Copyright by Pierian Data Inc.
Created by Jose Marcial Portilla.

Keras Syntax Basics

With TensorFlow 2.0 , Keras is now the main API choice. Let's work through a simple regression project to understand the basics of the Keras syntax and adding layers.

The Data

To learn the basic syntax of Keras, we will use a very simple fake data set, in the subsequent lectures we will focus on real datasets, along with feature engineering! For now, let's focus on the syntax of TensorFlow 2.0.

Let's pretend this data are measurements of some rare gem stones, with 2 measurement features and a sale price. Our final goal would be to try to predict the sale price of a new gem stone we just mined from the ground, in order to try to set a fair price in the market.

Load the Data

In [1]:
```
import pandas as pd
```

In [2]:
```
df = pd.read_csv('../DATA/fake_reg.csv')
```

In [3]:
```
df.head()
```

Out[3]:

	price	feature1	feature2
0	461.527929	999.787558	999.766096
1	548.130011	998.861615	1001.042403
2	410.297162	1000.070267	998.844015
3	540.382220	999.952251	1000.440940
4	546.024553	1000.446011	1000.338531

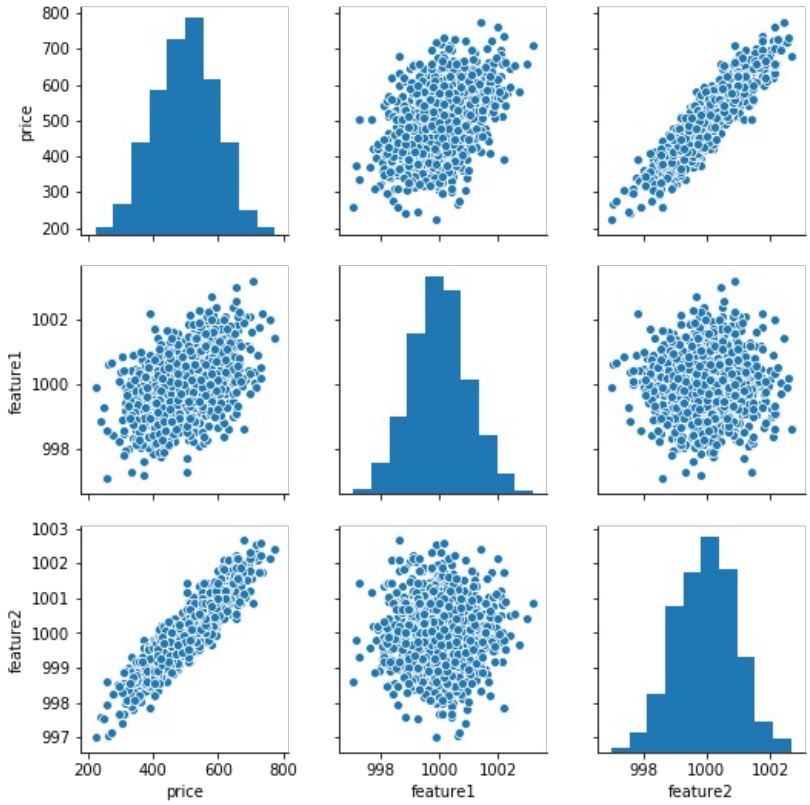
Explore the data

Let's take a quick look, we should see strong correlation between the features and the "price" of this made up product.

In [4]:
```
import seaborn as sns
import matplotlib.pyplot as plt
```

In [5]:
```
sns.pairplot(df)
```

Out[5]:
```
<seaborn.axisgrid.PairGrid at 0x259a3d72188>
```



Feel free to visualize more, but this data is fake, so we will focus on feature engineering and exploratory data analysis later on in the course in much more detail!

Test/Train Split

In [6]:
```
from sklearn.model_selection import train_test_split
```

In [7]:
```
# Convert Pandas to Numpy for Keras

# Features
X = df[['feature1','feature2']].values

# Label
y = df['price'].values

# Split
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.3,random_state=42)
```

In [8]:
```
X_train.shape
```

Out[8]:
```
(700, 2)
```

In [9]:
```
X_test.shape
```

Out[9]:
```
(300, 2)
```

In [10]:
```
y_train.shape
```

Out[10]:
```
(700,)
```

In [11]:
```
y_test.shape
```

Out[11]:
```
(300,)
```

Normalizing/Scaling the Data

We scale the feature data.

Why we don't need to scale the label

In [12]:
```
from sklearn.preprocessing import MinMaxScaler
```

In [13]:
```
help(MinMaxScaler)
```

```
Help on class MinMaxScaler in module sklearn.preprocessing.data:

class MinMaxScaler(sklearn.base.BaseEstimator, sklearn.base.TransformerMixin)
 |  MinMaxScaler(feature_range=(0, 1), copy=True)
 |
 |  Transforms features by scaling each feature to a given range.
 |
 |  This estimator scales and translates each feature individually such
 |  that it is in the given range on the training set, e.g. between
 |  zero and one.
 |
 |  The transformation is given by::
 |
 |      X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
 |      X_scaled = X_std * (max - min) + min
 |
 |  where min, max = feature_range.
 |
 |  The transformation is calculated as::
 |
 |      X_scaled = scale * X + min - X.min(axis=0) * scale
 |      where scale = (max - min) / (X.max(axis=0) - X.min(axis=0))
 |
 |  This transformation is often used as an alternative to zero mean,
 |  unit variance scaling.
 |
 |  Read more in the :ref:`User Guide <preprocessing_scaler>`.
 |
 |  Parameters
 |  ----------
 |  feature_range : tuple (min, max), default=(0, 1)
 |      Desired range of transformed data.
 |
 |  copy : boolean, optional, default True
 |      Set to False to perform inplace row normalization and avoid a
 |      copy (if the input is already a numpy array).
 |
 |  Attributes
 |  ----------
 |  min_ : ndarray, shape (n_features,)
 |      Per feature adjustment for minimum. Equivalent to
 |      ``min - X.min(axis=0) * self.scale_``
 |
 |  scale_ : ndarray, shape (n_features,)
 |      Per feature relative scaling of the data. Equivalent to
 |      ``(max - min) / (X.max(axis=0) - X.min(axis=0))``
 |
 |      .. versionadded:: 0.17
 |         *scale_* attribute.
 |
 |  data_min_ : ndarray, shape (n_features,)
 |      Per feature minimum seen in the data
 |
 |      .. versionadded:: 0.17
 |         *data_min_*
 |
 |  data_max_ : ndarray, shape (n_features,)
 |      Per feature maximum seen in the data
 |
 |      .. versionadded:: 0.17
 |         *data_max_*
 |
 |  data_range_ : ndarray, shape (n_features,)
 |      Per feature range ``(data_max_ - data_min_)`` seen in the data
```

```
|
|      .. versionadded:: 0.17
|         *data_range_*
|
|  Examples
|  --------
|  >>> from sklearn.preprocessing import MinMaxScaler
|  >>> data = [[-1, 2], [-0.5, 6], [0, 10], [1, 18]]
|  >>> scaler = MinMaxScaler()
|  >>> print(scaler.fit(data))
|  MinMaxScaler(copy=True, feature_range=(0, 1))
|  >>> print(scaler.data_max_)
|  [ 1. 18.]
|  >>> print(scaler.transform(data))
|  [[0.   0.  ]
|   [0.25 0.25]
|   [0.5  0.5 ]
|   [1.   1.  ]]
|  >>> print(scaler.transform([[2, 2]]))
|  [[1.5 0. ]]
|
|  See also
|  --------
|  minmax_scale: Equivalent function without the estimator API.
|
|  Notes
|  -----
|  NaNs are treated as missing values: disregarded in fit, and maintained in
|  transform.
|
|  For a comparison of the different scalers, transformers, and normalizers,
|  see :ref:`examples/preprocessing/plot_all_scaling.py
|  <sphx_glr_auto_examples_preprocessing_plot_all_scaling.py>`.
|
|  Method resolution order:
|      MinMaxScaler
|      sklearn.base.BaseEstimator
|      sklearn.base.TransformerMixin
|      builtins.object
|
|  Methods defined here:
|
|  __init__(self, feature_range=(0, 1), copy=True)
|      Initialize self.  See help(type(self)) for accurate signature.
|
|  fit(self, X, y=None)
|      Compute the minimum and maximum to be used for later scaling.
|
|      Parameters
|      ----------
|      X : array-like, shape [n_samples, n_features]
|          The data used to compute the per-feature minimum and maximum
|          used for later scaling along the features axis.
|
|  inverse_transform(self, X)
|      Undo the scaling of X according to feature_range.
|
|      Parameters
|      ----------
|      X : array-like, shape [n_samples, n_features]
|          Input data that will be transformed. It cannot be sparse.
|
|  partial_fit(self, X, y=None)
|      Online computation of min and max on X for later scaling.
|      All of X is processed as a single batch. This is intended for cases
|      when `fit` is not feasible due to very large number of `n_samples`
|      or because X is read from a continuous stream.
|
|      Parameters
|      ----------
|      X : array-like, shape [n_samples, n_features]
|          The data used to compute the mean and standard deviation
|          used for later scaling along the features axis.
|
|      y
|          Ignored
|
|  transform(self, X)
|      Scaling features of X according to feature_range.
|
|      Parameters
|      ----------
|      X : array-like, shape [n_samples, n_features]
|          Input data that will be transformed.
|
|  ----------------------------------------------------------------------
|  Methods inherited from sklearn.base.BaseEstimator:
|
|  __getstate__(self)
```

```
|
|  __repr__(self, N_CHAR_MAX=700)
|      Return repr(self).
|
|  __setstate__(self, state)
|
|  get_params(self, deep=True)
|      Get parameters for this estimator.
|
|      Parameters
|      ----------
|      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.
|
|  set_params(self, **params)
|      Set the parameters of this estimator.
|
|      The method works on simple estimators as well as on nested objects
|      (such as pipelines). The latter have parameters of the form
|      ``<component>__<parameter>`` so that it's possible to update each
|      component of a nested object.
|
|      Returns
|      -------
|      self
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from sklearn.base.BaseEstimator:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
|
|  ----------------------------------------------------------------------
|  Methods inherited from sklearn.base.TransformerMixin:
|
|  fit_transform(self, X, y=None, **fit_params)
|      Fit to data, then transform it.
|
|      Fits transformer to X and y with optional parameters fit_params
|      and returns a transformed version of X.
|
|      Parameters
|      ----------
|      X : numpy array of shape [n_samples, n_features]
|          Training set.
|
|      y : numpy array of shape [n_samples]
|          Target values.
|
|      Returns
|      -------
|      X_new : numpy array of shape [n_samples, n_features_new]
|          Transformed array.
```

In [14]:
```
scaler = MinMaxScaler()
```

In [15]:
```
# Notice to prevent data leakage from the test set, we only fit our scaler to the training set
```

In [16]:
```
scaler.fit(X_train)
```

Out[16]:
```
MinMaxScaler(copy=True, feature_range=(0, 1))
```

In [17]:
```
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

TensorFlow 2.0 Syntax

Import Options

There are several ways you can import Keras from Tensorflow (this is hugely a personal style choice, please use any import methods you prefer). We will use the method shown in the **official TF documentation**.

In [18]:
```
import tensorflow as tf
```

In [19]:
```
from tensorflow.keras.models import Sequential
```

In [20]:
```
help(Sequential)
```

```
Help on class Sequential in module tensorflow.python.keras.engine.sequential:

class Sequential(tensorflow.python.keras.engine.training.Model)
 |  Sequential(layers=None, name=None)
 |
 |  Linear stack of layers.
 |
 |  Arguments:
 |      layers: list of layers to add to the model.
 |
 |  Example:
 |
 |  ```python
 |  # Optionally, the first layer can receive an `input_shape` argument:
 |  model = Sequential()
 |  model.add(Dense(32, input_shape=(500,)))
 |  # Afterwards, we do automatic shape inference:
 |  model.add(Dense(32))
 |
 |  # This is identical to the following:
 |  model = Sequential()
 |  model.add(Dense(32, input_dim=500))
 |
 |  # And to the following:
 |  model = Sequential()
 |  model.add(Dense(32, batch_input_shape=(None, 500)))
 |
 |  # Note that you can also omit the `input_shape` argument:
 |  # In that case the model gets built the first time you call `fit` (or other
 |  # training and evaluation methods).
 |  model = Sequential()
 |  model.add(Dense(32))
 |  model.add(Dense(32))
 |  model.compile(optimizer=optimizer, loss=loss)
 |  # This builds the model for the first time:
 |  model.fit(x, y, batch_size=32, epochs=10)
 |
 |  # Note that when using this delayed-build pattern (no input shape specified),
 |  # the model doesn't have any weights until the first call
 |  # to a training/evaluation method (since it isn't yet built):
 |  model = Sequential()
 |  model.add(Dense(32))
 |  model.add(Dense(32))
 |  model.weights  # returns []
 |
 |  # Whereas if you specify the input shape, the model gets built continuously
 |  # as you are adding layers:
 |  model = Sequential()
 |  model.add(Dense(32, input_shape=(500,)))
 |  model.add(Dense(32))
 |  model.weights  # returns list of length 4
 |
 |  # When using the delayed-build pattern (no input shape specified), you can
 |  # choose to manually build your model by calling `build(batch_input_shape)`:
 |  model = Sequential()
 |  model.add(Dense(32))
 |  model.add(Dense(32))
 |  model.build((None, 500))
 |  model.weights  # returns list of length 4
 |  ```
 |
 |  Method resolution order:
 |      Sequential
 |      tensorflow.python.keras.engine.training.Model
 |      tensorflow.python.keras.engine.network.Network
 |      tensorflow.python.keras.engine.base_layer.Layer
 |      tensorflow.python.module.module.Module
 |      tensorflow.python.training.tracking.tracking.AutoTrackable
 |      tensorflow.python.training.tracking.base.Trackable
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, layers=None, name=None)
 |
 |  add(self, layer)
 |      Adds a layer instance on top of the layer stack.
 |
 |      Arguments:
 |          layer: layer instance.
 |
 |      Raises:
 |          TypeError: If `layer` is not a layer instance.
```

