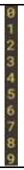
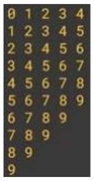
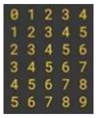
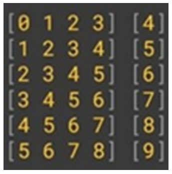
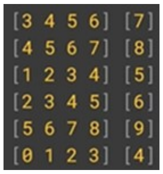
Single Layer Neural Network for Time Series, TensorFlow & Keras

window size	Number of values that will treat as our feature, where we're taking a window of the data and training an ML model to predict the next value.
tf.data.Dataset	Object represents a sequence of elements, in which each element contains one or more **Tensors** / A class to create e.g. (tf.data.Dataset.range (10) -create some data for us, we'll make a range of 10 values. When we print them we'll see a series of data from 0 to 9)  0 1 2 3 4 5 6 7 8 9
~~dataset.~~window ~~dataset~~ (Variable name)	To ***expand our data set using Windowing***. Its parameters are the size of the window and how much we want to shift by each time. > E.g. dataset.window(5,shift=1) > E.g. dataset.window (5, shift=1, drop_remainder=True) i.e. To make sure we have regular sized data, we need to make use of additional parameter called drop_remainder. And if we set it to true, it will truncate the data by dropping all of the remainders. This means it will only give us windows of five items.  0 1 2 3 4 1 2 3 4 5 2 3 4 5 6 3 4 5 6 7 4 5 6 7 8 5 6 7 8 9 6 7 8 9 7 8 9 8 9 9  0 1 2 3 4 1 2 3 4 5 2 3 4 5 6 3 4 5 6 7 4 5 6 7 8 5 6 7 8 9 < drop_remainder
flat_map	***Maps map_func across this dataset and flattens the result.*** [[1, 2, 3], [4, 5, 6], [7, 8, 9]] >> [1, 2, 3, 4, 5, 6, 7, 8, 9]
buffer_size	Representing the number of elements from this dataset from which the new dataset will sample.
~~dataset~~.shuffle(buffer_size=10)	***Randomly shuffles the elements of this dataset.*** This dataset fills a buffer with buffer_size elements, then randomly samples elements from this buffer, replacing the selected elements with new elements. For perfect shuffling, a buffer size greater than or equal to the full size of the dataset is required.  [0 1 2 3] [4] [1 2 3 4] [5] [2 3 4 5] [6] [3 4 5 6] [7] [4 5 6 7] [8] [5 6 7 8] [9] >>>  [3 4 5 6] [7] [4 5 6 7] [8] [1 2 3 4] [5] [2 3 4 5] [6] [5 6 7 8] [9] [0 1 2 3] [4] In practice: Shuffle the dataset before training using the shuffle method. We call it with the buffer size of ten, because that's the amount of data items that we have.

batch	***Combines consecutive elements of this dataset into batches.***
	>>> dataset = tf.data.Dataset.range(8)
	>>> dataset = dataset.batch(3)
	>>> list(dataset.as_numpy_iterator())
	[array([0, 1, 2]), array([3, 4, 5]), array([6, 7])]
	>>> dataset = tf.data.Dataset.range(8)
	>>> dataset = dataset.batch(3, drop_remainder=True)
	>>> list(dataset.as_numpy_iterator())
	[array([0, 1, 2]), array([3, 4, 5])]
	The components of the resulting element will have an additional outer dimension, which will be batch_size (or N % batch_size for the last element if batch_size does not divide the number of input elements N evenly and drop_remainder is False). If your program depends on the batches having the same outer dimension, you should set the drop_remainder argument to True to prevent the smaller batch from being produced.

prefetch	***Creates a Dataset that prefetches elements from this dataset.*** || Most dataset input pipelines should end with a call to prefetch. This allows later elements to be prepared while the current element is being processed. This often improves latency and throughput, at the cost of using additional memory to store prefetched elements. Note: Like other Dataset methods, prefetch operates on the elements of the input dataset. It has no concept of examples vs. batches. examples.prefetch(2) will prefetch two elements (2 examples), while examples.batch(20).prefetch(2) will prefetch 2 elements (2 batches, of 20 examples each).

<table>
<tr>
<td></td>
<td>

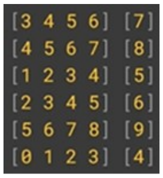
```
>>> dataset = tf.data.Dataset.range(3)
>>> dataset = dataset.prefetch(2)
>>> list(dataset.as_numpy_iterator())
[0, 1, 2]
```

</td>
</tr>
<tr>
<td>

~~dataset~~.batch(2).prefetch(1)

