Appendix H

1×1 Convolutional Network

§ 6.7.4 explains 1×1 convolution layers and their purpose. An illustrative convolutional network with 1×1 convolutional layer is shown in Listing H.

A regular convolution layer has a kernel size larger than 1 to learn spatial features. Differently, a 1×1 convolution layer has a kernel size equal to 1. Its primary purpose is to reduce the channels for network dimension reduction. Sometimes they are also used to learn features from the information in the channels. For this, multiple 1×1 layers are stacked in parallel.

The illustrative example in Listing H has a single 1×1 convolutional layer. As shown in Figure H.1, the layer resulted in a reduction of the channels from 64 to 32. The network is built and trained on temporalized data with `lookback=240`. The results are shown in Figure H.2a-H.2c.

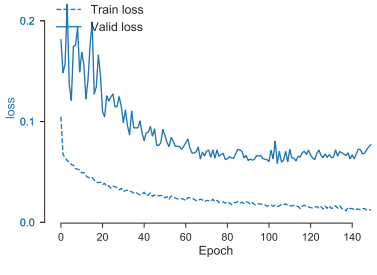
```
## 1x1 convolutional network

model = Sequential()
model.add(Input(shape=(TIMESTEPS,
                       N_FEATURES),
                name='input'))
model.add(Conv1D(filters=64,
                 kernel_size=4,
                 activation='relu',
                 name='Convlayer'))
```

```
model.add(Dropout(rate=0.5,
                  name='dropout'))
model.add(Conv1D(filters=32,
                 kernel_size=1,
                 activation='relu',
                 name='Conv1x1'))
model.add(MaxPool1D(pool_size=4,
                    name='maxpooling'))
model.add(Flatten(name='flatten'))
model.add(Dense(units=16,
                activation='relu',
                name='dense'))
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=150,
                    validation_data=(X_valid_scaled,
                                     y_valid),
                    verbose=0).history
```

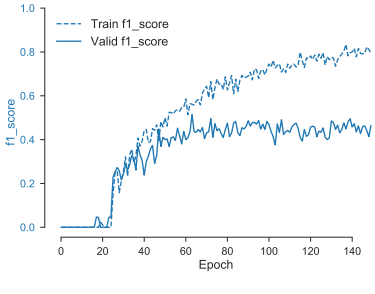
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Convlayer (Conv1D)           (None, 237, 64)           17728
_________________________________________________________________
dropout (Dropout)            (None, 237, 64)           0
_________________________________________________________________
Conv1x1 (Conv1D)             (None, 237, 32)           2080
_________________________________________________________________
maxpooling (MaxPooling1D)    (None, 59, 32)            0
_________________________________________________________________
flatten (Flatten)            (None, 1888)              0
_________________________________________________________________
dense (Dense)                (None, 16)                30224
_________________________________________________________________
output (Dense)               (None, 1)                 17
=================================================================
Total params: 50,049
Trainable params: 50,049
Non-trainable params: 0
_________________________________________________________________
```

The Conv 1x1 layer reduces the number of channels. The output of the previous layer had 64 channels (equal to the number of filters). The Conv 1x1 layer reduced it to half.

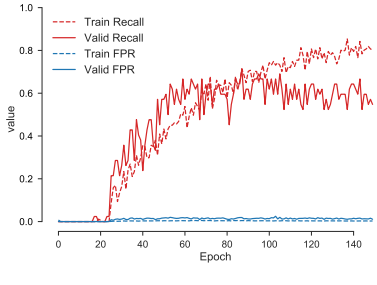
Figure H.1. *Summary of a* 1×1 *convolutional network.*



(a) *Loss.*



(b) *F1-score.*



(c) *Recall and FPR.*

Figure H.2. 1×1 *convolutional network results.*

Appendix I

CNN: Visualization for Interpretation

A benefit of convolutional networks is that they have filters and feature representations which can be visualized to understand the spatio-temporal patterns and dependencies. These visuals help diagnose a model.

In this appendix, illustrations for filters and feature representation visualization are presented. The visuals are for the network constructed in Appendix H.

Filter Visualization

We can look at the layers in a `model` by printing `model.layers` as shown in Listing I.1. Every convolutional layer makes a set of filters. A layer weights (the filters) can be taken using the layer index or the name.

For example, the convolutional layer weights in the model in Listing H can be fetched as `model.layers[0].get_weights()` because it is the first layer which has index 0, or using a named call as `model.get_layer('Convlayer').get_weights()` where `Convlayer` is the user-defined name for the layer.

Listing I.1. CNN model layers.

