tf.contrib.factorization.WALSMatrixFactorization

Contents

Class **WALSMatrixFactorization**

Inherits From: `Estimator`

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

An Estimator for Weighted Matrix Factorization, using the WALS method.

WALS (Weighted Alternating Least Squares) is an algorithm for weighted matrix factorization. It computes a low-rank approximation of a given sparse (n x m) matrix A, by a product of two matrices, U * V^T, where U is a (n x k) matrix and V is a (m x k) matrix. Here k is the rank of the approximation, also called the embedding dimension. We refer to U as the row factors, and V as the column factors. See tensorflow/contrib/factorization/g3doc/wals.md for the precise problem formulation.

The training proceeds in sweeps: during a row_sweep, we fix V and solve for U. During a column sweep, we fix U and solve for V. Each one of these problems is an unconstrained quadratic minimization problem and can be solved exactly (it can also be solved in mini-batches, since the solution decouples nicely). The alternating between sweeps is achieved by using a hook during training, which is responsible for keeping track of the sweeps and running preparation ops at the beginning of each sweep. It also updates the global_step variable, which keeps track of the number of batches processed since the beginning of training. The current implementation assumes that the training is run on a single machine, and will fail if config.num_worker_replicas is not equal to one. Training is done by calling self.fit(input_fn=input_fn), where input_fn provides two tensors: one for rows of the input matrix, and one for rows of the transposed input matrix (i.e. columns of the original matrix). Note that during a row sweep, only row batches are processed (ignoring column batches) and vice-versa. Also note that every row (respectively every column) of the input matrix must be processed at least once for the sweep to be considered complete. In particular, training will not make progress if input_fn does not generate some rows.

For prediction, given a new set of input rows A' (e.g. new rows of the A matrix), we compute a corresponding set of row factors U', such that U' * V^T is a good approximation of A'. We call this operation a row projection. A similar operation is defined for columns. Projection is done by calling self.get_projections(input_fn=input_fn), where input_fn satisfies the constraints given below.

The input functions must satisfy the following constraints: Calling input_fn must return a tuple (features, labels) where labels is None, and features is a dict containing the following keys: TRAIN: - WALSMatrixFactorization.INPUT_ROWS: float32 SparseTensor (matrix). Rows of the input matrix to process (or to project). - WALSMatrixFactorization.INPUT_COLS: float32 SparseTensor (matrix). Columns of the input matrix to process (or to project), transposed. INFER: - WALSMatrixFactorization.INPUT_ROWS: float32 SparseTensor (matrix). Rows to project. - WALSMatrixFactorization.INPUT_COLS: float32 SparseTensor (matrix). Columns to project. - WALSMatrixFactorization.PROJECT_ROW: Boolean Tensor. Whether to project the rows or columns. - WALSMatrixFactorization.PROJECTION_WEIGHTS (Optional): float32 Tensor (vector). The weights to use in the projection. EVAL: - WALSMatrixFactorization.INPUT_ROWS: float32 SparseTensor (matrix). Rows to project. -

WALSMatrixFactorization.INPUT_COLS: float32 SparseTensor (matrix). Columns to project. - WALSMatrixFactorization.PROJECT_ROW: Boolean Tensor. Whether to project the rows or columns.

Properties

config

model_dir

Methods

__init__

```
__init__(
    num_rows,
    num_cols,
    embedding_dimension,
    unobserved_weight=0.1,
    regularization_coeff=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_for_training=True,
    use_gramian_cache_for_training=True,
    max_sweeps=None,
    model_dir=None,
    config=None
)
```

Creates a model for matrix factorization using the WALS method.

