TextVectorization layer

TextVectorization class [source]

```
tf.keras.layers.TextVectorization(
    max_tokens=None,
    standardize="lower_and_strip_punctuation",
    split="whitespace",
    ngrams=None,
    output_mode="int",
    output_sequence_length=None,
    pad_to_max_tokens=False,
    vocabulary=None,
    idf_weights=None,
    sparse=False,
    ragged=False,
    **kwargs
)
```

A preprocessing layer which maps text features to integer sequences.

This layer has basic options for managing text in a Keras model. It transforms a batch of strings (one example = one string) into either a list of token indices (one example = 1D tensor of integer token indices) or a dense representation (one example = 1D tensor of float values representing data about the example's tokens). This layer is meant to handle natural language inputs. To handle simple string inputs (categorical strings or pre-tokenized strings) see `tf.keras.layers.StringLookup`.

The vocabulary for the layer must be either supplied on construction or learned via `adapt()`. When this layer is adapted, it will analyze the dataset, determine the frequency of individual string values, and create a vocabulary from them. This vocabulary can have unlimited size or be capped, depending on the configuration options for this layer; if there are more unique values in the input than the maximum vocabulary size, the most frequent terms will be used to create the vocabulary.

The processing of each example contains the following steps:

1. Standardize each example (usually lowercasing + punctuation stripping)
2. Split each example into substrings (usually words)
3. Recombine substrings into tokens (usually ngrams)
4. Index tokens (associate a unique int value with each token)
5. Transform each example using this index, either into a vector of ints or a dense float vector.

Some notes on passing callables to customize splitting and normalization for this layer:

1. Any callable can be passed to this Layer, but if you want to serialize this object you should only pass functions that are registered Keras serializables (see `tf.keras.utils.register_keras_serializable` for more details).
2. When using a custom callable for `standardize`, the data received by the callable will be exactly as passed to this layer. The callable should return a tensor of the same shape as the input.
3. When using a custom callable for `split`, the data received by the callable will have the 1st dimension squeezed out - instead of `[["string to split"], ["another string to split"]]`, the Callable will see `["string to split", "another string to split"]`. The callable should return a Tensor with the first dimension containing the split tokens - in this example, we should see something like `[["string", "to", "split"], ["another", "string", "to", "split"]]`. This makes the callable site natively compatible with `tf.strings.split()`.

For an overview and full list of preprocessing layers, see the preprocessing guide.

Arguments

- **max_tokens**: Maximum size of the vocabulary for this layer. This should only be specified when adapting a vocabulary or when setting `pad_to_max_tokens=True`. Note that this vocabulary

contains 1 OOV token, so the effective number of tokens is `(max_tokens - 1 - (1 if output_mode == "int" else 0))`.

- **standardize**: Optional specification for standardization to apply to the input text. Values can be: - `None`: No standardization. - `"lower_and_strip_punctuation"`: Text will be lowercased and all punctuation removed. - `"lower"`: Text will be lowercased. - `"strip_punctuation"`: All punctuation will be removed. - Callable: Inputs will passed to the callable function, which should standardized and returned.
- **split**: Optional specification for splitting the input text. Values can be: - `None`: No splitting. - `"whitespace"`: Split on whitespace. - `"character"`: Split on each unicode character. - Callable: Standardized inputs will passed to the callable function, which should split and returned.
- **ngrams**: Optional specification for ngrams to create from the possibly-split input text. Values can be None, an integer or tuple of integers; passing an integer will create ngrams up to that integer, and passing a tuple of integers will create ngrams for the specified values in the tuple. Passing None means that no ngrams will be created.
- **output_mode**: Optional specification for the output of the layer. Values can be `"int"`, `"multi_hot"`, `"count"` or `"tf_idf"`, configuring the layer as follows: - `"int"`: Outputs integer indices, one integer index per split string token. When `output_mode == "int"`, 0 is reserved for masked locations; this reduces the vocab size to `max_tokens - 2` instead of `max_tokens - 1`. - `"multi_hot"`: Outputs a single int array per batch, of either vocab_size or max_tokens size, containing 1s in all elements where the token mapped to that index exists at least once in the batch item. - `"count"`: Like `"multi_hot"`, but the int array contains a count of the number of times the token at that index appeared in the batch item. - `"tf_idf"`: Like `"multi_hot"`, but the TF-IDF algorithm is applied to find the value in each token slot. For `"int"` output, any shape of input and output is supported. For all other output modes, currently only rank 1 inputs (and rank 2 outputs after splitting) are supported.
- **output_sequence_length**: Only valid in INT mode. If set, the output will have its time dimension padded or truncated to exactly `output_sequence_length` values, resulting in a tensor of shape `(batch_size, output_sequence_length)` regardless of how many tokens resulted from the splitting step. Defaults to None.
- **pad_to_max_tokens**: Only valid in `"multi_hot"`, `"count"`, and `"tf_idf"` modes. If True, the output will have its feature axis padded to `max_tokens` even if the number of unique tokens in the vocabulary is less than max_tokens, resulting in a tensor of shape `(batch_size, max_tokens)` regardless of vocabulary size. Defaults to False.
- **vocabulary**: Optional. Either an array of strings or a string path to a text file. If passing an array, can pass a tuple, list, 1D numpy array, or 1D tensor containing the string vocbulary terms. If passing a file path, the file should contain one line per term in the vocabulary. If this argument is set, there is no need to `adapt()` the layer.
- **idf_weights**: Only valid when `output_mode` is `"tf_idf"`. A tuple, list, 1D numpy array, or 1D tensor or the same length as the vocabulary, containing the floating point inverse document frequency weights, which will be multiplied by per sample term counts for the final `tf_idf` weight. If the `vocabulary` argument is set, and `output_mode` is `"tf_idf"`, this argument must be supplied.
- **ragged**: Boolean. Only applicable to `"int"` output mode. If True, returns a `RaggedTensor` instead of a dense `Tensor`, where each sequence may have a different length after string splitting. Defaults to False.
- **sparse**: Boolean. Only applicable to `"multi_hot"`, `"count"`, and `"tf_idf"` output modes. If True, returns a `SparseTensor` instead of a dense `Tensor`. Defaults to False.

