COMS 4030A
Adaptive Computation and Machine Learning

6. Convolutional Neural Networks

'Convolutional layers' are layers within a neural network that are used for datasets where relationships between neighbouring values within data points matter. For example, in image data, pixels that are close to each in an image create features that form integral parts of the larger image. Convolutional layers seek to identify relevant features within the data that contribute to the goal of classifying the input data.

A neural network that uses one or more convolutional layers in its architecture is called a **convolutional neural network**, or **CNN**, for short.

As mentioned above, CNNs are useful for classification tasks on image datasets. One of the first successful applications of CNNs was on the MNIST dataset which consists of images of hand-drawn letters of the alphabet where the target is to correctly identify each hand-drawn letter. Later applications include recognition of faces and classification of images (e.g., identifying images of dogs). However, CNNs have applications in other domains as well, such as time-series data, sensor or signal data.

CNNs use maps called **kernels**, or **feature maps**, that, essentially, scan the data to pick up small features that are relevant to the classification task. The intuition behind kernels is that they are searching for particular kinds of patterns in the data. A large output value obtained from applying a kernel to part of the data is considered a strong response to the feature, while a smaller value is a weak response.

We first consider one-dimension convolution, after which we look at two-dimensional convolution.

6.1. **One-dimensional convolution.**

Example. Consider an input vector $\boldsymbol{x} = (1, 0, 2, 3, 1, 4, 7, 5, 6, 9)$ and a (1-dimensional) **kernel** $\boldsymbol{w} = (0.25, 0.5, 0.25)$. The kernel $\boldsymbol{w}$ can be applied to the vector $\boldsymbol{x}$ to obtain an output vector by passing a 'window' of size 3 (the length of the kernel) over the input from left to right and taking the dot-product with $\boldsymbol{w}$ in each case, as follows:

$$\begin{aligned}
&\boldsymbol{w} \cdot (1,0,2) = (0.25, 0.5, 0.25) \cdot (1,0,2) = (0.25)(1) + (0.5)(0) + (0.25)(2) = 0.75 \\
&\boldsymbol{w} \cdot (0,2,3) = (0.25, 0.5, 0.25) \cdot (0,2,3) = (0.25)(0) + (0.5)(2) + (0.25)(3) = 1.75 \\
&\boldsymbol{w} \cdot (2,3,1) = (0.25, 0.5, 0.25) \cdot (2,3,1) = (0.25)(2) + (0.5)(3) + (0.25)(1) = 2.25 \\
&\boldsymbol{w} \cdot (3,1,4) = (0.25, 0.5, 0.25) \cdot (3,1,4) = (0.25)(3) + (0.5)(1) + (0.25)(4) = 2.25 \\
&\boldsymbol{w} \cdot (1,4,7) = (0.25, 0.5, 0.25) \cdot (1,4,7) = (0.25)(1) + (0.5)(4) + (0.25)(7) = 4 \\
&\boldsymbol{w} \cdot (4,7,5) = (0.25, 0.5, 0.25) \cdot (4,7,5) = (0.25)(4) + (0.5)(7) + (0.25)(5) = 5.5 \\
&\boldsymbol{w} \cdot (7,5,6) = (0.25, 0.5, 0.25) \cdot (1,0,2) = (0.25)(7) + (0.5)(5) + (0.25)(6) = 5.75 \\
&\boldsymbol{w} \cdot (5,6,9) = (0.25, 0.5, 0.25) \cdot (5,6,9) = (0.25)(5) + (0.5)(6) + (0.25)(9) = 6.5.
\end{aligned}$$

The output obtained is $(0.75, 1.75, 2.25, 2.25, 4, 5.5, 5.75, 6.5)$.
Note that the length of the output in this case is 8, which is less than the length of $\boldsymbol{x}$.
This particular kernel has performed a smoothing operation on the input by taking a weighted average of every group of three entries.

