DEEP LEARNING – UNIT I: *Foundations of Deep Learning*

1. What is Machine Learning (ML) and Deep Learning (DL)?

- **Machine Learning** is a subset of AI where systems learn from data using statistical techniques.
- **Deep Learning** is a subset of ML that uses **multi-layered artificial neural networks** to learn from vast amounts of data.
- DL automates **feature extraction**, unlike traditional ML which relies on manual feature engineering.

---

2. Supervised vs. Unsupervised Learning

Criteria	Supervised Learning	Unsupervised Learning
Data	Labeled	Unlabeled
Goal	Predict output	Find patterns or structure
Example	Spam detection	Customer segmentation

---

3. Bias-Variance Tradeoff

- **Bias**: Error due to overly simplistic model assumptions (underfitting).
- **Variance**: Error from model complexity (overfitting).
- **Tradeoff**: The goal is to minimize both to improve generalization on unseen data.

---

4. Hyperparameters in Deep Learning

- These are **configurable values** set before training.
- Examples:
  - Learning rate
  - Batch size
  - Number of epochs
  - Number of layers and neurons
  - Dropout rate
- Tuned using techniques like **grid search**, **random search**, or **Bayesian optimization**.

---

5. Underfitting and Overfitting

- **Underfitting**: Model is too simple; poor accuracy on training and test data.
- **Overfitting**: Model learns noise from training data; performs poorly on test data.
- **Solution**: Use regularization, dropout, more data, or simpler models.

---

6. Regularization Techniques

- **L1 (Lasso)**: Adds absolute value of weights to loss function; leads to sparse models.
- **L2 (Ridge)**: Adds square of weights; avoids large weights.
- **Dropout**: Randomly turns off neurons during training.

- Helps reduce overfitting and improve generalization.

---

7. Limitations of Traditional Machine Learning

- Requires extensive **manual feature engineering**
- Struggles with **unstructured data** (images, text, audio)
- Limited scalability for **large datasets**
- Inadequate for tasks like speech recognition or image captioning

---

8. History of Deep Learning

- 1943: McCulloch-Pitts neuron model
- 1958: Perceptron by Rosenblatt
- 1980s: Backpropagation popularized
- 2006: Geoffrey Hinton revived DL with Deep Belief Networks
- 2012: AlexNet won ImageNet competition, bringing DL to mainstream

---

9. Advantages and Challenges of Deep Learning

Advantages:

- Automates feature learning
- High accuracy in image/audio/text tasks
- Scalable with data and computation

Challenges:

- Requires large labeled datasets
- Needs high computing power (GPUs/TPUs)
- Lacks interpretability
- Can overfit easily

---

10. Learning Representations from Data

- DL learns **hierarchical representations**:
  - Low-level: Edges, colors
  - Mid-level: Shapes, patterns
  - High-level: Objects, semantics
- Eliminates the need for manual feature engineering

---

11. How Deep Learning Works (Explained in Three Figures)

1. **Neuron Model**: Shows how a single artificial neuron computes output.
2. **Layered Network**: Input → Hidden Layers → Output

3. **Training Flow**: Forward pass, loss computation, backpropagation, weight updates

---

12. Common Architectural Principles of Deep Networks

- **Layered Design**: Input, hidden, and output layers
- **Non-linear Activations**: ReLU, Tanh, Sigmoid
- **Parameter Sharing**: Especially in CNNs
- **Skip Connections**: Like in ResNets, to solve vanishing gradient issues
- **Modularity**: Networks built from reusable blocks

---

13. Architecture Design in Deep Learning

- Factors:
  - Number of layers
  - Type of layers (dense, conv, RNN)
  - Activation functions
  - Loss functions
  - Initialization methods
- Trade-off between **model depth, accuracy, and computational cost**

---

14. Applications of Deep Learning

- **Computer Vision**: Object detection, face recognition
- **Natural Language Processing**: Translation, sentiment analysis
- **Speech Recognition**: Voice assistants like Siri
- **Healthcare**: Disease prediction, medical imaging
- **Finance**: Fraud detection, credit scoring

---

15. Popular Deep Learning Tools

Tool	Description
TensorFlow	Open-source ML library by Google
Keras	High-level API for building DL models (runs on TensorFlow)
PyTorch	Flexible dynamic graph framework by Facebook
Caffe	CNN-focused library, fast deployment
Shogun	General ML library (supports SVMs, kernels, etc.)

---

16. Exemplar/Case Studies

- **AlphaGo (DeepMind)**: Reinforcement learning model that defeated world Go champion.
- **Boston Dynamics**: Uses DL in robotic locomotion and perception.
- **DeepMind's Research**: Pioneers in medical diagnosis and protein folding (AlphaFold).

DEEP LEARNING – UNIT II: *Deep Neural Networks (DNNs)*

1. What is a Neural Network?

- Composed of interconnected **artificial neurons**.
- **Each neuron** receives inputs, applies weights and bias, then passes output via activation function.
- Learns to map inputs to desired outputs by adjusting weights.

---

2. Biological vs Artificial Neurons

- Biological neurons transmit signals via synapses.
- Artificial neurons compute:

  y=f(w1x1+w2x2+...+b)y = f(w_1x_1 + w_2x_2 + ... + b)y=f(w1x1+w2x2+...+b)

  where fff is activation function

---

3. What is a Perceptron?

- **Single-layer neural network** used for binary classification.
- Can't solve non-linear problems like XOR.

---

4. What is a Multilayer Feedforward Network?

- Consists of multiple layers (input, hidden, output).
- **Feedforward**: Data flows in one direction, no cycles.

