Large-Scale Evolution of Image Classifiers

Supplementary Material

S1. Methods Details

This section contains additional implementation details, roughly following the order in Section 3. Short code snippets illustrate the ideas. The code is not intended to run on its own and it has been highly edited for clarity.

In our implementation, each worker runs an outer loop that is responsible for selecting a pair of random individuals from the population. The individual with the highest fitness usually becomes a parent and the one with the lowest fitness is usually killed (Section 3.1). Occasionally, either of these two actions is not carried out in order to keep the population size close to a set-point:

```
def evolve_population(self):
  # Iterate indefinitely.
  while True:
    # Select two random individuals from the population.
    valid_individuals = []
    for individual in self.load_individuals():  # Only loads the IDs and states.
      if individual.state in [TRAINING, ALIVE]:
        valid_individuals.append(individual)
    individual_pair = random.sample(valid_individuals, 2)

    for individual in individual_pair:
      # Sync changes from other workers from file-system. Loads everything else.
      individual.update_if_necessary()

      # Ensure the individual is fully trained.
      if individual.state == TRAINING:
        self._train(individual)

    # Select by fitness (accuracy).
    individual_pair.sort(key=lambda i: i.fitness, reverse=True)
    better_individual = individual_pair[0]
    worse_individual = individual_pair[1]

    # If the population is not too small, kill the worst of the pair.
    if self._population_size() >= self._population_size_setpoint:
      self._kill_individual(worse_individual)

    # If the population is not too large, reproduce the best of the pair.
    if self._population_size() < self._population_size_setpoint:
      self._reproduce_and_train_individual(better_individual)
```

Much of the code is wrapped in try-except blocks to handle various kinds of errors. These have been removed from the code snippets for clarity. For example, the method above would be wrapped like this:

```
def evolve_population(self):
  while True:
  try:
    # Select two random individuals from the population.
    ...
  except:
    except exceptions.PopulationTooSmallException:
      self._create_new_individual()
      continue
```

```
    except exceptions.ConcurrencyException:
      # Another worker did something that interfered with the action of this worker.
      # Abandon the current task and keep going.
      continue
```

The encoding for an individual is represented by a serializable `DNA` class instance containing all information except for the trained weights (Section 3.2). For all results in this paper, this encoding is a directed, acyclic graph where edges represent convolutions and vertices represent nonlinearities. This is a sketch of the `DNA` class:

```
class DNA(object):

  def __init__(self, dna_proto):
    """Initializes the `DNA` instance from a protocol buffer.

    The `dna_proto` is a protocol buffer used to restore the DNA state from disk.
    Together with the corresponding `to_proto` method, they allow for a
    serialization-deserialization mechanism.
    """
    # Allows evolving the learning rate, i.e. exploring the space of
    # learning rate schedules.
    self.learning_rate = dna_proto.learning_rate

    self._vertices = {}  # String vertex ID to `Vertex` instance.
    for vertex_id in dna_proto.vertices:
      vertices[vertex_id] = Vertex(vertex_proto=dna_sproto.vertices[vertex_id])

    self._edges = {}  # String edge ID to `Edge` instance.
    for edge_id in dna_proto.edges:
      mutable_edges[edge_id] = Edge(edge_proto=dna_proto.edges[edge_id])

    ...

  def to_proto(self):
    """Returns this instance in protocol buffer form."""
    dna_proto = dna_pb2.DnaProto(learning_rate=self.learning_rate)

    for vertex_id, vertex in self._vertices.iteritems():
      dna_proto.vertices[vertex_id].CopyFrom(vertex.to_proto())

    for edge_id, edge in self._edges.iteritems():
      dna_proto.edges[edge_id].CopyFrom(edge.to_proto())

    ...

    return dna_proto

  def add_edge(self, dna, from_vertex_id, to_vertex_id, edge_type, edge_id):
    """Adds an edge to the DNA graph, ensuring internal consistency."""
    # `EdgeProto` defines defaults for other attributes.
    edge = Edge(EdgeProto(
        from_vertex=from_vertex_id, to_vertex=to_vertex_id, type=edge_type))
    self._edges[edge_id] = edge
    self._vertices[from_vertex_id].edges_out.add(edge_id)
    self._vertices[to_vertex].edges_in.add(edge_id)
    return edge

  # Other methods like `add_edge` to manipulate the graph structure.
  ...
```