In general, given a vector $\boldsymbol{x} = (x_1, x_2, \ldots, x_m)$, and a **kernel** $\boldsymbol{w} = (w_1, w_2, \ldots, w_k)$, where $k \leq m$, applying the kernel to $\boldsymbol{x}$ results in an output vector $\boldsymbol{y}$, where

$$y_i = \sum_{j=1}^{k} w_j x_{j+i-1}.$$

The length of the output is $m - k + 1$.
The application of a kernel to an input, as above, is called **convolution**.
A kernel may also have a **bias**, which is applied after the above windowing process, and it may also have an **activation function** which is applied to every entry in the ouptput.

It is possible to obtain an output that is of the same length as the input by **padding** the input. This means that values are added to the left and/or the right of the input to allow additional windows over the data. Usually a value of 0 is used for padding.

For example, in the above example, a padded version of the input could be

$$\boldsymbol{x}' = (0, 1, 0, 2, 3, 1, 4, 7, 5, 6, 9, 0).$$

Then the output obtained by applying the kernel $\boldsymbol{w}$ to $\boldsymbol{x}'$ would be

$$(0.5, 0.75, 1.75, 2.25, 2.25, 4, 5.5, 5.75, 6.5, 6),$$

which is the same as what was obtained originally but with an additional value at the start and at the end.

In the above example, the window moving over the input moved in single increments; we say that the kernel has **stride** 1. It is possible to use other values for the stride. This affects the size of the output, however. The general rule is as follows.

If no padding is used, then, given a vector $\boldsymbol{x} = (x_1, x_2, \ldots, x_m)$, kernel $\boldsymbol{w} = (w_1, w_2, \ldots, w_k)$, where $k \leq m$, and **stride** s, applying $\boldsymbol{w}$ to $\boldsymbol{x}$ results in an output vector $\boldsymbol{y}$, where

$$y_i = \sum_{j=1}^{k} w_j x_{j+s(i-1)}.$$

As an exercise, find a formula for the length of the output.

Example. This example shows how a kernel can be used to identify a particular feature in an input. Using the input $\boldsymbol{x} = (1, 0, 2, 3, 1, 4, 7, 5, 6, 9)$, consider the kernel $\boldsymbol{w} = (-1, 1)$. The result of applying $\boldsymbol{w}$ to $\boldsymbol{x}$ is

$$(-1, 2, 1, -2, 3, 3, -2, 1, 3). \tag{1}$$

The largest values in the output above occur where there is a sharp increase in consecutive values in $\boldsymbol{x}$. Also, the greater the increase the greater output value, whereas a negative value indicates that there is no increase. We say that the kernel recognises the 'feature' corresponding to an increase in the data values. A large output value is considered a **strong response**, while a small value is considered a **weak response**.

Suppose that the *relu* activation function is used with kernel $(-1, 1)$ in the above example. Then the output in (1) becomes

$$(0, 2, 1, 0, 3, 3, 0, 1, 3)$$

in which the non-zero responsees correspond to a consecutive increase in the data.

It is also possible for a kernel to use a **bias** value. Note that the bias is applied before the activation function. Using the above example again, suppose a bias value of -2 is used. Then, adding the bias to the output in (1), gives

$$(-3, 0, -1, -4, 1, 1, -4, -1, 1).$$

If the *relu* activation function is then applied, the final output would be

$$(0, 0, 0, 0, 1, 1, 0, 0, 1).$$

In this case, one could consider that the bias is used to pick up only the strong responses to the kernel, that is, where there is a sharp increase in consecutive data values.

Example. Suppose a kernel is required to pick up large 'spikes' within the data, i.e., values that are greater than the values on either side of it. Consider the kernel $\boldsymbol{w} = (-1, 1, -1)$, with bias -2 and *relu* activation function. If $\boldsymbol{w}$ is applied to the input $\boldsymbol{x} = (1, 4, 2, 9, 1, 0, 1, 7, 3, 2)$, the output is obtained as follows. First, the result of applying the kernel to windows of the data is

$$(1, -11, 6, -8, -2, -6, 3, -6).$$

Next, appying the bias, gives

$$(-1, -13, 4, -10, -4, -8, 1, -8)$$

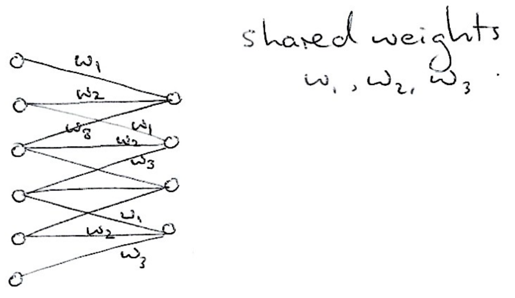
and then applying *relu* gives

$$(0, 0, 4, 0, 0, 0, 1, 0).$$

Thus, the kernel has a strong response to the large spikes in the data.

