B
Primer on convolutional neural networks

Contents

In this section, we provide a brief primer on deep learning, with a particular emphasis on explaining the basics for training an object classification CNN. This primer is primarily for readers with minimal background knowledge. For a primer on the mathematical notation and concepts used, see appendix A. In particular, see appendix A.7 for a detailed description of the notation we use to describe CNNs.

B.1 Machine learning set-up

A machine learning set-up is primarily defined by the following four ingredients: the *model* being trained as well as the *data*, *task*, and *training procedure* used.

B.1.1 Data

The *training data* is most often defined as pairs of inputs and outputs[1] (*i.e.* $\mathcal{D} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^{N}$), where a model's aim is to predict an output from its corresponding input. For object classification, the inputs are RGB images (*i.e.* $\boldsymbol{x} \in \mathbb{R}^{3 \times H \times W}$) and the outputs are labels of the most dominant object in the corresponding images (*i.e.* "sheepdog", "sea snake", "soup bowl", "alps", etc.). The output label is also known as the **ground truth** label. The output label for object classification is usually represented as a *one-hot vector*[2] of length C, where C denotes the number of object categories (*i.e.* $\boldsymbol{y} \in \mathbb{R}^{C}$) and the indices of the vector denote different object categories. For a given input-output pair (*i.e.* $(\boldsymbol{x}, \boldsymbol{y})$), the index of the output vector (*i.e.* $\boldsymbol{y}$) containing 1 denotes the most dominant object in the input image (*i.e.* $\boldsymbol{x}$).

B.1.2 Task

The format of the training data is deeply related to the training task, which is typically an informal description of the prediction task that the model is being trained for.

B.1.2.1 Object classification

For example, *object classification* refers to the task of predicting the dominant object category of an image, while *colorization* (R. Zhang et al., 2016) refers to predicting a color image from a black-and-white version of it. More formally, the task is typically defined by the training data and the *loss function* used to train the model. The majority of machine learning tasks aim to minimize a loss function, which captures a notion of error or incorrect behavior and can be *optimized*.[3]

B.1.2.2 Cross-entropy loss

For classification problems, the *cross-entropy loss* (*a.k.a. log loss*) function is typically used.[4][5] In its simplest form, it measures the performance of a classification model

[1]When the outputs require human annotation (*i.e.* labeling), this set-up is known as *supervised learning*. When the outputs do not require human annotation (*i.e.* they are free), this set-up is known as *self-supervised learning*. An example of a self-supervised set-up is predicting a color image from a black-and-white version of it (R. Zhang et al., 2016).

[2]A *one-hot vector* is a vector which is filled with zeros except at one position, where it is filled with a 1 (*e.g.* $\begin{bmatrix}0 & 0 & 1 & 0\end{bmatrix}$).

[3]*Optimization* refers to the selection of the best solution from a set of possible solutions. See the Wikipedia article on "Mathematical optimization".

[4]This paragraph is paraphrased from (*Loss Functions* 2017).

[5]See this explanation for a more thorough and accessible, visual treatment of the cross-entropy loss function.

whose output is a probability between 0 and 1 (*i.e.* a binary classifier, in which positive examples should be predicted as 1 and negative examples as 0). The cross-entropy loss is large when the predicted probability is relatively far away from the true (*a.k.a.* ground-truth) label and is small when the predicted probability is relatively close to the true label. Thus a perfect model would have a loss of 0 and the goal of the model is to minimize the loss so that its predictions are as close as possible to the true labels.

Mathematically, for a binary classifier for which the label is either 0 or 1 (*i.e.* $y \in \{0, 1\}$) and its prediction is between 0 and 1 (*i.e.* $\hat{y} \in [0, 1]$), the cross-entropy loss function is defined as follows:

$$\mathcal{L} = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})). \tag{B.1}$$

To build intuition, consider a poor prediction of $\hat{y} = 0.2$ vs. a better prediction of $\hat{y} = 0.9$ for a positive example (*i.e.* $y = 1$). For the poor prediction, the value of the loss is computed as follows:

$$\mathcal{L} = -(1 \log(0.2) + 0 \log(0.8)) = -1 \log(0.2) \approx 0.70. \tag{B.2}$$

For the good prediction, the value of the loss is as follows:

$$\mathcal{L} = -(1 \log(0.9) + 0 \log(0.1)) = -1 \log(0.9) \approx 0.05. \tag{B.3}$$

Thus, this simple example, makes clear that the value of the cross-entropy loss function is higher for bad predictions and lower for good ones.

