24EC504 – Unit IV – Part 2: TensorFlow Lite Micro, Embedded Vision and Predictive Analytics

1. TensorFlow Lite for Microcontrollers

1.1 What is TensorFlow Lite?

TensorFlow Lite (TFLite) is an open-source, lightweight deep-learning framework for on-device inference (edge computing). It deploys trained models on mobile phones, embedded Linux systems and microcontrollers without the cloud, and is a production-ready, cross-platform framework. It supports Android, iOS, embedded Linux (for example Raspberry Pi) and MCUs, and suits real-time work such as object detection and image recognition.

1.2 Architecture

- **TensorFlow model:** a trained model saved from the standard workflow (Keras, SavedModel, concrete functions).
- **TFLite Converter:** transforms the model into TFLite format and applies optimisations such as quantization.
- **TFLite model file (.tflite):** a lightweight, platform-independent, FlatBuffers-based format optimised for low latency, performance and minimal memory.

1.3 Deployment and Execution

- **Mobile/embedded application:** the .tflite file is packaged inside the app.
- **Java API:** high-level wrapper over the C++ API, mainly for Android.
- **C++ API:** the core API (Android and iOS) that loads the model and invokes the interpreter.
- **Interpreter:** runs the model with optimised operators and selective (on-demand) kernel loading, using far less memory than older TensorFlow Mobile.
- **Hardware acceleration:** uses the Android Neural Networks API (NNAPI) where supported; otherwise the CPU.
- **Custom kernels:** developers can write custom operators in the C++ API.

1.4 TensorFlow Lite Workflow

1. **Train** the model with TensorFlow on a high-performance system.
2. **Convert** to .tflite with the TFLite Converter.
3. **Optimise** (quantization and so on) to reduce size and improve speed.
4. **Deploy** the .tflite model into a mobile, embedded or IoT application.
5. **Run inference** with the TFLite Interpreter.

TFLite uses **FlatBuffers** instead of Protocol Buffers: data can be accessed directly without an unpacking/parsing step, giving faster load times and lower memory use.

1.5 TensorFlow Lite for Microcontrollers (TFLite Micro)

A variant stripped down for devices with tens or hundreds of KB of memory and no OS. It is written in portable C++ (C++11) with no dynamic memory allocation, no standard-library dependency and no OS reliance, so it ports to virtually any 32-bit MCU (ARM Cortex-M, ESP32).

Key characteristics:

- Extremely small runtime: the core interpreter fits in under 20 KB on a Cortex-M3.
- No dynamic memory allocation: everything lives in a fixed "tensor arena" buffer supplied by the application.
- No OS or file system: the model is compiled into the firmware as a C byte array.
- Same `.tflite` FlatBuffer format, so the training/conversion pipeline is unchanged.
- Optimised kernels (for example CMSIS-NN) accelerate convolution and fully-connected layers on Cortex-M.

Typical application steps:

1. Train and quantize a model on a workstation.
2. Convert to `.tflite`, then to a C source array with a tool such as `xxd`.
3. Include the C array and the TFLite Micro library in the embedded project.
4. At start-up, create an interpreter, allocate a static tensor arena and load the model.
5. Every sampling cycle: copy sensor data into the input tensor, call `Invoke()`, read the output tensor.

1.6 TensorFlow vs TensorFlow Lite / TFLite Micro

Aspect	TensorFlow	TFLite / TFLite Micro
Purpose	Training and general ML/DL development	On-device inference only
Target platform	Servers, desktops, cloud (GPU/TPU)	Mobile, embedded Linux, microcontrollers
Model size	Large, often hundreds of MB	Compressed to KB–MB
File format	SavedModel / Protocol Buffers	`.tflite` FlatBuffer
Memory management	Dynamic, OS-managed	Static / pre-allocated (TFLite Micro)
Typical use	Research and training	Real-time, low-power, offline inference

1.7 Applications

Mobile apps (on-device image, text and speech); IoT devices (inference on sensors and wearables without the cloud); edge computing (Raspberry Pi, microcontrollers); computer vision (object detection, face recognition, OCR, real-time camera apps); speech and audio (speech recognition, keyword spotting, audio classification).