---

5. Forward and Backward Propagation

- **Forward Propagation**: Calculates predictions
- **Backward Propagation**: Computes gradients using chain rule and updates weights

---

6. Activation Functions

Function	Description
Linear	No non-linearity
Sigmoid	Output between 0 and 1; suffers from vanishing gradient
Tanh	Output between -1 and 1
Hard Tanh	Approximated version of tanh
ReLU	Faster convergence, avoids vanishing gradient
Softmax	Outputs probability distribution for classification

---

7. Loss Functions

- Used to compute error between predicted and true values
Type	Example
Regression	Mean Squared Error (MSE), MAE
Classification	Cross-Entropy Loss
Reconstruction	Binary cross-entropy, Kullback–Leibler divergence

---

8. Gradient Descent & Variants

- **Gradient Descent**: Minimizes loss function by updating weights
- **Momentum**: Helps accelerate descent in the right direction
- **Stochastic GD**: Updates weights for each training example
- **Mini-batch GD**: Compromise between batch and stochastic GD

---

9. Vanishing and Exploding Gradients

- **Vanishing**: Gradients shrink → no learning in early layers (tanh/sigmoid cause this)
- **Exploding**: Gradients blow up → unstable training
- **Solutions**: ReLU, normalization, proper initialization, residual networks

---

10. Example: XOR using Neural Networks

- **XOR is non-linear**; cannot be solved by a perceptron.
- Solved using a neural network with:
  - 2 input nodes
  - 1 hidden layer with non-linear activation
  - 1 output node

---

11. Deep Feedforward Networks

- Also called Multi-Layer Perceptrons (MLPs)
- Layers: Input → Hidden layers (non-linear transformations) → Output

---

12. Hyperparameters in Training

- **Learning Rate**: Step size in weight update
- **Epochs**: Full pass through training data
- **Batch size**: Number of samples processed before weight update
- **Regularization**: Prevents overfitting
- **Momentum**: Speeds up convergence

---

13. Tools for Implementing DNNs

- **PyTorch**: Dynamic graphs, great for research
- **Google Colab**: Free GPU/TPU for training models
- **Jupyter Notebook**: Easy to write and visualize code

---

14. Case Study: Music Genre Classification

- Input: Audio features (MFCCs, tempo, rhythm)
- Model: Deep neural network or CNN
- Task: Classify music into genres like rock, jazz, classical
- Dataset: GTZAN music genre dataset (example)

DEEP LEARNING – Unit III: *Convolutional Neural Networks (CNN)*

1. Introduction to CNN

1. CNNs are a type of deep learning model designed specifically for processing grid-like data, such as images.
2. They are used for computer vision tasks like image classification, object detection, and segmentation.
3. CNNs take advantage of the spatial structure in images by using a mathematical operation called convolution.
4. CNNs consist of multiple layers, each learning different levels of abstraction from the raw image data.

2. CNN Architecture Overview

5. A typical CNN architecture includes convolutional layers, pooling layers, fully connected layers, and sometimes normalization layers.
6. The architecture is designed to reduce the dimensionality of images while preserving important features.
7. The primary purpose of CNNs is to capture spatial hierarchies of patterns in images.

3. The Basic Structure of a Convolutional Network

8. **Padding**: Adding extra pixels to the borders of input images to maintain spatial dimensions during convolution.
9. **Strides**: Determines how much the filter moves across the input image. Strides control the output size of the convolution.
10. **Valid Padding**: No padding, output is smaller than input.
11. **Same Padding**: Padding is added so the output size matches the input size.
12. **Filters (Kernels)**: Small filters (typically 3x3 or 5x5) are used to extract features from the input.

4. ReLU Layer (Rectified Linear Unit)

13. The ReLU activation function replaces all negative values with zero.
14. It introduces non-linearity into the network, allowing CNNs to learn more complex patterns.
15. The ReLU function is defined as: f(x)=max⁡(0,x)f(x) = \max(0, x)f(x)=max(0,x).

5. Pooling Layer

16. Pooling layers reduce the spatial dimensions of feature maps to decrease computation and control overfitting.
17. **Max Pooling**: Takes the maximum value from each region of the feature map.
18. **Average Pooling**: Averages the values within each region of the feature map.

6. Fully Connected Layers

19. After convolution and pooling layers, the final layers are usually fully connected layers that perform classification or regression tasks.
20. Each neuron in a fully connected layer is connected to every neuron in the previous layer.

7. Local Response Normalization (LRN)

21. LRN normalizes the output from a neuron with respect to its neighbors.
22. Helps CNNs generalize better by penalizing large activations and improving performance in deeper networks.

8. Training a CNN

23. CNNs are trained using backpropagation and gradient descent to minimize the error between predictions and actual outcomes.
24. Typically, training is done with a large dataset and can be accelerated using GPUs.
25. **Hyperparameter tuning**: Important parameters such as learning rate, batch size, and number of epochs need to be optimized.

9. Exemplar/Case Studies

26. **AlexNet**: First deep CNN to win the ImageNet competition; significantly outperformed traditional methods.
27. **VGG**: Uses deeper layers (up to 19 layers) with smaller filters, and was successful in object detection.

DEEP LEARNING – Unit IV: *Recurrent and Recursive Neural Networks*

1. Introduction to RNNs

1. Recurrent Neural Networks (RNNs) are a class of neural networks designed for sequential data.
2. Unlike traditional feedforward networks, RNNs have connections that allow them to maintain memory from previous time steps.
3. RNNs are widely used for tasks like speech recognition, language modeling, and time series prediction.

2. Unfolding Computational Graphs

4. RNNs are unfolded over time steps to visualize how information flows through the network at each time step.
5. Each node in an RNN corresponds to a time step in the sequence, and each time step shares parameters.

3. Recurrent Neural Networks

6. RNNs take input at each time step and update the hidden state based on the current input and the previous hidden state.
7. The output of the network at each time step is influenced by the sequence of past inputs.

4. Bidirectional RNNs

8. Bidirectional RNNs consist of two RNN layers: one processes the sequence forward, and the other processes it backward.
9. This is beneficial when both past and future information are required for tasks like text translation.

5. Encoder-Decoder Architecture

10. The **encoder** processes the input sequence and compresses it into a fixed-size vector (called the context vector).
11. The **decoder** generates the output sequence from the context vector, typically used in sequence-to-sequence tasks like machine translation.

6. Deep Recurrent Networks

12. Stacking multiple RNN layers can lead to deeper networks that learn complex patterns in sequential data.
13. Deep RNNs can be used for more advanced sequence generation tasks.

7. Recursive Neural Networks

14. Recursive Neural Networks are designed to process hierarchical structured data, such as parse trees.
15. They are useful for tasks like sentiment analysis and syntactic parsing in natural language processing.

8. The Challenge of Long-Term Dependencies

16. Traditional RNNs struggle with long-term dependencies due to the vanishing and exploding gradient problem.
17. Information from earlier in the sequence can become "forgotten" or have its influence reduced as it propagates through time steps.