The `DNA` holds `Vertex` and `Edge` instances. The `Vertex` class looks like this:

```
class Vertex(object):

  def __init__(self, vertex_proto):
    # Relationship to the rest of the graph.
```

```
    self.edges_in = set(vertex_proto.edges_in)  # Incoming edge IDs.
    self.edges_out = set(vertex_proto.edges_out)  # Outgoing edge IDs.

    # The type of activations.
    if vertex_proto.HasField('linear'):
      self.type = LINEAR  # Linear activations.
    elif vertex_proto.HasField('bn_relu'):
      self.type = BN_RELU  # ReLU activations with batch-normalization.
    else:
      raise NotImplementedError()

    # Some parts of the graph can be prevented from being acted upon by mutations.
    # The following boolean flags control this.
    self.inputs_mutable = vertex_proto.inputs_mutable
    self.outputs_mutable = vertex_proto.outputs_mutable
    self.properties_mutable = vertex_proto.properties_mutable

    # Each vertex represents a 2^s x 2^s x d block of nodes. s and d are positive
    # integers computed dynamically from the in-edges. s stands for "scale" so
    # that 2^x x 2^s is the spatial size of the activations. d stands for "depth",
    # the number of channels.

  def to_proto(self):
    ...
```

The `Edge` class looks like this:

```
class Edge(object):

  def __init__(self, edge_proto):
    # Relationship to the rest of the graph.
    self.from_vertex = edge_proto.from_vertex  # Source vertex ID.
    self.to_vertex = edge_proto.to_vertex  # Destination vertex ID.

    if edge_proto.HasField('conv'):
      # In this case, the edge represents a convolution.
      self.type = CONV

      # Controls the depth (i.e. number of channels) in the output, relative to the
      # input. For example if there is only one input edge with a depth of 16 channels
      # and `self._depth_factor` is 2, then this convolution will result in an output
      # depth of 32 channels. Multiple-inputs with conflicting depth must undergo
      # depth resolution first.
      self.depth_factor = edge_proto.conv.depth_factor

      # Control the shape of the convolution filters (i.e. transfer function).
      # This parameterization ensures that the filter width and height are odd
      # numbers: filter_width = 2 * filter_half_width + 1.
      self.filter_half_width = edge_proto.conv.filter_half_width
      self.filter_half_height = edge_proto.conv.filter_half_height

      # Controls the strides of the convolution. It will be 2^stride_scale.
      # Note that conflicting input scales must undergo scale resolution. This
      # controls the spatial scale of the output activations relative to the
      # spatial scale of the input activations.
      self.stride_scale = edge_proto.conv.stride_scale
    elif edge_spec.HasField('identity'):
      self.type = IDENTITY
    else:
      raise NotImplementedError()

    # In case depth or scale resolution is necessary due to conflicts in inputs,
    # These integer parameters determine which of the inputs takes precedence in
    # deciding the resolved depth or scale.
    self.depth_precedence = edge_proto.depth_precedence
```

```
    self.scale_precedence = edge_proto.scale_precedence

  def to_proto(self):
    ...
```

Mutations act on `DNA` instances. The set of mutations restricts the space explored somewhat (Section 3.2). The following are some example mutations. The `AlterLearningRateMutation` simply randomly modifies the attribute in the `DNA`:

```
class AlterLearningRateMutation(Mutation):
  """Mutation that modifies the learning rate."""

  def mutate(self, dna):
    mutated_dna = copy.deepcopy(dna)

    # Mutate the learning rate by a random factor between 0.5 and 2.0,
    # uniformly distributed in log scale.
    factor = 2**random.uniform(-1.0, 1.0)
    mutated_dna.learning_rate = dna.learning_rate * factor

    return mutated_dna
```