Args:

- `num_rows`: Total number of rows for input matrix.
- `num_cols`: Total number of cols for input matrix.
- `embedding_dimension`: Dimension to use for the factors.
- `unobserved_weight`: Weight of the unobserved entries of matrix.
- `regularization_coeff`: Weight of the L2 regularization term. Defaults to None, in which case the problem is not regularized.
- `row_init`: Initializer for row factor. Must be either:
  - A tensor: The row factor matrix is initialized to this tensor,
  - A numpy constant,
  - "random": The rows are initialized using a normal distribution.
- `col_init`: Initializer for column factor. See row_init.
- `num_row_shards`: Number of shards to use for the row factors.
- `num_col_shards`: Number of shards to use for the column factors.
- `row_weights`: Must be in one of the following three formats:
  - None: In this case, the weight of every entry is the unobserved_weight and the problem simplifies to ALS. Note

that, in this case, col_weights must also be set to "None".
    - List of lists of non-negative scalars, of the form [[w_0, w_1, ...], [w_k, ...], [...]], where the number of inner lists equal to the number of row factor shards and the elements in each inner list are the weights for the rows of that shard. In this case, w_ij = unonbserved_weight + row_weights[i] * col_weights[j].
    - A non-negative scalar: This value is used for all row weights. Note that it is allowed to have row_weights as a list and col_weights as a scalar, or vice-versa.
- `col_weights`: See row_weights.
- `use_factors_weights_cache_for_training`: Boolean, whether the factors and weights will be cached on the workers before the updates start, during training. Defaults to True. Note that caching is disabled during prediction.
- `use_gramian_cache_for_training`: Boolean, whether the Gramians will be cached on the workers before the updates start, during training. Defaults to True. Note that caching is disabled during prediction.
- `max_sweeps`: integer, optional. Specifies the number of sweeps for which to train the model, where a sweep is defined as a full update of all the row factors (resp. column factors). If **`steps`** or **`max_steps`** is also specified in model.fit(), training stops when either of the steps condition or sweeps condition is met.
- `model_dir`: The directory to save the model results and log files.
- `config`: A Configuration object. See Estimator.

Raises:

- `ValueError`: If config.num_worker_replicas is strictly greater than one. The current implementation only supports running on a single worker.

evaluate

```
evaluate(
    x=None,
    y=None,
    input_fn=None,
    feed_fn=None,
    batch_size=None,
    steps=None,
    metrics=None,
    name=None,
    checkpoint_path=None,
    hooks=None,
    log_progress=True
)
```

See **`Evaluable`**. (deprecated arguments)

SOME ARGUMENTS ARE DEPRECATED. They will be removed after 2016-12-01. Instructions for updating: Estimator is decoupled from Scikit Learn interface by moving into separate class SKCompat. Arguments x, y and batch_size are only available in the SKCompat class, Estimator will only accept input_fn. Example conversion: est = Estimator(...) -> est = SKCompat(Estimator(...))

Raises:

- `ValueError`: If at least one of **`x`** or **`y`** is provided, and at least one of **`input_fn`** or **`feed_fn`** is provided. Or if **`metrics`** is not **`None`** or **`dict`**.

export

```
export(
    export_dir,
    input_fn=export._default_input_fn,
    input_feature_key=None,
    use_deprecated_input_fn=True,
    signature_fn=None,
    prediction_key=None,
    default_batch_size=1,
    exports_to_keep=None,
    checkpoint_path=None
)
```

Exports inference graph into given dir. (deprecated)

THIS FUNCTION IS DEPRECATED. It will be removed after 2017-03-25. Instructions for updating: Please use Estimator.export_savedmodel() instead.

Args:

- `export_dir`: A string containing a directory to write the exported graph and checkpoints.
- `input_fn`: If `use_deprecated_input_fn` is true, then a function that given `Tensor` of `Example` strings, parses it into features that are then passed to the model. Otherwise, a function that takes no argument and returns a tuple of (features, labels), where features is a dict of string key to `Tensor` and labels is a `Tensor` that's currently not used (and so can be `None`).
- `input_feature_key`: Only used if `use_deprecated_input_fn` is false. String key into the features dict returned by `input_fn` that corresponds to a the raw `Example` strings `Tensor` that the exported model will take as input. Can only be `None` if you're using a custom `signature_fn` that does not use the first arg (examples).
- `use_deprecated_input_fn`: Determines the signature format of `input_fn`.
- `signature_fn`: Function that returns a default signature and a named signature map, given `Tensor` of `Example` strings, `dict` of `Tensor`s for features and `Tensor` or `dict` of `Tensor`s for predictions.
- `prediction_key`: The key for a tensor in the `predictions` dict (output from the `model_fn`) to use as the `predictions` input to the `signature_fn`. Optional. If `None`, predictions will pass to `signature_fn` without filtering.
- `default_batch_size`: Default batch size of the `Example` placeholder.
- `exports_to_keep`: Number of exports to keep.
- `checkpoint_path`: the checkpoint path of the model to be exported. If it is `None` (which is default), will use the latest checkpoint in export_dir.