2. Computer Vision using Embedded Platforms

2.1 Introduction

Embedded computer vision runs image-processing and vision algorithms (object detection, face recognition, tracking, OCR) directly on the resource-limited hardware inside a camera, sensor node or controller, rather than only on a PC or cloud. A **smart camera**, an imaging sensor plus a compute unit whose output is a processed result (for example the location and identity of a person) rather than raw video, is a typical example.

2.2 Hardware Platforms

Platform	Key points
DSP (Digital Signal Processor)	Fixed instruction set executed sequentially, optimised for matrix operations and multiply-accumulate (fixed-point, increasingly floating-point). Deep pipelines, linear program flow with few jumps, SIMD instructions; many are VLIW (several instructions in parallel). Faster, lower power, cheaper for specialised tasks
FPGA (Field Programmable Gate Array)	Logic can be modified after manufacture; the logic program is effectively the microcontroller. Modern FPGAs integrate hard/soft cores (PowerPC, ARM, DSP), multipliers, interface logic and memory blocks. Design flow: technology mapping → logic synthesis into a netlist (usually EDIF) → place-and-route to a bitstream loaded via JTAG
ASIC (Application-Specific IC)	Designed for one application with only the components needed. Drawbacks: long production cycle, high one-time (NRE) cost, costly respins (months, often hundreds of thousands of dollars). A mass-produced ASIC gives the best power, performance and per-unit cost
SoC (System on Chip)	All essential components on one chip: a general-

Platform	Key points
	purpose core (ARM, MIPS, PowerPC, x86), DSP, MCU peripherals (buses, clocks, memory controllers) and interfaces (Ethernet MAC, USB, Bluetooth, UART, display, ADC/DAC). Ideal for phones, media players and set-top boxes; many include video codecs (MPEG-2, MPEG-4, H.263)
Smart camera chips and boards	Processors/SoCs supporting higher-level vision, from CMOS image-capture chips with small processors to frame-grabber PCI boards with several CPUs; from ultra-low-power embedded vision to industrial vision

2.3 Software and Programming Tools

- **DSPs:** programmed in C, with critical routines hand-optimised in assembly; usually controlled by a small RTOS. Linux is common on SoCs; embedded OSs have footprints of a few MB.
- **FPGAs:** logic described in a hardware description language (VHDL or Verilog); EDA suites combine synthesis, hardware libraries and macros (memory, multipliers, DSP and GPP soft cores) and place-and-route.
- **JTAG (Joint Test Action Group):** standard boundary-scan interface for testing chips and boards and loading the bitstream onto an FPGA.

2.4 Core Algorithms

- **Tensor product methods:** a theoretical tool comparing how algorithms (convolution/FIR filtering, PCA, FFT, Walsh-Hadamard, Discrete Hartley/Cosine transforms, Strassen multiplication) map onto processor architectures, efficient on SIMD hardware.
- **Sorting:** Quicksort (fastest average) and Heapsort (slightly slower, better worst-case, in-place).
- **Golden search:** finds the single maximum of a function in an interval when evaluation is expensive.
- **Kalman filtering:** a recursive estimator of a dynamic system's states from noisy measurements; needs only the previous estimate and the latest measurement, so it suits real time. Particle filtering handles non-Gaussian, non-linear systems.
- **FFT:** O(n log n) algorithms for the DFT; speed up convolution, correlation and morphological operations.
- **PCA-based detection and recognition:** eigenpictures/eigenfaces for face detection and classification by reconstruction error. Using 8-bit fixed point instead of 32-bit floating

point on a TI DM642 DSP cut classification time seventeen-fold while keeping about 48 dB SNR.

