```
# Deep learning for timeseries

by Sanjeev Gupta
```

Deep learning for timeseries

by Sanjeev Gupta

A temperature-forecasting example

Problem:

Given a timeseries of hourly measurements of various atmospheric parameters, **predict** the temperature 24 hours in the future.

Goal:

To demonstrate that RNNs performm better than Dense or Conv Networks for time-series data

```
!wget https://s3.amazonaws.com/keras-datasets/jena_climate_2009_2016.csv.zip
!unzip jena_climate_2009_2016.csv.zip
```

```
--2022-06-03 16:33:17--  https://s3.amazonaws.com/keras-datasets/jena_climate_2009_2016.csv.zip
Resolving s3.amazonaws.com (s3.amazonaws.com)... 52.217.203.80
Connecting to s3.amazonaws.com (s3.amazonaws.com)|52.217.203.80|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 13565642 (13M) [application/zip]
Saving to: ‘jena_climate_2009_2016.csv.zip’

jena_climate_2009_2 100%[===================>]  12.94M  57.6MB/s    in 0.2s

2022-06-03 16:33:18 (57.6 MB/s) - ‘jena_climate_2009_2016.csv.zip’ saved [13565642/13565642]

Archive:  jena_climate_2009_2016.csv.zip
  inflating: jena_climate_2009_2016.csv
  inflating: __MACOSX/._jena_climate_2009_2016.csv
```

```
import tensorflow as tf
import os
import pandas as pd
```

```
zip_path = tf.keras.utils.get_file(
    origin='https://storage.googleapis.com/tensorflow/tf-keras-datasets/jena_climate_2009_2016.csv.zip',
    fname='jena_climate_2009_2016.csv.zip',
    extract=True)
csv_path, _ = os.path.splitext(zip_path)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/jena_climate_2009_2016.csv.zip
13574144/13568290 [==============================] - 0s 0us/step
13582336/13568290 [==============================] - 0s 0us/step
```

Inspecting the data of the Jena weather dataset

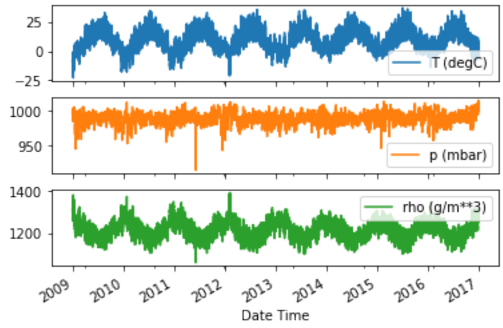
```
df = pd.read_csv(csv_path)
# Slice [start:stop:step], starting from index 5 take every 6th record.
#df = df[5::6]

date_time = pd.to_datetime(df.pop('Date Time'), format='%d.%m.%Y %H:%M:%S')
```

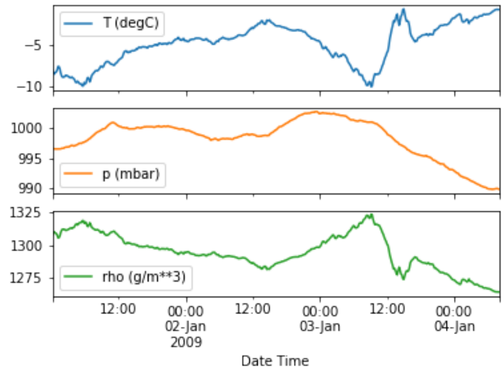
```
df.head()
```

	p (mbar)	T (degC)	Tpot (K)	Tdew (degC)	rh (%)	VPmax (mbar)	VPact (mbar)	VPdef (mbar)	sh (g/kg)	H2OC (mmol/mol)	rho (g/m**3)	wv (m/s)	max. wv (m/s)	wd (deg)
0	996.52	-8.02	265.40	-8.90	93.3	3.33	3.11	0.22	1.94	3.12	1307.75	1.03	1.75	152.3
1	996.57	-8.41	265.01	-9.28	93.4	3.23	3.02	0.21	1.89	3.03	1309.80	0.72	1.50	136.1
2	996.53	-8.51	264.91	-9.31	93.9	3.21	3.01	0.20	1.88	3.02	1310.24	0.19	0.63	171.6
3	996.51	-8.31	265.12	-9.07	94.2	3.26	3.07	0.19	1.92	3.08	1309.19	0.34	0.50	198.0
4	996.51	-8.27	265.15	-9.04	94.1	3.27	3.08	0.19	1.92	3.09	1309.00	0.32	0.63	214.3

Double-click (or enter) to edit

```
# **Visualize the data**
plot_cols = ['T (degC)', 'p (mbar)', 'rho (g/m**3)']
plot_features = df[plot_cols]
plot_features.index = date_time
_ = plot_features.plot(subplots=True)

# Lets zoom in to a smaller time-window
plot_features = df[plot_cols][:480]
plot_features.index = date_time[:480]
_ = plot_features.plot(subplots=True)
```



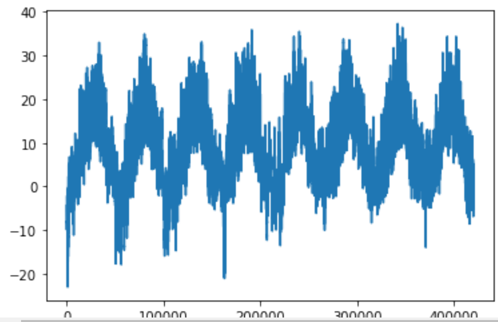


Let's just focus on the temperature and plot.

