**FSDS MAY BATCH 2022(DL Assignment -4)**

**Submitted by: Shubham Tiwari.**

Q1:How would you describe TensorFlow in a short sentence? What are its main features? Can you name other popular Deep Learning libraries?

Ans: TensorFlow is an open-source library for dataflow and differentiable programming across a range of tasks. Its main features include support for deep learning, efficient execution on a variety of platforms, and the ability to train models on distributed systems. Other popular deep learning libraries include PyTorch, Caffe, and Theano.

Q2: Is TensorFlow a drop-in replacement for NumPy? What are the main differences between the two?

Ans**: TensorFlow is not a drop-in replacement for NumPy. Both TensorFlow and NumPy are powerful libraries for numerical computations, but they are used for different purposes.**

NumPy is a library for the Python programming language that provides support for large, multi-dimensional arrays and matrices of numerical data, as well as a large collection of mathematical functions to operate on these arrays. It is mainly used for numerical computations, such as linear algebra, Fourier transform, and random number generation.

TensorFlow, on the other hand, is a library for machine learning and deep learning. It provides a wide range of tools for training and deploying machine learning models, including support for neural networks, as well as an execution engine for efficient computation of mathematical operations on large arrays and matrices. TensorFlow also allows for distributed computing, which makes it possible to train models on large data sets and take advantage of powerful hardware such as GPUs and TPUs.

In summary, while both TensorFlow and NumPy can be used for numerical computations, TensorFlow is more suited for machine learning and deep learning tasks, while NumPy is more suited for general numerical computations.

Q3: Do you get the same result with tf.range(10) and tf.constant(np.arange(10))?

Ans: **No, the result of tf.range(10) and tf.constant(np.arange(10)) are not the same.**

tf.range(10) creates a tensorflow range object with values from 0 to 9, while np.arange(10) creates a numpy array with values from 0 to 9.

tf.constant(np.arange(10)) creates a tensorflow constant tensor with values from 0 to 9 taken from numpy array.

Therefore, even though the values in the tensor are the same, the type of the object is different in both cases. We can use .numpy() method to convert a tensorflow tensor to numpy array, and .numpy() or .tolist() to convert a numpy array to list.

Q4: Can you name six other data structures available in TensorFlow, beyond regular tensors?

Ans: Here are six other data structures available in TensorFlow beyond regular tensors:

1. **SparseTensor:** A sparse tensor is a tensor with a small number of non-zero elements compared to the total number of elements. TensorFlow provides a SparseTensor class to represent sparse tensors.
2. **RaggedTensor:** A ragged tensor is a tensor with variable-length sub-lists. TensorFlow provides a RaggedTensor class to represent ragged tensors.
3. **Dataset:** A Dataset is a collection of elements, where each element consists of one or more components. It is used to represent input data for a TensorFlow model.
4. **TensorArray:** A TensorArray is a data structure that allows to build dynamic computation graphs. It can be useful when the size of a tensor is not known at graph construction time.
5. **Variables:** A Variable is a special kind of tensor that is used to store and update the parameters of a model during training.
6. **Queue:** A Queue is a data structure that can be used to store and retrieve elements in a first-in, first-out (FIFO) order. It can be useful for implementing multi-threading and parallelism in TensorFlow.

Q5: A custom loss function can be defined by writing a function or by subclassing the keras.losses.Loss class. When would you use each option?

Ans: A custom loss function can be defined by writing a function or by subclassing the keras.losses.Loss class. The choice of which option to use depends on the specific requirements of the problem and the desired level of flexibility.

If the custom loss function is a simple mathematical expression that can be defined using basic operations, it may be more efficient to write it as a standalone function. This can be passed to the model during compilation and used during training.

On the other hand, if the custom loss function requires more complex logic or additional methods, it may be more appropriate to define it by subclassing the keras.losses.Loss class. This allows for the creation of a fully-fledged class with properties and methods that can be used to define the loss function, and it allows the reusability of the loss function in multiple models. Also it will have more flexibility and can be used to add any additional arguments for the loss function that a simple function wouldn't have.

In summary, if the custom loss function is simple, writing a standalone function is sufficient, but if it requires more complexity, subclassing the keras.losses.Loss class is the recommended approach.

Q6: Similarly, a custom metric can be defined in a function or a subclass of keras.metrics.Metric.When would you use each option?

