
- Then, it combines these hidden layer outputs to make the final prediction.

Key Differences:

1. **Logistic Regression**:
    - Simple, with a single linear decision boundary.
    - Easier to interpret.
    - Works well for simple problems.
2. **Neural Network**:
    - More complex, can handle non-linear relationships.
    - Uses multiple layers and activation functions to learn intricate patterns.
    - Requires more computational power and data to train effectively.

Summary:

- **Logistic Regression** is a simple, interpretable model for binary classification tasks (like predicting pass/fail based on hours studied).
- **Neural Networks** are more powerful models that can capture complex, non-linear relationships, but they are harder to interpret and require more data and computation to train.

Both models adjust their weights during training to minimize the error between predictions and actual outcomes using gradient descent, but **Neural Networks** do this with more layers and more complex relationships.

Train-Test Split

- **Why?** Prevents overfitting, evaluates performance.
- **Common Ratios:** 80-20, 70-30
- **Example (Python):**

python Copy code

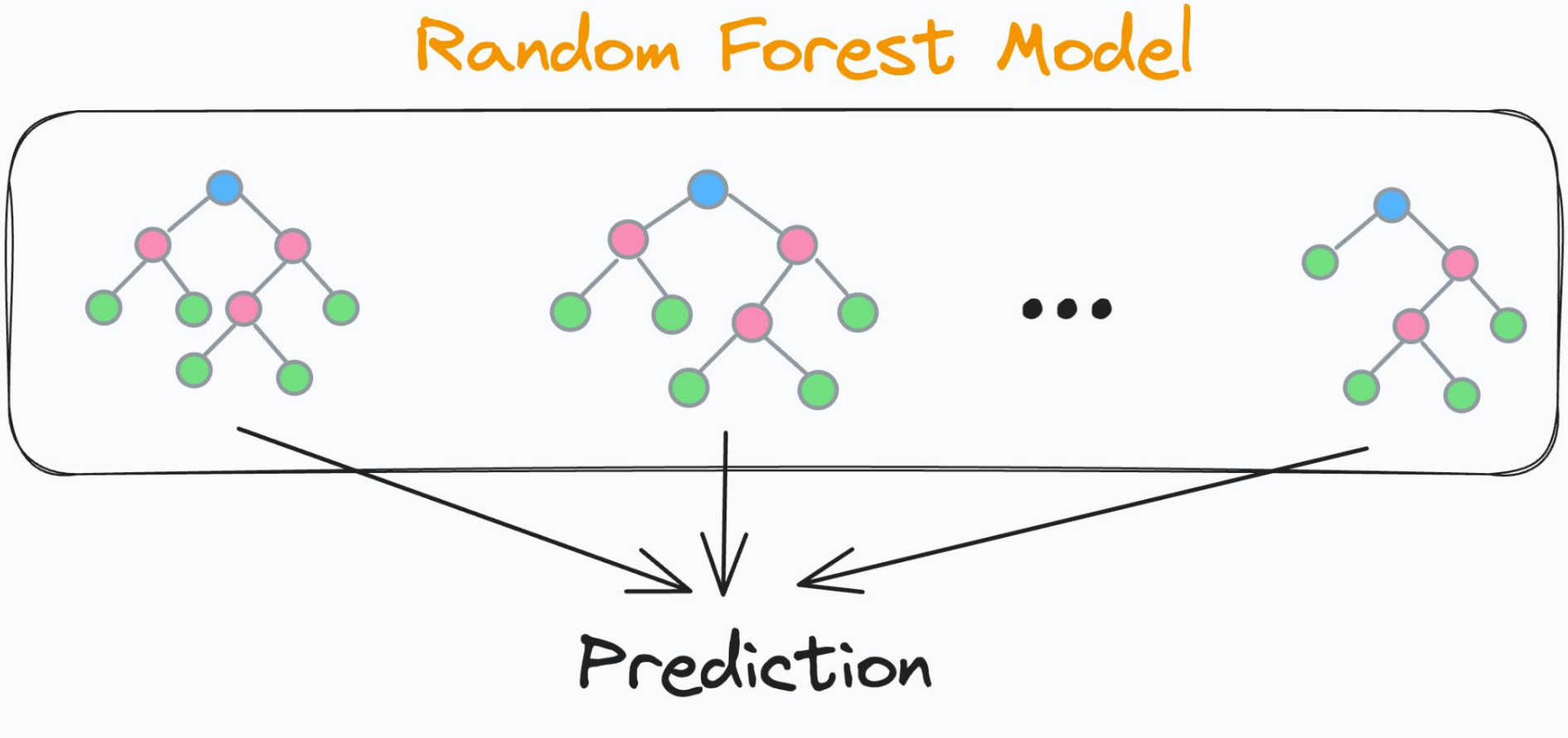
```python
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

Random Forest

- **Ensemble of Decision Trees**
- **Reduces overfitting**
- **Example (Python):**

