Figure 6.5. *An illustration of a filter in a dense layer to detect letter "2." The filter size is the same as that of the input. Consequently, the parameter space is significantly large. This is also referred to as a strong filter because it can alone detect the letter "2." Although the filter is strong, the excess amount of parameters in it make a dense layer statistically inefficient. Due to statistical inefficiency, a dense layer would require a large number of samples to automatically learn a filter.*

Compare the sizes of dense layer filter in Figure 6.5 with either of the convolutional filters, semi-circle, and angle, in Figure 6.2a and 6.2b. The latter are clearly smaller.

The convolutional filter is **smaller** than the input. But to cover the entire input, it sweeps through it from top to bottom and left to right. This is called *parameter sharing*.

A filter is a kernel that is made of some parameters. When the same filter is swept on different parts of the input, it is referred to as the parameters are shared. This is in contrast to a dense layer filter which is as big as the input and, hence, does not share parameters.

What is the benefit of parameter sharing? Parameter sharing makes a convolutional network statistically efficient. Statistical efficiency is the ability to learn the model parameters with as few samples as possible.

Due to parameter sharing, a convolutional layer can work with small-sized filters—smaller than the input. The small filters have fewer parameters compared to an otherwise dense layer. For instance, if the input

Figure 6.6. *An example of a typical-sized* 2400×1600 *image. A convolutional filter to detect a pair of eyes is shown with an orange box. The filter is significantly smaller than the image. Therefore, it is termed as a weak filter. The filter is essentially a set of parameters. This filter sweeps—convolves—the entire image in search of a pair of eyes. This is called parameter sharing. Besides, the convolution process also yields the location of the pair of eyes. This ability to detect positional information is called equivariance. –Photo by Elizaveta Dushechkina on Unsplash.*

image has $m \times n$ pixels, a filter in the dense layer (the weight matrix) is also $m \times n$ but a convolutional filter will be $m' \times n'$, where $m' << m$ and $n' << n$.

The benefit is evident if m and n are large. For instance, a typical-sized image shown in Figure 6.6 is 2400×1600 pixels. In this image, a convolutional filter to detect eyes is 20×60 with $20 * 60 = 1,200$ parameters. This size is significantly smaller than a dense layer weight parameter which is equal to the input size of $2400 * 1600 = 3,840,000$.

Since there are fewer parameters, a convolutional network can learn them with a relatively smaller number of samples. Due to this property, convolutional networks are sometimes also referred to as regularized versions of multilayer perceptrons.

Parameter sharing makes convolutional network statistically efficient, i.e., the network parameters can be learned with fewer samples.

6.3.2 Weak Filters

The statistical efficiency is boosted due to the filter re-usability brought by *weak filters*.

Convolutional layer deploy filters smaller than the input. These are referred to as *weak filters*. The name is because one weak filter by itself is not sufficient. One needs the help of other weak filters for inferencing. For instance, the illustrative example of the letter "**2**" detection required two (weak) filters: a semi-circle and an angle.

The use of weak filters brings an important attribute of *filter reuse* to convolutional nets. **Filter re-usability** is the ability to use a filter for detecting a pattern in multiple objects. For example, the semi-circle filter can be reused to distinguish "**0**," "**2**," "**3**," "**6**," "**8**," and "**9**" from other letters as shown in Figure 6.7. In a letters detection problem, where the letters can be in 0-9, the filter reuse becomes extremely beneficial. Instead of learning strong filters for every letter, a set of weak filters collaborate to detect multiple objects.

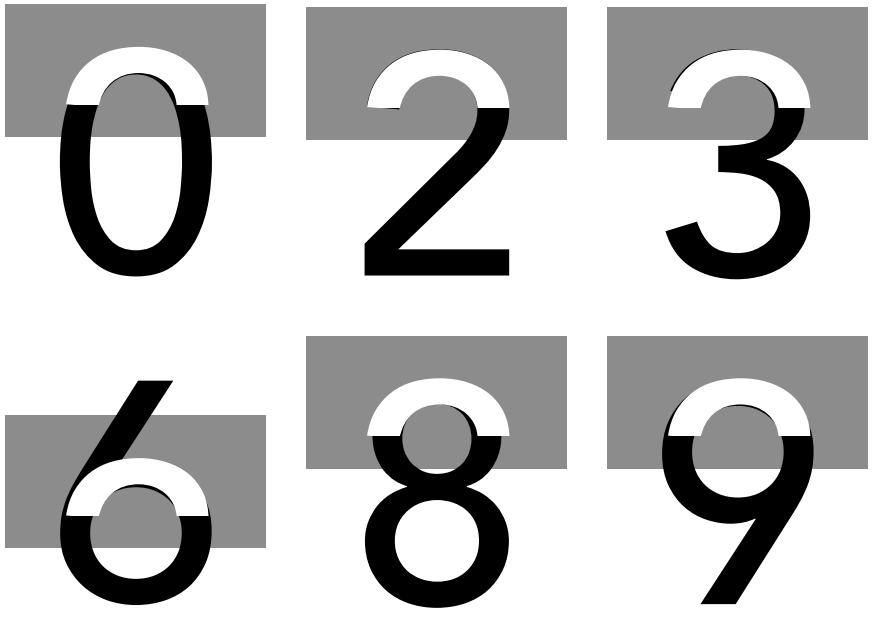


Figure 6.7. *A weak filter semi-circle can detect a distinctive pattern in multiple letters, viz. "**0**," "**2**," "**3**," "**6**," "**8**," and "**9**." In a letters detection problem where the letters can be in 0-9, the semi-circle can separate six of the ten letters. Additional filters, for example, an inverted semi-circle to further separate "**0**," "**3**," "**6**," and "**8**," can be employed. These filters collaborate to infer the letters accurately with fewer parameters to learn. These weak filters and the possibility of their reuse make convolutional networks statistically efficient.*

The benefit of weak filters is that they are reused. A set of weak filters collaborate to detect multiple objects.

A strong filter is capable of detecting an object just by itself. However, it comes at the cost of excessive parameters. Moreover, a strong filter by definition cannot be re-used to detect any other object. An example of a strong filter for detecting "**2**" is in Figure 6.5. This filter cannot be used for any other object. Besides, strong filters are sensitive to distortions (noise) in samples. For example, if the letter "**2**" is written slightly differently, the filter will not work.

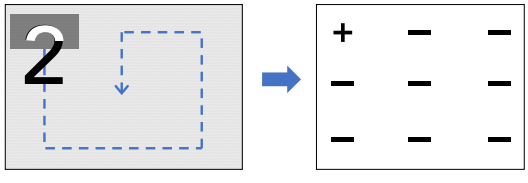
Weak filters, on the other hand, are small in size, reusable, and robust to noise. Due to this, they can be learned with fewer samples. Moreover, due to the reusability and robustness of a weak filter, it is learned from samples of multiple objects. For instance, the semi-circle filter will be automatically learned in a convolutional network using samples of "**0**," "**2**," "**3**," "**6**," "**8**," and "**9**." As a result, the presence of weak filters boosts the statistical efficiency of convolutional networks.

A collection of weak filters make a convolutional network strong. Weak filters are small filters with fewer parameters and also allows filter re-usability. These attributes boost the networks' statistical efficiency.

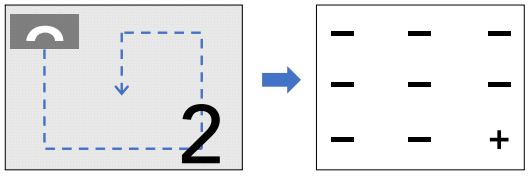
6.3.3 Equivariance to Translation

Convolution exhibits the property of *equivariance to translation*. This means if there is any variation in the input, the output changes in the same way. Due to this property, convolution also preserves the positional information of objects in the input. This is explained below.

As mentioned in the previous § 6.2, convolution is a process of sweeping an input with a filter. At every (sweep) stride, the filter processes the input at the location and emits an output. This is illustrated in



(a) *Letter "2" on the top-left of image.*



(b) *Letter "2" on the bottom-right of image.*

Figure 6.8. *Illustration of the equivariance property of convolution. Convolution is equivariant to translation, which implies if the input changes, the output changes in the same way. For example, "2" is at different positions: top-left (top) and bottom-right (bottom). The convolution output changed at the same locations showing its equivariance.*

Figure 6.8a and 6.8b.

In the figures, the letter "**2**" is placed at the top-left and the bottom-right, respectively. The images are convolved (swept) with the semi-circle filter. The convolution output is numeric but for simplicity, the output is represented as +, if there is a match with the filter, and −, otherwise.

Figure 6.8a and 6.8b show that based on the location of the letter the convolution output changes. In Figure 6.8a, the output has a + at the top-left and the rest are −. But as the position of "**2**" changed to the bottom-right in Figure 6.8b the convolution output also changed to + at the same bottom-right location.

This shows that convolution maps the object location variation in the input. In effect, convolution preserves the information of the object's location.

Equivariance to translation is a property due to which if there is a variation in the input, the convolution output changes in the same way.