Convolution can be considered as a special type of layer in a neural network, where the same weights are used for multiple edges. The values in the kernel are the weights of corresponding edges in the convolutional layer. Kernels can have positive and negative values.

The following diagram shows how a convolutional layer with kernel $\boldsymbol{w} = (w_1, w_2, w_3)$ would look within a neural network. (Having such a layer makes it a convolutional neural network, or CNN.) The three weights w_1, w_2 and w_3 are shared by a number of edges.



There are a few observation to make here. Firstly, the convolutional layer is not a fully connected layer; in fact it is a **sparsely connected** layer. Secondly, the convolutional layer uses **parameter sharing**, i.e., the same weights are used for many edges. Together, these two properties mean that there will be far fewer weights for the neural network training algorithm to learn. This is important in applications that use images, for example, since the size of the input images can be in the thousands of pixels. Having fewer weights improves the efficiency of the training algorithm. Thirdly, by sharing weights over the whole input, the convolutional layer is **equivariant to translation**, meaning that if the input data is translated, i.e., shifted, in some way, then the responses to the kernel will still be picked up but translated in the same way as the data. Thus, features that respond to the kernel will do so regardless of their position within the data.

Example. To see what is meant by equivariance to translation, consider the kernel $\boldsymbol{w} = (-1, 1, -1)$, with bias -2 and *relu* activation function, as in the previous example. If $\boldsymbol{w}$ is applied to the input $\boldsymbol{x} = (1, 4, 2, 9, 1, 0, 1, 7, 3, 2)$, the output is $(0, 0, 4, 0, 0, 0, 1, 0)$. Now, suppose the input values are translated 1 step to the right, so $\boldsymbol{x}' = (0, 1, 4, 2, 9, 1, 0, 1, 7, 3)$, where a 0 has been added to the left. Then the result of applying $\boldsymbol{w}$ to $\boldsymbol{x}'$ is $(0, 0, 0, 4, 0, 0, 0, 1)$, which is the output translated by 1 step to the right. The features are still detected by the kernel.

If f denotes the application of the kernel and g denotes the operation of the translation, then invariance to translation can be written as $f(g(\boldsymbol{x})) = g(f(\boldsymbol{x}))$.
All this means that features will be detected by kernels if they are present somewhere in the data, even if the data is translated in some way (except in cases where the feature is translated out of the data).

6.2. **Two-dimensional convolution.**

Example: Consider the following matrix which represents a small image.

$$\begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

A (two-dimensional) **kernel** is a square matrix of values that is smaller than the image such as the following:

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}.$$

The kernel can be applied to the image to obtain an output vector by passing a 'window' of size 3×3 over the image and performing pointwise multiplication (i.e., a dot-product) with the kernel in each case, as shown below.

First, we must see how the kernel is applied to another 3×3 matrix.
Given the following matrix (which could be part of a larger matrix)

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

the kernel is applied by doing **pointwise multiplication** and then **summing** the values as follows:

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = \begin{matrix} 0(a) & + & 0(b) & + & 1(c) \\ +\,0(d) & + & 1(e) & + & 0(f) \\ +\,1(g) & + & 0(h) & + & 0(i) \end{matrix}$$

which equals $c + e + g$.