```python
from sklearn.ensemble import RandomForestClassifier
clf = RandomForestClassifier(n_estimators=100)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
```

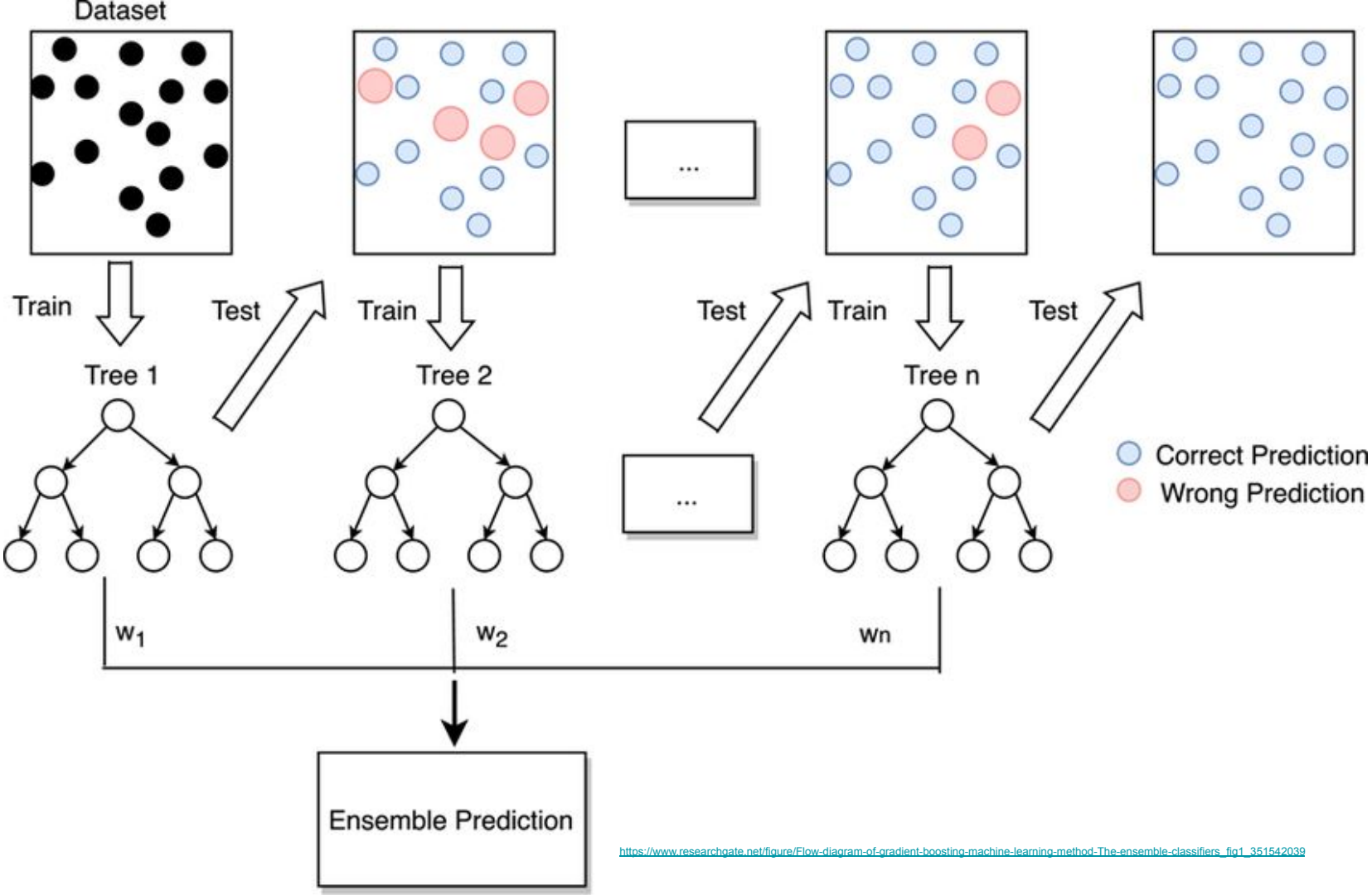
https://blog.dailydoseofds.com/p/your-random-forest-is-underperforming



Gradient Boosting

- **Sequential tree improvement**
- **Popular Variants:** XGBoost, LightGBM, CatBoost
- **Example (Python):**

```python
from sklearn.ensemble import GradientBoostingClassifier
clf = GradientBoostingClassifier(n_estimators=100, learning_rate=0.1)
clf.fit(X_train, y_train)
```

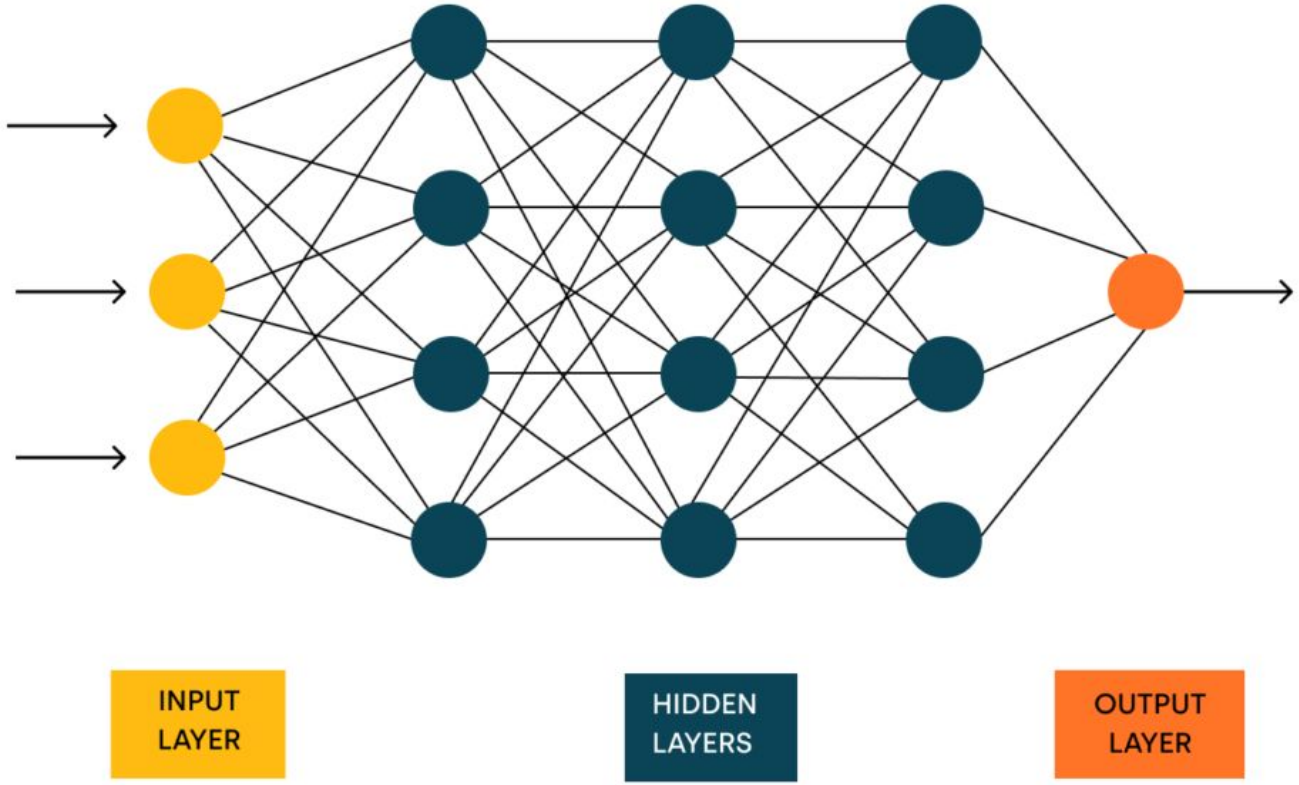


https://www.researchgate.net/figure/Flow-diagram-of-gradient-boosting-machine-learning-method-The-ensemble-classifiers_fig1_351542039

Neural Networks

- **Input → Hidden Layers → Output**
- **Example (PyTorch):**

