

The vote is over, but the fight for net neutrality isn't. Show your support for a free and open internet.[Learn more](#)


tensorflow / tensorflow

Tree: fc49f43817


tensorflow / tensorflow / tools / graph_transforms / README.md

Find file

Copy path

 tensorflow-gardener Delete trailing whitespace

191825e on 27 Nov 2017

8 contributors 

1094 lines (894 sloc) 46.7 KB

Graph Transform Tool

Table of Contents

- [Introduction](#)
- [Using the Graph Transform Tool](#)
- [Inspecting Graphs](#)
- [Common Use Cases](#)
 - [Optimizing for Deployment](#)
 - [Fixing Missing Kernel Errors on Mobile](#)
 - [Shrinking File Size](#)
 - [Eight-bit Calculations](#)
- [Transform Reference](#)
 - [add_default_attributes](#)
 - [backport_concatv2](#)
 - [flatten_atrous_conv](#)
 - [fold_batch_norms](#)
 - [fold_constants](#)
 - [fold_old_batch_norms](#)
 - [freeze_requantization_ranges](#)
 - [fuse_convolutions](#)
 - [insert_logging](#)
 - [merge_duplicate_nodes](#)
 - [obfuscate_names](#)
 - [quantize_nodes](#)
 - [quantize_weights](#)
 - [remove_attribute](#)
 - [remove_device](#)
 - [remove_nodes](#)
 - [rename_attribute](#)
 - [rename_op](#)
 - [round_weights](#)
 - [sparsify_gather](#)
 - [set_device](#)
 - [sort_by_execution_order](#)

- [strip_unused_nodes](#)
- [Writing Your Own Transforms](#)
 - [Transform Functions](#)
 - [Pattern Syntax](#)
 - [ReplaceMatchingOpTypes](#)
 - [Parameters](#)
 - [Function Libraries](#)
 - [Registering](#)

Introduction

When you have finished training a model and want to deploy it in production, you'll often want to modify it to better run in its final environment. For example if you're targeting a phone you might want to shrink the file size by quantizing the weights, or optimize away batch normalization or other training-only features. The Graph Transform framework offers a suite of tools for modifying computational graphs, and a framework to make it easy to write your own modifications.

This guide is structured into three main parts, first giving some tutorials on how to perform common tasks, second a reference covering all of the different transformations that are included, together with the options that apply to them, and third a guide to creating your own transforms.

Using the Graph Transform Tool

The Graph Transform tool is designed to work on models that are saved as GraphDef files, usually in a binary protobuf format. This is the low-level definition of a TensorFlow computational graph, including a list of nodes and the input and output connections between them. If you're using a Python API to train your model, this will usually be saved out in the same directory as your checkpoints, and usually has a '.pb' suffix.

If you want to work with the values of your trained parameters, for example to quantize weights, you'll need to run [tensorflow/python/tools/freeze_graph.py](#) to convert the checkpoint values into embedded constants within the graph file itself.

You call the Graph Transform tool itself like this:

```
bazel build tensorflow/tools/graph_transforms:transform_graph
bazel-bin/tensorflow/tools/graph_transforms/transform_graph \
--in_graph=tensorflow_inception_graph.pb \
--out_graph=optimized_inception_graph.pb \
--inputs='Mul:0' \
--outputs='softmax:0' \
--transforms='
strip_unused_nodes(type=float, shape="1,299,299,3")
remove_nodes(op=Identity, op=CheckNumerics)
fold_old_batch_norms
'
```

The arguments here are specifying where to read the graph from, where to write the transformed version to, what the input and output layers are, and what transforms to modify the graph with. The transforms are given as a list of names, and can each have arguments themselves. These transforms define the pipeline of modifications that are applied in order to produce the output. Sometimes you need some transforms to happen before others, and the ordering within the list lets you specify which happen first. Note that the optimization `remove_nodes(op=Identity, op=CheckNumerics)` will break the model with control flow operations, such as `tf.cond`, `tf.map_fn`, and `tf.while`.

Inspecting Graphs

Many of the transforms that the tool supports need to know what the input and output layers of the model are. The best source for these is the model training process, where for a classifier the inputs will be the nodes that receive the data from the training set, and the output will be the predictions. If you're unsure, the [summarize_graph](#) tool can inspect the model and provide guesses about likely input and output nodes, as well as other information that's useful for debugging. Here's an example of how to use it on the [Inception V3 graph](#):

```
bazel build tensorflow/tools/graph_transforms:summarize_graph
bazel-bin/tensorflow/tools/graph_transforms/summarize_graph --in_graph=tensorflow_inception_graph.pb
```

Common Use Cases

This section has small guides for some of the most frequently-used transformation pipelines, aimed at users who want to quickly accomplish one of these tasks. A lot of them will use the Inception V3 model for their examples, which can be downloaded from <http://download.tensorflow.org/models/image/imagenet/inception-2015-12-05.tgz>.

