CHAPTER 10 — Introduction to Artificial Neural Networks with Keras

Artificial Neural Networks (ANNs) are Machine Learning models inspired by biological neural systems, but over time they have evolved into mathematically driven architectures that differ significantly from their biological origins. ANNs form the foundation of **Deep Learning** and have proven to be highly scalable and expressive, enabling breakthroughs in image recognition, speech processing, recommendation systems, and strategic game playing. This chapter introduces the historical evolution of neural networks, explains their core building blocks, and demonstrates how to implement, train, and deploy them using **Keras**, a high-level Deep Learning API.

From Biological to Artificial Neurons

The earliest ANN model was proposed in 1943 by McCulloch and Pitts, who introduced a simplified abstraction of biological neurons capable of performing logical computations. Although early optimism suggested imminent artificial intelligence, limitations in data, computing power, and training methods led to several periods of stagnation known as "AI winters." The modern resurgence of neural networks is driven by massive datasets, powerful GPUs, improved training algorithms, and practical success across industries.

Biological neurons consist of dendrites, a cell body, and an axon that transmits electrical signals to other neurons. While individual neurons are simple, their large-scale organization enables complex computation. Studies suggest that biological neurons are often arranged in layered structures, which inspired layered ANN architectures.

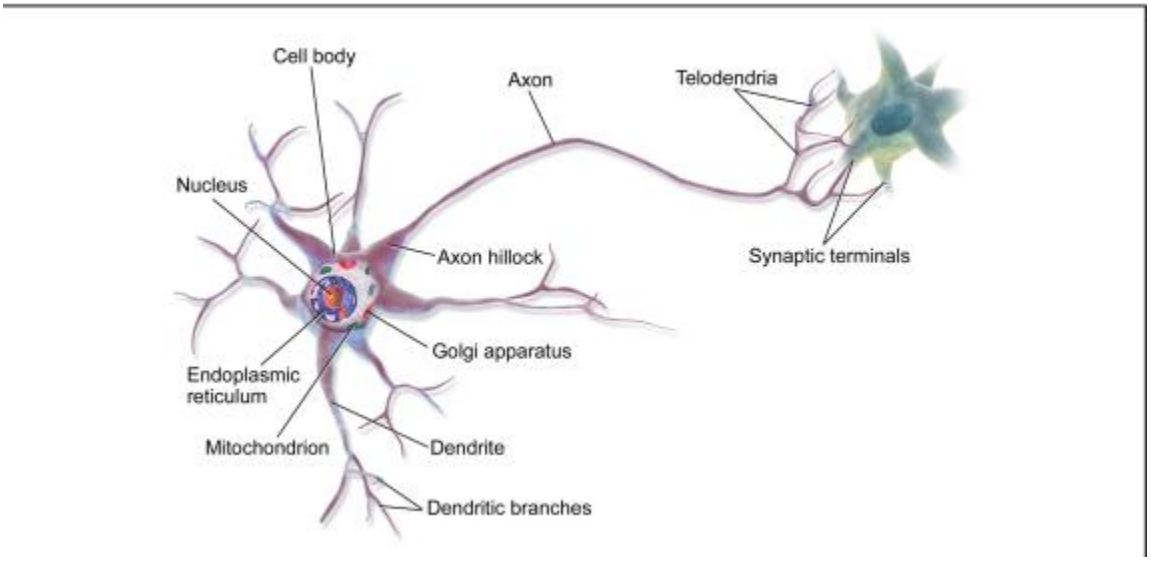

Figure 10-1. Biological neuron

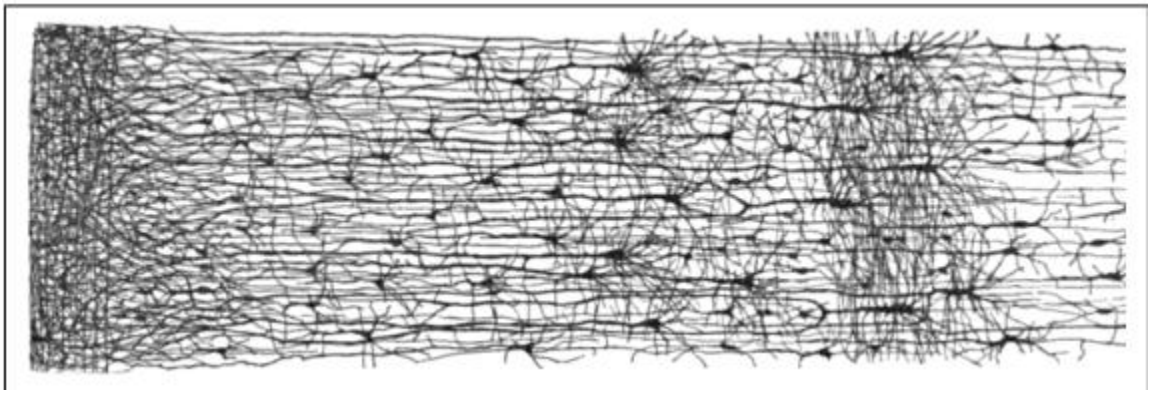

Figure 10-2. Multiple layers in a biological neural network (human cortex)

Logical Computations with Artificial Neurons

McCulloch–Pitts neurons operate with binary inputs and outputs and activate based on a threshold. Despite their simplicity, networks of such neurons can represent logical operations such as AND, OR, and NOT. By combining these operations, complex logical expressions can be computed, demonstrating that networks of simple units can produce sophisticated behavior.

