Machine Learning with Keras

The "Friendly Architect" of Neural Networks

A Complete, Deep-Dive Guide — From Fundamentals to Production

COVERS

Keras 3.x • TensorFlow Backend • Sequential & Functional APIs
CNNs • RNNs/LSTMs • Transfer Learning • Callbacks
Model Deployment • Best Practices • Real-world Projects

Wreetojyoti Ray

Author & Educator

Table of Contents

Chapter 1 Foundations & Philosophy

Introduction to Machine Learning & Keras

1.1 What is Machine Learning?

Machine Learning (ML) is a subdiscipline of Artificial Intelligence (AI) that enables computers to learn from data rather than being explicitly programmed with hand-crafted rules. Instead of an engineer writing "if pixel brightness > 200 and shape is round then it is a ball," an ML system is shown thousands of labelled examples of balls and non-balls, and it autonomously discovers the patterns that distinguish them.

Formally, Tom Mitchell's celebrated definition states: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E." This elegant formulation encapsulates the three pillars of every ML project: data (experience), a goal (task), and a way to measure success (performance).

The Three Paradigms of Machine Learning

Supervised Learning: The training data contains labelled input-output pairs. The model learns a mapping function f(x) → y. Examples: image classification (input=image, label=cat/dog), regression (input=house features, label=price), natural language processing (input=sentence, label=sentiment).

Unsupervised Learning: The training data has no labels. The model discovers hidden structure such as clusters, latent dimensions, or generative factors. Examples: k-means clustering, Principal Component Analysis (PCA), autoencoders, Generative Adversarial Networks (GANs).

Reinforcement Learning: An agent interacts with an environment by taking actions and receiving scalar rewards. It learns a policy (mapping from states to actions) that maximises cumulative reward over time. Examples: game-playing AI (AlphaGo, Atari agents), robot locomotion, recommendation systems.

1.2 The Machine Learning Landscape

Deep Learning is a subset of Machine Learning that uses neural networks with many layers (hence "deep") to learn hierarchical representations of data. A shallow neural network with a single hidden layer can theoretically approximate any function (Universal Approximation Theorem), but in practice, depth dramatically reduces the number of parameters needed and enables learning of increasingly abstract features.

The deep learning renaissance was sparked by three converging forces: (1) the availability of massive labelled datasets (ImageNet with 1.4 million images), (2) the repurposing of Graphics Processing Units (GPUs) for parallel tensor arithmetic, and (3) algorithmic improvements such as ReLU activations, dropout regularisation, batch normalisation, and the Adam optimiser. Since 2012 (when AlexNet won ImageNet by a large margin), deep learning has achieved superhuman

performance in domains ranging from image recognition to protein folding.

1.3 Why Keras? The Philosophy of the Friendly Architect

Keras was created in 2015 by François Chollet, a deep learning researcher at Google, with a single guiding principle: **developer experience is paramount**. Chollet observed that the most powerful research tools were often the most inaccessible, requiring thousands of lines of boilerplate for a simple experiment. Keras was designed to be the antidote.

- User-friendliness first — consistent, simple APIs that minimise cognitive load
- Modularity — every component (layers, losses, optimisers, metrics) is a standalone, composable building block
- Extensibility — power users can subclass anything and inject custom logic at any level
- Python-native — no domain-specific language, no graph compilation required for debugging
- Multi-backend — run the same code on TensorFlow, JAX, PyTorch, or NumPy (Keras 3.x)

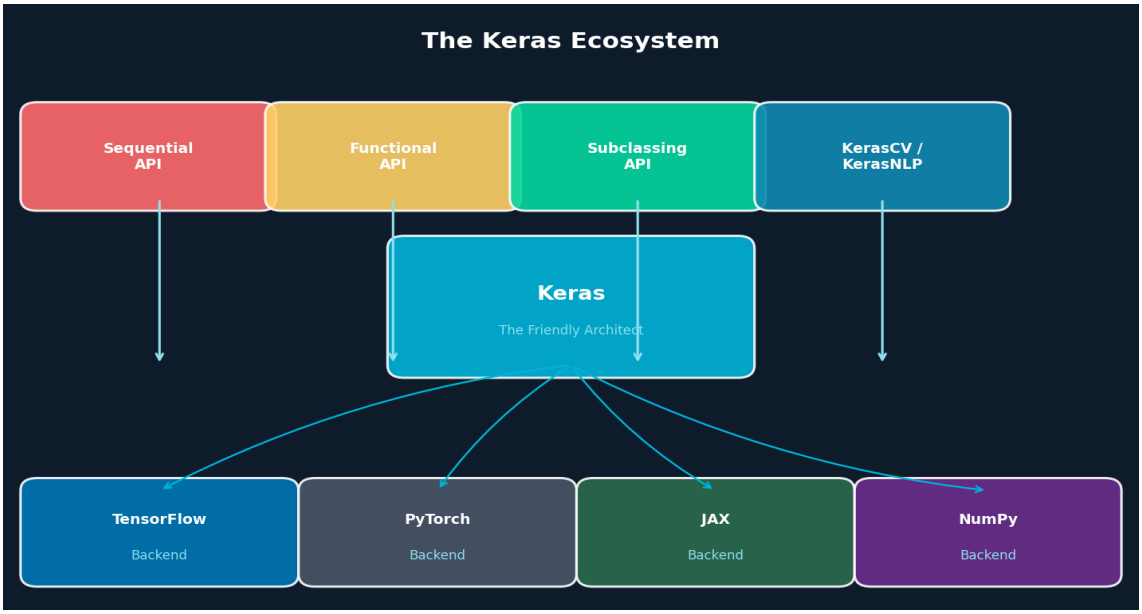

Figure 1.1 — The Keras ecosystem and its relationship to backends and high-level APIs.

1.4 Keras vs Other Frameworks

Feature	Keras	PyTorch (raw)	TensorFlow 1.x
Learning Curve	Gentle	Moderate	Steep
Debugging	Pythonic / eager	Pythonic / eager	Session-graph (complex)
Production Deploy	TF Serving / TFLite	TorchServe	TF Serving
Research Flex	High (subclassing)	Very High	Moderate
Community	Large	Very Large	Large (legacy)
Multi-backend	Yes (Keras 3)	No	No

■ Key Takeaways

- ■ Machine Learning enables systems to learn from data rather than explicit programming rules.
- ■ Deep Learning uses many-layered neural networks to learn hierarchical data representations.
- ■ Keras was purpose-built to make deep learning accessible, prioritising developer experience above all.
- ■ Keras 3.x is backend-agnostic: the same model code runs on TensorFlow, PyTorch, JAX, or NumPy.
- ■ The three ML paradigms are Supervised, Unsupervised, and Reinforcement Learning.
- ■ Keras's modular design means every component is composable and can be customised or replaced.

Chapter 2 Installation, Configuration & First Steps

Setting Up Your Environment

2.1 Installing Keras and TensorFlow

Keras ships as a standalone package (keras) and also as tf.keras inside TensorFlow. For Keras 3.x the recommended installation creates a clean Python virtual environment and installs both keras and the desired backend. Python 3.9 – 3.12 is supported.

```
# Create and activate a virtual environment
python -m venv ml_env
source ml_env/bin/activate          # Windows: ml_env\Scripts\activate

# Install Keras 3 with TensorFlow backend
pip install keras tensorflow

# Or with JAX backend (faster on TPUs)
pip install keras jax[cuda12]       # GPU JAX

# Or with PyTorch backend
pip install keras torch torchvision

# Verify installation
python -c "import keras; print(keras.__version__)"
```

2.2 GPU Support and Hardware Acceleration

Neural network training is dominated by matrix multiplications. Modern GPUs contain thousands of small cores optimised for such parallel arithmetic. Training a ResNet-50 on ImageNet takes ~days on a CPU, but only hours on a single GPU. For enterprise workloads, multi-GPU or TPU setups are standard.

```
import tensorflow as tf

# List available GPUs
gpus = tf.config.list_physical_devices('GPU')
print(f"GPUs available: {len(gpus)}")

# Enable memory growth (prevents TF from allocating all GPU RAM upfront)
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)

# Run a specific operation on GPU
with tf.device('/GPU:0'):
    a = tf.random.normal([1000, 1000])
    b = tf.random.normal([1000, 1000])
    c = tf.matmul(a, b)
```

```
print("GPU computation result shape:", c.shape)
```

2.3 Your First Keras Model

The quintessential "Hello World" of deep learning is training a network to classify the MNIST handwritten digit dataset — 70,000 grayscale 28×28 images of digits 0–9. It demonstrates the full Keras workflow in fewer than 20 lines of code.

```
import keras
from keras import layers

# Load and preprocess data
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
x_train = x_train.astype("float32") / 255.0     # Normalise to [0,1]
x_test  = x_test.astype("float32") / 255.0

# Build model
model = keras.Sequential([
    layers.Input(shape=(28, 28)),
    layers.Flatten(),                           # 28x28 → 784
    layers.Dense(128, activation="relu"),
    layers.Dropout(0.2),
    layers.Dense(10, activation="softmax"),      # 10 classes
])

# Compile
model.compile(optimizer="adam",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])

# Train
model.fit(x_train, y_train, epochs=5, batch_size=64,
          validation_split=0.1)

# Evaluate
loss, acc = model.evaluate(x_test, y_test)
print(f"Test accuracy: {acc:.4f}")              # Typically ~98%
```

■ Key Takeaways

- ■ Always use a virtual environment to isolate project dependencies.
- ■ Keras 3.x separates the framework from the backend — choose TensorFlow, JAX, or PyTorch per project.
- ■ GPU acceleration is essential for training large models; enable memory growth to avoid OOM errors.
- ■ The Keras workflow is always: load data → normalise → build model → compile → fit → evaluate.
- ■ model.summary() is your best friend for inspecting layer shapes and parameter counts.