Optimizing for Deployment

If you've finished training your model and want to deploy it on a server or a mobile device, you'll want it to run as fast as possible, and with as few non-essential dependencies as you can. This recipe removes all of the nodes that aren't called during inference, shrinks expressions that are always constant into single nodes, and optimizes away some multiply operations used during batch normalization by pre-multiplying the weights for convolutions.

```
bazel build tensorflow/tools/graph_transforms:transform_graph
bazel-bin/tensorflow/tools/graph_transforms/transform_graph \
--in_graph=tensorflow_inception_graph.pb \
--out_graph=optimized_inception_graph.pb \
--inputs='Mul' \
--outputs='softmax' \
--transforms='
  strip_unused_nodes(type=float, shape="1,299,299,3")
  remove_nodes(op=Identity, op=CheckNumerics)
  fold_constants(ignore_errors=true)
  fold_batch_norms
  fold_old_batch_norms'
```

The batch norm folding is included twice because there are two different flavors of batch normalization used in TensorFlow. The older version was implemented with a single BatchNormWithGlobalNormalization op, but it was deprecated in favor of a more recent approach using individual ops to implement the same computation. The two transforms are in there so that both styles are recognized and optimized.

Fixing Missing Kernel Errors on Mobile

The mobile version of TensorFlow is focused on inference, and so by default the list of supported ops (defined in [tensorflow/core/kernels/BUILD:android_extended_ops](#) for Bazel and [tensorflow/contrib/makefile/tf_op_files.txt](#) for make builds) doesn't include a lot that are training related. This can cause No OpKernel was registered to support Op errors when a GraphDef is loaded, even if the op isn't going to be executed.

If you see this error and it's an op that you do actually want to run on mobile, then you'll need to make local modifications to the build files to include the right .cc file that defines it. In a lot of cases the op is just a vestigial remnant from the training process though, and if that's true then you can run the [strip_unused_nodes](#), specifying the inputs and outputs of your inference usage, to remove those unnecessary nodes:

```
bazel build tensorflow/tools/graph_transforms:transform_graph
bazel-bin/tensorflow/tools/graph_transforms/transform_graph \
--in_graph=tensorflow_inception_graph.pb \
--out_graph=optimized_inception_graph.pb \
--inputs='Mul' \
```

```
--outputs='softmax' \  
--transforms='  
  strip_unused_nodes(type=float, shape="1,299,299,3")  
  fold_constants(ignore_errors=true)  
  fold_batch_norms  
  fold_old_batch_norms'
```

Shrinking File Size

If you're looking to deploy your model as part of a mobile app, then keeping the download size as small as possible is important. For most TensorFlow models, the largest contributors to the file size are the weights passed in to convolutional and fully-connected layers, so anything that can reduce the storage size for those is very useful. Luckily most neural networks are resistant to noise, so it's possible to change the representation of those weights in a lossy way without losing very much accuracy overall.

On both iOS and Android app packages are compressed before download, so the simplest way to reduce the bandwidth your users need to receive your app is to provide raw data that compresses more easily. By default the weights are stored as floating-point values, and even tiny differences between numbers result in very different bit patterns, and so these don't compress very well. If you round the weights so that nearby numbers are stored as exactly the same values, the resulting bit stream has a lot more repetition and so compresses down a lot more effectively. To try this technique on your model, run the [round_weights](#) transform.

```
bazel build tensorflow/tools/graph_transforms:transform_graph  
bazel-bin/tensorflow/tools/graph_transforms/transform_graph \  
--in_graph=tensorflow_inception_graph.pb \  
--out_graph=optimized_inception_graph.pb \  
--inputs='Mul' \  
--outputs='softmax' \  
--transforms='  
  strip_unused_nodes(type=float, shape="1,299,299,3")  
  fold_constants(ignore_errors=true)  
  fold_batch_norms  
  fold_old_batch_norms  
  round_weights(num_steps=256)'
```

You should see that the `optimized_inception_graph.pb` output file is the same size as the input, but if you run `zip` on it to compress it, it's almost 70% smaller than if you zip the original! The nice thing about this transform is that it doesn't change the structure of the graph at all, so it's running exactly the same operations and should have the same latency and memory usage as before. You can adjust the `num_steps` parameter to control how many values each weight buffer is rounded to, so lower numbers will increase the compression at the cost of accuracy.

As a further step, you can store the weights into eight-bit values directly. Here's the recipe for that:

```
bazel build tensorflow/tools/graph_transforms:transform_graph  
bazel-bin/tensorflow/tools/graph_transforms/transform_graph \  
--in_graph=tensorflow_inception_graph.pb \  
--out_graph=optimized_inception_graph.pb \  
--inputs='Mul' \  
--outputs='softmax' \  
--transforms='  
  strip_unused_nodes(type=float, shape="1,299,299,3")  
  fold_constants(ignore_errors=true)  
  fold_batch_norms  
  fold_old_batch_norms  
  quantize_weights'
```

You should see that the size of the output graph is about a quarter of the original. The downside to this approach compared to `round_weights` is that extra decompression ops are inserted to convert the eight-bit values back into floating point, but optimizations in TensorFlow's runtime should ensure these results are cached and so you shouldn't see the graph run any more slowly.

