Custom Models and Training with TensorFlow

TensorFlow

- TensorFlow (TF) is a powerful library for computation, especially large-scale machine learning projects.
- Many operations have multiple implementations called kernels.
    - Each kernel is dedicated to a specific device type, such as CPUs, GPUs, and TPUs (tensor processing units).
    - GPUs significantly speed up computations by splitting them into smaller chunks and running them in parallel.
    - TPUs are even faster than GPUs since they were built specifically for deep learning operations.
- TF has a very detailed architecture.
    -

Operation	Code
High-level deep learning API	tf.keras
Low-level deep learning API	tf.nn
Math	tf.math, tf.linalg, tf.signal, tf.random, tf.bitwise
Autodiff	tf.GradientTape
I/O preprocessing	tf.audio, tf.data, tf.image, tf.io, tf.queue
Visualization with TensorBoard	tf.summary
Deployment and optimization	tf.distribute, tf.saved_model, tf.autograph, tf.graph_util, tf.lite, tf.quantization, tf.tpu, tf.xla
Special data structures	tf.lookup, tf.nest, tf.ragged, tf.sets, tf.sparse, tf.strings
Misc	tf.experimental, tf.config

- TF can run on Windows, Linux, macOS, and mobile devices (using TensorFlow Lite) for iOS and Android.

Using TensorFlow like NumPy

- TF's API revolves around tensors, which flow from operation to operation.
  - Tensors are similar to NumPy's ndarrays: they are multidimensional arrays that can hold scalars.
  - We can create a tensor using **tf.constant()**. Tensors have a shape and data type.
  - Indexing works the same way as in NumPy.
  - We can perform basic mathematical operations using **tf.add()**, **tf.multiply()**, **tf.square()**, **tf.exp()**, **tf.sqrt()**, etc.
    - Some functions have unique names. TF uses **tf.reduce_mean()**, **tf.reduce_sum()**, and **tf.reduce_max()** instead of mean(), sum(), and max().
- Tensors have good compatibility with NumPy arrays.
  - We can apply TF operations to NumPy arrays and NumPy operations to tensors.
  - It is important to note that NumPy arrays have 64-bit precision while tensors have 32-bit. We must initialize NumPy arrays with *dtype=tf.float32* to use operations on tensors and arrays safely.
  - Also, we cannot perform operations on tensor constants with different types (i.e., adding an integer and a float).
  - We can explicitly cast an integer to a float (and vice versa) by using **tf.cast()**.
  - These tensor values are all immutable, meaning we cannot use regular tensors to implement weights in a neural network.
- We can use a **tf.Variable**, which acts like a tensor, and we can modify them using their **assign()** method. Direct assignment does not work.
- TF supports many other data structures like sparse tensors (**tf.SparseTensor**), tensor arrays (**tf.TensorArray**), ragged tensors (**tf.RaggedTensor**), string tensors (**tf.string** and **tf.strings**), sets (**tf.sets**), and queues (**tf.queue**).

Customizing Models and Training Algorithms

- We can create custom models, loss functions, optimizers, regularizers, and layers.

Custom Loss Function

- Sometimes, we may want to modify or create new loss functions. We can create functions in Python that define these loss functions.
- When we compile a model, we can apply our new loss function by specifying the *loss* argument.

Saving and Loading Models that Contain Custom Components

- Saving a model with a custom loss function works fine.
- However, when we load the model, we must provide a dictionary that maps the function name to the actual function.
  - We can do this by specifying the *custom_objects* argument when loading the model.

    - If the function has any parameters, we must specify those in the *custom_objects* parameter.
- We can also create a subclass of **tf.keras.losses.Loss()** and implement its **get_config()** method to create the mapping for us.
    - We would have to use the **super()** method so that the subclass inherits the parameters and methods of the parent class.

Custom Activation Functions, Initializers, Regularizers, and Constraints

- Like losses, we can create functions that create custom activation functions, initializers, regularizers, and constraints.
- When we create a new layer, we can use those functions as the parameters.
- If we want to create subclasses, we must use the parent classes **tf.keras.regularizers.Regularizer**, **tf.keras.constraints.Constraint**, **tf.keras.initializers.Initializer**, or **tf.keras.layers.Layer**.

