tavolo

Release 0.7.0

Elior Cohen

Oct 08, 2020

CONTENTS

tavolo aims to package together valuable modules and functionality written for TensorFlow high-level Keras API for ease of use.

You see, the deep learning world is moving fast, and new ideas keep on coming.

tavolo gathers implementations of these useful ideas from the community (by contribution, from Kaggle etc.) and makes them accessible in a single PyPI hosted package that compliments the tf.keras module.

CHAPTER

ONE

SHOWCASE

tavolo's API is straightforward and adopting its modules is as easy as it gets.

In tavolo, you'll find implementations for basic layers like *PositionalEncoding* to complex modules like the Transformer's *MultiHeadedAttention*. You'll also find non-layer implementations that can ease development, like the *LearningRateFinder*.

For example, if we wanted to add head a Yang-style attention mechanism into our model and look for the optimal learning rate, it would look something like:

```
import tensorflow as tf
import tavolo as tvl

model = tf.keras.Sequential([
    tf.keras.layers.Embedding(input_dim=vocab_size, output_dim=embedding_size, input_
↪length=max_len),
    tvl.seq2vec.YangAttention(n_units=64),  # <--- Add Yang style attention
    tf.keras.layers.Dense(n_hidden_units, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')])

model.compile(optimizer=tf.keras.optimizers.SGD(), loss=tf.keras.losses.
↪BinaryCrossentropy())

# Run learning rate range test
lr_finder = tvl.learning.LearningRateFinder(model=model)

learning_rates, losses = lr_finder.scan(train_data, train_labels, min_lr=0.0001, max_
↪lr=1.0, batch_size=128)

### Plot the results to choose your learning rate
```

You are welcome continue to the *Installation* page, or explore the different modules available:

1.1 Installation

tavolo is hosted on PyPI, and the source code is available on Github.

Note: Tavolo will not install tensorflow by itself, this is to prevent installations of CPU and GPU versions together. It is the user's responsibility to install the tensorflow library

1.1.1 Install from PyPI

To install it using `pip`, simply run inside your environment

```
pip install tavolo
```

1.1.2 Install from source code

If you prefer to install from source code, first clone the repository

```
git clone https://github.com/eliorc/tavolo.git
```

then navigate into the directory, and install

```
cd tavolo
python setup.py install
```

1.2 Contributing

First of all, thanks for considering contributing code to tavolo!

Before contributing please open an issue in the Github repository explaining the module you wish to contribute.

Assuming the module is accepted, it will be tagged so in the issue opened so you can start implementing to avoid wasting contributor's time for code that won't be accepted.

tavolo is built to compliment the tf.keras module, make sure your contributions are focused at it.

Once your suggested module is accepted, follow the guidelines in *Code and Documentation* and *Testing*, and once completed you can open a pull request to the `dev` branch.

Note: Do not create pull requests into the `master` branch. Pull requests should be made to the `dev` branch, from which changes will be merged into `master` on releases.

1.2.1 Code and Documentation

tavolo is open source, viewing the source code of a module and understanding every step in its implementation should be easy and straightforward, so users can trust the module they wish to use.
To fulfill this requirement, follow these guidelines:

1. Comments - Even if the code is clear, use comments to explain steps (step comment example).
2. Variable verbosity - Use verbose variable names that imply the meaning of their content, e.g. use `mask` instead of `m`.
3. Clear tensor shapes - When applying operations on tensors, include the shape of the result in a comment. (tensor shape example).
4. Format - reStructuredText is the documentation format use, and specifically PEP 287 (PyCharm's default) for class methods. On class level docstring, make sure you always include the following sections:
   - Arguments - For the `__init__` arguments (Arguments section example).
   - Examples - For examples (Examples section example)
   - References - For sources (articles etc.) for further reading (References section example).

   If you are contributing a `tf.keras.layers.Layer` subclass, also include:
   - Input Shape - Input shape accepted by the layer's `call` method (Input Shape section example).
   - Output Shape - Output shape accepted by the layer's `call` method (Output Shape section example).