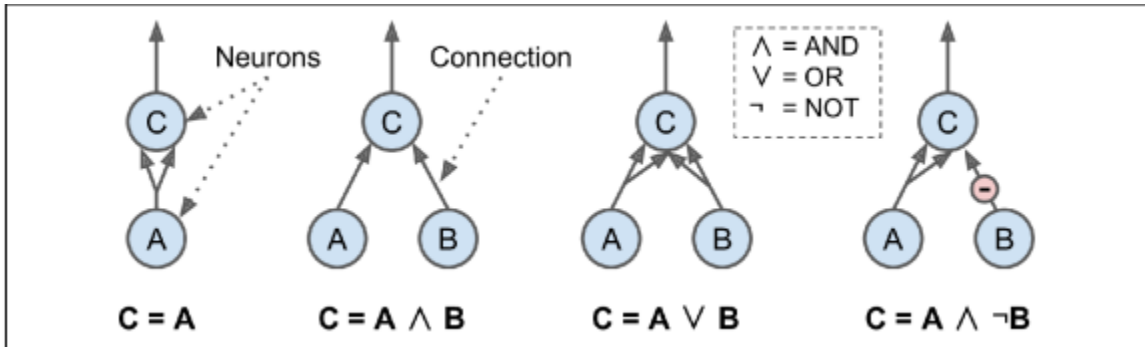

Figure 10-3. ANNs performing simple logical computations

The Perceptron

The **Perceptron**, introduced by Rosenblatt in 1957, is one of the earliest trainable ANN architectures. It is based on a **Threshold Logic Unit (TLU)** that computes a weighted sum of inputs followed by a step function. Perceptrons perform linear binary classification, similar in spirit to Logistic Regression or linear SVMs. A Perceptron consists of a single fully connected layer and can handle multi-output classification problems.

