tf.contrib.factorization.WALSModel

Contents

Class **WALSModel**

Defined in `tensorflow/contrib/factorization/python/ops/factorization_ops.py`.

A model for Weighted Alternating Least Squares matrix factorization.

It minimizes the following loss function over U, V: \(\|\sqrt W \odot (A - U V^T) \|*F^2 + \lambda (\|U\|_F^2 + \|V\|_F^2))\ *where, A: input matrix, W: weight matrix. Note that the (element-wise) square root of the weights is used in the objective function. U, V: row_factors and column_factors matrices, \(\lambda)\: regularization. Also we assume that W is of the following special form:* \(W{ij} = W_0 + R_i * C_j)\ if \(A_{ij} \ne 0)\, \(W_{ij} = W_0)\ otherwise. where, \(W_0)\: unobserved_weight, \(R_i)\: row_weights, \(C_j)\: col_weights.

Note that the current implementation supports two operation modes: The default mode is for the condition where row_factors and col_factors can individually fit into the memory of each worker and these will be cached. When this condition can't be met, setting use_factors_weights_cache to False allows the larger problem sizes with slight performance penalty as this will avoid creating the worker caches and instead the relevant weight and factor values are looked up from parameter servers at each step.

Loss computation: The loss can be computed efficiently by decomposing it into a sparse term and a Gramian term, see wals.md. The loss is returned by the update_{col, row}*factors(sp_input), and is normalized as follows:* , _, unregularized_loss, regularization, sum_weights = update_row_factors(sp_input) if sp_input contains the rows {A_i, i \in I}, and the input matrix A has n total rows, then the minibatch loss = unregularized_loss + regularization is \((\|\sqrt W_I \odot (A_I - U_I V^T)\|_F^2 + \lambda \|U_I\|_F^2) * n / |I| + \lambda \|V\|_F^2)\ The sum_weights tensor contains the normalized sum of weights sum(W_I) * n / |I|.

A typical usage example (pseudocode):

with tf.Graph().as_default(): # Set up the model object. model = tf.contrib.factorization.WALSModel(....)

```
# To be run only once as part of session initialization. In distributed
# training setting, this should only be run by the chief trainer and all
# other trainers should block until this is done.
model_init_op = model.initialize_op

# To be run once per worker after session is available, prior to
# the prep_gramian_op for row(column) can be run.
worker_init_op = model.worker_init

# To be run once per interation sweep before the row(column) update
# initialize ops can be run. Note that in the distributed training
# situations, this should only be run by the chief trainer. All other
# trainers need to block until this is done.
row_update_prep_gramian_op = model.row_update_prep_gramian_op
```

```
row_update_prep_gramian_op = model.row_update_prep_gramian_op
col_update_prep_gramian_op = model.col_update_prep_gramian_op

# To be run once per worker per iteration sweep. Must be run before
# any actual update ops can be run.
init_row_update_op = model.initialize_row_update_op
init_col_update_op = model.initialize_col_update_op

# Ops to upate row(column). This can either take the entire sparse tensor
# or slices of sparse tensor. For distributed trainer, each trainer
# handles just part of the matrix.
_, row_update_op, unreg_row_loss, row_reg, _ = model.update_row_factors(
     sp_input=matrix_slices_from_queue_for_worker_shard)
row_loss = unreg_row_loss + row_reg
_, col_update_op, unreg_col_loss, col_reg, _ = model.update_col_factors(
     sp_input=transposed_matrix_slices_from_queue_for_worker_shard,
     transpose_input=True)
col_loss = unreg_col_loss + col_reg

...

# model_init_op is passed to Supervisor. Chief trainer runs it. Other
# trainers wait.
sv = tf.train.Supervisor(is_chief=is_chief,
                  ...,
                  init_op=tf.group(..., model_init_op, ...), ...)
...

with sv.managed_session(...) as sess:
  # All workers/trainers run it after session becomes available.
  worker_init_op.run(session=sess)

  ...

  while i in iterations:

    # All trainers need to sync up here.
    while not_all_ready:
      wait

    # Row update sweep.
    if is_chief:
      row_update_prep_gramian_op.run(session=sess)
    else:
      wait_for_chief

    # All workers run upate initialization.
    init_row_update_op.run(session=sess)

    # Go through the matrix.
    reset_matrix_slices_queue_for_worker_shard
    while_matrix_slices:
      row_update_op.run(session=sess)

    # All trainers need to sync up here.
    while not_all_ready:
      wait

    # Column update sweep.
    if is_chief:
      col_update_prep_gramian_op.run(session=sess)
    else:
      wait_for_chief

    # All workers run upate initialization.
    init_col_update_op.run(session=sess)
```

```
# Go through the matrix.
reset_transposed_matrix_slices_queue_for_worker_shard
while_transposed_matrix_slices:
  col_update_op.run(session=sess)
```

Properties

`col_factors`

Returns a list of tensors corresponding to column factor shards.