Example

This example instantiates a `TextVectorization` layer that lowercases text, splits on whitespace, strips punctuation, and outputs integer vocab indices.

```
>>> text_dataset = tf.data.Dataset.from_tensor_slices(["foo", "bar", "baz"])
>>> max_features = 5000  # Maximum vocab size.
>>> max_len = 4  # Sequence length to pad the outputs to.
>>>
>>> # Create the layer.
>>> vectorize_layer = tf.keras.layers.TextVectorization(
...  max_tokens=max_features,
...  output_mode='int',
...  output_sequence_length=max_len)
>>>
>>> # Now that the vocab layer has been created, call `adapt` on the text-only
>>> # dataset to create the vocabulary. You don't have to batch, but for large
>>> # datasets this means we're not keeping spare copies of the dataset.
>>> vectorize_layer.adapt(text_dataset.batch(64))
>>>
>>> # Create the model that uses the vectorize text layer
>>> model = tf.keras.models.Sequential()
>>>
>>> # Start by creating an explicit input layer. It needs to have a shape of
>>> # (1,) (because we need to guarantee that there is exactly one string
>>> # input per batch), and the dtype needs to be 'string'.
>>> model.add(tf.keras.Input(shape=(1,), dtype=tf.string))
>>>
>>> # The first layer in our model is the vectorization layer. After this
>>> # layer, we have a tensor of shape (batch_size, max_len) containing vocab
>>> # indices.
>>> model.add(vectorize_layer)
>>>
>>> # Now, the model can map strings to integers, and you can add an embedding
>>> # layer to map these integers to learned embeddings.
>>> input_data = [["foo qux bar"], ["qux baz"]]
>>> model.predict(input_data)
array([[2, 1, 4, 0],
       [1, 3, 0, 0]])
```

Example

This example instantiates a `TextVectorization` layer by passing a list of vocabulary terms to the layer's `__init__()` method.

```
>>> vocab_data = ["earth", "wind", "and", "fire"]
>>> max_len = 4  # Sequence length to pad the outputs to.
>>>
>>> # Create the layer, passing the vocab directly. You can also pass the
>>> # vocabulary arg a path to a file containing one vocabulary word per
>>> # line.
>>> vectorize_layer = tf.keras.layers.TextVectorization(
...  max_tokens=max_features,
...  output_mode='int',
...  output_sequence_length=max_len,
...  vocabulary=vocab_data)
>>>
>>> # Because we've passed the vocabulary directly, we don't need to adapt
>>> # the layer - the vocabulary is already set. The vocabulary contains the
>>> # padding token ('') and OOV token ('[UNK]') as well as the passed tokens.
>>> vectorize_layer.get_vocabulary()
['', '[UNK]', 'earth', 'wind', 'and', 'fire']
```