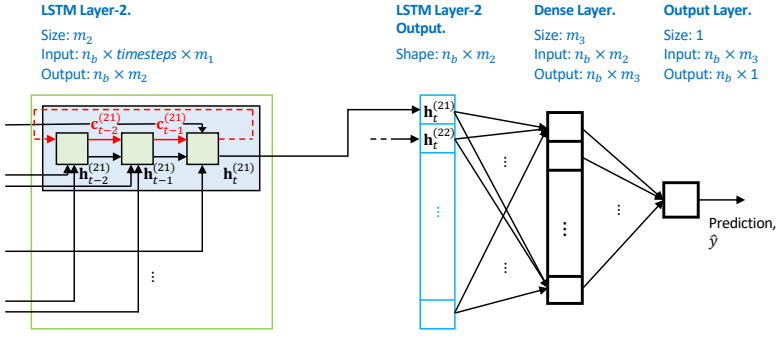
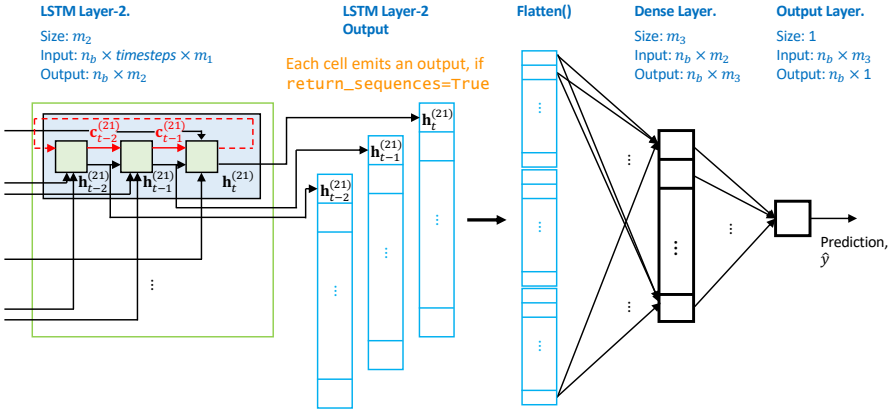


(b1) *Restricted LSTM network. In a restricted network, the last LSTM layer emits only the final hidden output. As shown above, the second LSTM layer returns only the last $\boldsymbol{h}_t$'s from each cell which makes up the feature map vector for input to the output dense layer (`(LSTM(..., return_sequences=False)`).*



(b2) *Unrestricted LSTM network. In an unrestricted network, the hidden outputs at each time-step, i.e., a sequence of outputs $\{\boldsymbol{h}_{t-\tau}, \ldots, \boldsymbol{h}_t\}$, are returned (`(LSTM(..., return_sequences=False)`). This makes up a 2D feature map of shape, (layer size, time-steps).*

Figure 5.7. *LSTM Network Structure. Part II.*

therein were explained. In this section, the view is zoomed out of the cell and the network operations at the layer level are explained.

Figure 5.7 here brings an LSTM layer's internal and external connections into perspective. It provides visibility on the layer's input-output mechanism.

The illustrations provide an understanding of stacking layers around an LSTM layer, and the way they interact. Their references to TensorFlow modeling is also given in this section.

Earlier, Figure 5.1 showed an abstraction of LSTM network. The network layers are expanded in Figure 5.7a-5.7b2. The expanded view is split into two parts for clarity. Figure 5.7a shows the left part of the network—from input to the middle. And Figure 5.7b1-5.7b2 show its continuation till the output in two major LSTM modes, viz., `return_sequences` is `True` versus `False`.

In the figures, the blue-shaded boxes in the LSTM layers are the layer's cells. And, as before, the green boxes within a cell is representative of a time-step iteration for input processing.

5.3.1 Input Processing

As also mentioned in § 5.2.1, LSTM takes a time-window of observations as an input sample. A τ sized time-window of p-dimensional observations $\boldsymbol{x}$ denoted as $\boldsymbol{x}_{(T-\tau):T}$ is a two-dimensional array. A batch of such samples is, thus, a three-dimensional array of shape: (n_batch, $timesteps$, $n\_features$).

The first LSTM layer in Figure 5.7a takes in samples from the input batch. A sample is shown as a two-dimensional array with features and time-steps along the rows and columns, respectively.

Each of the time-steps is processed sequentially. For illustration, this is shown by connecting each input time-step with the corresponding time-step iteration in the cell.

Similarly, every cell in the layer takes all the time-step inputs. The cells transmit the cell states and hidden states within themselves to perform their internal operations (described in four steps in § 5.2.4).

The interesting part is the way the states are transmitted outside. There are two major transmission modes to understand in LSTM: **stateful** and **return sequences**. These modes allow building a stateful or stateless LSTM, and/or (not) return sequences. They are described next.

5.3.2 Stateless versus Stateful

Stateless

An LSTM layer is built to learn temporal patterns. A default LSTM layer in TensorFlow learns these patterns in a time-window of observations $\boldsymbol{x}_{(T-\tau)} : T$ presented to it. This default setting is called a *stateless* LSTM and is enforced with `LSTM(..., stateful=False)`.

It is called *stateless* because the cell states are transmitted and contained within the time-window $(T - \tau) : T$. This means a long-term pattern will be only up to τ long.

Also, in this setting, the model processes each time-window independently. That is, there is no interaction or learning between two time-windows. Consequently, the default input sample shuffle during model training is allowed.

Stateful

A stateless model constrains LSTM to learn patterns only within a fixed time-window. It does not provide visibility beyond the window.

But LSTMs were conceived to learn any long-term patterns. One might desire to learn patterns as long back in the past as the data goes.

The *stateful* mode in LSTM layers enables this with `LSTM(..., stateful=True)`. In this mode, the last hidden and cell states of a batch go back in the LSTM cell as the initial hidden and cell states for the next batch.

This is shown with dashed lines exiting and entering a cell in Figure 5.7a. Note again that this reuse of states happens in stateful mode only. Also, in this mode, the batches are not shuffled, i.e.,

`model.fit(..., shuffle=False)`. This is to maintain the time order of the samples to use the learning from the chronologically previous sample.

This mode appears tempting. It promises to provide a useful power of learning exhaustive long-term patterns. Still, it is and should **not** be a preferred choice in most problems.

Stateful LSTM is appropriate if the time series/sequence data set is stationary. In simple words, it means if the relationships stay the same from the beginning to the end (in time). For example, text documents.

However, this is not true in many real-world problems. And, therefore, a stateful LSTM is not the default choice.

5.3.3 Return Sequences vs Last Output

Return sequences

The cells in an LSTM layer can be made to return sequences as an output by setting `LSTM(..., return_sequences=True)`. In this setting, each cell emits a sequence of length same as the input. Therefore, if a layer has l cells, the output shape is $(n\_batch, timesteps, l)$.

Sequences should be returned when the temporal structure needs to be preserved. This requirement is usually when

- the model output is a sequence. For example, in sequence-to-sequence models for language translation. Or,
- the subsequent layer needs sequence inputs. For example, a stack of LSTM (or convolutional) layers to sequentially extract temporal patterns at different levels of abstraction.

The illustration in Figure 5.7a is sequentially extracting temporal patterns using two LSTM layers. For this, the first LSTM layer (`LSTM Layer-1`) has to return a sequence.

This is shown in the figure in which the time-step iterations (green boxes within a cell) emit an output that is transmitted to the corresponding time-step iteration in the cell of the next LSTM layer.

In this network, returning sequences from the last LSTM layer

(`LSTM Layer-2`) becomes a choice. Figure 5.7b2 shows this choice. In this setting, a cell emits a sequence of outputs. The output is a sequence $\{\boldsymbol{h}_{T-2}, \boldsymbol{h}_{T-1}, \boldsymbol{h}_T\}$ with the same time-steps as the input sample $\{\boldsymbol{x}_{T-2}, \boldsymbol{x}_{T-1}, \boldsymbol{x}_T\}$. This output is flattened to a vector before sending to the output dense layer[1].

The last LSTM layer emitting sequences is termed as *unrestricted* LSTM network. Because the model is not restricting the output layer to use only the last outputs of the LSTM cells. These networks are larger but have the potential to yield better results.

Return last output

In this setting, a cell in an LSTM layer emits only the last time-step output $\boldsymbol{h}_t$. This is done by setting `LSTM(..., return_sequences=False)`. The output shape is (n_batch, l).

The last time-step output $\boldsymbol{h}_T$ is an amalgamation of information present in all the cell states $\{\boldsymbol{c}_T, \boldsymbol{c}_{T-1}, \ldots, \boldsymbol{c}_{T-\tau}\}$ and the prior cell outputs
$\{\boldsymbol{h}_{T-1}, \boldsymbol{h}_{T-2}, \ldots, \boldsymbol{h}_{T-\tau}\}$.

This is usually required in the following scenarios,

- The encoder in an LSTM autoencoder. The encodings are generally a vector. In some LSTM autoencoders, the encoder LSTM layer emits the last output vector as the encodings.

- Sequence to scaler model. A classifier is a good example of such models in which the input is a sequence and the output is a class (a scaler).

Figure 5.7b1 illustrates this setting in the last layer of the LSTM network. It is called a *restricted* LSTM network because the last layer's output is restricted. As shown in the figure, only the last time-step output $\boldsymbol{h}_T$ (from the last green box) is emitted and sent to the next layer.

[1]Flattening is, although, optional in TensorFlow because the dense layer automatically takes shape based on the input.

5.4 Initialization and Data Preparation

5.4.1 Imports and Data

We get started with importing the libraries. LSTM related classes are taken from `tensorflow` library. Also, the user-defined libraries, viz. `datapreprocessing`, `performancemetrics`, and `simpleplots`, are imported.

Listing 5.1. LSTM library imports.