9. Echo State Networks (ESNs)

18. ESNs use a fixed, randomly initialized recurrent layer and only train the output layer, making them faster to train.
19. They are often used for time series prediction and other sequence-based tasks.

10. Leaky Units and Other Strategies

20. Leaky units help alleviate the vanishing gradient problem by allowing a small proportion of the signal to "leak" through.
21. Leaky ReLU and other activation functions like the ELU (Exponential Linear Unit) are used to help RNNs learn more effectively.

11. The Long Short-Term Memory (LSTM)

22. LSTM is a specialized type of RNN designed to combat the vanishing gradient problem.
23. LSTMs use memory cells to store information over long periods and gates to control the flow of information (input, forget, and output gates).

24. LSTM networks are ideal for tasks that involve long sequences, such as machine translation and speech recognition.

12. Gated Recurrent Units (GRU)

25. GRUs are a simpler variant of LSTM, with fewer gates and thus fewer parameters to train.
26. GRUs combine the forget and input gates into a single "update gate" and are often used in machine translation and speech recognition tasks.

13. Optimization for Long-Term Dependencies

27. Techniques like gradient clipping, appropriate weight initialization, and using gated units (LSTM/GRU) help address the issues with long-term dependencies.
28. Proper learning rate schedules and regularization methods like dropout can also help in optimizing performance for long-term tasks.

14. Explicit Memory Mechanisms

29. Neural Turing Machines (NTMs) and Memory Networks introduce explicit memory, allowing networks to store and retrieve information more effectively over long sequences.
30. These architectures extend the capabilities of RNNs by providing external memory that can be directly addressed during training.

15. Practical Methodology

31. **Performance Metrics**: For sequential tasks, metrics like accuracy, precision, recall, F1 score, and BLEU (for machine translation) are used.
32. **Default Baseline Models**: Start with simple models like logistic regression or a shallow RNN to serve as a baseline before moving to more complex models.
33. **Data Augmentation**: Techniques such as random cropping, rotation, and flipping can improve model generalization by increasing the diversity of training data.
34. **Hyperparameter Tuning**: Learning rate, batch size, number of layers, and sequence length need to be optimized through experimentation or automated search methods (e.g., grid search or random search).

16. Exemplar/Case Studies

35. **Multi-Digit Number Recognition**: A case study that combines CNNs and RNNs for digit recognition from images (e.g., house number recognition in street images).

DEEP LEARNING – Unit V: *Deep Generative Models*

1. Introduction to Deep Generative Models

1. Deep generative models are a class of models in deep learning that generate new data instances similar to the training data.
2. These models learn the underlying distribution of the data to generate new samples that resemble the original dataset.
3. Common applications include image synthesis, text generation, and music composition.
4. Generative models are often compared with discriminative models, which are focused on classifying data into categories.

2. Boltzmann Machines

5. The **Boltzmann Machine (BM)** is a type of stochastic recurrent neural network that learns to model probability distributions over binary vectors.
6. It uses energy-based learning, where the system aims to minimize the energy of the configuration to learn the distribution of input data.
7. Boltzmann Machines are mainly used for feature learning and building probabilistic models of data.

3. Deep Belief Networks (DBN)

8. **DBNs** are a class of generative models composed of multiple layers of stochastic, latent variables, with connections between the layers.
9. DBNs consist of multiple **Restricted Boltzmann Machines (RBMs)** stacked together, allowing for a deep architecture.
10. DBNs are used in unsupervised learning to learn hierarchical representations of data.
11. They are effective for tasks such as dimensionality reduction, classification, and feature extraction.

4. Generative Adversarial Networks (GAN)

12. **GANs** consist of two neural networks: the **generator** and the **discriminator**.
13. The **generator** tries to create fake data (e.g., fake images), while the **discriminator** tries to differentiate between real and fake data.
14. GANs are trained in a zero-sum game where the generator improves to fool the discriminator, and the discriminator improves to detect fake data.
15. The objective is for the generator to generate data indistinguishable from real data, which can be used in image, video, and text generation.

5. Discriminator Network

16. The **discriminator** network's role is to classify input data as real (from the training set) or fake (from the generator).
17. The discriminator is typically a binary classifier that learns to distinguish between real and generated data.
18. Its feedback helps the generator adjust and improve the quality of generated data.

6. Generator Network

19. The **generator** network creates new data based on random input, often called latent vectors or noise.
20. It aims to generate data that looks like the real data, fooling the discriminator.
21. The generator typically uses layers like fully connected, deconvolutional, and activation functions (e.g., ReLU) to create realistic data.

7. Types of GAN

22. **Vanilla GAN**: The standard GAN model, where both the generator and discriminator are simple neural networks.
23. **DCGAN (Deep Convolutional GAN)**: Uses convolutional layers for both the generator and discriminator, making it more suitable for image generation.
24. **Conditional GAN (CGAN)**: Allows conditional inputs to both the generator and discriminator to generate data conditioned on specific information (e.g., generating images of a certain class).
25. **CycleGAN**: Used for image-to-image translation tasks where paired data is not available.
26. **WGAN (Wasserstein GAN)**: A variant that improves the stability of training by using a different loss function based on the Wasserstein distance.

8. Applications of GANs

27. **Image Generation**: GANs are widely used for generating realistic images, such as generating faces or artwork.
28. **Image Super-Resolution**: GANs can enhance the resolution of images, which is useful in medical imaging or video enhancement.
29. **Style Transfer**: GANs are used for transferring the artistic style of one image to another, like in image-to-image translation tasks.
30. **Data Augmentation**: GANs can generate synthetic data to augment small datasets, particularly useful in training machine learning models.

9. Exemplar/Case Study: GAN for Detection of Real or Fake Images

31. GANs have been used to detect deepfake images, which are artificially generated images made by altering real images.
32. Discriminators in GANs can be trained to classify images as real or fake by distinguishing features generated by a neural network.
33. This technique is important for verifying the authenticity of digital content, especially in the age of social media and fake news.

DEEP LEARNING – Unit VI: *Reinforcement Learning*

1. Introduction to Deep Reinforcement Learning

1. **Reinforcement Learning (RL)** is a type of machine learning where agents learn to make decisions by interacting with an environment.
2. The agent receives rewards or punishments based on its actions, which helps it learn optimal behaviors or policies.
3. **Deep Reinforcement Learning (DRL)** combines deep learning with reinforcement learning to solve complex decision-making tasks in environments with large state spaces.