So far we've been concentrating on weights because those generally take up the most space. If you have a graph with a lot of small nodes in it, the names of those nodes can start to take up a noticeable amount of space too. To shrink those down, you can run the [obfuscate_names](#) transform, which replaces all the names (except for inputs and outputs) with short, cryptic but unique ids:

```
bazel build tensorflow/tools/graph_transforms:transform_graph
bazel-bin/tensorflow/tools/graph_transforms/transform_graph \
--in_graph=tensorflow_inception_graph.pb \
--out_graph=optimized_inception_graph.pb \
--inputs='Mul:0' \
--outputs='softmax:0' \
--transforms='
  obfuscate_names'
```

Eight-bit Calculations

For some platforms it's very helpful to be able to do as many calculations as possible in eight-bit, rather than floating-point. The support for this in TensorFlow is still experimental and evolving, but you can convert models into quantized form using the graph transform tool:

```
bazel build tensorflow/tools/graph_transforms:transform_graph
bazel-bin/tensorflow/tools/graph_transforms/transform_graph \
--in_graph=tensorflow_inception_graph.pb \
--out_graph=optimized_inception_graph.pb \
--inputs='Mul' \
--outputs='softmax' \
--transforms='
  add_default_attributes
  strip_unused_nodes(type=float, shape="1,299,299,3")
  remove_nodes(op=Identity, op=CheckNumerics)
  fold_constants(ignore_errors=true)
  fold_batch_norms
  fold_old_batch_norms
  quantize_weights
  quantize_nodes
  strip_unused_nodes
  sort_by_execution_order'
```

This process converts all the operations in the graph that have eight-bit quantized equivalents, and leaves the rest in floating point. Only a subset of ops are supported, and on many platforms the quantized code may actually be slower than the float equivalents, but this is a way of increasing performance substantially when all the circumstances are right.

A full guide to optimizing for quantization is beyond the scope of this guide, but one thing that can help is using the `FakeQuantWithMinMaxVars` op after `Conv2D` or similar operations during training. This trains the min/max variables that control the range used for quantization, so that the range doesn't have to be calculated dynamically by `RequantizationRange` during inference.

Transform Reference

The `--transforms` string is parsed as a series of transform names, each of which can have multiple named arguments inside parentheses. Arguments are separated by commas, and double-quotes (") can be used to hold argument values if they themselves contain commas (for example shape definitions).

The `--inputs` and `--outputs` are shared across all transforms, since it's common to need to know what the ingoing and outgoing nodes in the graph are. You should make sure you set these correctly before calling the graph transform tool, and if you're in doubt check with the model's author, or use the [summarize_graph](#) tool to examine likely inputs and outputs.

All transforms can be passed the `ignore_errors` flag, with the value set to either true or false. By default any errors that happen within a transform will abort the whole process, but if you enable this then an error will just be logged and the transform skipped. This is especially useful for optional transforms where version errors or other unimportant problems may trigger an error.

add_default_attributes

Args: None

When attributes are added to ops in new versions of TensorFlow, they often have defaults to ensure backwards compatible behavior with their original versions. These defaults usually get added when the graph is loaded by the runtime, but if your model is going to be processed outside of the main TensorFlow framework it can be useful to run this update process as a transform. This process finds any op attributes that are defined in the current TensorFlow list of ops but not within the saved model, and sets them to the defined default for that attribute.

backport_concatv2

Args: None

If you have a GraphDef file that has been produced by a newer version of the TensorFlow framework and includes ConcatV2, and you want to run it on an older version that only supports Concat, this transform will take care of converting those newer ops to the equivalent older form.

flatten_atrous_conv

Args: None

Prerequisites: [fold_constants](#)

This transform flattens atrous convolution, corresponding to a sequence of SpaceToBatchND-Conv2D-BatchToSpaceND operations, converting it to a regular Conv2D op with upsampled filters. This transform should only be used in order to run graphs having atrous convolution on platforms that do not yet natively support SpaceToBatchND and BatchToSpaceND operations. You will need to make sure you run [fold_constants](#) after this transform. If applicable, you should run this transform before [fold_batch_norms](#).

fold_batch_norms

Args: None

Prerequisites: [fold_constants](#)

This transform tries to optimize away the Mul that's introduced after a Conv2D (or a MatMul) when batch normalization has been used during training. It scans the graph for any channel-wise multiplies immediately after convolutions, and multiplies the convolution's (or matrix multiplication's) weights with the Mul instead so this can be omitted at inference time. You'll need to make sure you run [fold_constants](#) first, since the pattern can only be spotted if the normal complex expression that's produced by training for the Mul input is collapsed down into a simple constant.

fold_constants

Args:

- `clear_output_shapes`: Clears tensor shape information saved as attributes. Some older graphs contains out-of-date information and may cause import errors. Defaults to true.