```
model.layers
# [<tensorflow.python.keras.layers.convolutional.
    Conv1D at 0x149fd59b0>,
# <tensorflow.python.keras.layers.core.Dropout at 0
    x149fd5780>,
# <tensorflow.python.keras.layers.convolutional.
    Conv1D at 0x14b123358>,
# <tensorflow.python.keras.layers.pooling.
    MaxPooling1D at 0x14d77d048>,
# <tensorflow.python.keras.layers.core.Flatten at 0
    x149d86198>,
# <tensorflow.python.keras.layers.core.Dense at 0
    x14cd69c50>,
# <tensorflow.python.keras.layers.core.Dense at 0
    x14c990a90>]
```

Listing I.2 fetches the convolutional filters and scales them in (0,1) for visualization. They are then plotted in Figure I.1. The plots have a clearer interpretation of image problems. In such problems, the filters have shapes that correspond to certain patterns in the objects.

The filter visuals here can be interpreted differently. Each filter is of shape (4, 69) corresponding to the kernel size and the input features, respectively. The plot shows which feature is active in a filter.

Besides, there are a total of 64 filters in the convolutional layer (see Listing H). Out of them, 16 filters are shown in the figure.

Listing I.2. CNN filter plotting.

```
## Plot filters

# retrieve weights from the first convolutional
    layer
filters, biases = model.layers[0].get_weights()
print(model.layers[0].name, filters.shape)
# Convlayer (4, 69, 64)

# normalize filter values to 0-1 so we can visualize
    them
f_min, f_max = filters.min(), filters.max()
filters = (filters - f_min) / (f_max - f_min)

```

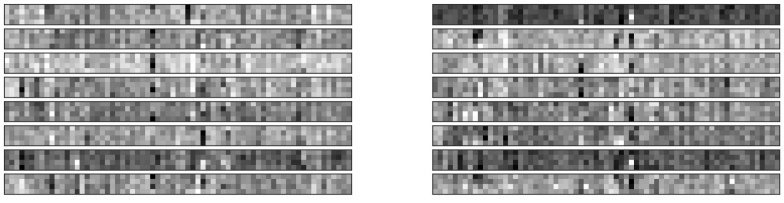


Figure I.1. *Convolutional layer filters visualization.*

```
from matplotlib import pyplot
from matplotlib.pyplot import figure
figure(num=None, figsize=(20,10), dpi=160, facecolor
    ='w', edgecolor='k')
# plot first 16 filters
n_filters, ix = 16, 1
for i in range(n_filters):
    # get the filter
    f = filters[:, :, i]
    # plot each channel separately

    # specify subplot and turn of axis
    ax = pyplot.subplot(n_filters, 2, ix)
    ax.set_xticks([])
    ax.set_yticks([])
    # plot filter channel in grayscale
    pyplot.imshow(f[:, :], cmap='gray')
    ix += 1
# show the figure
pyplot.show()
```

Feature Representation Visuals

Visualizing the feature representations helps to learn the model's reactions to positive and negative samples.

In Listing I.3, a few true positive and true negative samples are taken based on the model inferences. Out of all the true positives, the ones with high probabilities are taken for better diagnosis.

The feature representation outputted by the first convolutional layer in Listing H is taken here. The steps for fetching the feature representations (map) are shown in Listing I.3.

As was shown in the model summary in Figure H.1, the output of the convolution layer is a feature representation of shape 237×64.

We visualize these 237×64 outputs for the true positive and true negative samples in Figure I.2a-I.2b.

Listing I.3. Convolutional network feature representation plotting.