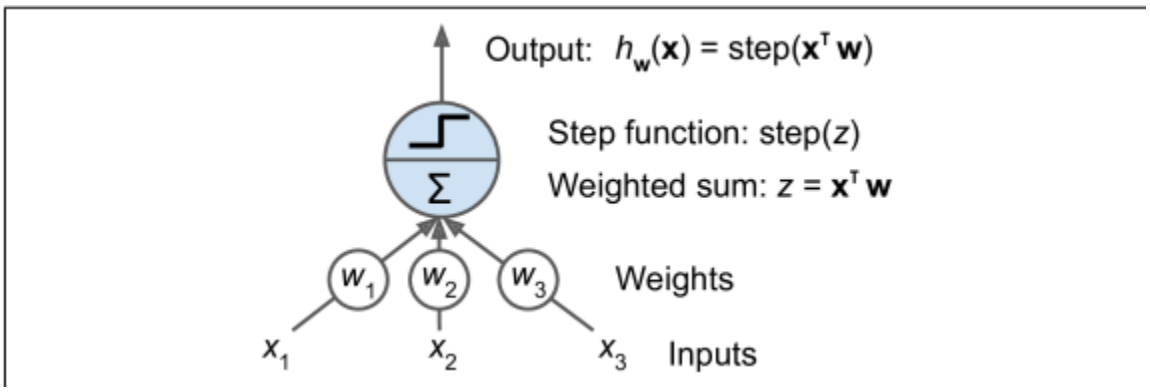

Figure 10-4. Threshold logic unit

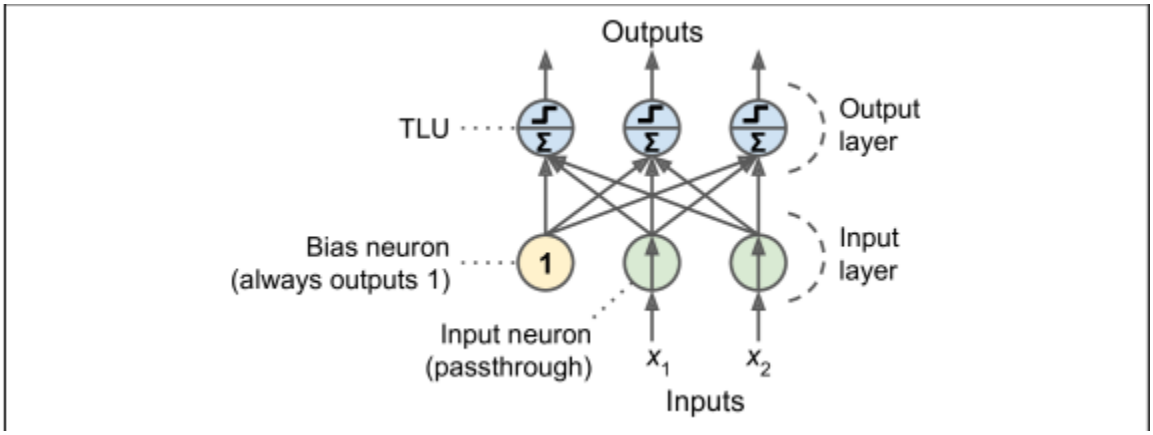

Figure 10-5. Architecture of a Perceptron

Training uses a rule inspired by **Hebbian learning**, reinforcing weights that reduce prediction error. The Perceptron convergence theorem guarantees convergence if the data is linearly separable. However, Perceptrons cannot solve nonlinearly separable problems such as XOR, a limitation that significantly slowed ANN research in the past

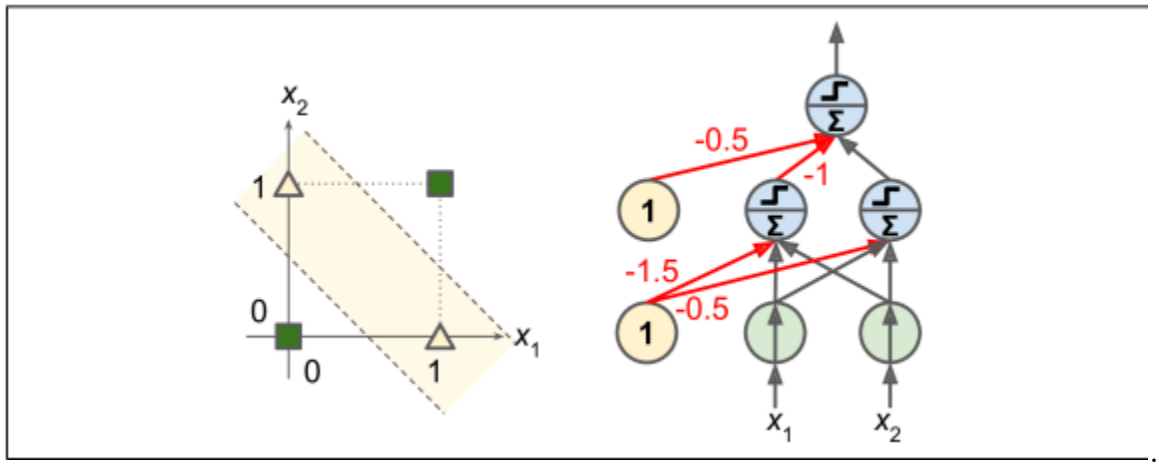

Figure 10-6. XOR classification problem and an MLP that solves it

Multilayer Perceptrons and Backpropagation

Stacking Perceptrons yields a **Multilayer Perceptron (MLP)**, which introduces one or more hidden layers between input and output layers. MLPs overcome the limitations of single-layer models and can represent complex nonlinear functions. When many hidden layers are used, the network is referred to as a **Deep Neural Network (DNN)**.