Many mutations modify the structure. Mutations to insert and excise vertex-edge pairs build up a main convolutional column, while mutations to add and remove edges can handle the skip connections. For example, the `AddEdgeMutation` can add a skip connection between random vertices.

```
class AddEdgeMutation(Mutation):
  """Adds a single edge to the graph."""

  def mutate(self, dna):
    # Try the candidates in random order until one has the right connectivity.
    for from_vertex_id, to_vertex_id  in self._vertex_pair_candidates(dna):
      mutated_dna = copy.deepcopy(dna)
      if (self._mutate_structure(mutated_dna, from_vertex_id, to_vertex_id)):
        return mutated_dna
    raise exceptions.MutationException()  # Try another mutation.

  def _vertex_pair_candidates(self, dna):
    """Yields connectable vertex pairs."""
    from_vertex_ids = _find_allowed_vertices(dna, self._to_regex, ...)
    if not from_vertex_ids:
      raise exceptions.MutationException()  # Try another mutation.
    random.shuffle(from_vertex_ids)

    to_vertex_ids = _find_allowed_vertices(dna, self._from_regex, ...)
    if not to_vertex_ids:
      raise exceptions.MutationException()  # Try another mutation.
    random.shuffle(to_vertex_ids)

    for to_vertex_id in to_vertex_ids:
      # Avoid back-connections.
      disallowed_from_vertex_ids, _ = topology.propagated_set(to_vertex_id)
      for from_vertex_id in from_vertex_ids:
        if from_vertex_id in disallowed_from_vertex_ids:
          continue
        # This pair does not generate a cycle, so we yield it.
        yield from_vertex_id, to_vertex_id

  def _mutate_structure(self, dna, from_vertex_id, to_vertex_id):
    """Adds the edge to the DNA instance."""
    edge_id = _random_id()
    edge_type = random.choice(self._edge_types)
    if dna.has_edge(from_vertex_id, to_vertex_id):
      return False
    else:
      new_edge = dna.add_edge(from_vertex_id, to_vertex_id, edge_type, edge_id)
```

```
      ...
      return True
```

For clarity, we omitted the details of a vertex ID targeting mechanism based on regular expressions, which is used to constrain where the additional edges are placed. This mechanism ensured the skip connections only joined points in the "main convolutional backbone" of the convnet. The precedence range is used to give the main backbone precedence over the skip connections when resolving scale and depth conflicts in the presence of multiple incoming edges to a vertex. Also omitted are details about the attributes of the edge to add.

To evaluate an individual's fitness, its `DNA` is unfolded into a TensorFlow model by the `Model` class. This describes how each `Vertex` and `Edge` should be interpreted. For example:

```
class Model(object):
  ...

  def _compute_vertex_nonlinearity(self, tensor, vertex):
    """Applies the necessary vertex operations depending on the vertex type."""
    if vertex.type == LINEAR:
      pass
    elif vertex.type == BN_RELU:
      tensor = slim.batch_norm(
          inputs=tensor, decay=0.9, center=True, scale=True,
          epsilon=self._batch_norm_epsilon,
          activation_fn=None, updates_collections=None,
          is_training=self.is_training, scope='batch_norm')
      tensor = tf.maximum(tensor, vertex.leakiness * tensor, name='relu')
    else:
      raise NotImplementedError()
    return tensor

  def _compute_edge_connection(self, tensor, edge, init_scale):
    """Applies the necessary edge connection ops depending on the edge type."""
    scale, depth = self._get_scale_and_depth(tensor)
    if edge.type == CONV:
      scale_out = scale
      depth_out = edge.depth_out(depth)
      stride = 2**edge.stride_scale
      # `init_scale` is used to normalize the initial weights in the case of
      # multiple incoming edges.
      weights_initializer = slim.variance_scaling_initializer(
          factor=2.0 * init_scale**2, uniform=False)
      weights_regularizer = slim.l2_regularizer(
          weight=self._dna.weight_decay_rate)
      tensor = slim.conv2d(
          inputs=tensor, num_outputs=depth_out,
          kernel_size=[edge.filter_width(), edge.filter_height()],
          stride=stride, weights_initializer=weights_initializer,
          weights_regularizer=weights_regularizer, biases_initializer=None,
          activation_fn=None, scope='conv')
    elif edge.type == IDENTITY:
      pass
    else:
      raise NotImplementedError()
    return tensor
```