Is location preservation important? Not always. In several problems, the objective is to determine the presence or absence of an object or pattern in the input. Its location is immaterial. In such problems, preserving the location makes the model over-representative. An over-representative model has more features than needed and, consequently, excess parameters. This hurts the statistical efficiency. To resolve this, *pooling* is used in conjunction with a convolutional layer.

Due to the equivariance property, the convolutional layer preserves the location information of an object or pattern in an input. This makes the network over-representative that counteracts its statistical efficiency. Adding a pooling layer addresses the issue.

6.4 Pooling

Pooling brings *invariance* to a convolutional network. If a function is invariant, its output is unaffected by any translational change in the input.

A pooling operation draws a summary statistic from the convolution output. This replaces the over-representative convolution output with a single or a few summary statistics. Pooling, therefore, further regularizes the network and maintains its statistical efficiency (illustrated in § 6.4.1).

However, just like equivariance, an invariance to translation is also sometimes counterproductive. Invariance makes the network blind to the location of patterns in the input. This sometimes leads to a network confuse the original input with its distortions (illustrated in Figure 6.10 and 6.11 in § 6.4.2).

Pooling provides a lever to modulate the network between equivariance and invariance. Somewhere between the two is usually optimal.

Thereby, a pooling layer complements a convolutional layer. Moreover, pooling does not have trainable parameters[1]. And, therefore, it does not add a computational overhead.

A pooling layer provides a mechanism to regularize a convolutional network without adding computational overhead.

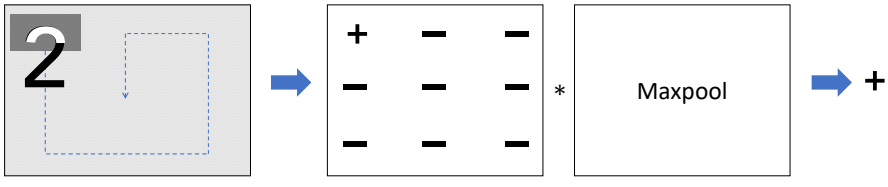
In the following, the first invariance as a means for regularization is explained. It is followed by showing pooling as a tool for modulating a network between equivariance and invariance.

6.4.1 Regularization via Invariance

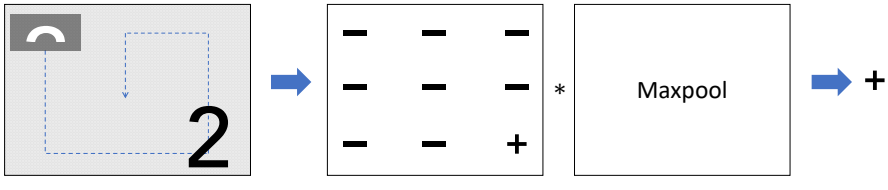
Invariance is the opposite of equivariance. If a function is invariant, its output is unaffected by any translational change in the input.

Pooling brings *invariance* to a convolutional network. A pooling

[1]Unknown parameters to estimate.



(a) *Letter "2" on the top-left of image.*



(b) *Letter "2" on the bottom-right of image.*

Figure 6.9. *Illustration of invariance property of pooling. This illustration shows an extreme level of pooling, called global pooling. In this, the network becomes absolutely invariant. Meaning, the location information of the object is not preserved. Instead, the network focuses only on determining the presence or absence of an object in an input. For example, although the letter "2" is present in top-left and bottom-right in the top and bottom figures, the pooling output remains the same. Here* `MaxPool` *is used for illustration but the pooling behavior stays the same for any other pooling. This indifference of the output to the translations in the input is called invariance.*

operation summarizes the convolutional output into a statistic(s), called a *summary statistic*. For example, the maximum summary statistic is returned in `MaxPool`. In doing so, the granular information about an object's location is lost.

This causes the network to become invariant to the location of an object in the input. The phenomenon is illustrated in Figure 6.9a and 6.9b. This illustration is a continuation of the one in Figure 6.8 in § 6.3.3 by applying `MaxPool` after the convolution.

As shown in the figures, the maxpool operand takes in the output from convolution and emits the maximum value. Consequently, despite the letter "**2**" being at the top-left or bottom-right in Figure 6.9a and 6.9b, respectively, the output from pooling is the same +.

In effect, pooling regularized the network by reducing the spatial size of the representation (the feature map). This reduces the parameters and, thereby, improves the statistical and computational efficiency. It also improves the network's *generalizability*, i.e., its applicability to more variety of inputs.

> "Invariance to local translation can be a useful property if we care more about whether some feature is present than exactly where it is."
>
> –Goodfellow, Bengio, and Courville 2016.

However, the efficiency improvement is achieved at the cost of losing the location information. Especially, in the illustration here the location information is completely lost. This is because *global pooling* was used in the illustration. Global pooling is the case when the pooling operand has the same size as the convolution output.

Global pooling is extreme pooling. It makes the network absolutely invariant. In simple words, this means that the network becomes indifferent to the location of an object in the input.

However, pooling is usually not used in this extremity. Instead, it is used as a means to modulate a network between equivariance and invariance. This is discussed next.

Pooling regularizes a network by making it invariant, i.e., indifferent, to the location of an object in the input.

6.4.2 Modulating between Equivariance and Invariance

Pooling is used to modulate a network between equivariance and invariance. Both the extremities have downsides.

Equivariance, on one hand, preserves the location information but makes the network over-representative. This hurts the network's computational and statistical efficiencies.

Invariance, on the other hand, regularizes the model to improve its efficiency but loses the location information. This makes the network unable to differentiate between original and manipulated inputs, thereby, hurting its reliability and general applicability.

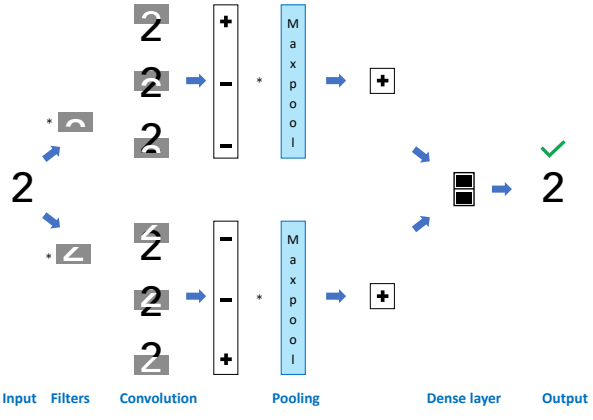
A network is optimal somewhere between the two extremities. Pooling leads to equivariance or invariance depending on whether the pool size is equal to **1** or equal to the convolutional feature map, respectively. Changing the pool size in this range modulates a network between them[2].

At its minimum size of **1**, the pooling output is the same as its input. At its maximum size, pooling makes the network invariant to location translations. The invariance effect is illustrated in Figure 6.10a and 6.10b.

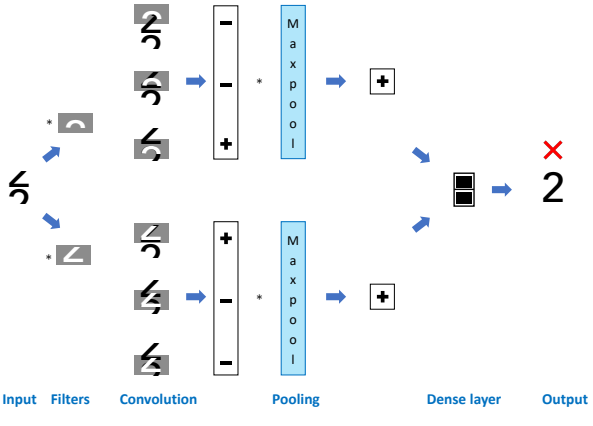
The network in both figures has global pooling. An image of "**2**" and an unknown inscription are analyzed using semi-circle and angle filters in Figure 6.10a and 6.10b, respectively.

The inscription also contains a semi-circle and an angle but in the opposite order compared to "**2**." But the network could not distinguish the inscription from "**2**." The network detected both the inputs, "**2**" and the inscription, as "**2**." The latter (Figure 6.10b) shows that the network

[2]Convolution and pooling, both, are performed along the spatial axes of an input. Therefore, the pool size is defined for the spatial axes. More details are in § 6.8.