Applying the kernel to the above matrix will produce the following output:

$$\begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} \longrightarrow \begin{bmatrix} 3 & 2 & 0 & 1 \\ 2 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 2 & 2 \end{bmatrix}$$

The output is obtained by passing a 3×3 window over the matrix, starting from the top left, and applying the kernel to the window, as follows:

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} = \begin{matrix} 0 & + & 0 & + & 1 \\ +\,0 & + & 1 & + & 0 \\ +\,1 & + & 0 & + & 0 \end{matrix} = 3$$

Next, the window is shifted one column to the right and the kernel applied to the next 3×3 window:

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} = \begin{matrix} 0 & + \; 0 & + \; 1 \\ +\,0 & + \; 0 & + \; 0 \\ +\,1 & + \; 0 & + \; 0 \end{matrix} = 2$$

Shifting one column right again and applying the kernel gives:

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} = \begin{matrix} 0 & + \; 0 & + \; 0 \\ +\,0 & + \; 0 & + \; 0 \\ +\,0 & + \; 0 & + \; 0 \end{matrix} = 0$$

Finally, shifting one column right again and applying the kernel gives:

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix} = \begin{matrix} 0 & + \; 0 & + \; 0 \\ +\,0 & + \; 1 & + \; 0 \\ +\,0 & + \; 0 & + \; 0 \end{matrix} = 1$$

This demonstrates how the top line of $3, 2, 0, 1$ in the output matrix is obtained.

Next, the window starts at the left-hand side of the matrix again but shifts one row down. Applying the kernel here produces the following value:

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} = \begin{matrix} 0 & + \; 0 & + \; 0 \\ +\,0 & + \; 1 & + \; 0 \\ +\,1 & + \; 0 & + \; 0 \end{matrix} = 2$$

So the first entry in the second row of the output matrix is 2.
The next entry is obtained by shifting the window one column to the right.
Continuing in this way, the final output matrix above is obtained (please check!).

Observe that strong responses (the 2's and 3's) in the above example are obtained in regions of the data that match the kernel and weak responses are received in regions that don't match the kernel.

Observe, also, that the input matrix is of size 6×6, and the output matrix that results from applying the kernel is of size 4×4. Thus, the size of the input has been reduced.

In the above example, the kernel used a **stride** of 1 **across** and 1 **down**. Different values could have been used for these strides. Of course, this affects the size of the resulting matrix. A large stride can be used to reduce the size of the input, if required.

As in the one-dimensional case, a kernel can have a **bias** value, as well as an **activation function** that is applied after the kernel application is complete. Any activation function is allowed, but the *relu* is most commonly used because of its computational efficiency.

As an example, suppose that in the above example the kernel has a bias of -1 and uses a *relu* activation function. Adding the bias at each application of the kernel gives the matrix:

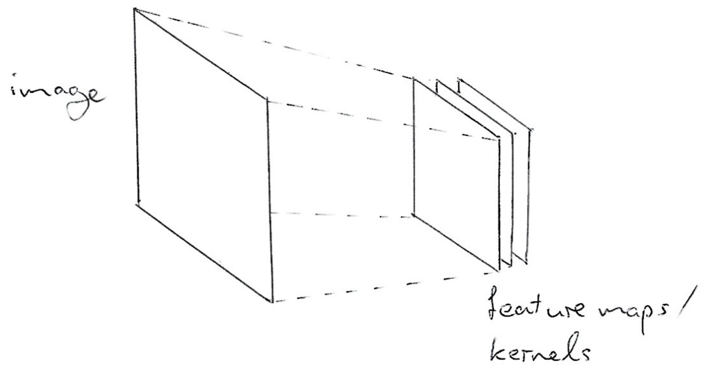
$$\begin{bmatrix} 2 & 1 & -1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 1 \end{bmatrix}$$

Applying *relu* to the above matrix replaces all negative values by 0, giving the following output:

$$\begin{bmatrix} 2 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

where we see that the regions matching the kernel have given a strong response and there is no response elsewhere.

For convolutional layers applied to image data, the following kind of diagram is often used to illustrate the layer. In the convolutional layer illustrated here, three kernels, or feature maps, have been applied to the input image.



It's important to note that the kernels used in a CNN are not selected by the user, but are learned by the neural network training algorithm; the algorithm finds the best kernels for the classification task at hand. Also, a convolutional layer can have a number of kernels, and a CNN can have a number of convolutional layers.