```
|          ValueError: In case the `layer` argument does not
|              know its input shape.
|          ValueError: In case the `layer` argument has
|              multiple output tensors, or is already connected
|              somewhere else (forbidden in `Sequential` models).
|
|  build(self, input_shape=None)
|      Builds the model based on input shapes received.
|
|      This is to be used for subclassed models, which do not know at instantiation
|      time what their inputs look like.
|
|      This method only exists for users who want to call `model.build()` in a
|      standalone way (as a substitute for calling the model on real data to
|      build it). It will never be called by the framework (and thus it will
|      never throw unexpected errors in an unrelated workflow).
|
|      Args:
|       input_shape: Single tuple, TensorShape, or list of shapes, where shapes
|           are tuples, integers, or TensorShapes.
|
|      Raises:
|        ValueError:
|          1. In case of invalid user-provided data (not of type tuple,
|             list, or TensorShape).
|          2. If the model requires call arguments that are agnostic
|             to the input shapes (positional or kwarg in call signature).
|          3. If not all layers were properly built.
|          4. If float type inputs are not supported within the layers.
|
|        In each of these cases, the user should build their model by calling it
|        on real tensor data.
|
|  call(self, inputs, training=None, mask=None)
|      Calls the model on new inputs.
|
|      In this case `call` just reapplies
|      all ops in the graph to the new inputs
|      (e.g. build a new computational graph from the provided inputs).
|
|      Arguments:
|          inputs: A tensor or list of tensors.
|          training: Boolean or boolean scalar tensor, indicating whether to run
|            the `Network` in training mode or inference mode.
|          mask: A mask or list of masks. A mask can be
|              either a tensor or None (no mask).
|
|      Returns:
|          A tensor if there is a single output, or
|          a list of tensors if there are more than one outputs.
|
|  compute_mask(self, inputs, mask)
|      Computes an output mask tensor.
|
|      Arguments:
|          inputs: Tensor or list of tensors.
|          mask: Tensor or list of tensors.
|
|      Returns:
|          None or a tensor (or list of tensors,
|              one per output tensor of the layer).
|
|  compute_output_shape(self, input_shape)
|      Computes the output shape of the layer.
|
|      If the layer has not been built, this method will call `build` on the
|      layer. This assumes that the layer will later be used with inputs that
|      match the input shape provided here.
|
|      Arguments:
|          input_shape: Shape tuple (tuple of integers)
|              or list of shape tuples (one per output tensor of the layer).
|              Shape tuples can include None for free dimensions,
|              instead of an integer.
|
|      Returns:
|          An input shape tuple.
|
|  get_config(self)
|      Returns the config of the layer.
|
|      A layer config is a Python dictionary (serializable)
|      containing the configuration of a layer.
|      The same layer can be reinstantiated later
|      (without its trained weights) from this configuration.
|
|      The config of a layer does not include connectivity
|      information, nor the layer class name. These are handled
|      by `Network` (one layer of abstraction above).
```

```
|
|      Returns:
|          Python dictionary.
|
|  pop(self)
|      Removes the last layer in the model.
|
|      Raises:
|          TypeError: if there are no layers in the model.
|
|  predict_classes(self, x, batch_size=32, verbose=0)
|      Generate class predictions for the input samples.
|
|      The input samples are processed batch by batch.
|
|      Arguments:
|          x: input data, as a Numpy array or list of Numpy arrays
|              (if the model has multiple inputs).
|          batch_size: integer.
|          verbose: verbosity mode, 0 or 1.
|
|      Returns:
|          A numpy array of class predictions.
|
|  predict_proba(self, x, batch_size=32, verbose=0)
|      Generates class probability predictions for the input samples.
|
|      The input samples are processed batch by batch.
|
|      Arguments:
|          x: input data, as a Numpy array or list of Numpy arrays
|              (if the model has multiple inputs).
|          batch_size: integer.
|          verbose: verbosity mode, 0 or 1.
|
|      Returns:
|          A Numpy array of probability predictions.
|
|  ----------------------------------------------------------------------
|  Class methods defined here:
|
|  from_config(config, custom_objects=None) from builtins.type
|      Instantiates a Model from its config (output of `get_config()`).
|
|      Arguments:
|          config: Model config dictionary.
|          custom_objects: Optional dictionary mapping names
|              (strings) to custom classes or functions to be
|              considered during deserialization.
|
|      Returns:
|          A model instance.
|
|      Raises:
|          ValueError: In case of improperly formatted config dict.
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  dynamic
|
|  input_spec
|      Gets the network's input specs.
|
|      Returns:
|          A list of `InputSpec` instances (one per input to the model)
|              or a single instance if the model has only one input.
|
|  layers
|
|  ----------------------------------------------------------------------
|  Methods inherited from tensorflow.python.keras.engine.training.Model:
|
|  compile(self, optimizer='rmsprop', loss=None, metrics=None, loss_weights=None, sample_weight_mode=None, wei
ghted_metrics=None, target_tensors=None, distribute=None, **kwargs)
|      Configures the model for training.
|
|      Arguments:
|          optimizer: String (name of optimizer) or optimizer instance.
|              See `tf.keras.optimizers`.
|          loss: String (name of objective function), objective function or
|              `tf.losses.Loss` instance. See `tf.losses`. If the model has
|              multiple outputs, you can use a different loss on each output by
|              passing a dictionary or a list of losses. The loss value that will
|              be minimized by the model will then be the sum of all individual
|              losses.
|          metrics: List of metrics to be evaluated by the model during training
|              and testing. Typically you will use `metrics=['accuracy']`.
|              To specify different metrics for different outputs of a
```

```
 |              multi-output model, you could also pass a dictionary, such as
 |              `metrics={'output_a': 'accuracy', 'output_b': ['accuracy', 'mse']}`.
 |              You can also pass a list (len = len(outputs)) of lists of metrics
 |              such as `metrics=[['accuracy'], ['accuracy', 'mse']]` or
 |              `metrics=['accuracy', ['accuracy', 'mse']]`.
 |          loss_weights: Optional list or dictionary specifying scalar
 |              coefficients (Python floats) to weight the loss contributions
 |              of different model outputs.
 |              The loss value that will be minimized by the model
 |              will then be the *weighted sum* of all individual losses,
 |              weighted by the `loss_weights` coefficients.
 |              If a list, it is expected to have a 1:1 mapping
 |              to the model's outputs. If a tensor, it is expected to map
 |              output names (strings) to scalar coefficients.
 |          sample_weight_mode: If you need to do timestep-wise
 |              sample weighting (2D weights), set this to `"temporal"`.
 |              `None` defaults to sample-wise weights (1D).
 |              If the model has multiple outputs, you can use a different
 |              `sample_weight_mode` on each output by passing a
 |              dictionary or a list of modes.
 |          weighted_metrics: List of metrics to be evaluated and weighted
 |              by sample_weight or class_weight during training and testing.
 |          target_tensors: By default, Keras will create placeholders for the
 |              model's target, which will be fed with the target data during
 |              training. If instead you would like to use your own
 |              target tensors (in turn, Keras will not expect external
 |              Numpy data for these targets at training time), you
 |              can specify them via the `target_tensors` argument. It can be
 |              a single tensor (for a single-output model), a list of tensors,
 |              or a dict mapping output names to target tensors.
 |          distribute: NOT SUPPORTED IN TF 2.0, please create and compile the
 |              model under distribution strategy scope instead of passing it to
 |              compile.
 |          **kwargs: Any additional arguments.
 |
 |      Raises:
 |          ValueError: In case of invalid arguments for
 |              `optimizer`, `loss`, `metrics` or `sample_weight_mode`.
 |
 |  evaluate(self, x=None, y=None, batch_size=None, verbose=1, sample_weight=None, steps=None, callbacks=None,
max_queue_size=10, workers=1, use_multiprocessing=False)
 |      Returns the loss value & metrics values for the model in test mode.
 |
 |      Computation is done in batches.
 |
 |      Arguments:
 |          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.
 |            - A generator or `keras.utils.Sequence` instance.
 |          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, generator or
 |            `keras.utils.Sequence` instance, `y` should not be specified (since
 |            targets will be obtained from the iterator/dataset).
 |          batch_size: Integer or `None`.
 |              Number of samples per gradient update.
 |              If unspecified, `batch_size` will default to 32.
 |              Do not specify the `batch_size` is your data is in the
 |              form of symbolic tensors, dataset,
 |              generators, or `keras.utils.Sequence` instances (since they generate
 |              batches).
 |          verbose: 0 or 1. Verbosity mode.
 |              0 = silent, 1 = progress bar.
 |          sample_weight: Optional Numpy array of weights for
 |              the test samples, used for weighting the loss function.
 |              You can either pass a flat (1D)
 |              Numpy array with the same length as the input samples
 |              (1:1 mapping between weights and samples),
 |              or in the case of temporal data,
 |              you can pass a 2D array with shape
 |              `(samples, sequence_length)`,
 |              to apply a different weight to every timestep of every sample.
 |              In this case you should make sure to specify
 |              `sample_weight_mode="temporal"` in `compile()`. This argument is not
 |              supported when `x` is a dataset, instead pass
 |              sample weights as the third element of `x`.
 |          steps: Integer or `None`.
 |              Total number of steps (batches of samples)
 |              before declaring the evaluation round finished.
 |              Ignored with the default value of `None`.
 |              If x is a `tf.data` dataset and `steps` is
```

```
 |              None, 'evaluate' will run until the dataset is exhausted.
 |              This argument is not supported with array inputs.
 |          callbacks: List of `keras.callbacks.Callback` instances.
 |              List of callbacks to apply during evaluation.
 |              See [callbacks](/api_docs/python/tf/keras/callbacks).
 |          max_queue_size: Integer. Used for generator or `keras.utils.Sequence`
 |              input only. Maximum size for the generator queue.
 |              If unspecified, `max_queue_size` will default to 10.
 |          workers: Integer. Used for generator or `keras.utils.Sequence` input
 |              only. Maximum number of processes to spin up when using
 |              process-based threading. If unspecified, `workers` will default
 |              to 1. If 0, will execute the generator on the main thread.
 |          use_multiprocessing: Boolean. Used for generator or
 |              `keras.utils.Sequence` input only. If `True`, use process-based
 |              threading. If unspecified, `use_multiprocessing` will default to
 |              `False`. Note that because this implementation relies on
 |              multiprocessing, you should not pass non-picklable arguments to
 |              the generator as they can't be passed easily to children processes.
 |
 |      Returns:
 |          Scalar test loss (if the model has a single output and no metrics)
 |          or list of scalars (if the model has multiple outputs
 |          and/or metrics). The attribute `model.metrics_names` will give you
 |          the display labels for the scalar outputs.
 |
 |      Raises:
 |          ValueError: in case of invalid arguments.
 |
 |  evaluate_generator(self, generator, steps=None, callbacks=None, max_queue_size=10, workers=1, use_multiproc
essing=False, verbose=0)
 |      Evaluates the model on a data generator.
 |
 |      The generator should return the same kind of data
 |      as accepted by `test_on_batch`.
 |
 |      Arguments:
 |          generator: Generator yielding tuples (inputs, targets)
 |              or (inputs, targets, sample_weights)
 |              or an instance of `keras.utils.Sequence`
 |              object in order to avoid duplicate data
 |              when using multiprocessing.
 |          steps: Total number of steps (batches of samples)
 |              to yield from `generator` before stopping.
 |              Optional for `Sequence`: if unspecified, will use
 |              the `len(generator)` as a number of steps.
 |          callbacks: List of `keras.callbacks.Callback` instances.
 |              List of callbacks to apply during evaluation.
 |              See [callbacks](/api_docs/python/tf/keras/callbacks).
 |          max_queue_size: maximum size for the generator queue
 |          workers: Integer. Maximum number of processes to spin up
 |              when using process-based threading.
 |              If unspecified, `workers` will default to 1. If 0, will
 |              execute the generator on the main thread.
 |          use_multiprocessing: Boolean.
 |              If `True`, use process-based threading.
 |              If unspecified, `use_multiprocessing` will default to `False`.
 |              Note that because this implementation relies on multiprocessing,
 |              you should not pass non-picklable arguments to the generator
 |              as they can't be passed easily to children processes.
 |          verbose: Verbosity mode, 0 or 1.
 |
 |      Returns:
 |          Scalar test loss (if the model has a single output and no metrics)
 |          or list of scalars (if the model has multiple outputs
 |          and/or metrics). The attribute `model.metrics_names` will give you
 |          the display labels for the scalar outputs.
 |
 |      Raises:
 |          ValueError: in case of invalid arguments.
 |
 |      Raises:
 |          ValueError: In case the generator yields data in an invalid format.
 |
 |  fit(self, x=None, y=None, batch_size=None, epochs=1, verbose=1, callbacks=None, validation_split=0.0, valid
ation_data=None, shuffle=True, class_weight=None, sample_weight=None, initial_epoch=0, steps_per_epoch=None, va
lidation_steps=None, validation_freq=1, max_queue_size=10, workers=1, use_multiprocessing=False, **kwargs)
 |      Trains the model for a fixed number of epochs (iterations on a dataset).
 |
 |      Arguments:
 |          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. 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, generator,
|            or `keras.utils.Sequence` instance, `y` should
|            not be specified (since targets will be obtained from `x`).
|          batch_size: Integer or `None`.
|              Number of samples per gradient update.
|              If unspecified, `batch_size` will default to 32.
|              Do not specify the `batch_size` if your data is in the
|              form of symbolic tensors, datasets,
|              generators, or `keras.utils.Sequence` instances (since they generate
|              batches).
|          epochs: Integer. Number of epochs to train the model.
|              An epoch is an iteration over the entire `x` and `y`
|              data provided.
|              Note that in conjunction with `initial_epoch`,
|              `epochs` is to be understood as "final epoch".
|              The model is not trained for a number of iterations
|              given by `epochs`, but merely until the epoch
|              of index `epochs` is reached.
|          verbose: 0, 1, or 2. Verbosity mode.
|              0 = silent, 1 = progress bar, 2 = one line per epoch.
|              Note that the progress bar is not particularly useful when
|              logged to a file, so verbose=2 is recommended when not running
|              interactively (eg, in a production environment).
|          callbacks: List of `keras.callbacks.Callback` instances.
|              List of callbacks to apply during training.
|              See `tf.keras.callbacks`.
|          validation_split: Float between 0 and 1.
|              Fraction of the training data to be used as validation data.
|              The model will set apart this fraction of the training data,
|              will not train on it, and will evaluate
|              the loss and any model metrics
|              on this data at the end of each epoch.
|              The validation data is selected from the last samples
|              in the `x` and `y` data provided, before shuffling. This argument is
|              not supported when `x` is a dataset, generator or
|             `keras.utils.Sequence` instance.
|          validation_data: Data on which to evaluate
|              the loss and any model metrics at the end of each epoch.
|              The model will not be trained on this data.
|              `validation_data` will override `validation_split`.
|              `validation_data` could be:
|                - tuple `(x_val, y_val)` of Numpy arrays or tensors
|                - tuple `(x_val, y_val, val_sample_weights)` of Numpy arrays
|                - dataset
|              For the first two cases, `batch_size` must be provided.
|              For the last case, `validation_steps` must be provided.
|          shuffle: Boolean (whether to shuffle the training data
|              before each epoch) or str (for 'batch').
|              'batch' is a special option for dealing with the
|              limitations of HDF5 data; it shuffles in batch-sized chunks.
|              Has no effect when `steps_per_epoch` is not `None`.
|          class_weight: Optional dictionary mapping class indices (integers)
|              to a weight (float) value, used for weighting the loss function
|              (during training only).
|              This can be useful to tell the model to
|              "pay more attention" to samples from
|              an under-represented class.
|          sample_weight: Optional Numpy array of weights for
|              the training samples, used for weighting the loss function
|              (during training only). You can either pass a flat (1D)
|              Numpy array with the same length as the input samples
|              (1:1 mapping between weights and samples),
|              or in the case of temporal data,
|              you can pass a 2D array with shape
|              `(samples, sequence_length)`,
|              to apply a different weight to every timestep of every sample.
|              In this case you should make sure to specify
|              `sample_weight_mode="temporal"` in `compile()`. This argument is not
|              supported when `x` is a dataset, generator, or
|             `keras.utils.Sequence` instance, instead provide the sample_weights
|              as the third element of `x`.
|          initial_epoch: Integer.
|              Epoch at which to start training
|              (useful for resuming a previous training run).
|          steps_per_epoch: Integer or `None`.
|              Total number of steps (batches of samples)
|              before declaring one epoch finished and starting the
|              next epoch. When training with input tensors such as
|              TensorFlow data tensors, the default `None` is equal to
|              the number of samples in your dataset divided by
|              the batch size, or 1 if that cannot be determined. If x is a
|              `tf.data` dataset, and 'steps_per_epoch'
|              is None, the epoch will run until the input dataset is exhausted.
|              This argument is not supported with array inputs.
```