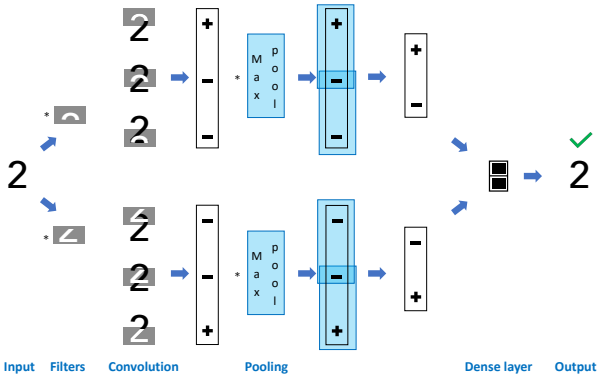


(a) *Letter "2" correctly identified as "2."*

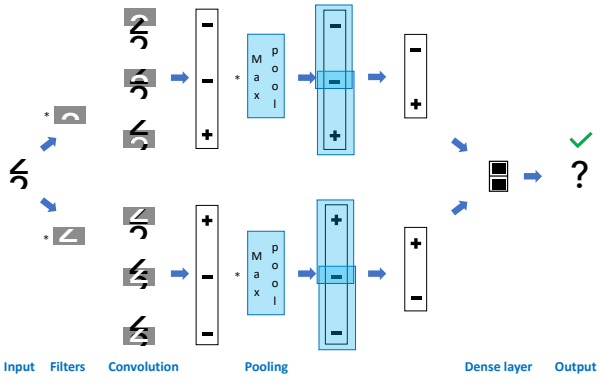


(b) *An inscription incorrectly identified as "2."*

Figure 6.10. *Limitation of invariance is illustrated with global pooling, i.e., pool size equal to its input. This is a pooling extremity that makes the network invariant to location translations. For example, a deformed inscription in the bottom figure has a semi-circle and an angle but not like in a letter "2" is still incorrectly identified as "2."*



(a) *Letter "2" correctly identified as "2."*



(b) *An inscription correctly identified as unknown.*

Figure 6.11. *Pooling modulates the network from invariance to equivariance by reducing the pool size. Doing this preserves the location information and also regularizes the network by reducing the spatial features. With appropriate modulation—reducing pool size to two—the network here is not confused by a manipulative inscription in the bottom figure. Instead, it could correctly identify the top as "2" and the bottom as unknown.*

got confused by the manipulated inscription and incorrectly identified it as "**2**."

This is due to the network's invariance to the location. Although the network's convolutional layer recognized the location of the patterns semi-circle and angle in the inscription as the opposite of "**2**," the pooling layer summarized the convolution output into a single value. As a result, the invariant network could be easily manipulated by an arbitrary distortion.

The issues with equivariance and invariance are resolved by choosing the pool size between **1** and the size of the feature map illustrated in Figure 6.11a and 6.11b.

The network in this illustration has the pool size reduced to two (which is in between the maximum size of 3 and the minimum size of 1). The pooling yields a summary of the feature map within the pool window. Similar to a convolution process, the pool window sweeps the feature map and emits the summary statistic (the maximum, here) at each stride.

A smaller pool size leads to the preservation of some of the location information of patterns. For example, in Figure 6.11a the pooling layer preserved the location information of the semi-circle and angle in the input. Due to which, the network could recognize the letter "**2**" as "**2**" because the semi-circle and angle locations are at the top and bottom of the image, respectively. On the other hand, the location of these patterns is the opposite of the inscription in Figure 6.11b. The network could not recognize this and, therefore, correctly labeled the inscription as *unknown*.

The amount of regularization can be modulated by changing the pool size in pooling. Larger the pool size, the more the regularization, and vice-versa.

A large pool size for higher regularization makes the network invariant to the location of patterns in the input. This makes the network easily confused by manipulative distortions in the input.

The pool size should be chosen such that it provides regularization while preserving the location information.

In sum, a pooling layer provides a mechanism to modulate the network between equivariance and invariance. An appropriately chosen pool size between **1** and the feature map size leads to,

- **a regularized network**. Pooling achieves parameter reduction while preserving location information. This makes the network computationally and statistically efficient. And,

- **a network's robustness** to manipulations by location translations. That is, the network is likely to correctly distinguish the original from its manipulative distortion.

Pooling summarizes the information in a sample. Maximum is one such *summary statistic* and `MaxPool` is one of the most popular pooling methods. More details on summary statistics and the reason for max-pool's popularity is explored towards the end of this chapter in § 6.11 and 6.12. Additionally, deeper insights into the theory behind pooling and a few effective choices are provided there.

6.5 Multi-channel Input

So far the convolutional network constructs were explained using monochrome images as examples. However, most data sets have polychrome, i.e., colorful images.

Despite the varied colors in an image, every color is made from a palette comprising of a few primary colors, e.g., red-green-blue (RGB). Each constituent of the palette is called a *channel*. Therefore, a color image from RGB palette has red, green, and blue channels.

For instance, a multi-channel version of the grayscale image in Figure 6.1 is shown in Figure 6.12a.

(a) *Letter 2 multi-channel: RGB.* (b) *Multi-channel semi-circle and angle filters.*

Figure 6.12. *Multi-channel image and filters.*

Letter "**2**" in Figure 6.12a is a polychrome image from the RGB palette. As shown, the purple-colored "**2**" is a composition of red, green, and blue shades. More specifically, $(Red : 230, Green : 157, Blue : 250)$. Decomposing them into individual channels yields images in each channel, red $(230, 0, 0)$, green $(0, 157, 0)$, and blue $(0, 0, 250)$. The colors in the image are true to scale.

The importance of visualizing an input as a composition of channels is to understand that it requires the filters (convolutional or pooling) to have the same number of channels. For example, Figure 6.12b shows the semi-circle and angle filters with three channels for red, green, and blue, respectively.

However, the example should not be misunderstood as the filter shapes must be the same in each channel. On the contrary, every channel in a filter can have a different shape.

As an example, Figure 6.13a has a flower with different features (shapes) in different channels. For such inputs, the filters take different shapes (learned in a convolutional network) in different channels as shown in Figure 6.13b.

Moreover, the channels in images are usually independent. However, it is not necessary for other data types such as a multivariate time series.

(a) *Multi-channel data with different features in different channels.* (b) *Multi-channel filter with different shapes in different channels.*

Figure 6.13. *Multi-channel image with different characteristic features in different channels.*

Channels in data can be interpreted in various ways. It is the palette constituents in images but it is the features in a multivariate time series. Therefore, one generic understanding of channels is: every channel provides **different information** about a sample.

A convolutional filter has the same number of channels as in the input.

The shape of a filter can be different in different channels.

6.6 Kernels

Convolution is a filtration process. A filter is made of kernel(s). Rather, a kernel is a mathematical representation of a filter.

Kernels can be either a continuous or a discrete function. Among them, discrete kernels are commonly used in convolutional networks due to their flexibility. This choice is similar to fitting an empirical distribution in statistical analysis. Discrete kernels make convolutional networks robust to the distribution of input features.

Suppose, f is a discrete kernel and g is a mono-channel image, then the convolution of f and g is expressed as,

$$f * g = \left[\sum_{u=-\infty}^{+\infty} \sum_{v=-\infty}^{+\infty} f(u,v) g(x-u, y-v) \right], \forall x, y. \tag{6.1}$$

Here, f is a two-dimensional kernel because g is two-dimensional. The dimensions here are *spatial*. For statisticians, the term dimension can be confused with the number of features. Therefore, in this chapter, *axis* is used to denote a spatial dimension.

As mentioned in § 6.2, a convolution operation involves sweeping the input. The sweeping occurs along each axes of the input. Therefore, because the input g has two axes the convolution operation in Equation 6.1 is a double summation along both.

A mono-channel image is a two-axes $m \times n$ matrix. For illustration, suppose the image is a 5×5 matrix,

$$G = \begin{bmatrix} g_{11} & g_{12} & g_{13} & g_{14} & g_{15} \\ g_{21} & g_{22} & g_{23} & g_{24} & g_{25} \\ g_{31} & g_{32} & g_{33} & g_{34} & g_{35} \\ g_{41} & g_{42} & g_{43} & g_{44} & g_{45} \\ g_{51} & g_{52} & g_{53} & g_{54} & g_{55} \end{bmatrix}$$

A discrete kernel in convolutional networks is essentially a tensor. For the two-axes G, a $k \times l$ tensor is taken as a kernel. Consider such an arbitrary 3×5 kernel,

$$F = \begin{bmatrix} f_{11} & f_{12} & f_{13} & f_{14} & f_{15} \\ f_{21} & f_{22} & f_{23} & f_{24} & f_{25} \\ f_{31} & f_{32} & f_{33} & f_{34} & f_{35} \end{bmatrix}$$

Remember from § 6.2 that a convolution operation is like filtration. Mathematically, this is equivalent to determining the similarity between the kernel F and the input G via a convolution operation.

Referring back to Figure 6.8, a convolution operation is simply sweeping a kernel over an image and an output is returned at every stride. The output is a **dot product** of the kernel and a section of the image.

The operation can be expressed by rewriting Equation 6.1 as,

$$F * G = \left[\sum_{u=1}^{k} \sum_{v=1}^{l} f_{uv} g_{x-u+1,y-v+1} \right], \forall x, y \tag{6.2}$$

where $x = k, \ldots, m$, and $y = l, \ldots, n$. Note that x and y range from k and l onward to keep the indexes $x - u + 1$ and $y - v + 1$ positive for all values of u and v. The convolution happens iteratively for every value of x and y. Every iteration is a dot product between the kernel f and a section of input g at (x, y).

The higher the dot product, the stronger is the similarity between the section and the kernel. The more instances of high dot products indicate a significant part of the input G is similar to the kernel F.

In the following, the convolution operation that was figuratively illustrated earlier is illustrated again with tensor (matrix) operations.