```
## Plot feature map
# Take out a part of the model to fetch the feature mapping
# We are taking the feature mapping from the first Convolutional layer
feature_mapping = Model(inputs=model.inputs, outputs=model.layers[0].output)

prediction_valid = model.predict(X_valid_scaled).squeeze()

top_true_positives = np.where(
    np.logical_and(prediction_valid > 0.78, y_valid == 1))[0]

top_true_negatives = np.where(
    np.logical_and(prediction_valid < 1e-10, y_valid == 0))[0]

# Plotting
from matplotlib import pyplot
from matplotlib.pyplot import figure
figure(num=None, figsize=(4, 4), dpi=160, facecolor='w')


def plot_feature_maps(top_predictions):

    n_feature_maps, ix = 10, 1

    for i in range(n_feature_maps):
```

```
25
26          samplex = X_valid_scaled[top_predictions[i],
                :, :]
27          samplex = samplex.reshape((1, samplex.shape
                [0], samplex.shape[1]))
28
29          feature_map = feature_mapping.predict(
                samplex).squeeze()
30
31          ax = pyplot.subplot(np.round(n_feature_maps
                / 2), 2, ix)
32          ax.set_xticks([])
33          ax.set_yticks([])
34
35          # plot filter channel in grayscale
36          pyplot.imshow(np.transpose(1 - (feature_map
                - feature_map.min()) /
37                                         (feature_map.max
                                               () -
                                               feature_map.
                                               min())),
38                        cmap='viridis')
39          ix += 1
40
41      # show the figure
42      pyplot.show()
43
44  plot_feature_maps(top_true_positives)
45
46  plot_feature_maps(top_true_negatives)
```

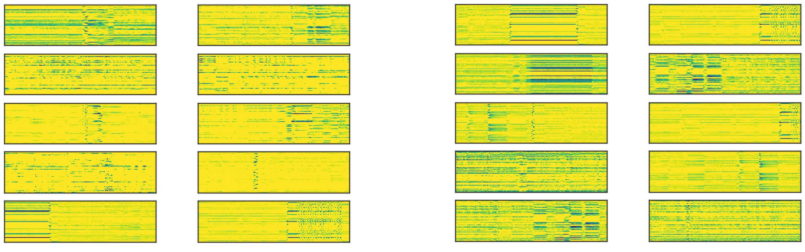
In the figure, the yellow indicates the feature is active while the opposite for green. At a high level, it can be interpreted that most of the features are activated for true positives but not for true negatives. Meaning, the activation of these features distinguishes a positive (sheet break) from a normal process (no sheet break).

However, the true positive feature map on the top-left in Figure I.2a does not follow this interpretation. To further diagnose, subsequent layers should be visualized.

These visualizations help diagnose the model. The diagnosis can help

in model improvement, new model development, or root cause analysis.

A different set of samples can be chosen, e.g., false positives, or false negatives, to diagnose the model to identify what happens when it is unable to correctly predict.



(a) *True positives.* (b) *True negatives.*

Figure I.2. *Convolutional network feature representation plotting.*

Appendix J

Multiple (Maximum and Range) Pooling Statistics in a Convolution Network

The concept of using summary statistics for pooling described in Chapter 6 opened possibilities for new ways of pooling. One of them is using ancillary statistics in parallel with a sufficient statistic.

In this appendix, a network is constructed in Listing J with *maximum* pooling (a sufficient statistic) and *range* pooling (an ancillary statistic) in parallel.

The network structure is shown in Figure J.1. As shown here, the ReLU (nonlinear) activation is added after the pooling. This is essential as described in § 6.12.2. Otherwise, if activation is added between convolutional and pooling layers (following the tradition), the *range* statistic becomes the same as the *maximum* whenever the feature representations have any negative value.