For a classification task with more than two possible label classes (*a.k.a.* multi-class classification), the cross-entropy loss function is the sum of individual cross-entropy loss terms for each label:

$$\mathcal{L} = \sum_{c=1}^{C} y_c \log(\hat{y}_c), \tag{B.4}$$

where $y_c \in \{0, 1\}$ refers to the value at index c of the one-hot vector $\boldsymbol{y} \in \{0, 1\}^C$ (*i.e.* it is either 0 or 1) and where $\hat{y}_c \in [0, 1]$ refers to the value at index c of the predicted output vector $\hat{\boldsymbol{y}}$ (*i.e.* it ranges from 0 to 1 and represents the predicted probability for class c).

Typically, a task is formulated to minimize the loss function when applied to all training examples (*i.e.* by summing up the loss values for each example). Thus, for an object classifier, the following loss would be used:

$$\mathcal{L} = \sum_{i=1}^{N} \sum_{c=1}^{C} y_{i,c} \log(\hat{y}_{i,c}), \tag{B.5}$$

where i represents the index of a specific training datapoint.

B.1.3 Model

Deep learning refers to the use of *artificial neural networks* (*a.k.a.* deep neural networks), which comprise a class of algorithms[6] that is very loosely inspired by how the human brain processes information. Deep neural networks consist of multiple *layers* that successively process the input to the network, much like how the human successively processes raw visual input from the eyes, in order to produce an output. Some layers contain *parameters* (*a.k.a. weights*) that need to be optimized in order for the network to perform the given task well. In appendix B.2, we go into detail for the class of models discussed in this thesis: convolutional neural networks.

B.1.4 Training procedure

A deep neural network is typically trained by updating its parameters via *back-propagation* in order to minimize a loss function.

B.1.4.1 Backpropagation

Backpropagation refers to a class of algorithms in which the gradients $\frac{\partial \mathcal{L}}{\partial \theta}$ of a loss function with respect to a network's parameters, θ, are computed efficiently[7] for a single input-output pair (*i.e.* $(\boldsymbol{x}, \boldsymbol{y})$). The gradient captures how much each parameter should change (*i.e.* magnitude) and in which direction (*i.e.* positive or negative) in order to increase the loss value for that particular input-output pair.

B.1.4.2 Gradient descent

Because we are interested in decreasing (*i.e.* minimizing) the loss, we typically update parameters by taking a step in the negative direction of the gradient; this is known as *gradient descent* and is given by the following *update rule*:[8]

$$\theta := \theta - \gamma \frac{\partial \mathcal{L}}{\partial \theta}, \tag{B.6}$$

where γ is the step size (*a.k.a. learning rate*).

[6]An *algorithm* is a well-defined process to perform a computation or specific task. It is analogous to a cooking recipe that instructs a novice chef how to make a scrumptious scone.

[7]As opposed to naively computing the gradient for every network parameter independently, which would be computationally expensive.

[8]$A := B$ denotes *assigning* the value of A to B in computer science notation. In the case of eq. (B.6), the new value of θ is equal to to right hand side of the equation.