```
 |          validation_steps: Only relevant if `validation_data` is provided and
 |              is a `tf.data` dataset. Total number of steps (batches of
 |              samples) to draw before stopping when performing validation
 |              at the end of every epoch. If validation_data is a `tf.data` dataset
 |              and 'validation_steps' is None, validation
 |              will run until the `validation_data` dataset is exhausted.
 |          validation_freq: Only relevant if validation data is provided. Integer
 |              or `collections_abc.Container` instance (e.g. list, tuple, etc.).
 |              If an integer, specifies how many training epochs to run before a
 |              new validation run is performed, e.g. `validation_freq=2` runs
 |              validation every 2 epochs. If a Container, specifies the epochs on
 |              which to run validation, e.g. `validation_freq=[1, 2, 10]` runs
 |              validation at the end of the 1st, 2nd, and 10th epochs.
 |          max_queue_size: Integer. Used for generator or `keras.utils.Sequence`
 |              input only. Maximum size for the generator queue.
 |              If unspecified, `max_queue_size` will default to 10.
 |          workers: Integer. Used for generator or `keras.utils.Sequence` input
 |              only. Maximum number of processes to spin up
 |              when using process-based threading. If unspecified, `workers`
 |              will default to 1. If 0, will execute the generator on the main
 |              thread.
 |          use_multiprocessing: Boolean. Used for generator or
 |              `keras.utils.Sequence` input only. If `True`, use process-based
 |              threading. If unspecified, `use_multiprocessing` will default to
 |              `False`. Note that because this implementation relies on
 |              multiprocessing, you should not pass non-picklable arguments to
 |              the generator as they can't be passed easily to children processes.
 |          **kwargs: Used for backwards compatibility.
 |
 |      Returns:
 |          A `History` object. Its `History.history` attribute is
 |          a record of training loss values and metrics values
 |          at successive epochs, as well as validation loss values
 |          and validation metrics values (if applicable).
 |
 |      Raises:
 |          RuntimeError: If the model was never compiled.
 |          ValueError: In case of mismatch between the provided input data
 |              and what the model expects.
 |
 |  fit_generator(self, generator, steps_per_epoch=None, epochs=1, verbose=1, callbacks=None, validation_data=N
one, validation_steps=None, validation_freq=1, class_weight=None, max_queue_size=10, workers=1, use_multiproces
sing=False, shuffle=True, initial_epoch=0)
 |      Fits the model on data yielded batch-by-batch by a Python generator.
 |
 |      The generator is run in parallel to the model, for efficiency.
 |      For instance, this allows you to do real-time data augmentation
 |      on images on CPU in parallel to training your model on GPU.
 |
 |      The use of `keras.utils.Sequence` guarantees the ordering
 |      and guarantees the single use of every input per epoch when
 |      using `use_multiprocessing=True`.
 |
 |      Arguments:
 |          generator: A generator or an instance of `Sequence`
 |            (`keras.utils.Sequence`)
 |              object in order to avoid duplicate data
 |              when using multiprocessing.
 |              The output of the generator must be either
 |              - a tuple `(inputs, targets)`
 |              - a tuple `(inputs, targets, sample_weights)`.
 |              This tuple (a single output of the generator) makes a single batch.
 |              Therefore, all arrays in this tuple must have the same length (equal
 |              to the size of this batch). Different batches may have different
 |                sizes.
 |              For example, the last batch of the epoch is commonly smaller than
 |                the
 |              others, if the size of the dataset is not divisible by the batch
 |                size.
 |              The generator is expected to loop over its data
 |              indefinitely. An epoch finishes when `steps_per_epoch`
 |              batches have been seen by the model.
 |          steps_per_epoch: Total number of steps (batches of samples)
 |              to yield from `generator` before declaring one epoch
 |              finished and starting the next epoch. It should typically
 |              be equal to the number of samples of your dataset
 |              divided by the batch size.
 |              Optional for `Sequence`: if unspecified, will use
 |              the `len(generator)` as a number of steps.
 |          epochs: Integer, total number of iterations on the data.
 |          verbose: Verbosity mode, 0, 1, or 2.
 |          callbacks: List of callbacks to be called during training.
 |          validation_data: This can be either
 |              - a generator for the validation data
 |              - a tuple (inputs, targets)
 |              - a tuple (inputs, targets, sample_weights).
 |          validation_steps: Only relevant if `validation_data`
 |              is a generator. Total number of steps (batches of samples)
 |              to yield from `generator` before stopping.
```

```
|              Optional for `Sequence`: if unspecified, will use
|              the `len(validation_data)` as a number of steps.
|          validation_freq: Only relevant if validation data is provided. Integer
|              or `collections_abc.Container` instance (e.g. list, tuple, etc.).
|              If an integer, specifies how many training epochs to run before a
|              new validation run is performed, e.g. `validation_freq=2` runs
|              validation every 2 epochs. If a Container, specifies the epochs on
|              which to run validation, e.g. `validation_freq=[1, 2, 10]` runs
|              validation at the end of the 1st, 2nd, and 10th epochs.
|          class_weight: Dictionary mapping class indices to a weight
|              for the class.
|          max_queue_size: Integer. Maximum size for the generator queue.
|              If unspecified, `max_queue_size` will default to 10.
|          workers: Integer. Maximum number of processes to spin up
|              when using process-based threading.
|              If unspecified, `workers` will default to 1. If 0, will
|              execute the generator on the main thread.
|          use_multiprocessing: Boolean.
|              If `True`, use process-based threading.
|              If unspecified, `use_multiprocessing` will default to `False`.
|              Note that because this implementation relies on multiprocessing,
|              you should not pass non-picklable arguments to the generator
|              as they can't be passed easily to children processes.
|          shuffle: Boolean. Whether to shuffle the order of the batches at
|              the beginning of each epoch. Only used with instances
|              of `Sequence` (`keras.utils.Sequence`).
|              Has no effect when `steps_per_epoch` is not `None`.
|          initial_epoch: Epoch at which to start training
|              (useful for resuming a previous training run)
|
|      Returns:
|          A `History` object.
|
|      Example:
|
|      ```python
|          def generate_arrays_from_file(path):
|              while 1:
|                  f = open(path)
|                  for line in f:
|                      # create numpy arrays of input data
|                      # and labels, from each line in the file
|                      x1, x2, y = process_line(line)
|                      yield ({'input_1': x1, 'input_2': x2}, {'output': y})
|                  f.close()
|
|          model.fit_generator(generate_arrays_from_file('/my_file.txt'),
|                              steps_per_epoch=10000, epochs=10)
|      ```
|      Raises:
|          ValueError: In case the generator yields data in an invalid format.
|
|  get_weights(self)
|      Retrieves the weights of the model.
|
|      Returns:
|          A flat list of Numpy arrays.
|
|  load_weights(self, filepath, by_name=False)
|      Loads all layer weights, either from a TensorFlow or an HDF5 file.
|
|  predict(self, x, batch_size=None, verbose=0, steps=None, callbacks=None, max_queue_size=10, workers=1, use_
multiprocessing=False)
|      Generates output predictions for the input samples.
|
|      Computation is done in batches.
|
|      Arguments:
|          x: Input samples. 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 `tf.data` dataset.
|            - A generator or `keras.utils.Sequence` instance.
|          batch_size: Integer or `None`.
|              Number of samples per gradient update.
|              If unspecified, `batch_size` will default to 32.
|              Do not specify the `batch_size` is your data is in the
|              form of symbolic tensors, dataset,
|              generators, or `keras.utils.Sequence` instances (since they generate
|              batches).
|          verbose: Verbosity mode, 0 or 1.
|          steps: Total number of steps (batches of samples)
|              before declaring the prediction round finished.
|              Ignored with the default value of `None`. If x is a `tf.data`
|              dataset and `steps` is None, `predict` will
|              run until the input dataset is exhausted.
|          callbacks: List of `keras.callbacks.Callback` instances.
```

```
|              List of callbacks to apply during prediction.
|              See [callbacks](/api_docs/python/tf/keras/callbacks).
|          max_queue_size: Integer. Used for generator or `keras.utils.Sequence`
|              input only. Maximum size for the generator queue.
|              If unspecified, `max_queue_size` will default to 10.
|          workers: Integer. Used for generator or `keras.utils.Sequence` input
|              only. Maximum number of processes to spin up when using
|              process-based threading. If unspecified, `workers` will default
|              to 1. If 0, will execute the generator on the main thread.
|          use_multiprocessing: Boolean. Used for generator or
|              `keras.utils.Sequence` input only. If `True`, use process-based
|              threading. If unspecified, `use_multiprocessing` will default to
|              `False`. Note that because this implementation relies on
|              multiprocessing, you should not pass non-picklable arguments to
|              the generator as they can't be passed easily to children processes.
|
|
|      Returns:
|          Numpy array(s) of predictions.
|
|      Raises:
|          ValueError: In case of mismatch between the provided
|              input data and the model's expectations,
|              or in case a stateful model receives a number of samples
|              that is not a multiple of the batch size.
|
|  predict_generator(self, generator, steps=None, callbacks=None, max_queue_size=10, workers=1, use_multiproce
ssing=False, verbose=0)
|      Generates predictions for the input samples from a data generator.
|
|      The generator should return the same kind of data as accepted by
|      `predict_on_batch`.
|
|      Arguments:
|          generator: Generator yielding batches of input samples
|              or an instance of `keras.utils.Sequence` object in order to
|              avoid duplicate data when using multiprocessing.
|          steps: Total number of steps (batches of samples)
|              to yield from `generator` before stopping.
|              Optional for `Sequence`: if unspecified, will use
|              the `len(generator)` as a number of steps.
|          callbacks: List of `keras.callbacks.Callback` instances.
|              List of callbacks to apply during prediction.
|              See [callbacks](/api_docs/python/tf/keras/callbacks).
|          max_queue_size: Maximum size for the generator queue.
|          workers: Integer. Maximum number of processes to spin up
|              when using process-based threading.
|              If unspecified, `workers` will default to 1. If 0, will
|              execute the generator on the main thread.
|          use_multiprocessing: Boolean.
|              If `True`, use process-based threading.
|              If unspecified, `use_multiprocessing` will default to `False`.
|              Note that because this implementation relies on multiprocessing,
|              you should not pass non-picklable arguments to the generator
|              as they can't be passed easily to children processes.
|          verbose: verbosity mode, 0 or 1.
|
|      Returns:
|          Numpy array(s) of predictions.
|
|      Raises:
|          ValueError: In case the generator yields data in an invalid format.
|
|  predict_on_batch(self, x)
|      Returns predictions for a single batch of samples.
|
|      Arguments:
|          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 `tf.data` dataset.
|
|      Returns:
|          Numpy array(s) of predictions.
|
|      Raises:
|          ValueError: In case of mismatch between given number of inputs and
|            expectations of the model.
|
|  reset_metrics(self)
|      Resets the state of metrics.
|
|  test_on_batch(self, x, y=None, sample_weight=None, reset_metrics=True)
|      Test the model on a single batch of samples.
|
|      Arguments:
|          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.
|          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 `y` should
|            not be specified (since targets will be obtained from the iterator).
|          sample_weight: Optional array of the same length as x, containing
|              weights to apply to the model's loss for each sample.
|              In the case of temporal data, you can pass a 2D array
|              with shape (samples, sequence_length),
|              to apply a different weight to every timestep of every sample.
|              In this case you should make sure to specify
|              sample_weight_mode="temporal" in compile(). This argument is not
|              supported when `x` is a dataset.
|          reset_metrics: If `True`, the metrics returned will be only for this
|            batch. If `False`, the metrics will be statefully accumulated across
|            batches.
|
|      Returns:
|          Scalar test loss (if the model has a single output and no metrics)
|          or list of scalars (if the model has multiple outputs
|          and/or metrics). The attribute `model.metrics_names` will give you
|          the display labels for the scalar outputs.
|
|      Raises:
|          ValueError: In case of invalid user-provided arguments.
|
|  train_on_batch(self, x, y=None, sample_weight=None, class_weight=None, reset_metrics=True)
|      Runs a single gradient update on a single batch of data.
|
|      Arguments:
|          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.
|          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, `y` should not be specified
|            (since targets will be obtained from the iterator).
|          sample_weight: Optional array of the same length as x, containing
|            weights to apply to the model's loss for each sample. In the case of
|            temporal data, you can pass a 2D array with shape (samples,
|            sequence_length), to apply a different weight to every timestep of
|            every sample. In this case you should make sure to specify
|            sample_weight_mode="temporal" in compile(). This argument is not
|            supported when `x` is a dataset.
|          class_weight: Optional dictionary mapping class indices (integers) to a
|            weight (float) to apply to the model's loss for the samples from this
|            class during training. This can be useful to tell the model to "pay
|            more attention" to samples from an under-represented class.
|          reset_metrics: If `True`, the metrics returned will be only for this
|            batch. If `False`, the metrics will be statefully accumulated across
|            batches.
|
|      Returns:
|          Scalar training loss
|          (if the model has a single output and no metrics)
|          or list of scalars (if the model has multiple outputs
|          and/or metrics). The attribute `model.metrics_names` will give you
|          the display labels for the scalar outputs.
|
|      Raises:
|        ValueError: In case of invalid user-provided arguments.
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from tensorflow.python.keras.engine.training.Model:
|
|  metrics
|      Returns the model's metrics added using `compile`, `add_metric` APIs.
|
|  metrics_names
|      Returns the model's display labels for all outputs.
|
|  run_eagerly
|      Settable attribute indicating whether the model should run eagerly.
|
|      Running eagerly means that your model will be run step by step,
|      like Python code. Your model might run slower, but it should become easier
```