Note: Data is recorded every 10 minutes, you get 24 × 6 = 144 data points per day

```
# Plotting the temperature timeseries
from matplotlib import pyplot as plt
temperature = df['T (degC)'] # extract out the temperature data
plt.plot(range(len(temperature)), temperature)
```

```
[<matplotlib.lines.Line2D at 0x7f56af0c7650>]
```



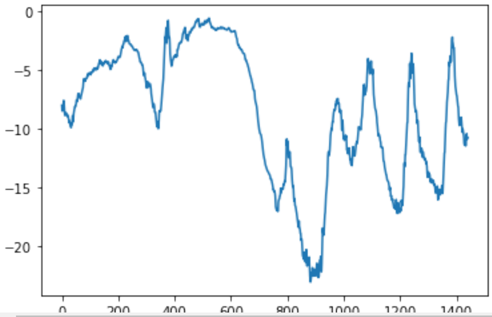
```
# Convert data to np array
import numpy as np
raw_data = np.array(df) #raw_data is a np array
print(raw_data.shape)
# Q: how many features are there? A:
```

```
(420551, 14)
```

Double-click (or enter) to edit

```
# Plotting the first 10 days of the temperature timeseries
plt.plot(range(1440), temperature[:1440])
print(temperature[:5])
```

```
0   -8.02
1   -8.41
2   -8.51
3   -8.31
4   -8.27
Name: T (degC), dtype: float64
```



```
# Compute the number of samples we'll use for each data split
num_train_samples = int(0.5 * len(raw_data)) #Q: What is len(raw_data) #A:
num_val_samples = int(0.25 * len(raw_data))
num_test_samples = len(raw_data) - num_train_samples - num_val_samples #Q: apx how many test samples? #A: apx 25%
print("num_train_samples:", num_train_samples)
print("num_val_samples:", num_val_samples)
print("num_test_samples:", num_test_samples)
```

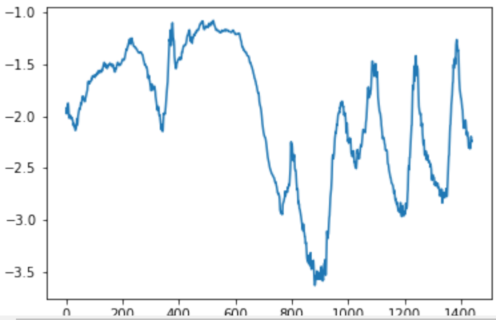
```
num_train_samples: 210275
num_val_samples: 105137
num_test_samples: 105139
```

Preparing the data

Normalizing the data

```
# Important: Let's not repeat mistakes
mean = raw_data[:num_train_samples].mean(axis=0) # Q: axis=0 means? Is mean a vector? #A:                                        tempora:
std = raw_data[:num_train_samples].std(axis=0)

raw_data -= mean
raw_data /= std

# Important Q: On what data was the mean and std computed? #A:
# Q: On what data is starndarization tranformation applied? #A:



# plot the normalized temperature data
print(raw_data[:,1])  # temperature data
plt.plot(raw_data[:1440,1])
#Q: What is the difference between the below plot and the prev one? #A: Scale. visible diff in y-axis
#Q: Does the plot look like it has 0 mean? Yes/No/Why? #A:
```

```
[-1.92080466 -1.96527448 -1.976677   ... -1.36664229 -1.48864923
 -1.55592409]
[<matplotlib.lines.Line2D at 0x7f56b22d5cd0>]
```



We have time series data from sensors/web/...

But how do we get datasets? What is the input? What is the label?

To make apt datasets, We will use the a keras utility called

```
timeseries_dataset_from_array()
```

So first, let's see how it works

(Q: Do you know of any other utility for some other kind of dataset?)

```
import numpy as np
from tensorflow import keras
int_sequence = np.arange(20)
print(f"Original Timeseries sequence: {int_sequence}\n")
print(f"data = int_sequence[:-3]={int_sequence[:-3]}\n")
print(f"targets= int_sequence[3:] = {int_sequence[3:]}\n")

# returns sequences and corresponding targets
dummy_dataset = keras.utils.timeseries_dataset_from_array(
                                      data=int_sequence[:-3],
                                      targets=int_sequence[3:],
                                      # The next 3 arguments are for you to manipulate the data (input)
                                      sequence_length=4,
                                      sampling_rate = 3,
                                      sequence_stride = 2, # stride applied to both data and targets
                                      # batch_size=2,  # see in next cell
                                  )


for inputs, targets in dummy_dataset:
    # print(inputs, targets)
    print(inputs.shape)
    for i in range(inputs.shape[0]): # range(batch_size)
        print([int(x) for x in inputs[i]], int(targets[i]))
```

```
Original Timeseries sequence: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]

data = int_sequence[:-3]=[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16]

targets= int_sequence[3:] = [ 3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]

(3, 4)
[0, 3, 6, 9] 3
[2, 5, 8, 11] 5
[4, 7, 10, 13] 7
```

Q: We want the following:

[0, 1, 2] : 3

[1, 2, 3] : 4

[2, 3, 4] : 5

...

...

Start coding or generate with AI.

Q: We want the following:

[0, 2, 4] : 3

[1, 3, 5] : 4

[2, 4, 6] : 5

...

...

Q: We want the following:

[0, 3, 6, 9] : 3

[2, 5, 8, 11] : 5

[4, 7, 10, 12] : 7

...

...

Q: What if the target list is too small?

```
int_sequence = np.arange(20)
print(f"Original Timeseries sequence: {int_sequence}")
print(f"data = int_sequence[:-3]={int_sequence[:-3]}")
print(f"targets= int_sequence[3:] = {int_sequence[3:]}")

dummy_dataset = keras.utils.timeseries_dataset_from_array(
                                      data=int_sequence[:-3],       #Q: What is the effect of the 3 #A: maintains delay of 3 between las
                                      targets=int_sequence[3:],
                                      # The next 3 arguments are for you to manipulate the data (input)
                                      sequence_length=5,
                                      # sampling_rate = 2,
                                      # sequence_stride = 3,
                                      # batch_size=2,
                                      # shuffle = True
                                  )


for inputs, targets in dummy_dataset:
    print(inputs)
    print(targets)
    print(f"\n inputs.shape = {inputs.shape }")
    for i in range(inputs.shape[0]):
        print([int(x) for x in inputs[i]], int(targets[i]))
```

```
Original Timeseries sequence: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
data = int_sequence[:-3]=[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16]
targets= int_sequence[3:] = [ 3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
tf.Tensor(
[[ 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 10]
 [ 7  8  9 10 11]
 [ 8  9 10 11 12]
 [ 9 10 11 12 13]
 [10 11 12 13 14]
 [11 12 13 14 15]
 [12 13 14 15 16]], shape=(13, 5), dtype=int64)
tf.Tensor([ 3  4  5  6  7  8  9 10 11 12 13 14 15], shape=(13,), dtype=int64)

 inputs.shape = (13, 5)
[0, 1, 2, 3, 4] 3
[1, 2, 3, 4, 5] 4
[2, 3, 4, 5, 6] 5
[3, 4, 5, 6, 7] 6
[4, 5, 6, 7, 8] 7
[5, 6, 7, 8, 9] 8
[6, 7, 8, 9, 10] 9
[7, 8, 9, 10, 11] 10
[8, 9, 10, 11, 12] 11
[9, 10, 11, 12, 13] 12
[10, 11, 12, 13, 14] 13
[11, 12, 13, 14, 15] 14
[12, 13, 14, 15, 16] 15
```