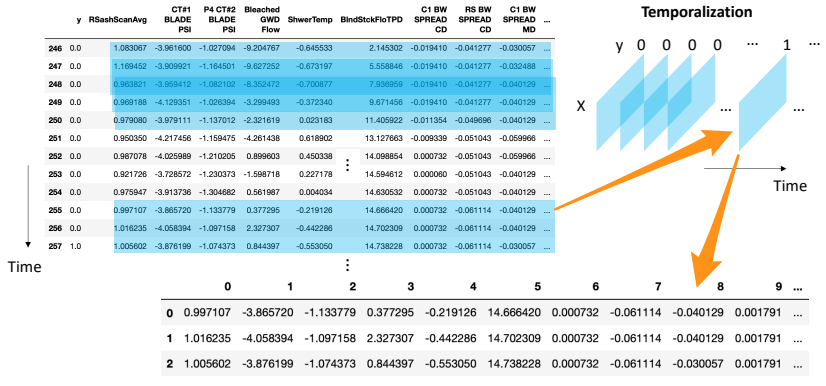
```
import pandas as pd
import numpy as np

import tensorflow as tf
from tensorflow.keras import optimizers
from tensorflow.keras.models import Model
from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Input
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import LSTM

from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Bidirectional

from tensorflow.python.keras import backend as K

from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

import matplotlib.pyplot as plt
import seaborn as sns

# user-defined libraries
import utilities.datapreprocessing as dp
import utilities.performancemetrics as pm
import utilities.simpleplots as sp

from numpy.random import seed
seed(1)
```

```
32
33 SEED = 123  # used to help randomly select the data
       points
34 DATA_SPLIT_PCT = 0.2
35
36 from pylab import rcParams
37 rcParams['figure.figsize'] = 8, 6
38 plt.rcParams.update({'font.size': 22})
```

Next, the data is read and the basic pre-processing steps are performed.

Listing 5.2. Data loading and pre-processing.

```
1  df = pd.read_csv("data/processminer-sheet-break-rare
       -event-dataset.csv")
2  df.head(n=5)  # visualize the data.
3
4  # Convert Categorical column to Dummy
5  hotencoding1 = pd.get_dummies(df['Grade&Bwt'])
6  hotencoding1 = hotencoding1.add_prefix('grade_')
7  hotencoding2 = pd.get_dummies(df['EventPress'])
8  hotencoding2 = hotencoding2.add_prefix('eventpress_'
       )
9
10 df = df.drop(['Grade&Bwt', 'EventPress'], axis=1)
11
12 df = pd.concat([df, hotencoding1, hotencoding2],
       axis=1)
13
14 # Rename response column name for ease of
       understanding
15 df = df.rename(columns={'SheetBreak': 'y'})
16
17 # Shift the response for training the model early
       prediction.
18 df = curve_shift(df, shift_by=-2)
19
20 # Sort by time and drop the time column.
21 df['DateTime'] = pd.to_datetime(df.DateTime)
22 df = df.sort_values(by='DateTime')
23 df = df.drop(['DateTime'], axis=1)
```



	y	RSashScanAvg	CT#1 BLADE PSI	P4 CT#2 BLADE PSI	Bleached GWD Flow	ShwerTemp	BindStckFloTPD	C1 BW SPREAD CD	RS BW SPREAD CD	C1 BW SPREAD MD	...
246	0.0	1.083067	-3.961600	-1.027094	-9.204767	-0.645533	2.145302	-0.019410	-0.041277	-0.030057	...
247	0.0	1.169452	-3.909921	-1.164501	-9.627252	-0.673197	5.558846	-0.019410	-0.041277	-0.032488	...
248	0.0	0.963821	-3.959412	-1.082102	-8.352472	-0.700877	7.936959	-0.019410	-0.041277	-0.040129	...
249	0.0	0.969188	-4.129351	-1.026394	-3.299493	-0.372340	9.671456	-0.019410	-0.041277	-0.040129	...
250	0.0	0.979080	-3.979111	-1.137012	-2.321619	0.023183	11.405922	-0.011354	-0.049696	-0.040129	...
251	0.0	0.950350	-4.217456	-1.159475	-4.261438	0.618902	13.127663	-0.009339	-0.051043	-0.059966	...
252	0.0	0.987078	-4.025989	-1.210205	0.899603	0.450338	14.098854	0.000732	-0.051043	-0.059966	...
253	0.0	0.921726	-3.728572	-1.230373	-1.598718	0.227178	14.594612	0.000060	-0.051043	-0.040129	...
254	0.0	0.975947	-3.913736	-1.304682	0.561987	0.004034	14.630532	0.000732	-0.051043	-0.040129	...
255	0.0	0.997107	-3.865720	-1.133779	0.377295	-0.219126	14.666420	0.000732	-0.061114	-0.040129	...
256	0.0	1.016235	-4.058394	-1.097158	2.327307	-0.442286	14.702309	0.000732	-0.061114	-0.040129	...
257	1.0	1.005602	-3.876199	-1.074373	0.844397	-0.553050	14.738228	0.000732	-0.061114	-0.030057	...

	0	1	2	3	4	5	6	7	8	9	...
0	0.997107	-3.865720	-1.133779	0.377295	-0.219126	14.666420	0.000732	-0.061114	-0.040129	0.001791	...
1	1.016235	-4.058394	-1.097158	2.327307	-0.442286	14.702309	0.000732	-0.061114	-0.040129	0.001791	...
2	1.005602	-3.876199	-1.074373	0.844397	-0.553050	14.738228	0.000732	-0.061114	-0.030057	0.001791	...

Figure 5.8. *Testing data temporalization.*

5.4.2 Temporalizing the Data

From an LSTM modeling standpoint, a usual two-dimensional input, also referred to as planar data, does not directly provide time-windows. But the time-windows are required to extract spatio-temporal patterns.

Therefore, the planar data is *temporalized*. In temporalization, a time-window spanning observations in $t : t - lookback$ is taken at a time point t and placed at index t in a three-dimensional array.

This is visualized in Figure 5.8. An example of real data is shown in the figure. Here a time-window of predictors of size three (`lookback=3`) at time index `257` is taken out and stacked in a 3-dimensional array at the same index `257`.

No reorientation in the response `y` is made. It is only ensured that the indexes of `y` are synchronized with the time-window stack of predictors which is now a 3-dimensional array.

With temporalized input, the model has access to the current and the past predictor observations $\boldsymbol{x}_{t:(t-lookback)}$ that led to the observed response y_t. This access enables the model to learn the temporal patterns.

Data temporalization is essential to learn temporal patterns.

The `lookback` is also referred to as `timesteps` to imply the prior time-steps the model is looking at for learning the patterns.

Data is temporalized in line 3 in Listing 5.3. The shape of the temporalized data is: (*samples*, *timesteps*, *features*). A lookback (or time-steps) of `5` is chosen. This implies the model will look at up to the past 5 observations. This equates to a time-window of 10 minutes in the sheet-break data set.

Listing 5.3. Data Temporalization.

```
1 # Temporalize the data
2 lookback = 5
3 X, y = temporalize(X=input_X, y=input_y, lookback=
      lookback)
```

5.4.3 Data Splitting

At this stage, the data is split into train, valid, and test. Fortunately, the `train_test_split()` function in `sklearn` can be used directly on higher-dimensional arrays. Irrespective of the array dimension, the function does the split along the first axis. This is done in Listing 5.4.

`sklearn.model_selection.train_test_split()` *is agnostic to the shape of the input array. It always samples w.r.t. the array's first dimension.*

Listing 5.4. Temporalized data split.

```
1 X_train, X_test, y_train, y_test =
2     train_test_split(X, y,
3                      test_size=DATA_SPLIT_PCT,
4                      random_state=SEED)
5 X_train, X_valid, y_train, y_valid =
6     train_test_split(X_train, y_train,
```

```
7                         test_size=DATA_SPLIT_PCT,
8                         random_state=SEED)
9
10 TIMESTEPS = X_train.shape[1]  # equal to the
       lookback
11 N_FEATURES = X_train.shape[2]  # the number of
       features
```

5.4.4 Scaling Temporalized Data

The 3-dimensional data is scaled using a udf, `scale()`, in Listing 5.5.

Listing 5.5. Scaling temporalized data.

```
1 # Fit a scaler using the training data.
2 scaler = StandardScaler().fit(dp.flatten(X_train))
3 X_train_scaled = dp.scale(X_train, scaler)
4 X_valid_scaled = dp.scale(X_valid, scaler)
5 X_test_scaled = dp.scale(X_test, scaler)
```

Besides, it is not preferred to scale the initial 2-dimensional data before temporalization because it will lead to the leakages in the time-window during the data split.

5.5 Baseline Model—A Restricted Stateless LSTM

Similar to the previous chapter, it is always advisable to begin with a baseline model. A restricted stateless LSTM network is taken as a baseline. In such a network, every LSTM layer is stateless and the final layer has a *restricted* output, i.e.,
`LSTM(..., stateful=False, return_sequences=False)`.

In the following, the baseline model will be constructed step-by-step.

5.5.1 Input layer

As mentioned before, the input layer in LSTM expects 3-dimensional inputs. The input shape should be: (*batch size*, *timesteps*, $features$).

A stateless LSTM does not require to explicitly specify the batch size (a stateful LSTM does require the batch size as mentioned in Appendix F). Therefore, the input shape is defined as follows in Listing 5.6.

Listing 5.6. LSTM input layer.

```
model = Sequential()
model.add(Input(shape=(TIMESTEPS, N_FEATURES),
                name='input'))
```

The input shape can also be provided as an argument to the first LSTM layer defined next. However, similar to the previous chapter this is explicitly defined for clarity.

5.5.2 LSTM layer

As a rule-of-thumb, two hidden LSTM layers are stacked in the baseline model. The `recurrent_activation` argument is left as its default `sigmoid` while the output `activation` is set as `relu` (refer to § 5.2.5). The `relu` activation came into existence after LSTMs. Therefore, they are not in the legacy LSTM definitions but can be used on the output.

Listing 5.7. LSTM layers.

```
model.add(
    LSTM(units=16,
         activation='relu',
         return_sequences=True,
         name='lstm_layer_1'))
model.add(
    LSTM(units=8,
         activation='relu',
         return_sequences=False,
         name='lstm_layer_2'))
```

The first LSTM layer has `return_sequences` set as `True`. This layer, therefore, yields the hidden outputs for every time-step. Consequently, the first layer output is (*batch size*, *timesteps*, 16), where 16 is the layer size.

Since this is a restricted LSTM network, the last LSTM layer is set with `return_sequences` as `False`. Therefore, it returns the output from

only the last time-step. Thus, the layer output is of shape: (*batch size*, 8), where 8 is the layer size.

5.5.3 Output layer

The output layer should be a `Dense` layer in an LSTM network and most other networks, in general.

Why? The output layer should be dense because it performs an affine transformation on the output of the ultimate hidden layer. The purpose of complex hidden layers, such as LSTM and Convolutional, is to extract predictive features. But these abstract features do not necessarily translate to the model output y. A dense layer's affine transformation puts together these features and translates them to the output y.