```
|      for you to debug it by stepping into individual layer calls.
|
|      By default, we will attempt to compile your model to a static graph to
|      deliver the best execution performance.
|
|      Returns:
|        Boolean, whether the model should run eagerly.
|
|  sample_weights
|
|  ----------------------------------------------------------------------
|  Methods inherited from tensorflow.python.keras.engine.network.Network:
|
|  __setattr__(self, name, value)
|      Support self.foo = trackable syntax.
|
|  get_layer(self, name=None, index=None)
|      Retrieves a layer based on either its name (unique) or index.
|
|      If `name` and `index` are both provided, `index` will take precedence.
|      Indices are based on order of horizontal graph traversal (bottom-up).
|
|      Arguments:
|          name: String, name of layer.
|          index: Integer, index of layer.
|
|      Returns:
|          A layer instance.
|
|      Raises:
|          ValueError: In case of invalid layer name or index.
|
|  reset_states(self)
|
|  save(self, filepath, overwrite=True, include_optimizer=True, save_format=None, signatures=None, options=Non
e)
|      Saves the model to Tensorflow SavedModel or a single HDF5 file.
|
|      The savefile includes:
|          - The model architecture, allowing to re-instantiate the model.
|          - The model weights.
|          - The state of the optimizer, allowing to resume training
|              exactly where you left off.
|
|      This allows you to save the entirety of the state of a model
|      in a single file.
|
|      Saved models can be reinstantiated via `keras.models.load_model`.
|      The model returned by `load_model`
|      is a compiled model ready to be used (unless the saved model
|      was never compiled in the first place).
|
|      Arguments:
|          filepath: String, path to SavedModel or H5 file to save the model.
|          overwrite: Whether to silently overwrite any existing file at the
|              target location, or provide the user with a manual prompt.
|          include_optimizer: If True, save optimizer's state together.
|          save_format: Either 'tf' or 'h5', indicating whether to save the model
|            to Tensorflow SavedModel or HDF5. The default is currently 'h5', but
|            will switch to 'tf' in TensorFlow 2.0. The 'tf' option is currently
|            disabled (use `tf.keras.experimental.export_saved_model` instead).
|        signatures: Signatures to save with the SavedModel. Applicable to the 'tf'
|          format only. Please see the `signatures` argument in
|          `tf.saved_model.save` for details.
|        options: Optional `tf.saved_model.SaveOptions` object that specifies
|          options for saving to SavedModel.
|
|      Example:
|
|      ```python
|      from keras.models import load_model
|
|      model.save('my_model.h5')  # creates a HDF5 file 'my_model.h5'
|      del model  # deletes the existing model
|
|      # returns a compiled model
|      # identical to the previous one
|      model = load_model('my_model.h5')
|      ```
|
|  save_weights(self, filepath, overwrite=True, save_format=None)
|      Saves all layer weights.
|
|      Either saves in HDF5 or in TensorFlow format based on the `save_format`
|      argument.
|
|      When saving in HDF5 format, the weight file has:
|        - `layer_names` (attribute), a list of strings
|            (ordered names of model layers).
```

```
|        - For every layer, a `group` named `layer.name`
|            - For every such layer group, a group attribute `weight_names`,
|                a list of strings
|                (ordered names of weights tensor of the layer).
|            - For every weight in the layer, a dataset
|                storing the weight value, named after the weight tensor.
|
|      When saving in TensorFlow format, all objects referenced by the network are
|      saved in the same format as `tf.train.Checkpoint`, including any `Layer`
|      instances or `Optimizer` instances assigned to object attributes. For
|      networks constructed from inputs and outputs using `tf.keras.Model(inputs,
|      outputs)`, `Layer` instances used by the network are tracked/saved
|      automatically. For user-defined classes which inherit from `tf.keras.Model`,
|      `Layer` instances must be assigned to object attributes, typically in the
|      constructor. See the documentation of `tf.train.Checkpoint` and
|      `tf.keras.Model` for details.
|
|      While the formats are the same, do not mix `save_weights` and
|      `tf.train.Checkpoint`. Checkpoints saved by `Model.save_weights` should be
|      loaded using `Model.load_weights`. Checkpoints saved using
|      `tf.train.Checkpoint.save` should be restored using the corresponding
|      `tf.train.Checkpoint.restore`. Prefer `tf.train.Checkpoint` over
|      `save_weights` for training checkpoints.
|
|      The TensorFlow format matches objects and variables by starting at a root
|      object, `self` for `save_weights`, and greedily matching attribute
|      names. For `Model.save` this is the `Model`, and for `Checkpoint.save` this
|      is the `Checkpoint` even if the `Checkpoint` has a model attached. This
|      means saving a `tf.keras.Model` using `save_weights` and loading into a
|      `tf.train.Checkpoint` with a `Model` attached (or vice versa) will not match
|      the `Model`'s variables. See the [guide to training
|      checkpoints](https://www.tensorflow.org/alpha/guide/checkpoints) for details
|      on the TensorFlow format.
|
|      Arguments:
|          filepath: String, path to the file to save the weights to. When saving
|              in TensorFlow format, this is the prefix used for checkpoint files
|              (multiple files are generated). Note that the '.h5' suffix causes
|              weights to be saved in HDF5 format.
|          overwrite: Whether to silently overwrite any existing file at the
|              target location, or provide the user with a manual prompt.
|          save_format: Either 'tf' or 'h5'. A `filepath` ending in '.h5' or
|              '.keras' will default to HDF5 if `save_format` is `None`. Otherwise
|              `None` defaults to 'tf'.
|
|      Raises:
|          ImportError: If h5py is not available when attempting to save in HDF5
|              format.
|          ValueError: For invalid/unknown format arguments.
|
|  summary(self, line_length=None, positions=None, print_fn=None)
|      Prints a string summary of the network.
|
|      Arguments:
|          line_length: Total length of printed lines
|              (e.g. set this to adapt the display to different
|              terminal window sizes).
|          positions: Relative or absolute positions of log elements
|              in each line. If not provided,
|              defaults to `[.33, .55, .67, 1.]`.
|          print_fn: Print function to use. Defaults to `print`.
|              It will be called on each line of the summary.
|              You can set it to a custom function
|              in order to capture the string summary.
|
|      Raises:
|          ValueError: if `summary()` is called before the model is built.
|
|  to_json(self, **kwargs)
|      Returns a JSON string containing the network configuration.
|
|      To load a network from a JSON save file, use
|      `keras.models.model_from_json(json_string, custom_objects={})`.
|
|      Arguments:
|          **kwargs: Additional keyword arguments
|              to be passed to `json.dumps()`.
|
|      Returns:
|          A JSON string.
|
|  to_yaml(self, **kwargs)
|      Returns a yaml string containing the network configuration.
|
|      To load a network from a yaml save file, use
|      `keras.models.model_from_yaml(yaml_string, custom_objects={})`.
|
|      `custom_objects` should be a dictionary mapping
|      the names of custom losses / layers / etc to the corresponding
```

```
|      functions / classes.
|
|      Arguments:
|          **kwargs: Additional keyword arguments
|              to be passed to `yaml.dump()`.
|
|      Returns:
|          A YAML string.
|
|      Raises:
|          ImportError: if yaml module is not found.
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from tensorflow.python.keras.engine.network.Network:
|
|  non_trainable_weights
|
|  state_updates
|      Returns the `updates` from all layers that are stateful.
|
|      This is useful for separating training updates and
|      state updates, e.g. when we need to update a layer's internal state
|      during prediction.
|
|      Returns:
|          A list of update ops.
|
|  stateful
|
|  trainable_weights
|
|  weights
|      Returns the list of all layer variables/weights.
|
|      Returns:
|        A list of variables.
|
|  ----------------------------------------------------------------------
|  Methods inherited from tensorflow.python.keras.engine.base_layer.Layer:
|
|  __call__(self, inputs, *args, **kwargs)
|      Wraps `call`, applying pre- and post-processing steps.
|
|      Arguments:
|        inputs: input tensor(s).
|        *args: additional positional arguments to be passed to `self.call`.
|        **kwargs: additional keyword arguments to be passed to `self.call`.
|
|      Returns:
|        Output tensor(s).
|
|      Note:
|        - The following optional keyword arguments are reserved for specific uses:
|          * `training`: Boolean scalar tensor of Python boolean indicating
|            whether the `call` is meant for training or inference.
|          * `mask`: Boolean input mask.
|        - If the layer's `call` method takes a `mask` argument (as some Keras
|          layers do), its default value will be set to the mask generated
|          for `inputs` by the previous layer (if `input` did come from
|          a layer that generated a corresponding mask, i.e. if it came from
|          a Keras layer with masking support.
|
|      Raises:
|        ValueError: if the layer's `call` method returns None (an invalid value).
|
|  __delattr__(self, name)
|      Implement delattr(self, name).
|
|  add_loss(self, losses, inputs=None)
|      Add loss tensor(s), potentially dependent on layer inputs.
|
|      Some losses (for instance, activity regularization losses) may be dependent
|      on the inputs passed when calling a layer. Hence, when reusing the same
|      layer on different inputs `a` and `b`, some entries in `layer.losses` may
|      be dependent on `a` and some on `b`. This method automatically keeps track
|      of dependencies.
|
|      This method can be used inside a subclassed layer or model's `call`
|      function, in which case `losses` should be a Tensor or list of Tensors.
|
|      Example:
|
|      ```python
|      class MyLayer(tf.keras.layers.Layer):
|        def call(inputs, self):
|           self.add_loss(tf.abs(tf.reduce_mean(inputs)), inputs=True)
|           return inputs
|      ```
|
```

```
|      This method can also be called directly on a Functional Model during
|      construction. In this case, any loss Tensors passed to this Model must
|      be symbolic and be able to be traced back to the model's `Input`s. These
|      losses become part of the model's topology and are tracked in `get_config`.
|
|      Example:
|
|      ```python
|      inputs = tf.keras.Input(shape=(10,))
|      x = tf.keras.layers.Dense(10)(inputs)
|      outputs = tf.keras.layers.Dense(1)(x)
|      model = tf.keras.Model(inputs, outputs)
|      # Actvity regularization.
|      model.add_loss(tf.abs(tf.reduce_mean(x)))
|      ```
|
|      If this is not the case for your loss (if, for example, your loss references
|      a `Variable` of one of the model's layers), you can wrap your loss in a
|      zero-argument lambda. These losses are not tracked as part of the model's
|      topology since they can't be serialized.
|
|      Example:
|
|      ```python
|      inputs = tf.keras.Input(shape=(10,))
|      x = tf.keras.layers.Dense(10)(inputs)
|      outputs = tf.keras.layers.Dense(1)(x)
|      model = tf.keras.Model(inputs, outputs)
|      # Weight regularization.
|      model.add_loss(lambda: tf.reduce_mean(x.kernel))
|      ```
|
|      The `get_losses_for` method allows to retrieve the losses relevant to a
|      specific set of inputs.
|
|      Arguments:
|        losses: Loss tensor, or list/tuple of tensors. Rather than tensors, losses
|          may also be zero-argument callables which create a loss tensor.
|        inputs: Ignored when executing eagerly. If anything other than None is
|          passed, it signals the losses are conditional on some of the layer's
|          inputs, and thus they should only be run where these inputs are
|          available. This is the case for activity regularization losses, for
|          instance. If `None` is passed, the losses are assumed
|          to be unconditional, and will apply across all dataflows of the layer
|          (e.g. weight regularization losses).
|
|  add_metric(self, value, aggregation=None, name=None)
|      Adds metric tensor to the layer.
|
|      Args:
|        value: Metric tensor.
|        aggregation: Sample-wise metric reduction function. If `aggregation=None`,
|          it indicates that the metric tensor provided has been aggregated
|          already. eg, `bin_acc = BinaryAccuracy(name='acc')` followed by
|          `model.add_metric(bin_acc(y_true, y_pred))`. If aggregation='mean', the
|          given metric tensor will be sample-wise reduced using `mean` function.
|          eg, `model.add_metric(tf.reduce_sum(outputs), name='output_mean',
|          aggregation='mean')`.
|        name: String metric name.
|
|      Raises:
|        ValueError: If `aggregation` is anything other than None or `mean`.
|
|  add_update(self, updates, inputs=None)
|      Add update op(s), potentially dependent on layer inputs. (deprecated arguments)
|
|      Warning: SOME ARGUMENTS ARE DEPRECATED: `(inputs)`. They will be removed in a future version.
|      Instructions for updating:
|      `inputs` is now automatically inferred
|
|      Weight updates (for instance, the updates of the moving mean and variance
|      in a BatchNormalization layer) may be dependent on the inputs passed
|      when calling a layer. Hence, when reusing the same layer on
|      different inputs `a` and `b`, some entries in `layer.updates` may be
|      dependent on `a` and some on `b`. This method automatically keeps track
|      of dependencies.
|
|      The `get_updates_for` method allows to retrieve the updates relevant to a
|      specific set of inputs.
|
|      This call is ignored when eager execution is enabled (in that case, variable
|      updates are run on the fly and thus do not need to be tracked for later
|      execution).
|
|      Arguments:
|        updates: Update op, or list/tuple of update ops, or zero-arg callable
|          that returns an update op. A zero-arg callable should be passed in
|          order to disable running the updates by setting `trainable=False`
|          on this Layer, when executing in Eager mode.
```