Now that we understand how the utility works, let's use it to make our dataset

Remember: Data is recorded every 10 minutes, you get 24 × 6 = 144 data points per day

```
# Instantiating datasets for training, validation, and testing

# Observations will be sampled at one data point per hour: we will only keep one data point out of 6.
sampling_rate = 6  # every 6th sample of the original sequence is a sample after an hour


sequence_length = 120 #Q: How many days worth data in a sequence? #A: x Days

# The target for a sequence will be the temperature 24 hours after the end of the sequenc
delay = sampling_rate * (sequence_length + 24 - 1)

batch_size = 256
```

```
train_dataset = keras.utils.timeseries_dataset_from_array(
    raw_data[:-delay],          # Q: What is this argument? #A: data
    targets=temperature[delay:],
    sampling_rate=sampling_rate,
    sequence_length=sequence_length,
    shuffle=True,
    batch_size=batch_size,
    start_index=0,
    end_index=num_train_samples)

val_dataset = keras.utils.timeseries_dataset_from_array(
    raw_data[:-delay],
    targets=temperature[delay:],
    sampling_rate=sampling_rate,
    sequence_length=sequence_length,
    shuffle=True,
    batch_size=batch_size,
    start_index=num_train_samples,
    end_index=num_train_samples + num_val_samples)

test_dataset = keras.utils.timeseries_dataset_from_array(
    raw_data[:-delay],
    targets=temperature[delay:],
    sampling_rate=sampling_rate,
    sequence_length=sequence_length,
    shuffle=True,
    batch_size=batch_size,
    start_index=num_train_samples + num_val_samples)
```

Q: Do we have to normalise the test data or is it already done?

```
# Inspecting the output of one of our datasets
for samples, targets in train_dataset:
    print("samples shape:", samples.shape)
    print("targets shape:", targets.shape)
    break
```

```
samples shape: (256, 120, 14)
targets shape: (256,)
```

A common-sense, non-machine-learning baseline

Computing the common-sense baseline MAE

Prediction based on the assumption that the temperature changes periodically.

Hence, the temperature prediction of 24 hrs later should be equal to the temperature right now

```
def evaluate_naive_method(dataset):
    total_abs_err = 0.
    samples_seen = 0
    for samples, targets in dataset:
        #Q: What is the shape of samples? What is idx 1? # A: (256, 120, 14), ; idx 1 means temperature. See dataframe.
        preds = samples[:, -1, 1] * std[1] + mean[1]  #Q: Why mult by std and add with mean?
        #Q: What are the next 2 lines doing?
        total_abs_err += np.sum(np.abs(preds - targets))
        samples_seen += samples.shape[0]
    return total_abs_err / samples_seen

print(f"Validation MAE: {evaluate_naive_method(val_dataset):.2f}")
print(f"Test MAE: {evaluate_naive_method(test_dataset):.2f}")
```

```
Validation MAE: 2.44
Test MAE: 2.62
```

The Test MAE is 2.62. It means our naive model is expected to give a prediction which is off by 2.62 deg on average.

Let's try a dense NN

We want to see how a densly connected network would perform on the problem.

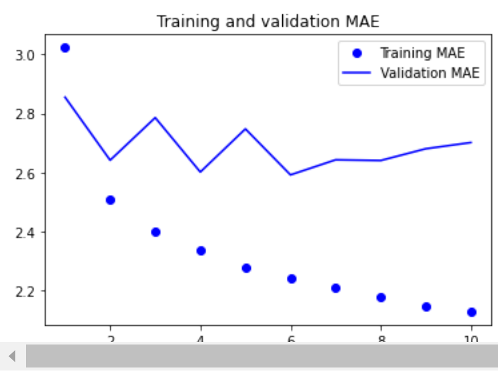
```
# **Training and evaluating a densely connected model**
from tensorflow import keras
from tensorflow.keras import layers
```

```
from tensorflow.keras.utils import plot_model

# define input layer: relate to how you would pass image data to a dense layer
# Remember: Shape of an input batch: (256, 120, 14)
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1])) #Q: What is raw_data.shape[-1]? #A: Last index, i.e., no. of features

x = layers.Flatten()(inputs) #Q: Why flatten? # A: Input is in the form of matrix
x = layers.Dense(16, activation="relu")(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_dense.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)

model = keras.models.load_model("jena_dense.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