■ MNIST is the standard "Hello World" dataset; expect ~98% test accuracy with a simple dense network.

Chapter 3 Tensors, Layers, Models, Weights

Core Keras Building Blocks

3.1 Tensors — The Universal Currency

Everything in deep learning flows through tensors — multi-dimensional arrays that generalise scalars, vectors, and matrices. A 0-D tensor is a scalar (3.14), a 1-D tensor is a vector ([1, 2, 3]), a 2-D tensor is a matrix, and higher-dimensional tensors represent batches of images (4-D: batch, height, width, channels) or sequences (3-D: batch, timesteps, features).

```
import numpy as np
import keras
from keras import ops   # Keras 3 backend-agnostic operations

# Create tensors
scalar  = keras.Variable(3.14)              # shape=()
vector  = keras.Variable([1.0, 2.0, 3.0])  # shape=(3,)
matrix  = keras.Variable(np.eye(4))         # shape=(4,4)

# dtype matters for memory and speed
x_float32 = ops.ones((32, 224, 224, 3), dtype="float32")  # ~19 MB
x_float16  = ops.ones((32, 224, 224, 3), dtype="float16") # ~9.5 MB (mixed precisio
n)

# Shape introspection
print(x_float32.shape)   # (32, 224, 224, 3)
print(x_float32.ndim)    # 4
print(x_float32.dtype)   # float32
```

3.2 Layers — The Fundamental Unit of Computation

A Layer is the fundamental building block of Keras. Every layer encapsulates a transformation: it holds learnable weights (parameters) and defines a call() method that computes outputs from inputs. When you add layers to a model and call model.fit(), Keras automatically builds a computation graph, computes gradients via automatic differentiation, and updates the weights using the chosen optimiser.

Layer Class	Use Case	Key Parameters
Dense	Fully connected, classic MLP	units, activation
Conv2D	2D spatial feature extraction (images)	filters, kernel_size, strides, padding
Conv1D	1D temporal feature extraction (sequences)	filters, kernel_size, padding
MaxPooling2D	Spatial down-sampling	pool_size, strides
LSTM	Long-range sequential dependencies	units, return_sequences, dropout

GRU	Efficient sequential modeling	units, return_sequences
Embedding	Integer tokens → dense vectors	input_dim, output_dim, mask_zero
BatchNormalization	Normalise layer activations	axis, momentum, epsilon
Dropout	Regularisation via random neuron zeroing	rate
Flatten	Collapse spatial dims to 1-D	—
Reshape	Arbitrary tensor reshaping	target_shape
Concatenate	Merge tensors along an axis	axis
MultiHeadAttention	Scaled dot-product attention	num_heads, key_dim
LayerNormalization	Normalise across features (Transformer)	axis, epsilon

3.3 Building Custom Layers

When the built-in layers don't cover your needs, you can subclass keras.layers.Layer. The two mandatory overrides are build() (create weights) and call() (define the forward pass). This is the same pattern used to implement novel research architectures.

```
import keras
from keras import layers, ops

class LinearWithBias(layers.Layer):
    """Custom layer: y = x @ W + b"""

    def __init__(self, units, **kwargs):
        super().__init__(**kwargs)
        self.units = units

    def build(self, input_shape):
        # Weights are created here — shape is now known
        self.W = self.add_weight(
            name="kernel",
            shape=(input_shape[-1], self.units),
            initializer="glorot_uniform",
            trainable=True,
        )
        self.b = self.add_weight(
            name="bias",
            shape=(self.units,),
            initializer="zeros",
            trainable=True,
        )
        super().build(input_shape)

    def call(self, x):
        return ops.matmul(x, self.W) + self.b

    def get_config(self):           # Needed for serialisation
        return {**super().get_config(), "units": self.units}
```

```
# Use like any built-in layer
layer = LinearWithBias(64)
output = layer(keras.random.normal((32, 128)))
print(output.shape)   # (32, 64)
```

■ Key Takeaways

- ■ Tensors are multi-dimensional arrays; understanding shape and dtype is essential for debugging.
- ■ Every Layer has trainable weights, a build() step, and a call() (forward pass) step.
- ■ Keras provides 50+ built-in layers covering dense, convolutional, recurrent, normalisation, and attention operations.
- ■ Custom layers are created by subclassing keras.layers.Layer and implementing build() and call().
- ■ Always implement get_config() in custom layers to enable model serialisation and loading.
- ■ Mixed-precision (float16) can reduce memory usage by ~50% with minimal accuracy impact.

Chapter 4 Sequential, Functional & Subclassing

Model-Building APIs

Keras offers three distinct APIs for constructing models, spanning the spectrum from simplest-but-limited to most-complex-but-unlimited. Choosing the right API is a key architectural decision.

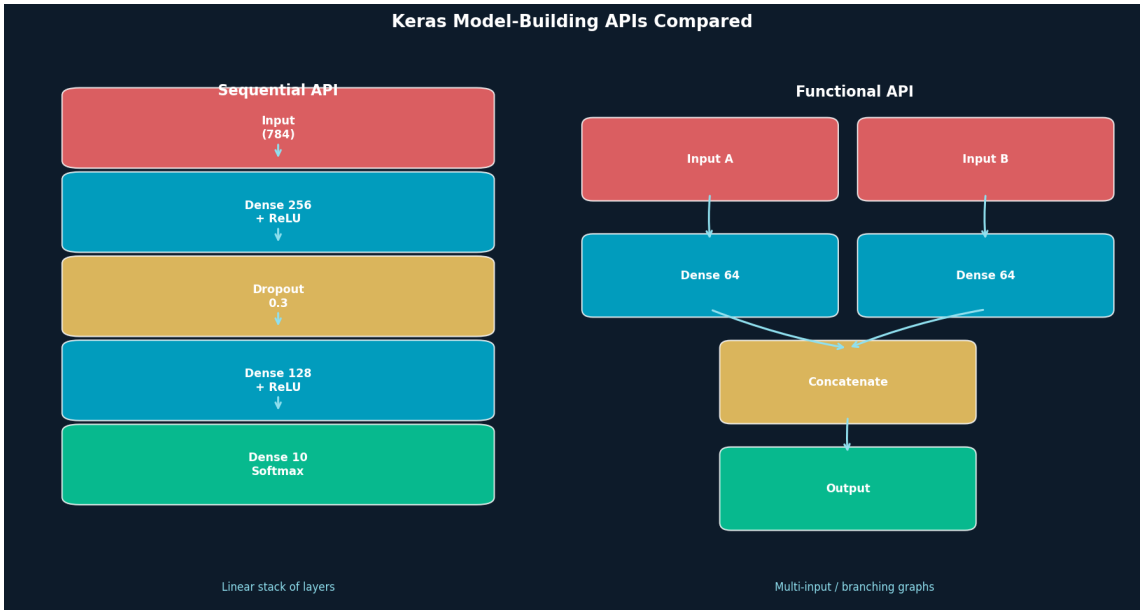

Figure 4.1 — Sequential API (left) vs Functional API (right) for a multi-input network.