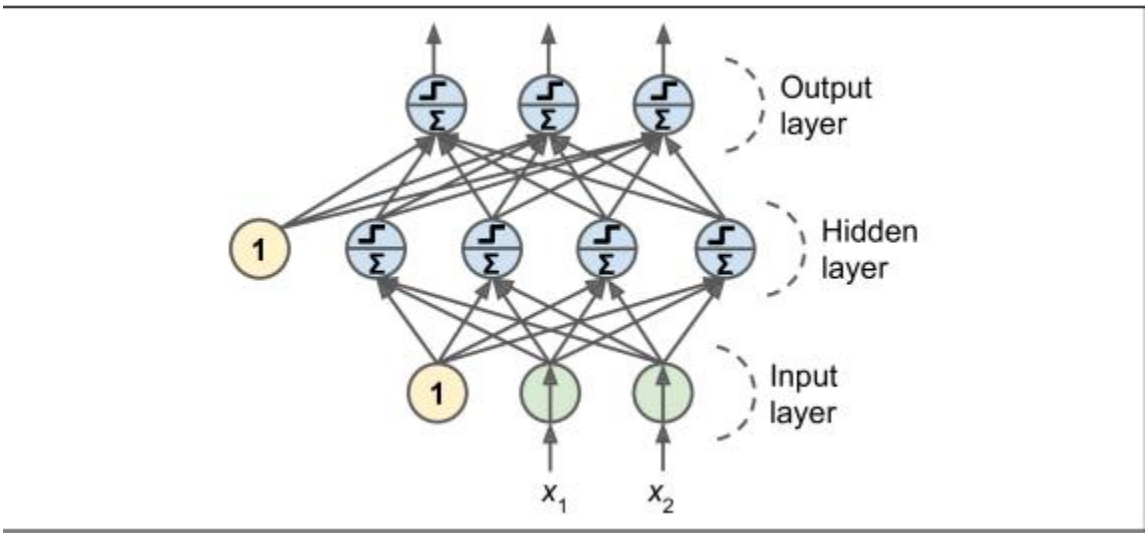

Figure 10-7. Architecture of a Multilayer Perceptron

The breakthrough that enabled effective training of MLPs was the **backpropagation algorithm**, which efficiently computes gradients using reverse-mode automatic differentiation. Training proceeds via repeated forward passes, error computation, backward gradient propagation, and Gradient Descent updates. Random weight initialization is essential to break symmetry and ensure diverse neuron behavior.

Activation functions introduce nonlinearity, which is crucial because stacked linear transformations collapse into a single linear function. Common activations include the logistic (sigmoid), hyperbolic tangent, and **ReLU**, the latter becoming the default due to its computational efficiency and favorable optimization properties.

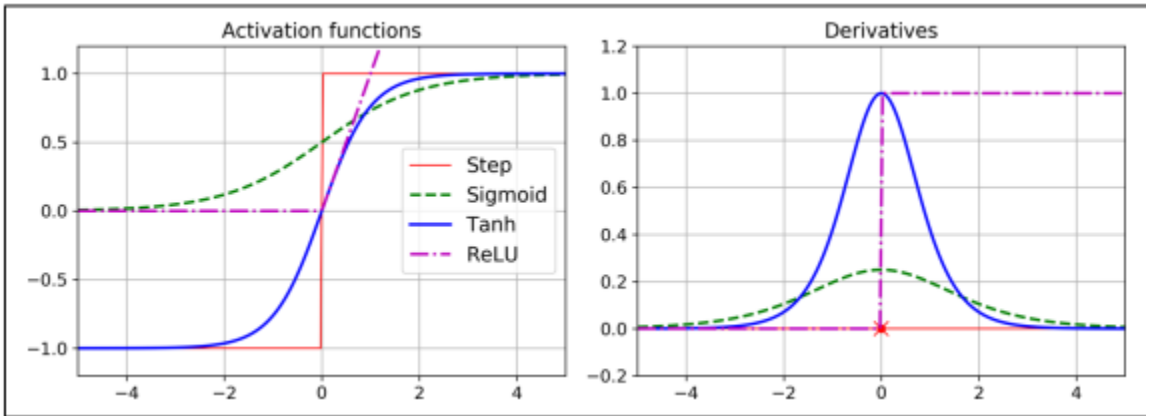

Figure 10-8. Activation functions and their derivatives

Regression and Classification with MLPs

MLPs can be adapted to regression tasks by using linear output neurons and losses such as Mean Squared Error or Huber loss. For classification, output layer design depends on the task: binary classification uses a single sigmoid output, multilabel classification uses multiple sigmoid outputs, and multiclass classification uses a softmax layer with cross-entropy loss.

Hyperparameter	Typical value
# input neurons	One per input feature (e.g., 28 x 28 = 784 for MNIST)
# hidden layers	Depends on the problem, but typically 1 to 5
# neurons per hidden layer	Depends on the problem, but typically 10 to 100
# output neurons	1 per prediction dimension
Hidden activation	ReLU (or SELU, see Chapter 11)
Output activation	None, or ReLU/softplus (if positive outputs) or logistic/tanh (if bounded outputs)
Loss function	MSE or MAE/Huber (if outliers)