```
Epoch 1/10
819/819 [==============================] - 43s 49ms/step - loss: 15.3177 - mae: 3.0231 - val_loss: 12.9447 - val_mae: 2.8549
Epoch 2/10
819/819 [==============================] - 40s 49ms/step - loss: 10.1642 - mae: 2.5066 - val_loss: 10.9849 - val_mae: 2.6413
Epoch 3/10
819/819 [==============================] - 41s 50ms/step - loss: 9.3269 - mae: 2.4004 - val_loss: 12.2271 - val_mae: 2.7855
Epoch 4/10
819/819 [==============================] - 41s 49ms/step - loss: 8.8321 - mae: 2.3372 - val_loss: 10.7274 - val_mae: 2.6012
Epoch 5/10
819/819 [==============================] - 40s 48ms/step - loss: 8.3972 - mae: 2.2775 - val_loss: 11.8348 - val_mae: 2.7474
Epoch 6/10
819/819 [==============================] - 40s 49ms/step - loss: 8.1313 - mae: 2.2416 - val_loss: 10.6627 - val_mae: 2.5916
Epoch 7/10
819/819 [==============================] - 41s 50ms/step - loss: 7.8942 - mae: 2.2082 - val_loss: 11.0262 - val_mae: 2.6427
Epoch 8/10
819/819 [==============================] - 40s 49ms/step - loss: 7.6486 - mae: 2.1763 - val_loss: 11.0131 - val_mae: 2.6403
Epoch 9/10
819/819 [==============================] - 40s 49ms/step - loss: 7.4516 - mae: 2.1465 - val_loss: 11.2592 - val_mae: 2.6802
Epoch 10/10
819/819 [==============================] - 40s 49ms/step - loss: 7.3133 - mae: 2.1287 - val_loss: 11.5753 - val_mae: 2.7012
405/405 [==============================] - 14s 33ms/step - loss: 2765.6377 - mae: 6.2533
Test MAE: 6.25
```

Plotting results

```
import matplotlib.pyplot as plt
loss = history.history["mae"]
val_loss = history.history["val_mae"]
epochs = range(1, len(loss) + 1)
plt.figure()
plt.plot(epochs, loss, "bo", label="Training MAE")
plt.plot(epochs, val_loss, "b", label="Validation MAE")
plt.title("Training and validation MAE")
plt.legend()
plt.show()
```



The dense network has not done very well. Achieves a val loss of 2.6.

Why?

- The hypothesis space is inapproptiate
    - perhaps too large.... needle in a haystack

- looking for a window-wise global pattern
- An example of the poweful nature of
  - good feature engineering
  - domain knowledge

Let's try a 1D convolutional model

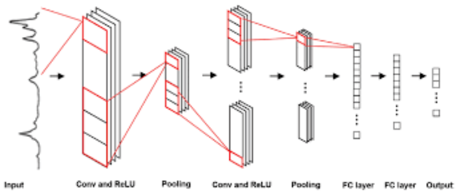
Now lets try a CNN.

But we don't have an image?

- We don't necessarily need one

We will use 1D convolutions.

- try to learn local patterns in the signal/sequence



Nice gif

```
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.utils import plot_model
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))

x = layers.Conv1D(8, 24, activation="relu", input_shape=(120,14))(inputs)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 12, activation="relu")(x)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 6, activation="relu")(x)
x = layers.GlobalAveragePooling1D()(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
callbacks = [
    keras.callbacks.ModelCheckpoint("jena_conv.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])

plot_model(model, show_shapes=True)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-2-64d4cf0dc158> in <module>
      2 from tensorflow.keras import layers
      3 from tensorflow.keras.utils import plot_model
----> 4 inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
      5 
      6 x = layers.Conv1D(8, 24, activation="relu", input_shape=(120,14))(inputs)

NameError: name 'sequence_length' is not defined
```

```
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)

model = keras.models.load_model("jena_conv.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

```
Epoch 1/10
819/819 [==============================] - 43s 51ms/step - loss: 21.6386 - mae: 3.6381 - val_loss: 16.0439 - val_mae: 3.1527
Epoch 2/10
819/819 [==============================] - 42s 51ms/step - loss: 15.2469 - mae: 3.1052 - val_loss: 14.3108 - val_mae: 2.9796
Epoch 3/10
819/819 [==============================] - 42s 51ms/step - loss: 13.8960 - mae: 2.9554 - val_loss: 13.9624 - val_mae: 2.9398
Epoch 4/10
```

```
819/819 [==============================] - 43s 52ms/step - loss: 12.9435 - mae: 2.8534 - val_loss: 14.6220 - val_mae: 2.9818
Epoch 5/10
819/819 [==============================] - 43s 52ms/step - loss: 12.2739 - mae: 2.7790 - val_loss: 13.5374 - val_mae: 2.8815
Epoch 6/10
819/819 [==============================] - 43s 52ms/step - loss: 11.7057 - mae: 2.7121 - val_loss: 13.9518 - val_mae: 2.9199
Epoch 7/10
819/819 [==============================] - 43s 52ms/step - loss: 11.1627 - mae: 2.6484 - val_loss: 13.2862 - val_mae: 2.8556
Epoch 8/10
819/819 [==============================] - 42s 51ms/step - loss: 10.7152 - mae: 2.5972 - val_loss: 14.8524 - val_mae: 3.0112
Epoch 9/10
819/819 [==============================] - 42s 51ms/step - loss: 10.4060 - mae: 2.5580 - val_loss: 14.5681 - val_mae: 2.9907
Epoch 10/10
819/819 [==============================] - 42s 51ms/step - loss: 10.1176 - mae: 2.5244 - val_loss: 15.4485 - val_mae: 3.0636
405/405 [==============================] - 14s 33ms/step - loss: 21641.4277 - mae: 12.9118
Test MAE: 12.91
```

A first recurrent baseline

Built-in RNN layers: a simple example

There are three built-in RNN layers in Keras:

- **keras.layers.SimpleRNN**, a fully-connected RNN where the output from previous timestep is to be fed to next timestep.
- **keras.layers.GRU**, first proposed in Cho et al., 2014.
- **keras.layers.LSTMs.**, first proposed in Hochreiter & Schmidhuber, 1997.

Let's see how an RNN performs.

We will have a detailed look at the architecture shortly.