```
 |        inputs: Deprecated, will be automatically inferred.
 |
 |  add_variable(self, *args, **kwargs)
 |      Deprecated, do NOT use! Alias for `add_weight`. (deprecated)
 |
 |      Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version.
 |      Instructions for updating:
 |      Please use `layer.add_weight` method instead.
 |
 |  add_weight(self, name=None, shape=None, dtype=None, initializer=None, regularizer=None, trainable=None, con
straint=None, partitioner=None, use_resource=None, synchronization=<VariableSynchronization.AUTO: 0>, aggregati
on=<VariableAggregation.NONE: 0>, **kwargs)
 |      Adds a new variable to the layer.
 |
 |      Arguments:
 |        name: Variable name.
 |        shape: Variable shape. Defaults to scalar if unspecified.
 |        dtype: The type of the variable. Defaults to `self.dtype` or `float32`.
 |        initializer: Initializer instance (callable).
 |        regularizer: Regularizer instance (callable).
 |        trainable: Boolean, whether the variable should be part of the layer's
 |          "trainable_variables" (e.g. variables, biases)
 |          or "non_trainable_variables" (e.g. BatchNorm mean and variance).
 |          Note that `trainable` cannot be `True` if `synchronization`
 |          is set to `ON_READ`.
 |        constraint: Constraint instance (callable).
 |        partitioner: Partitioner to be passed to the `Trackable` API.
 |        use_resource: Whether to use `ResourceVariable`.
 |        synchronization: Indicates when a distributed a variable will be
 |          aggregated. Accepted values are constants defined in the class
 |          `tf.VariableSynchronization`. By default the synchronization is set to
 |          `AUTO` and the current `DistributionStrategy` chooses
 |          when to synchronize. If `synchronization` is set to `ON_READ`,
 |          `trainable` must not be set to `True`.
 |        aggregation: Indicates how a distributed variable will be aggregated.
 |          Accepted values are constants defined in the class
 |          `tf.VariableAggregation`.
 |        **kwargs: Additional keyword arguments. Accepted values are `getter` and
 |          `collections`.
 |
 |      Returns:
 |        The created variable. Usually either a `Variable` or `ResourceVariable`
 |        instance. If `partitioner` is not `None`, a `PartitionedVariable`
 |        instance is returned.
 |
 |      Raises:
 |        RuntimeError: If called with partitioned variable regularization and
 |          eager execution is enabled.
 |        ValueError: When giving unsupported dtype and no initializer or when
 |          trainable has been set to True with synchronization set as `ON_READ`.
 |
 |  apply(self, inputs, *args, **kwargs)
 |      Deprecated, do NOT use! (deprecated)
 |
 |      Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version.
 |      Instructions for updating:
 |      Please use `layer.__call__` method instead.
 |
 |      This is an alias of `self.__call__`.
 |
 |      Arguments:
 |        inputs: Input tensor(s).
 |        *args: additional positional arguments to be passed to `self.call`.
 |        **kwargs: additional keyword arguments to be passed to `self.call`.
 |
 |      Returns:
 |        Output tensor(s).
 |
 |  compute_output_signature(self, input_signature)
 |      Compute the output tensor signature of the layer based on the inputs.
 |
 |      Unlike a TensorShape object, a TensorSpec object contains both shape
 |      and dtype information for a tensor. This method allows layers to provide
 |      output dtype information if it is different from the input dtype.
 |      For any layer that doesn't implement this function,
 |      the framework will fall back to use `compute_output_shape`, and will
 |      assume that the output dtype matches the input dtype.
 |
 |      Args:
 |        input_signature: Single TensorSpec or nested structure of TensorSpec
 |          objects, describing a candidate input for the layer.
 |
 |      Returns:
 |        Single TensorSpec or nested structure of TensorSpec objects, describing
 |          how the layer would transform the provided input.
 |
 |      Raises:
 |        TypeError: If input_signature contains a non-TensorSpec object.
 |
```

```
|  count_params(self)
|      Count the total number of scalars composing the weights.
|
|      Returns:
|          An integer count.
|
|      Raises:
|          ValueError: if the layer isn't yet built
|            (in which case its weights aren't yet defined).
|
|  get_input_at(self, node_index)
|      Retrieves the input tensor(s) of a layer at a given node.
|
|      Arguments:
|          node_index: Integer, index of the node
|              from which to retrieve the attribute.
|              E.g. `node_index=0` will correspond to the
|              first time the layer was called.
|
|      Returns:
|          A tensor (or list of tensors if the layer has multiple inputs).
|
|      Raises:
|        RuntimeError: If called in Eager mode.
|
|  get_input_mask_at(self, node_index)
|      Retrieves the input mask tensor(s) of a layer at a given node.
|
|      Arguments:
|          node_index: Integer, index of the node
|              from which to retrieve the attribute.
|              E.g. `node_index=0` will correspond to the
|              first time the layer was called.
|
|      Returns:
|          A mask tensor
|          (or list of tensors if the layer has multiple inputs).
|
|  get_input_shape_at(self, node_index)
|      Retrieves the input shape(s) of a layer at a given node.
|
|      Arguments:
|          node_index: Integer, index of the node
|              from which to retrieve the attribute.
|              E.g. `node_index=0` will correspond to the
|              first time the layer was called.
|
|      Returns:
|          A shape tuple
|          (or list of shape tuples if the layer has multiple inputs).
|
|      Raises:
|        RuntimeError: If called in Eager mode.
|
|  get_losses_for(self, inputs)
|      Retrieves losses relevant to a specific set of inputs.
|
|      Arguments:
|        inputs: Input tensor or list/tuple of input tensors.
|
|      Returns:
|        List of loss tensors of the layer that depend on `inputs`.
|
|  get_output_at(self, node_index)
|      Retrieves the output tensor(s) of a layer at a given node.
|
|      Arguments:
|          node_index: Integer, index of the node
|              from which to retrieve the attribute.
|              E.g. `node_index=0` will correspond to the
|              first time the layer was called.
|
|      Returns:
|          A tensor (or list of tensors if the layer has multiple outputs).
|
|      Raises:
|        RuntimeError: If called in Eager mode.
|
|  get_output_mask_at(self, node_index)
|      Retrieves the output mask tensor(s) of a layer at a given node.
|
|      Arguments:
|          node_index: Integer, index of the node
|              from which to retrieve the attribute.
|              E.g. `node_index=0` will correspond to the
|              first time the layer was called.
|
|      Returns:
|          A mask tensor
```

```
|          (or list of tensors if the layer has multiple outputs).
|
|  get_output_shape_at(self, node_index)
|      Retrieves the output shape(s) of a layer at a given node.
|
|      Arguments:
|          node_index: Integer, index of the node
|              from which to retrieve the attribute.
|              E.g. `node_index=0` will correspond to the
|              first time the layer was called.
|
|      Returns:
|          A shape tuple
|          (or list of shape tuples if the layer has multiple outputs).
|
|      Raises:
|        RuntimeError: If called in Eager mode.
|
|  get_updates_for(self, inputs)
|      Retrieves updates relevant to a specific set of inputs.
|
|      Arguments:
|        inputs: Input tensor or list/tuple of input tensors.
|
|      Returns:
|        List of update ops of the layer that depend on `inputs`.
|
|  set_weights(self, weights)
|      Sets the weights of the layer, from Numpy arrays.
|
|      Arguments:
|          weights: a list of Numpy arrays. The number
|              of arrays and their shape must match
|              number of the dimensions of the weights
|              of the layer (i.e. it should match the
|              output of `get_weights`).
|
|      Raises:
|          ValueError: If the provided weights list does not match the
|              layer's specifications.
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from tensorflow.python.keras.engine.base_layer.Layer:
|
|  activity_regularizer
|      Optional regularizer function for the output of this layer.
|
|  dtype
|
|  inbound_nodes
|      Deprecated, do NOT use! Only for compatibility with external Keras.
|
|  input
|      Retrieves the input tensor(s) of a layer.
|
|      Only applicable if the layer has exactly one input,
|      i.e. if it is connected to one incoming layer.
|
|      Returns:
|          Input tensor or list of input tensors.
|
|      Raises:
|        RuntimeError: If called in Eager mode.
|        AttributeError: If no inbound nodes are found.
|
|  input_mask
|      Retrieves the input mask tensor(s) of a layer.
|
|      Only applicable if the layer has exactly one inbound node,
|      i.e. if it is connected to one incoming layer.
|
|      Returns:
|          Input mask tensor (potentially None) or list of input
|          mask tensors.
|
|      Raises:
|          AttributeError: if the layer is connected to
|          more than one incoming layers.
|
|  input_shape
|      Retrieves the input shape(s) of a layer.
|
|      Only applicable if the layer has exactly one input,
|      i.e. if it is connected to one incoming layer, or if all inputs
|      have the same shape.
|
|      Returns:
|          Input shape, as an integer shape tuple
|          (or list of shape tuples, one tuple per input tensor).
```

```
|
|      Raises:
|          AttributeError: if the layer has no defined input_shape.
|          RuntimeError: if called in Eager mode.
|
|  losses
|      Losses which are associated with this `Layer`.
|
|      Variable regularization tensors are created when this property is accessed,
|      so it is eager safe: accessing `losses` under a `tf.GradientTape` will
|      propagate gradients back to the corresponding variables.
|
|      Returns:
|        A list of tensors.
|
|  name
|      Returns the name of this module as passed or determined in the ctor.
|
|      NOTE: This is not the same as the `self.name_scope.name` which includes
|      parent module names.
|
|  non_trainable_variables
|
|  outbound_nodes
|      Deprecated, do NOT use! Only for compatibility with external Keras.
|
|  output
|      Retrieves the output tensor(s) of a layer.
|
|      Only applicable if the layer has exactly one output,
|      i.e. if it is connected to one incoming layer.
|
|      Returns:
|        Output tensor or list of output tensors.
|
|      Raises:
|        AttributeError: if the layer is connected to more than one incoming
|          layers.
|        RuntimeError: if called in Eager mode.
|
|  output_mask
|      Retrieves the output mask tensor(s) of a layer.
|
|      Only applicable if the layer has exactly one inbound node,
|      i.e. if it is connected to one incoming layer.
|
|      Returns:
|          Output mask tensor (potentially None) or list of output
|          mask tensors.
|
|      Raises:
|          AttributeError: if the layer is connected to
|          more than one incoming layers.
|
|  output_shape
|      Retrieves the output shape(s) of a layer.
|
|      Only applicable if the layer has one output,
|      or if all outputs have the same shape.
|
|      Returns:
|          Output shape, as an integer shape tuple
|          (or list of shape tuples, one tuple per output tensor).
|
|      Raises:
|          AttributeError: if the layer has no defined output shape.
|          RuntimeError: if called in Eager mode.
|
|  trainable
|
|  trainable_variables
|      Sequence of variables owned by this module and it's submodules.
|
|      Note: this method uses reflection to find variables on the current instance
|      and submodules. For performance reasons you may wish to cache the result
|      of calling this method if you don't expect the return value to change.
|
|      Returns:
|        A sequence of variables for the current module (sorted by attribute
|        name) followed by variables from all submodules recursively (breadth
|        first).
|
|  updates
|
|  variables
|      Returns the list of all layer variables/weights.
|
|      Alias of `self.weights`.
|
```

```
|      Returns:
|        A list of variables.
|
|  ----------------------------------------------------------------------
|  Class methods inherited from tensorflow.python.module.module.Module:
|
|  with_name_scope(method) from builtins.type
|      Decorator to automatically enter the module name scope.
|
|      ```
|      class MyModule(tf.Module):
|        @tf.Module.with_name_scope
|        def __call__(self, x):
|          if not hasattr(self, 'w'):
|            self.w = tf.Variable(tf.random.normal([x.shape[1], 64]))
|          return tf.matmul(x, self.w)
|      ```
|
|      Using the above module would produce `tf.Variable`s and `tf.Tensor`s whose
|      names included the module name:
|
|      ```
|      mod = MyModule()
|      mod(tf.ones([8, 32]))
|      # ==> <tf.Tensor: ...>
|      mod.w
|      # ==> <tf.Variable ...'my_module/w:0'>
|      ```
|
|      Args:
|        method: The method to wrap.
|
|      Returns:
|        The original method wrapped such that it enters the module's name scope.
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from tensorflow.python.module.module.Module:
|
|  name_scope
|      Returns a `tf.name_scope` instance for this class.
|
|  submodules
|      Sequence of all sub-modules.
|
|      Submodules are modules which are properties of this module, or found as
|      properties of modules which are properties of this module (and so on).
|
|      ```
|      a = tf.Module()
|      b = tf.Module()
|      c = tf.Module()
|      a.b = b
|      b.c = c
|      assert list(a.submodules) == [b, c]
|      assert list(b.submodules) == [c]
|      assert list(c.submodules) == []
|      ```
|
|      Returns:
|        A sequence of all submodules.
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from tensorflow.python.training.tracking.base.Trackable:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
```

Creating a Model

There are two ways to create models through the TF 2 Keras API, either pass in a list of layers all at once, or add them one by one.

Let's show both methods (its up to you to choose which method you prefer).

In [21]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
```

Model - as a list of layers

In [22]:
```
model = Sequential([
    Dense(units=2),
    Dense(units=2),
```

```
    Dense(units=2)
])
```

Model - adding in layers one by one

In [23]:
```
model = Sequential()

model.add(Dense(2))
model.add(Dense(2))
model.add(Dense(2))
```

Let's go ahead and build a simple model and then compile it by defining our solver

In [24]:
```
model = Sequential()

model.add(Dense(4,activation='relu'))
model.add(Dense(4,activation='relu'))
model.add(Dense(4,activation='relu'))

# Final output node for prediction
model.add(Dense(1))

model.compile(optimizer='rmsprop',loss='mse')
```

Choosing an optimizer and loss

Keep in mind what kind of problem you are trying to solve:

```
# For a multi-class classification problem
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# For a binary classification problem
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# For a mean squared error regression problem
model.compile(optimizer='rmsprop',
              loss='mse')
```

Training

Below are some common definitions that are necessary to know and understand to correctly utilize Keras:

- Sample: one element of a dataset.
  - Example: one image is a sample in a convolutional network
  - Example: one audio file is a sample for a speech recognition model
- Batch: a set of N samples. The samples in a batch are processed independently, in parallel. If training, a batch results in only one update to the model.A batch generally approximates the distribution of the input data better than a single input. The larger the batch, the better the approximation; however, it is also true that the batch will take longer to process and will still result in only one update. For inference (evaluate/predict), it is recommended to pick a batch size that is as large as you can afford without going out of memory (since larger batches will usually result in faster evaluation/prediction).
- Epoch: an arbitrary cutoff, generally defined as "one pass over the entire dataset", used to separate training into distinct phases, which is useful for logging and periodic evaluation.
- When using validation_data or validation_split with the fit method of Keras models, evaluation will be run at the end of every epoch.
- Within Keras, there is the ability to add callbacks specifically designed to be run at the end of an epoch. Examples of these are learning rate changes and model checkpointing (saving).