Prerequisites: None

Looks for any sub-graphs within the model that always evaluate to constant expressions, and replaces them with those constants. This optimization is always executed at run-time after the graph is loaded, so running it offline first won't help latency, but it can simplify the graph and so make further processing easier. It's often useful to call this with `fold_constants(ignore_errors=true)` to continue on past transient errors, since this is just an optimization phase.

fold_old_batch_norms

Args: None

Prerequisites: None

In the early days of TensorFlow, batch normalization was implemented using a single monolithic `BatchNormWithGlobalNormalization` op. In modern versions, adding batch normalization from Python will give you a series of smaller math ops instead, to achieve the same effect without special-purpose code. If you have a graph that uses the older-style, this transform will recognize and optimize those ops for inference, in the same way that the [fold_batch_norms](#) transform does for the new approach.

freeze_requantization_ranges

Args:

- `min_max_log_file`: Path to a log file containing ranges for ops.
- `min_percentile`: Percentage cutoff to use to calculate an overall min. Defaults to 5.
- `max_percentile`: Percentage cutoff to use to calculate an overall max. Defaults to 5.

Quantized operations like convolution or matrix multiplies take their inputs as 8-bit, but produce 32-bit results. To do further operations on these, they need to be converted back down to the lower depth. To make the most of those eight bits, you need to scale the thirty-two bits of original data down using a scale that matches the range that's actually being used.

Because that range information isn't stored in the original graph, the [quantization process](#) inserts `RequantizationRange` ops before each conversion from 32 to 8 bits. This op looks at the 32-bit output and calculates the current min and max every time it's run.

This isn't incredibly time-consuming, but it is extra work that's nice to avoid if possible. One way of optimizing that away is replacing those `RequantizationRange` ops with a pair of `Const` nodes holding known min/max values, so the scaling down can be done without having to inspect the output every time.

That's what this transform does. It's usually used in conjunction with a copy of the graph that's had [insert_logging](#) run on it to instrument it to record the min/max values to stderr. Why is logging used rather than writing to a normal file? As you'll see later, to get best results you want to collect data from a lot of runs on real data, and for mobile apps especially it's a lot easier to do this by copying log files. As an example, here's how you'd add the logging operations for a quantized version of the Inception v3 graph:

```
bazel build tensorflow/tools/graph_transforms:transform_graph
bazel-bin/tensorflow/tools/graph_transforms/transform_graph \
--in_graph=/tmp/quantized_inception.pb \
--out_graph=/tmp/logged_quantized_inception.pb \
--inputs=Mul \
--outputs=softmax \
--transforms='
insert_logging(op=RequantizationRange, show_name=true, message="__requant_min_max:")\
'
```

Now, when you run the `/tmp/logged_quantized_inception.pb` graph, it will write out log statements that show the value of the min and max calculated by each `RequantizationRange` op. Here's an example of running `label_image` and saving the log:

```
bazel build tensorflow/examples/label_image:label_image
bazel-bin/tensorflow/examples/label_image/label_image \
--image=${HOME}/Downloads/grace_hopper.jpg \
--input_layer=Mul \
--output_layer=softmax \
--graph=/tmp/logged_quantized_inception.pb \
--labels=${HOME}/Downloads/imagenet_comp_graph_label_strings.txt \
2>/tmp/min_max_log_small.txt
```

If you look in `/tmp/min_max_log_small.txt`, you'll see a lot of lines like this:

```
I0108 21:45:42.261883    1972 logging_ops.cc:79] ;conv/Conv2D/eightbit
/requant_range_print__:_requant_min_max:[-20.887871][22.274715]
```

This is a simple way of serializing the name of the RequantizationRange op and its min/max values every time it's run. It's a file like this that you pass into the transform as the `min_max_log_file` argument. The transform will attempt to extract all of the min/max values associated with ops, ignoring any irrelevant lines in the log, and replace the RequantizationRange ops with two Const nodes containing the found values.

This isn't the whole story though. The min/max values can vary a lot depending on what the particular inputs to the graph are on any given run, which means picking ranges based on just one run can lead to clipping of values and a loss of accuracy. To get better results, you need to run your network against a range of different inputs. In Inception's case, I often use a thousand different images from the training set. You can then pass the whole concatenated log from all of the runs into the transform, and it will pick ranges based on the aggregate of the values found for each RequantizationRange op.