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.LSTM(16)(inputs)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_lstm.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)

model = keras.models.load_model("jena_lstm.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

```
Epoch 1/10
819/819 [==============================] - 127s 149ms/step - loss: 36.1634 - mae: 4.3527 - val_loss: 11.3400 - val_mae: 2.5657
Epoch 2/10
819/819 [==============================] - 122s 148ms/step - loss: 10.5254 - mae: 2.5235 - val_loss: 9.5128 - val_mae: 2.4038
Epoch 3/10
819/819 [==============================] - 122s 149ms/step - loss: 9.7208 - mae: 2.4238 - val_loss: 9.5953 - val_mae: 2.4128
Epoch 4/10
819/819 [==============================] - 122s 149ms/step - loss: 9.3853 - mae: 2.3779 - val_loss: 9.7360 - val_mae: 2.4320
Epoch 5/10
819/819 [==============================] - 123s 149ms/step - loss: 9.1134 - mae: 2.3424 - val_loss: 9.5317 - val_mae: 2.4168
Epoch 6/10
819/819 [==============================] - 122s 148ms/step - loss: 8.9050 - mae: 2.3153 - val_loss: 9.5940 - val_mae: 2.4203
Epoch 7/10
819/819 [==============================] - 121s 147ms/step - loss: 8.7368 - mae: 2.2925 - val_loss: 9.8174 - val_mae: 2.4390
Epoch 8/10
819/819 [==============================] - 120s 146ms/step - loss: 8.6120 - mae: 2.2743 - val_loss: 9.7383 - val_mae: 2.4391
Epoch 9/10
819/819 [==============================] - 122s 149ms/step - loss: 8.4287 - mae: 2.2489 - val_loss: 9.8159 - val_mae: 2.4562
```

The thest MAE is 2.57

- Slightly better than it's Dense counterpart!
- Easy to use. Just a line of code.
- We can do much better. But we will not spend much time on training RNN based architecture. Instead we will invest that time on Transformers.

Understanding recurrent neural networks

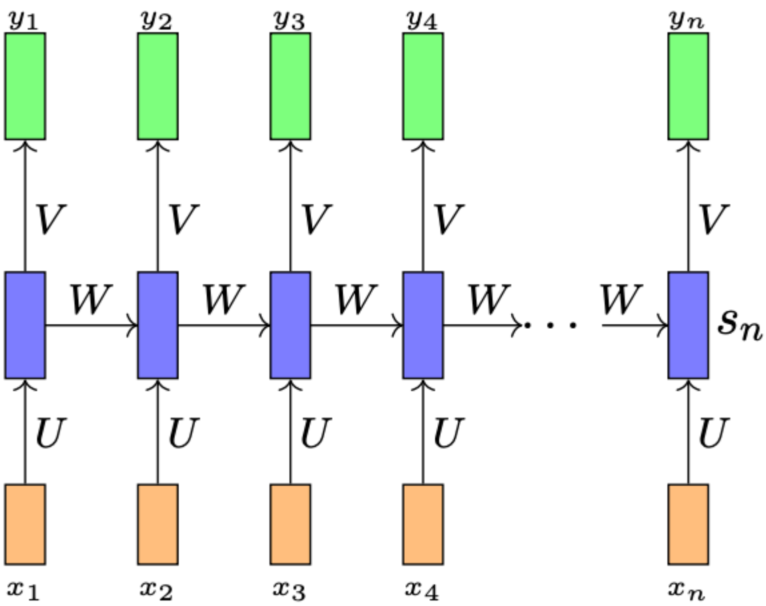
Sequence Learning problems:

- Inputs are no longer independent
- Input sizes not fixed. Ex- Length of sentences

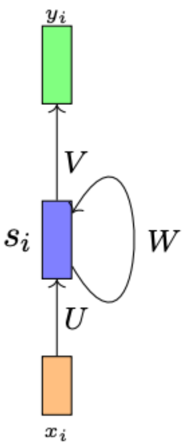
So an RNN should take care of:

- Dependence between inputs
- Variable no. of inputs
- Function executed at each time must be the same

NumPy implementation of a simple RNN



$$s_i = \sigma(Ux_i + Ws_{i-1} + b)$$
$$y_i = \mathcal{O}(Vs_i + c)$$
$$or$$
$$y_i = f(x_i, s_{i-1}, W, U, V, b, c)$$




```
import numpy as np
timesteps = 100   #:number of timesteps in the input sequence
input_features = 32
```

```
output_features = 64
inputs = np.random.random((timesteps, input_features))
state_t = np.zeros((output_features,))
U = np.random.random((output_features, input_features))
W = np.random.random((output_features, output_features))
b = np.random.random((output_features,))
successive_outputs = []
for input_t in inputs:
    output_t = np.tanh(np.dot(U, input_t) + np.dot(W, state_t) + b)
    successive_outputs.append(output_t)
    state_t = output_t
final_output_sequence = np.stack(successive_outputs, axis=0) # Q: seq/vec-to-seq/vec #A: seq-to-seq due to stack
```

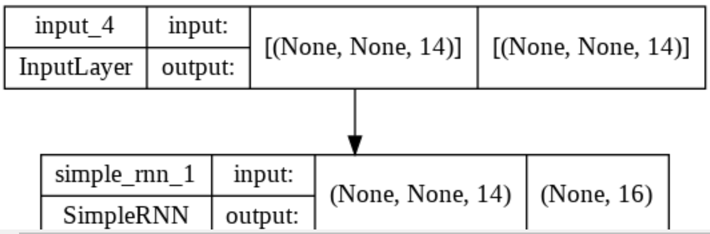
Notice the sequential nature of computation. We will discuss this later.