The letter "**2**" is represented as a matrix G in Equation 6.3, and *semi-circle* and *angle* filters as kernel matrices F_s and F_a, respectively, in Equation 6.4 and 6.5. The non-zero entries are highlighted in yellow and pink in G and F's, respectively.

$$G = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} \tag{6.3}$$

$$F_s = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{6.4}$$

$$F_a = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} \tag{6.5}$$

Here the G is a 5×5 matrix and $F.$'s are 3×5 tensors, i.e., $m = 5, n = 5, k = 3$ and $l = 5$. Therefore, the range of the iterators x, y in Equation 6.2 are, $x \in \{3, 4, 5\}$ and $y \in \{5\}$. The convolution operation occurs for every (x, y) pairs $(3, 5), (4, 5), (5, 5)$ as shown below.

$$F. * G = \Big[\underbrace{\sum_{u=1}^{3}\sum_{v=1}^{5} f_{uv} g_{3-u+1,5-v+1}}_{A\,(x=3,y=5)}, \underbrace{\sum_{u=1}^{3}\sum_{v=1}^{5} f_{uv} g_{4-u+1,5-v+1}}_{B\colon (x=4,y=5)}, \underbrace{\sum_{u=1}^{3}\sum_{v=1}^{5} f_{uv} g_{5-u+1,5-v+1}}_{C\colon (x=5,y=5)} \Big] \tag{6.6}$$

The operation for F_s is shown below in parts (A), (B), and (C) for (x, y) pairs $(3, 5), (4, 5), (5, 5)$, respectively.

$$A = \begin{array}{l} 0*0+0*0+1*1+0*0+0*0+ \\ 0*0+1*1+0*0+1*1+0*0+ \\ 0*0+0*0+0*0+0*1+0*0 \end{array} = 3,$$

$$B = \begin{array}{l} 0*0+0*1+1*0+0*1+0*0+ \\ 0*0+1*0+0*0+1*1+0*0+ \\ 0*0+0*0+0*1+0*0+0*0 \end{array} = 1, \text{ and}$$

$$C = \begin{array}{l} 0*0+0*0+1*0+0*1+0*0+ \\ 0*0+1*0+0*1+1*0+0*0+ \\ 0*0+0*1+0*1+0*1+0*0 \end{array} = 0$$

Putting (A), (B), and (C) together, the convolution output for F_s is

$$F_s * G = [3, 1, 0]\,. \tag{6.7}$$

Similarly, the convolution output for F_a is,

$$F_a * G = [1, 2, 5]\,. \tag{6.8}$$

Earlier in Figure 6.10a and 6.11a in § 6.4.2, the convolution outputs corresponding to Equation 6.7 and 6.8 were represented as $F_s * G = [+, -, -]$ and $F_a * G = [-, -, +]$ for simplicity. However, as shown in this section, the convolution output is a continuous real number instead of binary.

So far, the kernels are illustrated for mono-channel inputs. But inputs can be multi-channel as shown in § 6.5 in which case a filter has as many channels as the input. A multi-channel filter is made up of multiple kernels—one kernel for each channel. The convolution is then expressed as

$$F * G = \left[\sum_{u=1}^{k}\sum_{v=1}^{l}\sum_{c=1}^{n_c} f_{uvc} g_{x-u+1,y-v+1,c}\right], \forall x, y \tag{6.9}$$

where n_c denotes the number of channels.

In summary, a few things to note in the convolution operation are,

- The cross-sectional size of a kernel is smaller than the input, i.e., $k \leq m$ and $l \leq n$, but they have the same number of channels; sometimes also referred to as the *depth*.
- The dot product is between the kernel and a section of the input's cross-section at (x, y) through its entire depth.
- Irrespective of the depth, the dot product still yields a scalar.
- Consequently, the convolution output shape does not change with the number of channels in the input. This is an important deduction that helps understand the importance of 1×1 convolution in § 6.7.4.

Besides, Equation 6.9 is applicable on a two-axes multi-channel inputs, e.g., a color image. A summation is added (removed) for inputs with additional (lesser) axes, e.g., a video has three axes while time series has a single axis.

A filter is mathematically a kernel. A discrete kernel, which is simply a matrix, is typically used in convolutional networks.

A filter has a kernel for every channel in the input.

The shape of a convolution output does not change with the number of channels in the input.

6.7 Convolutional Variants

6.7.1 Padding

In the previous section, convolution was illustrated using an arbitrary input and kernel of sizes 5×5 and 3×5, respectively. It was observed in Equations 6.7 and 6.8 that the 5×5 input reduced to a 3×1 output.

Consider a network with several such layers. At every layer the features size will shrink considerably.

The shrinking of the feature map is not a problem if the network is shallow or the input is high-dimensional. The feature reduction (regularization) is, in fact, beneficial as it reduces the dimension.

However, a rapid reduction in the feature map size prohibits the addition of any subsequent layer or makes the higher-level layers futile. It is because we can quickly run out of the features available for the next convolutional layer. This issue easily becomes an impediment against constructing deep networks.

Feature maps shrink considerably in traditional convolution. It becomes an issue in deep networks as we quickly run out of features.

$$G' = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(a) *Zero-padding shown in gray.*

$$F_s * G' = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 3 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 2 & 0 \\ 1 & 1 & 3 & 1 & 1 \end{bmatrix}$$

(b) *Convolution output from the "semi-circle" kernel, F_s.*

$$F_a * G' = \begin{bmatrix} 1 & 1 & 3 & 1 & 1 \\ 0 & 2 & 1 & 2 & 1 \\ 1 & 1 & 2 & 2 & 0 \\ 1 & 2 & 5 & 2 & 1 \\ 0 & 2 & 1 & 1 & 0 \end{bmatrix}$$

(c) *Convolution output from the "angle" kernel, F_a.*

Figure 6.14. *Illustration of padding in convolutional layers on a 5×5 image of "2." Convolving the input with kernels F_s and F_a of size 3×5 yielded a 3×1 output in Equation 6.7 and 6.8. This significantly downsized the feature map. The precipitous downsizing prohibits stacking multiple layers as the network quickly runs out of features. This is resolved with padding. Padding works by appending zeros on the periphery of the input. Convolving the padded input (top) with F_s and F_a again results in a 5×5 outputs (bottom) which are of the same size as the input. Padding, thus, prevents the feature map from diminishing and, hence, enables constructing deeper networks.*

The issue can be resolved with a padded convolutional layer. In padding, the size of the input sample is artificially increased to augment the outputted features map size.

Padding was originally designed to make the size of the feature map equal to the size of the input. It is done by appending 0's on the periphery of input. Figure 6.14a illustrates padding of the 5×5 matrix G for the letter "**2**" by adding rows and columns of 0's to form an artificially enlarged input G'. Convolving the enlarged G' with F_s and F_a resulted in an 5×5 output in Figure 6.14b and 6.14c. The outputs are of the **same** size as the original 5×5 input G. It is, thus, called "same" padding.

"Same" padding employs the **maximum** amount of padding. More than this is pointless as it will only add rows and/or columns of zeros in the output. The default convolutional layer with **no** padding is referred to as "valid" padding.

The amount of padding can be fine-tuned between "valid" and "same" padding to optimize the feature size. However, such a fine-tuning should generally be avoided as feature size reduction can be achieved better with pooling.

"Same" padding is useful to construct deep networks. It prevents the feature maps from diminishing allowing high-level layers to extract predictive patterns.

Padding can be done in TensorFlow by setting the `padding` argument as `Conv(..., padding="same")`. Convolution layer, otherwise, has no padding, by default, and can be explicitly set as `Conv(..., padding="valid")`.

6.7.2 Stride

Stride in the context of convolution virtually means the same as a stride in plain English. Stride means the distance between two steps. During a convolution, every iteration in (x, y) in Equation 6.9 is a step. By

default, x and y are iterated with steps of one. In this case, the stride is 1. However, the stride can be increased to skip some steps.

For instance, in the example of convolving a 5×5 input with 3×5 kernels in § 6.6, x is in $\{3, 4, 5\}$. It means the convolution operation is taking steps in the direction of x as $3 \rightarrow 4$ and $4 \rightarrow 5$. The stride here is 1, which is the lowest possible.

But if the input size is large, convolving every step in the input is computationally intensive. In such cases, it could be useful to skip steps by setting the stride greater than 1. In the above example, a `stride=2` reduces the convolution iterations from 3 ($x = \{3, 4, 5\}$) to 2 ($x = \{3, 5\}$).

In large inputs, a stride of 2 significantly reduces the computation time but generally does not affect the accuracy. Despite the benefit, a larger stride (> 2) is often not used as it can adversely affect the network's performance.

A stride of 2 can help reduce computation when the input size is large. However, a larger stride should be avoided.

In TensorFlow, the `strides` argument is a tuple of size $1, 2$, or 3 for `Conv1D`, `Conv2D`, and `Conv3D`, respectively. The tuple specifies the stride along each spatial axes, e.g., height, or width. If a single number is given in the argument, the same stride is applied along all the axes.

6.7.3 Dilation

Dilation means to spread out or expand. This is precisely the purpose of dilation in convolution. When a filter is dilated, it expands by adding empty spaces between its elements.

An illustration of the dilated *semi-circle* and *angle* filters are in Figure 6.15a and 6.15b. In its implementation, the filters' kernels are interspersed with 0's.

