tf.contrib.learn.DynamicRnnEstimator

Contents

Class **`DynamicRnnEstimator`**

Inherits From: `Estimator`

Defined in `tensorflow/contrib/learn/python/learn/estimators/dynamic_rnn_estimator.py`.

Properties

`config`

`model_dir`

Methods

`__init__`

```
__init__(
    problem_type,
    prediction_type,
    sequence_feature_columns,
    context_feature_columns=None,
    num_classes=None,
    num_units=None,
    cell_type='basic_rnn',
    optimizer='SGD',
    learning_rate=0.1,
    predict_probabilities=False,
    momentum=None,
    gradient_clipping_norm=5.0,
    dropout_keep_probabilities=None,
    model_dir=None,
    feature_engineering_fn=None,
    config=None
)
```

Initializes a `DynamicRnnEstimator`.

The input function passed to this `Estimator` optionally contains keys `RNNKeys.SEQUENCE_LENGTH_KEY`. The value corresponding to `RNNKeys.SEQUENCE_LENGTH_KEY` must be vector of size `batch_size` where entry `n` corresponds to the length of the `n` th sequence in the batch. The sequence length feature is required for batches of varying sizes. It will be used to calculate loss and evaluation metrics. If `RNNKeys.SEQUENCE_LENGTH_KEY` is not included, all sequences are assumed to have length equal to the size of dimension 1 of the input to the RNN.

In order to specify an initial state, the input function must include keys `STATE_PREFIX_i` for all `0 <= i < n` where `n` is the number of nested elements in `cell.state_size`. The input function must contain values for all state components or none of them. If none are included, then the default (zero) state is used as an initial state. See the documentation for `dict_to_state_tuple` and `state_tuple_to_dict` for further details. The input function can call rnn_common.construct_rnn_cell() to obtain the same cell type that this class will select from arguments to **init**.

The `predict()` method of the `Estimator` returns a dictionary with keys `STATE_PREFIX_i` for `0 <= i < n` where `n` is the number of nested elements in `cell.state_size`, along with `PredictionKey.CLASSES` for problem type `CLASSIFICATION` or `PredictionKey.SCORES` for problem type `LINEAR_REGRESSION`. The value keyed by `PredictionKey.CLASSES` or `PredictionKey.SCORES` has shape `[batch_size, padded_length]` in the multi-value case and shape `[batch_size]` in the single-value case. Here, `padded_length` is the largest value in the `RNNKeys.SEQUENCE_LENGTH` `Tensor` passed as input. Entry `[i, j]` is the prediction associated with sequence `i` and time step `j`. If the problem type is `CLASSIFICATION` and `predict_probabilities` is `True`, it will also include key `PredictionKey.PROBABILITIES`.

Args:

- `problem_type`: whether the `Estimator` is intended for a regression or classification problem. Value must be one of `ProblemType.CLASSIFICATION` or `ProblemType.LINEAR_REGRESSION`.
- `prediction_type`: whether the `Estimator` should return a value for each step in the sequence, or just a single value for the final time step. Must be one of `PredictionType.SINGLE_VALUE` or `PredictionType.MULTIPLE_VALUE`.
- `sequence_feature_columns`: An iterable containing all the feature columns describing sequence features. All items in the iterable should be instances of classes derived from `FeatureColumn`.
- `context_feature_columns`: An iterable containing all the feature columns describing context features, i.e., features that apply across all time steps. All items in the set should be instances of classes derived from `FeatureColumn`.
- `num_classes`: the number of classes for a classification problem. Only used when `problem_type=ProblemType.CLASSIFICATION`.
- `num_units`: A list of integers indicating the number of units in the `RNNCell`s in each layer.
- `cell_type`: A subclass of `RNNCell` or one of 'basic_rnn,' 'lstm' or 'gru'.
- `optimizer`: The type of optimizer to use. Either a subclass of `Optimizer`, an instance of an `Optimizer`, a callback that returns an optimizer, or a string. Strings must be one of 'Adagrad', 'Adam', 'Ftrl', 'Momentum', 'RMSProp' or 'SGD. See `layers.optimize_loss` for more details.
- `learning_rate`: Learning rate. This argument has no effect if `optimizer` is an instance of an `Optimizer`.
- `predict_probabilities`: A boolean indicating whether to predict probabilities for all classes. Used only if `problem_type` is `ProblemType.CLASSIFICATION`
- `momentum`: Momentum value. Only used if `optimizer_type` is 'Momentum'.
- `gradient_clipping_norm`: Parameter used for gradient clipping. If `None`, then no clipping is performed.
- `dropout_keep_probabilities`: a list of dropout probabilities or `None`. If a list is given, it must have length `len(num_units) + 1`. If `None`, then no dropout is applied.
- `model_dir`: The directory in which to save and restore the model graph, parameters, etc.
- `feature_engineering_fn`: Takes features and labels which are the output of `input_fn` and returns features and labels which will be fed into `model_fn`. Please check `model_fn` for a definition of features and labels.
- `config`: A `RunConfig` instance.

Raises:

- `ValueError`: `problem_type` is not one of `ProblemType.LINEAR_REGRESSION` or `ProblemType.CLASSIFICATION`.

- `ValueError`: `problem_type` is `ProblemType.CLASSIFICATION` but `num_classes` is not specifieProblemType
- `ValueError`: `prediction_type` is not one of `PredictionType.MULTIPLE_VALUE` or `PredictionType.SINGLE_VALUE`.

`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_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_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.

Last updated November 2, 2017.

Stay Connected

Blog

GitHub

Twitter

Support

Issue Tracker

Release Notes

Stack Overflow

English

Terms | **Privacy**