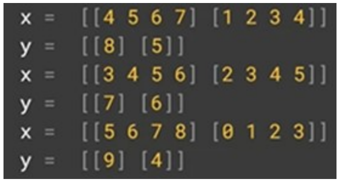
</td>
<td>

Batching the data, this is done with the batch method. It'll take a size parameter, and in this case it's 2. So what we'll do is we'll batch the data into sets of two, and if we print them out, we'll see this. We now have three batches of two data items each. And if you look at the first set, you'll see the corresponding x and y.



```
[3 4 5 6] [7]
[4 5 6 7] [8]
[1 2 3 4] [5]
[2 3 4 5] [6]
[5 6 7 8] [9]
[0 1 2 3] [4]
```

>>>



```
x =  [[4 5 6 7] [1 2 3 4]]
y =  [[8] [5]]
x =  [[3 4 5 6] [2 3 4 5]]
y =  [[7] [6]]
x =  [[5 6 7 8] [0 1 2 3]]
y =  [[9] [4]]
```

</td>
</tr>
<tr>
<td>

np.where

</td>
<td>

where(condition, [x, y])

Return elements chosen from x or y depending on condition.

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.where(a < 5, a, 10*a)
array([ 0,  1,  2,  3,  4, 50, 60, 70, 80, 90])
```

</td>
</tr>
<tr>
<td>

np.cos

</td>
<td>

Cosine element-wise.

</td>
</tr>
<tr>
<td>

np.pi

</td>
<td>

Returns pi value

</td>
</tr>
<tr>
<td>

np.exp

</td>
<td>

Calculate the exponential of all elements in the input array.

</td>
</tr>
<tr>
<td>

np.random.RandomState (seed)

</td>
<td>

np.random.RandomState () - a class that provides several methods based on different probability distributions. **np.random.RandomState.seed** () - called when RandomState () is initialised.

```
>>> rng = np.random.RandomState(42)
>>> rng.randn(4)
array([ 0.49671415, -0.1382643 ,  0.64768854,  1.52302986])
```

</td>
</tr>
<tr>
<td>

from_tensor_slices

</td>
<td>

Creates a Dataset whose elements are slices of the given tensors.

The given tensors are sliced along their first dimension. This operation preserves the structure of the input tensors, removing the first dimension of each tensor and using it as the dataset dimension. All input tensors must have the same size in their first dimensions.

```
>>> # Slicing a 1D tensor produces scalar tensor elements.
>>> dataset = tf.data.Dataset.from_tensor_slices([1, 2, 3])
>>> list(dataset.as_numpy_iterator())
[1, 2, 3]
>>> # Slicing a 2D tensor produces 1D tensor elements.
>>> dataset = tf.data.Dataset.from_tensor_slices([[1, 2], [3, 4]])
>>> list(dataset.as_numpy_iterator())
[array([1, 2], dtype=int32), array([3, 4], dtype=int32)]
>>> # Slicing a tuple of 1D tensors produces tuple elements containing
>>> # scalar tensors.
>>> dataset = tf.data.Dataset.from_tensor_slices(([1, 2], [3, 4], [5, 6]))
>>> list(dataset.as_numpy_iterator())
[(1, 3, 5), (2, 4, 6)]
```

</td>
</tr>
<tr>
<td>

Layers API

Dense layer

tf.keras.layers.Dense()

Ref:

Dense layer (keras.io)

</td>
<td>

to create a single dense layer

tf.keras.layers.Dense(1, input_shape=[window_size])

Dense **class**

```
tf.keras.layers.Dense(
    units,
    activation=None,
    use_bias=True,
    kernel_initializer="glorot_uniform",
    bias_initializer="zeros",
    kernel_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None,
    bias_constraint=None,
    **kwargs
)
```

</td>
</tr>
</table>

Just your regular densely-connected NN layer.

`Dense` implements the operation: `output = activation(dot(input, kernel) + bias)` where `activation` is the element-wise activation function passed as the `activation` argument, `kernel` is a weights matrix created by the layer, and `bias` is a bias vector created by the layer (only applicable if `use_bias` is `True`). These are all attributes of `Dense`.

Example