In practice, *stochastic gradient descent* (SGD), an iterative method for updating parameters based on random (*i.e.* stochastic) subsets (*a.k.a.* *batches*) of the training dataset,[9] is typically used:

Algorithm 1: Stochastic gradient descent (SGD)

```
Data: Training data $\mathcal{D}$
Hyperparameters: $T$ (# training steps), $B$ (batch size), $\gamma$ (learning rate)
Randomly initialize network parameters $\theta$
for $t = 1 \ldots T$ do
    Randomly sample a batch: $\{\boldsymbol{x}_b, \boldsymbol{y}_b\}_{b=1}^{B} \sim \mathcal{D}$
    Compute loss for every item in the batch: $\{\mathcal{L}_b\}_{b=1}^{B}$
    Update $\theta$ using batch's gradient: $\theta := \theta - \frac{\gamma}{B} \sum_{b=1}^{B} \frac{\partial \mathcal{L}_b}{\partial \theta}$
end
```

The optimization settings T (number of training steps), B (batch size), and γ (learning rate) are examples of **hyperparameters** for a network. A hyperparameter is a parameter that is typically set prior to training by a human; this is in contrast to the network's parameters, which are automatically learned.

B.2 Convolutional neural networks

Convolutional neural networks (CNNs) are a particular kind of deep neural networks that is most frequently used on *visual data* (*e.g.* an image). They are distinguished by their use of *convolutional layers*, which enable them to recognize distinctive visual patterns (*e.g.* furry ears) regardless of their location in an image. This property is known as *shift invariance*.[10]

In this section, we briefly survey the basic components of a CNN.

A *layer* refers to an operation that is applied to an input *tensor* and produces an output tensor, which may or may not be the same shape as the input tensor.

B.2.1 Linear layers

The basic building block of deep neural networks are *linear layers* that *linearly combine* (see appendix A.4.1) an input tensor with learned parameters (*i.e.* weights) to produce an output tensor.

There are two basic kinds of linear layers: fully-connected layers and convolutional layers.

[9] By taking random subsets, SGD approximates the actual gradient, which would need to be computed for the entire dataset.

[10] Here, *shift* refers to shifting the position of a pattern and *invariance* connotes remaining unchanged regardless of changes in another property (in this case, changes in spatial shift).

B.2.1.1 Fully-connected layer

A fully-connected layer is a layer in which a unique weight is used to "connect" every input neuron (*i.e.* element in the input tensor) to every output neuron (*i.e.* element in the output tensor).

2-layer example. Consider the following neural network $\Phi : \mathbb{R}^3 \rightarrow \mathbb{R}^2$ that contains two linear layers. Let the following matrices and vectors be the parameters of the first and second layer (superscript denotes the layer index) defined as follows:

$$\boldsymbol{W}^1 \in \mathbb{R}^{3\times 4}, \boldsymbol{W}^2 \in \mathbb{R}^{4\times 2}, \boldsymbol{b}^1 \in \mathbb{R}^3, \boldsymbol{b}^2 \in \mathbb{R}^2, \tag{B.7}$$

and let the following tensors be defined as the input, intermediate (*i.e.* activation), and output tensors respectively:

$$\boldsymbol{x} \in \mathbb{R}^3, \boldsymbol{z} \in \mathbb{R}^4, \hat{\boldsymbol{y}} \in \mathbb{R}^2. \tag{B.8}$$

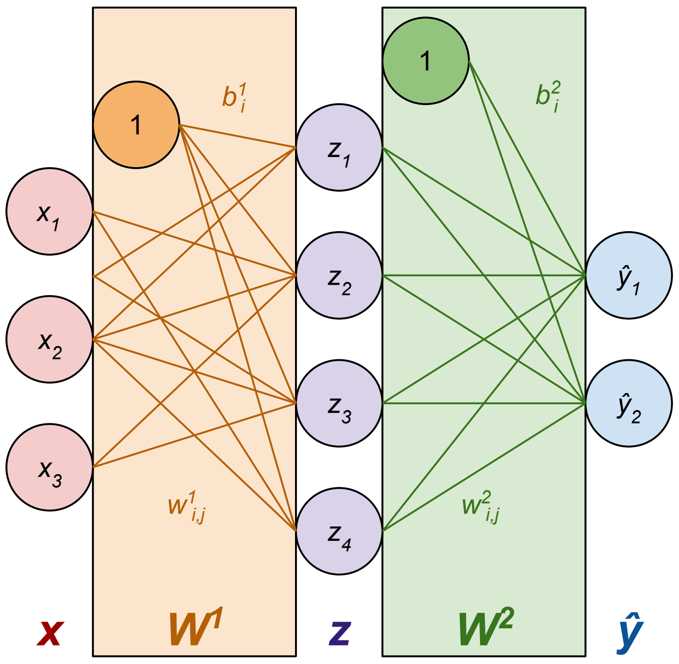

Figure B.1: Diagram of two fully-connected layers.