Table 10-1. Typical regression MLP architecture

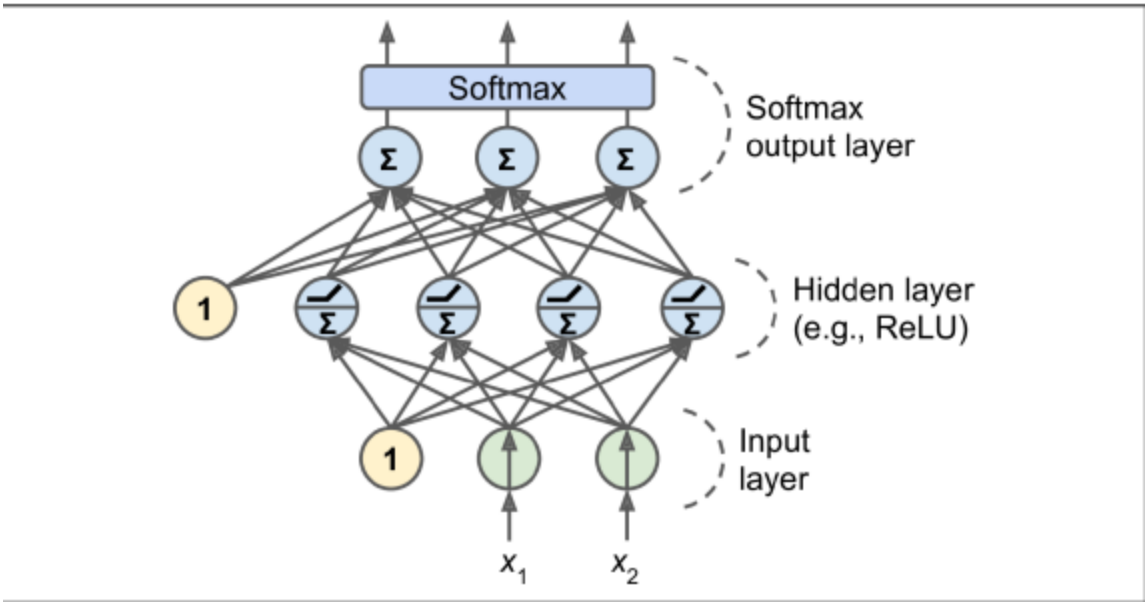

Figure 10-9. A modern MLP for classification

Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
Input and hidden layers	Same as regression	Same as regression	Same as regression
# output neurons	1	1 per label	1 per class
Output layer activation	Logistic	Logistic	Softmax
Loss function	Cross entropy	Cross entropy	Cross entropy

Table 10-2. Typical classification MLP architecture

Implementing Neural Networks with Keras

Keras is a high-level API that simplifies building, training, and deploying neural networks. In this book, tf.keras is used due to its tight integration with TensorFlow. Keras supports several modeling paradigms, making it suitable for both beginners and advanced practitioners.

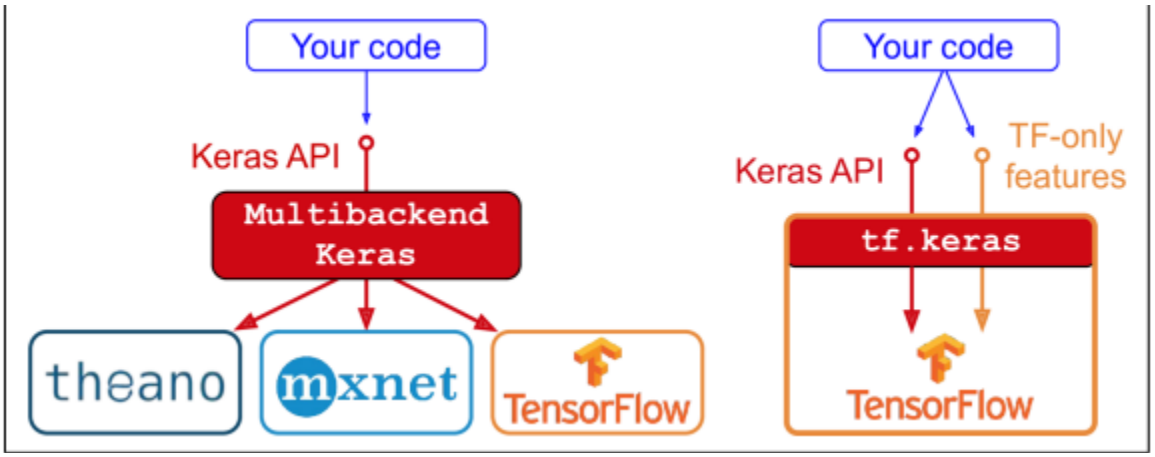

Figure 10-10. Keras implementations

Using the **Sequential API**, neural networks are defined as a simple stack of layers. This approach is ideal for standard feedforward architectures and is demonstrated using the Fashion MNIST dataset. Model construction, compilation, training, evaluation, and prediction follow a consistent and intuitive workflow.

