```
#@title Licensed under the Apache License, Version 2.0 (the "Licen
# you may not use this file except in compliance with the Licen
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, sof
# distributed under the License is distributed on an "AS IS" BAS
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
# See the License for the specific language governing permissions
# limitations under the License.
```

Licensed under the Apache License, Version 2.0 (the "License");


```
try:
    # %tensorflow_version only exists in Colab.
    %tensorflow_version 2.x
except Exception:
    pass
```

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
print(tf.__version__)
```

```
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 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 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")
```

```
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


tf.keras.backend.clear_session()
tf.random.set_seed(51)
np.random.seed(51)

train_set = windowed_dataset(x_train, window_size, batch_size=128, shuffle_buffer=shuffle_buf

model = tf.keras.models.Sequential([
    tf.keras.layers.Lambda(lambda x: tf.expand_dims(x, axis=-1),
                           input_shape=[None]),
    tf.keras.layers.SimpleRNN(40, return_sequences=True),
    tf.keras.layers.SimpleRNN(40),
    tf.keras.layers.Dense(1),
    tf.keras.layers.Lambda(lambda x: x * 100.0)
])

lr_schedule = tf.keras.callbacks.LearningRateScheduler(
        lambda epoch: 1e-8 * 10**(epoch / 20))
optimizer = tf.keras.optimizers.SGD(lr=1e-8, momentum=0.9)
model.compile(loss=tf.keras.losses.Huber(),
                          optimizer=optimizer,
                          metrics=["mae"])
history = model.fit(train_set, epochs=100, callbacks=[lr_schedule])
```

```
plt.semilogx(history.history["lr"],  history.history["loss"])
plt.axis([1e-8,  1e-4,  0,  30])


tf.keras.backend.clear_session()
tf.random.set_seed(51)
np.random.seed(51)

dataset  =  windowed_dataset(x_train,  window_size,  batch_size=128,  shuffle_buffer=shuffle_buffer

model  =  tf.keras.models.Sequential([
    tf.keras.layers.Lambda(lambda  x:  tf.expand_dims(x,  axis=-1),
                                       input_shape=[None]),
    tf.keras.layers.SimpleRNN(40,  return_sequences=True),
    tf.keras.layers.SimpleRNN(40),
    tf.keras.layers.Dense(1),
    tf.keras.layers.Lambda(lambda  x:  x  *  100.0)
])

optimizer  =  tf.keras.optimizers.SGD(lr=5e-5,  momentum=0.9)
model.compile(loss=tf.keras.losses.Huber(),
                          optimizer=optimizer,
                          metrics=["mae"])
history  =  model.fit(dataset,epochs=400)


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)


tf.keras.metrics.mean_absolute_error(x_valid,  results).numpy()


import  matplotlib.image    as  mpimg
import  matplotlib.pyplot  as  plt

#-----------------------------------------------------------
#  Retrieve  a  list  of  list  results  on  training  and  test  data
#  sets  for  each  training  epoch
#-----------------------------------------------------------
mae=history.history['mae']
loss=history.history['loss']

epochs=range(len(loss))  #  Get  number  of  epochs

#------------------------------------------------
```

```
# Plot MAE and Loss
#---------------------------------------------
plt.plot(epochs, mae, 'r')
plt.plot(epochs, loss, 'b')
plt.title('MAE and Loss')
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend(["MAE", "Loss"])

plt.figure()

epochs_zoom = epochs[200:]
mae_zoom = mae[200:]
loss_zoom = loss[200:]

#---------------------------------------------
# Plot Zoomed MAE and Loss
#---------------------------------------------
plt.plot(epochs_zoom, mae_zoom, 'r')
plt.plot(epochs_zoom, loss_zoom, 'b')
plt.title('MAE and Loss')
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend(["MAE", "Loss"])

plt.figure()
```