2. Markov Decision Process (MDP)

4. An **MDP** is a mathematical framework for modeling decision-making, consisting of states, actions, and rewards.
5. MDPs are defined by:
    - A set of states SSS
    - A set of actions AAA
    - A reward function RRR
    - A transition probability function PPP
6. MDPs are used to formalize reinforcement learning problems, where an agent's objective is to maximize the expected cumulative reward.

3. Basic Framework of Reinforcement Learning

7. The RL framework includes:
    - **Agent**: The learner or decision maker.
    - **Environment**: The world the agent interacts with.
    - **Action**: The choices the agent can make.
    - **Reward**: The feedback signal the agent receives after taking an action.
8. The agent’s goal is to find a policy that maximizes the cumulative reward over time.

4. Challenges of Reinforcement Learning

9. **Exploration vs Exploitation**: The agent must balance exploring new actions and exploiting known actions to maximize reward.
10. **Partial Observability**: The agent may not have access to the full state of the environment.
11. **Delayed Rewards**: Rewards may not be immediately received after actions, making learning more complex.
12. **Large State and Action Spaces**: The agent may need to handle environments with large or continuous state and action spaces.

5. Dynamic Programming Algorithms for RL

13. **Value Iteration** and **Policy Iteration** are key dynamic programming algorithms used to solve MDPs.
14. These methods iteratively update the value function or policy until convergence, helping the agent make optimal decisions.

6. Q-Learning and Deep Q-Networks (DQN)

15. **Q-Learning** is a model-free reinforcement learning algorithm where the agent learns the value of state-action pairs (Q-values).
16. The agent updates its Q-values using the Bellman equation.
17. **Deep Q-Networks (DQN)** use deep neural networks to approximate the Q-values, making it possible to handle large, high-dimensional state spaces like images.
18. DQNs combine Q-learning with deep learning for more complex tasks, like playing video games.

7. Deep Q-Recurrent Networks (DQRN)

19. **DQRN** is an extension of DQN that incorporates recurrent neural networks (RNNs) to handle environments where the agent must maintain memory of past actions.
20. It helps in environments with partial observability, where the agent needs to remember previous states to make decisions.

8. Simple Reinforcement Learning for Tic-Tac-Toe

21. A basic example of reinforcement learning is using Q-learning to train an agent to play Tic-Tac-Toe.
22. The agent learns to choose optimal moves based on the state of the board and rewards given for winning or losing.

9. Exemplar/Case Studies

23. **Self-Driving Cars**: RL is used in autonomous driving systems, where agents learn optimal driving policies through trial and error while interacting with a simulated or real-world environment.
24. **Deep Learning for Chatbots**: RL can improve chatbots by learning to generate human-like responses, optimizing for user satisfaction and engagement.

High Performance Computing - Unit I: *Introduction to Parallel Computing*

1. Introduction to Parallel Computing

- **Motivating Parallelism**: The need for parallelism arises from the limitations of single-threaded processors in handling large datasets, complex computations, and time-sensitive tasks. Parallelism allows multiple processes to be executed simultaneously, speeding up computations.
- **Modern Processor**: Modern processors are designed to handle multiple tasks simultaneously by using multiple cores and threads, improving performance for parallel computing tasks.
- **Stored-program Computer Architecture**: The architecture where both program instructions and data are stored in the same memory, and the CPU fetches, decodes, and executes the program instructions one at a time or in parallel.

2. General-purpose Cache-based Microprocessor Architecture

- This architecture uses cache memory to store frequently accessed data close to the processor to reduce access latency, improving performance in parallel systems.

3. Parallel Programming Platforms

- **Implicit Parallelism**: A type of parallelism where the underlying system automatically handles the parallel execution of tasks without explicit instructions from the programmer.
- **Dichotomy of Parallel Computing Platforms**: Differentiation between two types of parallel systems:
    - Shared-memory systems: Multiple processors share a common memory space.
    - Distributed-memory systems: Each processor has its own memory, and communication occurs via a network.

4. Physical Organization of Parallel Platforms

- Parallel systems may be organized as clusters of machines, multi-core processors, or grids. The physical setup affects the communication costs and performance of parallel algorithms.

5. Communication Costs in Parallel Machines

- Communication between processors or nodes can become a bottleneck in parallel systems, as transferring data between processors may incur latency or bandwidth issues.

6. Levels of Parallelism

- **Instruction-Level Parallelism**: The ability to execute multiple instructions simultaneously within a single processor.
- **Data-Level Parallelism**: Performing the same operation on multiple pieces of data concurrently.
- **Task-Level Parallelism**: Decomposing tasks into independent subtasks that can run in parallel.

7. Models of Parallel Computing

- **SIMD (Single Instruction, Multiple Data)**: One instruction is executed on multiple data points simultaneously.
- **MIMD (Multiple Instructions, Multiple Data)**: Different processors execute different instructions on different data.

- **SIMT (Single Instruction, Multiple Threads)**: A model primarily used in GPUs where many threads execute the same instruction on different data points.
- **SPMD (Single Program, Multiple Data)**: A parallel programming model where multiple processors execute the same program but on different data.
- **Data Flow Models**: Computational models where operations are performed when the required data is available.
- **Demand-driven Computation**: A computational approach where computations are performed only when required data or resources are available.

8. Architectures of Parallel Platforms

- **N-wide Superscalar Architectures**: Multiple functional units allow the simultaneous execution of multiple instructions from a single instruction stream.
- **Multi-core Systems**: Processors with multiple cores that can independently process different threads or data in parallel.
- **Multi-threaded Architectures**: Processors capable of managing multiple threads, enabling concurrent execution of tasks.

High Performance Computing - Unit II: *Parallel Algorithm Design*

1. Global System for Mobile Communications (GSM) Architecture

- **Mobile Station (MS)**: A device used by the user to access the mobile network.
- **Base Station System (BSS)**: Manages communication between mobile stations and the network. It includes Base Transceiver Station (BTS) and Base Station Controller (BSC).
- **Switching Subsystem (SS)**: Manages call setup, routing, and switching between base stations.
- **Security**: Implements encryption and authentication mechanisms for secure communication.
- **Data Services**: GSM supports various data services like HSCSD (High-Speed Circuit-Switched Data) and GPRS (General Packet Radio Service) for mobile internet access.