```python
import torch.nn as nn
model = nn.Sequential(nn.Linear(10, 50), nn.ReLU(), nn.Linear(50, 1), nn.Sigmoid()
```


INPUT LAYER
HIDDEN LAYERS
OUTPUT LAYER

Confusion Matrix

Example Data:

Actual / Predicted	Positive	Negative
Positive (P)	TP = 56	FN = 18
Negative (N)	FP = 15	TN = 90

Python Code:

python

Copy code

```python
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
print(cm)
```

Metrics

- **Accuracy:**

$$\frac{TP + TN}{TP + FP + TN + FN} = \frac{56 + 90}{56 + 15 + 90 + 18} = 0.83$$

- **Precision:**

$$\frac{TP}{TP + FP} = \frac{56}{56 + 15} = 0.79$$

- **Recall:**

$$\frac{TP}{TP + FN} = \frac{56}{56 + 18} = 0.76$$

- **F1-Score:**

$$2 \times \frac{0.79 \times 0.76}{0.79 + 0.76} = 0.77$$

Machine Learning

Titanic

Objectives

- ✅ Understand the Titanic dataset
- ✅ Perform Exploratory Data Analysis (EDA)
- ✅ Preprocess the data for machine learning
- ✅ Train and evaluate multiple classification models
- ✅ Compare model performance and determine the best one

Introduction to the Titanic Dataset

- The Titanic dataset contains information on passengers aboard the Titanic
- The goal is to predict whether a passenger survived (1) or not (0)
- Features include demographic information, ticket details, and fare price

Dataset Dictionary

Feature	Description
PassengerId	Unique ID for each passenger
Survived	Target variable (1 = Survived, 0 = Not survived)
Pclass	Passenger class (1st, 2nd, 3rd)
Name	Passenger's full name
Sex	Gender (male/female)
Age	Age in years
SibSp	Number of siblings/spouses aboard
Parch	Number of parents/children aboard
Ticket	Ticket number
Fare	Ticket fare price
Cabin	Cabin number (if available)
Embarked	Port of embarkation (C = Cherbourg, Q = Queenstown, S = Southampton)

Step 1: Import Required Libraries

- Libraries for data manipulation and visualization: `pandas`, `numpy`, `seaborn`, `matplotlib`
- Machine learning libraries: `sklearn` for model training and evaluation

python Copy Edit

```python
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import confusion_matrix, classification_report, f1_score, precis
```

Step 2: Load the Titanic Dataset

- Load the dataset from a GitHub link
- Display the first few rows

python Copy Edit

```python
url = 'https://github.com/kaopanboonyuen/OCSB-AI/blob/main/dataset/titanic-dataset.cs
data = pd.read_csv(url)
data.head()
```

Step 3: Exploratory Data Analysis (EDA) 🔍

Checking for Missing Values

python Copy Edit

```python
data.isnull().sum()
```

- Identify missing values in the dataset

Visualizing Data Distributions 📊

python

Copy Edit

```python
sns.countplot(x='Survived', data=data)
plt.title('Distribution of Survived')
plt.show()

sns.histplot(data['Age'].dropna(), kde=True)
plt.title('Age Distribution')
plt.show()
```

Step 4: Data Preprocessing ⚙️

- Handle missing values in `Age` and `Embarked`
- Convert categorical features (`Sex` , `Embarked`) into numerical values

python Copy Edit

```python
data['Age'].fillna(data['Age'].mean(), inplace=True)
data['Embarked'].fillna(data['Embarked'].mode()[0], inplace=True)
data['Sex'] = data['Sex'].map({'male': 0, 'female': 1})
data = pd.get_dummies(data, columns=['Embarked'], drop_first=True)
```

- Select relevant features and define target variable

python Copy Edit

```python
X = data.drop(['Survived', 'Name', 'Ticket', 'Cabin', 'PassengerId'], axis=1)
y = data['Survived']
```

Step 5: Train-Test Split

- Split the data into training (80%) and testing (20%)

python Copy Edit

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state
```

Step 6: Train Machine Learning Models 🏗️

Random Forest Classifier

python Copy Edit

```python
rf_model = RandomForestClassifier(random_state=42)
rf_model.fit(X_train, y_train)
rf_pred = rf_model.predict(X_test)
print(classification_report(y_test, rf_pred, digits=4))
```

Gradient Boosting Classifier 🔥

python Copy Edit

```python
gb_model = GradientBoostingClassifier(random_state=42)
gb_model.fit(X_train, y_train)
gb_pred = gb_model.predict(X_test)
print(classification_report(y_test, gb_pred, digits=4))
```

Neural Network Classifier 🏆

python Copy Edit

```python
nn_model = MLPClassifier(random_state=42)
nn_model.fit(X_train, y_train)
nn_pred = nn_model.predict(X_test)
print(classification_report(y_test, nn_pred, digits=4))
```

Step 7: Confusion Matrix and Model Evaluation 📊

Confusion Matrix for Each Model

python Copy Edit

```python
sns.heatmap(confusion_matrix(y_test, rf_pred), annot=True, fmt='d', cmap='Blues')
plt.title("Random Forest Confusion Matrix")
plt.show()

sns.heatmap(confusion_matrix(y_test, gb_pred), annot=True, fmt='d', cmap='Blues')
plt.title("Gradient Boosting Confusion Matrix")
plt.show()

sns.heatmap(confusion_matrix(y_test, nn_pred), annot=True, fmt='d', cmap='Blues')
plt.title("Neural Network Confusion Matrix")
plt.show()
```