1.2.2 Testing

Tavolo uses pytest and codecov for its testing. Make sure you write your tests to cover the full functionality of the contributed code.
The tests should be written as a separate file for each module and in the destination `tests/<parent_module>/<module_name>_test.py`.
For example for the module `tvl.normalization.LayerNormalization`, the tests should be written in `tests/normalization/layer_normalization_test.py`.
It is quite difficult to define in advance which tests are mandatory, but you can draw insipration from the existing modules.
In the specific case of `tf.keras.layers.Layer` implementation, always include:

1. `test_shapes()` - Given accepted input shapes, make sure the output shape is as expected (test_shapes() example).
2. `test_masking()` - Make sure layer supports masking (test_masking() example).
3. `test_serialization()` - Make sure layer can be saved and loaded using `get_config` and `from_config` (test_serialization() example).

If possible, also include `test_logic()` for evaluating expected output given known input (test_logic() example).

When done, run tests locally to make sure everything works fine, to do so, make sure you have installed the test requirements from the requirements/test.py file and run tests locally using the following command from the main directory

```
pytest --cov=tavolo tests/
```

Strive for 100% coverage, and if all is well, create a pull request (to the `dev` branch) and it will be added to the package in a following release.

1.3 Embeddings

Modules related to embeddings

Modules

- *PositionalEncoding*
- *DynamicMetaEmbedding*
- *ContextualDynamicMetaEmbedding*

1.3.1 PositionalEncoding

Create a positional encoding layer, usually added on top of an embedding layer. Embeds information about the position of the elements using the formula

$$PE[pos, 2i] = sin\left(\frac{pos}{normalize\_factor^{\frac{2i}{embedding\_dim}}}\right)$$
$$PE[pos, 2i+1] = cos\left(\frac{pos}{normalize\_factor^{\frac{2i}{embedding\_dim}}}\right)$$

The resulting embedding gets added (point-wise) to the input.

Arguments

- *max_sequence_length* (`int`): Maximum sequence length of input
- *embedding_dim* (`int`): Dimensionality of the of the input's last dimension
- *normalize_factor* (`float`): Normalize factor
- *name* (`str`): Layer name

Input shape

(batch_size, time_steps, channels) where time_steps equals to the `max_sequence_length` and channels to `embedding_dim`

Output shape

Same shape as input.

Examples

```
import tensorflow as tf
import tavolo as tvl

model = tf.keras.Sequential([tf.keras.layers.Embedding(vocab_size, 8, input_
↪length=max_sequence_length),
                             tvl.embeddings.PositionalEncoding(max_sequence_
↪length=max_sequence_length,
                                                               embedding_dim=8)])   #␣
↪Add positional encoding
```

References

Attention Is All You Need

1.3.2 `DynamicMetaEmbedding`

Applies learned attention to different sets of embeddings matrices per token, to mix separate token representations into a joined one. Self attention is word-dependent, meaning each word's representation in the output is only dependent on the word's original embeddings in the given matrices, and the attention vector.

Arguments

- *embedding_matrices* (`List[np.ndarray]`): List of embedding matrices
- *output_dim* (`int`): Dimension of the output embedding
- *mask_zero* (`bool`): Whether or not the input value 0 is a special "padding" value that should be masked out
- *input_length* (`Optional[int]`): Parameter to be passed into internal `tf.keras.layers.Embedding` matrices
- *name* (`str`): Layer name

Input shape

(batch_size, time_steps)

Output shape

(batch_size, time_steps, output_dim)

Examples

Create Dynamic Meta Embeddings using 2 separate embedding matrices. Notice it is the user's responsibility to make sure all the arguments needed in the embedding lookup are passed to the `tf.keras.layers.Embedding` constructors (like `trainable=False`).

```
import tensorflow as tf
import tavolo as tvl

w2v_embedding = np.array(...)  # Pre-trained embedding matrix

glove_embedding = np.array(...)  # Pre-trained embedding matrix

model = tf.keras.Sequential([tf.keras.layers.Input(shape=(MAX_SEQUENCE_LENGTH,),␣
↪dtype='int32'),
                             tvl.embeddings.DynamicMetaEmbedding([w2v_embedding,␣
↪glove_embedding],
                                                                  input_length=MAX_
↪SEQUENCE_LENGTH)])  # Use DME embeddings
```