4.1 The Sequential API

The Sequential API is ideal for models that are a simple linear stack of layers — each layer has exactly one input tensor and one output tensor. It is perfect for beginners and for the vast majority of classification and regression tasks.

```
import keras
from keras import layers

# Method 1: Pass layers at construction time
model = keras.Sequential([
    layers.Input(shape=(784,)),
    layers.Dense(512, activation="relu"),
    layers.BatchNormalization(),
    layers.Dropout(0.4),
    layers.Dense(256, activation="relu"),
    layers.Dropout(0.3),
    layers.Dense(10, activation="softmax"),
], name="digit_classifier")

# Method 2: Incrementally add layers
model2 = keras.Sequential(name="incremental")
model2.add(layers.Input(shape=(784,)))
```

```
model2.add(layers.Dense(256, activation="relu"))
model2.add(layers.Dense(10, activation="softmax"))

model.summary()   # prints layer table with shapes and param counts
```

4.2 The Functional API

The Functional API treats the model as a directed acyclic graph (DAG) of layers. It supports multi-input and multi-output models, shared layers, and residual/skip connections — all patterns that are impossible with the Sequential API. Almost all research-grade architectures (ResNet, Inception, BERT) are expressed using Functional-style graphs.

```
# Multi-input functional model
import keras
from keras import layers

# Define inputs
image_input = keras.Input(shape=(224, 224, 3), name="image")
meta_input  = keras.Input(shape=(10,), name="metadata")

# Image branch
x = layers.Conv2D(32, 3, activation="relu", padding="same")(image_input)
x = layers.GlobalAveragePooling2D()(x)
x = layers.Dense(64, activation="relu")(x)

# Metadata branch
y = layers.Dense(32, activation="relu")(meta_input)

# Merge
merged = layers.Concatenate()([x, y])
merged = layers.Dense(64, activation="relu")(merged)
outputs = layers.Dense(1, activation="sigmoid", name="prediction")(merged)

# Create model
model = keras.Model(inputs=[image_input, meta_input],
                    outputs=outputs,
                    name="multi_input_model")

# Residual / skip connection example
def residual_block(x, filters):
    shortcut = x
    x = layers.Conv2D(filters, 3, padding="same", activation="relu")(x)
    x = layers.Conv2D(filters, 3, padding="same")(x)
    x = layers.Add()([x, shortcut])   # Skip connection!
    return layers.Activation("relu")(x)
```

4.3 Model Subclassing

Subclassing keras.Model gives you the maximum flexibility. You define __init__ (to create sub-layers) and call() (to define the forward pass with full Python control flow). This is the approach used when

you need conditionals, loops, or dynamic computation graphs.

```
class TransformerBlock(keras.Model):
    def __init__(self, embed_dim, num_heads, ff_dim, rate=0.1):
        super().__init__()
        self.att  = layers.MultiHeadAttention(num_heads=num_heads, key_dim=embed_di
m)
        self.ffn  = keras.Sequential([
            layers.Dense(ff_dim, activation="relu"),
            layers.Dense(embed_dim),
        ])
        self.norm1   = layers.LayerNormalization(epsilon=1e-6)
        self.norm2   = layers.LayerNormalization(epsilon=1e-6)
        self.drop1   = layers.Dropout(rate)
        self.drop2   = layers.Dropout(rate)

    def call(self, inputs, training=False):
        # Self-attention
        attn_out = self.att(inputs, inputs)
        attn_out = self.drop1(attn_out, training=training)
        out1     = self.norm1(inputs + attn_out)    # Residual

        # Feed-forward
        ffn_out  = self.ffn(out1)
        ffn_out  = self.drop2(ffn_out, training=training)
        return self.norm2(out1 + ffn_out)           # Residual
```

■ Key Takeaways

- ■ Sequential API: use for simple linear stacks — fast to write, easy to understand.
- ■ Functional API: use for multi-input, multi-output, shared layers, or skip connections.
- ■ Subclassing API: use when you need full Python control flow (dynamic graphs, custom training loops).
- ■ All three APIs produce the same underlying computation — choose based on your architecture's complexity.
- ■ Residual/skip connections (Add layer) are critical for training very deep networks without vanishing gradients.
- ■ model.summary() and keras.utils.plot_model() are essential tools for model introspection.

Chapter 5 Compilation, Fitting, Loss & Optimisers

Training Neural Networks

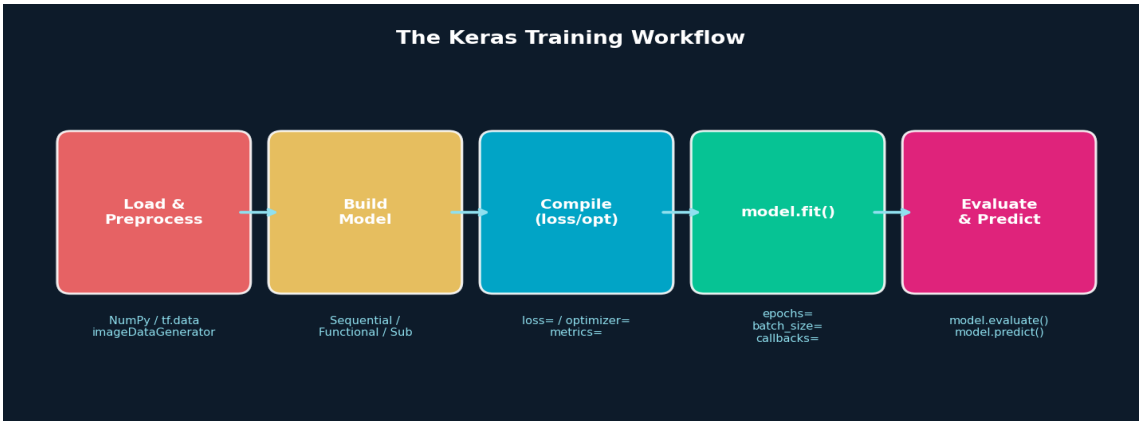

Figure 5.1 — The complete Keras training workflow from data to predictions.

5.1 The Compilation Step

Before training, a Keras model must be compiled. This step configures three essential components: the optimiser (how to update weights), the loss function (what to minimise), and the metrics (what to monitor during training). Compilation is not just configuration — Keras uses it to build and cache the computation graph for efficiency.

```
model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=1e-3),
    loss=keras.losses.CategoricalCrossentropy(from_logits=False),
    metrics=[
        keras.metrics.CategoricalAccuracy(name="accuracy"),
        keras.metrics.AUC(name="auc"),
        keras.metrics.Precision(name="precision"),
    ]
)

# Shorthand strings also work
model.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"],
)
```

5.2 The model.fit() Deep Dive

model.fit() is where the magic happens. It orchestrates the forward pass, loss computation, backward pass (gradient computation via automatic differentiation), and weight update for every batch over every epoch. Understanding its parameters is essential for efficient training.

```
history = model.fit(
    x_train, y_train,              # Data (NumPy arrays or tf.data.Dataset)
    batch_size=64,                 # Samples per gradient update
    epochs=50,                     # Full passes through the training data
    validation_split=0.15,         # 15% of training data held out for val
    # OR:
    validation_data=(x_val, y_val),
    shuffle=True,                  # Shuffle before each epoch
    class_weight={0: 1.0, 1: 3.0},# For imbalanced datasets
    sample_weight=None,            # Per-sample importance weights
    callbacks=[                    # See Chapter 10
        keras.callbacks.EarlyStopping(patience=5, restore_best_weights=True),
        keras.callbacks.ModelCheckpoint("best_model.keras"),
    ],
    verbose=1,                     # 0=silent, 1=progress bar, 2=one line/epoch
)

# history.history is a dict: {"loss": [...], "val_loss": [...], "accuracy": [...]}
import matplotlib.pyplot as plt
plt.plot(history.history["val_accuracy"])
```

5.3 Loss Functions

Loss Function	Task	Notes
BinaryCrossentropy	Binary classification	from_logits=True if no sigmoid at output
CategoricalCrossentropy	Multi-class (one-hot labels)	Most common for classification
SparseCategoricalCrossentropy	Multi-class (integer labels)	Saves one-hot encoding step
MeanSquaredError (MSE)	Regression	Penalises large errors heavily
MeanAbsoluteError (MAE)	Regression	More robust to outliers than MSE
Huber	Regression	Hybrid: MSE for small errors, MAE for large
KLDivergence	Distribution matching	Used in VAEs and distillation
CosineSimilarity	Embedding tasks	Useful for sentence similarity
Hinge / SquaredHinge	SVM-style classification	Used for margin-based objectives
LogCosh	Regression	Smooth, less sensitive to outliers

5.4 Optimisers — Navigating the Loss Landscape

An optimiser adjusts the model's weights to minimise the loss function. The update rule determines how aggressively and intelligently weights shift after each gradient computation. The choice of optimiser and learning rate are among the most impactful hyperparameters in deep learning.