```
>>> # Create a `Sequential` model and add a Dense layer as the first layer.
>>> model = tf.keras.models.Sequential()
>>> model.add(tf.keras.Input(shape=(16,)))
>>> model.add(tf.keras.layers.Dense(32, activation='relu'))
>>> # Now the model will take as input arrays of shape (None, 16)
>>> # and output arrays of shape (None, 32).
>>> # Note that after the first layer, you don't need to specify
>>> # the size of the input anymore:
>>> model.add(tf.keras.layers.Dense(32))
>>> model.output_shape
(None, 32)
```

Arguments

- **units**: Positive integer, dimensionality of the output space.
- **activation**: Activation function to use. If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix.
- **bias_initializer**: Initializer for the bias vector.
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix.
- **bias_regularizer**: Regularizer function applied to the bias vector.
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation").
- **kernel_constraint**: Constraint function applied to the `kernel` weights matrix.
- **bias_constraint**: Constraint function applied to the bias vector.

Input shape
N-D tensor with shape: `(batch_size, ..., input_dim)`. The most common situation would be a 2D input with shape `(batch_size, input_dim)`.
Output shape
N-D tensor with shape: `(batch_size, ..., units)`. For instance, for a 2D input with shape `(batch_size, input_dim)`, the output would have shape `(batch_size, units)`.

What is a Dense Layer in Neural Network?
The dense layer is a neural network layer that is connected deeply, which means each neuron in the dense layer receives input from all neurons of its previous layer. The dense layer is found to be the most commonly used layer in the models.
In the background, the dense layer performs a matrix-vector multiplication. The values used in the matrix are actually parameters that can be trained and updated with the help of backpropagation.
The output generated by the dense layer is an 'm' dimensional vector. Thus, dense layer is basically used for changing the dimensions of the vector. Dense layers also applies operations like rotation, scaling, translation on the vector.

Keras Dense Layer Parameters
Let us see different parameters of dense layer function of Keras below –

1. Units
The **most basic parameter** of all the parameters, it uses positive integer as it value and represents the **output size** of the layer.
It is the unit parameter itself that plays a major role in the **size of the weight matrix** along with the **bias vector**.

2. Activation

The activation parameter is helpful in applying the element-wise activation function in a dense layer. By default, Linear Activation is used but we can alter and switch to any one of many options that Keras provides for this.

3. Use_Bias

Another straightforward parameter, **use_bias** helps in deciding whether we should include a bias vector for calculation purposes or not. By default, **use_bias** is set to true.

4. Initializers

As its name suggests, the initializer parameter is used for providing input about how values in the layer will be initialized. In case of the Dense Layer, the weight matrix and bias vector has to be initialized.

5. Regularizers

Regularizers contain three parameters that carry out regularization or penalty on the model. Generally, these parameters are not used regularly but they can help in the generalization of the model.

6. Constraints

This last parameter determines the constraints on the values that the weight matrix or bias vector can take.

Keras Dense Layer Operation

The dense layer function of Keras implements following operation –

output = activation(dot(input, kernel) + bias)

In the above equation, **activation** is used for performing **element-wise activation** and the **kernel** is the **weights matrix** created by the layer, and **bias** is a bias vector created by the layer.

Keras dense layer on the output layer performs **dot product** of **input tensor** and **weight kernel matrix**.

A bias vector is added and element-wise activation is performed on output values.

Models API

The Sequential class

tf.keras.models.Sequential()

Ref:

The Sequential class (keras.io)

Simply define my model as a sequential containing the sole layer.

`Sequential` **class**

tf.keras.Sequential(layers=None, name=None)

`Sequential` groups a linear stack of layers into a `tf.keras.Model`.

`Sequential` provides training and inference features on this model.

Examples

```
>>> # Optionally, the first layer can receive an `input_shape` argument:
>>> model = tf.keras.Sequential()
>>> model.add(tf.keras.layers.Dense(8, input_shape=(16,)))
>>> # Afterwards, we do automatic shape inference:
>>> model.add(tf.keras.layers.Dense(4))
```

```
>>> # This is identical to the following:
>>> model = tf.keras.Sequential()
>>> model.add(tf.keras.Input(shape=(16,)))
>>> model.add(tf.keras.layers.Dense(8))
```

```
>>> # Note that you can also omit the `input_shape` argument.
>>> # In that case the model doesn't have any weights until the first call
>>> # to a training/evaluation method (since it isn't yet built):
>>> model = tf.keras.Sequential()
>>> model.add(tf.keras.layers.Dense(8))
>>> model.add(tf.keras.layers.Dense(4))
>>> # model.weights not created yet
>>> # Whereas if you specify the input shape, the model gets built
>>> # continuously as you are adding layers:
>>> model = tf.keras.Sequential()
>>> model.add(tf.keras.layers.Dense(8, input_shape=(16,)))
>>> model.add(tf.keras.layers.Dense(4))
>>> len(model.weights)
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 = tf.keras.Sequential()
>>> model.add(tf.keras.layers.Dense(8))
>>> model.add(tf.keras.layers.Dense(4))
>>> model.build((None, 16))
>>> len(model.weights)
4
```

```
# Note that when using the delayed-build pattern (no input shape specified),
# the model gets built the first time you call `fit`, `eval`, or `predict`,
# or the first time you call the model on some input data.
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(8))
model.add(tf.keras.layers.Dense(1))
model.compile(optimizer='sgd', loss='mse')
# This builds the model for the first time:
model.fit(x, y, batch_size=32, epochs=10)
```

`add` method

```
Sequential.add(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).

`pop` method

```
Sequential.pop()
```

Removes the last layer in the model.

Raises

- **TypeError**: if there are no layers in the model.

Optimizers

SGD

tf.keras.optimizers.SGD()

Ref:
Regression metrics (keras.io)

Optimizers a built in module.

Gradient descent (with momentum) optimizer.