The output layer should be a `Dense` *layer for most deep learning networks.*

We add a `Dense` layer of size 1. As also mentioned in §4.4.4, the size is based on the number of responses. For a binary classifier, like here, the size is one with `sigmoid` activation. If we have a multi-class classifier, the size should be the number of labels with `softmax` activation.

Listing 5.8. LSTM network output layer.

```
model.add(Dense(units=1,
                activation='sigmoid',
                name='output'))
```

5.5.4 Model Summary

At this stage, the structure of the baseline LSTM model is ready. Before moving forward, the model structure should be glanced at using the `model.summary()` function.

Listing 5.9. LSTM baseline model summary.

```
model.summary()
```

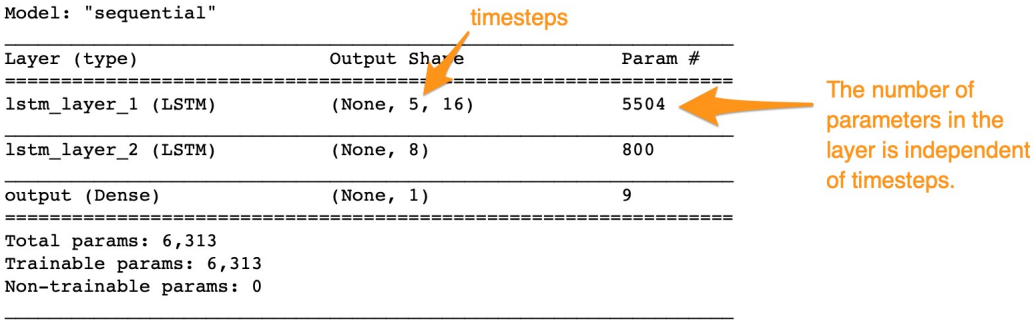


Figure 5.9. *LSTM baseline model summary.*

```
2
3 # Number of parameters = 4l(p + l + 1),
4 # l = layer size, p = number of features.
5 4*16*(n_features + 16 + 1) # Parameters in
      lstm_layer_1
6 # 5504
```

The summary in Figure 5.9 shows the number of parameters in each layer. For self-learning, it can be computed by hand using Eq. 5.2. For example, the number of parameters in the first LSTM layer is $4 \times 16(n\_features + 16 + 1) = 5504$, where 16 is the layer size.

5.5.5 Compile and Fit

The model `compile()` and `fit()` arguments are explained in § 4.4.6 and 4.4.7. They are directly applied here.

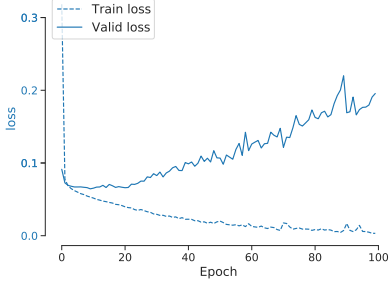
Listing 5.10. LSTM baseline model compile and fit.

```
1 model.compile(optimizer='adam',
2                loss='binary_crossentropy',
3                metrics=[
4                    'accuracy',
5                    tf.keras.metrics.Recall(),
6                    pm.F1Score(),
7                    pm.FalsePositiveRate()
8                ])
```

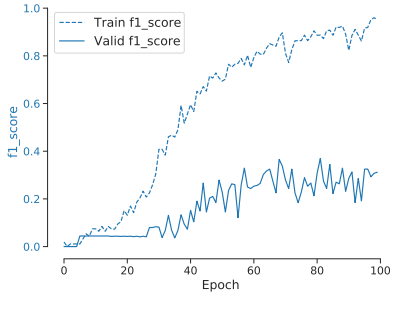
```
9
10 history = model.fit(x=X_train_scaled,
11                      y=y_train,
12                      batch_size=128,
13                      epochs=100,
14                      validation_data=(X_valid_scaled,
15                                       y_valid),
16                      verbose=0).history
```

The results are shown in Figure 5.10a-5.10c.

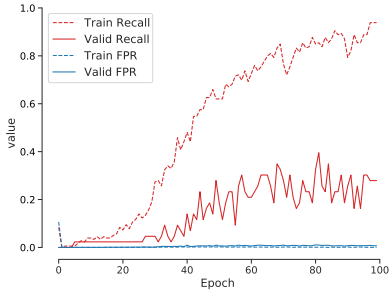
The accuracy metrics for the baseline LSTM model is already better than the best achieved with MLP. This was expected from an LSTM model because it is capable of drawing the temporal patterns. A few model improvements are attempted next to further improve accuracy.



(a) *Loss.*



(b) *F1-score.*



(c) *Recall and FPR.*

Figure 5.10. *LSTM baseline model results.*

5.6 Model Improvements

5.6.1 Unrestricted LSTM Network

§ 5.3.3 discussed two choices in LSTM networks: return sequences vs return last output at the final LSTM layer. The latter is typically the default choice and used in the baseline restricted model above. However, the former can likely improve the model.

The former choice enables the ultimate LSTM layer to emit a sequence of hidden outputs. Since the last LSTM layer is not restricted to emitting only the final hidden output, this network is called an *unrestricted* LSTM network. A potential benefit of this network is the presence of more intermediate features.

Based on this hypothesis, an unrestricted network is constructed in Listing 5.11.

Listing 5.11. Unrestricted LSTM model.

```
model = Sequential()
model.add(Input(shape=(TIMESTEPS, N_FEATURES),
                name='input'))
model.add(
    LSTM(units=16,
         activation='relu',
         return_sequences=True,
         name='lstm_layer_1'))
model.add(
    LSTM(units=8,
         activation='relu',
         return_sequences=True,
         name='lstm_layer_2'))
model.add(Flatten())
model.add(Dense(units=1,
                activation='sigmoid',
                name='output'))

model.summary()

model.compile(optimizer='adam',
              loss='binary_crossentropy',
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_layer_1 (LSTM)          (None, 5, 16)             5504
_________________________________________________________________
lstm_layer_2 (LSTM)          (None, 5, 8)              800
_________________________________________________________________
flatten (Flatten)            (None, 40)                0
_________________________________________________________________
output (Dense)               (None, 1)                 41
=================================================================
Total params: 6,345
Trainable params: 6,345
Non-trainable params: 0
_________________________________________________________________
```

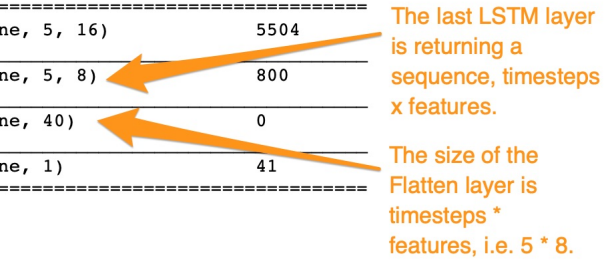


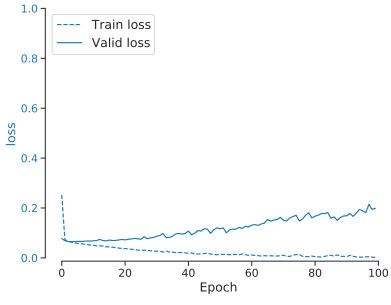
Figure 5.11. *LSTM full model summary.*

```
23                metrics=[
24                    'accuracy',
25                    tf.keras.metrics.Recall(),
26                    pm.F1Score(),
27                    pm.FalsePositiveRate()
28                ])
29 history = model.fit(x=X_train_scaled,
30                      y=y_train,
31                      batch_size=128,
32                      epochs=100,
33                      validation_data=(X_valid_scaled,
34                                       y_valid),
35                      verbose=0).history
```

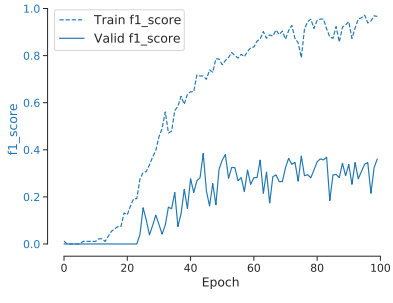
In the construction, the ultimate LSTM layer `lstm_layer_2` is set with `return_sequences=True`. The model summary in Figure 5.11 is showing its effect.

Unlike the baseline model, `lstm_layer_2` is now returning 3-dimensional outputs of shape (*batchsize*, 5, 8), where 5 and 8 are the time-steps and the layer size, respectively. This 3D tensor is flattened before passing on to the output `Dense` layer (also depicted in Figure 5.7b2).

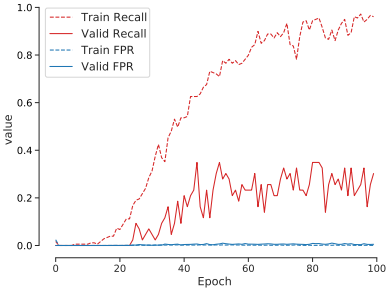
In this network, the parameters in the output `Dense` layer increases from 9 (= 8 weights + 1 bias) in the baseline network (Figure 5.9) to 41 (= 40 weights + 1 bias).



(a) *Loss.*



(b) *F1-score.*



(c) *Recall and FPR.*

Figure 5.12. *Unrestricted LSTM model results.*

The results are shown in Figure 5.12a-5.12c. The accuracy improved with the unrestricted model. For the given problem, the temporal intermediate features seem to be predictive but this cannot be generalized.

5.6.2 Dropout and Recurrent Dropout

Dropout is a common technique used for deep learning network improvement. However, it does not always work with RNNs including LSTMs. Gal and Ghahramani 2016 made an extreme claim stating,

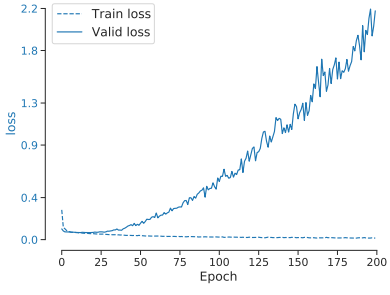
> "Dropout is a popular regularization technique with deep networks where network units are randomly masked during training (dropped). But the technique has never been applied successfully to RNNs."

True to the claim, a regular dropout does not always work with LSTMs. However, there is another type of dropout available in RNNs called *recurrent dropout*. In this technique, a fraction of inputs to the recurrent states is dropped. Both these dropouts are applied together in Listing 5.12 and are found to improve the model.