Step 8: Performance Comparison 🏆

F1-Score, Precision, and Recall

python Copy Edit

```python
print(f"Random Forest F1-Score: {f1_score(y_test, rf_pred):.4f}")
print(f"Gradient Boosting F1-Score: {f1_score(y_test, gb_pred):.4f}")
print(f"Neural Network F1-Score: {f1_score(y_test, nn_pred):.4f}")
```

python Copy Edit

```python
print(f"Random Forest Precision: {precision_score(y_test, rf_pred):.4f}, Recall: {rec
print(f"Gradient Boosting Precision: {precision_score(y_test, gb_pred):.4f}, Recall:
print(f"Neural Network Precision: {precision_score(y_test, nn_pred):.4f}, Recall: {re
```

Step 9: Model Deployment 🚀

- Save and load the best model

python Copy Edit

```python
import joblib
joblib.dump(rf_model, 'mars_random_forest.pth')
rf_model_loaded = joblib.load('mars_random_forest.pth')
```

Inference with Sample Data

python Copy Edit

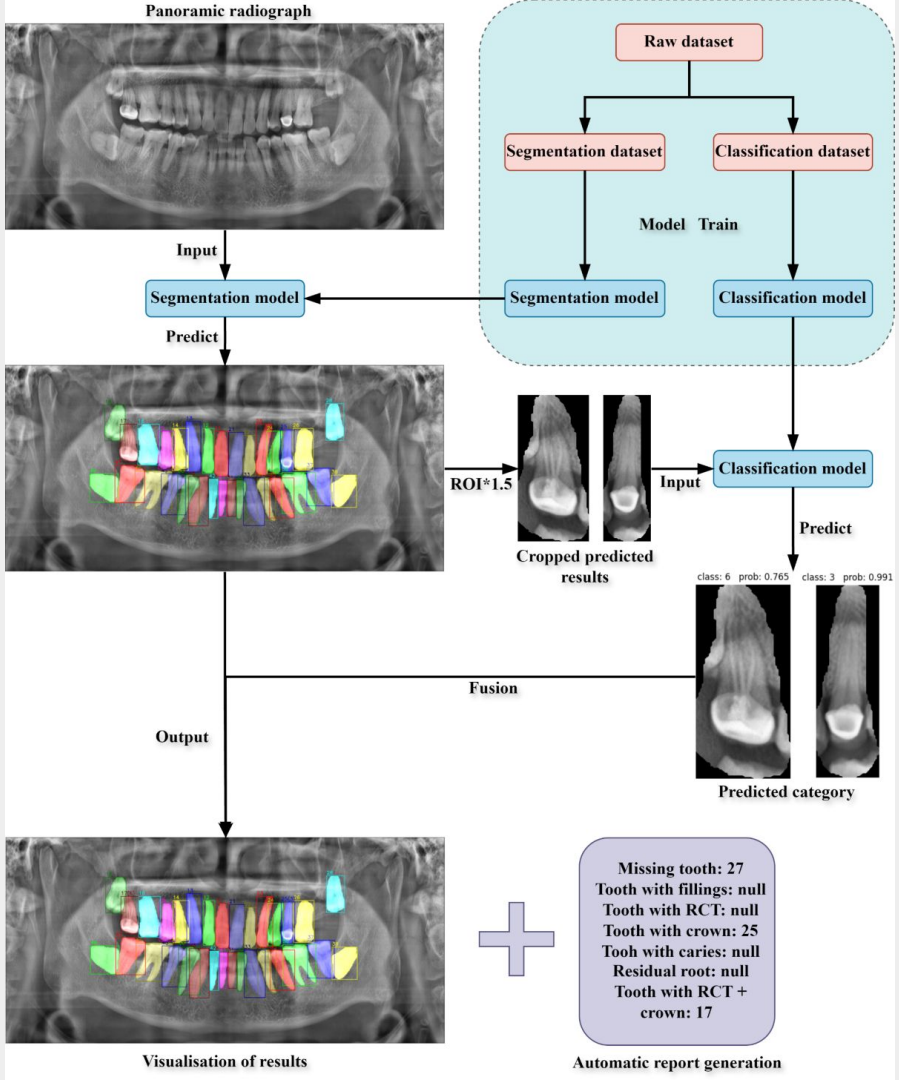
```python
mars_data = pd.DataFrame({
    'Pclass': [3], 'Sex_male': [1], 'Age': [30],
    'SibSp': [0], 'Parch': [0], 'Fare': [7.5],
    'Embarked_Q': [0], 'Embarked_S': [1]
})

# Ensure feature alignment
missing_cols = set(X_train.columns) - set(mars_data.columns)
for col in missing_cols:
    mars_data[col] = 0

mars_data = mars_data[X_train.columns]
prediction = rf_model_loaded.predict(mars_data)[0]
result = "Survived 😃" if prediction == 1 else "Did not survive 😔"
print(f"Prediction: {result}")
```