`SGD` class

```
tf.keras.optimizers.SGD(
    learning_rate=0.01, momentum=0.0, nesterov=False, name="SGD", **kwargs
)
```

Gradient descent (with momentum) optimizer.
Update rule for parameter `w` with gradient `g` when `momentum` is 0:

```
w = w - learning_rate * g
```

Update rule when `momentum` is larger than 0:

```
velocity = momentum * velocity - learning_rate * g
w = w + velocity
```

When `nesterov=True`, this rule becomes:

```
velocity = momentum * velocity - learning_rate * g
w = w + momentum * velocity - learning_rate * g
```

Arguments

- **learning_rate**: A `Tensor`, floating point value, or a schedule that is a `tf.keras.optimizers.schedules.LearningRateSchedule`, or a callable that takes no arguments and returns the actual value to use. The learning rate. Defaults to 0.01.
- **momentum**: float hyperparameter >= 0 that accelerates gradient descent in the relevant direction and dampens oscillations. Defaults to 0, i.e., vanilla gradient descent.
- **nesterov**: boolean. Whether to apply Nesterov momentum. Defaults to `False`.
- **name**: Optional name prefix for the operations created when applying gradients. Defaults to `"SGD"`.
- ****kwargs**: Keyword arguments. Allowed to be one of `"clipnorm"` or `"clipvalue"`. `"clipnorm"` (float) clips gradients by norm; `"clipvalue"` (float) clips gradients by value.

Usage:

```
>>> opt = tf.keras.optimizers.SGD(learning_rate=0.1)
>>> var = tf.Variable(1.0)
>>> loss = lambda: (var ** 2)/2.0         # d(loss)/d(var1) = var1
>>> step_count = opt.minimize(loss, [var]).numpy()
>>> # Step is `- learning_rate * grad`
>>> var.numpy()
0.9
```

```
>>> opt = tf.keras.optimizers.SGD(learning_rate=0.1, momentum=0.9)
>>> var = tf.Variable(1.0)
>>> val0 = var.value()
>>> loss = lambda: (var ** 2)/2.0         # d(loss)/d(var1) = var1
>>> # First step is `- learning_rate * grad`
>>> step_count = opt.minimize(loss, [var]).numpy()
>>> val1 = var.value()
>>> (val0 - val1).numpy()
0.1
```

```
>>> # On later steps, step-size increases because of momentum
>>> step_count = opt.minimize(loss, [var]).numpy()
>>> val2 = var.value()
>>> (val1 - val2).numpy()
0.18
```

Metrics

tf.keras.metrics.mean_absolute_error()

Ref:
[Regression metrics (keras.io)]

Built-in metrics. | Class MeanAbsoluteError: Computes the mean absolute error between the labels and predictions.

Regression metrics

`MeanAbsoluteError` **class**

```
tf.keras.metrics.MeanAbsoluteError(name="mean_absolute_error", dtype=None)
```

Computes the mean absolute error between the labels and predictions.

Arguments

- **name**: (Optional) string name of the metric instance.
- **dtype**: (Optional) data type of the metric result.

Standalone usage:

```
>>> m = tf.keras.metrics.MeanAbsoluteError()
>>> m.update_state([[0, 1], [0, 0]], [[1, 1], [0, 0]])
>>> m.result().numpy()
0.25
>>> m.reset_state()
>>> m.update_state([[0, 1], [0, 0]], [[1, 1], [0, 0]],
...                sample_weight=[1, 0])
>>> m.result().numpy()
0.5
```

Usage with `compile()` API:

```
model.compile(
    optimizer='sgd',
    loss='mse',
    metrics=[tf.keras.metrics.MeanAbsoluteError()])
```

Models API

model.compile()

Ref:
Model training APIs (keras.io)

compile method

```
Model.compile(
    optimizer="rmsprop",
    loss=None,
    metrics=None,
    loss_weights=None,
    weighted_metrics=None,
    run_eagerly=None,
    steps_per_execution=None,
    **kwargs
)
```

Configures the model for training.

Example

```
model.compile(optimizer=tf.keras.optimizer.Adam(learning_rate=1e-3),
              loss=tf.keras.losses.BinaryCrossentropy(),
              metrics=[tf.keras.metrics.BinaryAccuracy(),
                       tf.keras.metrics.FalseNegatives()])
```