The training and evaluation (Section 3.4) is done in a fairly standard way, similar to that in the tensorflow.org tutorials for image models. The individual's fitness is the accuracy on a held-out validation dataset, as described in the main text.

Parents are able to pass some of their learned weights to their children (Section 3.6). When a child is constructed from a parent, it inherits IDs for the different sets of trainable weights (convolution filters, batch norm shifts, etc.). These IDs are embedded in the TensorFlow variable names. When the child's weights are initialized, those that have a matching ID in the parent are inherited, provided they have the same shape:

```
graph = tf.Graph()
```

```
with graph.as_default():
  # Build the neural network using the `Model` class and the `DNA` instance.
  ...

  tf.Session.reset(self._master)
  with tf.Session(self._master, graph=graph) as sess:
    # Initialize all variables
    ...

    # Make sure we can inherit batch-norm variables properly.
    # The TF-slim batch-norm variables must be handled separately here because some
    # of them are not trainable (the moving averages).
    batch_norm_extras = [x for x in tf.all_variables() if (
        x.name.find('moving_var') != -1 or
        x.name.find('moving_mean') != -1)]

    # These are the variables that we will attempt to inherit from the parent.
    vars_to_restore = tf.trainable_variables() + batch_norm_extras

    # Copy as many of the weights as possible.
    if mutated_weights:
      assignments = []
      for var in vars_to_restore:
        stripped_name = var.name.split(':')[0]
        if stripped_name in mutated_weights:
          shape_mutated = mutated_weights[stripped_name].shape
          shape_needed = var.get_shape()
          if shape_mutated == shape_needed:
            assignments.append(var.assign(mutated_weights[stripped_name]))
      sess.run(assignments)
```

S2. FLOPs estimation

This section describes how we estimate the number of floating point operations (FLOPs) required for an entire evolution experiment. To obtain the total FLOPs, we sum the FLOPs for each individual ever constructed. An individual's FLOPs are the sum of its training and validation FLOPs. Namely, the individual FLOPs are given by $F_t N_t + F_v N_v$, where F_t is the FLOPs in one training step, N_t is the number of training steps, F_v is the FLOPs required to evaluate one validation batch of examples and N_v is the number of validation batches.

The number of training steps and the number of validation batches are known in advance and are constant throughout the experiment. F_t was obtained analytically as the sum of the FLOPs required to compute each operation executed during training (that is, each node in the TensorFlow graph). F_v was found analogously.