A dilated kernel expands its coverage while having fewer elements. Due to lesser elements, a dilated filter reduces the computation. It is

(a) *Dilated "semi-circle" filter.*

(b) *Dilated "angle" filter.*

Figure 6.15. *Illustration of dilated filters. In dilated convolutional layers, the filters are expanded by uniformly adding rows and columns of zero along its spatial axes. This helps in increased coverage of the input with fewer parameters.*

also shown to improve accuracy in certain problems. For example, F. Yu and Koltun 2016 exemplified the benefits of dilated convolution in semantic segmentation problem.

However, replacing a regular filter with a dilated filter sometimes reduces accuracy. To address this, an un-dilated layer and dilated layer are put together in a network such that the latter is larger than the former. This approach achieves an increased coverage of the input without a significant increase in computation.

A dilated convolution layer used together with a regular un-dilated convolution layer increases the coverage of input that typically improves the accuracy without a significant increase in computation.

The dilation argument in Tensorflow is `dilation_rate`. Similar to `strides`, this is also a tuple of size $1, 2$, or 3 for `Conv1D`, `Conv2D`, and `Conv3D`, respectively, that specifies the dilation along each spatial axes. And, if a single number is given, the same dilation is applied along all the axes. Also, note that the term `dilation_rate` should **not** be confused to be a float between 0 and 1. Instead, it a positive integer, i.e.,

$dilation\_rate \in \mathbb{I}^+$. The integer denotes the number of zero rows/-columns interspersed in a kernel.

Besides, dilation should not be confused with stride. Both reduce computation for large inputs. But a dilated convolutional layer has expanded filters. On the other hand, increasing stride does not change the filter size. It only skips some steps.

6.7.4 1x1 Convolution

There is a special and popular convolutional layer that has filters of size 1×1. This means the filter is unlikely to have a pattern to detect in the input. This is contrary to the filtration purpose of convolutional layers. Then, what is its purpose?

Its purpose is different from conventional convolutional layers. Conventional convolutions' purpose is to detect the presence of certain patterns in the input with a filtration-like approach. Unlike them, a 1×1 convolution's purpose is only to **amalgamate** the input channels.

Figure 6.16 illustrates a 1×1 convolution that aggregates the color channels of an image to translate it into a grayscale. As shown in the figure, the 1×1 convolution kernel has the same depth, i.e., channels, as the input. The 1×1 filter moves systematically with a stride of one across the input without any padding (dilation is not applicable in a 1×1 filter) to output a feature map with the same height and width as the input **minus** the depth.

A 1×1 filter does not attempt to learn any pattern. Instead, it is a linear projection of the input. It significantly reduces the dimension of the features and, consequently, the parameters in a network. Besides, some researchers consider a 1×1 convolution as "pooling" of the channels. It is because a pooling layer does not summarize features along the channels but a 1×1 convolution does exactly that.

Moreover, while a single 1×1 convolution reduces the features depth to 1, multiple 1×1 convolution cells in a layer bring different summaries of the channels. The number of cells can be adjusted to increase or decrease the resultant feature map depth.

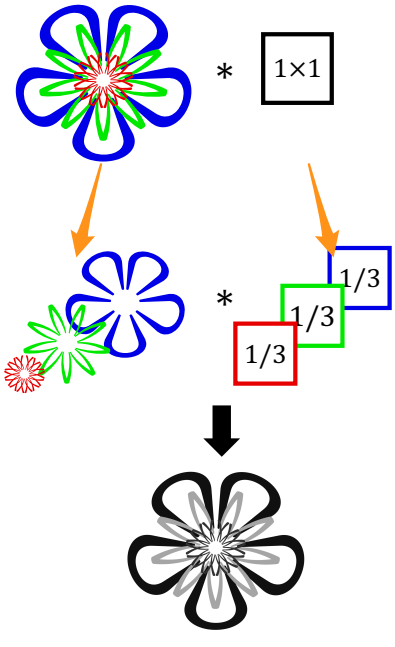


Figure 6.16. *Illustration of an application of a* 1×1 *convolution. The input is a color image with red, blue, and green channels. The channels are intended to be summarized to reduce the input's dimension. A* 1×1 *convolution can be used here to transform the image into a mono-channel grayscale image. For this, a* 1×1 *convolutional kernel with a weight parameter* $1/3$ *is taken. The image and the kernel are expanded here to show the original channels which get amalgamated into a single channel grayscale output. In a convolutional layer, the weight parameter is learned during the network training. A layer can have multiple* 1×1 *convolution cells. Each cell will yield a new transformed channel, e.g., different color schemes in images such as gray, sepia, chrome, vivid, etc. A* 1×1 *convolutional layer can, thus, be used to increase or reduce the channels.*

A 1×1 convolutional layer does not look for shape or object patterns. Instead, its purpose is to amalgamate the signals in the input channels. It can be imagined as the "pooling" of channels.

1×1 convolution plays a critical role in constructing deep networks where the feature maps have to be modulated at different stages. It was initially investigated in Lin, Q. Chen, and Yan 2013. Afterward, Szegedy et al. 2015; He et al. 2016 used them to develop notably deep networks.

6.8 Convolutional Network

6.8.1 Structure

This section exemplifies the structure of a convolutional network. Figure 6.17 illustrates an elementary convolutional network. The components of the network are as follows,

- **Grid-like input**. Convolutional layers take grid-like inputs. The input in the figure is like an image, i.e., it has two axes and three channels each for blue, red, and green.

- **Convolutional layer**. A layer comprises filters. A filter is made up of kernels. It has one kernel for each input channel. The size of a layer is essentially the number of filters in it which is a network configuration. Here, five illustrative filters, viz., diagonal stripes, horizontal stripes, diamond grid, shingles, and waves, are shown in the convolutional layer. Each of them has blue, red, and green channels to match the input.

- **Convolutional output**. A filter sweeps the input to tell the presence/absence of a pattern and its location in the input. In the figure, the outputs corresponding to each filter are shown with a black square of the same pattern. It must be noted that the colored channels in the input are absent in the layer's output. This is

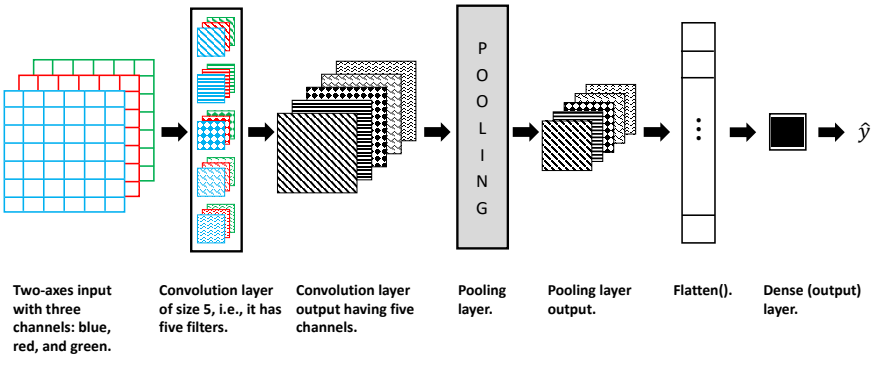


Figure 6.17. *Convolutional networks have a grid-like input. In this illustration, the input is like an image, i.e., it has two axes and three channels for blue, red, and green. The convolutional layer has five illustrative filters, viz., diagonal stripes, horizontal stripes, diamond grid, shingles, and waves. Each filter has as many channels as the input which are blue, red, and green here. The convolutional output from a filter aggregates the information from all the channels. As a result, the input's original channels are relinquished. Instead, each filter's output becomes a channel for the next layer. The output from each filter is, therefore, shown in a respective same pattern black square as a channel. They form a feature map with five channels which is the input to the pooling layer. This layer summarizes and reduces the spatial features but maintains the channel structure. More sets of (higher level) convolutional and pooling layers can be stacked with a caution that sufficient features are passed on after every layer. Towards the end of the network, the output of the convolutional/pooling layer is flattened and followed by a dense output layer.*

because during the convolution operation the information across the channels is aggregated. Consequently, the original channels in the input are relinquished. Instead, the output of each filter becomes a channel for the next layer.

- **Pooling layer**. A convolutional layer is conjoined with a pooling layer. In some texts, e.g., Goodfellow, Bengio, and Courville 2016, a convolutional layer is defined as a combination of a convolutional and pooling layer. The pooling layer summarizes the spatial features, which are the horizontal and vertical axes in the figure.

- **Pooling output**. Pooling reduces the sizes of the spatial axes due to a data summarization along the axes. This makes the network invariant to minor translations and, consequently, robust to noisy inputs. It is important to note that the pooling occurs only along the spatial axes. Therefore, the number of channels remains intact.

- **Flatten**. The feature map thus far is still in a grid-like structure. The flatten operation vectorizes the grid feature map. This is necessary before passing the feature map on to a dense output layer.

- **Dense (output) layer**. Ultimately, a dense layer maps the convolution derived features with the response.

The purpose of a convolutional network is to automatically learn predictive filters from data. Multiple layers are often stacked to learn from low- to high-level features. For instance, a face recognition network could learn the edges of a face in the lower layers and the shape of eyes in the higher layers.