Arguments

- **optimizer**: String (name of optimizer) or optimizer instance. See `tf.keras.optimizers`.
- **loss**: Loss function. Maybe be a string (name of loss function), or a `tf.keras.losses.Loss` instance. See `tf.keras.losses`. A loss function is any callable with the signature `loss = fn(y_true, y_pred)`, where `y_true` are the ground truth values, and `y_pred` are the model's predictions. `y_true` should have shape `(batch_size, d0, .. dN)` (except in the case of sparse loss functions such as sparse categorical crossentropy which expects integer arrays of shape `(batch_size, d0, .. dN-1)`). `y_pred` should have shape `(batch_size, d0, .. dN)`. The loss function should return a float tensor. If a custom `Loss` instance is used and reduction is set to `None`, return value has shape `(batch_size, d0, .. dN-1)` i.e. per-sample or per-timestep loss values; otherwise, it is a scalar. 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, unless `loss_weights` is specified.
- **metrics**: List of metrics to be evaluated by the model during training and testing. Each of this can be a string (name of a built-in function), function or a `tf.keras.metrics.Metric` instance. See `tf.keras.metrics`. Typically you will use `metrics=['accuracy']`. A function is any callable with the signature `result = fn(y_true, y_pred)`. 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 to specify a metric or a list of metrics for each output, such as `metrics=[['accuracy'], ['accuracy', 'mse']]` or `metrics=['accuracy', ['accuracy', 'mse']]`. When you pass the strings 'accuracy' or 'acc', we convert this to one of `tf.keras.metrics.BinaryAccuracy`, `tf.keras.metrics.CategoricalAccuracy`, `tf.keras.metrics.SparseCategoricalAccuracy` based on the loss function used and the model output shape. We do a similar conversion for the strings 'crossentropy' and 'ce' as well.
- **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 dict, it is expected to map output names (strings) to scalar coefficients.
- **weighted_metrics**: List of metrics to be evaluated and weighted by `sample_weight` or `class_weight` during training and testing.
- **run_eagerly**: Bool. Defaults to `False`. If `True`, this `Model`'s logic will not be wrapped in a `tf.function`. Recommended to leave this as `None` unless your `Model` cannot be run inside a `tf.function`. `run_eagerly=True` is not supported when using `tf.distribute.experimental.ParameterServerStrategy`.
- **steps_per_execution**: Int. Defaults to 1. The number of batches to run during each `tf.function` call. Running multiple batches inside a single `tf.function` call can greatly improve performance on TPUs or small models with a large Python overhead. At

most, one full epoch will be run each execution. If a number larger than the size of the epoch is passed, the execution will be truncated to the size of the epoch. Note that if steps_per_execution is set to N, Callback.on_batch_begin and Callback.on_batch_end methods will only be called every N batches (i.e. before/after each tf.function execution).

- ****kwargs**: Arguments supported for backwards compatibility only.

Raises

- **ValueError**: In case of invalid arguments for optimizer, loss or metrics.

model.fit()

Ref:
[Model training APIs (keras.io)]()

fit method

```
Model.fit(
    x=None,
    y=None,
    batch_size=None,
    epochs=1,
    verbose="auto",
    callbacks=None,
    validation_split=0.0,
    validation_data=None,
    shuffle=True,
    class_weight=None,
    sample_weight=None,
    initial_epoch=0,
    steps_per_epoch=None,
    validation_steps=None,
    validation_batch_size=None,
    validation_freq=1,
    max_queue_size=10,
    workers=1,
    use_multiprocessing=False,
)
```

