Chapter 12: Custom Models and Training with TensorFlow

- A Quick Tour of TensorFlow
    - Similar to NumPy but with GPU support
    - Supports distributed computing
    - Just-in-time compiler allowing optimal computations for speed and memory use
        - Extracts computation graph, optimizing, and then running in parallel
    - Can export computation graphs to train TF model in one environment and run in another (Java on Android
    - Implement Autodiff and provides other optimizers
    - Runs on major Oss and iOS and Android, and on JavaScript
- Using TensorFlow like NumPy
    - Tensor and Operations
        - Tensor is like a numpy ndarray: it is a multidimensional array, but can also hold scalars
        - Create with tf.constant([1,2,3],[4,5,6])
        - Has shape and data type, indexing works too
        - Basic math: .add(), .multiply(), .exp()
        - Operations like np: .reshape(), .squeeze(), .tile()
        - Some different: .reduce_mean(), reduce_sum(), reduce_max()
        - Reasons for difference: other names, tf.tranpsose() creates new array that is the transpose
        - Keras has own low-level API in keras.backend. If want to port coded to other keras implementations should use these functions
    - Tensors and NumPy
        - Numpy uses 64-bit precision, TF uses 32 by default
    - Type Conversions
        - TF does not perform type conversions automatically
            - Can hurt performance
            - If try operation on tensors with incompatible types, will get exception
    - Variables
        - Value tf.Tensor is immutable
        - Use tf.Variable()
            - Plays nicely with numpy
            - Can be modified in place using assign()
            - scatter_nd_update updates values at different indices
    - Other Data Structures
        - Sparse tensors, tensor arrays, ragged tensors
        - String tensors, Sets, Queues
        - tf.string is atomic: length does not appear in shape
- Customizing Models and Training Algorithms
    - Custom Loss Functions

        - Noisy data, MSE penalizes too much, MAE training takes long time to converge
        - Use Huber loss. Build custom loss function
        - Call function in model.compile()
    - Saving and Loading Models That Contain Custom Components
        - Use dictionary to map names to objects
        - Can crate a custom function that configures loss function for a threshold
        - But when save model, threshold won't be saved
            - Solve by creating subclass of keras.losses.Loss class and then implement get_config()
            - Method call() takes labels and predictions, and computes losses
            - Method get_config() returns a dictionary mapping each hyperparameter name to value. Calls parent class's get_config()
        - When load model map class name to class in custom_objects = {"Class_function_name": class_function}
    - Custom Activation Functions, Initializers, Regularizers, and Constraints
        - Can customize most keras functions
        - If function has hyperparameters that need to be saved with model, need to subclass appropriate such as .Regularizer, .Constraint, .Initializer, or .Layer
        - Must implement the call() method for losses, layers, activations, and models
        - Must implement __call__() method for regularizers, initializers, and constraints
    - Custom Metrics
        - Losses and metrics are not the same
            - Losses must be differentiable where evaluated by Gradient Descent
                - Gradients should not be 0 everywhere
                - OK if not interpretable by humans
            - Metrics are used to evaluate the model and must be interpretable
                - Can be non-differentiable and have 0 gradients everywhere
        - Defining a custom metric function same as custom loss function
        - Each batch Keras computes metric and keeps track of mean since beginning of epoch
            - Sometimes want streaming metric across all batches, true positive rate example
            - Can create a class to do this using add_weight()
    - Custom Layers
        - Build architecture that contains exotic layer or want to build repetitive architecture so build one layer of many sublayers that gets repeated
        - Create a simple layer with no weights: keras.layers.Lambda()
        - Building custom stateful layer need to subclass keras.layers.Layer
            - Method build() creates layer's variables by calling add_weight()
            - Must call parent's build method—super().build() to tell Keras that layer is built (self.built=True)

        - To create layer with multiple inputs (e.g., Concatenate) argument to call() method should be tuple containing all inputs
        - To create layer with multiple outputs, call() method should return list of outputs
        - If layer gives more than one output, need to use Functional or Subclassing API, Sequential only accepts layers with one input and one output
        - Need a layer with different behavior during training and testing, add training argument to call() method
    - Custom Models
        - Subclass keras.Model class , create layers and variables in constructor, implement call() method
    - Losses and Metrics Based on Model Internals
        - Define losses based on other parts of your model, such as weights or activations of hidden layers
        - To define custom loss based on model internals compute it based on any part of model want, then pass result to add_loss() method
        - Reconstruction loss: auxiliary output on top of upper hidden layer, loss associated with it is mean squared difference between reconstruction and inputs
            - Adding reconstruction loss to main loss encourages model to preserve as much info as possible through hidden layers
        - There is an issue
    - Computing Gradients Using Autodiff
        - Calling partial derivatives analytically intractable for large NNs → use autodiff: tf.gradientTape()
        - If need to call gradient() method twice, add persistent=True
        - Tape only tracks operations involving variables, so if use tf.constant() returns None
        - Can force tape to record every operation using tape.watch
        - Gradient tape used to compute gradients of single value (usually loss) wrt to set of values model parameters
            - Reverse mode autodiff needs only one forward and one reverse pass to get all gradients at once
        - If try to compute gradients of vector ,TF will compute gradients of vector's sum
            - To get individual gradients call jacobian() method
        - When want to sop gradients from backpropagating through some part of NN use tf.stop_gradient()
        - Sometimes get numerical issues when computing gradients due to floating point precision errors.
            - If know analytical solution can tell TF to use stable version by decorating with @tf.custom_gradient(
    - Custom Training Loops
        - Cases where fit() not flexible enough need custom training loop

        - E.g., if have wide and deep path, and use different optimizer for each → need custom loop
    - If want to apply transformation to gradients, do so before calling apply_gradients() method
    - If add weight constraint to model (e.g., kernel_constraint() or bias_constraint(), update training loop to apply after apply_gradients()
    - If have layers that behave differently during training and testing, e.g., BatchNormalization() or Dropout(), call training=True
- TensorFlow Functions and Graphs
    - Intro
        - TF optimizes computation graph, pruning unused nodes, simplifying expressions
        - TF functions will run faster than Original Python
        - When right a custom function and use it in Keras model, Keras automatically converts to TF function
        - If call TF function many times with different numerical python values, slows down program and need to delete function. Save python values for hyperparameters and those arguments that will have few unique values
    - Autograph and Tracing
        - To generate graphs TF analyzes Python functions source code and control flow statements and then replaces with TF operations
    - TF Function Rules
        - Converting python function that performs TF ops into TF function usually only requires one to decorate with @tf.function or let Keras take care of it
        - Rules to follow:
            - If call external library (e.g. Numpy) call will not be part of graph, graphs only include TF constructs so need to use TF functions → tf.reduce-sum() instead of np.sum()
            - TF function that calls random number using numpy only generated when function traced, to generate new random number on every call use tf.random
            - If function has side effects, like logging or updating Python counter, side effects might not occur every time call TF function → only occur during function tracing
            - Can wrap python code in tf.py_function() but not optimal because TF will not be able to do any graph optimization. Reduces portability as will only run on platforms with Python and right libraries are installed.
            - If function calls TF variable or stateful object, must do so on first call and only then, else get an exception
                - Preferable to create variables outside TF function (build() method)
                - Want to assign new value to variable, call assign() instead of = operator

- Python source code for function needs to be available to TF—not in shell or .pyc files—otherwise will fail
- TF only captures for loops that iterate over tensor or dataset. Use “for I in tf.range(x)” otherwise won’t be captured in graph.
    - May not want it to be captured as may want for loop to build the graph to create each layer of the network