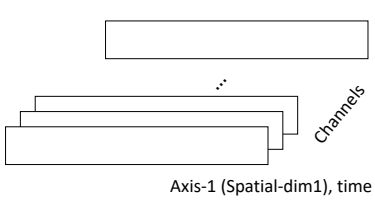
The purpose of a convolutional network is to automatically learn the filters.

6.8.2 `Conv1D`, `Conv2D`, and `Conv3D`

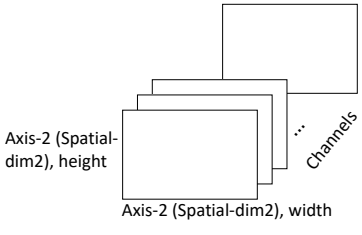
In Tensorflow, the convolutional layer can be chosen from `Conv1D`, `Conv2D`, and `Conv3D`. The three types of layers are designed for inputs with one, two, or three spatial axes, respectively. This section explains when each of them is applicable and their interchangeability.

Convolutional networks work with grid-like inputs. Such inputs are categorized based on their axes and channels. Table 6.1 summarizes them for a few grid-like data, viz. time series, image, and video.

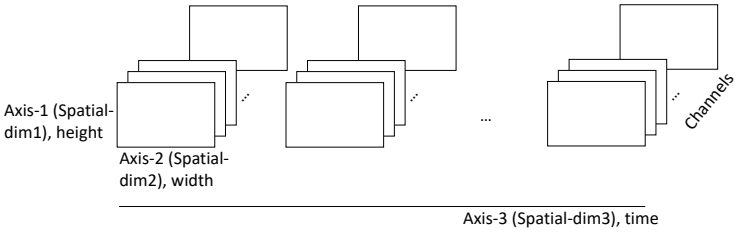
A (univariate) time series has a single spatial axis corresponding to the time. If it is multivariate then the features make the channels. Irrespective of the number of channels, a time series is modeled with `Conv1D` as it has only one spatial dimension.



(a) *Input shape for `Conv1D`.*



(b) *Input shape for `Conv2D`.*



(c) *Input shape for `Conv3D`.*

Figure 6.18. *`Conv` layer for different types of input shapes.*

Images, on the other hand, have two spatial axes along their height and width. Videos have an additional **spatial** axis oxymoronically along **time**. `Conv2D` and `Conv3D` are, therefore, applicable to them, respectively. The channels in them are the palette colors such as red, green, and blue.

Table 6.1. Axes and channels in grid-like inputs to convolutional networks.

	Time series	**Image**	**Video**
Axis-1 (Spatial dim1)	Time	Height	Height
Axis-2 (Spatial dim2)	-	Width	Width
Axis-3 (Spatial dim3)	-	-	Time
Channels	Features (one in univariate time series)	Colors	Colors
Conv'x'd	`Conv1D`	`Conv2D`	`Conv3D`
Input Shape	(samples, time, features)	(samples, height, width, colors)	(samples, height, width, time, colors)
`kernel_size`[3]	An integer, t, specifying the time window	An integer tuple, (h, w), specifying the height and width window.	An integer tuple, (h, w, t), specifying the height, width, and time window.
Kernel shape	(t, features)	(h, w, colors)	(h, w, t, colors)

`Conv1D`, `Conv2D`, and `Conv3D` are used to model inputs with one, two, and three spatial axes, respectively.

The `Conv'x'D` selection is independent of the channels. There could be any number of channels of arbitrary features. Regardless, the `Conv'x'D` is chosen based on the number of spatial axes only.

Inputs to `Conv1D`, `Conv2D`, and `Conv3D` are structured as N-D tensors of shape `(samples, time_steps, features)`, `(samples, height, width, channels)`, and `(samples, height, width, channels)`, respectively. The first axis is reserved for samples for almost every layer in TensorFlow. The shape of a sample is defined by the rest of the axes (shown in Figure 6.18a-6.18c). Among them, the last axis corresponds to the channels (by default) in any of the `Conv'x'D` layers[4] and the rest are the spatial axes.

The `kernel_size` argument in `Conv'x'D` determines the spatial dimension of the convolution kernel. The argument is a tuple of integers. Each element of the tuple corresponds to the kernel's size along the respective spatial dimension. The depth of the kernel is fixed and equal to the number of channels. The depth is, therefore, not included in the argument.

`Conv` layers are agnostic to the number of channels. They differ only by the shape of the input's spatial axes.

Besides, one might observe that a `Conv2D` can be used to model the inputs of `Conv1D` by appropriately reshaping the samples. For example, a time series can be reshaped as `(samples, time, 1, features)`. Similarly, a `Conv3D` can be used to model the inputs of both `Conv1D` and `Conv2D` by reshaping a time series as `(samples, 1, 1, time, features)`

[4]The position of the channel is set with `data_format` argument. It is `channels_last` (default) or `channels_first`.

and an image as `(samples, height, width, 1, colors)`[5]. Essentially, due to their interchangeability, a universal `Conv3D` layer could be made to work with a variety of inputs. The three variants are, still, provided in TensorFlow for convenience.

Additionally, it is worth to learn that a network can be formulated differently by moving the features on the channels to a spatial axis. For example, a multivariate time series can be modeled like an image with a `Conv2D` by reshaping it from `(samples, time, features)` to `(samples, time, features, 1)`. This approach is shown in § 6.10. And, similarly, the channels of an image can be put on a spatial axis as `(sample, height, width, colors, 1)` and modeled with a `Conv3D`.

In short, the input types and the convolutional layer variants are **not** rigidly interlocked. Instead, it is upon the practitioner to formulate the problem as appropriate.

The choice of `Conv1D`, `Conv2D`, and `Conv3D` are not tightly coupled with specific input types. They can be used interchangeably based on a problem.

6.8.3 Convolution Layer Output Size

It is essential to understand the size of a convolutional layer output in a different scenario to construct a network. A typical un-padded convolutional layer, for example, downsizes the feature map. Constructing deep networks without keeping the track of output downsizing may result in an ineffective network.

The output size of a filter in a convolutional layer is,

$$\boldsymbol{o} = \left\lfloor \frac{\boldsymbol{i} - \boldsymbol{k} - (\boldsymbol{k} - 1)(\boldsymbol{d} - 1) + 2\boldsymbol{p}}{\boldsymbol{s}} \right\rfloor + 1 \tag{6.10}$$

where,

[5] In such a restructuring, the `kernel_size` along the unit axes is also made `1`.

- $\boldsymbol{i}$ Size of input's spatial axes,
- $\boldsymbol{k}$ Kernel size,
- $\boldsymbol{s}$ Stride size,
- $\boldsymbol{d}$ Dilation rate,
- $\boldsymbol{p}$ Padding size,

and, each parameter in the equation is a tuple of the same size as the number of spatial axes in the input. For instance, in the convolution example between G and $F_{s,a}$ in § 6.6,

- $\boldsymbol{i}$ $= (5, 5)$, size of the spatial axes of input G,
- $\boldsymbol{k}$ $= (3, 5)$, size of the kernel $F.$,
- $\boldsymbol{s}$ $= (1, 1)$, the default single stride,
- $\boldsymbol{d}$ $= (1, 1)$, the default no dilation, and
- $\boldsymbol{p}$ $= (0, 0)$, the default no padding.

The output size is computed as $\boldsymbol{o} = \left\lfloor \frac{(5,5)-(3,5)-((3,5)-1)((1,1)-1)+2*(0,0)}{(1,1)} \right\rfloor + 1 = (3, 1)$.

Furthermore, a convolutional layer with l filters has an $(\boldsymbol{o}, l)$ tensor as the output where l corresponds to the channels. Extending the example in § 6.6, suppose a convolutional layer has the semi-circle, F_s, and angle, F_a, filters. The output will then be a $(3, 1, 2)$ tensor where the last tuple element 2 correspond to channels—one from each filter.

It is worth noting that the original channels in the input are not included in the output size computation. As mentioned in the previous § 6.8.1, the convolution operation aggregates the channels due to which the original channels are lost in the output. This also implies that a network construction by tracking the output downsizing is unrelated to the input channels.

6.8.4 Pooling Layer Output Size

The output size of the pooling layer is governed by similar parameters as the convolutional. Analogous to the kernel size, a pooling layer has pool size. Strides and padding are also choices in pooling. Dilation, however, is not available in pooling.

Based on them, the pooling output size for a channel is,

$$\boldsymbol{o} = \left\lfloor \frac{\boldsymbol{i} - \boldsymbol{k} + 2\boldsymbol{p}}{\boldsymbol{s}} \right\rfloor + 1 \tag{6.11}$$

where,

- $\boldsymbol{i}$ Size of input's spatial axes,
- $\boldsymbol{k}$ Pool size,
- $\boldsymbol{s}$ Stride size, and
- $\boldsymbol{p}$ Padding size.

If there are l channels, the pooling output is an $(\boldsymbol{o}, l)$ tensor. Essentially, the pooling happens independently for every channel, i.e., the values in the channels are not merged.

Moreover, note that "same" padding does not result in downsampling. Still, the pooling operation brings the invariance attribute to the network and, hence, useful.