`col_update_prep_gramian_op`

Op to form the gramian before starting col updates.

Must be run before initialize_col_update_op and should only be run by one trainer (usually the chief) when doing distributed training.

Returns:

Op to form the gramian.

`col_weights`

Returns a list of tensors corresponding to col weight shards.

`initialize_col_update_op`

Op to initialize worker state before starting column updates.

`initialize_op`

Returns an op for initializing tensorflow variables.

`initialize_row_update_op`

Op to initialize worker state before starting row updates.

`row_factors`

Returns a list of tensors corresponding to row factor shards.

`row_update_prep_gramian_op`

Op to form the gramian before starting row updates.

Must be run before initialize_row_update_op and should only be run by one trainer (usually the chief) when doing distributed training.

Returns:

Op to form the gramian.

`row_weights`

Returns a list of tensors corresponding to row weight shards.

`worker_init`

Op to initialize worker state once before starting any updates.

Note that specifically this initializes the cache of the row and column weights on workers when `use_factors_weights_cache` is True. In this case, if these weights are being calcualted and reset after the object is created, it is important to ensure this ops is run afterwards so the cache reflects the correct values.

Methods

`__init__`

```
__init__(
    input_rows,
    input_cols,
    n_components,
    unobserved_weight=0.1,
    regularization=None,
    row_init='random',
    col_init='random',
    num_row_shards=1,
    num_col_shards=1,
    row_weights=1,
    col_weights=1,
    use_factors_weights_cache=True,
    use_gramian_cache=True
)
```

Creates model for WALS matrix factorization.

Args:

- `input_rows` : total number of rows for input matrix.
- `input_cols` : total number of cols for input matrix.
- `n_components` : number of dimensions to use for the factors.
- `unobserved_weight` : weight given to unobserved entries of matrix.
- `regularization` : weight of L2 regularization term. If None, no regularization is done.
- `row_init` : initializer for row factor. Can be a tensor or numpy constant. If set to "random", the value is initialized randomly.
- `col_init` : initializer for column factor. See row_init for details.
- `num_row_shards` : number of shards to use for row factors.
- `num_col_shards` : number of shards to use for column factors.
- `row_weights` : Must be in one of the following three formats: None, a list of lists of non-negative real numbers (or equivalent iterables) or a single non-negative real number.
    - When set to None, w_ij = unobserved_weight, which simplifies to ALS. Note that col_weights must also be set to

"None" in this case.

  - If it is a list of lists of non-negative real numbers, it needs to be in the form of [[w_0, w_1, ...], [w_k, ...], [...]], with the number of inner lists matching the number of row factor shards and the elements in each inner list are the weights for the rows of the corresponding row factor shard. In this case, w_ij = unonbserved_weight + row_weights[i] * col_weights[j].
  - If this is a single non-negative real number, this value is used for all row weights and w_ij = unobserved_weight + row_weights * col_weights[j]. Note that it is allowed to have row_weights as a list while col_weights a single number or vice versa.
- `col_weights`: See row_weights.
- `use_factors_weights_cache`: When True, the factors and weights will be cached on the workers before the updates start. Defaults to True. Note that the weights cache is initialized through `worker_init`, and the row/col factors cache is initialized through `initialize_{col/row}_update_op`. In the case where the weights are computed outside and set before the training iterations start, it is important to ensure the `worker_init` op is run afterwards for the weights cache to take effect.
- `use_gramian_cache`: When True, the Gramians will be cached on the workers before the updates start. Defaults to True.

project_col_factors

```
project_col_factors(
    sp_input=None,
    transpose_input=False,
    projection_weights=None
)
```

Projects the column factors.

This computes the column embedding v_j for an observed column a_j by solving one iteration of the update equations.