2. GSM System and Protocol Architecture

- **HSCSD**: Provides faster data transfer speeds compared to regular circuit-switched services.
- **GPRS**: Provides packet-based data services, enabling mobile internet access by breaking data into small packets.
- **UTRAN (Universal Terrestrial Radio Access Network)**: Part of the 3G network that connects mobile devices to the core network.
- **UMTS (Universal Mobile Telecommunications System) Core Network**: Provides all the functionalities required for mobile communication, such as call routing, user authentication, and data transmission.

3. Improvements on Core Network

- **802.11 Architecture**: Wireless Local Area Network (WLAN) architecture that defines standards for communication between devices over wireless networks.
- **802.11a**: High-speed wireless communication standard operating in the 5 GHz band.
- **802.11b**: A popular wireless communication standard operating in the 2.4 GHz band, offering speeds of up to 11 Mbps.

4. IPoC (Internet Protocol over Core Network)

- **IPoC for 5G Networks**: A new core networking protocol for 5G networks, offering more efficient data transfer and reduced latency.

Case Study Examples

- **Multi-core Systems**: Explore how multi-core processors work and the impact on parallel computing performance.
- **IPoC: A New Core Networking Protocol for 5G**: How the new core protocol facilitates improved communication, low latency, and more efficient handling of diverse data services in 5G networks.

High Performance Computing - Unit III: *Parallel Communication*

1. Basic Communication

- **One-to-All Broadcast**: A communication operation where a message from one processor is sent to all other processors in a system.
- **All-to-One Reduction**: Each processor sends data to one processor (usually the root processor), which then performs a reduction operation (like summing the data).
- **All-to-All Broadcast and Reduction**: Data is broadcast from each processor to all others, or a reduction operation is performed across all processors.
- **All-Reduce**: Combines the reduction operation with a broadcast, where the reduced value is sent to all processors.
- **Prefix-Sum Operations**: Also known as "scan," a prefix-sum computes partial sums of a sequence and can be used for parallel prefix operations.

2. Collective Communication using MPI (Message Passing Interface)

- **Scatter**: Distributes data from one processor (root) to all other processors in a system.
- **Gather**: Collects data from all processors to a single processor (root).
- **Broadcast**: A single processor sends data to all others in the system.
- **Blocking MPI**: The communication call does not return control to the program until the communication operation is complete.
- **Non-Blocking MPI**: The communication call returns control to the program immediately, allowing further computation while communication is in progress.

3. Personalized Communication

- **All-to-All Personalized Communication**: Each processor sends data to every other processor, and the data may differ between processors.
- **Circular Shift**: A type of communication where data is shifted around in a circular fashion across processors (e.g., in a circular buffer).

4. Improving Speed of Communication Operations

- Optimizing communication operations involves reducing latency and improving bandwidth efficiency. Techniques such as overlapping communication and computation, reducing synchronization overhead, and optimizing the number of messages can enhance performance.

Exemplar/Case Study: Monte-Carlo Pi Computing using MPI

- The Monte Carlo method can be parallelized using MPI to estimate the value of Pi by running multiple simulations across different processors. Each processor computes a portion of the simulations, and the results are combined to calculate the final approximation of Pi. This case study demonstrates the use of parallel communication for performance improvement in scientific computing.

High Performance Computing - Unit IV: *Analytical Modeling of Parallel Programs*

1. Sources of Overhead in Parallel Programs

- Overhead can arise from various factors such as communication costs, synchronization delays, load imbalance, and memory access contention. Identifying these overheads helps optimize parallel performance.

2. Performance Measures and Analysis

- **Amdahl's Law**: Describes the potential speedup of a parallel system, stating that the speedup is limited by the serial portion of the computation. It's used to assess the theoretical maximum improvement in performance for a parallel program.
- **Gustafson's Law**: A more optimistic model that accounts for the problem size growing with the number of processors. It states that as more processors are added, the parallel portion of the problem grows, leading to higher speedup.
- **Speedup Factor**: The ratio of the execution time of the serial version of the program to the execution time of the parallel version. It measures how much faster the parallel program is compared to the serial program.
- **Efficiency**: Efficiency of parallel computation is the ratio of the speedup to the number of processors. It reflects how well the processors are utilized.
- **Cost and Utilization**: Cost refers to the total time taken by a program, including communication and computation. Utilization indicates how effectively the processors are being used.
- **Execution Rate and Redundancy**: Execution rate refers to the amount of work done per unit time, while redundancy involves repeating tasks due to inefficiency or overhead.
- **Granularity of Computation**: The size of individual tasks in a parallel program. Fine-grained parallelism involves breaking the task into many small sub-tasks, while coarse-grained parallelism involves fewer, larger tasks.

3. Scalability of Parallel Systems

- Scalability refers to the ability of a parallel system to maintain or improve performance as the number of processors increases. A system is said to scale well if it continues to show significant speedup with additional processors.

4. Performance Metrics and Analysis

- **Minimum Execution Time and Minimum Cost**: The goal is to find the balance between execution time and cost to achieve optimal performance.
- **Optimal Execution Time**: The best possible time to complete a task on a parallel system, achieved by perfect load balancing and communication optimization.

- **Asymptotic Analysis of Parallel Programs**: Analyzing the time complexity of parallel programs as the input size grows, considering both the computation and communication costs.

5. Matrix Computation

- **Matrix-Vector Multiplication**: A key operation in many numerical algorithms. In parallel computing, matrix-vector multiplication can be distributed across processors, improving efficiency.
- **Matrix-Matrix Multiplication**: A more complex operation that can also be parallelized by distributing the matrix blocks across multiple processors.

Exemplar/Case Study: The DAG Model of Parallel Computation

- Directed Acyclic Graph (DAG) models can be used to represent the dependencies between tasks in a parallel program. Tasks are represented as nodes, and edges represent the dependencies. The DAG model helps in determining the execution order and optimizing parallel execution by minimizing idle time and dependencies.

High Performance Computing - Unit V: *CUDA Architecture*

1. Introduction to GPU

- **GPU Architecture Overview**: GPUs are specialized hardware designed for parallel computation. They are optimized for tasks that can be parallelized, such as graphics rendering and scientific computing.
- **Parallel Processing in GPUs**: GPUs consist of many smaller processing units called cores, which can execute tasks in parallel, making them much faster than traditional CPUs for parallel workloads.