Deep Learning

Dental Radiography

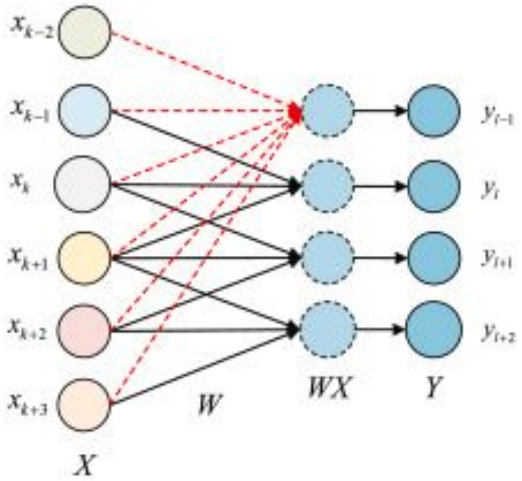


Basic Vision Network Architecture for Satellite Classification

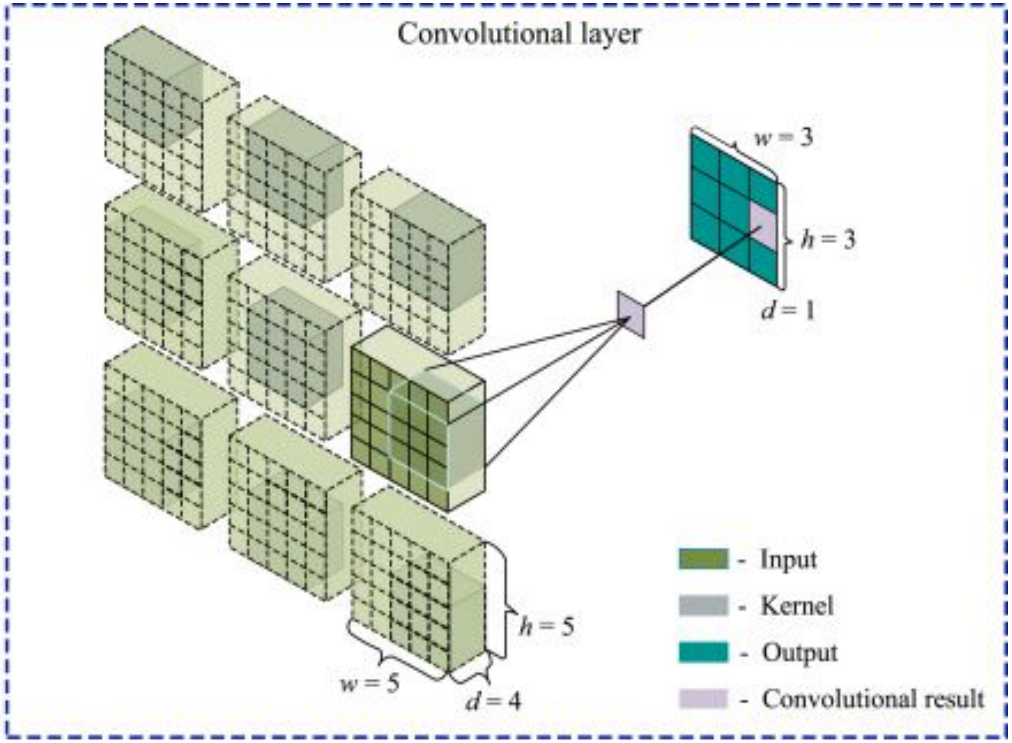
1. **Input Image:** Satellite image of a specific region.
2. **Convolution Layer:** Extracts spatial features.
3. **Pooling Layer:** Reduces feature map size.
4. **Attention Layer:** Focuses on key areas.
5. **Fully Connected Layers + Softmax:** Final classification.

Convolutional Neural Networks (CNNs)

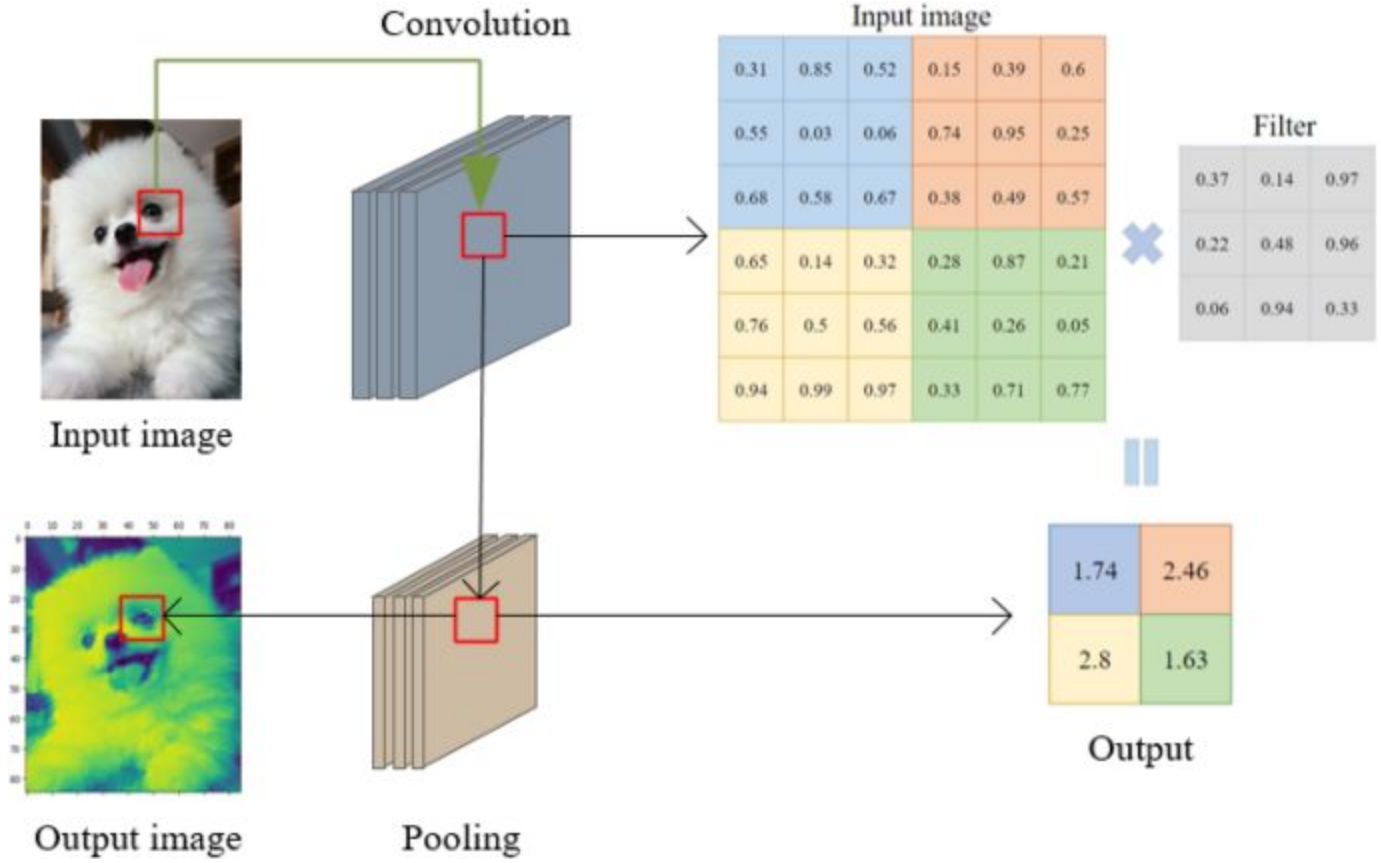
- **Uses:** Image classification, object detection, segmentation
- **Layers:** Convolution → Activation → Pooling → Fully Connected