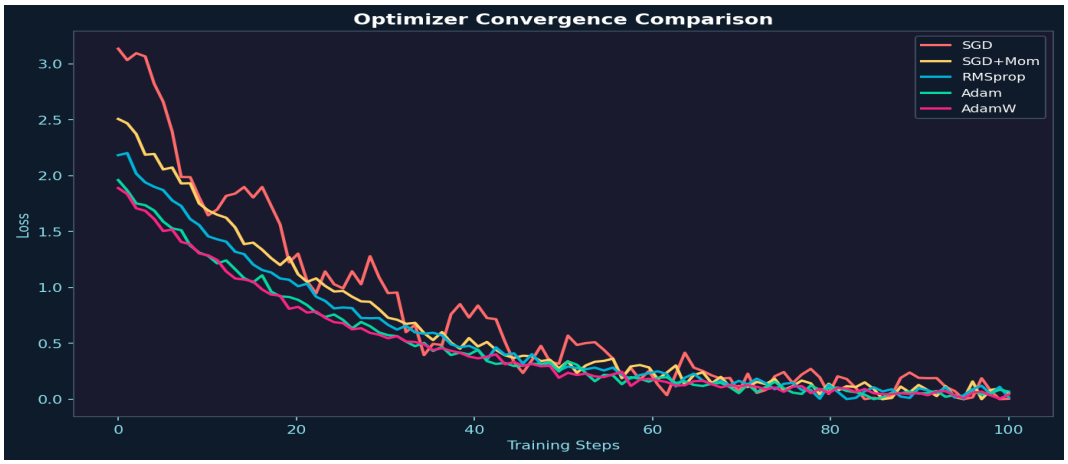

Figure 5.2 — Convergence speed comparison of popular Keras optimisers.

SGD (Stochastic Gradient Descent): The simplest optimiser. Update: w = w - lr * grad. With momentum=0.9, it accumulates a velocity vector in the direction of the gradient, dampening oscillations and speeding convergence. Nesterov momentum is a look-ahead variant that typically improves convergence further.

Adam (Adaptive Moment Estimation): The most popular optimiser for most tasks. Combines ideas from RMSprop (per-parameter adaptive learning rates) and momentum (first moment). Maintains running estimates of both first (mean) and second (variance) moments of gradients. Default lr=1e-3.

AdamW: Adam with decoupled weight decay. The original Adam conflates L2 regularisation with weight decay; AdamW corrects this, leading to better generalisation. Recommended default for most modern architectures.

RMSprop: Maintains a per-parameter moving average of squared gradients. Divides the learning rate by this average, giving larger updates for infrequent features. Popular for RNNs.

■ Key Takeaways

- ■ model.compile() sets the optimiser, loss function, and metrics — this cannot be changed after fitting.
- ■ Adam / AdamW is the go-to optimiser for most tasks; fine-tune learning rate first.
- ■ Use SparseCategoricalCrossentropy when your labels are integers, Categorical when one-hot encoded.
- ■ batch_size controls the trade-off between gradient noise (small batches) and memory usage (large batches).
- ■ history.history stores all training metrics per epoch — always plot these to diagnose training issues.
- ■ From-logits losses (from_logits=True) are numerically more stable than applying softmax/sigmoid first.

Chapter 6 Non-linearity, Bias-Variance & Generalisation

Activation Functions & Regularisation

6.1 Activation Functions

Without activation functions, a neural network — regardless of how many layers it has — would reduce to a single linear transformation. Activation functions introduce non-linearity after each layer, allowing the network to learn complex, non-linear mappings from inputs to outputs.

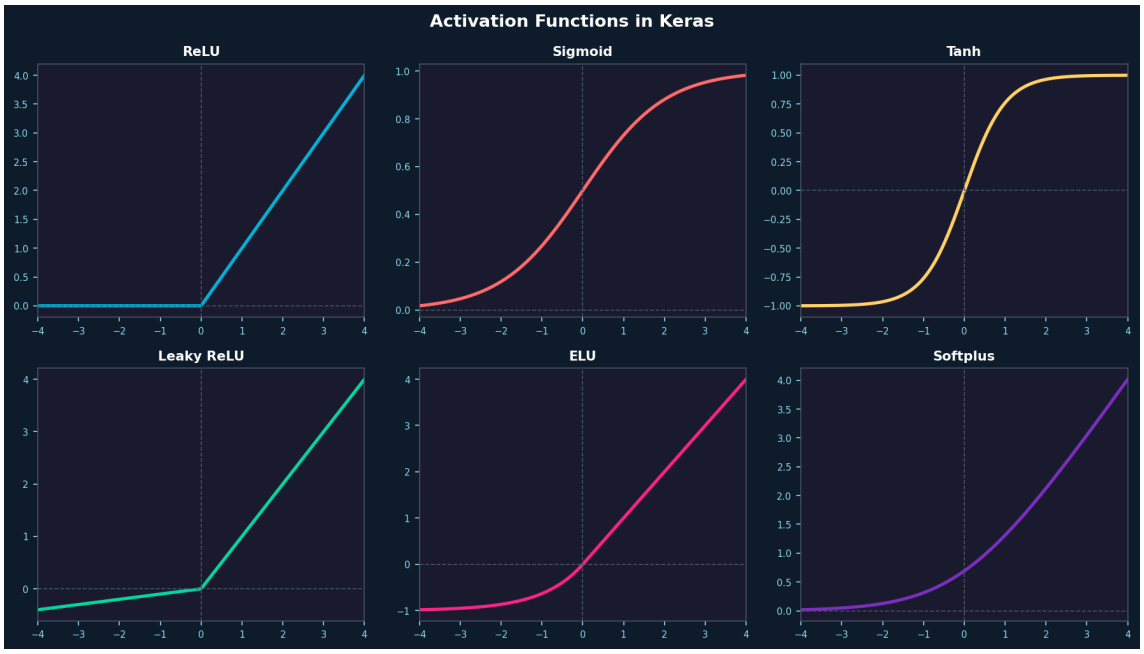

Figure 6.1 — Six common activation functions and their output shapes.

Activation	Formula	Range	Use Case
ReLU	max(0, x)	[0, ∞)	Default for hidden layers; fast, sparse
Leaky ReLU	x if x>0 else 0.01x	(-∞, ∞)	Fixes "dying ReLU" problem
ELU	x if x>0 else a(e^x-1)	(-a, ∞)	Smoother than ReLU; zero-centred
GELU	x * Phi(x)	(-∞, ∞)	Used in Transformers (BERT, GPT)
Sigmoid	1/(1+e^-x)	(0, 1)	Binary classification output
Tanh	(e^x-e^-x)/(e^x+e^-x)	(-1, 1)	Zero-centred; used in LSTMs/GRUs
Softmax	e^xi / sum(e^xj)	(0,1) sum=1	Multi-class output layer
Swish	x * sigmoid(x)	(-∞, ∞)	EfficientNet; often beats ReLU

6.2 Regularisation Techniques

Overfitting occurs when a model memorises training examples rather than learning generalisable patterns. The model achieves near-perfect training accuracy but fails on new data. Regularisation techniques constrain the model's capacity or introduce noise to prevent overfitting.

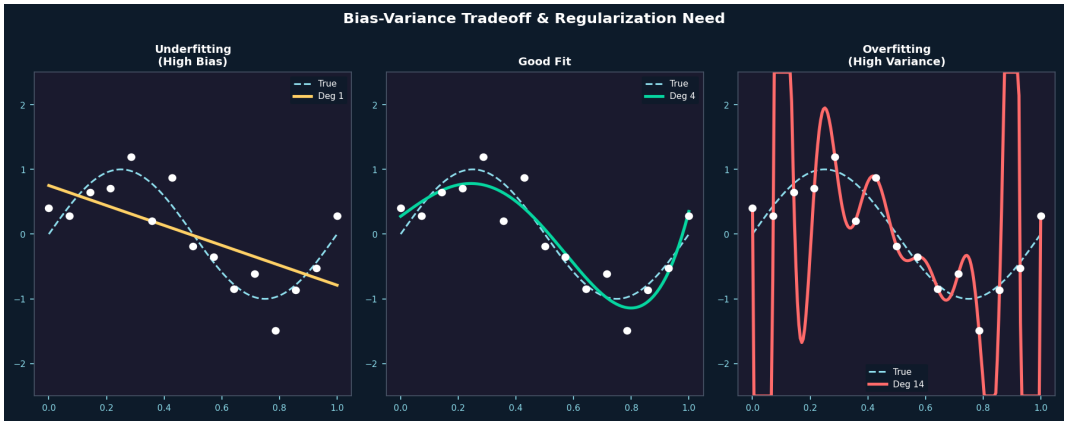

Figure 6.2 — Underfitting vs Good Fit vs Overfitting illustrated with polynomial regression.