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).
  - A tf.keras.utils.experimental.DatasetCreator, which wraps a callable that takes a single argument of type tf.distribute.InputContext, and returns a tf.data.Dataset. DatasetCreator should be used when users prefer to specify the per-replica batching and sharding logic for the Dataset. See tf.keras.utils.experimental.DatasetCreator doc for more information. A more detailed description of unpacking behavior for iterator types (Dataset, generator, Sequence) is given below. If using tf.distribute.experimental.ParameterServerStrategy, only DatasetCreator type is supported for x.
- **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 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**: 'auto', 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = one line per epoch. 'auto' defaults to 1 for most cases, but 2 when used with `ParameterServerStrategy`. 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`. Note `tf.keras.callbacks.ProgbarLogger` and `tf.keras.callbacks.History` callbacks are created automatically and need not be passed into `model.fit`. `tf.keras.callbacks.ProgbarLogger` is created or not based on `verbose` argument to `model.fit`. Callbacks with batch-level calls are currently unsupported with `tf.distribute.experimental.ParameterServerStrategy`, and users are advised to implement epoch-level calls instead with an appropriate `steps_per_epoch` value.
- **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_split` is not yet supported with `tf.distribute.experimental.ParameterServerStrategy`.
- **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. Thus, note the fact that the validation loss of data provided using `validation_split` or `validation_data` is not affected by regularization layers like noise and dropout. `validation_data` will override `validation_split`. `validation_data` could be: - A tuple `(x_val, y_val)` of Numpy arrays or tensors. - A tuple `(x_val, y_val, val_sample_weights)` of NumPy arrays. - A `tf.data.Dataset`. - A Python generator or `keras.utils.Sequence` returning `(inputs, targets)` or `(inputs, targets, sample_weights)`. `validation_data` is not yet supported with `tf.distribute.experimental.ParameterServerStrategy`.
- **shuffle**: Boolean (whether to shuffle the training data before each epoch) or str (for 'batch'). This argument is ignored when `x` is a generator or an object of tf.data.Dataset. '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. 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. When passing an infinitely repeating dataset, you must specify the `steps_per_epoch` argument. If `steps_per_epoch=-1` the training will run indefinitely with an infinitely repeating dataset. This argument is not supported with array inputs. When using `tf.distribute.experimental.ParameterServerStrategy`:
  * `steps_per_epoch=None` is not supported.
- **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_steps' is None, validation will run until the `validation_data` dataset is exhausted. In the case of an infinitely repeated dataset, it will run into an infinite loop. If 'validation_steps' is specified and only part of the dataset will be consumed, the evaluation will start from the beginning of the dataset at each epoch. This ensures that the same validation samples are used every time. - **validation_batch_size**: Integer or `None`. Number of samples per validation batch. If unspecified, will default to `batch_size`. Do not specify the `validation_batch_size` if your data is in the form of datasets, generators, or `keras.utils.Sequence` instances (since they generate batches). - **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. - **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. Unpacking behavior for iterator-like inputs: A common pattern is to pass a tf.data.Dataset, generator, or tf.keras.utils.Sequence to the `x` argument of fit, which will in fact yield not only features (x) but optionally targets (y) and sample weights. Keras requires that the output of such iterator-likes be unambiguous. The iterator should return a tuple of length 1, 2, or 3, where the optional second and third elements will be used for y and sample_weight respectively. Any other type provided will be wrapped in a length one tuple, effectively treating everything as 'x'. When yielding dicts, they should still adhere to the top-level tuple structure. e.g. `({"x0": x0, "x1": x1}, y)`. Keras will not attempt to separate features, targets, and weights from the keys of a single dict. A notable unsupported data type is the namedtuple. The reason is that it behaves like both an ordered datatype (tuple) and a mapping datatype (dict). So given a namedtuple of the form: `namedtuple("example_tuple", ["y", "x"])` it is ambiguous whether to reverse the order of the elements when interpreting the value. Even worse is a tuple of the form: `namedtuple("other_tuple", ["x", "y", "z"])` where it is unclear if the tuple was intended to be unpacked into x, y, and sample_weight or passed through as a single element to `x`. As a result the data processing code will simply raise a ValueError if it encounters a namedtuple. (Along with instructions to remedy the issue.) **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**: 1. If the model was never compiled or, 2. If `model.fit` is wrapped in `tf.function`. - **ValueError**: In case of mismatch between the provided input data and what the model expects or when the input data is empty.

model.evaluate()	**`evaluate` method**
	`Model.evaluate(`
	`    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,
    return_dict=False,
    **kwargs
)
```

Returns the loss value & metrics values for the model in test mode.

Computation is done in batches (see the `batch_size` arg.)

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)`. A more detailed description of unpacking behavior for iterator types (Dataset, generator, Sequence) is given in the `Unpacking behavior for iterator-like inputs` section of `Model.fit`.
- **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 batch of computation. If unspecified, `batch_size` will default to 32. Do not specify the `batch_size` if your data is in the form of a 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. 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.
- **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.
- **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

<table>
<tr>
<td></td>
<td>

should not pass non-picklable arguments to the generator as they can't be passed easily to children processes.

- **return_dict**: If `True`, loss and metric results are returned as a dict, with each key being the name of the metric. If `False`, they are returned as a list.
- ****kwargs**: Unused at this time.

See the discussion of `Unpacking behavior for iterator-like inputs` for `Model.fit`.

`Model.evaluate` is not yet supported with `tf.distribute.experimental.ParameterServerStrategy`.

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

- **RuntimeError**: If `model.evaluate` is wrapped in `tf.function`.
- **ValueError**: in case of invalid arguments.

</td>
</tr>
<tr>
<td>

model.predict()

</td>
<td>

`predict` method

```
Model.predict(
    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.

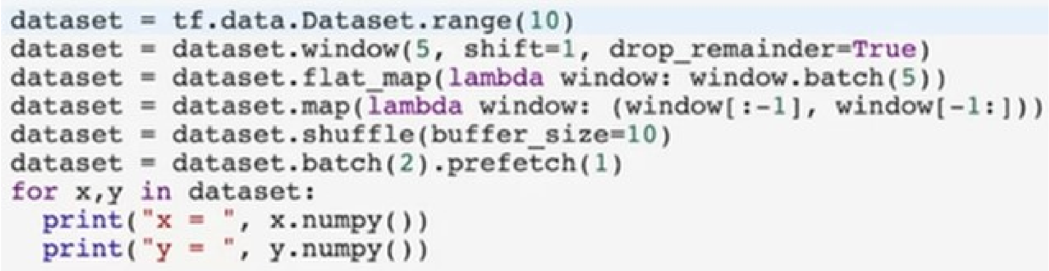
Computation is done in batches. This method is designed for performance in large scale inputs. For small amount of inputs that fit in one batch, directly using `__call__` is recommended for faster execution, e.g., `model(x)`, or `model(x, training=False)` if you have layers such as `tf.keras.layers.BatchNormalization` that behaves differently during inference. Also, note the fact that test loss is not affected by regularization layers like noise and dropout.