In [25]:
```
model.fit(X_train,y_train,epochs=250)
```

```
Train on 700 samples
Epoch 1/250
700/700 [==============================] - 1s 1ms/sample - loss: 257046.0205
Epoch 2/250
700/700 [==============================] - 0s 74us/sample - loss: 256727.5741
Epoch 3/250
700/700 [==============================] - 0s 78us/sample - loss: 256522.9977
Epoch 4/250
700/700 [==============================] - 0s 81us/sample - loss: 256373.0434
Epoch 5/250
700/700 [==============================] - 0s 91us/sample - loss: 256208.8084
Epoch 6/250
700/700 [==============================] - 0s 79us/sample - loss: 256017.8742
```

```
Epoch 7/250
700/700 [==============================] - 0s 83us/sample - loss: 255794.6387
Epoch 8/250
700/700 [==============================] - 0s 77us/sample - loss: 255536.6145
Epoch 9/250
700/700 [==============================] - 0s 80us/sample - loss: 255246.5084
Epoch 10/250
700/700 [==============================] - 0s 77us/sample - loss: 254920.9007
Epoch 11/250
700/700 [==============================] - 0s 76us/sample - loss: 254556.8057
Epoch 12/250
700/700 [==============================] - 0s 76us/sample - loss: 254155.8077
Epoch 13/250
700/700 [==============================] - 0s 76us/sample - loss: 253712.6570
Epoch 14/250
700/700 [==============================] - 0s 74us/sample - loss: 253222.9419
Epoch 15/250
700/700 [==============================] - 0s 77us/sample - loss: 252682.8450
Epoch 16/250
700/700 [==============================] - 0s 79us/sample - loss: 252086.9584
Epoch 17/250
700/700 [==============================] - 0s 78us/sample - loss: 251436.2046
Epoch 18/250
700/700 [==============================] - 0s 74us/sample - loss: 250726.1148
Epoch 19/250
700/700 [==============================] - 0s 75us/sample - loss: 249952.4523
Epoch 20/250
700/700 [==============================] - 0s 80us/sample - loss: 249109.9209
Epoch 21/250
700/700 [==============================] - 0s 77us/sample - loss: 248200.0201
Epoch 22/250
700/700 [==============================] - 0s 74us/sample - loss: 247215.9578
Epoch 23/250
700/700 [==============================] - 0s 81us/sample - loss: 246154.1205
Epoch 24/250
700/700 [==============================] - 0s 76us/sample - loss: 245005.0884
Epoch 25/250
700/700 [==============================] - 0s 77us/sample - loss: 243780.4235
Epoch 26/250
700/700 [==============================] - 0s 83us/sample - loss: 242465.3764
Epoch 27/250
700/700 [==============================] - 0s 74us/sample - loss: 241055.4634
Epoch 28/250
700/700 [==============================] - 0s 77us/sample - loss: 239559.6619
Epoch 29/250
700/700 [==============================] - 0s 75us/sample - loss: 237966.3670
Epoch 30/250
700/700 [==============================] - 0s 75us/sample - loss: 236272.6269
Epoch 31/250
700/700 [==============================] - 0s 76us/sample - loss: 234471.0364
Epoch 32/250
700/700 [==============================] - 0s 78us/sample - loss: 232554.9543
Epoch 33/250
700/700 [==============================] - 0s 79us/sample - loss: 230538.9865
Epoch 34/250
700/700 [==============================] - 0s 77us/sample - loss: 228401.8140
Epoch 35/250
700/700 [==============================] - 0s 77us/sample - loss: 226150.3127
Epoch 36/250
700/700 [==============================] - 0s 77us/sample - loss: 223783.8135
Epoch 37/250
700/700 [==============================] - 0s 78us/sample - loss: 221290.6057
Epoch 38/250
700/700 [==============================] - 0s 76us/sample - loss: 218679.4083
Epoch 39/250
700/700 [==============================] - 0s 77us/sample - loss: 215924.6912
Epoch 40/250
700/700 [==============================] - 0s 77us/sample - loss: 213063.5624
Epoch 41/250
700/700 [==============================] - 0s 77us/sample - loss: 210046.3804
Epoch 42/250
700/700 [==============================] - 0s 73us/sample - loss: 206900.9284
Epoch 43/250
700/700 [==============================] - 0s 77us/sample - loss: 203611.7862
Epoch 44/250
700/700 [==============================] - 0s 78us/sample - loss: 200231.5254
Epoch 45/250
700/700 [==============================] - 0s 74us/sample - loss: 196671.4621
Epoch 46/250
700/700 [==============================] - 0s 76us/sample - loss: 192974.4341
Epoch 47/250
700/700 [==============================] - 0s 78us/sample - loss: 189174.2291
Epoch 48/250
700/700 [==============================] - 0s 76us/sample - loss: 185202.6423
Epoch 49/250
700/700 [==============================] - 0s 76us/sample - loss: 181108.9913
Epoch 50/250
700/700 [==============================] - 0s 78us/sample - loss: 176864.3965
Epoch 51/250
```

```
700/700 [==============================] - 0s 76us/sample - loss: 172504.7179
Epoch 52/250
700/700 [==============================] - 0s 78us/sample - loss: 168003.5879
Epoch 53/250
700/700 [==============================] - 0s 77us/sample - loss: 163362.0865
Epoch 54/250
700/700 [==============================] - 0s 71us/sample - loss: 158614.5780
Epoch 55/250
700/700 [==============================] - 0s 70us/sample - loss: 153737.6910
Epoch 56/250
700/700 [==============================] - 0s 69us/sample - loss: 148756.8652
Epoch 57/250
700/700 [==============================] - 0s 70us/sample - loss: 143625.3819
Epoch 58/250
700/700 [==============================] - 0s 71us/sample - loss: 138388.0890
Epoch 59/250
700/700 [==============================] - 0s 73us/sample - loss: 133087.5437
Epoch 60/250
700/700 [==============================] - 0s 71us/sample - loss: 127688.3186
Epoch 61/250
700/700 [==============================] - 0s 71us/sample - loss: 122186.6950
Epoch 62/250
700/700 [==============================] - 0s 73us/sample - loss: 116597.9215
Epoch 63/250
700/700 [==============================] - 0s 74us/sample - loss: 110948.6829
Epoch 64/250
700/700 [==============================] - 0s 73us/sample - loss: 105236.5175
Epoch 65/250
700/700 [==============================] - 0s 71us/sample - loss: 99482.9941
Epoch 66/250
700/700 [==============================] - 0s 71us/sample - loss: 93701.2023
Epoch 67/250
700/700 [==============================] - 0s 70us/sample - loss: 87938.2565
Epoch 68/250
700/700 [==============================] - 0s 67us/sample - loss: 82171.9058
Epoch 69/250
700/700 [==============================] - 0s 74us/sample - loss: 76383.8975
Epoch 70/250
700/700 [==============================] - 0s 72us/sample - loss: 70635.6462
Epoch 71/250
700/700 [==============================] - 0s 70us/sample - loss: 64962.1172
Epoch 72/250
700/700 [==============================] - 0s 72us/sample - loss: 59347.5444
Epoch 73/250
700/700 [==============================] - 0s 71us/sample - loss: 53863.8244
Epoch 74/250
700/700 [==============================] - 0s 69us/sample - loss: 48459.2807
Epoch 75/250
700/700 [==============================] - 0s 66us/sample - loss: 43236.1913
Epoch 76/250
700/700 [==============================] - 0s 67us/sample - loss: 38164.6003
Epoch 77/250
700/700 [==============================] - 0s 68us/sample - loss: 33331.5632
Epoch 78/250
700/700 [==============================] - 0s 70us/sample - loss: 28683.7689
Epoch 79/250
700/700 [==============================] - 0s 71us/sample - loss: 24321.4795
Epoch 80/250
700/700 [==============================] - 0s 71us/sample - loss: 20278.4696
Epoch 81/250
700/700 [==============================] - 0s 70us/sample - loss: 16523.3231
Epoch 82/250
700/700 [==============================] - 0s 71us/sample - loss: 13159.0727
Epoch 83/250
700/700 [==============================] - 0s 73us/sample - loss: 10194.2917
Epoch 84/250
700/700 [==============================] - 0s 76us/sample - loss: 7640.4085
Epoch 85/250
700/700 [==============================] - 0s 73us/sample - loss: 5576.2742
Epoch 86/250
700/700 [==============================] - 0s 77us/sample - loss: 4007.2732
Epoch 87/250
700/700 [==============================] - 0s 70us/sample - loss: 2910.7986
Epoch 88/250
700/700 [==============================] - 0s 72us/sample - loss: 2304.2850
Epoch 89/250
700/700 [==============================] - 0s 70us/sample - loss: 2099.1053
Epoch 90/250
700/700 [==============================] - 0s 68us/sample - loss: 2047.8926
Epoch 91/250
700/700 [==============================] - 0s 71us/sample - loss: 2011.3148
Epoch 92/250
700/700 [==============================] - 0s 70us/sample - loss: 1977.7156
Epoch 93/250
700/700 [==============================] - 0s 68us/sample - loss: 1943.7099
Epoch 94/250
700/700 [==============================] - 0s 68us/sample - loss: 1906.4393
Epoch 95/250
700/700 [==============================] - 0s 67us/sample - loss: 1873.7698
```

```
Epoch 96/250
700/700 [==============================] - 0s 70us/sample - loss: 1839.4941
Epoch 97/250
700/700 [==============================] - 0s 70us/sample - loss: 1807.7019
Epoch 98/250
700/700 [==============================] - 0s 70us/sample - loss: 1773.0187
Epoch 99/250
700/700 [==============================] - 0s 70us/sample - loss: 1742.6451
Epoch 100/250
700/700 [==============================] - 0s 68us/sample - loss: 1706.0911
Epoch 101/250
700/700 [==============================] - 0s 68us/sample - loss: 1671.9197
Epoch 102/250
700/700 [==============================] - 0s 70us/sample - loss: 1641.0000
Epoch 103/250
700/700 [==============================] - 0s 70us/sample - loss: 1610.9062
Epoch 104/250
700/700 [==============================] - 0s 73us/sample - loss: 1579.4019
Epoch 105/250
700/700 [==============================] - 0s 74us/sample - loss: 1545.4603
Epoch 106/250
700/700 [==============================] - 0s 72us/sample - loss: 1518.5771
Epoch 107/250
700/700 [==============================] - 0s 71us/sample - loss: 1485.8637
Epoch 108/250
700/700 [==============================] - 0s 73us/sample - loss: 1457.9632
Epoch 109/250
700/700 [==============================] - 0s 74us/sample - loss: 1424.8218
Epoch 110/250
700/700 [==============================] - 0s 71us/sample - loss: 1394.5545
Epoch 111/250
700/700 [==============================] - 0s 71us/sample - loss: 1364.8500
Epoch 112/250
700/700 [==============================] - 0s 70us/sample - loss: 1337.1092
Epoch 113/250
700/700 [==============================] - 0s 71us/sample - loss: 1305.2350
Epoch 114/250
700/700 [==============================] - 0s 68us/sample - loss: 1277.8466
Epoch 115/250
700/700 [==============================] - 0s 73us/sample - loss: 1248.9611
Epoch 116/250
700/700 [==============================] - 0s 69us/sample - loss: 1218.9898
Epoch 117/250
700/700 [==============================] - 0s 73us/sample - loss: 1188.1648
Epoch 118/250
700/700 [==============================] - 0s 71us/sample - loss: 1159.2658
Epoch 119/250
700/700 [==============================] - 0s 70us/sample - loss: 1131.0404
Epoch 120/250
700/700 [==============================] - 0s 68us/sample - loss: 1098.9464
Epoch 121/250
700/700 [==============================] - 0s 71us/sample - loss: 1068.8702
Epoch 122/250
700/700 [==============================] - 0s 71us/sample - loss: 1034.1245
Epoch 123/250
700/700 [==============================] - 0s 74us/sample - loss: 1004.9061
Epoch 124/250
700/700 [==============================] - 0s 76us/sample - loss: 974.5798
Epoch 125/250
700/700 [==============================] - 0s 75us/sample - loss: 949.0737
Epoch 126/250
700/700 [==============================] - 0s 68us/sample - loss: 925.6563
Epoch 127/250
700/700 [==============================] - 0s 71us/sample - loss: 898.7898
Epoch 128/250
700/700 [==============================] - 0s 67us/sample - loss: 876.4439
Epoch 129/250
700/700 [==============================] - 0s 71us/sample - loss: 846.4307
Epoch 130/250
700/700 [==============================] - 0s 67us/sample - loss: 829.9402
Epoch 131/250
700/700 [==============================] - 0s 73us/sample - loss: 802.4260
Epoch 132/250
700/700 [==============================] - 0s 69us/sample - loss: 777.9863
Epoch 133/250
700/700 [==============================] - 0s 68us/sample - loss: 751.9605
Epoch 134/250
700/700 [==============================] - 0s 70us/sample - loss: 724.9629
Epoch 135/250
700/700 [==============================] - 0s 68us/sample - loss: 702.3697
Epoch 136/250
700/700 [==============================] - 0s 74us/sample - loss: 679.3208
Epoch 137/250
700/700 [==============================] - 0s 70us/sample - loss: 655.4713
Epoch 138/250
700/700 [==============================] - 0s 73us/sample - loss: 630.9569
Epoch 139/250
700/700 [==============================] - 0s 71us/sample - loss: 608.5175
Epoch 140/250
```

```
700/700 [==============================] - 0s 67us/sample - loss: 585.3118
Epoch 141/250
700/700 [==============================] - 0s 70us/sample - loss: 565.7603
Epoch 142/250
700/700 [==============================] - 0s 72us/sample - loss: 542.5649
Epoch 143/250
700/700 [==============================] - 0s 71us/sample - loss: 520.9780
Epoch 144/250
700/700 [==============================] - 0s 70us/sample - loss: 501.1878
Epoch 145/250
700/700 [==============================] - 0s 71us/sample - loss: 483.3268
Epoch 146/250
700/700 [==============================] - 0s 72us/sample - loss: 466.3138
Epoch 147/250
700/700 [==============================] - 0s 71us/sample - loss: 448.6379
Epoch 148/250
700/700 [==============================] - 0s 70us/sample - loss: 426.9134
Epoch 149/250
700/700 [==============================] - 0s 74us/sample - loss: 411.4831
Epoch 150/250
700/700 [==============================] - 0s 70us/sample - loss: 393.4419
Epoch 151/250
700/700 [==============================] - 0s 67us/sample - loss: 376.3282
Epoch 152/250
700/700 [==============================] - 0s 69us/sample - loss: 358.0180
Epoch 153/250
700/700 [==============================] - 0s 72us/sample - loss: 344.0322
Epoch 154/250
700/700 [==============================] - 0s 70us/sample - loss: 327.1666
Epoch 155/250
700/700 [==============================] - 0s 68us/sample - loss: 312.2332
Epoch 156/250
700/700 [==============================] - 0s 71us/sample - loss: 300.4993
Epoch 157/250
700/700 [==============================] - 0s 71us/sample - loss: 285.0985
Epoch 158/250
700/700 [==============================] - 0s 73us/sample - loss: 270.9902
Epoch 159/250
700/700 [==============================] - 0s 70us/sample - loss: 257.9236
Epoch 160/250
700/700 [==============================] - 0s 68us/sample - loss: 242.5731
Epoch 161/250
700/700 [==============================] - 0s 70us/sample - loss: 228.5526
Epoch 162/250
700/700 [==============================] - 0s 70us/sample - loss: 217.2323
Epoch 163/250
700/700 [==============================] - 0s 74us/sample - loss: 204.0378
Epoch 164/250
700/700 [==============================] - 0s 70us/sample - loss: 192.4924
Epoch 165/250
700/700 [==============================] - 0s 70us/sample - loss: 181.9402
Epoch 166/250
700/700 [==============================] - 0s 70us/sample - loss: 170.3420
Epoch 167/250
700/700 [==============================] - 0s 70us/sample - loss: 161.1426
Epoch 168/250
700/700 [==============================] - 0s 70us/sample - loss: 149.8766
Epoch 169/250
700/700 [==============================] - 0s 68us/sample - loss: 140.5106
Epoch 170/250
700/700 [==============================] - 0s 73us/sample - loss: 131.7382
Epoch 171/250
700/700 [==============================] - 0s 72us/sample - loss: 122.9225
Epoch 172/250
700/700 [==============================] - 0s 73us/sample - loss: 114.9365
Epoch 173/250
700/700 [==============================] - 0s 73us/sample - loss: 107.1022
Epoch 174/250
700/700 [==============================] - 0s 71us/sample - loss: 99.2103
Epoch 175/250
700/700 [==============================] - 0s 68us/sample - loss: 92.2679
Epoch 176/250
700/700 [==============================] - 0s 71us/sample - loss: 84.8423
Epoch 177/250
700/700 [==============================] - 0s 71us/sample - loss: 78.4221
Epoch 178/250
700/700 [==============================] - 0s 68us/sample - loss: 73.0699
Epoch 179/250
700/700 [==============================] - 0s 67us/sample - loss: 68.8864
Epoch 180/250
700/700 [==============================] - 0s 71us/sample - loss: 63.6099
Epoch 181/250
700/700 [==============================] - 0s 71us/sample - loss: 59.0425
Epoch 182/250
700/700 [==============================] - 0s 70us/sample - loss: 54.4923
Epoch 183/250
700/700 [==============================] - 0s 71us/sample - loss: 50.5602
Epoch 184/250
700/700 [==============================] - 0s 73us/sample - loss: 46.5751
```

```
Epoch 185/250
700/700 [==============================] - 0s 76us/sample - loss: 43.6547
Epoch 186/250
700/700 [==============================] - 0s 68us/sample - loss: 41.2024
Epoch 187/250
700/700 [==============================] - 0s 70us/sample - loss: 38.5470
Epoch 188/250
700/700 [==============================] - 0s 74us/sample - loss: 36.8927
Epoch 189/250
700/700 [==============================] - 0s 76us/sample - loss: 35.2618
Epoch 190/250
700/700 [==============================] - 0s 67us/sample - loss: 33.3906
Epoch 191/250
700/700 [==============================] - 0s 70us/sample - loss: 32.2971
Epoch 192/250
700/700 [==============================] - 0s 72us/sample - loss: 31.1117
Epoch 193/250
700/700 [==============================] - 0s 71us/sample - loss: 29.7924
Epoch 194/250
700/700 [==============================] - 0s 68us/sample - loss: 28.9351
Epoch 195/250
700/700 [==============================] - 0s 74us/sample - loss: 28.2614
Epoch 196/250
700/700 [==============================] - 0s 71us/sample - loss: 27.3432
Epoch 197/250
700/700 [==============================] - 0s 72us/sample - loss: 27.3696
Epoch 198/250
700/700 [==============================] - 0s 71us/sample - loss: 26.7970
Epoch 199/250
700/700 [==============================] - 0s 71us/sample - loss: 26.4991
Epoch 200/250
700/700 [==============================] - 0s 68us/sample - loss: 25.8932
Epoch 201/250
700/700 [==============================] - 0s 71us/sample - loss: 25.6222
Epoch 202/250
700/700 [==============================] - 0s 68us/sample - loss: 25.3937
Epoch 203/250
700/700 [==============================] - 0s 71us/sample - loss: 25.4349
Epoch 204/250
700/700 [==============================] - 0s 71us/sample - loss: 25.1963
Epoch 205/250
700/700 [==============================] - 0s 73us/sample - loss: 25.0983
Epoch 206/250
700/700 [==============================] - 0s 70us/sample - loss: 25.0211
Epoch 207/250
700/700 [==============================] - 0s 70us/sample - loss: 24.9933
Epoch 208/250
700/700 [==============================] - 0s 70us/sample - loss: 24.8173
Epoch 209/250
700/700 [==============================] - 0s 71us/sample - loss: 24.6101
Epoch 210/250
700/700 [==============================] - 0s 68us/sample - loss: 24.3901
Epoch 211/250
700/700 [==============================] - 0s 68us/sample - loss: 24.0644
Epoch 212/250
700/700 [==============================] - 0s 68us/sample - loss: 24.5279
Epoch 213/250
700/700 [==============================] - 0s 70us/sample - loss: 24.3929
Epoch 214/250
700/700 [==============================] - 0s 74us/sample - loss: 24.5920
Epoch 215/250
700/700 [==============================] - 0s 74us/sample - loss: 24.2903
Epoch 216/250
700/700 [==============================] - 0s 64us/sample - loss: 24.5403
Epoch 217/250
700/700 [==============================] - 0s 71us/sample - loss: 24.2580
Epoch 218/250
700/700 [==============================] - 0s 71us/sample - loss: 24.2239
Epoch 219/250
700/700 [==============================] - 0s 68us/sample - loss: 24.3374
Epoch 220/250
700/700 [==============================] - 0s 70us/sample - loss: 24.3673
Epoch 221/250
700/700 [==============================] - 0s 67us/sample - loss: 24.2660
Epoch 222/250
700/700 [==============================] - 0s 73us/sample - loss: 24.2365
Epoch 223/250
700/700 [==============================] - 0s 71us/sample - loss: 24.0305
Epoch 224/250
700/700 [==============================] - 0s 71us/sample - loss: 24.2925
Epoch 225/250
700/700 [==============================] - 0s 68us/sample - loss: 24.4897
Epoch 226/250
700/700 [==============================] - 0s 72us/sample - loss: 24.0593
Epoch 227/250
700/700 [==============================] - 0s 71us/sample - loss: 24.2092
Epoch 228/250
700/700 [==============================] - 0s 71us/sample - loss: 24.0628
Epoch 229/250
```

```
700/700 [==============================] - 0s 73us/sample - loss: 24.5156
Epoch 230/250
700/700 [==============================] - 0s 70us/sample - loss: 24.1111
Epoch 231/250
700/700 [==============================] - 0s 71us/sample - loss: 23.9795
Epoch 232/250
700/700 [==============================] - 0s 72us/sample - loss: 24.2185
Epoch 233/250
700/700 [==============================] - 0s 71us/sample - loss: 24.1126
Epoch 234/250
700/700 [==============================] - 0s 73us/sample - loss: 24.4198
Epoch 235/250
700/700 [==============================] - 0s 71us/sample - loss: 24.2241
Epoch 236/250
700/700 [==============================] - 0s 72us/sample - loss: 24.0720
Epoch 237/250
700/700 [==============================] - 0s 72us/sample - loss: 24.0071
Epoch 238/250
700/700 [==============================] - 0s 71us/sample - loss: 24.3686
Epoch 239/250
700/700 [==============================] - 0s 71us/sample - loss: 23.9888
Epoch 240/250
700/700 [==============================] - 0s 70us/sample - loss: 24.3374
Epoch 241/250
700/700 [==============================] - 0s 71us/sample - loss: 24.1188
Epoch 242/250
700/700 [==============================] - 0s 70us/sample - loss: 24.1437
Epoch 243/250
700/700 [==============================] - 0s 75us/sample - loss: 24.0613
Epoch 244/250
700/700 [==============================] - 0s 74us/sample - loss: 24.4479
Epoch 245/250
700/700 [==============================] - 0s 78us/sample - loss: 24.5535
Epoch 246/250
700/700 [==============================] - 0s 71us/sample - loss: 24.5116
Epoch 247/250
700/700 [==============================] - 0s 69us/sample - loss: 24.2632
Epoch 248/250
700/700 [==============================] - 0s 72us/sample - loss: 24.0891
Epoch 249/250
700/700 [==============================] - 0s 75us/sample - loss: 23.9893
Epoch 250/250
700/700 [==============================] - 0s 75us/sample - loss: 24.7449
```