Dropout

Dropout randomly sets a fraction rate of neuron activations to zero during each training step. This forces the network to learn redundant representations and prevents neurons from co-adapting. At inference time, all neurons are active but their outputs are scaled by (1-rate) for consistency. Dropout rates of 0.2–0.5 are typical; higher rates for larger models.

```
# Dropout in practice
model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dropout(0.5),           # Drop 50% of neurons during training
    layers.Dense(256, activation="relu"),
    layers.Dropout(0.3),
    layers.Dense(10, activation="softmax"),
])
# Dropout is automatically disabled during model.evaluate() and model.predict()
```

Batch Normalisation

Batch Normalisation (BN) normalises each mini-batch's activations to have zero mean and unit variance, then applies learnable scale (gamma) and shift (beta) parameters. Benefits: (1) allows higher learning rates, (2) reduces sensitivity to weight initialisation, (3) acts as a regulariser (often replacing Dropout in CNNs). BN is applied after the linear transformation and before the activation function.

```
layers.Conv2D(64, 3, use_bias=False),   # BN has its own bias term
layers.BatchNormalization(),              # Normalise, then scale+shift
layers.Activation("relu"),               # Activation after BN
```

L1 and L2 Regularisation

L2 regularisation (weight decay) adds the sum of squared weights × lambda to the loss, penalising large weights and encouraging smaller, more distributed representations. L1 regularisation adds the sum of absolute weights, pushing less-important weights to exactly zero, producing sparse models.

L1+L2 (ElasticNet) combines both.

```
from keras import regularizers

layers.Dense(256, activation="relu",
             kernel_regularizer=regularizers.L2(l2=1e-4),
             bias_regularizer=regularizers.L1(l1=1e-5))

# Combined L1 + L2
layers.Dense(256, activation="relu",
             kernel_regularizer=regularizers.L1L2(l1=1e-5, l2=1e-4))
```

■ Key Takeaways

- ■ ReLU is the default activation for hidden layers; use GELU for Transformer architectures.
- ■ Sigmoid is for binary output (one neuron); Softmax is for multi-class output (N neurons).
- ■ Dropout is a powerful stochastic regulariser; always disable it during inference (Keras does this automatically).
- ■ Batch Normalisation stabilises training and often allows higher learning rates.
- ■ L2 regularisation penalises large weights; L1 drives weights to zero (sparsity).
- ■ Always monitor val_loss vs train_loss — a large gap is the primary signal of overfitting.

Chapter 7 Spatial Feature Extraction for Images

Convolutional Neural Networks (CNNs)

7.1 The Convolution Operation

Convolutional Neural Networks exploit the spatial structure of images. Instead of connecting every pixel to every neuron (fully connected), a Conv2D layer slides a small filter (kernel) across the image, computing a dot product at each position. This gives CNNs three key properties: (1) local connectivity — each neuron sees only a local patch, (2) weight sharing — the same filter is used across all positions, drastically reducing parameters, and (3) translation equivariance — a detected feature is recognised regardless of where it appears.

```
from keras import layers

# Basic Conv2D
conv = layers.Conv2D(
    filters=32,          # Number of output feature maps
    kernel_size=(3, 3),  # Spatial extent of each filter
    strides=(1, 1),      # Step size (default 1)
    padding="same",      # "same" preserves spatial dims; "valid" shrinks them
    activation="relu",
    use_bias=True,
    kernel_initializer="he_normal",  # Good for ReLU
)

# Depthwise Separable Conv — far fewer params, used in MobileNet
layers.SeparableConv2D(filters=64, kernel_size=3, activation="relu")

# Dilated / Atrous Conv — expands receptive field without pooling
layers.Conv2D(64, 3, dilation_rate=2, padding="same", activation="relu")
```

7.2 Pooling and Spatial Hierarchies

Pooling layers reduce spatial dimensions, building a hierarchy where early layers detect edges and textures while deeper layers detect parts and objects. MaxPooling2D takes the maximum value in each pooling window (retains dominant features); AveragePooling2D averages them. GlobalAveragePooling2D collapses the entire spatial map to a single vector — the modern replacement for the Flatten + Dense head in classification CNNs.

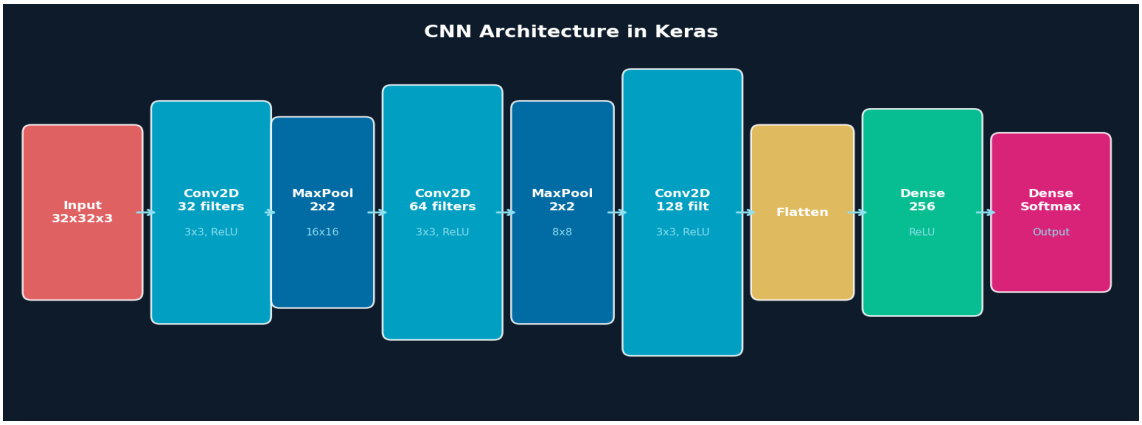

Figure 7.1 — A typical CNN architecture progressing from raw pixels to class probabilities.

7.3 Building Production-Grade CNNs

```
def build_cnn(input_shape=(224,224,3), num_classes=10):
    inputs = keras.Input(shape=input_shape)

    # Block 1
    x = layers.Conv2D(32, 3, padding="same")(inputs)
    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)
    x = layers.MaxPooling2D(2)(x)           # 224->112

    # Block 2
    x = layers.Conv2D(64, 3, padding="same")(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)
    x = layers.MaxPooling2D(2)(x)           # 112->56

    # Block 3 with residual
    shortcut = layers.Conv2D(128, 1, padding="same")(x)  # 1x1 projection
    x = layers.Conv2D(128, 3, padding="same")(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)
    x = layers.Conv2D(128, 3, padding="same")(x)
    x = layers.BatchNormalization()(x)
    x = layers.Add()([x, shortcut])
    x = layers.Activation("relu")(x)
    x = layers.MaxPooling2D(2)(x)           # 56->28

    # Classification head
    x = layers.GlobalAveragePooling2D()(x)   # 28x28x128 -> 128
    x = layers.Dense(256, activation="relu")(x)
    x = layers.Dropout(0.5)(x)
    outputs = layers.Dense(num_classes, activation="softmax")(x)

    return keras.Model(inputs, outputs, name="custom_cnn")
```

7.4 Data Augmentation for Images

Data augmentation artificially expands the training dataset by applying random but realistic transformations to images. This dramatically reduces overfitting because the model never sees exactly the same image twice. Keras provides a preprocessing layer-based augmentation pipeline that runs on the GPU during training.

```
# GPU-accelerated data augmentation as model layers
data_augmentation = keras.Sequential([
    layers.RandomFlip("horizontal"),
    layers.RandomRotation(0.1),              # ±10% rotation
    layers.RandomZoom(0.15),
    layers.RandomTranslation(0.1, 0.1),
    layers.RandomContrast(0.2),
    layers.RandomBrightness(0.2),
], name="augmentation")

# Only active during training — automatically disabled at inference
model = keras.Sequential([
    data_augmentation,
    layers.Conv2D(32, 3, activation="relu"),
    ...
])
```

■ Key Takeaways

- ■ Conv2D layers detect local spatial patterns via shared weight filters — far more efficient than Dense for images.
- ■ padding="same" preserves spatial dimensions; padding="valid" shrinks them (kernel_size - 1) per side.
- ■ MaxPooling2D reduces spatial size; GlobalAveragePooling2D collapses to a 1-D feature vector.
- ■ Batch Normalisation + ReLU after every Conv2D is the modern standard pattern.
- ■ Residual (skip) connections are the secret behind training 50–1000+ layer networks without vanishing gradients.
- ■ Use Keras preprocessing layers for data augmentation — they run on GPU and are part of the model graph.

Chapter 8 LSTMs, GRUs & Temporal Patterns

Recurrent Networks & Sequence Modelling

8.1 The Recurrence Idea

Many real-world problems involve sequences: text, audio, time series, video. The defining characteristic of sequences is that order matters and earlier elements influence the interpretation of later ones. Standard Dense or Conv layers treat each input independently, ignoring temporal order. Recurrent Neural Networks (RNNs) maintain a hidden state that is updated at each timestep, allowing information to flow from one step to the next.