Figure 10-11. Samples from Fashion MNIST

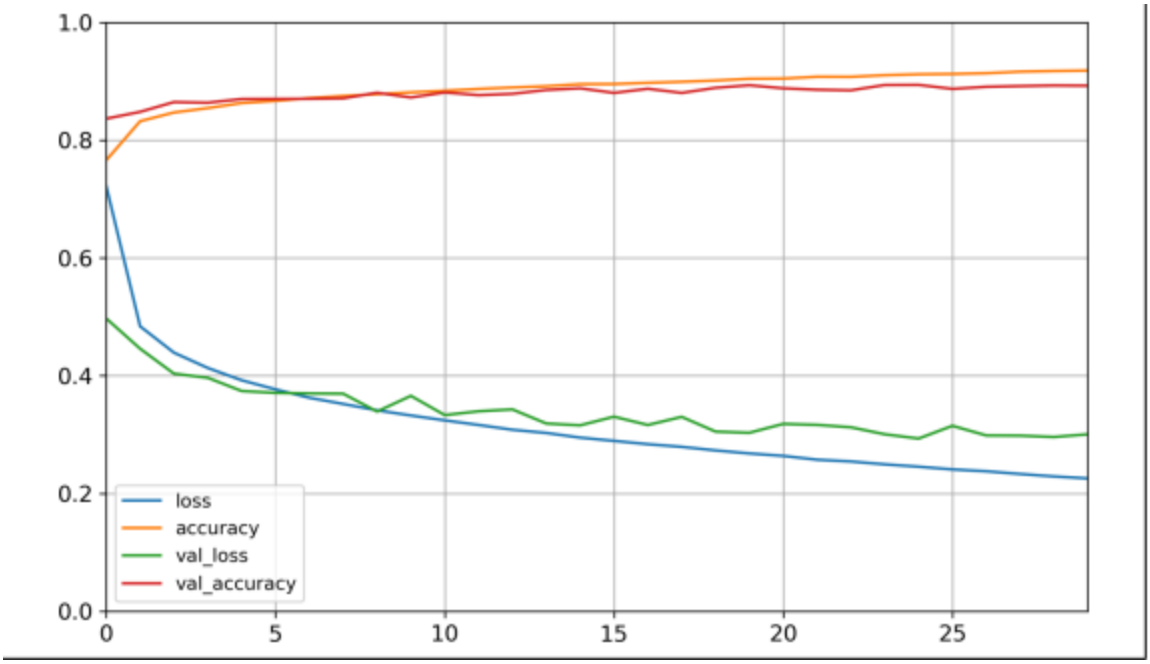

Figure 10-12. Learning curves

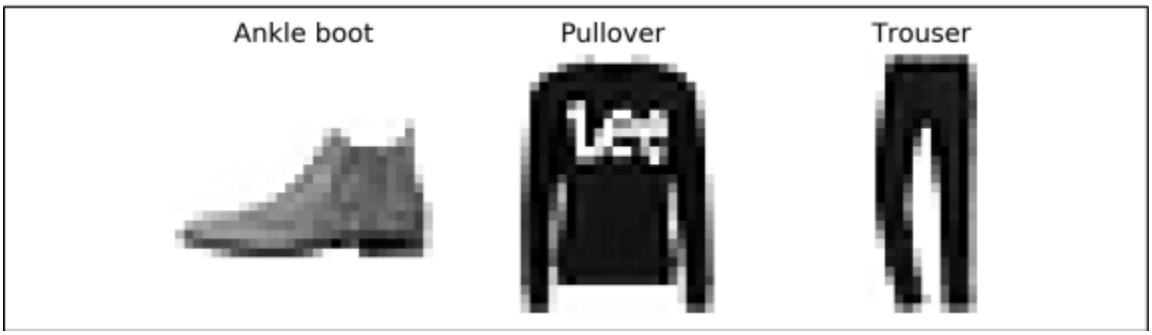

Figure 10-13. Correctly classified Fashion MNIST images

Advanced Model Architectures

For more complex topologies, Keras provides the **Functional API**, which supports multiple inputs, multiple outputs, and non-sequential connections. This enables architectures such as **Wide & Deep networks**, which combine simple linear paths with deep nonlinear paths to capture both low-level and high-level patterns.