Arguments

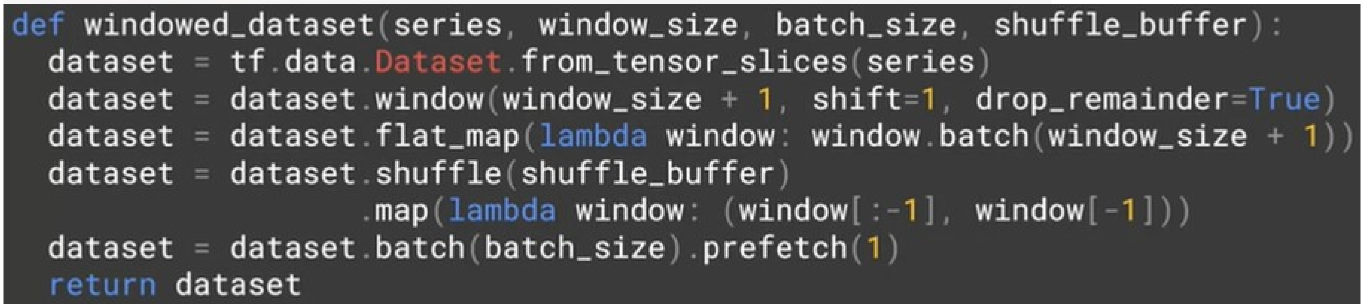
- **x**: Input samples. It could be:
  - A Numpy array (or array-like), or a list of arrays (in case the model has multiple inputs).
  - A TensorFlow tensor, or a list of tensors (in case the model has multiple inputs).
  - A `tf.data` dataset.
  - A generator or `keras.utils.Sequence` instance. A more detailed description of unpacking behavior for iterator types (Dataset, generator, Sequence) is given in the `Unpacking behavior for iterator-like inputs` section of `Model.fit`.
- **batch_size**: Integer or `None`. Number of samples per batch. If unspecified, `batch_size` will default to 32. Do not specify the `batch_size` if your data is in the form of 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.
- **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

</td>
</tr>
</table>

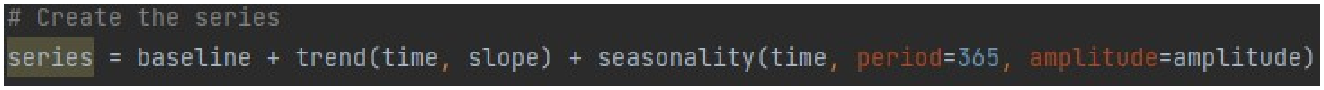
	unspecified, `workers` will default to 1. - **use_multiprocessing**: Boolean. Used for generator or `keras.utils.Sequence` input only. If `True`, use process-based threading. If unspecified, `use_multiprocessing` will default to `False`. Note that because this implementation relies on multiprocessing, you should not pass non-picklable arguments to the generator as they can't be passed easily to children processes. See the discussion of `Unpacking behavior for iterator-like inputs` for `Model.fit`. Note that Model.predict uses the same interpretation rules as `Model.fit` and `Model.evaluate`, so inputs must be unambiguous for all three methods. **Returns** Numpy array(s) of predictions. **Raises** - **RuntimeError**: If `model.predict` is wrapped in `tf.function`. - **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.
	tf.keras.Sequential | TensorFlow Core v2.7.0



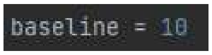
```
dataset = tf.data.Dataset.range(10)
dataset = dataset.window(5, shift=1, drop_remainder=True)
dataset = dataset.flat_map(lambda window: window.batch(5))
dataset = dataset.map(lambda window: (window[:-1], window[-1:]))
dataset = dataset.shuffle(buffer_size=10)
dataset = dataset.batch(2).prefetch(1)
for x,y in dataset:
  print("x = ", x.numpy())
  print("y = ", y.numpy())
```



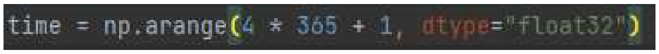
```
def windowed_dataset(series, window_size, batch_size, shuffle_buffer):
  dataset = tf.data.Dataset.from_tensor_slices(series)
  dataset = dataset.window(window_size + 1, shift=1, drop_remainder=True)
  dataset = dataset.flat_map(lambda window: window.batch(window_size + 1))
  dataset = dataset.shuffle(shuffle_buffer)
                   .map(lambda window: (window[:-1], window[-1]))
  dataset = dataset.batch(batch_size).prefetch(1)
  return dataset
```



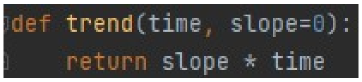
```
# Create the series
series = baseline + trend(time, slope) + seasonality(time, period=365, amplitude=amplitude)
```