Ans: A custom metric function should be used when we want to create a simple metric that can be easily reused across different models. This option is useful when the logic of the metric is relatively simple and can be implemented in a single function. For example, if we want to create a metric that calculates the mean squared error, you could create a custom metric function.

A custom metric subclass should be used when we want to create a more complex metric that requires additional functionality or state. This option is useful when the logic of the metric is more complex and requires multiple methods or when you want to implement more complex functionality like streaming data, or creating a metric from multiple other metrics.

For example, if we want to create a metric that keeps track of running mean and variance of the data over time, we would create a custom metric subclass.

Q7: When should you create a custom layer versus a custom model?

Ans: A custom layer should be used when we want to reuse the layer in multiple models, or when we want to share the layer across different parts of a single model. For example, if we want to use the same convolutional layer in multiple models, you would create a custom layer.

A custom model should be used when we want to create a new, unique architecture that is not simply a combination of existing layers. For example, if we want to create a model that includes a novel combination of layers, or if we want to add new functionality to an existing model, we would create a custom model.

Q8: What are some use cases that require writing your own custom training loop?

Ans: There are several use cases where writing your own custom training loop may be necessary:

1. **Custom Metrics:** If you want to use a custom metric that is not built into a framework, you'll need to write your own training loop to calculate and track it.
2. **Specialized Training:** Some models may require specialized training techniques that are not built into the framework, for example, a particular type of reinforcement learning algorithm or a GAN training technique.
3. **Distributed Training:** Some models may require distributed training, where the data is split across multiple GPUs or machines. In this case, a custom training loop will be necessary to coordinate the training process and aggregate the gradients.
4. **Handling Non-Standard Data:** If the data you are working with is not in a standard format, you'll need to write a custom training loop to handle preprocessing and loading of the data.
5. **Debugging and Experimentation:** Custom training loops allow you to have more control over the training process, which can be useful for debugging and experimenting with different training techniques.
6. **Resource constraints:** When working with memory-intensive tasks, a custom training loop allows you to optimize memory usage and avoid out of memory errors.

Q9: Can custom Keras components contain arbitrary Python code, or must they be convertible to TF Functions?

Ans: Custom Keras components can contain arbitrary Python code, but they must also be convertible to TensorFlow functions in order to be used during training and evaluation.

When defining a custom component, such as a custom layer, loss function, or metric, it is necessary to also define a TensorFlow function that implements the same computation as the custom component. This function will be used during training and evaluation, and it must be able to take tensors as input and return tensors as output.

The conversion of the custom component to a TensorFlow function is done automatically by Keras during the model compilation process. The Keras component is wrapped with the **tf.function** decorator, which converts the Python code into TensorFlow graph operations.

In summary, custom Keras components can contain arbitrary Python code, but they must also be convertible to TensorFlow functions in order to be used during training and evaluation.

Q10: What are the main rules to respect if you want a function to be convertible to a TF Function?

Ans: If you want a function to be convertible to a TensorFlow function, it is important to follow these rules:

1. The function should only use TensorFlow operations and should not have any side-effects.
2. The function should not use any Python state, such as global variables or mutable objects.
3. The function should not use any control flow operations, such as if and while statements.
4. The function should not use any non-TensorFlow libraries.
5. The function should be decorated with @tf.function to indicate that it should be converted to a TensorFlow function.
6. The function should be called inside a **tf.Graph** context.

By following these rules, the function can be safely converted to a TensorFlow function and executed efficiently on GPUs or TPUs.

Q11: When would you need to create a dynamic Keras model? How do you do that? Why not make all your models dynamic?

Ans: A dynamic Keras model is a model where the architecture can change during runtime, based on the input data or other factors. There are a few situations where creating a dynamic model can be useful:

* When the architecture of the model needs to change depending on the input data. For example, a model that needs to handle variable-length sequences, where the number of timesteps in the input can vary.
* When we want to use a model as a building block in a larger ensemble model.
* When we need to create a model that adapts to new data, like online learning.

We can create a dynamic Keras model by using the functional API to build the model, rather than using the Sequential API. The functional API allows you to define the model as a directed acyclic graph of layers, where the layers are connected by calling the layers' connect method. The advantage of this approach is that we can create complex architectures with multiple inputs and outputs and easily reuse layers.

It is not always necessary to make all models dynamic as dynamic models have more complexity and also more computation time. For example, if we are working with a fixed dataset or architecture, a static model is sufficient and easier to maintain.