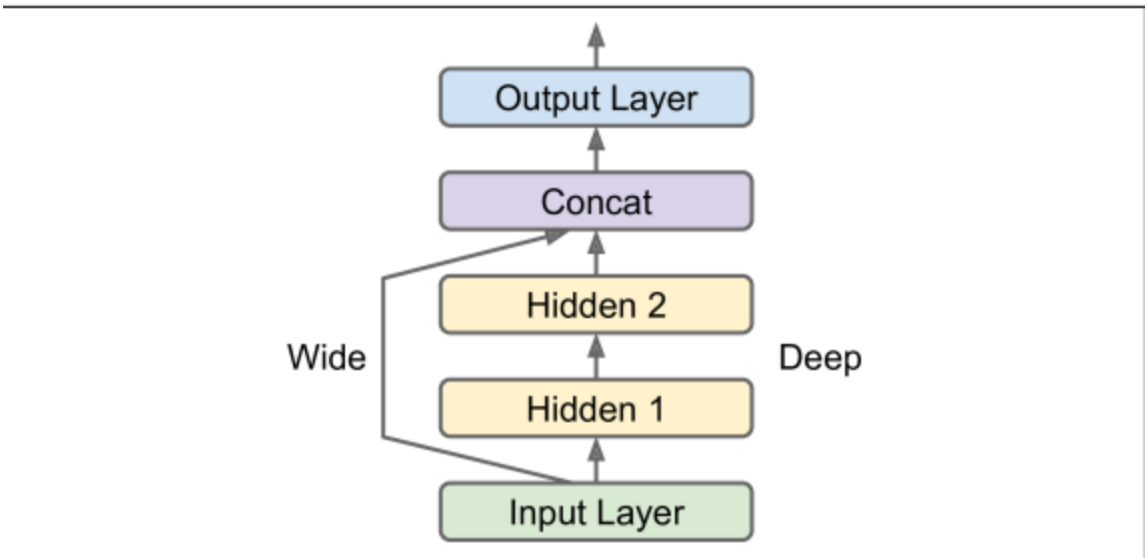

Figure 10-14. Wide & Deep neural network

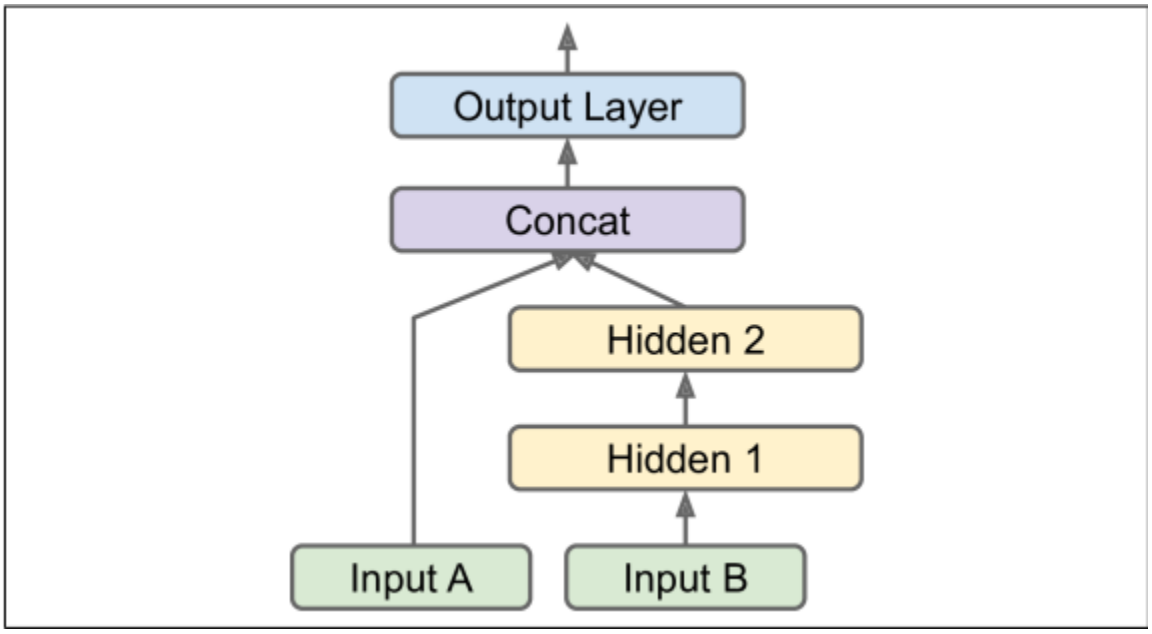

Figure 10-15. Handling multiple inputs

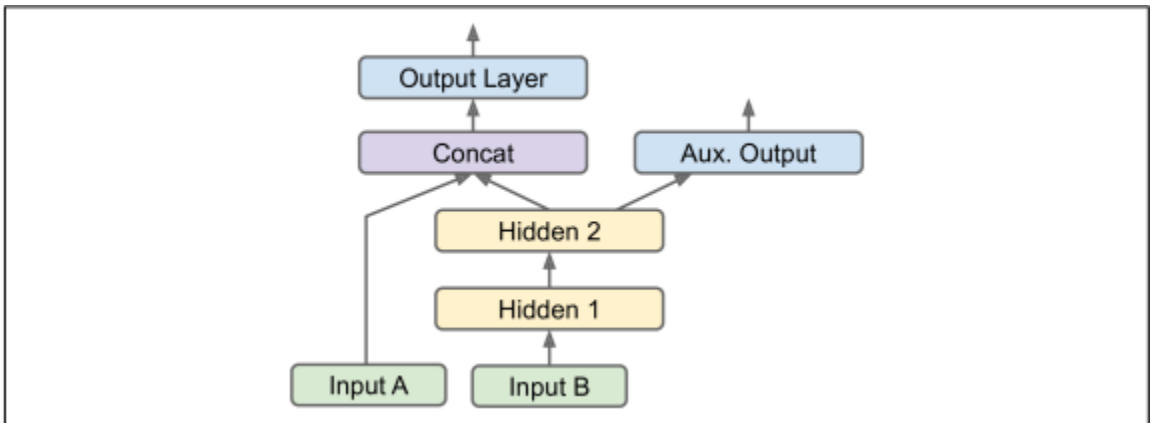

Figure 10-16. Handling multiple outputs

For maximum flexibility, Keras also offers the **Subclassing API**, allowing models to be defined imperatively using Python control flow. While powerful, this approach sacrifices model inspectability and is best reserved for research or highly dynamic architectures.