Then, layer 1 can be defined as the function $f^1 : \mathbb{R}^3 \rightarrow \mathbb{R}^4$, where the element at the j-th position in the output tensor $\boldsymbol{z}$ is given by

$$z_j = (\sum_{i=1}^{3} x_i \cdot w^1_{i,j}) + b^1_j. \tag{B.9}$$

Using matrix multiplication, we can write the function f^1 as follows:

$$\boldsymbol{z} = f^1(\boldsymbol{x}) = \boldsymbol{W}^{1^T}\boldsymbol{x} + \boldsymbol{b}^1, \tag{B.10}$$

where $\boldsymbol{W}^{1^T} : \mathbb{R}^4 \to \mathbb{R}^3$ is the transposed matrix of $\boldsymbol{W}^1$ (*i.e.* matrix with its rows and columns swapped). Layer 2 can be similarly defined as the function $f^2 : \mathbb{R}^4 \to \mathbb{R}^2$, with the j-th position element in its output given by

$$\hat{y}_j = (\sum_{i=1}^{4} z_i \cdot w^2_{i,j}) + b^2_j, \tag{B.11}$$

and re-written using matrix multiplication as

$$\hat{\boldsymbol{y}} = f^2(\boldsymbol{z}) = \boldsymbol{W}^2\boldsymbol{z} + \boldsymbol{b}^2. \tag{B.12}$$

Thus, a fully-connected layer linearly combines an input tensor with learned parameters: a weights tensor (*i.e.* $\boldsymbol{W}^1$ and $\boldsymbol{W}^2$) and a bias tensor (*i.e.* $\boldsymbol{b}^1$ and $\boldsymbol{b}^2$).

As in this example, the input and output tensors of a fully-connected layer are most commonly vectors (*i.e.* 1st-order tensors); thus it follows that the weights tensor is a matrix (*a.k.a.* weights matrix) and the bias tensor is a vector (*a.k.a.* bias vector or bias term).

General form. Now, we can write a general function $f_{\text{fc}} : \mathbb{R}^M \to \mathbb{R}^N$ with weight matrix $\boldsymbol{W} \in \mathbb{R}^{M\times N}$ and bias term $\boldsymbol{b} \in \mathbb{R}^N$ to describe a fully-connected layer that takes as input M-D vectors and outputs N-D vectors:

$$f_{\text{fc}}(\boldsymbol{x}) = \boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}. \tag{B.13}$$

By definition, the j-th element of the output tensor $\boldsymbol{z} = f_{\text{fc}}(x)$ is given by

$$z_j = (\sum_{i=1}^{M} x_i \cdot w_{i,j}) + b_j. \tag{B.14}$$

Working out an example. Finally, let us work out an example. Let us define the weight matrices in fig. B.1 as follows:

$$\boldsymbol{W}^1 = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \end{bmatrix}, \boldsymbol{W}^2 = \begin{bmatrix} 0 & 1 \\ 1 & 2 \\ 2 & 3 \\ 3 & 4 \end{bmatrix}, \tag{B.15}$$

and let both bias vectors be filled with 1. Then, the transposed weight matrices are as follows (it's easier to reason with the transposed matrices):

$$\boldsymbol{W}^{1T} = \begin{bmatrix} 0 & 1 & 2 \\ 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix}, \boldsymbol{W}^{2T} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 2 & 3 & 4 \end{bmatrix}. \tag{B.16}$$

Now, consider the input vector $\boldsymbol{x} = \begin{bmatrix} -2 & 0 & 1 \end{bmatrix}$.