Using the same example as above, it is possible to define the output's channel size

```
model = tf.keras.Sequential([tf.keras.layers.Input(shape=(MAX_SEQUENCE_LENGTH,),␣
↪dtype='int32'),
                             tvl.embeddings.DynamicMetaEmbedding([w2v_embedding,␣
↪glove_embedding],
                                                                  input_length=MAX_
↪SEQUENCE_LENGTH,
                                                                  output_dim=200)])
```

References

Dynamic Meta-Embeddings for Improved Sentence Representations

1.3.3 `ContextualDynamicMetaEmbedding`

Applies learned attention to different sets of embeddings matrices per token, to mix separate token representations into a joined one. Self attention is context-dependent, meaning each word's representation in the output is only dependent on the sentence's original embeddings in the given matrices, and the attention vector. The context is generated by a BiLSTM.

Arguments

- *embedding_matrices* (`List[np.ndarray]`): List of embedding matrices
- *output_dim* (`int`): Dimension of the output embedding
- *mask_zero* (`bool`): Whether or not the input value 0 is a special "padding" value that should be masked out
- *input_length* (`Optional[int]`): Parameter to be passed into internal `tf.keras.layers.Embedding` matrices
- *n_lstm_units* (`int`): Number of units in each LSTM, (notated as *m* in the original article)
- *name* (`str`): Layer name

Input shape

(batch_size, time_steps)

Output shape

(batch_size, time_steps, output_dim)

Examples

Create Dynamic Meta Embeddings using 2 separate embedding matrices. Notice it is the user's responsibility to make sure all the arguments needed in the embedding lookup are passed to the `tf.keras.layers.Embedding` constructors (like `trainable=False`).

```
import tensorflow as tf
import tavolo as tvl

w2v_embedding = np.array(...)  # Pre-trained embedding matrix

glove_embedding = np.array(...)  # Pre-trained embedding matrix

model = tf.keras.Sequential([tf.keras.layers.Input(shape=(MAX_SEQUENCE_LENGTH,),␣
↪dtype='int32'),
                             tvl.embeddings.DynamicMetaEmbedding([w2v_embedding,␣
↪glove_embedding],
                                                                  input_length=MAX_
↪SEQUENCE_LENGTH)])  # Use CDME embeddings
```

Using the same example as above, it is possible to define the output's channel size and number of units in each LSTM

```
model = tf.keras.Sequential([tf.keras.layers.Input(shape=(MAX_SEQUENCE_LENGTH,),␣
↪dtype='int32'),
                             tvl.embeddings.DynamicMetaEmbedding([w2v_embedding,␣
↪glove_embedding],
                                                                  input_length=MAX_
↪SEQUENCE_LENGTH,
                                                                  n_lstm_units=128,␣
↪output_dim=200)])
```

References

Dynamic Meta-Embeddings for Improved Sentence Representations

1.4 Learning

Modules for altering the learning process

Modules

- *CyclicLearningRateCallback*
- *LearningRateFinder*

1.4.1 `CyclicLearningRateCallback`

Apply cyclic learning rate. Supports the following scale schemes:

- `triangular` - Triangular cycle
- `triangular2` - Triangular cycle that shrinks amplitude by half each cycle
- `exp_range` - Triangular cycle that shrinks amplitude by `gamma ** <cycle iterations>` each cycle