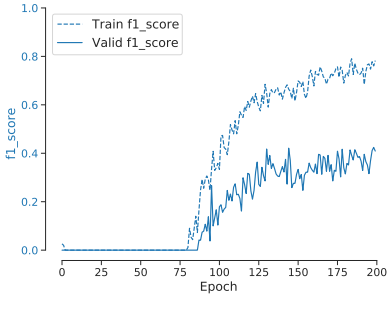
Listing 5.12. Unrestricted LSTM with regular and recurrent dropout.

```
model = Sequential()
model.add(Input(shape=(TIMESTEPS, N_FEATURES),
                name='input'))
model.add(
    LSTM(units=16,
         activation='relu',
         return_sequences=True,
         recurrent_dropout=0.5,
         name='lstm_layer_1'))
model.add(Dropout(0.5))
model.add(
    LSTM(units=8,
         activation='relu',
         return_sequences=True,
         recurrent_dropout=0.5,
         name='lstm_layer_2'))
model.add(Flatten())
model.add(Dropout(0.5))
model.add(Dense(units=1,
                activation='sigmoid',
                name='output'))

model.summary()

model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=[
                  'accuracy',
                  tf.keras.metrics.Recall(),
                  pm.F1Score(),
                  pm.FalsePositiveRate()
              ])
history = model.fit(x=X_train_scaled,
                    y=y_train,
```

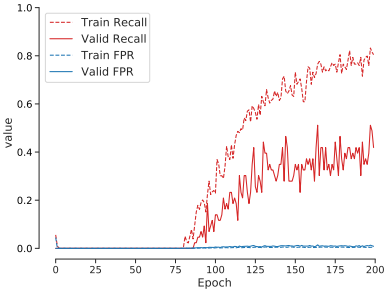
```
                    batch_size=128,
                    epochs=200,
                    validation_data=(X_valid_scaled,
                                     y_valid),
                    verbose=0).history
```



(a) *Loss.*



(b) *F1-score.*



(c) *Recall and FPR.*

Figure 5.13. *Unrestricted LSTM with dropout and recurrent dropout results.*

The results are shown in Figure 5.13a-5.13c. The accuracy was further improved. However, increasing validation loss is still observed. Besides, this model has trained over 200 epochs (as opposed to 100 in previous models) for the metrics to stabilize.

`Recurrent_dropout` is more applicable to LSTM networks than the regular dropout.

5.6.3 Go Backwards

Researchers at Google published a paper by Sutskever, Vinyals, and Le 2014. This paper is now one of the most popular papers on LSTMs. In this paper, they had an interesting finding. They quote,

> "...we found that reversing the order of the words in all source sentences (but not target sentences) improved the LSTM's performance markedly, because doing so introduced many short-term dependencies between the source and the target sentence which made the optimization problem easier."

Such a model can be made by setting `go_backwards=True` in the **first** LSTM layer. This processes the input sequence backward.

Note that only the first LSTM layer should be made backward because only the inputs need to be processed backward. Subsequent LSTM layers work on arbitrary features of the model. Making them backward is meaningless in most cases.

Listing 5.13. LSTM network with input sequences processed backward.

```
model = Sequential()
model.add(Input(shape=(TIMESTEPS, N_FEATURES),
                name='input'))
model.add(
    LSTM(units=16,
         activation='relu',
         return_sequences=True,
         go_backwards=True,
         name='lstm_layer_1'))
model.add(
    LSTM(units=8,
         activation='relu',
         return_sequences=True,
         name='lstm_layer_2'))
model.add(Flatten())
model.add(Dense(units=1,
                activation='sigmoid',
                name='output'))

```

```
model.summary()

model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=[
                  'accuracy',
                  tf.keras.metrics.Recall(),
                  pm.F1Score(),
                  pm.FalsePositiveRate()
              ])
history = model.fit(x=X_train_scaled,
                    y=y_train,
                    batch_size=128,
                    epochs=100,
                    validation_data=(X_valid_scaled,
                                     y_valid),
                    verbose=0).history
```

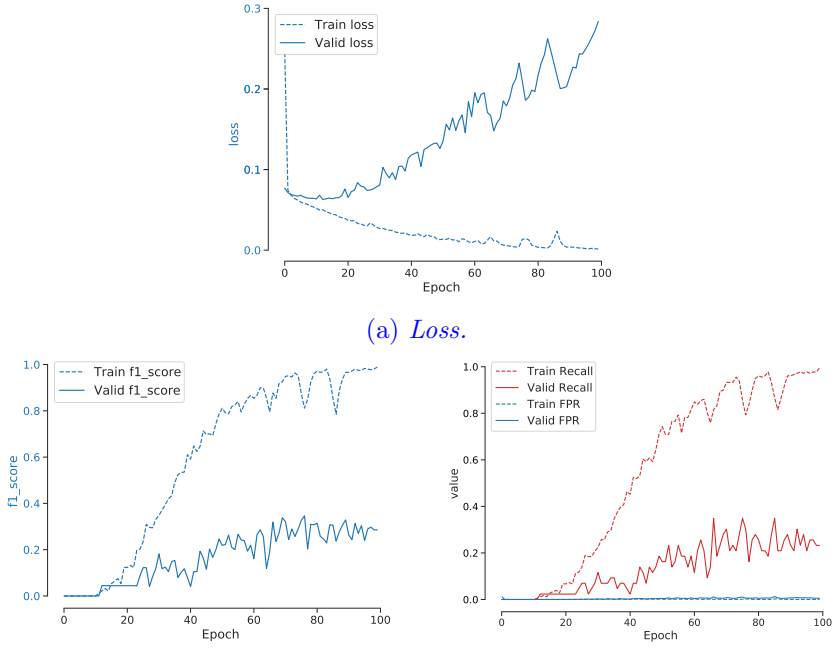
Backward sequence processing worked quite well for sequence-to-sequence model. Perhaps because the output was also a sequence and the initial elements in the input sequence were dependent on the last elements of the output sequence. Therefore, reversing the input sequence brought the related input-output elements closer.

However, as shown in the results in Figure 5.14a-5.14c, this approach fared close to or less than the baseline in our problem.

`Backward` LSTM models are effective on sequences in which the early segments have a larger influence on the future. For example, in language translations because the beginning of a sentence usually has a major influence on how the sentence ends.

5.6.4 Bi-directional

A regular LSTM, or any RNN, learn the temporal patterns in a forward direction—going from past to the future. Meaning, at any time-step the cell state learns only from the past. It does not have visibility of the



(a) *Loss.*

(b) *F1-score.*

(c) *Recall and FPR.*

Figure 5.14. *LSTM with input processed backward results.*

future. This concept is clearer in Figure 5.2a. As shown in the figure, the truck of memories (cell state) moves from left-to-right.

Schuster and Paliwal 1997 made a significant contribution by proposing a bi-directional RNN. A bi-directional RNN adds a mirror RNN layer to the original RNN layer. The input sequence is passed as is to the original layer and in reverse to the mirror.

This enables the cell states to learn from all the input information, i.e., both the past and the future. This is illustrated in Figure 5.15. Similar to a regular LSTM layer, the information from the past flows into the cell state from left-to-right in the top lane. Additionally, the information from the future flows back to the cell states right-to-left in the bottom lane.

This ability to have collective information from the past and the future make a bi-directional LSTM layer more powerful. A popular

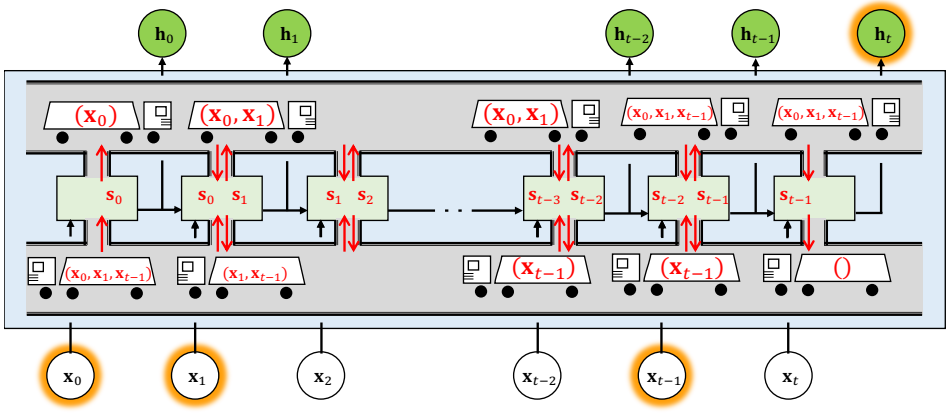


Figure 5.15. *LSTM bi-directional cell state mechanism.*

work by Graves and Schmidhuber 2005 on bi-directional LSTMs show that they significantly outperformed the traditional LSTMs in speech recognition. Graves and Schmidhuber 2005 also quote an important requirement for using a bi-directional LSTM,

> ...for temporal problems like speech recognition, relying on knowledge of the future seems at first sight to violate causality—at least if the task is online.
>
> ...However, human listeners do exactly that. Sounds, words, and even whole sentences that at first mean nothing are found to make sense in the light of the future context.

Therefore, we must recognize whether the problem is truly online, i.e., requiring an output for every new input, or semi-online when the output is needed at end of some input segment. Bi-directional LSTM does not apply to the former but could significantly improve performance in the latter.

The sheet-break problem at hand can be treated as semi-online. A prediction can be made after a window of sensor observations are made. In fact, from an LSTM standpoint, most problems are either offline (e.g., text documents) or semi-online (e.g., time series). A bi-directional LSTM network is, therefore, built in Listing 5.14.