```
1 ## Multiple-pooling layer convolutional network
2 x = Input(shape=(TIMESTEPS, N_FEATURES))
3
4 conv = Conv1D(filters=16,
5               kernel_size=4,
6               activation='linear',
```

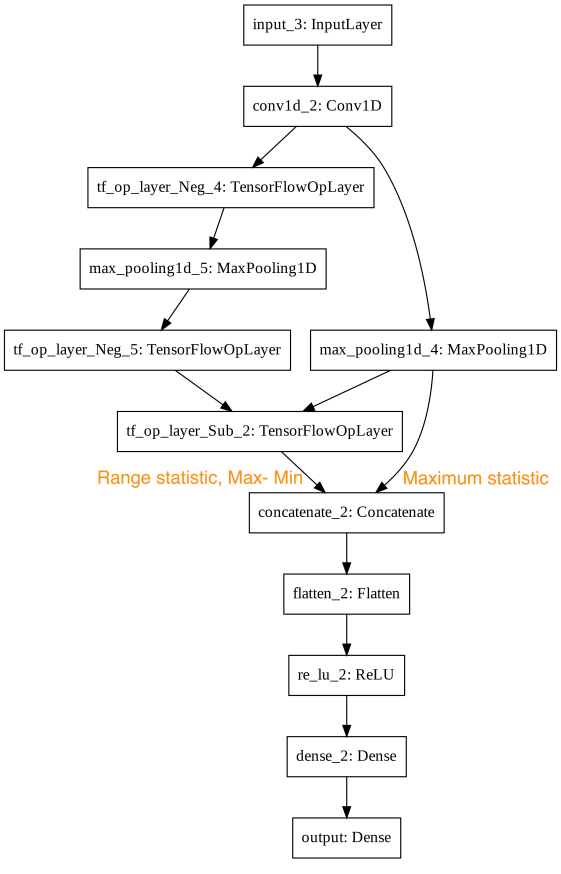


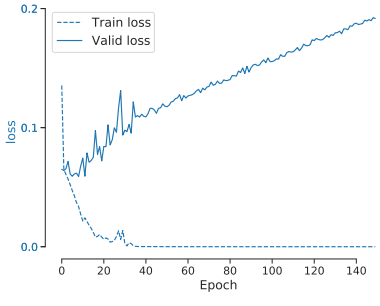
Figure J.1. *A convolutional network with maximum and range pooling placed in parallel.*

```
                padding='valid')(x)

# left operations
max_statistic = MaxPool1D(pool_size=4,
                          padding='valid')(conv)

# right operations
# 1. negative of feature map
range_statistic = tf.math.negative(conv)
# 2. apply maxpool to get the min statistics
range_statistic = MaxPool1D(pool_size=4,
                        padding='valid')(
                           range_statistic)
# 3. negative of negative in step (1) to revert to
    original
range_statistic = tf.math.negative(range_statistic)
# 4. subtract with max_statistic to get the
# range statistic max(x) - min(x)
range_statistic = tf.math.subtract(max_statistic,
                                   range_statistic)

# Concatenate the pool
concatted =
tf.keras.layers.Concatenate()([max_statistic,
                                 range_statistic])

features = Flatten()(concatted)

features = ReLU()(features)

# 128 nodes for lookback = 20 or 40.
dense = Dense(units=256,
              activation='relu')(features)

predictions = Dense(units=1,
                    activation='sigmoid',
                    name='output')(dense)

model = Model(inputs=x,
              outputs=predictions)

# Plot the network structure
```

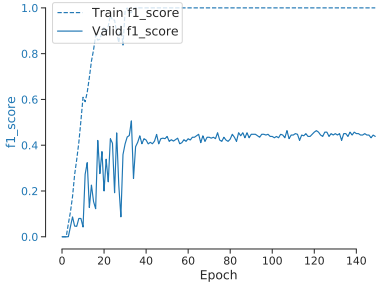
```
47 tf.keras.utils.plot_model(model,
48                            show_shapes=False,
49                            dpi=600)
50
51 # Train the model
52 model.compile(optimizer='adam',
53               loss='binary_crossentropy',
54               metrics=[
55                   'accuracy',
56                   tf.keras.metrics.Recall(),
57                   pm.F1Score(),
58                   pm.FalsePositiveRate()
59               ])
60 history = model.fit(x=X_train_scaled,
61                     y=y_train,
62                     batch_size=128,
63                     epochs=150,
64                     validation_data=(X_valid_scaled,
65                                      y_valid),
66                     verbose=1).history
```

The network construction in Listing J shows a work-around with existing TensorFlow functions to get the range statistic. A custom range-pooling layer can also be built.

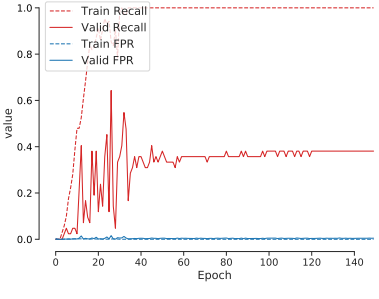
The results from training the network on a temporalized data with `lookback=240` shown in Figure J.2a-J.2c have a more stable accuracy performance than every other model. The stability could be attributed to the informative features provided by the parallel pooling.



(a) *Loss.*



(b) *F1-score.*



(c) *Recall and FPR.*

Figure J.2. *Convolutional network with range and maximum pooling results.*

Appendix K

Convolutional Autoencoder-Classifier

One of the applications of autoencoders is learning encodings that are used to train a classifier. In this appendix, an illustrative example is presented with the steps.

Sparse Convolutional Autoencoder

A sparse autoencoder is preferred for use in a classifier. The sparsity is applied on the encodings as shown in Listing K.1. The implementation approach here is different from the implementations in Chapter 7 to show another way to construct an autoencoder.

Listing K.1. Sparse convolutional autoencoder.