Arguments

- *base_lr* (`float`): Lower boundary of each cycle
- *max_lr* (`float`): Upper boundary of each cycle, may not be reached depending on the scaling function
- *step_size* (`int`): Number of batches per half-cycle (step)
- *scale_scheme* (`str`): One of `{'triangular', 'triangular2', 'exp_range'}`. If `scale_fn` is passed, this argument is ignored
- *gamma* (`float`): Constant used for the `exp_range`'s `scale_fn`, used as (`gamma ** <cycle iterations>`)
- *scale_fn* (`callable`): Custom scaling policy, accepts cycle index / iterations depending on the `scale_mode` and must return a value in the range [0, 1]. If passed, ignores `scale_scheme`
- *scale_mode* (`str`): Define whether `scale_fn` is evaluated on cycle index or cycle iterations

Examples

Apply a triangular cyclic learning rate (default), with a step size of 2000 batches

```
import tensorflow as tf
import tavolo as tvl

clr = tvl.learning.CyclicLearningRateCallback(base_lr=0.001, max_lr=0.006, step_
↪size=2000)

model.fit(X_train, Y_train, callbacks=[clr])
```

Apply a cyclic learning rate that shrinks amplitude by half each cycle

```
import tensorflow as tf
import tavolo as tvl

clr = tvl.learning.CyclicLearningRateCallback(base_lr=0.001, max_lr=0.006, step_
↪size=2000, scale_scheme='triangular2')

model.fit(X_train, Y_train, callbacks=[clr])
```

Apply a cyclic learning rate with a custom scaling function

```
import tensorflow as tf
import tavolo as tvl

scale_fn = lambda x: 0.5 * (1 + np.sin(x * np.pi / 2))
clr = tvl.learning.CyclicLearningRateCallback(base_lr=0.001, max_lr=0.006, step_
↪size=2000, scale_fn=scale_fn)

model.fit(X_train, Y_train, callbacks=[clr])
```

References

- Cyclical Learning Rates for Training Neural Networks
- Original implementation

1.4.2 `LearningRateFinder`

Learning rate finding utility for conducting the "LR range test", see article reference for more information

Use the `scan` method for finding the loss values for learning rates in the given range

Arguments

- *model* (`tf.keras.Model`): Model for conduct test for. Must call `model.compile` before using this utility

Examples

Run a learning rate range test in the domain `[0.0001, 1.0]`

```
import tensorflow as tf
import tavolo as tvl

train_data = ...
train_labels = ...

# Build model
model = tf.keras.Sequential([tf.keras.layers.Input(shape=(784,)),
                             tf.keras.layers.Dense(128, activation=tf.nn.relu),
                             tf.keras.layers.Dense(10, activation=tf.nn.softmax)])

# Must call compile with optimizer before test
model.compile(optimizer=tf.keras.optimizers.SGD(), loss=tf.keras.losses.
↪CategoricalCrossentropy())

# Run learning rate range test
lr_finder = tvl.learning.LearningRateFinder(model=model)

learning_rates, losses = lr_finder.scan(train_data, train_labels, min_lr=0.0001, max_
↪lr=1.0, batch_size=128)

### Plot the results to choose your learning rate
```

References

- Cyclical Learning Rates for Training Neural Networks

`learning.LearningRateFinder.`**`scan`**(*x*, *y*, *min_lr: float = 0.0001*, *max_lr: float = 1.0*, *batch_size: Optional[int] = None*, *steps: int = 100*) → Tuple[List[float], List[float]]

Scans the learning rate range `[min_lr, max_lr]` for loss values

Parameters

- **x** – Input data. It could be: - A Numpy array (or array-like), or a list of arrays (in case the model has multiple inputs) - A TensorFlow tensor, or a list of tensors (in case the model has multiple inputs) - A dict mapping input names to the corresponding array/tensors, if the model has named inputs - A `tf.data` dataset or a dataset iterator. Should return a tuple of either `(inputs, targets)` or `(inputs, targets, sample_weights)` - A generator or `keras.utils.Sequence` returning `(inputs, targets)` or `(inputs, targets, sample weights)`
- **y** – Target data. Like the input data *x*, it could be either Numpy array(s) or TensorFlow tensor(s). It should be consistent with `x` (you cannot have Numpy inputs and tensor targets, or inversely). If `x` is a dataset, dataset iterator, generator, or `tf.keras.utils.Sequence` instance, `y` should not be specified (since targets will be obtained from `x`).
- **min_lr** – Minimum learning rate