Listing 5.14. Bi-directional LSTM network.

```
model = Sequential()
model.add(Input(shape=(TIMESTEPS, N_FEATURES),
                name='input'))
model.add(
    Bidirectional(
        LSTM(units=16,
             activation='relu',
             return_sequences=True),
        name='bi_lstm_layer_1'))
model.add(Dropout(0.5))
model.add(
    Bidirectional(
        LSTM(units=8,
             activation='relu',
             return_sequences=True),
        name='bi_lstm_layer_2'))
model.add(Flatten())
model.add(Dense(units=1,
                activation='sigmoid',
                name='output'))

model.summary()

model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=[
                  'accuracy',
                  tf.keras.metrics.Recall(),
                  pm.F1Score(),
                  pm.FalsePositiveRate()
              ])

history = model.fit(x=X_train_scaled,
                    y=y_train,
                    batch_size=128,
                    epochs=100,
                    validation_data=(X_valid_scaled,
                                     y_valid),
                    verbose=0).history
```

In TensorFlow, a bi-directional LSTM layer is made by wrapping

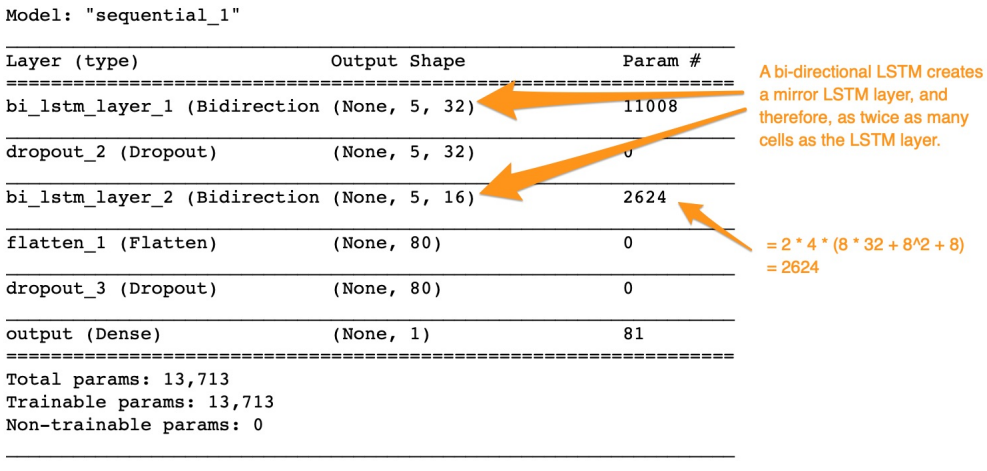


Figure 5.16. *LSTM bi-directional network summary.*

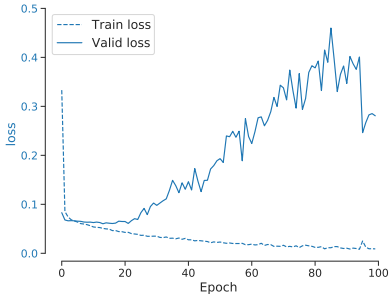
an `LSTM` layer within a `Bidirectional` layer. The `Bidirectional` layer creates a mirror layer for any RNN layer passed as an argument to it (an LSTM here).

As shown in Figure 5.16, this results in twice the number of parameters for a bi-directional layer compared to the ordinary LSTM layer. Expressed as

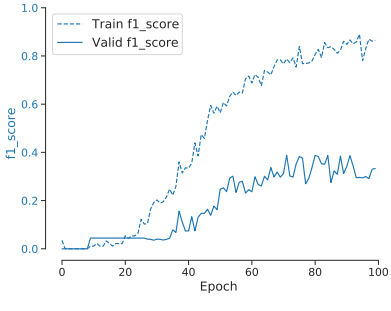
$$n\_parameters = 2 \times 4l(p + l + 1) \tag{5.6}$$

where l and p are the size of the layer and number of features, respectively. Bi-directional networks, therefore, require a sufficient amount of training data. Absence of which may render it less effective.

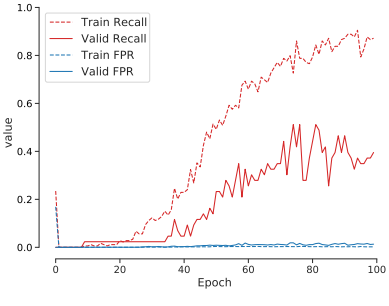
The results of the bi-directional network are shown in Figure 5.17a-5.17c. The accuracy is found to be higher than the prior models. This could be attributed to the bi-directional LSTM's ability to capture temporal patterns both retrospectively (backward) and prospectively (forward).



(a) *Loss.*



(b) *F1-score.*



(c) *Recall and FPR.*

Figure 5.17. *Bi-directional LSTM results.*

Bi-directional LSTMs learn temporal patterns both retrospectively and prospectively.

5.6.5 Longer Lookback/Timesteps

LSTMs are known to learn long-term dependencies. In the previous models, the lookback period was set as 5. A hypothesis could be that a longer lookback could capture more predictive patterns and improve accuracy. This is tried in this section.

First, the data is re-prepared with a longer lookback of 20 in Listing 5.15 by setting the `lookback = 20`.

Listing 5.15. Data re-preparation with longer lookback.

```
lookback = 20  # Equivalent to 40 min of past data.
# Temporalize the data
X, y = dp.temporalize(X=input_X,
                      y=input_y,
                      lookback=lookback)

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=DATA_SPLIT_PCT, random_state=
       SEED)
X_train, X_valid, y_train, y_valid =
   train_test_split(
    X_train, y_train, test_size=DATA_SPLIT_PCT,
       random_state=SEED)

X_train = X_train.reshape(X_train.shape[0],
                          lookback,
                          n_features)
X_valid = X_valid.reshape(X_valid.shape[0],
                          lookback,
                          n_features)
X_test = X_test.reshape(X_test.shape[0],
                        lookback,
                        n_features)

# Initialize a scaler using the training data.
scaler = StandardScaler().fit(dp.flatten(X_train))

X_train_scaled = dp.scale(X_train, scaler)
X_valid_scaled = dp.scale(X_valid, scaler)
X_test_scaled = dp.scale(X_test, scaler)
```

The unrestricted-LSTM network is then trained with the longer lookback data in Listing 5.16.

Listing 5.16. LSTM model with longer lookback.

```
timesteps = X_train_scaled.shape[1]

model = Sequential()
model.add(Input(shape=(timesteps, N_FEATURES),
                name='input'))
```

```
model.add(
    LSTM(units=16,
         activation='relu',
         return_sequences=True,
         recurrent_dropout=0.5,
         name='lstm_layer_1'))
model.add(Dropout(0.5))
model.add(
    LSTM(units=8,
         activation='relu',
         return_sequences=True,
         recurrent_dropout=0.5,
         name='lstm_layer_2'))
model.add(Flatten())
model.add(Dropout(0.5))
model.add(Dense(units=1,
                activation='sigmoid',
                name='output'))

model.summary()

model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=[
                  'accuracy',
                  tf.keras.metrics.Recall(),
                  pm.F1Score(),
                  pm.FalsePositiveRate()
              ])
history = model.fit(x=X_train_scaled,
                    y=y_train,
                    batch_size=128,
                    epochs=200,
                    validation_data=(X_valid_scaled,
                                     y_valid),
                    verbose=0).history
```

It was mentioned in § 5.2.6 that the number of parameters in an LSTM layer does not increase with the lookback. That is now evident in the model summary in Figure 5.18.

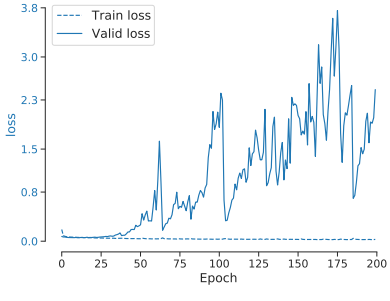
The results are in Figure 5.19a-5.19c. The performance deteriorated

Layer (type)	Output Shape	Param #
lstm_layer_1 (LSTM)	(None, 20, 16)	5504
dropout_15 (Dropout)	(None, 20, 16)	0
lstm_layer_2 (LSTM)	(None, 20, 8)	800
flatten_16 (Flatten)	(None, 160)	0
dropout_16 (Dropout)	(None, 160)	0
output (Dense)	(None, 1)	161

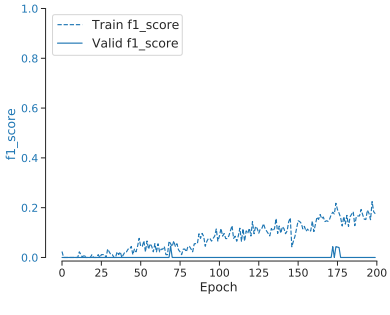
Total params: 6,465
Trainable params: 6,465
Non-trainable params: 0

The number of parameters is the same with 20 timesteps as it was with 5 timesteps.

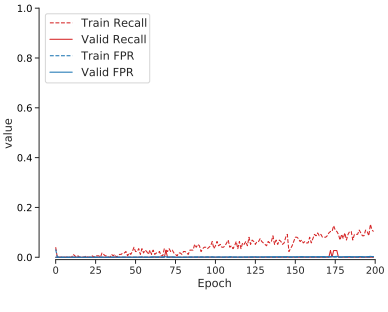
Figure 5.18. *LSTM with longer lookback network summary.*



(a) *Loss.*



(b) *F1-score.*



(c) *Recall and FPR.*

Figure 5.19. *Unrestricted-LSTM model with longer lookback results.*

with a longer lookback. These results confirm the statement by Jozefowicz, Zaremba, and Sutskever 2015. They stated that if the lookback period is long, the cell states fuse information across a wide window. Due to this, the extracted information gets smeared. As a result, protracting the lookback does not always work with LSTMs.

Increasing the lookback may smear the features learned by LSTMs and, therefore, cause poorer performance.

5.7 History of LSTMs

Until the early 1990s, RNNs were learned using real-time recurrent learning (RTRL, Robinson and Fallside 1987) and back-propagation through time (BPTT, R. J. Williams and Zipser 1995). But they had critical limitations: vanishing and exploding gradient.

These limitations were due to the error propagation mechanism in a recurrent network. In the early methods, the errors were progressively multiplied as it traveled back in time. Due to this, the resultant error could exponentially increase or decrease. Consequently, the backpropagated error can quickly either vanish or explode.

The gradient explosion was empirically addressed by capping any high values. But the vanishing gradient issue was still an unsolved problem. It was casting doubt on whether RNNs can indeed exhibit significant practical advantages.

Then, long short term memory (LSTM) was developed by Hochreiter and Schmidhuber 1997. They stated,

> "...as the time lag increases, (1) stored information must be protected against perturbation for longer and longer periods, and—especially in advanced stages of learning—(2) more and more already correct outputs also require protection against perturbation."
>
> – p7 in Hochreiter and Schmidhuber 1997.

Table 5.2. History of LSTM.