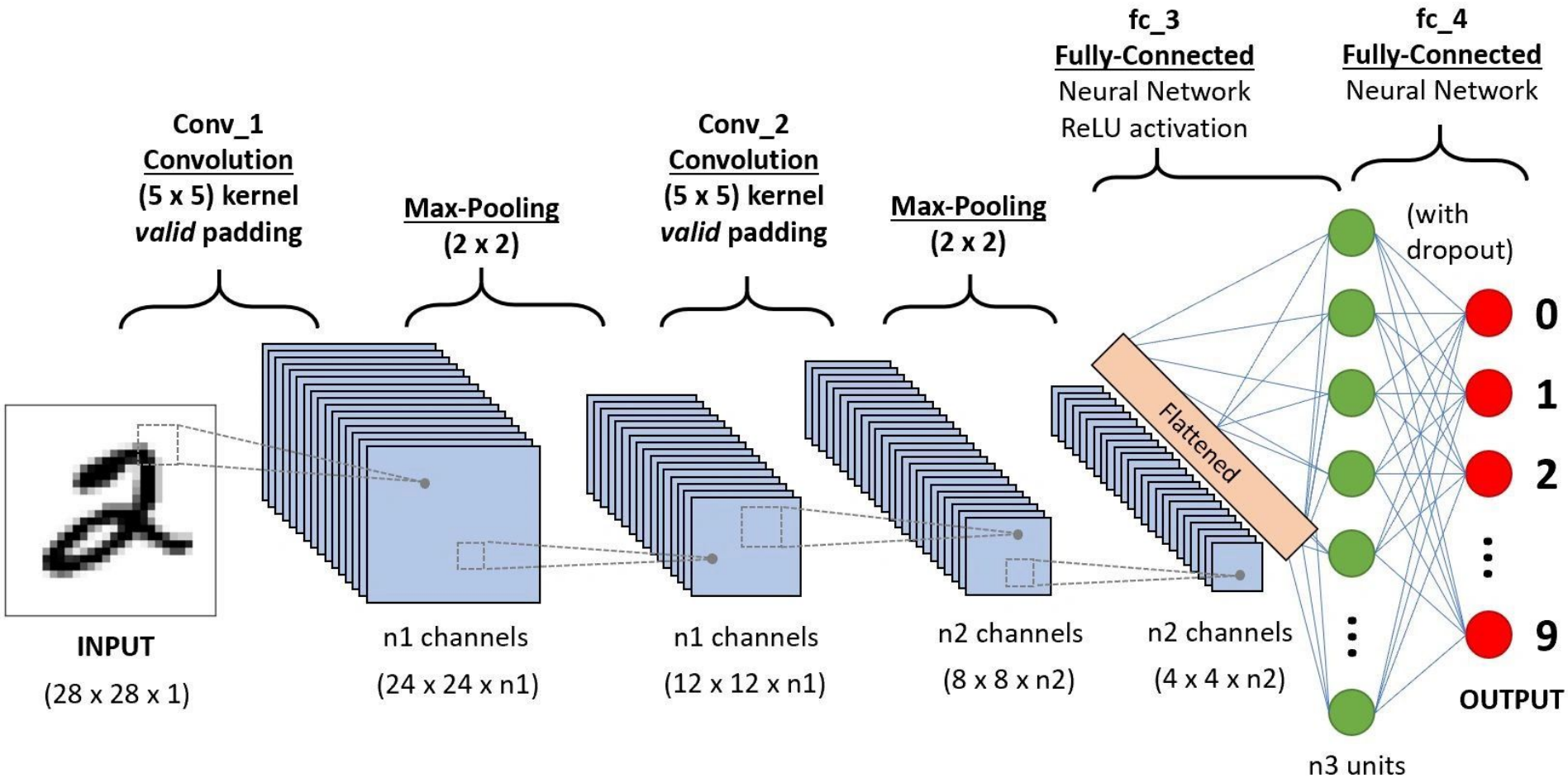

x_{k-2}
x_{k-1}
x_k
x_{k+1}
x_{k+2}
x_{k+3}
y_{t-1}
y_t
y_{t+1}
y_{t+2}
W
WX
Y
X

(a)


Convolutional layer
w = 3
h = 3
d = 1
h = 5
w = 5
d = 4
- Input
- Kernel
- Output
- Convolutional result

(b)


Convolution
Input image
Input image
0.31 0.85 0.52 0.15 0.39 0.6
0.55 0.03 0.06 0.74 0.95 0.25
0.68 0.58 0.67 0.38 0.49 0.57
0.65 0.14 0.32 0.28 0.87 0.21
0.76 0.5 0.56 0.41 0.26 0.05
0.94 0.99 0.97 0.33 0.71 0.77
Filter
0.37 0.14 0.97
0.22 0.48 0.96
0.06 0.94 0.33
1.74 2.46
2.8 1.63
Output
Output image
Pooling


Conv_1
Convolution
(5 x 5) kernel
valid padding
Max-Pooling
(2 x 2)
Conv_2
Convolution
(5 x 5) kernel
valid padding
Max-Pooling
(2 x 2)
fc_3
Fully-Connected
Neural Network
ReLU activation
fc_4
Fully-Connected
Neural Network
(with dropout)
Flattened
INPUT
(28 x 28 x 1)
n1 channels
(24 x 24 x n1)
n1 channels
(12 x 12 x n1)
n2 channels
(8 x 8 x n2)
n2 channels
(4 x 4 x n2)
n3 units
0
1
2
9
OUTPUT

Convolution Layer

Formula to Compute Output Size:

$$O = \frac{(I - K + 2P)}{S} + 1$$

Where:

- I = Input size (Height/Width)
- K = Kernel size
- P = Padding
- S = Stride
- O = Output size

Example Calculation:

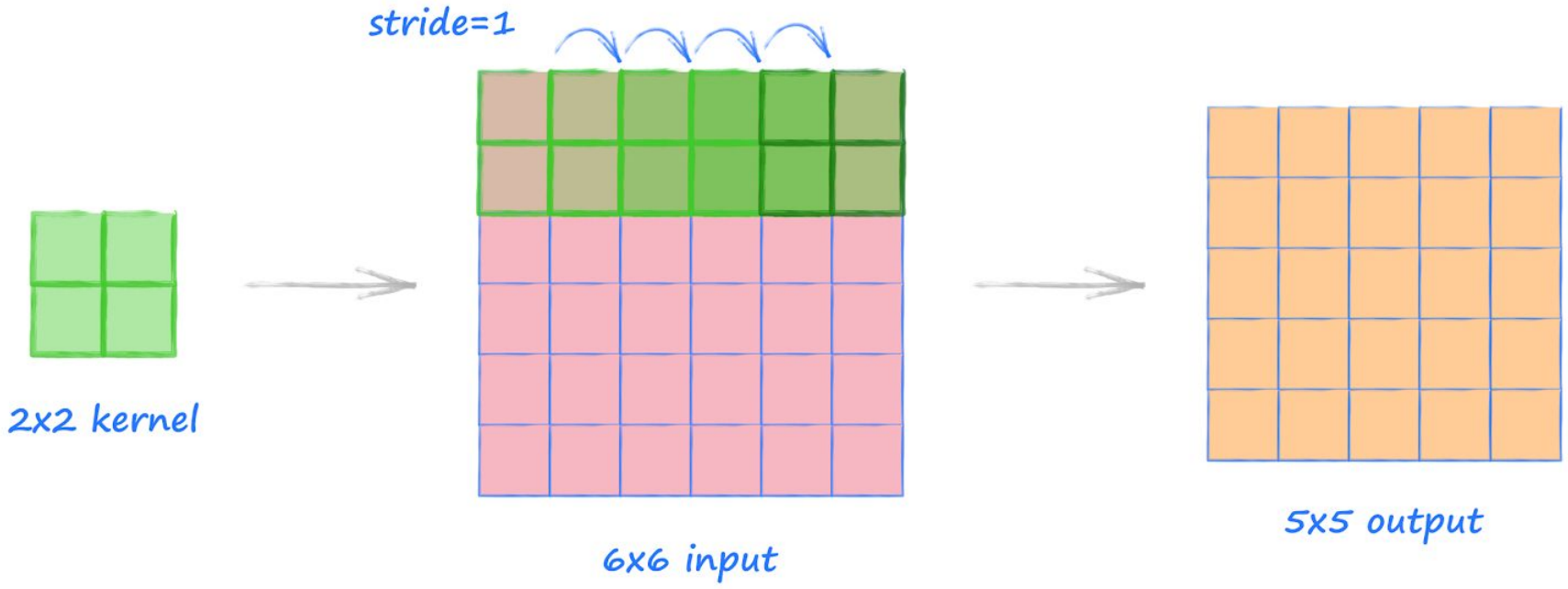
Input: 32×32 image, **Kernel:** 3×3, **Stride:** 1, **Padding:** 1

$$O = \frac{(32 - 3 + 2(1))}{1} + 1 = \frac{(32 - 3 + 2)}{1} + 1 = 32$$

💡 **Same size output when** $P = \frac{(K-1)}{2}$

Stride & Padding

- **Stride:** How far the filter moves
  - **Stride = 1** → Small movement, detailed features
  - **Stride = 2** → Faster, lower resolution
- **Padding:** Adds zeros around input
  - **Valid (No Padding, P=0)** → Output shrinks
  - **Same (P=1 for 3×3 filter)** → Keeps size same


stride=1
2x2 kernel
6x6 input
5x5 output

Pooling Layer (Downsampling)

Max Pooling (Example 2×2, Stride 2)

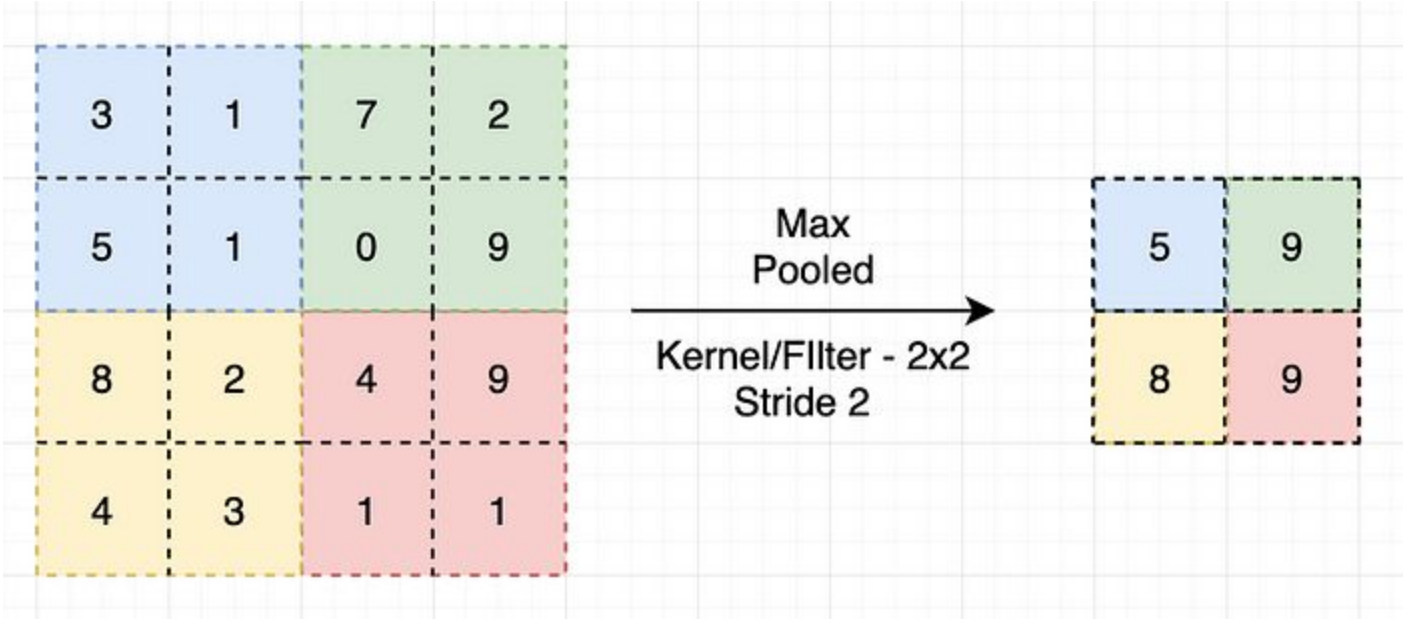
- Takes the **maximum** value in each 2×2 region
- **Reduces size, retains important features**