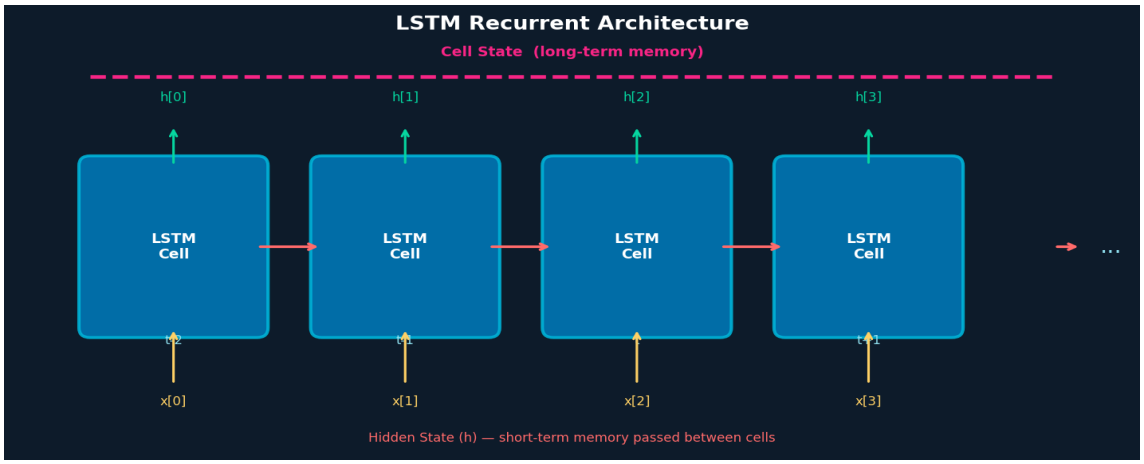

Figure 8.1 — LSTM architecture unrolled over time, showing hidden state and cell state flow.

8.2 LSTM — Long Short-Term Memory

The vanishing gradient problem plagues vanilla RNNs: gradients shrink exponentially as they flow back through many timesteps, making it impossible to learn long-range dependencies. LSTMs solve this by introducing a separate cell state (long-term memory) and three learnable gates: the Forget Gate (what to erase from memory), the Input Gate (what new information to write), and the Output Gate (what to read out as the hidden state). The cell state highway allows gradients to flow back without decay.

```
# Stacked Bidirectional LSTM for sequence classification
import keras
from keras import layers

model = keras.Sequential([
    layers.Input(shape=(100, 64)),          # (timesteps, features)

    layers.Bidirectional(
        layers.LSTM(128, return_sequences=True, dropout=0.2, recurrent_dropout=0.1)
    ),

    layers.Bidirectional(
        layers.LSTM(64, return_sequences=False)  # returns final state only
```

```
    ),

    layers.Dense(64, activation="relu"),
    layers.Dropout(0.3),
    layers.Dense(5, activation="softmax"),
], name="bilstm_classifier")

# For many-to-many: return_sequences=True in all LSTM layers
# For many-to-one: return_sequences=False in the final LSTM layer
```

8.3 GRU — Gated Recurrent Unit

GRU is a simplified version of LSTM with only two gates: the Update Gate (combined forget+input) and the Reset Gate. It has fewer parameters than LSTM, trains faster, and often achieves comparable performance on shorter sequences. A good rule of thumb: try GRU first for speed, switch to LSTM if you need to model very long-range dependencies.

```
# Text classification with GRU
vocab_size = 20000
embed_dim  = 128
max_len    = 200

model = keras.Sequential([
    layers.Input(shape=(max_len,), dtype="int32"),
    layers.Embedding(vocab_size, embed_dim, mask_zero=True),
    layers.GRU(128, return_sequences=True),
    layers.GRU(64),
    layers.Dense(1, activation="sigmoid"),
])
model.compile("adam", "binary_crossentropy", metrics=["accuracy"])
```

8.4 Text Preprocessing with Keras

```
from keras.layers import TextVectorization

# Adapt vocabulary to your training corpus
vectorizer = TextVectorization(
    max_tokens=20000,
    output_mode="int",
    output_sequence_length=200,
)
vectorizer.adapt(train_texts)              # Learns vocabulary

# Use as first layer in model
inputs  = keras.Input(shape=(1,), dtype="string")
x       = vectorizer(inputs)              # str → token IDs
x       = layers.Embedding(20000, 128)(x)
x       = layers.GRU(64)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model   = keras.Model(inputs, outputs)
```

■ Key Takeaways

- ■ RNNs process sequences by maintaining a hidden state that accumulates context from previous timesteps.
- ■ LSTMs use a separate cell state and three gates to preserve long-range dependencies and fight vanishing gradients.
- ■ GRUs are lighter than LSTMs with two gates; prefer GRU for shorter sequences and faster training.
- ■ Bidirectional wrappers process sequences forward and backward, doubling contextual information.
- ■ return_sequences=True passes the full sequence to the next layer; False returns only the final hidden state.
- ■ TextVectorization + Embedding is the Keras-native pipeline for raw text → dense vector sequences.

Chapter 9 Standing on the Shoulders of Giants

Transfer Learning & Fine-Tuning

9.1 The Transfer Learning Paradigm

Training a large CNN from scratch requires millions of labelled examples and days of GPU compute. Transfer learning sidesteps this by starting with a model pre-trained on a massive dataset (ImageNet, with 1.4 million images and 1000 classes) and adapting it to your specific task, often with just hundreds or thousands of examples. The intuition: the early convolutional layers of any image model learn universal features (edges, textures, shapes) that are useful for almost any visual task.

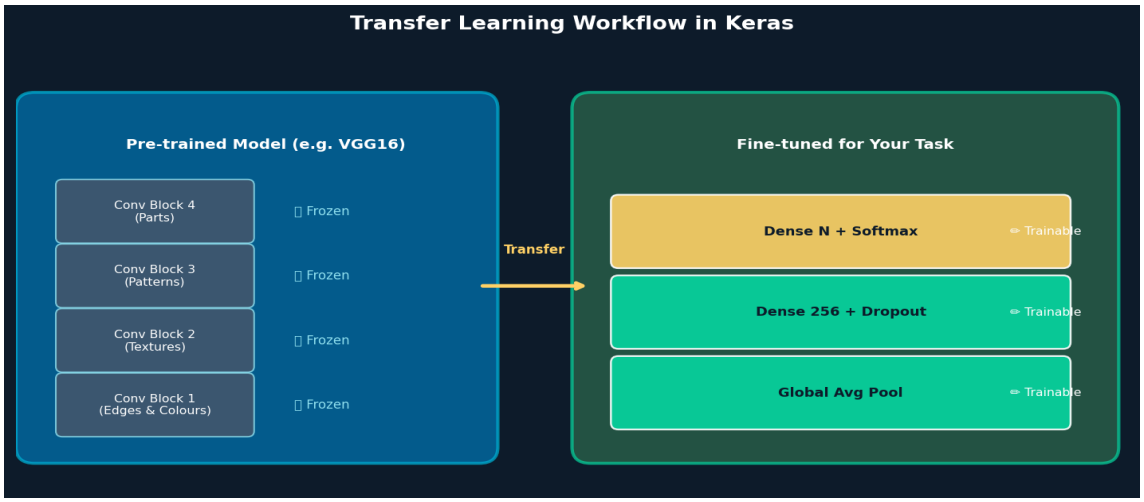

Figure 9.1 — Transfer learning: frozen pre-trained layers + trainable new head.

9.2 Feature Extraction

In the simplest form of transfer learning, the pre-trained model acts as a fixed feature extractor. Its weights are frozen (not updated during training), and only the newly added classification head is trained. This is computationally cheap and works well when your dataset is small and similar to the pre-training dataset.

```
import keras

# Load pre-trained base — weights from ImageNet, no top classification head
base_model = keras.applications.EfficientNetV2B0(
    weights="imagenet",
    include_top=False,          # Remove the 1000-class softmax head
    input_shape=(224, 224, 3),
)

# Freeze all layers — weights are not updated during training
base_model.trainable = False

# Add a new classification head
inputs  = keras.Input(shape=(224, 224, 3))
```

```
x       = base_model(inputs, training=False)   # training=False → BN in inference
mode
x       = keras.layers.GlobalAveragePooling2D()(x)
x       = keras.layers.Dense(256, activation="relu")(x)
x       = keras.layers.Dropout(0.5)(x)
outputs = keras.layers.Dense(5, activation="softmax")(x)

model = keras.Model(inputs, outputs)
model.compile("adam", "sparse_categorical_crossentropy", metrics=["accuracy"])
model.fit(train_dataset, epochs=10)    # Only head trains — fast!
```

9.3 Fine-Tuning

After training the new head for several epochs, you can optionally "unfreeze" some of the top layers of the base model and continue training with a very low learning rate. This allows the pre-trained features to be gently adapted to your specific domain. Fine-tuning too many layers or with too high a learning rate risks destroying the pre-trained representations (catastrophic forgetting).

```
# Phase 1: Train head only (already done above)

# Phase 2: Fine-tune top layers of base model
base_model.trainable = True

# Unfreeze only the top 30 layers (tune the most task-specific ones)
for layer in base_model.layers[:-30]:
    layer.trainable = False

# Recompile with a much lower learning rate
model.compile(
    optimizer=keras.optimizers.Adam(1e-5),    # 100x lower than Phase 1
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"],
)
model.fit(train_dataset, epochs=20,
          callbacks=[keras.callbacks.EarlyStopping(patience=5)])
```

