# Masking and padding with Keras

Run in Google Colab (https://colab.research.google.com/github/tensorflow/docs/blob/snapshot-keras/site/en/guide/keras/masking_and_padding.ipynb)

View source on GitHub (https://github.com/tensorflow/docs/blob/master/site/en/guide/keras/masking_and_padding.ipynb)

## Setup

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

## Introduction

**Masking** is a way to tell sequence-processing layers that certain timesteps in an input are missing, and thus should be skipped when processing the data.

**Padding** is a special form of masking where the masked steps are at the start or the end of a sequence. Padding comes from the need to encode sequence data into contiguous batches: in order to make all sequences in a batch fit a given standard length, it is necessary to pad or truncate some sequences.

Let's take a close look.

## Padding sequence data

When processing sequence data, it is very common for individual samples to have different lengths. Consider the following example (text tokenized as words):

```
[
  ["Hello", "world", "!"],
  ["How", "are", "you", "doing", "today"],
  ["The", "weather", "will", "be", "nice", "tomorrow"],
]
```

After vocabulary lookup, the data might be vectorized as integers, e.g.:

```
[
  [71, 1331, 4231]
  [73, 8, 3215, 55, 927],
  [83, 91, 1, 645, 1253, 927],
]
```

The data is a nested list where individual samples have length 3, 5, and 6, respectively. Since the input data for a deep learning model must be a single tensor (of shape e.g. `(batch_size, 6, vocab_size)` in this case), samples that are shorter than the longest item need to be padded with some placeholder value (alternatively, one might also truncate long samples before padding short samples).

Keras provides a utility function to truncate and pad Python lists to a common length: `tf.keras.preprocessing.sequence.pad_sequences` (https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/sequence/pad_sequences).

```
raw_inputs = [
    [711, 632, 71],
    [73, 8, 3215, 55, 927],
    [83, 91, 1, 645, 1253, 927],
]

# By default, this will pad using 0s; it is configurable via the
# "value" parameter.
# Note that you could "pre" padding (at the beginning) or
# "post" padding (at the end).
# We recommend using "post" padding when working with RNN layers
# (in order to be able to use the
# CuDNN implementation of the layers).
padded_inputs = tf.keras.preprocessing.sequence.pad_sequences(
```

```
    raw_inputs, padding="post"
)
print(padded_inputs)
```

```
[[ 711  632   71    0    0    0]
 [  73    8 3215   55  927    0]
 [  83   91    1  645 1253  927]]
```

## Masking

Now that all samples have a uniform length, the model must be informed that some part of the data is actually padding and should be ignored. That mechanism is **masking**.

There are three ways to introduce input masks in Keras models:

- Add a `keras.layers.Masking` (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Masking) layer.
- Configure a `keras.layers.Embedding` (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Embedding) layer with `mask_zero=True`.
- Pass a `mask` argument manually when calling layers that support this argument (e.g. RNN layers).

## Mask-generating layers: `Embedding` and `Masking`

Under the hood, these layers will create a mask tensor (2D tensor with shape `(batch, sequence_length)`), and attach it to the tensor output returned by the `Masking` or `Embedding` layer.

```
embedding = layers.Embedding(input_dim=5000, output_dim=16, mask_zero=True)
masked_output = embedding(padded_inputs)

print(masked_output._keras_mask)
```

```
masking_layer = layers.Masking()
# Simulate the embedding lookup by expanding the 2D input to 3D,
# with embedding dimension of 10.
unmasked_embedding = tf.cast(
    tf.tile(tf.expand_dims(padded_inputs, axis=-1), [1, 1, 10]), tf.float32
)

masked_embedding = masking_layer(unmasked_embedding)
print(masked_embedding._keras_mask)
```

```
tf.Tensor(
[[ True  True  True False False False]
 [ True  True  True  True  True False]
 [ True  True  True  True  True  True]], shape=(3, 6), dtype=bool)
tf.Tensor(
[[ True  True  True False False False]
 [ True  True  True  True  True False]
 [ True  True  True  True  True  True]], shape=(3, 6), dtype=bool)
```

As you can see from the printed result, the mask is a 2D boolean tensor with shape `(batch_size, sequence_length)`, where each individual `False` entry indicates that the corresponding timestep should be ignored during processing.

## Mask propagation in the Functional API and Sequential API

When using the Functional API or the Sequential API, a mask generated by an `Embedding` or `Masking` layer will be propagated through the network for any layer that is capable of using them (for example, RNN layers). Keras will automatically fetch the mask corresponding to an input and pass it to any layer that knows how to use it.