```
# Sparse CNN Autoencoder

inputs = Input(shape=(TIMESTEPS,
                      N_FEATURES),
               name='encoder_input')

def encoder(inp):
  '''
  Encoder.

```

```
    Input
    inp    A tensor of input data.

    Process
    Extract the essential features of the input as
    its encodings by filtering it through convolutional
    layer(s). Pooling can also be used to further
    summarize the features.

    A linearly activated dense layer is added as the
    final layer in encoding to perform any affine
    transformation required. The dense layer is not
    for any feature extraction. Instead, it is only
    to make the encoding and decoding connections
    simpler for training.

    Output
    encoding  A tensor of encodings.
    '''

    # Multiple (conv, pool) blocks can be added here
    conv1 = Conv1D(filters=N_FEATURES,
                   kernel_size=4,
                   activation='relu',
                   padding='same',
                   name='encoder-conv1')(inp)
    pool1 = MaxPool1D(pool_size=4,
                      strides=1,
                      padding='same',
                      name='encoder-pool1')(conv1)

    # The last layer in encoding
    encoding = Dense(units=N_FEATURES,
                     activation='linear',
                     activity_regularizer=
                      tf.keras.regularizers.L1(l1
                          =0.01),
                     name='encoder-dense1')(pool1)

    return encoding

```

```
51 def decoder(encoding):
52   '''
53   Decoder.
54
55   Input
56   encoding    The encoded data.
57
58   Process
59   The decoding process requires a transposed
60   convolutional layer, a.k.a. a deconvolution
61   layer. Decoding must not be done with a
62   regular convolutional layer. A regular conv
63   layer is meant to extract a downsampled
64   feature map. Decoding, on the other hand,
65   is reconstruction of the original data
66   from the downsampled feature map. A
67   regular convolutional layer would try to
68   extract further higher level features from
69   the encodings instead of a reconstruction.
70
71   For a similar reason, pooling must not be
72   used in a decoder. A pooling operation is
73   for summarizing a data into a few summary
74   statistics which is useful in tasks such
75   as classification. The purpose of
76   decoding is the opposite, i.e., reconstruct
77   the original data from the summarizations.
78   Adding pooling in a decoder makes it lose
79   the variations in the data and,
80   hence, a poor reconstruction.
81
82   If the purpose is only reconstruction, a
83   linear activation should be used in
84   decoding. A nonlinear activation is useful
85   for predictive features but not for
86   reconstruction.
87
88   Batch normalization helps a decoder by
89   preventing the reconstructions
90   from exploding.
91
92   Output
```

```
93    decoding    The decoded data.
94
95    '''
96
97    convT1 = Conv1DTranspose(filters=N_FEATURES,
98                             kernel_size=4,
99                             activation='linear',
100                            padding='same')(encoding)
101
102   decoding = BatchNormalization()(convT1)
103
104   decoding = Dense(units=N_FEATURES,
105                    activation='linear')(decoding)
106
107   return decoding
108
109 autoencoder = Model(inputs=inputs,
110                     outputs=decoder(encoder(inputs))
                            )
111
112 autoencoder.summary()
113 autoencoder.compile(loss='mean_squared_error',
        optimizer = 'adam')
114
115 history =
116     autoencoder.fit(x=X_train_y0_scaled,
117                     y=X_train_y0_scaled,
118                     epochs=100,
119                     batch_size=128,
120                     validation_data=
121                         (X_valid_y0_scaled,
122                          X_valid_y0_scaled),
123                     verbose=1).history
```

Convolutional Classifier Initialized with Encoder

A classifier can be (potentially) enhanced with an autoencoder. Listing K.2 constructs a feed-forward convolutional classifier with an encoder attached to it.

Listing K.2. Convolutional feed-forward network initialized with autoencoder.