1997	Hochreiter & Schmidhuber • Memory state.
.	
1999	Gers, Schmidhuber, & Cummins • Memory state, and • Forget gate.
2002	Gers, Schraudolph & Schmidhuber • Memory state, • Forget gate, and • Peeping hole.
2007-2008	Graves, Fernández, & Schmidhuber, Graves & Schmidhuber • Memory state, • Forget gate, • Peeping hole, and • Multi-dimensional.
2012-2013	Graves 2012, Graves 2013 • Memory state, • Forget gate, • Peeping hole, • Multi-dimensional, and • refined formulation
2015	Jozefowicz, Zaremba, & Sutskever • Memory state, • Forget gate, • Multi-dimensional, • refined formulation, and • *implemented in TensorFlow.*
2017	Baytas, Xiao, Zhang, et. al. • Handle irregular time-intervals (time aware),
⋮	

They recognized that the vanishment problem is due to error propagation. If only the error could be left untouched during the time travel, the vanishment and explosion would not occur.

Hochreiter and Schmidhuber 1997 implemented this idea by enforcing *constant* error flow through what they called "constant error carousals (CECs)." Their CEC implementation was done with the help of adding a recurring cell state. An abstract and simplified representation of their formulation is shown below.

$$
\begin{aligned}
i_t &= f(\boldsymbol{w}_i \boldsymbol{y}_{t-1} + b_i) \\
o_t &= f(\boldsymbol{w}_o \boldsymbol{y}_{t-1} + b_o) \\
\tilde{c}_t &= g(\boldsymbol{w}_c \boldsymbol{y}_{t-1} + b_c) \\
c_t &= c_{t-1} + i_t \tilde{c}_t \\
y_t &= o_t h(s_t)
\end{aligned}
$$

The first three expressions are called *input* gate, *output* gate, and *state* gate, respectively. The last two expressions are the *cell state* and *cell output*, respectively. In this section, it is okay to ignore the equation details. Here the focus is on the formulation and their key differentiating elements highlighted in red.

The key element in Hochreiter and Schmidhuber's formulation above is the cell state c_t. The cell state acts as long-term memory.

It has an additive expression instead of multiplicative. The expression can also be seen as computing the delta, $\Delta c_t = i_t \tilde{c}_t$, at each time-step and adding it to the cell state c_t. While it is true that additive c_t does not necessarily result in a more powerful model, the gradients of such RNNs are better behaved as they do not cause vanishment (explained in § 5.2.8).

But this approach has another issue. The additive cell state expression does not forget a past. It will keep the memories from all the time-steps in the past. Consequently, Hochreiter and Schmidhuber's LSTM will not work if the memories have limited relevance in time.

Hochreiter and Schmidhuber worked around this by performing an *apriori* segmentation of time series into subsequences such that all time-steps in the subsequence are relevant. But such an apriori processing is

a methodological limitation.

Gers, Schmidhuber, and Cummins 1999 addressed this by bringing *forget* gates into the formulation. They stated, "any training procedure for RNNs which is powerful enough to span long time lags must also address the issue of **forgetting** in short-term memory."

In Gers, Schmidhuber, and Cummins 1999, it is emphasized that the cell state c_t tends to grow linearly during a time series traversal. If a continuous time series stream is presented, the cell states may grow in an unbounded fashion. This causes saturation of the output squashing function $h(c_t)$ at the end[2].

Gers et. al. countered this with *adaptive* forget gates in Gers, Schmidhuber, and Cummins 1999. These gates learn to reset the cell states (the memory) once their contents are out-of-date and, hence, useless. This is done by a multiplicative forget gate activation f_t. f_t can also be seen as a weight on the prior memory shown in their high-level formulation below.

$$
\begin{aligned}
i_t &= f(\boldsymbol{w}_i \boldsymbol{y}_{t-1} + b_i) \\
o_t &= f(\boldsymbol{w}_o \boldsymbol{y}_{t-1} + b_o) \\
f_t &= f(\boldsymbol{w}_f \boldsymbol{y}_{t-1} + b_f) \\
\tilde{c}_t &= g(\boldsymbol{w}_c \boldsymbol{y}_{t-1} + b_c) \\
c_t &= f_t c_{t-1} + i_t \tilde{c}_t \\
y_t &= o_t h(c_t)
\end{aligned}
$$

After forget gates, Gers and Schmidhuber were back on this. They, along with Schrandolph, devised what they called as a *peeping hole* in Gers, Schraudolph, and Schmidhuber 2002. The name may sound creepy but the approach was scientific.

In the LSTMs, thus far, each gate receives connections from the input and the output of the cells. But there is no direct connection between the gates and the cell state (memory) they are supposed to control. The resulting lack of essential information (cell state) may harm a network's

[2]**Saturation** meaning: At a saturation point of a function, any change in its input does not change the output. That is, $y = f(x) = f(x + \Delta x)$ if $x, x + \Delta x \in$ saturation region.

performance (Gers, Schraudolph, and Schmidhuber 2002).

As a remedy, Gers et. al. added weighted "peephole" connections from the cell states to the input, output, and forget gates shown below.

$$
\begin{aligned}
i_t &= f(\boldsymbol{w}_i^{(y)}\boldsymbol{y}_{t-1} + \boldsymbol{w}_i^{(c)}\boldsymbol{c}_{t-1} + b_i) \\
o_t &= f(\boldsymbol{w}_o^{(y)}\boldsymbol{y}_{t-1} + \boldsymbol{w}_o^{(c)}\boldsymbol{c}_{t-1} + b_o) \\
f_t &= f(\boldsymbol{w}_f^{(y)}\boldsymbol{y}_{t-1} + \boldsymbol{w}_f^{(c)}\boldsymbol{c}_{t-1} + b_f) \\
\tilde{c}_t &= g(\boldsymbol{w}_c^{(y)}\boldsymbol{y}_{t-1} + b_c) \\
c_t &= f_t c_{t-1} + i_t \tilde{c}_t \\
y_t &= o_t h(c_t)
\end{aligned}
$$

The next wave of LSTM progress can be attributed to Alex Graves for his work in 2005–2015. He along with Santiago Fernandex and Jurger Schmidhuber developed the foundation of multi-dimensional RNNs in Graves, Fernández, and Schmidhuber 2007.

To avoid confusion for statisticians, the multi-dimension here refers to the number of input sample axes and not the features. For example, a time series has one axes while an image has two axes.

Until this work, LSTM/RNNs were applicable only to single-axes sequence problems, such as speech recognition. Applying RNNs to data with more than one spatio-temporal axes was not straightforward. Graves et. al. (2007) laid down the formulation for multi-dimension/axes sequences.

The multi-dimensional extension was a significant leap that made RNNs, in general, and LSTMs, specifically, applicable to multivariate time series, video processing, and other areas.

This work was carried forward by Graves and Schmidhuber in Graves and Schmidhuber 2009 that won the ICDAR handwriting competition in 2009.

Then in 2012-2013 Graves laid a refined LSTM version in Graves 2012; Graves 2013 that we are familiar with today. His formulation for multi-dimensional sequences is shown below.

$$
\begin{aligned}
i_t &= f(W_i^{(x)}\boldsymbol{x}_t + W_i^{(h)}\boldsymbol{h}_{t-1} + W_i^{(c)}\boldsymbol{c}_{t-1} + b_i) \\
o_t &= f(W_o^{(x)}\boldsymbol{x}_t + W_o^{(h)}\boldsymbol{h}_{t-1} + W_o^{(c)}\boldsymbol{c}_{t-1} + b_o) \\
f_t &= f(W_f^{(x)}\boldsymbol{x}_t + W_f^{(h)}\boldsymbol{h}_{t-1} + W_f^{(c)}\boldsymbol{c}_{t-1} + b_f) \\
\tilde{c}_t &= g(W_c^{(x)}\boldsymbol{x}_t + W_c^{(h)}\boldsymbol{h}_{t-1} + b_c) \\
c_t &= f_t c_{t-1} + i_t \tilde{c}_t \\
h_t &= o_t g(c_t)
\end{aligned}
$$

In this formulation, Graves included the features $\boldsymbol{x}$ present along the axes of a multi-dimensional sample. This refined version was also a simplification of the previous versions in Hochreiter and Schmidhuber 1997, and Gers, Schmidhuber, and Cummins 1999; Gers, Schraudolph, and Schmidhuber 2002 (might not be apparent here because the formulations above are simplified representations).

While the previous works had a complex memory block concept with byzantine architecture (not shown here for clarity), Graves new formulation had a simpler memory cell.

Jozefowicz and Sutseker from Google, and Zaremba from FaceBook took forward Graves formulation that ultimately led to the current LSTM implementation in TensorFlow. They explored the LSTM variants in Jozefowicz, Zaremba, and Sutskever 2015 and recommended the formulation by Graves 2012 but without the peephole.

$$
\begin{aligned}
i_t &= f(W_i^{(x)}\boldsymbol{x}_t + W_i^{(h)}\boldsymbol{h}_{t-1} + b_i) \\
o_t &= f(W_o^{(x)}\boldsymbol{x}_t + W_o^{(h)}\boldsymbol{h}_{t-1} + b_o) \\
f_t &= f(W_f^{(x)}\boldsymbol{x}_t + W_f^{(h)}\boldsymbol{h}_{t-1} + b_f) \\
\tilde{c}_t &= g(W_c^{(x)}\boldsymbol{x}_t + W_c^{(h)}\boldsymbol{h}_{t-1} + b_c) \\
c_t &= f_t c_{t-1} + i_t \tilde{c}_t \\
h_t &= o_t g(c_t)
\end{aligned}
$$

In Jozefowicz et. al. (2015) and, consequently, in TensorFlow the LSTM's hidden state is a tuple $(\boldsymbol{h}_t, \boldsymbol{c}_t)$. The cell state, $\boldsymbol{c}_t$, is called a "slow" state that addresses the vanishing gradient problem, and $\boldsymbol{h}_t$ is called a "fast" state that allows the LSTM to make complex decision

over short periods of time.

The developments in LSTMs and RNNs have continued. Time aware LSTM by Baytas et al. 2017 was proposed to handle irregular time intervals between the time-steps in a sequence. A novel memory cell called Legendre memory unit (LMU) was developed recently for RNNs by Voelker, Kajić, and Eliasmith 2019. Moreover, a time-segment LSTM and temporal inception by Ma et al. 2019 show interesting applications.

In sum, LSTM is an *enhanced* RNN. LSTM can learn long-term memories and also dispose of them when they become irrelevant. This is achieved due to the advancements and refinements over the years.

5.8 Summary

LSTM models showed to work better than MLPs. This was expected because they can learn temporal patterns. The baseline restricted LSTM model beat the best MLP model in the previous chapter. The unrestricted LSTM proved to perform even better. Adding a recurrent dropout for regularization further improved and stabilized the model.