Below is the code snippet that computes FLOPs for the training of one individual, for example.

```
import tensorflow as tf
tfprof_logger = tf.contrib.tfprof.python.tools.tfprof.tfprof_logger


def compute_flops():
  """Compute flops for one iteration of training."""
  graph = tf.Graph()
  with graph.as_default():
    # Build model
    ...

    # Run one iteration of training and collect run metadata.
    # This metadata will be used to determine the nodes which were
    # actually executed as well as their argument shapes.
    run_meta = tf.RunMetadata()
    with tf.Session(graph=graph) as sess:
      feed_dict = {...}
      _ = sess.run(
```

```
        [train_op],
        feed_dict=feed_dict,
        run_metadata=run_meta,
        options=tf.RunOptions(trace_level=tf.RunOptions.FULL_TRACE))

    # Compute analytical FLOPs for all nodes in the graph.
    logged_ops = tfprof_logger._get_logged_ops(graph, run_meta=run_metadata)

    # Determine which nodes were executed during one training step
    # by looking at elapsed execution time of each node.
    elapsed_us_for_ops = {}
    for dev_stat in run_metadata.step_stats.dev_stats:
      for node_stat in dev_stat.node_stats:
        name = node_stat.node_name
        elapsed_us = node_stat.op_end_rel_micros - node_stat.op_start_rel_micros
        elapsed_us_for_ops[name] = elapsed_us

    # Compute FLOPs of executed nodes.
    total_flops = 0
    for op in graph.get_operations():
      name = op.name
      if elapsed_us_for_ops.get(name, 0) > 0 and name in logged_ops:
        total_flops += logged_ops[name].float_ops

    return total_flops
```

Note that we also need to declare how to compute FLOPs for each operation type present (that is, for each node type in the TensorFlow graph). We did this for the following operation types (and their gradients, where applicable):

- unary math operations: square, squre root, log, negation, element-wise inverse, softmax, L2 norm;
- binary element-wise operations: addition, subtraction, multiplication, division, minimum, maximum, power, squared difference, comparison operations;
- reduction operations: mean, sum, argmax, argmin;
- convolution, average pooling, max pooling;
- matrix multiplication.

For example, for the element-wise addition operation type:

```
from tensorflow.python.framework import graph_util
from tensorflow.python.framework import ops

@ops.RegisterStatistics("Add", "flops")
def _add_flops(graph, node):
  """Compute flops for the Add operation."""
  out_shape = graph_util.tensor_shape_from_node_def_name(graph, node.name)
  out_shape.assert_is_fully_defined()
  return ops.OpStats("flops", out_shape.num_elements())
```

S3. Escaping Local Optima Details

S3.1. Local optima and mutation rate

Entrapment at a local optimum may mean a general lack of exploration in our search algorithm. To encourage more exploration, we increased the *mutation rate* (Section 5). In more detail, we carried out experiments in which we first waited until the populations converged. Some reached higher fitnesses and others got trapped at poor local optima. At this point, we modified the algorithm slightly: instead of performing 1 mutation at each reproduction event, we performed 5 mutations. We evolved with this increased mutation rate for a while and finally we switched back to the original single-mutation version. During the 5-mutation stage, some populations escape the local optimum, as in Figure 4 (top), and none

get worse. Across populations, however, the escape was not frequent enough (8 out of 10) and took too long for us to propose this as an efficient technique to escape optima. An interesting direction for future work would be to study more elegant methods to manage the exploration vs. exploitation trade-off in large-scale neuro-evolution.

S3.2. Local optima and weight resetting

The identity mutation offers a mechanism for populations to get trapped in local optima. Some individuals may get trained more than their peers just because they happen to have undergone more identity mutations. It may, therefore, occur that a poor architecture may become more accurate than potentially better architectures that still need more training. In the extreme case, the well-trained poor architecture may become a super-fit individual and take over the population. Suspecting this scenario, we performed experiments in which we simultaneously reset all the weights in a population that had plateaued (Section 5). The simultaneous reset should put all the individuals on the same footing, so individuals that had accidentally trained more no longer have the unfair advantage. Indeed, the results matched our expectation. The populations suffer a temporary degradation in fitness immediately after the reset, as the individuals need to retrain. Later, however, the populations end up reaching higher optima (for example, Figure 4, bottom). Across 10 experiments, we find that three successive resets tend to cause improvement ($p < 0.001$). We mention this effect merely as evidence of this particular drawback of weight inheritance. In our main results, we circumvented the problem by using longer training times and larger populations. Future work may explore more efficient solutions.