Out[25]:
```
<tensorflow.python.keras.callbacks.History at 0x25bf824aa88>
```

Evaluation

Let's evaluate our performance on our training set and our test set. We can compare these two performances to check for overfitting.

In [26]:
```
model.history.history
```

Out[26]:
```
{'loss': [257046.02053571428,
  256727.57410714286,
  256522.99767857144,
  256373.04339285713,
  256208.80839285714,
  256017.87419642857,
  255794.63875,
  255536.61446428573,
  255246.50839285715,
  254920.9007142857,
  254556.80571428573,
  254155.80767857144,
  253712.6569642857,
  253222.941875,
  252682.845,
  252086.95839285714,
  251436.20464285713,
  250726.11482142858,
  249952.45232142857,
  249109.92089285713,
  248200.0200892857,
  247215.95776785715,
  246154.1205357143,
  245005.08839285714,
  243780.42348214285,
  242465.37642857144,
  241055.46339285714,
  239559.661875,
  237966.36696428573,
  236272.626875,
  234471.03642857142,
  232554.95428571428,
  230538.98651785715,
  228401.81401785713,
  226150.31267857144,
```

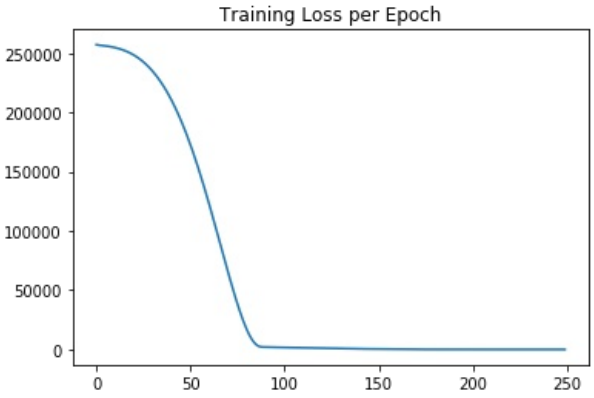
223783.81348214287,
221290.60571428572,
218679.40830357143,
215924.69125,
213063.56241071428,
210046.38044642858,
206900.92839285714,
203611.78616071428,
200231.52544642857,
196671.46205357142,
192974.43410714285,
189174.22910714286,
185202.64232142858,
181108.9913392857,
176864.39651785715,
172504.71794642857,
168003.58785714285,
163362.08651785715,
158614.57803571428,
153737.69098214287,
148756.86517857143,
143625.381875,
138388.08901785716,
133087.5436607143,
127688.31861607142,
122186.69504464285,
116597.92147321429,
110948.68290178571,
105236.5175,
99482.9940625,
93701.20227678571,
87938.2564732143,
82171.90580357143,
76383.89745535715,
70635.64622767858,
64962.11716517857,
59347.544352678575,
53863.82444196429,
48459.28073660714,
43236.19125,
38164.600323660714,
33331.563203125,
28683.768895089284,
24321.479464285716,
20278.469587053572,
16523.323091517857,
13159.072672991071,
10194.291685267857,
7640.408490513393,
5576.274188058036,
4007.2731919642856,
2910.798597935268,
2304.2849581473215,
2099.1053069196428,
2047.892582310268,
2011.3148486328125,
1977.7155691964285,
1943.7098918805802,
1906.4392619977677,
1873.7697509765626,
1839.494091796875,
1807.7019468470983,
1773.0187297712052,
1742.6450809151786,
1706.0910637555803,
1671.919691685268,
1640.9999825613838,
1610.9062388392856,
1579.4019182477678,
1545.4603473772322,
1518.577080078125,
1485.8637158203126,
1457.9632101004465,
1424.8217975725447,
1394.5545382254463,
1364.850046735491,
1337.1091573660715,
1305.234978376116,
1277.8466336495535,
1248.961134905134,
1218.9898291015625,
1188.1648325892857,
1159.2657979910714,
1131.0403955078125,
1098.9464425223214,
1068.8702406529019,
1034.1244656808035,
1004.9060944475447,
974.5797921316964,