Returns:

The string path to the exported directory. NB: this functionality was added ca. 2016/09/25; clients that depend on the return value may need to handle the case where this function returns None because subclasses are not returning a value.

`export_savedmodel`

```
export_savedmodel(
    export_dir_base,
    serving_input_fn,
    default_output_alternative_key=None,
    assets_extra=None,
    as_text=False,
    checkpoint_path=None,
    graph_rewrite_specs=(GraphRewriteSpec((tag_constants.SERVING,), ()),)
)
```

Exports inference graph as a SavedModel into given dir.

Args:

- `export_dir_base`: A string containing a directory to write the exported graph and checkpoints.
- `serving_input_fn`: A function that takes no argument and returns an `InputFnOps`.
- `default_output_alternative_key`: the name of the head to serve when none is specified. Not needed for single-headed models.
- `assets_extra`: A dict specifying how to populate the assets.extra directory within the exported SavedModel. Each key should give the destination path (including the filename) relative to the assets.extra directory. The corresponding value gives the full path of the source file to be copied. For example, the simple case of copying a single file without renaming it is specified as `{'my_asset_file.txt': '/path/to/my_asset_file.txt'}`.
- `as_text`: whether to write the SavedModel proto in text format.
- `checkpoint_path`: The checkpoint path to export. If None (the default), the most recent checkpoint found within the model directory is chosen.
- `graph_rewrite_specs`: an iterable of `GraphRewriteSpec`. Each element will produce a separate MetaGraphDef within the exported SavedModel, tagged and rewritten as specified. Defaults to a single entry using the default serving tag ("serve") and no rewriting.

Returns:

The string path to the exported directory.

Raises:

- `ValueError`: if an unrecognized export_type is requested.

`fit`

```
fit(
    x=None,
    y=None,
    input_fn=None,
    steps=None,
    batch_size=None,
    monitors=None,
    max_steps=None
)
```

See `Trainable`. (deprecated arguments)

SOME ARGUMENTS ARE DEPRECATED. They will be removed after 2016-12-01. Instructions for updating: Estimator is decoupled from Scikit Learn interface by moving into separate class SKCompat. Arguments x, y and batch_size are only available in the SKCompat class, Estimator will only accept input_fn. Example conversion: est = Estimator(...) -> est = SKCompat(Estimator(...))

Raises:

- `ValueError`: If `x` or `y` are not `None` while `input_fn` is not `None`.
- `ValueError`: If both `steps` and `max_steps` are not `None`.

`get_col_factors`

```
get_col_factors()
```

Returns the column factors of the model, loading them from checkpoint.

Should only be run after training.

Returns:

A list of the column factors of the model.

`get_params`

```
get_params(deep=True)
```

Get parameters for this estimator.

Args:

- `deep`: boolean, optional

  If `True`, will return the parameters for this estimator and contained subobjects that are estimators.

Returns:

- `params`: mapping of string to any Parameter names mapped to their values.

`get_projections`

```
get_projections(input_fn)
```

Computes the projections of the rows or columns given in input_fn.

Runs predict() with the given input_fn, and returns the results. Should only be run after training.

Args:

- `input_fn`: Input function which specifies the rows or columns to project.

Returns:

A generator of the projected factors.

`get_row_factors`

```
get_row_factors()
```

Returns the row factors of the model, loading them from checkpoint.

Should only be run after training.