```
# CNN Classifier initialized with Encoder

def fully_connected(encoding):
  conv1 = Conv1D(filters=16,
                 kernel_size=4,
                 activation='relu',
                 padding='valid',
                 name='fc-conv1')(encoding)
  pool1 = MaxPool1D(pool_size=4,
                    padding='valid',
                    name='fc-pool1')(conv1)
  flat1 = Flatten()(pool1)

  den = Dense(units=16,
              activation='relu')(flat1)

  output = Dense(units=1,
                 activation='sigmoid',
                 name='output')(den)

  return(output)

encoding = encoder(inputs)
classifier =
    Model(inputs=inputs,
          outputs=fully_connected(
              encoding=encoder(inputs)))
```

The encoder can be used to have either a,

- **Pre-trained classifier**. A trained encoder can be used as a part of a feed-forward classifier network. Or,
- **Encoded features as input**. The features produced by an encoder used as input to a classifier.

Corresponding to the two approaches, an argument `retrain_encoding` is defined in Listing K.3.

The argument when set to `False` results in the classifier using the **encoded features as input**. This is achieved by making the layers in

the encoder section of the model as non-trainable in line 13 in the listing (Listing K.3). This is also shown in Figure K.1.

The argument, `retrain_encoding`, when set to `True` uses the encoding weights to initialize the model and retrain them while learning the classifier.

Listing K.3. Training an autoencoder-classifier.

```
1  # Classifier layer initialized with encoder
2  retrain_encoding = False
3
4  for classifier_layer in classifier.layers:
5    for autoencoder_layer in autoencoder.layers:
6      if classifier_layer.name == autoencoder_layer.
           name:
7        # Set the weights of classifier same as the
8        # corresponding autoencoder (encoder) layer
9        classifier_layer.set_weights(
10           autoencoder_layer.get_weights())
11
12       if retrain_encoding == False:
13         classifier_layer.trainable = False
14       print(classifier_layer.name +
15             ' in classifier set to ' +
16             autoencoder_layer.name +
17             ' in the encoder' +
18             'is trainable: ' +
19             str(classifier_layer.trainable))
20
21 classifier.summary()
22 classifier.compile(optimizer='adam',
23               loss='binary_crossentropy',
24               metrics=[
25                   'accuracy',
26                   tf.keras.metrics.Recall(),
27                   pm.F1Score(),
28                   pm.FalsePositiveRate()
29               ])
30
31 history = classifier.fit(x=X_train_scaled,
32                     y=y_train,
33                     batch_size=128,
```

Layer (type)	Output Shape	Param #
encoder_input (InputLayer)	[(None, 20, 69)]	0
encoder-conv1 (Conv1D)	(None, 20, 69)	19113
encoder-pool1 (MaxPooling1D)	(None, 20, 69)	0
encoder-dense1 (Dense)	(None, 20, 69)	4830
fc-conv1 (Conv1D)	(None, 17, 16)	4432
fc-pool1 (MaxPooling1D)	(None, 4, 16)	0
flatten (Flatten)	(None, 64)	0
dense_1 (Dense)	(None, 16)	1040
output (Dense)	(None, 1)	17

Total params: 29,432
Trainable params: 5,489
Non-trainable params: 23,943

Figure K.1. *Model summary of convolutional autoencoder-classifier using the encodings as classifier input.*