949.073662109375,
925.6563427734375,
898.7897991071428,
876.443902936663,
846.4307498604911,
829.9401977539062,
802.4260260881697,
777.9862841796875,
751.9605353655134,
724.9629045758928,
702.3697220284598,
679.32080078125,
655.4712688337054,
630.956916155134,
608.5175149972098,
585.3117515345982,
565.7603156389509,
542.5648521205358,
520.9779725864955,
501.187763671875,
483.32678588867185,
466.3137942940848,
448.6378601074219,
426.9133588518415,
411.4831403459821,
393.4418805803571,
376.3281863839286,
358.0179701450893,
344.0322297014509,
327.16663940429686,
312.23322230747766,
300.4992583356585,
285.098472202846,
270.99015023367747,
257.9235995047433,
242.57313310895648,
228.55255998883928,
217.23230460030692,
204.03783900669643,
192.49236258370536,
181.94021946498327,
170.34200622558595,
161.14263549804687,
149.8765545654297,
140.51057817731584,
131.7381551688058,
122.92249625069755,
114.93649026053292,
107.10216094970703,
99.21032762799945,
92.26787187848772,
84.84226850237165,
78.42208199637277,
73.06991385323661,
68.88644858224052,
63.60992636544364,
59.04246122087751,
54.492334987095425,
50.56024594988142,
46.57507073538644,
43.65467989240374,
41.20235630580357,
38.54703225272043,
36.89274923052107,
35.26179874965123,
33.39060076032366,
32.297111097063336,
31.1116964503697,
29.79240364074707,
28.935144544328963,
28.26140598842076,
27.343234013148717,
27.36963134765625,
26.79698026384626,
26.499113333565848,
25.893150612967354,
25.62219348362514,
25.393682414463587,
25.434924599783763,
25.196334571838378,
25.098290459769114,
25.021088354928153,
24.9933164705549,
24.817250110081265,
24.610098855154856,
24.39006670815604,
24.064426999773296,
24.527889840262276,
24.3929073878697,

```
  24.59200389317104,
  24.290332630702427,
  24.540304859706335,
  24.25796793256487,
  24.22394993373326,
  24.33741833278111,
  24.367307946341377,
  24.26597143990653,
  24.236519066946848,
  24.03049336024693,
  24.29250689915248,
  24.48969031197684,
  24.059261234828405,
  24.20917403084891,
  24.062846450805665,
  24.515583092825754,
  24.111098273141042,
  23.9794596862793,
  24.218495330810548,
  24.112635650634765,
  24.419767041887557,
  24.22412246159145,
  24.0720441872733,
  24.00706672668457,
  24.368646545410158,
  23.988784593854632,
  24.33735505240304,
  24.118764997209823,
  24.143690828595844,
  24.061329509190152,
  24.44786605834961,
  24.55346090044294,
  24.511635371616908,
  24.26320677621024,
  24.089111475263323,
  23.989297572544643,
  24.744917046683174]}
```

In [27]:
```
loss = model.history.history['loss']
```

In [28]:
```
sns.lineplot(x=range(len(loss)),y=loss)
plt.title("Training Loss per Epoch");
```



Compare final evaluation (MSE) on training set and test set.

These should hopefully be fairly close to each other.

In [29]:
```
model.metrics_names
```

Out[29]:
```
['loss']
```

In [30]:
```
training_score = model.evaluate(X_train,y_train,verbose=0)
test_score = model.evaluate(X_test,y_test,verbose=0)
```

In [31]:
```
training_score
```

Out[31]:
```
24.55682439531599
```

In [32]:
```
test_score
```

Out[32]:
```
26.798187001546225
```

Further Evaluations

In [33]:
```
test_predictions = model.predict(X_test)
```

In [34]:
```
test_predictions
```

Out[34]:
```
array([[406.26343],
       [625.07983],
       [593.59314],
       [573.62714],
       [367.47824],
       [580.6027 ],
       [516.3    ],
       [460.22568],
       [550.58673],
       [448.60873],
       [613.2959 ],
       [550.30396],
       [420.10864],
       [409.8501 ],
       [652.86633],
       [438.35947],
       [509.70355],
       [661.5566 ],
       [664.221  ],
       [566.93195],
       [335.011  ],
       [445.90964],
       [383.43677],
       [379.66046],
       [567.987  ],
       [612.09296],
       [533.69293],
       [428.89386],
       [657.0607 ],
       [415.08276],
       [443.71158],
       [486.30923],
       [439.4752 ],
       [683.53503],
       [425.78995],
       [418.6874 ],
       [503.1647 ],
       [551.8273 ],
       [510.96967],
       [396.29428],
       [620.1216 ],
       [417.63516],
       [605.9426 ],
       [447.04782],
       [503.214  ],
       [583.16064],
       [670.66925],
       [491.6024 ],
       [319.3331 ],
       [486.60828],
       [518.5312 ],
       [382.71024],
       [543.24634],
       [409.57797],
       [643.1107 ],
       [492.27206],
       [629.48804],
       [628.5606 ],
       [448.15475],
       [485.87952],
       [492.33035],
       [475.69794],
       [684.4861 ],
       [404.25476],
       [702.8802 ],
       [587.84174],
       [584.55994],
       [539.17975],
       [485.75665],
       [517.82074],
       [362.08646],
       [542.08276],
       [572.03156],
       [529.7627 ],
       [454.82465],
       [532.44696],
       [508.45544],
       [444.4056 ],
       [544.70044],
       [642.3479 ],
       [467.1369 ],
       [568.65845],
       [692.49036],
       [459.848  ],
       [710.67194],
       [473.8478 ],
       [404.25616],
```

```
[586.54626],
[437.8215 ],
[490.16046],
[618.5658 ],
[440.54614],
[456.4309 ],
[436.31238],
[508.1484 ],
[609.9504 ],
[322.3879 ],
[437.33047],
[537.66205],
[519.9028 ],
[606.54333],
[526.7351 ],
[334.9314 ],
[577.5914 ],
[432.82236],
[563.9483 ],
[514.66473],
[392.0228 ],
[567.6409 ],
[455.911  ],
[449.57327],
[642.5517 ],
[525.6672 ],
[552.0467 ],
[418.70322],
[479.90872],
[587.9719 ],
[669.03143],
[702.20355],
[661.02466],
[561.94727],
[504.41043],
[391.29007],
[281.9948 ],
[480.78778],
[617.9032 ],
[374.2134 ],
[513.51825],
[512.27576],
[494.6914 ],
[481.63385],
[424.8122 ],
[494.66434],
[472.623  ],
[601.9391 ],
[575.005  ],
[415.9636 ],
[632.2133 ],
[467.5123 ],
[565.7332 ],
[406.6662 ],
[533.484  ],
[574.05035],
[357.92883],
[551.1896 ],
[604.85095],
[385.2252 ],
[543.72565],
[563.984  ],
[453.99207],
[633.7756 ],
[373.1417 ],
[475.4092 ],
[530.16583],
[373.19534],
[462.35925],
[437.36887],
[499.71695],
[346.94308],
[396.24692],
[606.0831 ],
[507.89963],
[469.75818],
[491.5713 ],
[536.60364],
[345.31842],
[513.5915 ],
[251.38142],
[505.32523],
[542.43054],
[490.56903],
[472.24612],
[393.4955 ],
[417.26862],
[550.93176],
[476.95383],
```

[581.37994],
[490.9674],
[602.4701],
[548.3496],
[543.30194],
[501.8112],
[647.44324],
[561.8009],
[579.37555],
[445.2174],
[416.52985],
[421.0515],
[570.5346],
[610.52203],
[438.8255],
[489.1685],
[589.20416],
[526.6729],
[358.21768],
[647.02264],
[529.33215],
[338.01492],
[493.89966],
[411.1523],
[608.1084],
[347.3649],
[523.7079],
[405.67038],
[259.00522],
[521.17804],
[341.71494],
[362.9102],
[577.8506],
[417.68906],
[552.3458],
[522.365],
[512.0057],
[325.52286],
[405.12326],
[603.8921],
[619.39307],
[604.7606],
[567.6965],
[474.58182],
[461.4935],
[509.68094],
[446.99258],
[512.57574],
[503.60657],
[401.1072],
[607.0318],
[258.73117],
[629.81726],
[590.60266],
[327.92664],
[480.71942],
[596.82367],
[379.30722],
[461.27716],
[325.78357],
[519.9732],
[410.61462],
[557.62585],
[643.49084],
[538.1467],
[504.41574],
[636.6225],
[516.51086],
[533.0186],
[520.09735],
[458.5575],
[507.0747],
[462.23328],
[593.1741],
[467.04837],
[428.21533],
[542.92456],
[495.01172],
[681.3526],
[373.66132],
[552.9735],
[579.3976],
[435.0165],
[544.3619],
[587.45667],
[580.6866],
[722.74384],
[433.78827],
[399.55917],

```
       [314.75165],
       [449.3715 ],
       [389.0275 ],
       [544.5414 ],
       [524.0208 ],
       [565.89056],
       [449.10574],
       [535.72797],
       [382.709  ],
       [502.7475 ],
       [638.6875 ],
       [497.77362],
       [569.79364],
       [471.25647],
       [273.95135],
       [518.6328 ],
       [622.9433 ],
       [351.34857],
       [451.53647],
       [500.51117],
       [544.3127 ],
       [613.3484 ],
       [389.1045 ],
       [450.48734],
       [483.59534],
       [599.79675],
       [500.60007],
       [322.21704],
       [556.5154 ],
       [445.71307],
       [530.3348 ],
       [516.57184],
       [611.15717],
       [417.96445],
       [411.85953]], dtype=float32)
```

In [35]:
```
pred_df = pd.DataFrame(y_test,columns=['Test Y'])
```

In [36]:
```
pred_df
```

Out[36]:

	Test Y
0	402.296319
1	624.156198
2	582.455066
3	578.588606
4	371.224104
...	...
295	525.704657
296	502.909473
297	612.727910
298	417.569725
299	410.538250

300 rows × 1 columns

In [37]:
```
test_predictions = pd.Series(test_predictions.reshape(300,))
```

In [38]:
```
test_predictions
```

Out[38]:
```
0      406.263428
1      625.079834
2      593.593140
3      573.627136
4      367.478241
          ...
295    530.334778
296    516.571838
297    611.157166
298    417.964447
299    411.859528
Length: 300, dtype: float32
```

In [39]:
```
pred_df = pd.concat([pred_df,test_predictions],axis=1)
```

In [40]:
```
pred_df.columns = ['Test Y','Model Predictions']
```

In [41]:
```
pred_df
```

Out[41]:

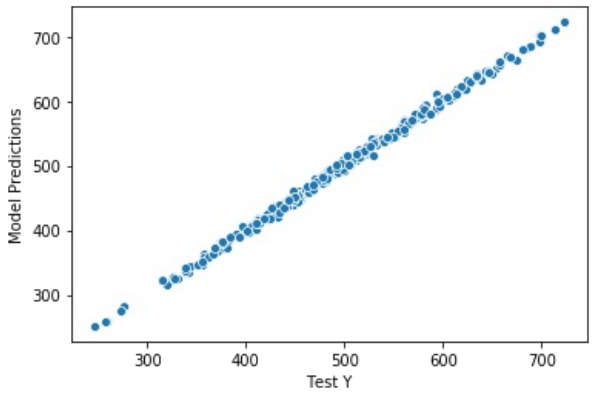
	Test Y	Model Predictions
0	402.296319	406.263428
1	624.156198	625.079834
2	582.455066	593.593140
3	578.588606	573.627136
4	371.224104	367.478241
...	...	...
295	525.704657	530.334778
296	502.909473	516.571838
297	612.727910	611.157166
298	417.569725	417.964447
299	410.538250	411.859528

300 rows × 2 columns

Let's compare to the real test labels!

In [42]:
```
sns.scatterplot(x='Test Y',y='Model Predictions',data=pred_df)
```

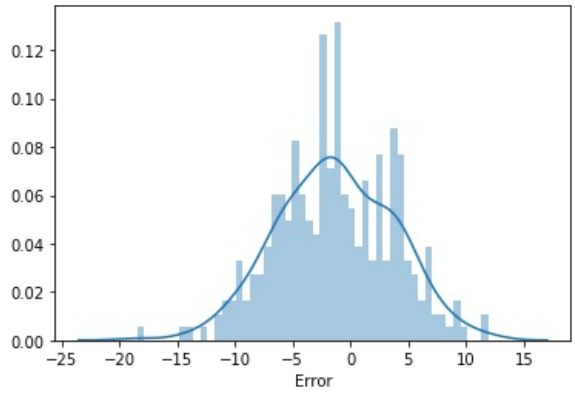
Out[42]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x2599b2824c8>
```



In [43]:
```
pred_df['Error'] = pred_df['Test Y'] - pred_df['Model Predictions']
```

In [44]:
```
sns.distplot(pred_df['Error'],bins=50)
```

Out[44]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x2599b269048>
```



In [45]:
```
from sklearn.metrics import mean_absolute_error,mean_squared_error
```

In [46]:
```
mean_absolute_error(pred_df['Test Y'],pred_df['Model Predictions'])
```

Out[46]:
```
4.197453664752528
```

In [47]:
```
mean_squared_error(pred_df['Test Y'],pred_df['Model Predictions'])
```

Out[47]:
```
26.798188995783907
```

In [48]:
```
# Essentially the same thing, difference just due to precision
test_score
```

Out[48]:
```
26.798187001546225
```

In [49]:
```
#RMSE
```

```
test_score**0.5
```

Out[49]:
```
5.176696533654085
```

Predicting on brand new data

What if we just saw a brand new gemstone from the ground? What should we price it at? This is the **exact** same procedure as predicting on a new test data!

In [50]:
```
# [[Feature1, Feature2]]
new_gem = [[998,1000]]
```

In [51]:
```
# Don't forget to scale!
scaler.transform(new_gem)
```

Out[51]:
```
array([[0.14117652, 0.53968792]])
```

In [52]:
```
new_gem = scaler.transform(new_gem)
```

In [53]:
```
model.predict(new_gem)
```

Out[53]:
```
array([[420.68692]], dtype=float32)
```

Saving and Loading a Model

In [54]:
```
from tensorflow.keras.models import load_model
```

In [55]:
```
model.save('my_model.h5')  # creates a HDF5 file 'my_model.h5'
```

In [57]:
```
later_model = load_model('my_model.h5')
```

```
WARNING:tensorflow:Sequential models without an `input_shape` passed to the first layer cannot reload their opt
imizer state. As a result, your model isstarting with a freshly initialized optimizer.
```

In [58]:
```
later_model.predict(new_gem)
```

Out[58]:
```
array([[420.68692]], dtype=float32)
```