2. Introduction to CUDA C

- **CUDA Programming Model**: CUDA (Compute Unified Device Architecture) is a parallel computing platform and API model developed by NVIDIA to harness the power of GPUs for general-purpose computing tasks. CUDA C allows developers to write software that runs on GPUs, providing an extension to the standard C programming language.
- **CUDA C Syntax**: The syntax is similar to C, but with extensions to support parallelism and GPU-specific functions.

3. Writing and Launching a CUDA Kernel

- **CUDA Kernel**: A function written in CUDA C that runs on the GPU. The function is executed by multiple threads in parallel, and each thread processes a part of the data.
- **Kernel Launch**: CUDA kernels are launched by specifying the number of blocks and threads that will execute the kernel, allowing for fine-grained control over parallel execution.

4. Handling Errors in CUDA

- **Error Handling**: CUDA provides functions for checking errors in kernel execution, memory allocation, and other operations. It is important to check for errors to ensure that the program runs correctly and efficiently.

5. CUDA Memory Model

- **Memory Hierarchy**: CUDA provides several types of memory, including global memory, shared memory, and local memory. Understanding the memory hierarchy is critical for optimizing performance.
    - **Global Memory**: Accessible by all threads, but slower to access.
    - **Shared Memory**: Shared by threads within the same block and faster than global memory.
    - **Local Memory**: Private to each thread and stored in global memory.
- **Memory Management**: Developers need to allocate and free memory explicitly to ensure efficient memory usage.

6. Communication and Synchronization in CUDA

- **Synchronization**: CUDA provides mechanisms for synchronizing threads within a block (e.g., __syncthreads()) and across blocks (e.g., using global memory or atomic operations).
- **Communication**: Threads in a block can communicate via shared memory. Communication between blocks usually happens through global memory.

7. Parallel Programming in CUDA-C

- **Data Parallelism**: CUDA allows for fine-grained parallelism, where each thread performs the same task on different pieces of data.
- **Optimizing Performance**: To maximize performance, the programmer must optimize the use of memory, reduce memory latency, and ensure that the computation is well-distributed across the GPU.

Exemplar/Case Study: GPU Applications using SYCL and CUDA on NVIDIA

- **SYCL**: A high-level programming model that provides a single-source programming interface for heterogeneous computing, allowing code to run on various devices (CPU, GPU, etc.). SYCL can be used in combination with CUDA to harness GPU capabilities for parallel computation.
- **Case Study**: This case study explores GPU applications such as image processing, deep learning, and scientific simulations using SYCL and CUDA to accelerate computational tasks and improve performance.

High Performance Computing - Unit VI: *High Performance Computing Applications*

1. Scope of Parallel Computing

- **Applications of Parallel Computing**: Parallel computing is used in various domains such as simulations, big data processing, scientific computing, AI/ML, and real-time systems. It allows for solving problems that are too large or complex for a single processor.

2. Parallel Search Algorithms

- **Depth First Search (DFS)**: A search algorithm for traversing or searching tree or graph data structures. It can be parallelized by exploring multiple branches concurrently.

- **Breadth First Search (BFS)**: Another search algorithm for traversing graphs. BFS can also be parallelized, with each processor handling a different level or subset of nodes.

3. Parallel Sorting Algorithms

- **Bubble Sort**: Although not efficient for large datasets, bubble sort can be parallelized by dividing the data into chunks and sorting each chunk simultaneously.
- **Merge Sort**: A more efficient sorting algorithm that is naturally parallelizable. Merge sort can be divided into sub-problems that are solved concurrently, reducing the total sorting time.

4. Distributed Computing

- **Document Classification**: Using parallel and distributed computing techniques to classify documents based on their content. Large-scale document classification tasks can be parallelized for better efficiency.
- **Frameworks for Distributed Computing**: Modern frameworks like Kubernetes (Kubernetes Orchestration) and cloud computing platforms help in distributing computation across large clusters of machines for scalable and efficient performance.

5. GPU Applications

- **GPU for AI/ML**: GPUs significantly accelerate AI and machine learning algorithms by allowing parallel processing of large datasets. Tasks like training neural networks and running deep learning models are highly optimized on GPUs.

6. Parallel Computing for AI/ML

- **AI and Machine Learning with HPC**: High-performance computing provides the computational power necessary for large-scale AI applications. Parallel computing is used for tasks such as deep learning, reinforcement learning, and optimization, which require substantial computational resources.

Exemplar/Case Study: Disaster Detection and Management/ Smart Mobility/ Urban Planning

- **Disaster Detection and Management**: Parallel computing is used to process large amounts of data from sensors, satellites, and IoT devices to predict and manage disasters. Simulations for emergency response can be executed in parallel to provide real-time decision-making capabilities.
- **Smart Mobility**: Parallel computing plays a significant role in optimizing traffic flow, vehicle routing, and autonomous vehicle operations, ensuring real-time decision-making in urban mobility systems.
- **Urban Planning**: HPC allows for large-scale simulations of urban environments, helping planners optimize resource allocation, traffic management, and infrastructure development.

PRACTICAL – I

1. Data Loading and Inspection

```
python
CopyEdit
df = pd.read_csv('Boston_house_Pricing_data.csv')
```

- `df`: A variable (short for DataFrame) used to store the dataset.
- `=`: Assignment operator.
- `pd`: Alias for the pandas library.
- `.read_csv()`: Pandas function to read a CSV (Comma-Separated Values) file.
- `'Boston_house_Pricing_data.csv'`: Name of the dataset file (assumes it's in the same folder).

```
python
CopyEdit
df.head()
```

- `.head()`: Displays the **first 5 rows** of the DataFrame to inspect the data.

---

2. Splitting Features and Target

```
python
CopyEdit
x = df.drop('MEDV', axis = 1)
```

- `x`: Stores all features (independent variables).
- `df.drop()`: Drops a specified column.
- `'MEDV'`: The target column (Median value of owner-occupied homes).
- `axis=1`: Axis=1 indicates a **column** should be dropped.

```
python
CopyEdit
y = df[['MEDV']]
```

- `y`: Stores the target column as a DataFrame (double brackets ensure 2D shape).