To ensure that outliers don't increase the range too much, and so decrease the accuracy by putting too many bits into rare extreme values, the `min_percentile` and `max_percentile` arguments control how the overall min and max are chosen. At their default values of 5, this means that the lowest 5% of the minimum values will be discarded, taking the minimum of the remainder, and the equivalent for the maximum.

fuse_convolutions

Args: None

Prerequisites: None

For graphs that use ResizeBilinear or MirrorPad ops before convolutions (e.g. to scale up in the later stages of an image style transfer model), it can improve memory usage and latency to combine the spatial transformations with the convolution's im2col patch generation. This transform looks out for that particular pattern of ops and replaces them with a fused version that combines the resizing and padding with the convolution.

insert_logging

Args:

- `op`: Insert a Print after every occurrence of this op type. Can be repeated to cover multiple types. If not present, all op types will be instrumented.
- `prefix`: Insert a Print after every node whose name starts with this value. Can be repeated to cover multiple nodes. If not present, all node names will be matched.
- `show_op`: If true, the op type will be prepended to all log messages.
- `show_name`: If true, the node's name will be prepended to all log messages.
- `message`: Arbitrary text to log before the values.
- `first_n`: How many times to print before suppressing. Defaults to -1, which means never stop.
- `summarize`: How long numerical results can be before they're truncated. Defaults to 1024.

The Print operator writes strings to stderr when it's run inside a graph, and prints out the numerical results of the node that it's reading from. This can be very useful when you're debugging and want to follow particular internal values while a graph is running. This transform allows you to insert those ops at particular points in the graph, and customize the message that's displayed. It's also used in conjunction with the [freeze_requantization_ranges](#) transform to output information that it needs.

merge_duplicate_nodes

Args: None

Prerequisites: None

If there are Const nodes with the same types and contents, or nodes with the same inputs and attributes, this transform will merge them together. It can be useful when you want to cut down the number of nodes in a graph that has a lot of redundancy (e.g. this transform is always run as part of [quantize_nodes](#) since the processing there can introduce duplicates of constants that are used in the quantize/dequantize process).

obfuscate_names

Args: None

Prerequisites: None

Replaces all nodes' names with short generated ids, other than the inputs and outputs. This also updates all references within the graph so that the structure is preserved. This can be useful if you want to shrink the file size, or if you want to make it harder to understand the architecture of your model before releasing it.

quantize_nodes

Args:

- `input_min`: The lowest float value for any quantized placeholder inputs.
- `input_max`: The highest float value for any quantized placeholder inputs. If both `input_min` and `input_max` are set, then any float placeholders in the graph will be replaced with quantized versions, and consts will be created to pass the range to subsequent operations.
- `fallback_min`: The lowest float value to use for requantizing activation layers.
- `fallback_max`: The highest float value to use for requantizing activation layers. If both `fallback_min` and `fallback_max` are set, then instead of using `RequantizationRange` ops to figure out the useful range dynamically when converting the 32-bit output of ops like `QuantizedConv2D` and `QuantizedBiasAdd`, hardwired consts with these values will be used instead. This can help performance, if you know the range of your activation layers ahead of time.

Prerequisites: [quantize_weights](#)

Replaces any calculation nodes with their eight-bit equivalents (if available), and adds in conversion layers to allow remaining float operations to interoperate. This is one of the most complex transforms, and involves multiple passes and a lot of rewriting. It's also still an active area of research, so results may vary depending on the platform and operations you're using in your model. You should run `quantize_weights` first to ensure your Const ops are in eight-bit form.

quantize_weights

Args:

- `minimum_size`: Tensors with fewer elements than this won't be quantized (defaults to 1024)

Prerequisites: None

Converts any large (more than `minimum_size`) float Const op into an eight-bit equivalent, followed by a float conversion op so that the result is usable by subsequent nodes. This is mostly useful for [shrinking file sizes](#), but also helps with the more advanced [quantize_nodes](#) transform. Even though there are no prerequisites, it is advisable to run [fold_batch_norms](#) or [fold_old_batch_norms](#), because rounding variances down to zero may cause significant loss of precision.

remove_attribute

Args:

- `attribute_name`: Name of the attribute you want to remove.
- `op_name`: Optional name of a single op to restrict the removal to.

Prerequisites: None

Deletes the given attribute from either all nodes, or just the one specified in `op_name`. This can be a dangerous transform since it's easy to leave your graph in an invalid state if you remove a required attribute. It can be useful in special circumstances though.

remove_device

Args: None

Prerequisites: None

All ops can have a hardware device specified. This can be a problem when you're loading a graph on a different system than the model was trained on, since some specified devices may not be available. In order to work with graphs like these, you can run this transform to wipe the slate clean and delete the device specifier from all ops.

remove_nodes

Args:

- `op`: The name of the op you want to remove. Can be repeated to remove multiple ops.

Prerequisites: None

This is a potentially dangerous transform that looks for single-input, single-output ops with the given names, removes them from the graph, and rewires all inputs that use to pull from them to pull from the preceding node instead. This is most useful for getting rid of ops like `CheckNumerics` that are useful during training but just complicate the graph and increase latency during inference. It's dangerous because it's possible that removing some ops may change the output of your graph, so make sure you check the overall accuracy after using this.

rename_attribute

Args:

- `old_attribute_name`: Current name of the attribute you want to rename.
- `new_attribute_name`: Name that you want the attribute to have now.
- `op_name`: If this is set, only change attributes for a given op type, otherwise apply to all nodes with attribute names that match.