Using the above equations, we compute the intermediate tensor $\boldsymbol{z}$ to be as follows:

$$\boldsymbol{z} = \begin{bmatrix} -2 \cdot 0 + 0 \cdot 1 + 1 \cdot 2 \\ -2 \cdot 1 + 0 \cdot 2 + 1 \cdot 3 \\ -2 \cdot 2 + 0 \cdot 3 + 1 \cdot 4 \\ -2 \cdot 3 + 0 \cdot 4 + 1 \cdot 5 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 0 \\ -1 \end{bmatrix}, \tag{B.17}$$

and the output tensor $\hat{\boldsymbol{y}}$ to be as follows:

$$\hat{\boldsymbol{y}} = \begin{bmatrix} 2 \cdot 0 + 1 \cdot 1 + 0 \cdot 2 + -1 \cdot 3 \\ 2 \cdot 1 + 1 \cdot 2 + 0 \cdot 3 + -1 \cdot 4 \end{bmatrix} = \begin{bmatrix} -2 \\ 0 \end{bmatrix}. \tag{B.18}$$

B.2.1.2 Convolutional layers

In contrast to a fully-connected layer, which applies a unique weight to every input neuron, a **convolutional layer** applies the same set of weights (*a.k.a. convolutional filters* or *convolutional kernels*) to "neighborhoods" of input neurons. This property of sharing weights makes convolutional layers well-suited for handling visual information, as each filter can be tuned to recognize a particular pattern (*i.e.* oriented edges, cat ear).

1D convolution. To build intuition, let us consider a simple example of a 1D convolution.[11]

In this example, the same weights vector $\boldsymbol{w}$ is applied to small neighborhoods comprised of 3 input neurons each (*i.e.* (x_1, x_2, x_3), (x_2, x_3, x_4), and (x_3, x_4, x_5)) in a "sliding window" fashion. To compute the value of an output neuron (*i.e.* z_2), for every input neuron connected to it, multiply its value with the value of its connecting weight (*i.e.* given by the line color), and sum up the products as follows:

$$z_i = x_i \cdot w_1 + x_{i+1} \cdot w_2 + x_{i+2} \cdot w_3. \tag{B.19}$$

[11] Here, the dimensionality refers to the number of ways the weight moves. In this case, it can only move along one direction (*i.e.* left-to-right). A 2-D convolution is used for images and can move along 2 dimensions (*i.e.* up-and-down and left-to-right).

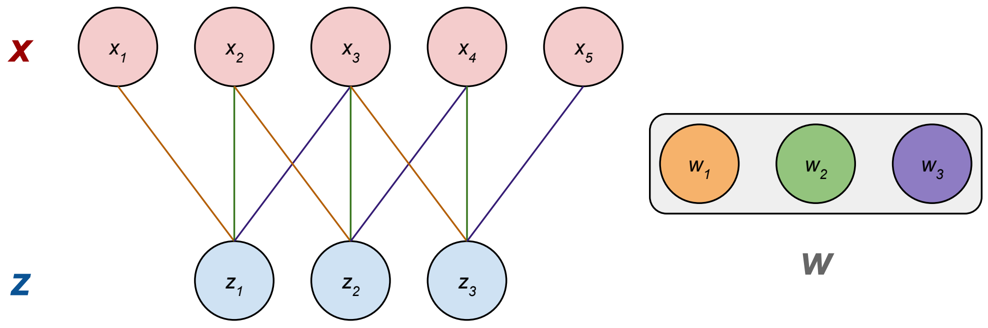

Figure B.2: 1D convolution. Notice how weights are re-used (i.e., line colors are repeated) and applied in a "sliding window" fashion.