- **max_lr** – Maximum learning rate
- **batch_size** – Number of samples per gradient update. Do not specify the `batch_size` if your data is in the form of symbolic tensors, dataset, dataset iterators, generators, or `tf.keras.utils.Sequence` instances (since they generate batches)
- **steps** – Number of steps to scan between min_lr and max_lr

Returns Learning rates, losses documented

1.5 Seq2seq

Layers mapping sequences to sequences

Modules

- *MultiHeadedAttention*

1.5.1 MultiHeadedAttention

Applies (multi headed) attention, as in the Transformer

Arguments

- *n_heads* (`int`): Number of attention heads
- *n_units* (`int`): Number of units per head, defaults to the last dimension of the input
- *causal* (`bool`): Use causality (make each time point in output dependent only on previous time points of input)
- *name* (`str`): Layer name

`call` Arguments

- `inputs` (`List[tf.Tensor]`): List of the following tensors
- query: Query Tensor of shape [batch_size, Tq, dim]
- value: Value Tensor of shape [batch_size, Tv, dim].
- **key: Optional key Tensor of shape [batch_size, Tv, dim].** If not given, will use value for both key and value, which is the most common case
- `mask` (`List[tf.Tensor]`): List of the following tensors
- **query_mask: A boolean mask Tensor of shape [batch_size, Tq].** If given, the output will be zero at the positions where mask==False
- **value_mask: A boolean mask Tensor of shape [batch_size, Tv].** If given, will apply the mask such that values at positions where mask==False do not contribute to the result

Input shape

(batch_size, time_steps, channels)

Output shape

Same shape as input.

Examples

Apply a 4 headed (default) self attention

```
import tensorflow as tf
import tavolo as tvl

# Inputs
inputs = tf.keras.Input(shape=(max_seq_length,), dtype='int32')

# Embedding lookup
embedding_layer = tf.keras.layers.Embedding(max_tokens, dimension)
embedded = embedding_layer(inputs)

# Apply multi headed self attention
mh_attention = tvl.seq2seq.MultiHeadedAttention()
attended = mh_attention([embedded, embedded])
```

Apply a 4 headed attention, using a query vector and masking

```
import tensorflow as tf
import tavolo as tvl

# Inputs
query_input = tf.keras.Input(shape=(max_seq_length,), dtype='int32')
value_input = tf.keras.Input(shape=(max_seq_length,), dtype='int32')

# Embedding lookup
embedding_layer = tf.keras.layers.Embedding(max_tokens, dimension, mask_zero=True)
embedded_query = embedding_layer(query_input)
embedded_value = embedding_layer(value_input)

# Masks
query_mask = embedding_layer.compute_mask(query_input)
value_mask = embedding_layer.compute_mask(value_input)

# Apply multi headed self attention
mh_attention = tvl.seq2seq.MultiHeadedAttention()
attended = mh_attention([embedded_query, embedded_value], mask=[query_mask, value_
↪mask])
```

Note: Since the query and value should be passed separately, it is recommended to use the functional API or model subclassing to use this layer.

References

Attention Is All You Need

1.6 Seq2vec

Layers mapping sequences to vectors

Modules

- *YangAttention*

1.6.1 `YangAttention`

Reduce time dimension by applying attention using learned variables

Arguments

- *n_units* (`int`): Attention's variables units
- *name* (`str`): Layer name

Input shape

(batch_size, time_steps, channels)

Output shape

(batch_size, channels)

Examples

```
import tensorflow as tf
import tavolo as tvl


model = tf.keras.Sequential([tf.keras.layers.Embedding(vocab_size, 8, input_
↪length=max_sequence_length),
                             tvl.seq2vec.YangAttention()])
```

References

Hierarchical Attention Networks for Document Classification

CHAPTER

TWO

CONTRIBUTING

Want to contribute? Please read our *Contributing*.

PYTHON MODULE INDEX

INDEX