Custom Metrics

- While losses (e.g., cross-entropy) are used by gradient descent to train a model, metrics (e.g., accuracy) are used to evaluate it.
- Despite this distinction, we can use a loss function as a metric.
    - When compiling the model, we can specify the *metrics* parameter with the custom loss function.
- In some cases, the loss function may not be an effective metric.
    - In these cases, we can use the **tf.keras.metrics.Precision()** class to keep track of the number of true and false positives.
    - When we create the Precision object, we can use it as a function, passing the labels and predictions.
    - We can call the **result()** method to get the metric's current value.
- If we want to create a subclass for a custom metric, we must use the **tf.keras.metrics.Metric()** class as the parent.
    - We will have to redefine its **update_step()**, result(), and get_config() methods.

Custom Layers

- We may want to create a custom layer object if TF does not support a default implementation, or we may want to create a function that builds a block of repetitive layers, in which case, we want to treat the block as a single layer.
    - For layers with no weights (like Flatten and ReLU), we can write a function and wrap it in a **tf.keras.layers.Lambda** layer.
    - If we want to create a class, we must use **tf.keras.layers.Layer** as a parent class.
        - Then, we just need to redefine its **build()**, **call()**, and get_config() methods.
        - For some layers, such as Dropout or BatchNormalization, we must ass a *training* argument to the call() method.

Custom Models

- To create a custom model, we can simply subclass the **tf.keras.Model** class.
    - We can build any model we want, even one that contains loops or skips connections.
    - We must define how the layers are created and redefine the call() method.
    - Keras automatically detects if any attribute contains trackable objects (e.g., layers).
    - If it helps, we can also use the **get_layer()** method which returns any of the model's layers by name or index.

Computing Gradients using Autodiff

- A <u>gradient tape</u> is used to compute the gradients of a single value (usually the loss) with regard to a set of values (usually the model parameters).
- TF's **GradientTape()** class automatically records every operation that involves a variable. This facilitates computing gradients.
    - Neural networks typically contain tens of thousands of parameters. We can measure how much a function's outputs change when we tweak one of its parameters.
    - Using GradientTape() makes this easier since we don't have to call the function numerous times to check its outputs.
        - We must pass the function's parameters into the tape's **gradient()** method.
    - We can access the gradient values using the tape's *gradients* variable.
    - The tape is automatically erased after we call its gradient() method. To change this behavior, we can set its *persistent* argument to true.
    - Sometimes, we may want to stop the gradient from backpropagating through the neural network. We can use the **tf.stop_gradient()** function for this.
    - In some cases, the gradients can compute infinity with a large parameter value.
        - Some functions, like softplus, have a second form to avoid this problem.
        - To tell TF which equation to use for the gradients instead of autodiff, we must use the **@tf.custom_gradients** decorates when defining the function.

TensorFlow Functions and Graphs

- We can convert a Python function to a TF function.
    - We use **tf.function()** to convert the Python function into a TF function. We pass the Python function into the TF function.
    - The TF function can be used the same way as the original function except that it returns the result as a tensor.
    - We can also use the **@tf.function** decorator when defining a function.
    - We can access the original function using the **python_function()** function.

AutoGraph and Tracing

- TF analyzes Python's source code to capture all the control flow statements. This first step is called AutoGraph.
- Then, TF returns an upgraded version of the function where all the control flow statements are replaced by the appropriate TF operations. This part is called tracing.
    - For example, **tf.while_loop()** replaces loops, and **tf.cond()** replaces if statements.
- To view the function's source code, we can call the **tf.autograph.to_code()** function.

TF Function Rules

- Keras has rules when using a TF function.
    - Any external libraries, including NumPy, will only run during tracing and will not be part of the graph.
        - We should use TF's functions instead of Python's built-in functions (e.g., tf.sort() instead of sorted()).
    - If the function creates a TF variable, it must do so in the first call.
        - It is preferable to create variables outside of the function or use TF's assign() method.
    - TF only captures for loops that iterate over a tensor or **tf.data.Dataset**.
        - In this case, we should use *for i in tf.range(x)* instead of *for i in range(x)*.