Inspired from other works on sequence modeling, backward and bi-directional LSTM models were developed. The backward model performed below the baseline. They work better for sequence-to-sequence problems like language translations. However, the bi-directional model outperformed the others. This could be attributed to a bi-directional LSTM's ability to capture temporal patterns both retrospectively and prospectively.

Lastly, owing to the expectation from LSTMs to learn even longer-term patterns a wider time-window of inputs are used. This is done by re-preparing the data by increasing the lookback from 5 to 20. However, contrary to the expectation the performance degraded. Primarily due to LSTM cell state's limitation in fusing temporal patterns from wide time-windows. Stateful LSTMs is an alternative to learning exhaustively long-term patterns. Their implementation is shown in Appendix F as they are not directly applicable to the non-stationary time series process here.

Besides, the LSTM models constructed here faced the issue of increasing validation loss. This is further touched upon in the exercises. Finally, the chapter is concluded with a few rules-of-thumb.

5.9 Rules-of-thumb

- The thumb-rules for the number of layers, number of nodes and activation functions for the intermediate and output layers are the same as that for MLPs in § 4.10.
- **Data Processing**. The initial data processing, e.g., converting the data to numeric is the same as in MLP thumb-rules in § 4.10. Additionally,
  - **Temporalize**. The data temporalization into 3-dimensional arrays of shape, (*batch size*, *timesteps*, *features*), is necessary.
  - **Split**. Randomly split the temporalized data using `train_test_split` from `sklearn.model_selection`. The data split should be done after temporalization to avoid observations to leak between train, valid and test sets. Besides, as discussed in § 4.3.2, the temporalized data windows are self-contained. Therefore, random sampling of the time series is applicable.
  - **Scale**. Scaling the temporalized 3D data is facilitated with custom-defined functions in § 5.4.4. Fit a `StandardScaler` on the train set and transform the valid and test sets.
- **Restricted vs unrestricted LSTM**. It is preferable to work with unrestricted LSTM. It will typically provide better accuracy. This is done as follows,
  - **Return sequences**. Set the argument `return_sequences=True` in all the LSTM layers, including the last in the stack.
  - **Flatten**. If the layer next to the last LSTM layer is `Dense()`, add a `Flatten()` layer. The `Flatten()` is a transformation layer converting the 3-dimensional (*batch size*, *timesteps*, *features*) output from the LSTM layer with a time-steps axis into a 2-dimensional array (*batch size*, *timesteps* * *features*)

- **Stateful LSTM**. It should not be used as a baseline. A stateful LSTM requires a deeper understanding of the process and problem formulation. For example, whether the process is stationary. If unclear, it is better to avoid stateful LSTM.

- **Dropout**. In LSTM we have two choices for Dropout, viz. the regular dropout using the `Dropout()` layer and recurrent dropout using the `recurrent_dropout` argument in the `LSTM` layer. Among them it is preferred to start with the recurrent dropout and its rate as `0.5` in each LSTM layer, `LSTM(..., recurrent_dropout=0.5)`.

- **Go backward**. The `go_backwards` argument in an LSTM layer allows processing the input sequences in the reverse order. This brings the long-term patterns closer while moving the short-term patterns backward. This switch is useful in some problems, such as language translation.

  The *backward* utility depends on the problem. This setting should be set to `False` in the baseline.

  To test whether it brings any improvement, only the **first** LSTM layer should be toggled to go backward by setting `LSTM(..., go_backwards=True)`.

- **Bi-directional**. A bi-directional LSTM can learn patterns both retrospectively (from the past) and prospectively (from the future). This makes it stronger than regular LSTMs. However, bi-directional LSTM has double the number of parameters. It should not be used in the baseline but must be attempted to validate if it brings any improvement.

  An LSTM layer can be made bi-directional by wrapping it as `bidirectional(LSTM(...))`. Similarly, any other RNN layer can also be made bi-directional using the `bidirectional()` wrapper.

5.10 Exercises

1. LSTM is said to be an enhanced RNN. It brought in the concept of cell state for long-term memory.

   (a) Explain how a simple RNN works and differentiate it with LSTMs.

   (b) In LSTMs, is it possible to identify which features are preserved or forgotten from the memory?

2. **Activation**. In this chapter, `relu` activation is used on the LSTM outputs. But `tanh` is its native activation.

   (a) Explain why `relu` is still applicable as LSTM output activation.

   (b) Train the baseline and bi-directional model with `tanh` activation and discuss the results.

   (c) Train the baseline and bi-directional models with ELU and SELU activations. Do they address the increasing validation loss issue? Discuss your findings.

3. **Peephole LSTM**. Peephole LSTMs are a useful variant. They have proved to work better than others in Graves 2013. Refer to the Peephole LSTM expressions in § 5.7.

   (a) Use the peephole LSTM in TensorFlow[3].

   (b) Implement the peephole LSTM as your custom LSTM cell.

   (c) Discuss the results.

4. **Gated Recurring Unit (GRU)**. GRU proposed in Cho et al. 2014 is an alternative to LSTM. Jozefowicz, Zaremba, and Sutskever 2015 found that GRU outperformed LSTM in many cases. The formulation for GRU is,

[3]Refer to `https://bit.ly/2JnxcyM`

$$
\begin{aligned}
r_t &= \sigma(W_r^{(x)}\boldsymbol{x}_t + W_r^{(h)}\boldsymbol{h}_{t-1} + b_r) \\
z_t &= \sigma(W_z^{(x)}\boldsymbol{x}_t + W_z^{(h)}\boldsymbol{h}_{t-1} + b_z) \\
\tilde{h}_t &= \tanh(W_h^{(x)}\boldsymbol{x}_t + W_h^{(h)}\boldsymbol{h}_{t-1} + b_h) \\
h_t &= z_t h_{t-1} + (1 - z_t)\tilde{h}_t
\end{aligned}
$$

   (a) Implement GRU in TensorFlow. Replace the LSTM layer with the GRU layer. Discuss the results.

   (b) Jozefowicz, Zaremba, and Sutskever 2015 found that initializing LSTM with forget gate bias as 1 improved the performance. Explain the premise of initializing the forget bias as 1.

   (c) Run the baseline and bi-directional LSTM model with forget bias initialized to 1 (set `unit_forget_bias` as `True`). Compare the results with the GRU model.

5. (Optional) **A higher-order cell state**. The cell state in traditional LSTM has a first-order autoregressive (AR-1) like structure. That is, the cell state c_t is additive with the prior c_{t-1}. Will a higher-order AR function work better?

   (a) Build a simple custom LSTM cell with a second-order autoregressive expression for cell state. The formulation is given below. This is a dummy LSTM cell in which cell state at a time t will always have a part of the cell state learned at $t-2$.

$$
\begin{aligned}
i_t &= \texttt{hard-sigmoid}(\boldsymbol{w}_i^{(x)}\boldsymbol{x}_t + \boldsymbol{w}_i^{(h)}\boldsymbol{h}_{t-1} + b_i) \\
o_t &= \texttt{hard-sigmoid}(\boldsymbol{w}_o^{(x)}\boldsymbol{x}_t + \boldsymbol{w}_o^{(h)}\boldsymbol{h}_{t-1} + b_o) \\
f_t &= \texttt{hard-sigmoid}(\boldsymbol{w}_f^{(x)}\boldsymbol{x}_t + \boldsymbol{w}_f^{(h)}\boldsymbol{h}_{t-1} + b_f) \\
\tilde{c}_t &= \texttt{tanh}(\boldsymbol{w}_c^{(x)}\boldsymbol{x}_t + \boldsymbol{w}_c^{(h)}\boldsymbol{h}_{t-1} + b_c) \\
c_t &= b + 0.5c_{t-2} + f_t c_{t-1} + i_t \tilde{c}_t \\
h_t &= o_t \texttt{tanh}(c_t)
\end{aligned}
$$

   where b is a bias parameter.

(b) Build an adaptive second order AR cell state with the help of an additional gate. The formulation is,

$$
\begin{aligned}
i_t &= \texttt{hard-sigmoid}(\boldsymbol{w}_i^{(x)}\boldsymbol{x}_t + \boldsymbol{w}_i^{(h)}\boldsymbol{h}_{t-1} + b_i) \\
o_t &= \texttt{hard-sigmoid}(\boldsymbol{w}_o^{(x)}\boldsymbol{x}_t + \boldsymbol{w}_o^{(h)}\boldsymbol{h}_{t-1} + b_o) \\
f_t &= \texttt{hard-sigmoid}(\boldsymbol{w}_f^{(x)}\boldsymbol{x}_t + \boldsymbol{w}_f^{(h)}\boldsymbol{h}_{t-1} + b_f) \\
g_t &= \texttt{hard-sigmoid}(\boldsymbol{w}_g^{(x)}\boldsymbol{x}_t + \boldsymbol{w}_g^{(h)}\boldsymbol{h}_{t-1} + b_g) \\
\tilde{c}_t &= \texttt{tanh}(\boldsymbol{w}_c^{(x)}\boldsymbol{x}_t + \boldsymbol{w}_c^{(h)}\boldsymbol{h}_{t-1} + b_c) \\
c_t &= b + g_t c_{t-2} + f_t c_{t-1} + i_t \tilde{c}_t \\
h_t &= o_t \texttt{tanh}(c_t)
\end{aligned}
$$

(c) The purpose of this exercise is to learn implementing new ideas for developing an RNN cell. An arbitrary formulation is presented above. How did this formulation work? Will a similar second-order AR formulation for GRUs work? Can you propose a formulation that can outperform LSTM and GRU cells?

Chapter 6

Convolutional Neural Networks

> "...We are suffering from a plethora of surmise, conjecture, and hypothesis. The difficulty is to detach the framework of fact—of absolute undeniable fact—from the embellishments..."
>
> – Sherlock Holmes, *Silver Blaze*.

6.1 Background

High-dimensional inputs are common. For example, images are high-dimensional in space; multivariate time series are high-dimensional in both space and time. In modeling such data, several deep learning (or machine learning) models get drowned in the excess confounding information.

Except for convolutional networks. They specialize in addressing this issue. These networks work by **filtering the essence** out of the excess.

Convolutional networks are built with simple constructs, viz. *convolution*, and *pooling*. These constructs were inspired by studies in neuroscience on human brain functioning. And, it is the simplicity of these

constructs that make convolutional networks robust, accurate, and efficient in most problems.