Another type of layer that is common in a CNN is a **pooling layer**. Such layers usually follow a convolutional layer. A pooling layer is a special kind of layer in a neural network that replaces blocks of some size in the current matrix by a single value, thereby substantially reducing the size of the matrix. Usually the block is replaced by the maximum value in the block – this is called **max-pooling**. Pooling is similar to convolution in the sense that it also uses the idea of a window passing over the data, and this window also uses a stride, which is usually the same as the width of the block.

The method of pooling is easy to understand from a small example. We apply max-pooling with a window of size 2×2 and with stride 2 to the 4×4 matrix above that resulted from the kernel application. The output from the pooling layer is

$$\begin{bmatrix} 2 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \longrightarrow \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$$

The above output matrix is obtained by first applying max-pooling to the top left 2×2 block

$$\begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix}.$$

The maximum value in the block is chosen, namely 2, and the whole block is replaced by a single value 2. Striding 2 columns to the right, max-pooling is then applied to the block

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}.$$

The maximum value in the block is chosen, namely 0, and the whole block is replaced by a single value 0. Starting at the left of the matrix again, but striding 2 rows down, max-pooling is applied to the bottom left block

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

which is replace by a 0. Lastly, striding 2 columns to the right, max-pooling is then applied to the block

$$\begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}$$

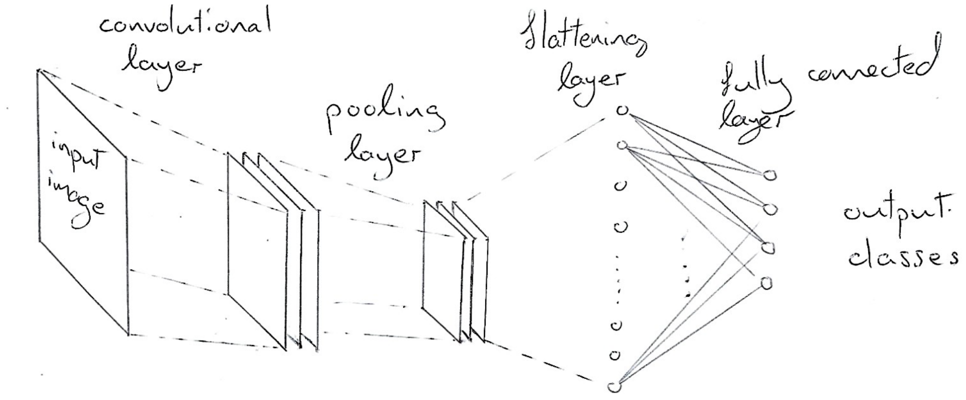
which is replaced by a 1.

The final matrix obtained from the pooling layer is, therefore,

$$\begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$$

Alternatives to max-pooling are **min-pooling** and **average-pooling**.

The structure of a simple convolutional neural network (CNN) is illustrated in the figure below.



In the above figure, there is a single convolutional layer which uses three kernels. This layer is followed by a pooling layer which is applied to each of the matrices obtained from the kernels. After that there is a **flattening layer** which simply takes all the values in the matrices in the previous layer and puts them into a single 1-dimensional array. Thereafter, a (standard) fully connected layer connects the flattened layer to the output layer.

In practice, a convolutional neural network may use many convolutional layers, together with pooling layers and a large number of kernels. In addition, there may be more fully connected layers at the end. Note that convolution and pooling serve to reduce the data size so that the fully connected layers require fewer weights.

Given a dataset consisting of inputs and corresponding targets, once a convolutional neural network structure has been chosen, the network is trained on the dataset using backpropagation as in the standard neural network training algorithm.

6.3. **Training a CNN.**

Suppose we are given a dataset consisting of inputs with corresponding targets and wish use a CNN to model the dataset. For example, the dataset could consist of images with classification labels, in which case the objective of the CNN would be to predict the correct label for an input in the dataset, and also to generalise to images not in the dataset.

An architecture for the CNN must be chosen, i.e., the number and types of layers (convolutional, pooling, standard), the number of kernels, the size of kernels, the number of nodes, activation functions, etc. In addition, the output should be suitably structured to match the target values.