```
34                          epochs=100,
35                          validation_data=
36                              (X_valid_scaled,
37                               y_valid),
38                          verbose=1).history
```

Appendix L

Oversampling

Oversampling techniques artificially increase the minority positive class samples to balance the data set. One of the most fundamental oversampling techniques is randomly selecting samples from the minority class with replacement. The random selection process is repeated multiple times to get as many minority samples as required for data balancing.

However, in an extremely unbalanced data, creating a large number of duplicates for the minority class may yield a biased model.

A more methodical approach called Synthetic Minority Over-sampling Technique (SMOTE) can address this.

SMOTE

In this approach, data is synthesized by randomly interpolating new points between the available (real) minority samples.

This procedure is illustrated in Figure L.1. In the figure, the +'s are the "rare" minority class and o's are the majority. SMOTE synthesizes a new minority sample between the existing samples. These interpolations work as follows,

- the interpolated point is drawn randomly from anywhere on the line (in the two-dimensional visual in Figure L.1, otherwise, a hyper-plane in general) that connects two samples denoted as $\mathbf{x}_1$

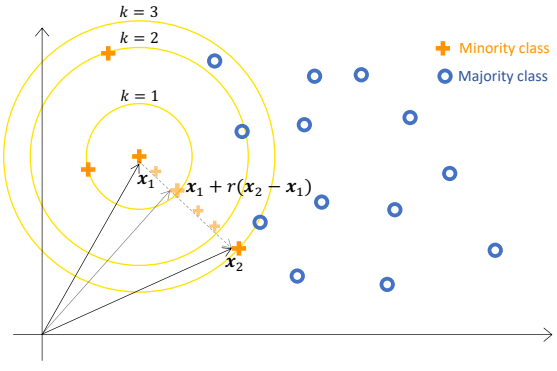


Figure L.1. *Illustration of SMOTE oversampling. SMOTE synthesizes a new sample (shown in translucent orange +) by interpolating between two samples.*

and $\mathbf{x}_2$.

- For this, a random number between 0 and 1 is generated. The new point is then synthesized as, $\mathbf{x}_1 + r(\mathbf{x}_2 - \mathbf{x}_1)$, shown as a translucent + in the figure.

- Note that, any interpolated sample between $\mathbf{x}_1$ and $\mathbf{x}_2$ will lie on the line connecting them (shown as small translucent +'s in the figure).

SMOTE is available in Python as `imblearn.over_sampling.SMOTE()` [1]. Its primary arguments are,

- `k_neighbors`. Denoted as k in Figure L.1, it is the number of nearest neighbors SMOTE will use to synthesize new samples. By default, $k = 5$. A lower k will have lesser noise but also less robust, and vice-versa for a higher k.

- `random_state` is used to control the randomization of the algorithm. It is useful to set this to reproduce the sampling.

[1] `https://imbalanced-learn.readthedocs.io/en/stable/generated/imblearn.over_sampling.SMOTE.html`

Listing L.1. SMOTE Oversampling.

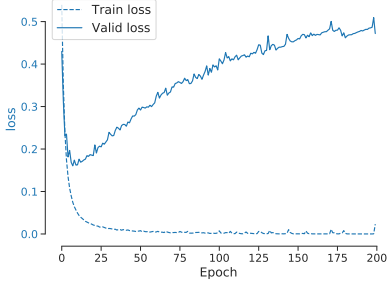
```
smote = SMOTE(random_state=212, k_neighbors=1)
X_train_scaled_resampled, y_train_resampled =
  smote.fit_resample(X_train_scaled, y_train)
print('Resampled dataset shape %s' %
  Counter(y_train_resampled))
```

Listing L.1 shows the `SMOTE` declaration. Here, `k_neighbors=1` because the data is noisy and a larger k will add more noise. However, a small k comes at the cost of making the model biased, and therefore, potentially poorer inferencing accuracy.

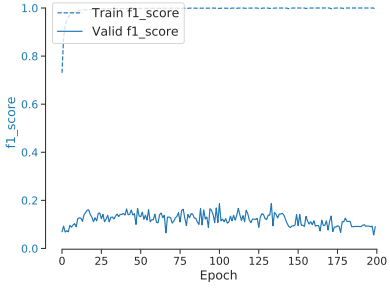
A model is built and trained with the oversampled data in Listing L.2. The results are shown in Figure L.2a-L.2c.

Listing L.2. MLP Model with SMOTE Oversampled Training Data.

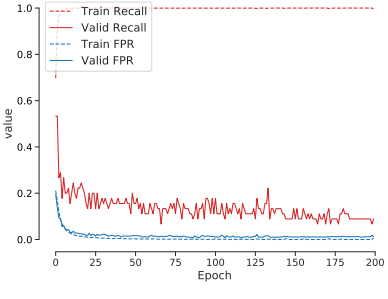
```
model = Sequential()
model.add(Input(shape=(N_FEATURES, )))
model.add(Dense(32, activation='relu'))
model.add(Dense(16, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

model.summary()

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

history = model.fit(x=X_train_scaled_resampled,
                    y=y_train_resampled,
                    batch_size=128,
                    epochs=200,
                    validation_data=(X_valid_scaled,
                      y_valid),
```



(a) *Loss.*



(b) *F1-score.*



(c) *Recall and FPR.*

Figure L.2. *MLP with SMOTE oversampling model results.*