9.4 Available Pre-trained Models in Keras

Model	Params	Top-1 Acc	Best For
VGG16/19	138M / 143M	71–74%	Simple baseline, interpretable
ResNet50/101	25M / 44M	76–77%	General purpose; good accuracy/speed
InceptionV3	23M	78%	Multi-scale features
MobileNetV2/V3	3.4M / 5.4M	71–75%	Mobile & edge deployment
EfficientNetB0–B7	5–66M	77–84%	Best accuracy-per-parameter ratio
EfficientNetV2-S/M/L	21–119M	84–86%	State-of-the-art image classification
ConvNeXt-T/S/B	28–88M	82–84%	Modern CNN rivalling ViT

Xception	22M	79%	Depthwise separable; fast inference

■ Key Takeaways

- ■ Transfer learning leverages representations from large pre-trained models, dramatically reducing data and compute needs.
- ■ Feature extraction: freeze base model, train only the new head — fast and data-efficient.
- ■ Fine-tuning: unfreeze top layers and train with a very low learning rate (1e-5 to 1e-6).
- ■ Always pass training=False to the base model during feature extraction to keep BatchNorm in inference mode.
- ■ EfficientNetV2 offers the best accuracy-per-parameter trade-off for image tasks as of 2024.
- ■ Prevent catastrophic forgetting by using a learning rate 10–100x lower during fine-tuning than during head training.

Chapter 10 Automated Monitoring, Saving & Scheduling

Callbacks & Training Control

Keras Callbacks are objects that can perform actions at various stages of training — at the start and end of epochs, before and after each batch, on training start and end. They are passed as a list to model.fit(callbacks=[...]).

Figure 10.1 — Six essential Keras callbacks and their roles.

10.1 Built-in Callbacks Deep Dive

ModelCheckpoint

Automatically saves the best model to disk during training. Use .keras format for full model serialisation.

```
keras.callbacks.ModelCheckpoint(
    filepath="best_model.keras",
    monitor="val_accuracy",
    save_best_only=True,        # Only save when metric improves
    mode="max",                 # "max" for accuracy, "min" for loss
    save_weights_only=False,
    verbose=1,
)
```

EarlyStopping

Prevents wasted compute and overfitting by halting training when validation performance stops improving.

```
keras.callbacks.EarlyStopping(
    monitor="val_loss",
    patience=10,              # Stop if no improvement for 10 epochs
    min_delta=1e-4,           # Minimum change to count as improvement
    restore_best_weights=True, # Roll back to the epoch with best val_loss
    mode="min",
    verbose=1,
```

```
)
```

ReduceLROnPlateau

Reduces the learning rate when training stagnates, often breaking through plateaus.

```
keras.callbacks.ReduceLROnPlateau(
    monitor="val_loss",
    factor=0.5,           # New LR = current LR * factor
    patience=5,
    min_lr=1e-7,
    cooldown=2,           # Epochs to wait before resuming monitoring
    verbose=1,
)
```

10.2 Custom Callbacks

```
class LearningRateLogger(keras.callbacks.Callback):
    """Logs the current learning rate at the end of each epoch."""

    def on_epoch_end(self, epoch, logs=None):
        lr = float(self.model.optimizer.learning_rate)
        logs["lr"] = lr
        print(f"\n  Epoch {epoch+1} — LR: {lr:.2e}")

class WarmupCosineDecay(keras.callbacks.Callback):
    """Linear warmup + cosine learning rate decay schedule."""

    def __init__(self, total_steps, warmup_steps, max_lr, min_lr=1e-7):
        super().__init__()
        self.total = total_steps
        self.warmup = warmup_steps
        self.max_lr = max_lr
        self.min_lr = min_lr

    def on_train_batch_begin(self, batch, logs=None):
        step = self.model.optimizer.iterations
        if step < self.warmup:
            lr = self.max_lr * step / self.warmup
        else:
            progress = (step - self.warmup) / (self.total - self.warmup)
            lr = self.min_lr + 0.5*(self.max_lr - self.min_lr)*(1 + np.cos(np.pi*pr
ogress))
        self.model.optimizer.learning_rate.assign(lr)
```

■ Key Takeaways

■ Callbacks execute at well-defined training hooks — epoch start/end, batch start/end, training start/end.

- Always use ModelCheckpoint + EarlyStopping together: save the best, stop when no longer improving.
- ReduceLROnPlateau is a simple and effective schedule — reduce LR when stuck, with patience and cooldown.
- TensorBoard provides rich, real-time visualisations of loss curves, histograms, images, and more.
- Custom callbacks inherit from keras.callbacks.Callback and override on_epoch_end, on_batch_begin, etc.
- Warmup + cosine decay is the modern learning rate schedule of choice for Transformer models.

Chapter 11 Autoencoders, GANs & Transformers

Advanced Architectures

11.1 Autoencoders

An autoencoder is an unsupervised neural network that learns to compress data into a low-dimensional latent representation (encoder) and then reconstruct it (decoder). The bottleneck forces the model to learn the most salient features. Applications include dimensionality reduction, anomaly detection (reconstruction error spikes for anomalies), image denoising, and as a component of generative models.

```
# Convolutional Autoencoder for image compression
import keras
from keras import layers

# Encoder
encoder_input  = keras.Input(shape=(28,28,1))
x = layers.Conv2D(32, 3, activation="relu", padding="same", strides=2)(encoder_inpu
t) # 14x14
x = layers.Conv2D(64, 3, activation="relu", padding="same", strides=2)(x)
   # 7x7
x = layers.Flatten()(x)
latent = layers.Dense(32, name="latent")(x)
    # Bottleneck

encoder = keras.Model(encoder_input, latent, name="encoder")

# Decoder
decoder_input = keras.Input(shape=(32,))
x = layers.Dense(7*7*64, activation="relu")(decoder_input)
x = layers.Reshape((7,7,64))(x)
x = layers.Conv2DTranspose(64, 3, activation="relu", padding="same", strides=2)(x)
   # 14x14
x = layers.Conv2DTranspose(32, 3, activation="relu", padding="same", strides=2)(x)
   # 28x28
decoder_output = layers.Conv2DTranspose(1, 3, activation="sigmoid", padding="same")
(x)

decoder = keras.Model(decoder_input, decoder_output, name="decoder")

# Full autoencoder
autoencoder_input = keras.Input(shape=(28,28,1))
encoded = encoder(autoencoder_input)
decoded = decoder(encoded)
autoencoder = keras.Model(autoencoder_input, decoded, name="autoencoder")
autoencoder.compile("adam", "binary_crossentropy")
```

11.2 Generative Adversarial Networks (GANs)

GANs pit two networks against each other in a minimax game. The Generator (G) produces fake samples from random noise; the Discriminator (D) tries to distinguish real from fake. G improves to fool D; D improves to catch G. At equilibrium, G produces samples indistinguishable from real data. Keras's subclassing API is ideal for GANs because the training loop requires custom logic.

```
class GAN(keras.Model):
    def __init__(self, discriminator, generator, latent_dim):
        super().__init__()
        self.discriminator = discriminator
        self.generator     = generator
        self.latent_dim    = latent_dim

    def compile(self, d_optimizer, g_optimizer, loss_fn):
        super().compile()
        self.d_opt = d_optimizer
        self.g_opt = g_optimizer
        self.loss_fn = loss_fn

    def train_step(self, real_images):
        batch_size = tf.shape(real_images)[0]
        noise      = tf.random.normal((batch_size, self.latent_dim))

        # Train Discriminator
        with tf.GradientTape() as tape:
            fake_images = self.generator(noise, training=True)
            real_logits = self.discriminator(real_images, training=True)
            fake_logits = self.discriminator(fake_images, training=True)
            d_real_loss = self.loss_fn(tf.ones_like(real_logits), real_logits)
            d_fake_loss = self.loss_fn(tf.zeros_like(fake_logits), fake_logits)
            d_loss = (d_real_loss + d_fake_loss) / 2
        d_grads = tape.gradient(d_loss, self.discriminator.trainable_weights)
        self.d_opt.apply_gradients(zip(d_grads, self.discriminator.trainable_weight
s))

        # Train Generator
        with tf.GradientTape() as tape:
            fake = self.generator(noise, training=True)
            logits = self.discriminator(fake, training=False)
            g_loss = self.loss_fn(tf.ones_like(logits), logits)
        g_grads = tape.gradient(g_loss, self.generator.trainable_weights)
        self.g_opt.apply_gradients(zip(g_grads, self.generator.trainable_weights))
        return {"d_loss": d_loss, "g_loss": g_loss}
```