Next, an appropriate loss function is chosen and backpropagation is used to train the weights of the CNN. The method is as before. First divide the data in training, validation and test sets. Next, an input from the test dataset is fed into the CNN and an output obtained. Then the gradient descent method is used to reduce the loss between the output and the target. The gradient descent method is applied using the backpropagation technique.

The updating of the weights in the convolutional layers, that is, the weights of the kernels, is done as follows. Suppose that w is one of the weights of a kernel. Since the kernel is applied in many places, it affects the activation values at a number of nodes in the network. We can think of the kernel as a weight that is shared by a number of edges. Suppose that the edges that share the weight w are the the edges between nodes m_i and n_i, for i from 1 to k.

Then the weight w contributes directly to the z values at the nodes $n_1, \ldots, n_k$, which we denote by $z_{n_1}, \ldots, z_{n_k}$, so

$$\begin{aligned}\frac{\partial L}{\partial w} &= \sum_{i=1}^{k} \frac{\partial L}{\partial z_{n_i}} \frac{\partial z_{n_i}}{\partial w} \\ &= \sum_{i=1}^{k} \frac{\partial L}{\partial z_{n_i}} y_{m_i}.\end{aligned}$$

In the second step above we have used the fact that z_{n_i} is the sum of the outputs (i.e., the y's) of the nodes that connect to the node n_i multiplied by the weight on the corresponding edge. Since m_i is one of those nodes and has corresponding edge weight w, we have $\frac{\partial z_{n_i}}{\partial w} = y_{m_i}$. Thus,

$$\frac{\partial L}{\partial w}\Big|_{\boldsymbol{xtW}} = \sum_{i=1}^{k} \left(\frac{\partial L}{\partial z_{n_i}}\Big|_{\boldsymbol{xtW}} \right) \left(y_{m_i}\Big|_{\boldsymbol{xtW}} \right) = \sum_{i=1}^{k} \delta_{n_i} a_{m_i}$$

Thus, the gradient descent update to weight w is

$$\underline{w} \leftarrow \underline{w} - \eta \left(\textstyle\sum_{i=1}^{k} \delta_{n_i} a_{m_i} \right)$$

where w is the shared weight on the edges connecting node m_i to n_i, for i from 1 to k. Note that the delta values at nodes are computed as before.

The edges leading to the flattening layer do not have any weights; the layer is simply a re-arrangement of the corresponding nodes, so no updates are required here. Similarly, the edges in the pooling layers do not have weights so no weight updates are required here either. However, the pooling layer does affect the delta values at certain nodes. For example, suppose that the max-pooling window of size 2×2 is applied to the following part of the data:

$$\begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix}$$

which results in the output of 2. The four values in the above matrix are the activation values at four specific nodes in the network. The effect of the pooling is that the top left node's activation value of 2 contributes to the final output, while the other three nodes do not contribute to the output. This means that the delta value at the top left node is obtained from the delta value at the nodes it is connected to in the next layer, while the delta values at the other nodes is 0.

EXERCISES

(1) Consider the following matrix:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

(i) Apply the following 3×3 kernel, using stride 1, to the above matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

(ii) Apply the above kernel using a bias of -1 and a *relu* activation function.

(iii) Apply the following 3×3 kernel, using stride 1, to the above matrix

$$\begin{bmatrix} 2 & -1 & 0 \\ -1 & 1 & -1 \\ 0 & -1 & 1 \end{bmatrix}.$$

(iv) Apply the above kernel using a bias of -1 and a *relu* activation function.

(v) Apply max-pooling using a 2×2 window with stride 2 to the output matrices obtained in exercises (i–iv).

(2) Consider the following matrix:

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

(i) Apply the following 2×2 kernel, using stride 1, to the above matrix

$$\begin{bmatrix} 1 & -1 \\ 2 & 1 \end{bmatrix}.$$

(ii) Apply the above kernel using a bias of -1 and a *relu* activation function.

(iii) Apply the following 2×2 kernel, using stride 2, to the above matrix

$$\begin{bmatrix} -1 & 1 \\ 1 & 1 \end{bmatrix}.$$

(iv) Apply the above kernel using a bias of -1 and a *relu* activation function.