For instance, in the following Sequential model, the `LSTM` layer will automatically receive a mask, which means it will ignore padded values:

```
model = keras.Sequential(
    [layers.Embedding(input_dim=5000, output_dim=16, mask_zero=True), layers.LST
```

```
)
```

This is also the case for the following Functional API model:

```
inputs = keras.Input(shape=(None,), dtype="int32")
x = layers.Embedding(input_dim=5000, output_dim=16, mask_zero=True)(inputs)
outputs = layers.LSTM(32)(x)

model = keras.Model(inputs, outputs)
```

## Passing mask tensors directly to layers

Layers that can handle masks (such as the `LSTM` layer) have a `mask` argument in their `__call__` method.

Meanwhile, layers that produce a mask (e.g. `Embedding`) expose a `compute_mask(input, previous_mask)` method which you can call.

Thus, you can pass the output of the `compute_mask()` method of a mask-producing layer to the `__call__` method of a mask-consuming layer, like this:

```
class MyLayer(layers.Layer):
    def __init__(self, **kwargs):
        super(MyLayer, self).__init__(**kwargs)
        self.embedding = layers.Embedding(input_dim=5000, output_dim=16, mask_ze
        self.lstm = layers.LSTM(32)

    def call(self, inputs):
        x = self.embedding(inputs)
        # Note that you could also prepare a `mask` tensor manually.
        # It only needs to be a boolean tensor
        # with the right shape, i.e. (batch_size, timesteps).
        mask = self.embedding.compute_mask(inputs)
        output = self.lstm(x, mask=mask)  # The layer will ignore the masked val
        return output
```

```
layer = MyLayer()
x = np.random.random((32, 10)) * 100
x = x.astype("int32")
layer(x)
```

```
<tf.Tensor: shape=(32, 32), dtype=float32, numpy=
array([[-3.6287602e-04,  8.8942451e-03, -4.5623952e-03, ...,
         3.6509466e-04, -4.3871473e-03, -1.7532009e-03],
       [ 2.6261162e-03, -2.5420082e-03,  7.6517118e-03, ...,
         5.8210879e-03, -1.5617531e-03, -1.7562184e-03],
       [ 6.8687932e-03,  1.2330032e-03, -1.2028826e-02, ...,
         2.0486799e-03,  5.7172528e-03,  2.6641595e-03],
       ...,
       [-3.4327951e-04,  1.3967649e-03, -1.2102776e-02, ...,
         3.8406218e-03, -2.3374180e-03, -4.9669710e-03],
       [-2.3023323e-03,  1.8474255e-03,  2.7329330e-05, ...,
         6.1798934e-03,  4.2709545e-04,  3.9026213e-03],
       [ 7.4090287e-03,  1.9879336e-03, -2.0261200e-03, ...,
         8.2100276e-03,  8.7051848e-03,  9.9167246e-03]], dtype=float32)>
```

## Supporting masking in your custom layers

Sometimes, you may need to write layers that generate a mask (like `Embedding`), or layers that need to modify the current mask.

For instance, any layer that produces a tensor with a different time dimension than its input, such as a `Concatenate` layer that concatenates on the time dimension, will need to modify the current mask so that downstream layers will be able to properly take masked timesteps into account.

To do this, your layer should implement the `layer.compute_mask()` method, which produces a new mask given the input and the current mask.

Here is an example of a `TemporalSplit` layer that needs to modify the current mask.

```
class TemporalSplit(keras.layers.Layer):
    """Split the input tensor into 2 tensors along the time dimension."""
```

```
    def call(self, inputs):
        # Expect the input to be 3D and mask to be 2D, split the input tensor in
        # subtensors along the time axis (axis 1).
        return tf.split(inputs, 2, axis=1)

    def compute_mask(self, inputs, mask=None):
        # Also split the mask into 2 if it presents.
        if mask is None:
            return None
        return tf.split(mask, 2, axis=1)


first_half, second_half = TemporalSplit()(masked_embedding)
print(first_half._keras_mask)
print(second_half._keras_mask)
```

```
tf.Tensor(
[[ True  True  True]
 [ True  True  True]
 [ True  True  True]], shape=(3, 3), dtype=bool)
tf.Tensor(
[[False False False]
 [ True  True False]
 [ True  True  True]], shape=(3, 3), dtype=bool)
```