Returns:

A list of the row factors of the model.

get_variable_names

```
get_variable_names()
```

Returns list of all variable names in this model.

Returns:

List of names.

get_variable_value

```
get_variable_value(name)
```

Returns value of the variable given by name.

Args:

- `name`: string, name of the tensor.

Returns:

Numpy array - value of the tensor.

partial_fit

```
partial_fit(
    x=None,
    y=None,
    input_fn=None,
    steps=1,
    batch_size=None,
    monitors=None
)
```

Incremental fit on a batch of samples. (deprecated arguments)

SOME ARGUMENTS ARE DEPRECATED. They will be removed after 2016-12-01. Instructions for updating: Estimator is decoupled from Scikit Learn interface by moving into separate class SKCompat. Arguments x, y and batch_size are only available in the SKCompat class, Estimator will only accept input_fn. Example conversion: est = Estimator(...) -> est = SKCompat(Estimator(...))

This method is expected to be called several times consecutively on different or the same chunks of the dataset. This either can implement iterative training or out-of-core/online training.

This is especially useful when the whole dataset is too big to fit in memory at the same time. Or when model is taking long time to converge, and you want to split up training into subparts.

Args:

- `x`: Matrix of shape [n_samples, n_features...]. Can be iterator that returns arrays of features. The training input samples for fitting the model. If set, `input_fn` must be `None`.
- `y`: Vector or matrix [n_samples] or [n_samples, n_outputs]. Can be iterator that returns array of labels. The training label values (class labels in classification, real numbers in regression). If set, `input_fn` must be `None`.
- `input_fn`: Input function. If set, `x`, `y`, and `batch_size` must be `None`.
- `steps`: Number of steps for which to train model. If `None`, train forever.
- `batch_size`: minibatch size to use on the input, defaults to first dimension of `x`. Must be `None` if `input_fn` is provided.
- `monitors`: List of `BaseMonitor` subclass instances. Used for callbacks inside the training loop.

Returns:

`self`, for chaining.

Raises:

- `ValueError`: If at least one of `x` and `y` is provided, and `input_fn` is provided.

`predict`

```
predict(
    x=None,
    input_fn=None,
    batch_size=None,
    outputs=None,
    as_iterable=True
)
```

Returns predictions for given features. (deprecated arguments)

SOME ARGUMENTS ARE DEPRECATED. They will be removed after 2016-12-01. Instructions for updating: Estimator is decoupled from Scikit Learn interface by moving into separate class SKCompat. Arguments x, y and batch_size are only available in the SKCompat class, Estimator will only accept input_fn. Example conversion: est = Estimator(...) -> est = SKCompat(Estimator(...))

Args:

- `x`: Matrix of shape [n_samples, n_features...]. Can be iterator that returns arrays of features. The training input samples for fitting the model. If set, `input_fn` must be `None`.
- `input_fn`: Input function. If set, `x` and 'batch_size' must be `None`.
- `batch_size`: Override default batch size. If set, 'input_fn' must be 'None'.
- `outputs`: list of `str`, name of the output to predict. If `None`, returns all.
- `as_iterable`: If True, return an iterable which keeps yielding predictions for each example until inputs are exhausted. Note: The inputs must terminate if you want the iterable to terminate (e.g. be sure to pass num_epochs=1 if you are using something like read_batch_features).

Returns:

A numpy array of predicted classes or regression values if the constructor's `model_fn` returns a `Tensor` for `predictions` or a `dict` of numpy arrays if `model_fn` returns a `dict`. Returns an iterable of predictions if as_iterable is True.

Raises:

- `ValueError`: If x and input_fn are both provided or both `None`.

set_params

```
set_params(**params)
```

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Args:

- `**params`: Parameters.

Returns:

self

Raises:

- `ValueError`: If params contain invalid names.

Class Members

COMPLETED_SWEEPS

INPUT_COLS

INPUT_ROWS

PROJECTION_RESULT

PROJECTION_WEIGHTS

PROJECT_ROW

Last updated November 2, 2017.