A recurrent layer in Keras

An RNN layer that can process sequences of any length

```
num_features = 14
inputs = keras.Input(shape=(None, num_features))
outputs = layers.SimpleRNN(16)(inputs)
model = keras.Model(inputs, outputs)
print(outputs.shape)
model.summary()
plot_model(model, show_shapes=True)
```

```
(None, 16)
Model: "model_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_4 (InputLayer)        [(None, None, 14)]        0

 simple_rnn_1 (SimpleRNN)    (None, 16)                496

=================================================================
Total params: 496
Trainable params: 496
Non-trainable params: 0
_________________________________________________________________
```

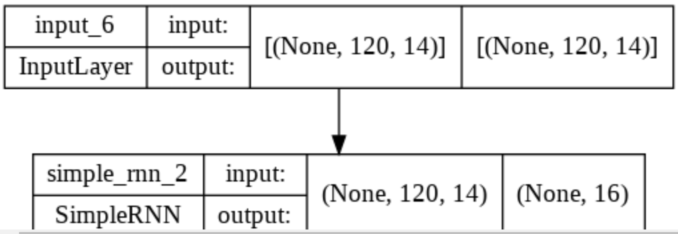



```
16*14 + 16*16 + 16
```

An RNN layer that returns only its last output step

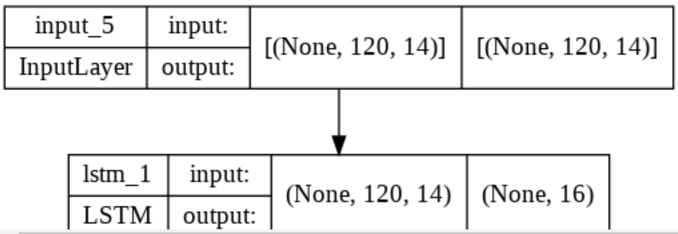
```
num_features = 14
steps = 120
inputs = keras.Input(shape=(steps, num_features))
outputs = layers.SimpleRNN(16, return_sequences=False)(inputs) # return_seq = False
model = keras.Model(inputs, outputs)
print(outputs.shape) # see output
model.summary()
plot_model(model, show_shapes=True)
```

```
(None, 16)
Model: "model_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_6 (InputLayer)        [(None, 120, 14)]         0

 simple_rnn_2 (SimpleRNN)    (None, 16)                496

=================================================================
Total params: 496
Trainable params: 496
Non-trainable params: 0
_________________________________________________________________
```




```
num_features = 14
steps = 120
inputs = keras.Input(shape=(steps, num_features))
outputs = layers.LSTM(16, return_sequences=False)(inputs) # return_seq = False
model = keras.Model(inputs, outputs)
print(outputs.shape) # see output
model.summary()
plot_model(model, show_shapes=True)
```

```
(None, 16)
Model: "model_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_5 (InputLayer)        [(None, 120, 14)]         0

 lstm_1 (LSTM)               (None, 16)                1984

=================================================================
Total params: 1,984
Trainable params: 1,984
Non-trainable params: 0
_________________________________________________________________
```



An RNN layer that returns its full output sequence

```
num_features = 14
steps = 120
inputs = keras.Input(shape=(steps, num_features))
outputs = layers.SimpleRNN(16, return_sequences=True)(inputs)
print(outputs.shape)
model = keras.Model(inputs, outputs)
model.summary()
plot_model(model, show_shapes=True)
```

```
(None, 120, 16)
```

Stacking RNN layers

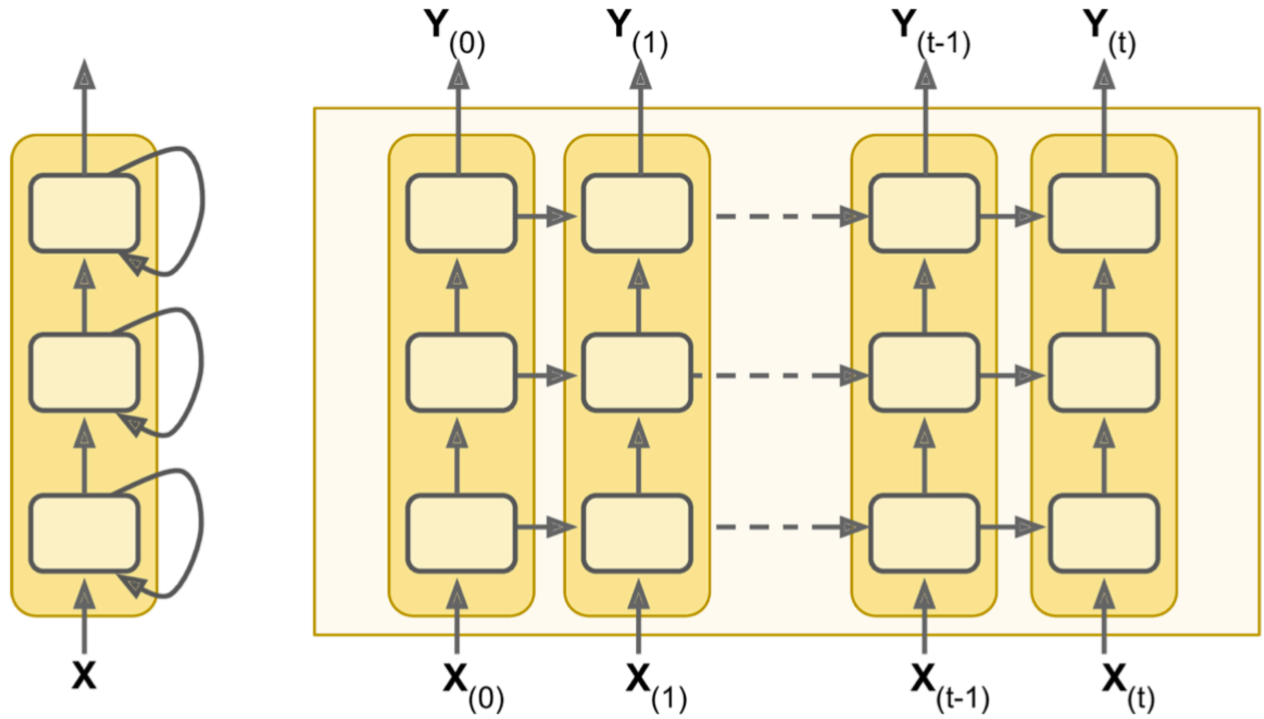


Figure 15-7. Deep RNN (left) unrolled through time (right)

```
inputs = keras.Input(shape=(steps, num_features))
x = layers.SimpleRNN(16, return_sequences=True)(inputs) # Q: can return_seq be False here? # A: NO. It will throw error.
x = layers.SimpleRNN(16, return_sequences=True)(x)
outputs = layers.SimpleRNN(16)(x)
model = keras.Model(inputs, outputs)
print(outputs.shape)
model.summary()
```

```
(None, 16)
Model: "model_9"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_11 (InputLayer)       [(None, 120, 14)]         0

 simple_rnn_5 (SimpleRNN)    (None, 120, 16)           496

 simple_rnn_6 (SimpleRNN)    (None, 120, 16)           528

 simple_rnn_7 (SimpleRNN)    (None, 16)                528

=================================================================
Total params: 1,552
Trainable params: 1,552
Non-trainable params: 0
_________________________________________________________________
```

```
16*16 + 16*16 + 16
```

Advanced use of recurrent neural networks

- Recurrent dropout
- Stacking recurrent layers
- Bi-directional RNNs

Using recurrent dropout to fight overfitting

Key Idea: The dropout pattern should remain the same across time.