2.5 What Makes an Algorithm Suitable for DSP/FPGA Hardware

1. Streaming/sequential data access (not random access).
2. Multiple, largely independent data streams.
3. High data rates.
4. Fixed or bounded data-packet sizes.
5. Computation that splits into pipeline stages.
6. Fixed-precision (integer/fixed-point) values rather than full floating point.
7. Analyzable at instruction and module level, with little communication between streams.

Digital video fits well: dense 2-D data of constant-size elements streamed line by line. Two execution strategies: **online/data-driven** (process data as it arrives: low latency and buffering, but possibly wasted computation) and **demand-driven** (compute only what is needed downstream: no waste, but extra latency).

2.6 Linear Algebra Backbone

Vision methods rely on classical linear algebra for non-image data such as feature vectors: matrix multiplication, inverse and transpose, Gaussian elimination, residual minimisation, singular value decomposition (SVD) and eigenvalue/eigenvector computation, all of which benefit from SIMD/pipeline-friendly hardware.

2.7 Conclusion

Real-time performance is the central challenge: hardware architecture, numeric representation (fixed vs floating point) and algorithm structure (streaming, pipelined, SIMD-friendly) decide whether a task fits the power, memory and latency budget. Applications span games, surveillance, automatic parking and airbag deployment.

3. Embedded Predictive Analytics

3.1 What is Embedded Analytics?

Embedded analytics (embedded BI) integrates data analysis directly into applications, websites, dashboards or devices so users need not switch to a separate tool. **Embedded predictive analytics** goes further: instead of only showing historical charts, the engine uses statistical and ML models to forecast values, detect anomalies or recommend actions, built into the product or device itself.

3.2 How It Works

1. **Data connection and flow:** the engine continuously pulls data from the system, sensors or external sources (often through APIs).
2. **Feature engineering and preprocessing:** raw values are cleaned and turned into features (hour of day, rolling averages, statistical summaries).

3. **Model inference:** a trained regression, classification or time-series forecasting model produces a prediction.
4. **Real-time visualization / action:** the prediction becomes a dashboard chart, an alert or an automated control action.
5. **Continuous learning / update:** the model is retrained or updated periodically as new data arrives.

3.3 Example: Energy Monitoring Pipeline

To forecast energy consumption, power factor or CO_2 levels: collect sensor data → check for missing values and outliers → optionally augment (for example synthetic data generation) → engineer time-based features (hour, month, running statistics) → train or continuously update a lightweight prediction model that can run at the edge.

3.4 Evolution from Traditional BI

Era	Characteristics
Early BI (1980s–2000s)	Centralised, IT-managed reporting tools; users had no direct access to timely data
Self-service analytics (2000s)	Non-technical users got direct access to dashboards without IT
Cloud and SaaS (2010s)	Cloud allowed flexible, scalable embedding of analytics into any SaaS product
AI and ML era (2015–present)	Predictive analytics and real-time processing move products from reporting the past to forecasting the future and recommending actions

3.5 Benefits

- Faster, in-context decisions without leaving the application.
- Proactive rather than reactive: issues (machine failure, demand spikes, fraud) flagged before they fully occur.
- Reduced tooling overhead: no export to a separate BI tool.
- Personalisation at scale per user, device or machine.

3.6 Requirements and Implementation Strategy

A clear strategy decides where embedded predictive analytics adds most value: choose the platform (off-the-shelf embedded-BI tool, integrated ML pipeline or custom solution), ensure high-quality relevant data, and train users to read the insight as part of their normal workflow.

Technical requirements:

- Reliable data pipelines (APIs, message queues) from sensors/systems to the analytics engine.

- A lightweight model runtime for the target (server-side for web dashboards, or an edge/TinyML runtime for devices).
- Monitoring for model drift: check predictions periodically and retrain as patterns change.
- Clear, simple visual presentation (charts, gauges, alerts).

3.7 Applications

Predictive maintenance (forecasting part failure from vibration/temperature); demand and sales forecasting; energy management (consumption and carbon emissions from smart-meter data); healthcare monitoring (patient deterioration risk from vital signs); fraud detection (anomalous transactions in real time).