CHAPTER 12

Custom Models and Training with TensorFlow

Chapter 12 shows how to go **below tf.keras** and build custom components and training logic using **TensorFlow's low-level API**, automatic differentiation, and tf.function graphs.

TensorFlow core and tensors

The chapter starts with a quick tour of TensorFlow as a NumPy-like numerical library with GPU/TPU kernels, distributed execution, autodiff, and JIT-style graph optimization via tf.function. Core concepts are:

- **Tensors** (tf.Tensor): multidimensional arrays with a shape and dtype, interoperable with NumPy, created via tf.constant, manipulated with math ops (tf.add, tf.square, tf.matmul, etc.).
- **Variables** (tf.Variable): mutable tensors used for weights and state, updated via assign, assign_add, etc., and typically created by Keras layers or custom code.
- Additional structures: sparse tensors, ragged tensors, string tensors, sets, and tensor arrays, each with dedicated submodules (tf.sparse, tf.ragged, tf.strings).

TensorFlow enforces explicit **dtypes** (no silent casts) and favors 32-bit floats for speed and memory, requiring tf.cast when mixing types.

Customizing Keras: losses, metrics, layers, models

The chapter then focuses on **custom Keras components**:

- **Custom loss functions**:
    - As simple Python callables taking (y_true, y_pred) (plus optional sample_weight) and returning a scalar tensor, or
    - As subclasses of keras.losses.Loss when you need configuration/state or want them to be serializable and reusable in SavedModels.
- **Custom metrics**: similarly defined as functions or as subclasses of keras.metrics.Metric implementing update_state, result, and reset_states for metrics that accumulate over batches (e.g., mean IoU, F1).
- **Custom activations, initializers, regularizers, constraints**: implemented as callables or classes and passed into layers via activation=, kernel_initializer=, kernel_regularizer=, kernel_constraint=, etc.; serialization support is needed to save/load models containing them.
- **Custom layers**: subclass keras.layers.Layer, define sublayers in __init__ or build, implement call(self, inputs, training=None); optionally override get_config for serialization.

    - Examples include a layer with a skip connection (residual block) and a Gaussian noise layer that adds noise only when training=True.
- **Custom models**: subclass keras.Model, create layers in __init__/build, implement call to wire them together, enabling arbitrary architectures with loops, branches, and shared submodules.
    - The example custom model stacks a Dense layer, a custom ResidualBlock used multiple times, and an output layer, illustrating complex graphs beyond Sequential/Functional models.

The chapter also shows **losses based on model internals** using add_loss (e.g., an auxiliary reconstruction head to regularize a regression model) and internal metrics using add_metric.

Automatic differentiation and gradient control

Using **autodiff** via tf.GradientTape, any differentiable computation can be differentiated:

- You watch variables inside a with tf.GradientTape() block, compute a scalar loss, then call tape.gradient(loss, [var1, var2, ...]) to get gradients and apply them with an optimizer.
- Tapes can be persistent for multiple gradient calls, and nested tapes allow higher-order derivatives; tape.stop_recording() or tf.stop_gradient can block parts from contributing to gradients.

The chapter also covers **custom gradients** using @tf.custom_gradient, where you define both the forward computation and a custom backward function to ensure numerical stability or implement nonstandard gradient flows.

Custom training loops

For maximum control (e.g., multiple optimizers, nonstandard updates, gradient transformations), you can write your own **training loop**:

- Iterate over tf.data.Dataset batches inside for epoch and for X_batch, y_batch in dataset:
    - Use GradientTape to compute loss and gradients.
    - Optionally modify gradients (e.g., manual clipping) and call optimizer.apply_gradients(zip(grads, variables)).
    - Update metrics via metric.update_state(...) and print progress.
- Handle training=True when calling the model to ensure layers like Dropout/BatchNorm behave correctly, and apply any weight **constraints** after optimizer steps.

This pattern allows implementing research-style algorithms or papers that require, for example, distinct optimizers per subnet or extra regularization steps that Keras's fit does not support directly.

tf.function, graphs, and AutoGraph

Finally, the chapter shows how tf.function converts Python functions into fast, portable **TensorFlow graph functions**:

- Decorating a function with @tf.function or calling tf.function(f) causes TensorFlow to trace it, build a computation graph, and optimize it; subsequent calls reuse the graph for matching input shapes/dtypes.
- **AutoGraph** rewrites Python control flow (for, while, if) over tensors into graph ops (tf.while_loop, tf.cond), enabling dynamic behaviors in graphs.
- Graph polymorphism: new graphs are generated for new input signatures (shapes/dtypes), but Python scalar arguments cause separate graphs per distinct value, so scalars should represent hyperparameters with few possible values.

Keras automatically wraps most custom components in TF Functions, so performance benefits are usually obtained without explicitly using tf.function, unless you disable it via dynamic=True or run_eagerly=True for easier debugging.