---

3. Feature Scaling

```
python
CopyEdit
scaler = StandardScaler()
```

- `scaler`: Variable storing the instance of `StandardScaler`.
- `StandardScaler()`: Standardizes features by removing the mean and scaling to unit variance (from `sklearn.preprocessing`).

```
python
CopyEdit
x_scaled = scaler.fit_transform(x)
y_scaled = scaler.fit_transform(y)
```

- `fit_transform()`: Fits the scaler to the data and then transforms it.

- `x_scaled`, `y_scaled`: Scaled versions of features and target.

□ 4. Dataset Info

```
python
CopyEdit
df.info()
```

- `.info()`: Displays summary including number of rows, columns, data types, and non-null values.

□ 5. Splitting Data into Train/Test

```
python
CopyEdit
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.3,
random_state=42)
```

- `train_test_split`: Splits the dataset into training and testing sets.
- `test_size=0.3`: 30% of data used for testing.
- `random_state=42`: Ensures reproducibility of the split.

```
python
CopyEdit
print('Training set shape:', X_train.shape, y_train.shape)
print('Testing set shape:', X_test.shape, y_test.shape)
```

- `.shape`: Returns the dimensions of the data.

□ 6. Building the Neural Network

```
python
CopyEdit
model = Sequential()
```

- `model`: Stores the neural network.
- `Sequential()`: Linear stack of layers in Keras.

```
python
CopyEdit
model.add(Dense(64, input_dim=13, activation='relu'))
```

- `add()`: Adds a layer to the model.
- `Dense(64)`: Fully connected layer with 64 neurons.
- `input_dim=13`: Number of input features (13 in Boston dataset).
- `activation='relu'`: ReLU activation function.

```
python
CopyEdit
model.add(Dropout(0.2))
```

- `Dropout(0.2)`: Randomly drops 20% of the neurons during training to reduce overfitting.

```
python
CopyEdit
model.add(Dense(32, activation='relu'))
```

- Adds another dense layer with 32 neurons and ReLU activation.

```
python
CopyEdit
model.add(Dense(1))
```

- Output layer with **1 neuron** (for regression output).

```
python
CopyEdit
print(model.summary())
```

- `.summary()`: Displays architecture and number of trainable parameters.

□ 7. Compile and Train the Model

```
python
CopyEdit
model.compile(loss=MeanSquaredError(), optimizer='adam',
metrics=['mean_absolute_error'])
```

- `compile()`: Configures the model for training.
- `loss=MeanSquaredError()`: MSE is used as loss function for regression.
- `optimizer='adam'`: Adam optimizer adjusts learning rate automatically.
- `metrics=['mean_absolute_error']`: MAE is used to track performance.

```
python
CopyEdit
early_stopping = EarlyStopping(monitor='val_loss', patience=5)
```

- `EarlyStopping`: Stops training when the validation loss doesn't improve for 5 epochs.

```
python
CopyEdit
history = model.fit(X_train, y_train, validation_split=0.2, epochs=100, batch_size=32,
                    callbacks=[early_stopping])
```

- `fit()`: Starts the training process.
- `validation_split=0.2`: 20% of training data used for validation.
- `epochs=100`: Maximum number of epochs to train.
- `batch_size=32`: Processes 32 samples at a time.
- `callbacks=[early_stopping]`: Applies early stopping during training.

□ 8. Plotting the Loss

```
python
CopyEdit
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
```

- Plots training and validation loss from the training history.

```
python
CopyEdit
plt.title('Model Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend(['Training', 'Validation'])
plt.show()
```

- Adds labels and legend to the plot for clarity.

□ 9. Evaluate Model on Test Set

```
python
CopyEdit
mae = model.evaluate(X_test, y_test)
```

- `evaluate()`: Evaluates model on the test set.
- Returns loss and metrics values.

```
python
CopyEdit
print('Mean Absolute Error:', mae)
```

- Prints the result (MAE) of the evaluation.

PRACTICAL – II

□ 1. Load the IMDB Dataset

```
python
CopyEdit
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=10000)
```

- `imdb.load_data(...)`: Loads the pre-tokenized IMDB dataset.
- `num_words=10000`: Keeps only the **top 10,000 most frequent words**.
- `(x_train, y_train), (x_test, y_test)`: Splits into training and testing sets.
  - `x_train`, `x_test`: List of integer-encoded reviews.
  - `y_train`, `y_test`: Labels (0 = negative, 1 = positive).

□ 2. Replace Out-of-Vocab Words

```
python
CopyEdit
x_train = [[word_index if word_index < 10000 else 0 for word_index in sequence] for
sequence in x_train]
x_test = [[word_index if word_index < 10000 else 0 for word_index in sequence] for
sequence in x_test]
```

- This list comprehension replaces any word with an index **≥ 10000** (out-of-vocabulary) with `0`.
- `sequence`: A single review (list of integers).
- `word_index`: Integer index of each word.

□ 3. Pad Sequences to Equal Length

```
python
CopyEdit
x_train = pad_sequences(x_train, maxlen=100)
x_test = pad_sequences(x_test, maxlen=100)
```

- `pad_sequences`: Pads each review to ensure the same length.
- `maxlen=100`: Truncate/pad each review to exactly **100 words**.
- Output shape becomes `(num_samples, 100)`.

□ 4. Build the Neural Network

```
python
CopyEdit
model = Sequential()
```

- `Sequential()`: Creates a linear stack of layers.

```
python
CopyEdit
model.add(Embedding(input_dim=10000, output_dim=128, input_length=100))
```

- `Embedding`: Maps word indices to dense vectors.
  - `input_dim=10000`: Vocabulary size (0 to 9999).
  - `output_dim=128`: Size of each word vector.
  - `input_length=100`: Input length of each sequence (padded to 100).

```
python
CopyEdit
model.add(LSTM(128, kernel_regularizer=l2(0.001)))
```

- `LSTM(128)`: Adds an LSTM layer with 128 memory units (neurons).
- `kernel_regularizer=l2(0.001)`: Adds L2 regularization to reduce overfitting.

```
python
CopyEdit
model.add(Dropout(0.4))
```

- `Dropout(0.4)`: Randomly disables 40% of the neurons to reduce overfitting.

```
python
CopyEdit
model.add(Dense(1, activation='sigmoid'))
```

- `Dense(1)`: Fully connected output layer with 1 neuron.
- `activation='sigmoid'`: Used for **binary classification** (0 or 1).