11.3 Transformers in Keras

The Transformer architecture, introduced in "Attention is All You Need" (Vaswani et al., 2017), replaced recurrence with self-attention mechanisms and has since become the dominant architecture for NLP (BERT, GPT), vision (ViT), and multimodal AI. Keras provides MultiHeadAttention and LayerNormalization as built-in layers for constructing Transformer blocks.

```
def build_text_transformer(vocab_size=20000, max_len=200,
                           embed_dim=128, num_heads=4, ff_dim=256, num_layers=2)
:
    inputs  = keras.Input(shape=(max_len,), dtype="int32")

    # Token + positional embeddings
    token_emb = layers.Embedding(vocab_size, embed_dim)(inputs)
    pos_emb   = layers.Embedding(max_len, embed_dim)(tf.range(max_len))
    x = token_emb + pos_emb

    # Stack Transformer blocks
    for _ in range(num_layers):
        # Multi-Head Self-Attention
        attn_out = layers.MultiHeadAttention(num_heads=num_heads, key_dim=embed_dim
//num_heads)(x, x)
        attn_out = layers.Dropout(0.1)(attn_out)
        x = layers.LayerNormalization(epsilon=1e-6)(x + attn_out)   # Add & Norm

        # Feed-Forward Network
        ffn = layers.Dense(ff_dim, activation="relu")(x)
        ffn = layers.Dense(embed_dim)(ffn)
        ffn = layers.Dropout(0.1)(ffn)
        x = layers.LayerNormalization(epsilon=1e-6)(x + ffn)        # Add & Norm

    # Classification head
    x = layers.GlobalAveragePooling1D()(x)
    outputs = layers.Dense(2, activation="softmax")(x)
    return keras.Model(inputs, outputs)
```

■ Key Takeaways

- ■ Autoencoders learn compact latent representations; reconstruction error enables anomaly detection.
- ■ GANs train a generator and discriminator in adversarial opposition — custom train_step is required.
- ■ The Transformer's self-attention mechanism weighs every position against every other — no recurrence needed.
- ■ MultiHeadAttention and LayerNormalization are the core Keras building blocks for Transformers.
- ■ Positional embeddings inject order information into the otherwise permutation-invariant Transformer.
- ■ Keras 3's backend agnosticism makes it easy to run Transformer models on JAX for TPU-scale training.

Chapter 12 From Notebook to the Real World

Model Saving, Deployment & Production

12.1 Saving and Loading Models

Keras supports multiple serialisation formats. The recommended format is the native .keras format (Keras 3), which is a ZIP archive containing the model architecture, weights, and compilation config. The legacy SavedModel format (used by TensorFlow Serving) and HDF5 (.h5) are also supported.

```
# Save complete model (architecture + weights + compile config)
model.save("my_model.keras")            # Recommended Keras 3 format
model.save("my_model/")                 # TensorFlow SavedModel format

# Load
loaded_model = keras.models.load_model("my_model.keras")
loaded_model.predict(x_test)

# Save/load weights only (useful during fine-tuning experiments)
model.save_weights("weights.weights.h5")
model.load_weights("weights.weights.h5")

# Export for inference — strips training-specific layers
model.export("inference_model/")        # TensorFlow SavedModel
```

12.2 TensorFlow Lite for Mobile & Edge

```
import tensorflow as tf

# Convert Keras model to TFLite
converter = tf.lite.TFLiteConverter.from_keras_model(model)

# Optional: quantise to INT8 (reduces model size 4x, speeds up inference)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_data_gen   # Calibration data

tflite_model = converter.convert()
with open("model.tflite", "wb") as f:
    f.write(tflite_model)

# Run inference with TFLite interpreter
interpreter = tf.lite.Interpreter(model_path="model.tflite")
interpreter.allocate_tensors()
input_details  = interpreter.get_input_details()
output_details = interpreter.get_output_details()
interpreter.set_tensor(input_details[0]["index"], input_data)
interpreter.invoke()
output = interpreter.get_tensor(output_details[0]["index"])
```

12.3 Production Best Practices

Reproducibility: Always set random seeds: keras.utils.set_random_seed(42). Log hyperparameters and dataset versions with tools like MLflow or Weights & Biases.

Mixed Precision Training: keras.mixed_precision.set_global_policy("mixed_float16") halves GPU memory usage and accelerates training on NVIDIA Ampere GPUs with minimal accuracy loss.

tf.data for Scalable Pipelines: Replace raw NumPy arrays with tf.data.Dataset for large datasets. Use .cache(), .prefetch(AUTOTUNE), and .map() with parallel calls.

Model Versioning: Treat model artifacts like code: use version control, semantic versioning (v1.2.0), and A/B testing before full rollout.

Monitoring in Production: Track data drift (input distribution shifts), model drift (output distribution shifts), and latency. Alert when accuracy drops below a threshold.

Explainability: For image models, use Grad-CAM to visualise which regions the model attends to. For tabular, use SHAP values. Explainability builds trust and surfaces biases.

12.4 Training Monitoring — Loss & Accuracy

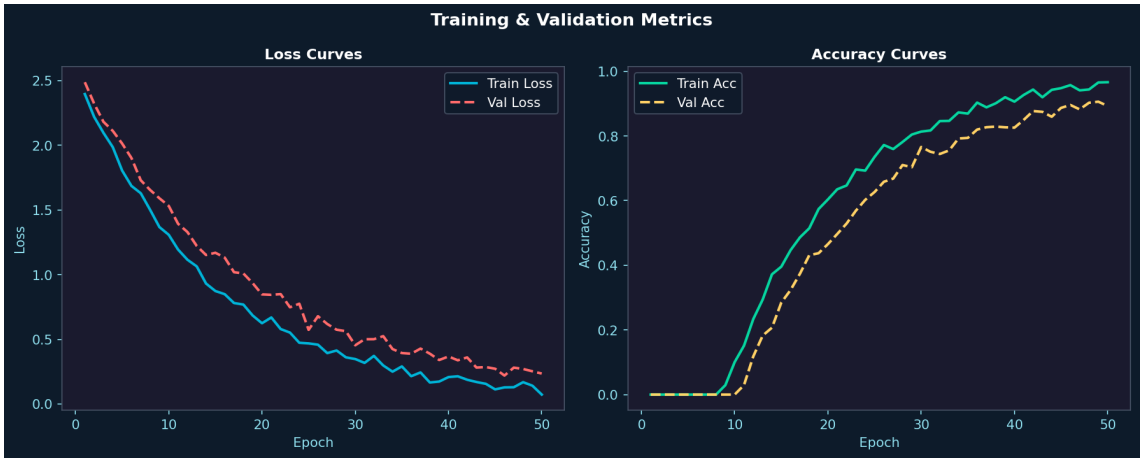

Figure 12.1 — Healthy training curves: loss decreasing and accuracy increasing with minimal train/val gap.

■ Key Takeaways

- ■ Always save models in .keras format for portability; use SavedModel for TensorFlow Serving deployment.
- ■ TFLite enables deployment on Android, iOS, microcontrollers, and edge devices with INT8 quantisation.
- ■ Mixed precision (float16) can double throughput on modern GPUs with negligible accuracy loss.
- ■ tf.data pipelines with .prefetch() are essential for keeping the GPU fed during training.
- ■ Monitor production models for data drift and accuracy degradation — models degrade silently.

■ Explainability tools (Grad-CAM, SHAP) are not optional — they are essential for responsible AI deployment.

About This Guide

This comprehensive study guide covers Machine Learning with Keras from first principles to production deployment. It is intended for students, practitioners, and researchers who want a thorough, practical, and beautifully structured reference for the Keras framework — "The Friendly Architect" of Neural Networks.

Topics covered span twelve chapters: ML fundamentals, environment setup, tensors and layers, the three model-building APIs (Sequential, Functional, Subclassing), training and optimisation, activation functions and regularisation, CNNs for images, RNNs/LSTMs/GRUs for sequences, transfer learning, callbacks, advanced architectures (autoencoders, GANs, Transformers), and production deployment via TFLite and TensorFlow Serving.

Recommended Next Steps

- Keras Official Documentation: keras.io — always check for the latest API changes.
- Deep Learning with Python (François Chollet) — the definitive Keras book by its creator.
- fast.ai Practical Deep Learning — hands-on course built atop PyTorch/Keras concepts.
- Kaggle Competitions — the best way to apply Keras skills to real-world messy data.
- Papers With Code (paperswithcode.com) — track state-of-the-art results and implementations.
- TensorFlow / JAX documentation — for backend-specific optimisation and deployment guides.

End of Guide — Happy Learning! ■