Prerequisites: None

Changes the name of the given attribute. This is often useful for upgrading graph files as op definitions change over versions, since the renaming is often enough to deal with minor changes.

rename_op

Args:

- `old_op_name`: Current name of the operation.
- `new_op_name`: Name to change to.

Prerequisites: None

Finds all ops with the given name, and changes them to the new one. This can be useful for version upgrading if the changes between ops are minor apart from the name.

round_weights

Args:

- `num_steps`: How many unique values to use in each buffer.

Prerequisites: None

Rounds all float values in large Const ops (more than 15 elements) to the given number of steps. The unique values are chosen per buffer by linearly allocating between the largest and smallest values present. This is useful when you'll be deploying on mobile, and you want a model that will compress effectively. See [shrinking file size](#) for more details. Even though there are no prerequisites, it is advisable to run [fold_batch_norms](#) or [fold_old_batch_norms](#), because rounding variances down to zero may cause significant loss of precision.

sparsify_gather

Args: None

Prerequisites: None

Transform 'Gather' op to a sparsified version where 'params' input of 'Gather' is replaced from a dense 'Const' to a 'HashTable'. 'Gather' op itself is replaced by a hashtable lookup. This is mostly useful for reducing sparse TF.learn linear model memory footprint.

set_device

Args:

- device: What device to assign to ops.
- if_default: If this is true, only assign to ops with empty existing devices.

Updates nodes to use the specified device. A device is a way to tell the code that executes the graph which piece of hardware it should run particular nodes on. The right assignment to use may change between training and deployment, so this transform (and [remove_device](#)) provide a way of updating the placement. If the `is_default` parameter is set, then only ops that don't have a device assigned already will be updated. This is mostly useful for preprocessing of graphs for other stages that expect all ops to have an explicit device assigned.

sort_by_execution_order

Args: None

Prerequisites: None

Arranges the nodes in the GraphDef in topological order, so that the inputs of any given node are always earlier than the node itself. This is especially useful when you're targeting a minimal inference engine, since you can just execute the nodes in the given order knowing that the inputs will be computed before they're needed.

strip_unused_nodes

Args:

- type: Default type for any new Placeholder nodes generated, for example int32, float, quint8.
- shape: Default shape for any new Placeholder nodes generated, as comma-separated dimensions. For example `shape="1,299,299,3"`. The double quotes are important, since otherwise the commas will be taken as argument separators.
- name: Identifier for the placeholder arguments.
- type_for_name: What type to use for the previously-given name.
- shape_for_name: What shape to use for the previously-given name.

Prerequisites: None

Removes all nodes not used in calculated the layers given in `--outputs`, fed by `--inputs`. This is often useful for removing training-only nodes like save-and-restore or summary ops. It's also handy for solving the [missing kernel errors problem](#) when there are decode or other ops you don't need in the inference path.

The biggest complication is that it sometimes has to create new Placeholder ops, so there are options to control their characteristics. This will happen if you bypass a DecodeJpeg op by specifying an input layer deeper in the network, for example, so you can pass in a raw image array instead of an encoded string as an input. The decode op will be removed, together with the Placeholder that fed it, but a new Placeholder is needed for the input layer you specify. The type and shape arguments let you control the attributes of any new Placeholders that are created. Plain `type` and `shape` set global defaults, but if you have different inputs with varying characteristics, you'll need to pass in a list of arguments where the preceding name specifies what layer each applies to. For example, if you had two inputs `in1` and `in2`, you could call

```
strip_unused_nodes(name=in1, type_for_name=int32, shape_for_name="2,3", name=in2, type_for_name=float,
shape_for_name="1,10,10,3") .
```

Writing Your Own Transforms

The Graph Transform Tool is designed to make it as easy as possible to create your own optimization, modification, and pre-processing transforms. At their heart, all of the transforms take in a valid `GraphDef`, make some changes, and output a new `GraphDef`. Each `GraphDef` is just a list of `NodeDefs`, each defining one node in the graph and its connections. You can find more information on the format at [this guide to TensorFlow model files](#), but for a simple example take a look at [tensorflow/tools/graph_transforms/rename_op.cc](#), which implements the `rename_op` transform:

```
Status RenameOp(const GraphDef& input_graph_def,
                const TransformFuncContext& context,
                GraphDef* output_graph_def) {
  if (!context.params.count("old_op_name") ||
      (context.params.at("old_op_name").size() != 1) ||
      !context.params.count("new_op_name") ||
      (context.params.at("new_op_name").size() != 1)) {
    return errors::InvalidArgument(
      "remove_nodes expects exactly one 'old_op_name' and 'new_op_name' "
      "argument, e.g. rename_op(old_op_name=Mul, new_op_name=Multiply)");
  }

  const string old_op_name = context.params.at("old_op_name")[0];
  const string new_op_name = context.params.at("new_op_name")[0];
  output_graph_def->Clear();
  for (const NodeDef& node : input_graph_def.node()) {
    NodeDef* new_node = output_graph_def->mutable_node()->Add();
    new_node->CopyFrom(node);
    if (node.op() == old_op_name) {
      new_node->set_op(new_op_name);
    }
  }

  return Status::OK();
}

REGISTER_GRAPH_TRANSFORM("rename_op", RenameOp);
```

The heart of this transform is the loop through the `input_graph_def`'s nodes. We go through each op, add a new one to the output, copy the original's contents, and then change the op over if it matches the parameters. There's a standard set of parameters for every transform, so they all take in a `GraphDef` and context, and write out into a new `GraphDef`. The registration macro at the bottom lets the tool know what function to call when it finds the `rename_op` string in a transforms list.