Formula for Output:

$$O = \frac{(I - K)}{S} + 1$$

Example: **Input 32×32, Pooling 2×2, Stride 2**

$$O = \frac{(32 - 2)}{2} + 1 = 16$$


3
1
7
2
5
1
0
9
8
2
4
9
4
3
1
1
Max
Pooled
Kernel/FIlter - 2x2
Stride 2
5
9
8
9

Fully Connected Layer

- **Flattens features** → Passes to Dense Layers
- **Example:**
    - Output from CNN: **512 values**
    - Dense layer: **256 neurons**
    - Weights: **512×256**
    - **Output:** 512×256 dot product + bias

Activation Functions

- **ReLU (Rectified Linear Unit):** $f(x) = \max(0, x)$
  - Introduces non-linearity
- **Softmax (for multi-class classification)**

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

  - Converts logits into probabilities

Dropout (Regularization)

- **Prevents overfitting** by randomly setting neurons to **0** during training
- Example: **Dropout(0.5) → 50% neurons inactive**

- **Objective**: Implement a machine learning classification model using PyTorch to classify dental images into various categories.
- **Tools**: PyTorch, torchvision, Matplotlib, seaborn, Scikit-learn, etc.
- **Dataset**: Dental images with several classes (e.g., various dental conditions).
- **Models Used**:
  - Simple CNN (from scratch)
  - ResNet18 (Pretrained)
  - DenseNet (Pretrained)
- **Performance Evaluation**: Accuracy, Confusion Matrix, Precision, Recall, F1-score.

Dataset Description

- **Dataset Name**: Dental Dataset (stored in `dental_dataset.zip`)
- **Classes**: Categories representing different dental conditions.
  - Example classes: `Cavity`, `Healthy`, `Gum Disease`, etc.
- **Structure**:
  - The dataset is organized in subdirectories, where each subdirectory corresponds to a specific class.
  - The images are loaded using the `ImageFolder` method from `torchvision.datasets`.
- **Splitting**: 70% training, 15% validation, 15% test.

Step 1: Installing and Importing Libraries

python Copy Edit

```
!pip install torch==2.0.0+cu117 torchvision torchaudio --index-url https://download.p
!pip install --upgrade torchvision

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms
import torchvision.datasets as datasets
import torchvision.models as models
from torch.utils.data import DataLoader, random_split
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
```

- **Key Libraries**:
  - `torch`, `torchvision`: Deep learning framework and pre-trained models.
  - `matplotlib`, `seaborn`: For visualization of results.
  - `sklearn.metrics`: For performance evaluation (accuracy, precision, recall, etc.).

Step 2: Load and Prepare the Dataset

- **Dataset Path**: `/content/dataset/dental_dataset`
- **Transformations**: Resize images to 224x224 and convert to tensor.

python Copy Edit

```python
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
])

dataset = datasets.ImageFolder(root=dataset_path, transform=transform)
```

- **Dataset Split**:
  - 70% for training
  - 15% for validation
  - 15% for testing

python Copy Edit

```python
train_size = int(0.7 * len(dataset))
val_size = int(0.15 * len(dataset))
test_size = len(dataset) - train_size - val_size

train_dataset, val_dataset, test_dataset = random_split(dataset, [train_size, val_siz
```

Step 3: Data Preview

- Visualize random samples from the training dataset.

python Copy Edit

```python
def show_images(dataset, num=6):
    fig, axes = plt.subplots(1, num, figsize=(15, 5))
    indices = random.sample(range(len(dataset)), num)
    for i, idx in enumerate(indices):
        img, label = dataset[idx]
        axes[i].imshow(img.permute(1, 2, 0))
        axes[i].set_title(classes[label])
        axes[i].axis('off')
    plt.show()

show_images(train_dataset)
```

- **Visualized Output**: Displays 6 random images from the dataset with corresponding labels.

Step 4: Define Models

- **Model 1: Simple CNN from Scratch**

  A basic Convolutional Neural Network architecture with two convolutional layers.

python Copy Edit

```python
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(64 * 56 * 56, 128)
        self.fc2 = nn.Linear(128, len(classes))

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

- **Model 2: Pretrained ResNet18**

  Utilizing the pretrained ResNet18 model from torchvision.

```python
model_resnet = models.resnet18(pretrained=True)
for param in model_resnet.parameters():
    param.requires_grad = False
model_resnet.fc = nn.Linear(model_resnet.fc.in_features, len(classes))
```

- **Model 3: Pretrained DenseNet**

  Similar to ResNet, but using DenseNet for potentially better feature extraction.

```python
model_densenet = models.densenet121(pretrained=True)
for param in model_densenet.parameters():
    param.requires_grad = False
model_densenet.classifier = nn.Linear(model_densenet.classifier.in_features, len(clas
```

Step 5: Training the Model

- **Training Function**: Define a function to train the model and log the losses and accuracies.

python Copy Edit

```python
def train_model(model, train_loader, val_loader, epochs=10, learning_rate=0.001):
    # Training loop here...
```

- **Training Models**: Train all three models (SimpleCNN, ResNet, DenseNet) using the training function.

python Copy Edit

```python
train_model(model1, train_loader, val_loader)
train_model(model_resnet, train_loader, val_loader)
train_model(model_densenet, train_loader, val_loader)
```

Step 6: Evaluate Model Performance

- **Metrics**: Accuracy, Precision, Recall, F1-Score.

```python
def evaluate_model(model, test_loader, classes):
    # Evaluation code...
```

- **Confusion Matrix**: Visualize predictions and true labels using a confusion matrix.

```python
def plot_confusion_matrix(true_labels, predictions, classes):
    # Plot confusion matrix code...
```

- **Output**: Classification report and confusion matrix for each model.

Step 7: Save, Load, and Inference

- **Saving Models**: Save the trained model for future use.

python Copy Edit

```
torch.save(model_densenet.state_dict(), 'DenseNet_model.pth')
```

- **Loading Models**: Load the saved model weights for inference.

python Copy Edit

```
model_densenet.load_state_dict(torch.load('DenseNet_model.pth'))
```

Step 8: Inference on New Data

- **Inference Function**: Use the trained model to predict new data.

```python
def imshow(image):
    # Code to unnormalize and display the image...
```

- **Display Prediction Results**: Display images along with the predicted and actual labels.

```python
for inputs, labels in test_loader:
    # Perform inference and display results...
```