Metrics

A metric is a function that is used to judge the performance of your model.

Metric functions are similar to loss functions, except that the results from evaluating a metric are not used when training the model. Note that you may use any loss function as a metric.

Available metrics

Accuracy metrics

- Accuracy class
- BinaryAccuracy class
- CategoricalAccuracy class
- TopKCategoricalAccuracy class
- SparseTopKCategoricalAccuracy class

Probabilistic metrics

- BinaryCrossentropy class
- CategoricalCrossentropy class
- SparseCategoricalCrossentropy class
- KLDivergence class
- Poisson class

Regression metrics

- MeanSquaredError class
- RootMeanSquaredError class
- MeanAbsoluteError class
- MeanAbsolutePercentageError class
- MeanSquaredLogarithmicError class
- CosineSimilarity class
- LogCoshError class

Classification metrics based on True/False positives & negatives

- AUC class
- Precision class
- Recall class
- TruePositives class
- TrueNegatives class
- FalsePositives class
- FalseNegatives class
- PrecisionAtRecall class
- SensitivityAtSpecificity class
- SpecificityAtSensitivity class

Image segmentation metrics

- MeanIoU class

Hinge metrics for "maximum-margin" classification

- Hinge class
- SquaredHinge class
- CategoricalHinge class

Usage with `compile()` & `fit()`

The `compile()` method takes a `metrics` argument, which is a list of metrics:

```
model.compile(
    optimizer='adam',
    loss='mean_squared_error',
    metrics=[
        metrics.MeanSquaredError(),
        metrics.AUC(),
    ]
)
```

Metric values are displayed during `fit()` and logged to the `History` object returned by `fit()`. They are also returned by `model.evaluate()`.

Note that the best way to monitor your metrics during training is via TensorBoard.

To track metrics under a specific name, you can pass the `name` argument to the metric constructor:

```
model.compile(
    optimizer='adam',
    loss='mean_squared_error',
    metrics=[
        metrics.MeanSquaredError(name='my_mse'),
        metrics.AUC(name='my_auc'),
    ]
)
```

All built-in metrics may also be passed via their string identifier (in this case, default constructor argument values are used, including a default metric name):

```
model.compile(
    optimizer='adam',
    loss='mean_squared_error',
    metrics=[
        'MeanSquaredError',
        'AUC',
    ]
)
```

Standalone usage

Unlike losses, metrics are stateful. You update their state using the `update_state()` method, and you query the scalar metric result using the `result()` method:

```
m = tf.keras.metrics.AUC()
m.update_state([0, 1, 1, 1], [0, 1, 0, 0])
print('Intermediate result:', float(m.result()))

m.update_state([1, 1, 1, 1], [0, 1, 1, 0])
print('Final result:', float(m.result()))
```

The internal state can be cleared via `metric.reset_states()`.

Here's how you would use a metric as part of a simple custom training loop:

```python
accuracy = tf.keras.metrics.CategoricalAccuracy()
loss_fn = tf.keras.losses.CategoricalCrossentropy(from_logits=True)
optimizer = tf.keras.optimizers.Adam()

# Iterate over the batches of a dataset.
for step, (x, y) in enumerate(dataset):
    with tf.GradientTape() as tape:
        logits = model(x)
        # Compute the loss value for this batch.
        loss_value = loss_fn(y, logits)

    # Update the state of the `accuracy` metric.
    accuracy.update_state(y, logits)

    # Update the weights of the model to minimize the loss value.
    gradients = tape.gradient(loss_value, model.trainable_weights)
    optimizer.apply_gradients(zip(gradients, model.trainable_weights))

    # Logging the current accuracy value so far.
    if step % 100 == 0:
        print('Step:', step)
        print('Total running accuracy so far: %.3f' % accuracy.result())
```

Creating custom metrics

As simple callables (stateless)

Much like loss functions, any callable with signature `metric_fn(y_true, y_pred)` that returns an array of losses (one of sample in the input batch) can be passed to `compile()` as a metric. Note that sample weighting is automatically supported for any such metric.

Here's a simple example:

```python
def my_metric_fn(y_true, y_pred):
    squared_difference = tf.square(y_true - y_pred)
    return tf.reduce_mean(squared_difference, axis=-1)  # Note the `axis=-1`

model.compile(optimizer='adam', loss='mean_squared_error', metrics=[my_metric_fn])
```

In this case, the scalar metric value you are tracking during training and evaluation is the average of the per-batch metric values for all batches see during a given epoch (or during a given call to `model.evaluate()`).

As subclasses of `Metric` (stateful)

Not all metrics can be expressed via stateless callables, because metrics are evaluated for each batch during training and evaluation, but in some cases the average of the per-batch values is not what you are interested in.

Let's say that you want to compute AUC over a given evaluation dataset: the average of the per-batch AUC values isn't the same as the AUC over the entire dataset.

For such metrics, you're going to want to subclass the `Metric` class, which can maintain a state across batches. It's easy:

- Create the state variables in `__init__`
- Update the variables given `y_true` and `y_pred` in `update_state()`
- Return the metric result in `result()`
- Clear the state in `reset_states()`

Here's a simple example computing binary true positives:

```
class BinaryTruePositives(tf.keras.metrics.Metric):

  def __init__(self, name='binary_true_positives', **kwargs):
    super(BinaryTruePositives, self).__init__(name=name, **kwargs)
    self.true_positives = self.add_weight(name='tp', initializer='zeros')

  def update_state(self, y_true, y_pred, sample_weight=None):
    y_true = tf.cast(y_true, tf.bool)
    y_pred = tf.cast(y_pred, tf.bool)

    values = tf.logical_and(tf.equal(y_true, True), tf.equal(y_pred, True))
    values = tf.cast(values, self.dtype)
    if sample_weight is not None:
      sample_weight = tf.cast(sample_weight, self.dtype)
      values = tf.multiply(values, sample_weight)
    self.true_positives.assign_add(tf.reduce_sum(values))

  def result(self):
    return self.true_positives

  def reset_states(self):
    self.true_positives.assign(0)

m = BinaryTruePositives()
m.update_state([0, 1, 1, 1], [0, 1, 0, 0])
print('Intermediate result:', float(m.result()))

m.update_state([1, 1, 1, 1], [0, 1, 1, 0])
print('Final result:', float(m.result()))
```

The `add_metric()` API

When writing the forward pass of a custom layer or a subclassed model, you may sometimes want to log certain quantities on the fly, as metrics. In such cases, you can use the `add_metric()` method.

Let's say you want to log as metric the mean of the activations of a Dense-like custom layer. You could do the following:

```
class DenseLike(Layer):
  """y = w.x + b"""

  ...

  def call(self, inputs):
      output = tf.matmul(inputs, self.w) + self.b
      self.add_metric(tf.reduce_mean(output), aggregation='mean', name='activation_mean')
      return output
```

The quantity will then tracked under the name "activation_mean". The value tracked will be the average of the per-batch metric metric values (as specified by `aggregation='mean'`).

See [the `add_metric()` documentation] for more details.