Conversion and Deployment

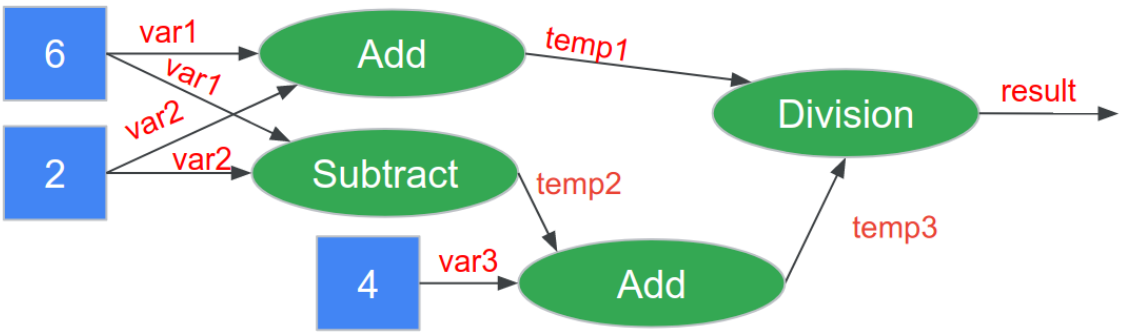
Now that we have talked about the difference between TensorFlow and TensorFlow Lite. Let's dive under the hood to understand how models are represented and how conversion is done.

Tensorflow's Computational Graph

Tensorflow represents models as a **computational graph**. To better understand what that means, let's use a simple example of the program shown below:

```
var1 = 6
var2 = 2
temp1 = var1 + var2
temp2 = var1 - var2
var3 = 4
temp3 = temp2 + var3
res = temp1/temp3
```

This program can be represented by the following computational graph which specifies the relationships between **constants**, **variables** (often represented as **tensors**), and **ops**:



The benefit of using such a computational graph is twofold:

1. Optimizations such as parallelism can be applied to the graph after it is created and managed by a backend compiled (e.g., Tensorflow) instead of needing to be specified by the user. For example, in the case of distributed training, the graph structure can be used for efficient partitions and learnable weights can be averaged easily after each distributed run. As another example, it enables efficient auto-differentiation for faster training by keeping all of the variables in a structured format.
2. Portability is increased as the graph can be designed in a high level language (like we do using Python in this course) and then converted into an efficient low level representation using C/C++ or even assembly language by the backend compiler.

Both of these reasons become increasingly important as we increase the size of our models --- even for TinyML many models have thousands of weights and biases!

If you'd like to learn more about this we suggest checkout this great article which was used for inspiration for this section of the reading.

Tensorflow Checkpoints

As you train your model you will notice that Tensorflow will save a series of .ckpt files. These files save a snapshot of the trained model at different points during the training process. While it may seem like a lot of files at first (e.g., you will notice later that that /train directory will get quite full when you train your custom keyword spotting model in the next section), you do not need to be able to understand their contents. **The most important thing to keep in mind about the checkpoint files is that you can rebuild a snapshot of your model.** However, at a high level:

- .ckpt-meta files contain the metagraph, i.e. the structure of your computation graph, without the values of the variables
- .ckpt-datacontainsthevaluesforalltheweightsandbiases,withoutthestructure.
- .ckpt-index contains the mappings between the -data and -meta files to enable the model to be restored. As such, to restore a model in python, you'll usually need all three files.

If you'd like to take Google's hands on crash course on all things checkpoint files check out this Colab!

Freezing a Model

You may also notice some .pb files appear as well. These are in the Google "Proto Buffers" format. A Proto Buffer represents a complete model (both the metadata and the weights/biases). A ".pb" file is the output from "freezing" a model --- a complete representation of the model in a ready to run format!

Google differentiates between checkpoint files and frozen model files (which it refers to as "saved models") as follows:

- Frozen/saved models are designed to be deployed into production environments from that format
- Checkpoint files are designed to be used to jumpstart future training

If you'd like to learn more about freezing/saving models you can check out this Colab made by Google.

Optimizing a Model (Converting to TensorFlow Lite)

The Converter can be used to turn Tensorflow models into quantized TensorFlow Lite models. We won't go into detail about that process here but suffice it to say that that process often reduces the size of the model by a factor of 4 by quantizing from Float32 to Int8.

TensorFlow Lite models are stored in the FlatBuffer file format as .tflite files.

The primary difference between FlatBuffers and ProtoBuffers is that ProtoBuffers are designed to be converted into a secondary representation before use (requiring memory to be allocated and copied at runtime), while FlatBuffers are designed to be per-object allocated into a compressed usable format. As such FlatBuffers offer less flexibility in what they can represent but are an order of magnitude more compact in terms of the amount of code required to create them, load them (i.e., do not require any dynamic memory allocation) and runt them, all of which are vitally important for TinyML!