□ 5. Compile the Model

```
python
```

```
CopyEdit
model.compile(loss='binary_crossentropy', optimizer=Adam(learning_rate=0.0003),
metrics=['accuracy'])
```

- `loss='binary_crossentropy'`: Suitable for binary classification.
- `optimizer=Adam(...)`: Adaptive optimizer with learning rate 0.0003.
- `metrics=['accuracy']`: Tracks accuracy during training.

□ 6. Add Early Stopping

```
python
CopyEdit
early_stop = EarlyStopping(monitor='val_loss', patience=2, restore_best_weights=True)
```

- `EarlyStopping`: Stops training early if validation loss doesn't improve.
  - `monitor='val_loss'`: Monitors the validation loss.
  - `patience=2`: Waits 2 epochs before stopping.
  - `restore_best_weights=True`: Restores the weights of the best-performing epoch.

□ 7. Train the Model

```
python
CopyEdit
history = model.fit(x_train, y_train, epochs=10, batch_size=64, validation_split=0.2,
callbacks=[early_stop])
```

- `fit(...)`: Trains the model.
- `epochs=10`: Maximum of 10 epochs.
- `batch_size=64`: Number of samples per batch.
- `validation_split=0.2`: Uses 20% of training data for validation.
- `callbacks=[early_stop]`: Applies early stopping during training.

□ 8. Evaluate the Model

```
python
CopyEdit
loss, acc = model.evaluate(x_test, y_test, batch_size=64)
```

- `evaluate(...)`: Calculates loss and accuracy on the test set.

```
python
CopyEdit
print(f'Test accuracy: {acc:.4f}, Test loss: {loss:.4f}')
```

- Displays test accuracy and loss rounded to 4 decimal places using **f-string formatting**.

PRACTICAL – III

? 1. Load Dataset

```
python
CopyEdit
fashion_mnist = keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
```

- Loads the **Fashion MNIST** dataset of 70,000 grayscale images (28×28) in 10 categories.
- `train_images`: 60,000 training images.
- `test_images`: 10,000 test images.
- `train_labels`: integer labels (0–9) for each image.

```
python
CopyEdit
print(train_labels.shape)
```

- Confirms label shape: should be `(60000,)`.

□ 2. Label Names

```
python
CopyEdit
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

- Maps integer labels (0–9) to human-readable names.

□ 3. Normalize the Images

```
python
CopyEdit
train_images = train_images / 255.0
test_images = test_images / 255.0
```

- Normalizes pixel values (0–255) to [0, 1] for faster training.

□ 4. Visualize Sample Images

```
python
CopyEdit
plt.figure(figsize = (10, 10))
for i in range(25):
  plt.subplot(5, 5, i + 1)
  plt.xticks([]); plt.yticks([]); plt.grid(False)
  plt.imshow(train_images[i], cmap = plt.cm.binary)
  plt.xlabel(class_names[train_labels[i]])
plt.show
```

- Displays the first 25 training images with labels.

- `plt.imshow(...)`: Shows grayscale image with binary colormap.
- `plt.xlabel(...)`: Displays the name of the class.

 Note: `plt.show` should be **`plt.show()`** (function call).

□ 5. Define the CNN Model

```
python
CopyEdit
model = keras.Sequential([
    keras.layers.Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28,
1)),
    keras.layers.MaxPooling2D(pool_size=(2, 2)),
    keras.layers.Flatten(),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dense(10, activation='softmax')
])
```

- **Conv2D**: 32 filters of size 3×3, ReLU activation.
- **MaxPooling2D**: Reduces spatial size (downsampling).
- **Flatten**: Flattens 2D feature maps to 1D.
- **Dense(128)**: Fully connected layer with 128 neurons.
- **Dense(10)**: Output layer with 10 neurons (softmax for multiclass classification).

□ 6. Model Summary

```
python
CopyEdit
model.summary()
```

- Displays total layers, parameters, and shapes.

□ 7. Compile the Model

```
python
CopyEdit
model.compile(
    optimizer='adam',
    loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=['accuracy']
)
```

- **Optimizer**: Adam (adaptive gradient).
- **Loss**: Sparse categorical crossentropy (for integer labels).
- `from_logits=True`: Misused here because the model already ends in `softmax`. Should be `from_logits=False` or remove `Softmax()` later.

□ 8. Train the Model

```
python
CopyEdit
model.fit(train_images, train_labels, epochs = 5, verbose=1)
```

- Trains the model for 5 epochs.
- `verbose=1`: Shows progress bar during training.

□ 9. Evaluate Model

```
python
CopyEdit
test_loss, test_acc = model.evaluate(test_images, test_labels, verbose = 2)
print("\n Test accuracy = ", test_acc*100)
print("\n Test loss = ", test_loss)
```

- Evaluates model performance on test data.

□ 10. Softmax Output & Predictions

```
python
CopyEdit
probability_model = keras.Sequential([model, keras.layers.Softmax()])
predictions = probability_model.predict(test_images)
```

- Incorrect usage: `model` already ends in `softmax`, so wrapping it again is **redundant**.
- Either:
  - Keep `from_logits=True` and wrap with `Softmax()`, or
  - Remove `Softmax()` wrapper and set `from_logits=False`.

Then this:

```
python
CopyEdit
predictions = model.predict(test_images)
```

- Again calls `predict`, so the earlier softmax-wrapped model is not used here. Only the last line is effective.

□ 11. Plot Prediction Results

```
python
CopyEdit
def plot_image(i, predictions_array, true_label, img):
    true_label, img = true_label[i], img[i]
    plt.grid(False)
    plt.xticks([]); plt.yticks([])
    plt.imshow(img, cmap=plt.cm.binary)

    predicted_label = np.argmax(predictions_array)

    color = 'blue' if predicted_label == true_label else 'red'
```

```
    plt.xlabel(f"{class_names[predicted_label]} {100*np.max(predictions_array):2.0f}%
({class_names[true_label]})", color=color)
```

- Plots one image with predicted vs. true label.
- `np.argmax(predictions_array)`: Gets highest probability class.

```
python
CopyEdit
rows = 5
cols = 3
total_images = rows * cols
plt.figure(figsize = (10, 10))
for i in range(total_images):
    plt.subplot(rows, cols, i + 1)
    plot_image(i, predictions[i], test_labels, test_images)
plt.tight_layout()
plt.show()
```

- Displays 15 predictions.
- Highlights incorrect predictions in red.