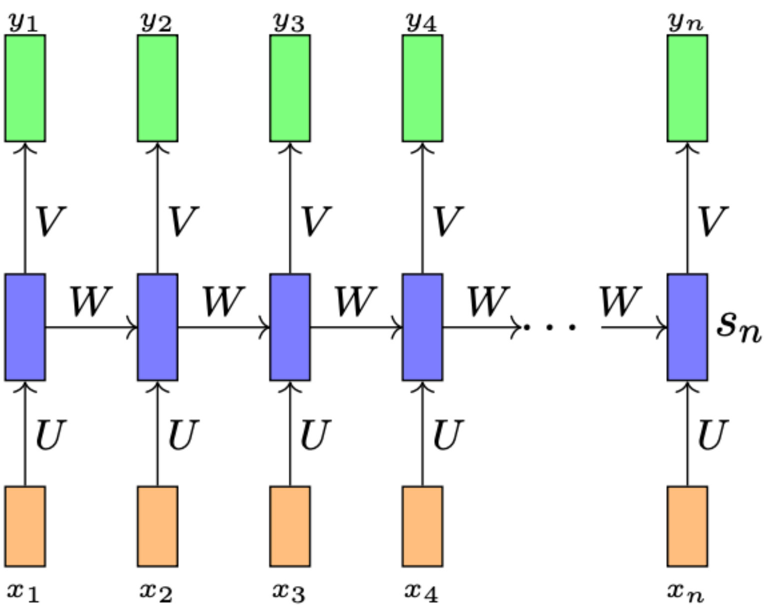
```
# Training and evaluating a dropout-regularized LSTM
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.LSTM(32, recurrent_dropout=0.25)(inputs)
```

```
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_lstm_dropout.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=50,
                    validation_data=val_dataset,
                    callbacks=callbacks)
```

```
WARNING:tensorflow:Layer lstm_1 will not use cuDNN kernels since it doesn't meet the criteria. It will use a generic GPU kernel as i
Epoch 1/50
819/819 [==============================] - 334s 405ms/step - loss: 27.6691 - mae: 3.8794 - val_loss: 9.8712 - val_mae: 2.4367
Epoch 2/50
819/819 [==============================] - 333s 406ms/step - loss: 14.7891 - mae: 2.9858 - val_loss: 9.2836 - val_mae: 2.3735
Epoch 3/50
819/819 [==============================] - 334s 408ms/step - loss: 14.0565 - mae: 2.9057 - val_loss: 9.0759 - val_mae: 2.3511
Epoch 4/50
819/819 [==============================] - 335s 409ms/step - loss: 13.3965 - mae: 2.8380 - val_loss: 8.9423 - val_mae: 2.3318
Epoch 5/50
819/819 [==============================] - 335s 408ms/step - loss: 12.9484 - mae: 2.7880 - val_loss: 8.9122 - val_mae: 2.3240
Epoch 6/50
819/819 [==============================] - 337s 411ms/step - loss: 12.4860 - mae: 2.7364 - val_loss: 9.2643 - val_mae: 2.3582
Epoch 7/50
819/819 [==============================] - 334s 408ms/step - loss: 12.1242 - mae: 2.7004 - val_loss: 9.3036 - val_mae: 2.3678
Epoch 8/50
819/819 [==============================] - 334s 407ms/step - loss: 11.7401 - mae: 2.6545 - val_loss: 9.2084 - val_mae: 2.3540
Epoch 9/50
819/819 [==============================] - 335s 409ms/step - loss: 11.5305 - mae: 2.6305 - val_loss: 9.2848 - val_mae: 2.3767
Epoch 10/50
819/819 [==============================] - 337s 411ms/step - loss: 11.3035 - mae: 2.6045 - val_loss: 9.4993 - val_mae: 2.3928
Epoch 11/50
819/819 [==============================] - 336s 410ms/step - loss: 11.1175 - mae: 2.5820 - val_loss: 9.3366 - val_mae: 2.3672
Epoch 12/50
819/819 [==============================] - 333s 406ms/step - loss: 10.8707 - mae: 2.5565 - val_loss: 9.4777 - val_mae: 2.3874
Epoch 13/50
819/819 [==============================] - 333s 407ms/step - loss: 10.7547 - mae: 2.5408 - val_loss: 9.9690 - val_mae: 2.4502
Epoch 14/50
819/819 [==============================] - 334s 407ms/step - loss: 10.6095 - mae: 2.5264 - val_loss: 9.8857 - val_mae: 2.4407
Epoch 15/50
819/819 [==============================] - 335s 409ms/step - loss: 10.5469 - mae: 2.5158 - val_loss: 9.9061 - val_mae: 2.4366
Epoch 16/50
819/819 [==============================] - 337s 412ms/step - loss: 10.3363 - mae: 2.4920 - val_loss: 9.9430 - val_mae: 2.4488
Epoch 17/50
483/819 [================>.............] - ETA: 2:10 - loss: 10.2551 - mae: 2.4851
```

How is the loss defined?

Especially for a seq-to-seq RNN model?