Here is another example of a `CustomEmbedding` layer that is capable of generating a mask from input values:

```
class CustomEmbedding(keras.layers.Layer):
    def __init__(self, input_dim, output_dim, mask_zero=False, **kwargs):
        super(CustomEmbedding, self).__init__(**kwargs)
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.mask_zero = mask_zero

    def build(self, input_shape):
        self.embeddings = self.add_weight(
            shape=(self.input_dim, self.output_dim),
            initializer="random_normal",
            dtype="float32",
```

```
        )

    def call(self, inputs):
        return tf.nn.embedding_lookup(self.embeddings, inputs)

    def compute_mask(self, inputs, mask=None):
        if not self.mask_zero:
            return None
        return tf.not_equal(inputs, 0)


layer = CustomEmbedding(10, 32, mask_zero=True)
x = np.random.random((3, 10)) * 9
x = x.astype("int32")

y = layer(x)
mask = layer.compute_mask(x)

print(mask)
```

```
tf.Tensor(
[[ True  True  True  True  True  True  True  True  True  True]
 [ True  True  True  True  True False  True  True  True  True]
 [ True  True  True  True  True  True  True  True False  True]], shape=(3, 10),
```

## Opting-in to mask propagation on compatible layers

Most layers don't modify the time dimension, so don't need to modify the current mask. However, they may still want to be able to **propagate** the current mask, unchanged, to the next layer. **This is an opt-in behavior.** By default, a custom layer will destroy the current mask (since the framework has no way to tell whether propagating the mask is safe to do).

If you have a custom layer that does not modify the time dimension, and if you want it to be able to propagate the current input mask, you should set `self.supports_masking = True` in the layer constructor. In this case, the default behavior of `compute_mask()` is to just pass the current mask through.

Here's an example of a layer that is whitelisted for mask propagation:

```
class MyActivation(keras.layers.Layer):
    def __init__(self, **kwargs):
        super(MyActivation, self).__init__(**kwargs)
        # Signal that the layer is safe for mask propagation
        self.supports_masking = True

    def call(self, inputs):
        return tf.nn.relu(inputs)
```

You can now use this custom layer in-between a mask-generating layer (like `Embedding`) and a mask-consuming layer (like `LSTM`), and it will pass the mask along so that it reaches the mask-consuming layer.

```
inputs = keras.Input(shape=(None,), dtype="int32")
x = layers.Embedding(input_dim=5000, output_dim=16, mask_zero=True)(inputs)
x = MyActivation()(x)  # Will pass the mask along
print("Mask found:", x._keras_mask)
outputs = layers.LSTM(32)(x)  # Will receive the mask

model = keras.Model(inputs, outputs)
```

```
Mask found: KerasTensor(type_spec=TensorSpec(shape=(None, None), dtype=tf.bool,
```

## Writing layers that need mask information

Some layers are mask *consumers*: they accept a `mask` argument in `call` and use it to determine whether to skip certain time steps.

To write such a layer, you can simply add a `mask=None` argument in your `call` signature. The mask associated with the inputs will be passed to your layer whenever it is available.

Here's a simple example below: a layer that computes a softmax over the time dimension (axis 1) of an input sequence, while discarding masked timesteps.

```
class TemporalSoftmax(keras.layers.Layer):
    def call(self, inputs, mask=None):
        broadcast_float_mask = tf.expand_dims(tf.cast(mask, "float32"), -1)
        inputs_exp = tf.exp(inputs) * broadcast_float_mask
        inputs_sum = tf.reduce_sum(
            inputs_exp * broadcast_float_mask, axis=-1, keepdims=True
        )
        return inputs_exp / inputs_sum


inputs = keras.Input(shape=(None,), dtype="int32")
x = layers.Embedding(input_dim=10, output_dim=32, mask_zero=True)(inputs)
x = layers.Dense(1)(x)
outputs = TemporalSoftmax()(x)

model = keras.Model(inputs, outputs)
y = model(np.random.randint(0, 10, size=(32, 100)), np.random.random((32, 100, 1
```

## Summary

That is all you need to know about padding & masking in Keras. To recap:

- "Masking" is how layers are able to know when to skip / ignore certain timesteps in sequence inputs.
- Some layers are mask-generators: `Embedding` can generate a mask from input values (if `mask_zero=True`), and so can the `Masking` layer.
- Some layers are mask-consumers: they expose a `mask` argument in their `__call__` method. This is the case for RNN layers.
- In the Functional API and Sequential API, mask information is propagated automatically.
- When using layers in a standalone way, you can pass the `mask` arguments to layers manually.
- You can easily write layers that modify the current mask, that generate a new mask, or that consume the mask associated with the inputs.

Last updated 2022-01-10 UTC.