Saving Models, Callbacks, and TensorBoard

Keras models can be saved and restored easily, including architecture, weights, and optimizer state. **Callbacks** such as ModelCheckpoint and EarlyStopping enable fault tolerance and efficient training control. For visualization and monitoring, **TensorBoard** provides real-time insights into learning curves, computation graphs, embeddings, and performance bottlenecks.

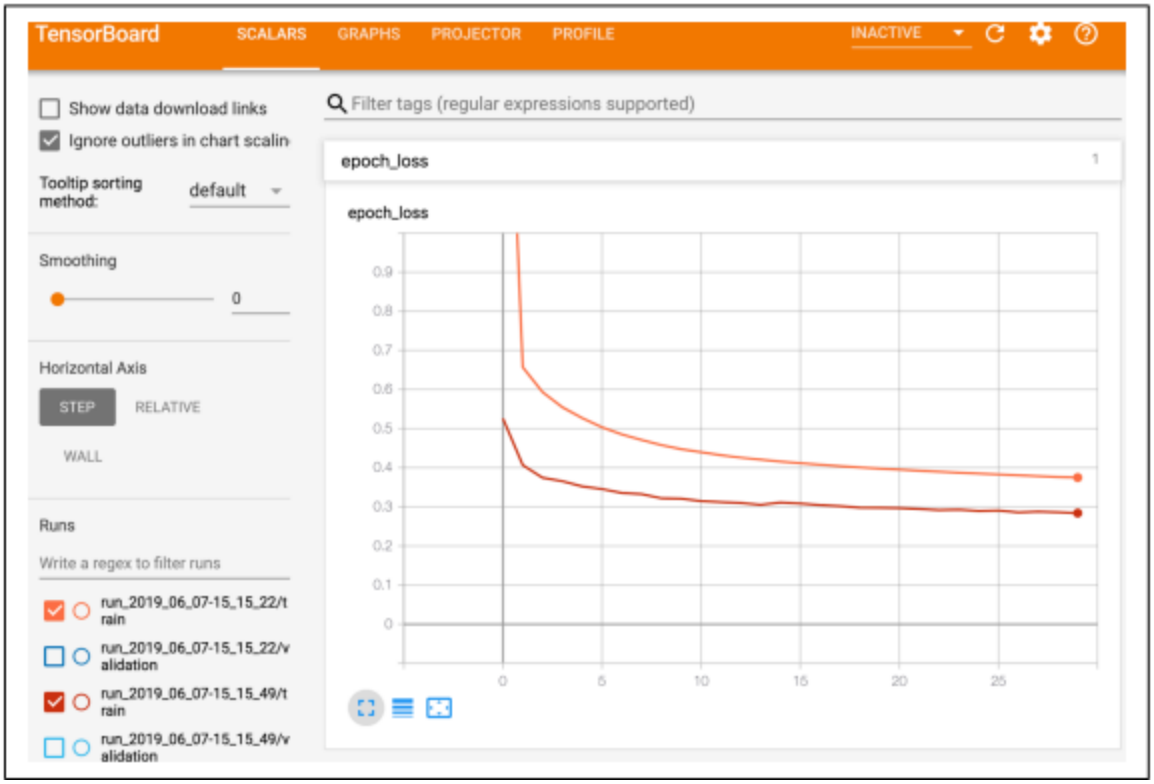

Figure 10-17. Visualizing learning curves with TensorBoard

Fine-Tuning Neural Network Hyperparameters

Neural networks expose many hyperparameters, including depth, width, learning rate, batch size, optimizer choice, and activation functions. While shallow networks can theoretically approximate complex functions, deep networks achieve far better parameter efficiency and generalization on structured data. Practical guidelines emphasize starting simple, scaling depth gradually, and relying on early stopping and regularization.

Hyperparameter optimization can be automated using randomized search, Bayesian optimization, or specialized libraries such as Keras Tuner and Hyperopt. Transfer learning further reduces training cost by reusing pretrained representations.

Summary

This chapter introduced the foundations of Artificial Neural Networks, explained how MLPs overcome the limitations of early models, and demonstrated practical implementation using Keras. You learned how to build models for regression and classification, design complex architectures, manage training with callbacks, visualize progress with TensorBoard, and

systematically tune hyperparameters. These concepts form the basis for more advanced Deep Learning techniques explored in subsequent chapters.