> "Convolutional networks are perhaps the greatest success story of biologically inspired artificial intelligence."
>
> – in Goodfellow, Bengio, and Courville 2016.

Convolutional networks were among the first working deep networks trained with back-propagation. They succeeded while other deep networks suffered from gradient issues due to their computational and statistical inefficiencies. And, both of these properties are attributed to the inherent simplistic constructs in convolutional networks.

This chapter describes the fundamentals of convolutional constructs, the theory behind them, and the approach to building a convolutional network.

It begins with a figurative illustration of the convolution concept in § 6.2 by describing a convolution as a simple filtration process. Thereafter, the unique properties of convolution, viz. parameter sharing, (use of) weak filters, and equivariance to translation, are discussed and illustrated.

The convolution equivariance makes a network sensitive to input variations. This hurts the network's efficiency. It is resolved by pooling discussed in § 6.4. Pooling regularizes the network and, also, allows modulating it between equivariance and invariance. These pooling attributes are explained in § 6.4.1 and § 6.4.2, respectively. The sections show that a pooled convolutional network is regularized and robust.

These sections used single-channel inputs for the illustrations. The convolution illustration is extended to multi-channel inputs such as a colored image in § 6.5.

After establishing the concepts, the mathematical kernel operations in convolution is explained in § 6.6. A few convolutional variants, viz. padding, stride, dilation, and 1×1 convolutions are then given in § 6.7.

Next, the elements of convolutional networks are described in § 6.8, e.g., input and output shapes, parameters, etc. Moreover, the choice between `Conv1D`, `Conv2D`, and `Conv3D` layers in TensorFlow is sometimes

confusing. The section also explains their interchangeability and compatibility with different input types.

At this point, the stage is set to start multivariate time series modeling with convolutional networks. The model construction and interpretations are in § 6.9. Within this section, it is also shown that convolutional networks are adept at learning long-term temporal dependencies in high-dimensional time series. A network in § 6.9.4 learned temporal patterns as wide as 8 hours compared to only up to 40 minutes with LSTMs in the previous chapter.

Furthermore, a flexible modeling of multivariate time series using a higher-order `Conv2D` layer is given in § 6.10.

After this section, the chapter discusses a few advanced topics in relation to pooling.

Pooling in a convolutional network summarizes a feature map. The summarization is done using *summary statistics*, e.g., average or maximum. § 6.11 delves into the theory of summary statistics intending to discover the statistics that are best for pooling. The section theoretically shows that maximum likelihood estimators (MLEs) make the best pooling statistic.

The theory also helps uncover the reason behind max-pool's superiority in § 6.12.1. Importantly, it provides several other strong pooling statistic choices in § 6.13. A few futuristic pooling methods such as adaptive and multivariate statistics are also discussed in § 6.14. A brief history of pooling is then laid in § 6.15.

Lastly, the chapter concludes with a few rules-of-thumb for constructing convolutional networks.

6.2 The Concept of Convolution

The concept of convolution is one of the simplest in deep learning. It is explained in this section with the help of an arbitrary image detection problem but the concept applies similarly to other problems.

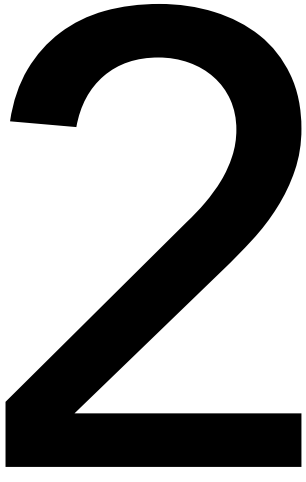
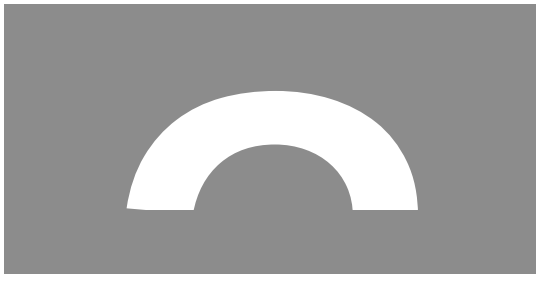
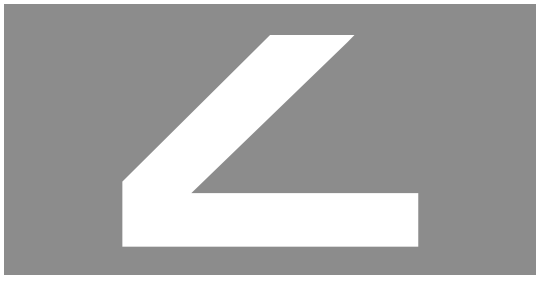


Figure 6.1. *An image of the letter "2." The problem is to detect the image as 2. Visually this is straightforward to a human but not to a machine. An approach for the detection is determining filters with distinctive shapes that match the letter and filtering the image (the convolution process) through them.*



(a) *Semi-circle filter.*



(b) *Angle filter.*

Figure 6.2. *Examples of filters for convolution to detect letters in an image. The filters—semi-circle and angle—together can detect letter 2 in an image.*

Problem. Consider an image of the letter "**2**" in Figure 6.1. The problem is to detect the image as "**2**." This is a straightforward task for a human. But not necessarily to a machine. The objective and challenge are to train a machine to perform the task. For which, one approach is *filtration.*

Filtration. There are several ways to perform the letter "**2**" detection. One of them is to learn distinctive *filters* and filter an image through them.

The filters should have shapes that distinguish "**2**." One such set of filters is a *semi-circle* and an *angle* shown in Figure 6.2a and 6.2b. A presence of these shapes in an image would indicate it has "**2**."

The presence of a shape in an image can be ascertained by a filtration-

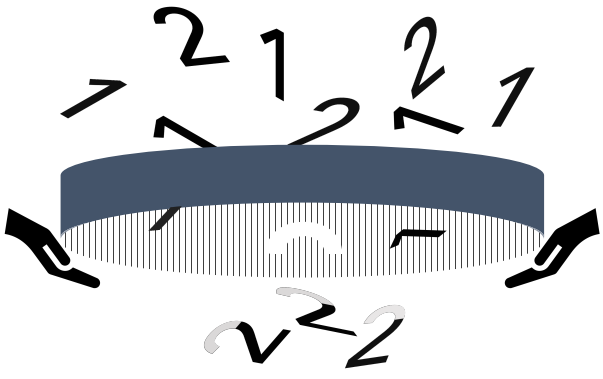
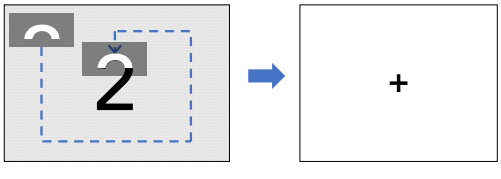
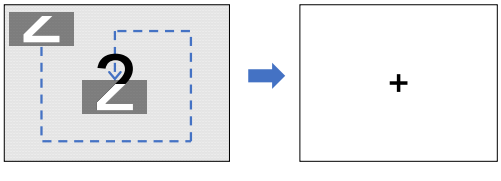


Figure 6.3. *An illustration of convolution as a filtration process through a sieve. The sieve has a semi-circular hole corresponding to a filter. Several letters are sifted through the sieve. The letters that have a semi-circular shape in them falls through it. Consequently, the **2**'s pass through the filter while the **1**'s do not.*



(a) *"**2**" image swept with semi-circle filter.*



(b) *"**2**" image swept with angle filter.*

Figure 6.4. *Formal implementation of the filtration process in convolution is sweeping the input as illustrated above. An image with "**2**" in it is swept with the semi-circle filter (top) and the angle filter (bottom). The filtration process of detecting the presence of the filter shapes is accomplished by sweeping the entire image to find a match.*

like process. Imagine there is a bucket of images of **1**'s and **2**'s. These images are sifted through a sieve which has a semi-circular hole as shown in Figure 6.3.

While (a part of) **2**'s could pass through the sieve, the **1**'s could not go through. The filtration through the sieve indicated the presence of semi-circle in the **2**'s. Similarly, another filtration through the *angle* filter could be performed to infer an image contains "**2**."

This filtration process is performed in **convolution**. A difference is that the filtration process in convolution appears like *sweeping* an image instead of sifting.

Sweeping. Formal implementation of filtration is sweeping an image with a filter.

Figure 6.4a and 6.4b illustrate the sweeping process in which the entire image is swept by the *semi-circle* and *angle* filters, respectively. In both the sweeps, the respective filter shapes were found and indicated with a + symbol in the output. These outputs enable detection of "**2**."

Convolution. Sweeping an input mathematically translates to a convolution operation.

In deep learning, convolution is performed with discrete kernels (details in § 6.6). The kernel corresponds to a filter and convolving an input with it indicates the presence/absence of the filter pattern.

A convolution operation is equivalent to sweeping an input with a filter in search of the filter's pattern.

Importantly, the pattern's location in the input is also returned. These locations are critical. Without their knowledge, an image with semi-circle and angle strewed anywhere in it will also be rendered as "**2**" (an illustration is in Figure 6.11 in § 6.4.2).

In sum, a convolution operation filters distinguishing patterns in

an input. The inputs are typically high-dimensional. The filtration by convolution displays the spirit of extracting the **essence** (patterns) from the **excess** (high-dimensional input).

The illustrations in this section might appear ordinary but remember for a machine it is different. The foremost challenge is that the filters are unknown. And, identifying appropriate filters by a machine is non-trivial. Convolutional networks provide a mechanism to **automatically** learn the filters (see § 6.9). Besides, convolution exhibits some special properties discussed next.

Earlier, filters were derived through feature engineering. Convolutional networks automated the filters learning.

6.3 Convolution Properties

A convolution process sweeps the entire input. Sweeping is done with a filter smaller than the input. This gives the property of *parameter sharing* to a convolutional layer in a deep learning network.

Moreover, the filters are small and, hence, called *weak*. This enables sparse interaction. Lastly, during the sweep the location of filter patterns in the input is also returned which brings the *equivariance* property to convolution.

This section explains the benefits, downsides, and intention behind these convolution properties.

6.3.1 Parameter Sharing

The property of parameter sharing is best understood by contrasting a dense layer with convolutional.

Suppose a dense layer is used for the image detection problem in the previous section. A dense layer would yield a filter of the **same shape and size** as the input image with letter "**2**" as shown in Figure 6.5.