Args:

- `sp_input`: A SparseTensor representing a set of columns. Please note that the row indices of this SparseTensor must match the model row feature indexing while the column indices are ignored. The returned results will be in the same ordering as the input columns.
- `transpose_input`: If true, the input will be logically transposed and the columns corresponding to the transposed input are projected.
- `projection_weights`: The column weights to be used for the projection. If None then 1.0 is used. This can be either a scaler or a rank-1 tensor with the number of elements matching the number of columns to be projected. Note that the row weights will be determined by the underlying WALS model.

Returns:

Projected column factors.

project_row_factors

```
project_row_factors(
    sp_input=None,
    transpose_input=False,
    projection_weights=None
)
```

Projects the row factors.

This computes the row embedding u_i for an observed row a_i by solving one iteration of the update equations.

Args:

- `sp_input`: A SparseTensor representing a set of rows. Please note that the column indices of this SparseTensor must match the model column feature indexing while the row indices are ignored. The returned results will be in the same ordering as the input rows.
- `transpose_input`: If true, the input will be logically transposed and the rows corresponding to the transposed input are projected.
- `projection_weights`: The row weights to be used for the projection. If None then 1.0 is used. This can be either a scaler or a rank-1 tensor with the number of elements matching the number of rows to be projected. Note that the column weights will be determined by the underlying WALS model.

Returns:

Projected row factors.

`scatter_update`

```
@classmethod
scatter_update(
    cls,
    factor,
    indices,
    values,
    sharding_func,
    name=None
)
```

Helper function for doing sharded scatter update.

`update_col_factors`

```
update_col_factors(
    sp_input=None,
    transpose_input=False
)
```

Updates the column factors.

Args:

- `sp_input`: A SparseTensor representing a subset of columns of the full input. Please refer to comments for update_row_factors for restrictions.
- `transpose_input`: If true, the input will be logically transposed and the columns corresponding to the transposed input are updated.

Returns:

A tuple consisting of the following elements: `new_values`: *New values for the column factors.* `update_op`: An op that assigns the newly computed values to the column factors. `unregularized_loss`: *A tensor (scalar) that contains the*

*normalized minibatch loss corresponding to sp_input, without the regularization term. If sp_input contains the columns {A_{:, j}, j \in J}, and the input matrix A has m total columns, then the unregularized loss is: (\|\sqrt W_J \odot (A_J - U V_J^T)\|_F^2 * m / |I| The total loss is unregularized_loss + regularization.* `regularization`: A tensor (scalar) that contains the normalized regularization term for the minibatch loss corresponding to sp_input. If sp_input contains the columns {A_{:, j}, j \in J}, and the input matrix A has m total columns, then the regularization term is: \lambda \|V_J\|_F^2) * m / |J| + \lambda \|U\|_F^2. * `sum_weights`: The sum of the weights W_J corresponding to sp_input, normalized by a factor of m / |J|. The root weighted squared error is: \sqrt(unregularized_loss / sum_weights).

`update_row_factors`

```
update_row_factors(
    sp_input=None,
    transpose_input=False
)
```

Updates the row factors.

Args:

- `sp_input`: A SparseTensor representing a subset of rows of the full input in any order. Please note that this SparseTensor must retain the indexing as the original input.
- `transpose_input`: If true, the input will be logically transposed and the rows corresponding to the transposed input are updated.

Returns:

A tuple consisting of the following elements: `new_values`: *New values for the row factors.* `update_op`: An op that assigns the newly computed values to the row factors. `unregularized_loss`: *A tensor (scalar) that contains the normalized minibatch loss corresponding to sp_input, without the regularization term. If sp_input contains the rows {A_{i, :}, i \in I}, and the input matrix A has n total rows, then the unregularized loss is: (\|\sqrt W_I \odot (A_I - U_I V^T)\|_F^2 * n / |I| The total loss is unregularized_loss + regularization.* `regularization`: A tensor (scalar) that contains the normalized regularization term for the minibatch loss corresponding to sp_input. If sp_input contains the rows {A_{i, :}, i \in I}, and the input matrix A has n total rows, then the regularization term is: \lambda \|U_I\|_F^2) * n / |I| + \lambda \|V\|_F^2. * `sum_weights`: The sum of the weights W_I corresponding to sp_input, normalized by a factor of n / |I|. The root weighted squared error is: \sqrt(unregularized_loss / sum_weights).

Last updated November 2, 2017.

English