Transform Functions

The standard signature that all transform functions have is defined as `TransformFunc`, which takes in an input `GraphDef`, a `TransformFuncContext` containing environment information, writes to an output `GraphDef`, and returns a `Status` indicating whether the transform succeeded.

The `TransformFuncContext` has a list of the inputs and outputs for the graph, and the [parameter arguments](#) that were passed into the transform by the user.

If you write a function that matches this signature, and [register it](#), the graph transform tool will take care of calling it.

Pattern Syntax

The `rename_op` example only needs to look at a single node at a time, but one of the most common needs is to modify small sub-graphs within a model. To make this easy, the Graph Transform Tool provides the `OpTypePattern` syntax. This is a simple and compact way to specify patterns of nodes that you want to look for. The format is:

```
OP_TYPE_PATTERN ::= "{" OP " ", " INPUTS "}"
INPUTS ::= OP_TYPE_PATTERN
```

The `OP` field can either contain a single `"*"`, which means match any op type, one op type (for example `"Const"`), or a set of op types separated by `|` symbols (for example `"Conv2D|MatMul|BiasAdd"`). General regex patterns are not supported, just these special cases.

You can think of these patterns as very limited regular expressions designed to pick out sub-trees in graphs. They are deliberately very constrained to the kind of things we commonly find ourselves needing to do, to make creating and debugging as straightforward as possible.

For example, if you want all `Conv2D` nodes that have a constant as their second input, you would set up a pattern like this, using C++ initializer lists to populate the structure:

```
OpTypePattern conv_pattern({"Conv2D", {"*"}, {"Const"}});
```

It can be easier to visualize these initializers using indentation to show the tree structure more clearly:

```
OpTypePattern conv_pattern({
  "Conv2D",
  {
    {"*"},
    {"Const"}
  }
});
```

In plain English this is saying, a `Conv2D` op with two inputs, the first of which is any op type, and the second is a `Const` op.

Here's a much more complex example, from the [quantize_nodes](#) transform:

```
{"QuantizeV2",
 {
   {"Dequantize"},
   {"Min",
    {
      {"Reshape",
       {
         {"Dequantize"},
         {"Const"},
       }
      },
    },
   {"Const"},
 }
 },
 {"Max",
 {
   {"Reshape",
    {
      {"Dequantize"},
      {"Const"},
    }
   }
 }
 }
```

```

    },
    {"Const"},
  }
},
}
}

```

This is looking for QuantizeV2 nodes, with three inputs, the first of which is a Dequantize, the second is a Min that ultimately pulls from a Dequantize, and the third is a Max which does the same. Assuming we know the Dequantize ops are pulling from the same eight-bit buffer, the end result of this sub-graph is a no-op, since it's just turning the eight-bit buffer into float, and then immediately converting it back to eight-bits, so if we look for this pattern and remove it we can optimize the graph without changing the result.

ReplaceMatchingOpTypes

It's very common to want to find all occurrences of a particular sub-graph in a model, and replace them all with a different sub-graph that keeps the same local input and output connections. For example with [fuse_convolutions](#), we needed to find all Conv2D ops that read their inputs from BilinearResizes, and replace those combinations with a single FusedResizeAndPadConv2D op, but without affecting other ops.

To make that sort of transformation easy, we created the `ReplaceMatchingOpTypes` helper. This takes in a graph, an `OpTypePattern` defining the sub-graph to look for, and a callback function to run for every occurrence it finds. The job of this callback function is to look at the `NodeMatch` that contains information about the current sub-graph, and return a new sub-graph in the `new_nodes` list that will be used to replace the old sub-graph.