6.8.5 Parameters

The number of parameters in a `Conv` layer can be simply expressed as the `((filter volume) + 1) * (number of filters)`.

A filter has weight parameters equal to its volume plus a bias parameter. The *volume* of a filter is the product of the spatial axes (kernel size, $\boldsymbol{k}$) and the depth (the number of channels, c, in the input). For example, in § 6.6 the kernel shape $\boldsymbol{k}$ is $(3, 5)$ and number of channels c for the grayscale input is 1. Therefore, the filter volume is $3 * 5 * 1 = 15$. If the input was in RGB, i.e., $c = 3$, the filter volume becomes $3*5*3 = 45$.

The number of filters is equal to the parameter `filters` set in the definition of a `Conv` layer.

Unlike the `Conv` layer, most of the pooling layers such as `MaxPool` and `AveragePooling` in TensorFlow does not have any trainable parameters.

6.9 Multivariate Time Series Modeling

The rare event prediction problem explored in this book is a multivariate time series. This section proceeds with modeling it with convolutional networks.

6.9.1 Convolution on Time Series

Before modeling, the filters and convolution operation in the context of multivariate time series is briefly explained.

A multivariate time series structure is illustrated in Figure 6.19a. The figure shows an illustrative example in which the x-, y-, and z-axis, show the time, the features, and the features' values, respectively[6].

The time series in the figure has three features with rectangular-, upward pulse-, and downward pulse-like movements. The features are placed along the depth which makes them the channels. A filter for such a time series is shown in Figure 6.19b.

The convolution operation between the filter and the time series is shown in Figure 6.20. As time series has only one spatial axis along time, the convolution sweeps it over time. At each stride, a similarity between the filter and a section of time series is emitted (not shown in the figure). The convolution variants, viz. padding, stride (>1), dilation, and 1×1, work similarly along the time axis.

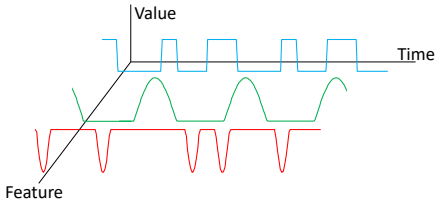
6.9.2 Imports and Data Preparation

Like always, the modeling starts with importing the required libraries, including the user-defined ones.

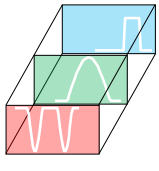
Listing 6.1. Imports for the convolutional network.

```
1 import pandas as pd
2 import numpy as np
3
```

[6]The z-axis with the values should not be confused as a spatial axis as it appears in images. The axis for values of pixels in an image is hidden for simplicity. It can be imagined as an additional axis perpendicular to the image.



(a) *Multivariate time-series.*



(b) *Multivariate time-series filter.*

Figure 6.19. *An illustrative example of a multivariate time series (top) and a filter (bottom). The time series has a single spatial axis along time shown on the x-axis. A univariate time series has a single channel while the features in a multivariate time series become its channels shown on the y-axis. The value of the features with time is shown along the z-axis. This axis should not be confused as a spatial axis like in an image. A time series filter with the same number of channels, i.e., features, is shown in the bottom figure.*

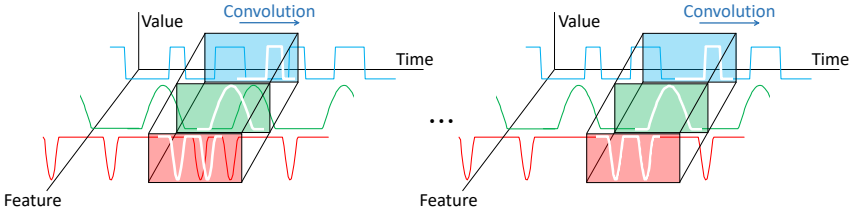


Figure 6.20. *Illustration of a convolution operation on a multivariate time series. Similar to the convolution in images, the filter is swept across the time series input. Since there is only one spatial axis along the time, the sweep happens over it. At each stride, the filter looks for a match between itself and the input. For example, a complete match is detected as the filter comes towards the right.*

```
4  import tensorflow as tf
5  from tensorflow.keras import optimizers
6  from tensorflow.keras.models import Model
7  from tensorflow.keras.models import Sequential
8  from tensorflow.keras.layers import Input
9  from tensorflow.keras.layers import Dense
10 from tensorflow.keras.layers import Dropout
11 from tensorflow.keras.layers import Conv1D
12 from tensorflow.keras.layers import Conv2D
13 from tensorflow.keras.layers import MaxPool1D
14 from tensorflow.keras.layers import AveragePooling1D
15 from tensorflow.keras.layers import MaxPool2D
16 from tensorflow.keras.layers import ReLU
17 from tensorflow.keras.layers import Flatten
18 from tensorflow.python.keras import backend as K
19
20 from sklearn.preprocessing import StandardScaler
21 from sklearn.model_selection import train_test_split
22
23 from collections import Counter
24
25 import matplotlib.pyplot as plt
26 import seaborn as sns
27
```

```
# 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)

SEED = 123  # used to help randomly select the data points
DATA_SPLIT_PCT = 0.2

from pylab import rcParams
rcParams['figure.figsize'] = 8, 6
plt.rcParams.update({'font.size': 22})
```

The tensor shape of a multivariate time series in a convolutional network is the same as in an LSTM network. The *temporalization* procedure discussed in § 5.4.2 is also applied here. The data preprocessing steps for converting categorical features to one-hot encoding and curve-shifting is in Listing 6.2.

Listing 6.2. Data pre-processing.

```
df = pd.read_csv("data/processminer-sheet-break-rare-event-dataset.csv")
df.head(n=5)  # visualize the data.

# Hot encoding
hotencoding1 = pd.get_dummies(df['Grade&Bwt'])
hotencoding1 = hotencoding1.add_prefix('grade_')
hotencoding2 = pd.get_dummies(df['EventPress'])
hotencoding2 = hotencoding2.add_prefix('eventpress_')

df = df.drop(['Grade&Bwt', 'EventPress'], axis=1)

df = pd.concat([df, hotencoding1, hotencoding2], axis=1)

# Rename response column name for ease of understanding
```

```
df = df.rename(columns={'SheetBreak': 'y'})

# Shift the response column y by 2 rows to do a 4-
    min ahead prediction.
df = dp.curve_shift(df, shift_by=-2)

# Sort by time and drop the time column.
df['DateTime'] = pd.to_datetime(df.DateTime)
df = df.sort_values(by='DateTime')
df = df.drop(['DateTime'], axis=1)

# Converts df to numpy array
input_X = df.loc[:, df.columns != 'y'].values
input_y = df['y'].values
```

6.9.3 Baseline

As usual, modeling starts with constructing a baseline model. The baseline models the temporal dependencies up to a lookback of 20, which is the same as the LSTM models in the previous chapter. The *temporalization* is done with this lookback in Listing 6.3. The resultant data in `X` is a `(samples, timesteps, features)` array.

Listing 6.3. Data temporalization and split

```
lookback = 20
X, y = dp.temporalize(X=input_X,
                      y=input_y,
                      lookback=lookback)

# Divide the data into train, valid, and test
X_train, X_test, y_train, y_test =
    train_test_split(np.array(X),
                     np.array(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)
```

```
17
18 # Scaler using the training data.
19 scaler = StandardScaler().fit(dp.flatten(X_train))
20
21 X_train_scaled = dp.scale(X_train, scaler)
22 X_valid_scaled = dp.scale(X_valid, scaler)
23 X_test_scaled = dp.scale(X_test, scaler)
24
25 # Axes lengths
26 TIMESTEPS = X_train_scaled.shape[1]
27 N_FEATURES = X_train_scaled.shape[2]
```

The baseline is a simple network constructed with the `Sequential()` framework in Listing 6.9.3.

Listing 6.4. Baseline convolutional neural network

```
1  model = Sequential()
2  model.add(Input(shape=(TIMESTEPS,
3                         N_FEATURES),
4                  name='input'))
5  model.add(Conv1D(filters=16,
6                   kernel_size=4,
7                   activation='relu',
8                   padding='valid'))
9  model.add(MaxPool1D(pool_size=4,
10                     padding='valid'))
11 model.add(Flatten())
12 model.add(Dense(units=16,
13                 activation='relu'))
14 model.add(Dense(units=1,
15                 activation='sigmoid',
16                 name='output'))
17 model.summary()
```

The network's components are as follows.