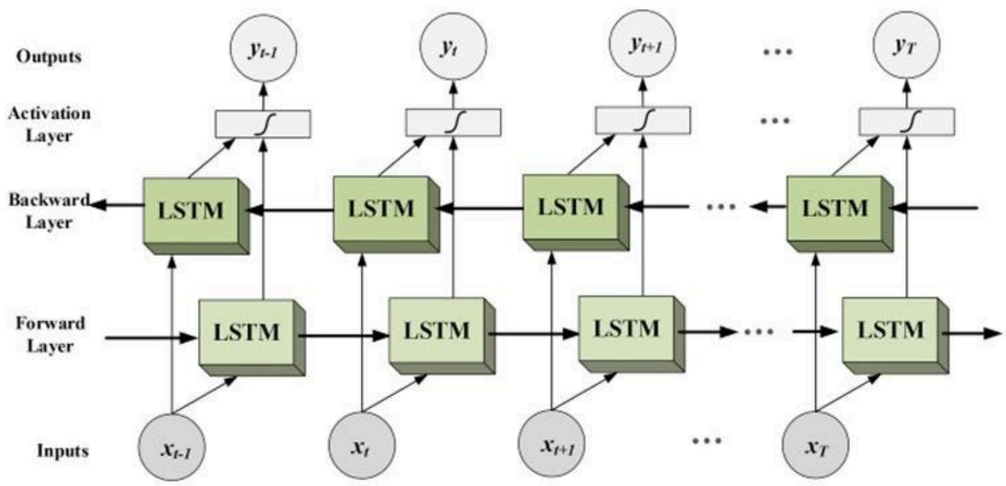


Stacking recurrent layers

We've already seen this. But repeating an example to show GRU.

```
# Training and evaluating a dropout-regularized, stacked GRU model
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.GRU(32, recurrent_dropout=0.5, return_sequences=True)(inputs) #Q: why ret_seq =True? #A: Stacked GRUs
x = layers.GRU(32, recurrent_dropout=0.5)(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_stacked_gru_dropout.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
# history = model.fit(train_dataset,
#                     epochs=50,
#                     validation_data=val_dataset,
#                     callbacks=callbacks)
# model = keras.models.load_model("jena_stacked_gru_dropout.keras")
# print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

Using bidirectional RNNs



Key Idea: Learn temporal patterns in both directions

Found to work well with text data.

```
# Training and evaluating a bidirectional LSTM
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Bidirectional(layers.LSTM(16))(inputs)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
# history = model.fit(train_dataset,
#                     epochs=10,
#                     validation_data=val_dataset)
model = keras.Model(inputs, outputs)
print(outputs.shape)
model.summary()
```

```
(None, 1)
Model: "model_10"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_9 (InputLayer)        [(None, 120, 14)]         0

 bidirectional_1 (Bidirectio  (None, 32)               3968
 nal)

 dense_4 (Dense)             (None, 1)                 33
```

```
=================================================================
Total params: 4,001
Trainable params: 4,001
Non-trainable params: 0
_________________________________________________________________
```

Let's compare it with a single LSTM layer

```
# Training and evaluating a bidirectional LSTM
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
# x = layers.Bidirectional(layers.LSTM(16))(inputs)
x = layers.LSTM(16)(inputs)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
# history = model.fit(train_dataset,
#                     epochs=10,
#                     validation_data=val_dataset)
model = keras.Model(inputs, outputs)
print(outputs.shape)
model.summary()
```

```
(None, 1)
Model: "model_12"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_10 (InputLayer)       [(None, 120, 14)]         0

 lstm_5 (LSTM)               (None, 16)                1984

 dense_5 (Dense)             (None, 1)                 17

=================================================================
Total params: 2,001
Trainable params: 2,001
Non-trainable params: 0
_________________________________________________________________
```

Note on runtime performance of RNNs:

- Small RNNs run faster on CPU
- Large RNNs can benifit from GPU
- GPU support not offered by cuDNN when using functionalities that are not optimized. E.g. Recurrent Dropout.
- Alternative: Unroll = True
- Caveat: Sequence length must be known apriori

```
inputs = keras.Input(shape=(sequence_length, num_features))
x = layers.LSTM(32, recurrent_dropout=0.2, unroll=True)(inputs)
```

Assignment 4

Given a sequence of atmospheric measurements, predict a forecast of the **Density** after **2** days.

Solve the sequence learning problem based on the following instructions:

0. Import necessary modules and set random seed to 42 using tf.random.set_seed(42).
1. Download the data and convert to dataframe as shown in the tutorial. Use the following code to download: (2)

```
!wget https://s3.amazonaws.com/keras-datasets/jena_climate_2009_2016.csv.zip
!unzip jena_climate_2009_2016.csv.zip
```

2. Identify and report the column index for the feature 'Density'. Plot the Density to visualize patterns
    - across multiple days
    - across multiple years.(3)
3. Find the numnber of datapoints for the train-val-test split of the sequence data based on the following percentages (2)
    - Training set: 55%
    - Validation set: 25%

- Test set : 20%

4. Normalize the raw_data (to mean=0, std=1). Report the mean and std of the training and test sets of the feature **'Density'**, pre and post normalization, i.e. compute the following. (5)
    - Training set:
        - pre normalization mean =
        - pre normalization std =
    - Test set:
        - post normalization mean =
        - post normalization std = Comment on observed results
5. Use keras.utils.timeseries_dataset_from_array to make appropriate datasets from the numbers computed in Q3. (7)
    - Use sampling rate = 6,
    - sequence_length must contain **6** days of temporal data (Remember: In the raw_data provided, data was recorded every 10 minutes),
    - Modify delay to predict the density **2** days from current time
    - use batch_size = 256
    - Shuffle = True
    - Info: Use appropriate start and end indices in arguments.
6. Modify the *evaluate_naive_method(dataset)* function to return MAE of the test and validation datasets. (Remember, this our common sense baseline, which assumes that the density changes are purely periodic with a period of 24 hours.) (2)
7. Build the [following model (click on this link)](). Comment on the activation argument in LSTM layer. (3)
8. Compile it using (1)
    - optimizer= rmsprop with learning rate 0.1
    - loss="mse",
    - metrics=["mae"]
9. Fit the model for 5 epochs. Plot and report Validation and Test MAE. Compare your results with the common sense baseline method in Q6