You can see how it's used in practice in the [fuse_convolutions](#) code:

```

TF_RETURN_IF_ERROR(ReplaceMatchingOpTypes(
  input_graph_def, // clang-format off
  {"Conv2D",
   {
     {"ResizeBilinear"},
     {"*"}
   }
  }, // clang-format on
  [](const NodeMatch& match, const std::set<string>& input_nodes,
    const std::set<string>& output_nodes,
    std::vector<NodeDef>* new_nodes) {
    // Find all the nodes we expect in the subgraph.
    const NodeDef& conv_node = match.node;
    const NodeDef& resize_node = match.inputs[0].node;
    const NodeDef& weights_node = match.inputs[1].node;

    // We'll be reusing the old weights.
    new_nodes->push_back(weights_node);

    // Create a 'no-op' mirror padding node that has no effect.
    NodeDef pad_dims_node;
    pad_dims_node.set_op("Const");
    pad_dims_node.set_name(conv_node.name() + "_dummy_paddings");
    SetNodeAttr("dtype", DT_INT32, &pad_dims_node);
    SetNodeTensorAttr<int32>("value", {4, 2}, {0, 0, 0, 0, 0, 0, 0, 0},
      &pad_dims_node);
    new_nodes->push_back(pad_dims_node);

    // Set up the new fused version of the convolution op.
    NodeDef fused_conv;
    fused_conv.set_op("FusedResizeAndPadConv2D");
    fused_conv.set_name(match.node.name());
    AddNodeInput(resize_node.input(0), &fused_conv);
    AddNodeInput(resize_node.input(1), &fused_conv);
    AddNodeInput(pad_dims_node.name(), &fused_conv);
    AddNodeInput(conv_node.input(1), &fused_conv);
  }
);

```

```

CopyNodeAttr(resize_node, "align_corners", "resize_align_corners",
              &fused_conv);
SetNodeAttr("mode", "REFLECT", &fused_conv);
CopyNodeAttr(conv_node, "T", "T", &fused_conv);
CopyNodeAttr(conv_node, "padding", "padding", &fused_conv);
CopyNodeAttr(conv_node, "strides", "strides", &fused_conv);
new_nodes->push_back(fused_conv);

return Status::OK();
},
{}, &replaced_graph_def));

```

Here you can see we define the pattern to look for, and in the callback function use information from each of the nodes in the old sub-graph to create a new fused node. We also copy over the old weights input node so that isn't lost.

There are a few things to know about the `ReplaceMatchingOpTypes` function:

- All of the nodes in any matching sub-graphs are removed from the new graph created by the function. If any of them are needed, it's the callback function's responsibility to add them back in. There's a `CopyOriginalMatch` convenience call that will copy over all of the original nodes if you decide you don't actually want to modify a particular sub-graph.
- It is assumed that the same nodes will never appear in more than one matched sub-graph. This is to ensure that sub-trees are only replaced once, but it may mean that some sub-graphs aren't spotted if they overlap with earlier matches.
- The calling framework tries to ensure that the graph remains sane, by looking at the `new_nodes` that are returned and making sure that no nodes which are needed as inputs by nodes outside the sub-graph are removed. These important nodes are listed in the `output_nodes` argument that's passed into each replacement function call. You can disable this checking by setting `allow_inconsistencies` to true in the options, but otherwise any replacements that break the graph constraints will be canceled. If you do allow inconsistencies, it's your transform's responsibility to fix them up before you return your final result. Functions like `RenameNodeInputs` can be useful if you are doing wholesale node renaming for example.

Parameters

The arguments that are in parentheses after the transform name when the tool is called are parsed and placed into the `params` member of the `TransformFuncContext` that's given to each transform. For every named argument, there's a vector of strings containing all the values that it was given, in the order they were given. These are treated a bit like command-line parameters, and it's the transform's responsibility to parse them into the data types it needs, and raise errors by returning a bad `Status` if any of them are ill-formed.

As an example, here's a hypothetical transform call:

```
some_transform(foo=a, foo=b, bar=2, bob="1,2,3")
```

Here's what the `std::map` of strings looks like in the `params` member:

```
{{"foo", {"a", "b"}}, {"bar", {"2"}}, {"bob", {"1,2,3"}}
```

The double quotes around the comma-separated argument to `bob` are important because otherwise they'll be treated as separate arguments, and the parsing will fail.

Here's an example of how `round_weights` reads its `num_steps` parameter:

```
TF_RETURN_IF_ERROR(context.GetOneInt32Parameter("num_steps", 256, &num_steps));
```

If the conversion fails or the parameter occurs more than once the helper function will raise a meaningful error through the status result of the transform. If the parameter isn't specified at all then the default will be used.

Function Libraries

A newer feature of TensorFlow is the ability to create libraries of functions as part of graphs. These are a bit like templates, which define macro operations in terms of smaller components, which can then be instantiated with different input and output connections inside the graph just like regular ops. Right now the graph transform tool just copies these libraries between the input and output graphs, but it's likely that more complex operations will be supported on them in the future.

Registering

The Graph Transform Tool associates names of transforms with the code to implement them using the `REGISTER_GRAPH_TRANSFORM()` macro. This takes a string and a function, and automagically registers the transform with the tool. You will need to watch out for a few things though:

- Because it's using global C++ objects in each file under the hood, the linker can sometimes strip them out and lose the registration. In Bazel you need to make sure you're linking any new transforms in as libraries, and use the `alwayslink` flag in your `cc_binary` call.
- You should be able to create your own copy of the `transform_graph` tool by linking against the `transform_graph_main_lib` library in `tensorflow/tools/graph_transforms/BUILD`. This contains all the `main()` logic to parse command line arguments and call transforms.