Input layer

The shape of an input sample is defined in the `Input()` layer. An input sample here is a `(timesteps, n_features)` tensor due to the *temporalization* during the data processing.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1d (Conv1D)              (None, 17, 16)            4432
_________________________________________________________________
max_pooling1d (MaxPooling1D) (None, 4, 16)             0
_________________________________________________________________
flatten (Flatten)            (None, 64)                0
_________________________________________________________________
dense (Dense)                (None, 16)                1040
_________________________________________________________________
output (Dense)               (None, 1)                 17
=================================================================
Total params: 5,489
Trainable params: 5,489
Non-trainable params: 0
_________________________________________________________________
```

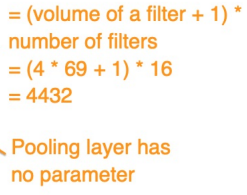


Figure 6.21. *Baseline convolutional network summary.*

Conv layer

The network begins with a `Conv1D` layer. The `kernel_size` is set as `4` and the number of `filters` is `16`. Therefore, there will be `16` convolutional filters each being a `(4, n_features)` tensor.

Here `n_features=69` and, therefore, each filter will have 4×69 weights plus 1 bias parameters. Combining this for all the 16 filters, the convolutional layer contains $(4 * 69 + 1) * 16 = 4,432$ trainable parameters.

The output size of the layer along the spatial axis can be computed using Equation 6.10 as $\boldsymbol{o} = \left\lfloor \frac{(20)-(4)-((3)-1)((1)-1)+2*(0)}{(1)} \right\rfloor + 1 = (17)$. The output size along the channels will be equal to the number of filters in the convolutional layer, i.e., 16. Therefore, the output shape is `(None, 17, 16)` as shown in the model summary in Figure 6.21. Here, `None` corresponds to the `batch_size`.

The number of parameters is computed as per § 6.8.5 as $(4*69+1)*16 = 4,432$, where 4 is the kernel size and 69 is the number of channels, and 16 is the number of filters in the layer.

The `activation` in the `Conv` layer is set to `relu`. Similar to other networks, `relu` is a good choice in a baseline model.

Pooling layer

Max-pooling is one of the most popular pooling. The reason behind its popularity is explained in 6.12.1. `MaxPool`, thus, becomes an obvious choice for the baseline.

Similar to convolutional layers, a pooling layer is chosen from `MaxPool1D`, `MaxPool2D`, and `MaxPool3D` based on the number of spatial axes in its input. Here the output of the `Conv` layer which is the input to the `MaxPool` layer has one spatial axis and, therefore, `MaxPool1D` is used.

The `pool_size` is set to `4` which results in an output of size $\boldsymbol{o} = \left\lfloor \frac{(17)-(4)+2*(0)}{(4)} \right\rfloor + 1 = (4)$ along its (single) spatial axis. Also, the output will have the same number of channels as the number of filters in its input, i.e., 16. Therefore, as also shown in Figure 6.21, the output shape is `(None, 4, 16)` where `None` corresponds to the `batch_size`.

Besides, the pooling layer has no trainable parameters. A few pooling methods such as in Kobayashi 2019b have trainable parameters which can be considered for model improvement.

Flatten layer

This is merely an operational layer. It, naturally, has no trainable parameters. The flattening layer is required to create a feature vector out of the multi-axes tensor outputted from the `MaxPool` layer.

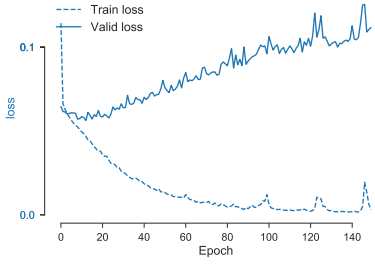
Dense layer

A penultimate `Dense` layer with `16` units is added to reduce the dimension of the features from `64` to `16` before passing them on to the final output dense layer. Since here we have a binary classification problem, the output dense layer requires a single unit with `sigmoid` activation.

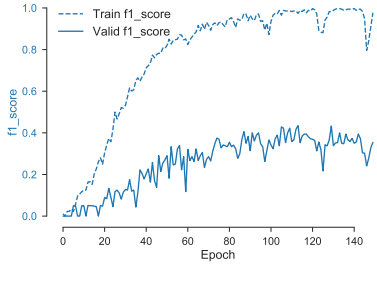
Model fitting and results

The model fit is performed similarly to the other networks discussed in the previous chapters.

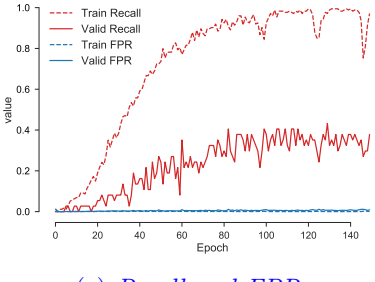
```
# Model fitting
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=[
                  'accuracy',
                  tf.keras.metrics.Recall(),
                  pm.F1Score(),
                  pm.FalsePositiveRate()
              ])
history = model.fit(x=X_train_scaled,
                    y=y_train,
                    batch_size=128,
                    epochs=150,
                    validation_data=(X_valid_scaled,
                        y_valid),
                    verbose=0).history
```



(a) *Loss.*



(b) *F1-score.*



(c) *Recall and FPR.*

Figure 6.22. *Baseline convolutional network results.*

The results are summarized in Figure 6.22a-6.22c. The validation

recall and FPR are close to 0.4 and 0, respectively. Also, f1-score is close to 0.4. These performance results from the baseline convolutional network are already close to the best output in the previous chapter with LSTM models.

It is important to remind ourselves that the models developed here are not for comparison because the performances differ with problems. Instead, the essential finding here is that a convolutional network can work as well or possibly even better than a recurrent neural network (e.g., LSTM) in some temporal data. This means convolutional networks are capable of learning temporal patterns by treating them as spatial.

Convolutional networks have a strong potential to learn temporal patterns. They perform even better than recurrent neural networks in some problems.

6.9.4 Learn Longer-term Dependencies

Recurrent neural networks such as LSTMs are intended to learn long-term dependencies. The amount of the long-term dependencies can be increased by making the `lookback` higher during the data *temporalization*.

In principle, LSTMs should work with any short or long `lookback` because the gradient of the long-term dependencies cannot vanish (see § 5.2.8 in the previous chapter). But, in practice, it does not work because the learned dependencies can get smeared.

This phenomenon was reported in Jozefowicz, Zaremba, and Sutskever 2015. The smearing occurs because LSTMs and similar recurrent neural networks fuse the state information causing them to become confounded in the long-term.

A convolutional network, on the other hand, works like a filtration process with small filters. These filters sweep the input. They are agnostic to the input's size. That means, from a network's accuracy standpoint the filtration with convolution is largely indifferent to the `lookback`.

A network is constructed following this principle in Listing 6.5 where the `lookback=240` as opposed to `20` in the baseline.

Listing 6.5. Convolutional network with longer-term dependencies

```
# Data Temporalize
lookback = 240  # Value 20 in baseline. Increased to
    40 and 240 to learn longer-term dependencies.
X, y = dp.temporalize(X=input_X,
                      y=input_y,
                      lookback=lookback)
X_train, X_test, y_train, y_test =
  train_test_split(np.array(X),
                   np.array(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)
# 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)

TIMESTEPS = X_train_scaled.shape[1]
N_FEATURES = X_train_scaled.shape[2]

# Network construction
model = Sequential()
model.add(Input(shape=(TIMESTEPS,
                       N_FEATURES),
                name='input'))
model.add(Conv1D(filters=16,
                 kernel_size=4,
                 activation='relu'))
model.add(Dropout(0.5))
model.add(MaxPool1D(pool_size=4))
model.add(Flatten())
model.add(Dense(units=16,
```

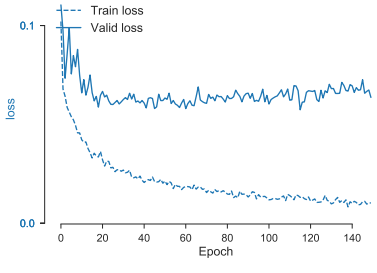
```
                  activation='relu'))
model.add(Dense(units=1,
                  activation='sigmoid',
                  name='output'))
model.summary()

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
```

The results are in Figure 6.23a-6.23c. The accuracy performance increased compared to the baseline. This implies longer-term dependencies exist in the problem.

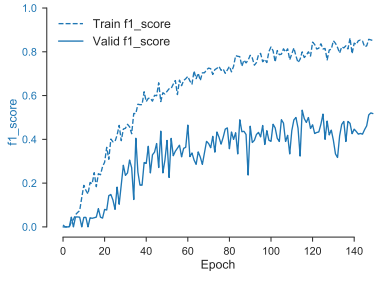
The presence of longer-term dependencies fits with the fact that the process settings in manufacturing processes such as paper manufacturing have lagged effects. For example, the density of pulp going into the machine impacts the paper strength a few hours later at the other end of the machine.

Convolutional networks can capture longer-term patterns in time series processes.

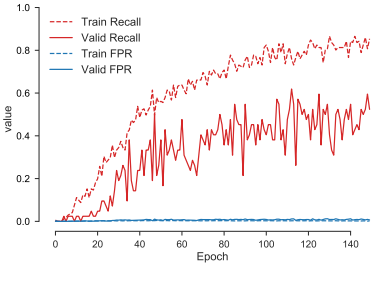
At this point, a question arises: why do not we make the `lookback` even longer? Although a longer `lookback` is usually better for accurate predictions, it significantly increases the computation because the convolutional feature map size rises dramatically. The increase in computation with `lookback` is not specific to convolutional networks. It is



(a) *Loss.*



(b) *F1-score.*



(c) *Recall and FPR.*

Figure 6.23. *Results of a convolutional neural network from learning longer-term dependencies.*