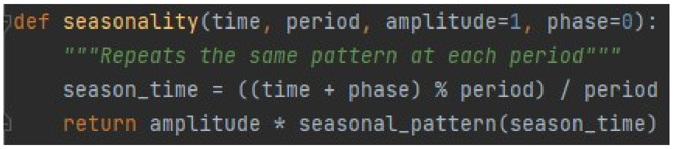
```
baseline = 10
```



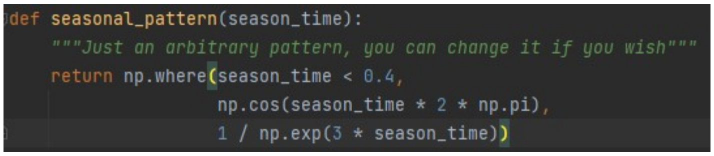
```
time = np.arange(4 * 365 + 1, dtype="float32")
```



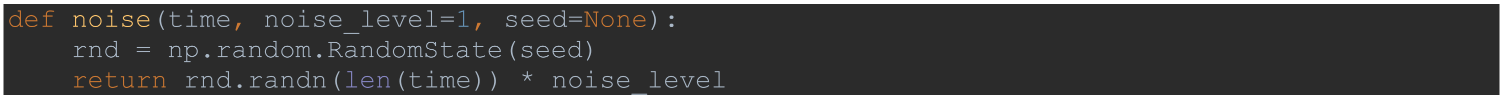
```
def trend(time, slope=0):
    return slope * time
```



```
def seasonality(time, period, amplitude=1, phase=0):
    """Repeats the same pattern at each period"""
    season_time = ((time + phase) % period) / period
    return amplitude * seasonal_pattern(season_time)
```



```
def seasonal_pattern(season_time):
    """Just an arbitrary pattern, you can change it if you wish"""
    return np.where(season_time < 0.4,
                    np.cos(season_time * 2 * np.pi),
                    1 / np.exp(3 * season_time))
```



```
def noise(time, noise_level=1, seed=None):
    rnd = np.random.RandomState(seed)
    return rnd.randn(len(time)) * noise_level
```

```
# Update with noise
series += noise(time, noise_level, seed=42)

split_time = 1000
time_train = time[:split_time]
x_train = series[:split_time]
time_valid = time[split_time:]
x_valid = series[split_time:]

window_size = 20
batch_size = 32
shuffle_buffer_size = 1000
```

Complete code for Single Layer Neural Network

```
def plot_series(time, series, format="-", start=0, end=None):
    plt.plot(time[start:end], series[start:end], format)
    plt.xlabel("Time")
    plt.ylabel("Value")
    plt.grid(True)

def trend(time, slope=0):
    return slope * time

def seasonal_pattern(season_time):
    """Just an arbitrary pattern, you can change it if you wish"""
    return np.where(season_time < 0.4,
                    np.cos(season_time * 2 * np.pi),
                    1 / np.exp(3 * season_time))

def noise(time, noise_level=1, seed=None):
    rnd = np.random.RandomState(seed)
    return rnd.randn(len(time)) * noise_level

time = np.arange(4 * 365 + 1, dtype="float32")
baseline = 10
series = trend(time, 0.1)
baseline = 10
amplitude = 40
slope = 0.05
noise_level = 5

# Create the series
series = baseline + trend(time, slope) + seasonality(time, period=365, amplitude=amplitude)
# Update with noise
series += noise(time, noise_level, seed=42)

split_time = 1000
time_train = time[:split_time]
x_train = series[:split_time]
time_valid = time[split_time:]
x_valid = series[split_time:]

window_size = 20
batch_size = 32
shuffle_buffer_size = 1000

def windowed_dataset(series, window_size, batch_size, shuffle_buffer):
  dataset = tf.data.Dataset.from_tensor_slices(series)
  dataset = dataset.window(window_size + 1, shift=1, drop_remainder=True)
  dataset = dataset.flat_map(lambda window: window.batch(window_size + 1))
  dataset = dataset.shuffle(shuffle_buffer).map(lambda window: (window[:-1], window[-1]))
  dataset = dataset.batch(batch_size).prefetch(1)
  return dataset


dataset = windowed_dataset(x_train, window_size, batch_size, shuffle_buffer_size)
print(dataset)
l0 = tf.keras.layers.Dense(1, input_shape=[window_size])
model = tf.keras.models.Sequential([l0])

model.compile(loss="mse", optimizer=tf.keras.optimizers.SGD(learning_rate=1e-6,
momentum=0.9))
model.fit(dataset,epochs=100,verbose=0)

print("Layer weights {}".format(l0.get_weights()))

forecast = []

for time in range(len(series) - window_size):
  forecast.append(model.predict(series[time:time + window_size][np.newaxis]))

forecast = forecast[split_time-window_size:]
results = np.array(forecast)[:, 0, 0]


plt.figure(figsize=(10, 6))

plot_series(time_valid, x_valid)
plot_series(time_valid, results)
plt.show()
print(tf.keras.metrics.mean_absolute_error(x_valid, results).numpy())
```