Now, let's work out an example. Suppose the input and weight vectors are given as follows:

$$\boldsymbol{x} = \begin{bmatrix} 1 & 6 & 5 & 4 & 1 \end{bmatrix}, \boldsymbol{w} = \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}, b = 0. \tag{B.20}$$

Then, the resulting output vector is

$$\boldsymbol{z} = \begin{bmatrix} 1 \cdot -1 + 6 \cdot 0 + 5 \cdot 1 \\ 6 \cdot -1 + 5 \cdot 0 + 4 \cdot 1 \\ 5 \cdot -1 + 4 \cdot 0 + 1 \cdot 1 \end{bmatrix} = \begin{bmatrix} 4 \\ -2 \\ -4 \end{bmatrix}. \tag{B.21}$$

This particular convolutional filter detects the presence and orientation of 1D "edges" (*i.e.* change in value within a neighborhood of neurons): A positive output neuron denotes an input neighborhood where the leftmost input neuron is smaller than the rightmost input neuron (*i.e.* value increases when comparing the left neuron with the right neuron), a negative output neuron denotes the reverse (*e.g.* value decreases), and the magnitude of the output neuron denotes the magnitude of the difference in values between the two outer input neurons.

2D convolution. For images, a 2D convolution is typically used, as images are two-dimensional. Similarly, to compute a value in the output tensor, consider the dot product between a convolutional filter (*i.e.* $\boldsymbol{w} \in \mathbb{R}^{3 \times H_k \times W_k}$) and a same-sized neighborhood in the input tensor.

In this example, we show one filter and the resulting output tensor slice (*i.e.* 3D-tensor with a channel depth of 1). Typically, a convolutional layer applies more than one filter. Because every filter produces an output tensor slice, the number of channels in the output tensor is equal to the number of filters used.

Thus, a 2D convolutional layer can be defined as $f_{\text{2DConv}} : \mathbb{R}^{C_i \times H_i \times W_i} \rightarrow \mathbb{R}^{C_o \times H_o \times W_o}$ with weights tensor $\boldsymbol{W} \in \mathbb{R}^{C_o \times C_i \times H_k \times W_k}$ and, if used, bias tensor $\boldsymbol{b} = \mathbb{R}^{C_o}$.[12]

[12]For conciseness and clarity, we do not use a bias tensor in the convolution examples.

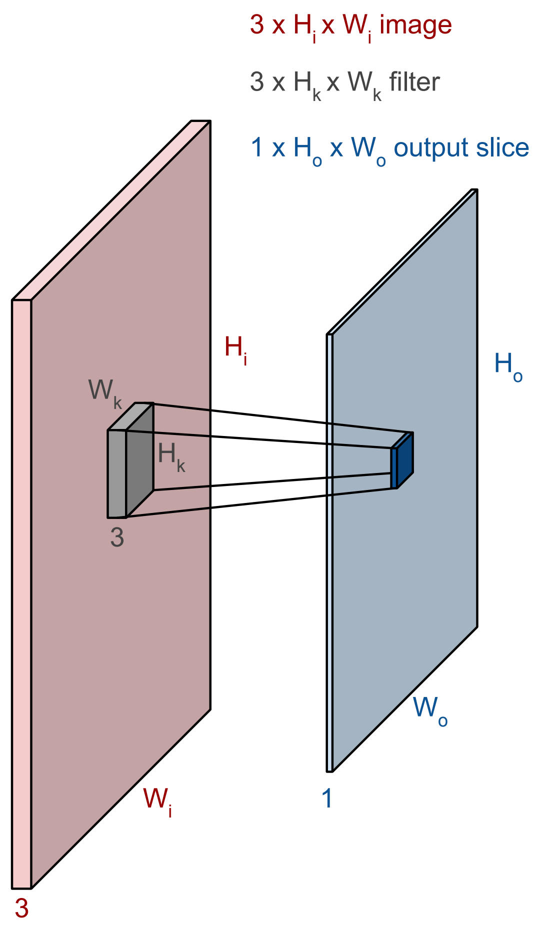

Figure B.3: 2D convolution.

Then, given a neighborhood in the input tensor that has the same spatial dimensions as the filters, *i.e.* $\boldsymbol{x}' \in \mathbb{R}^{C_i \times H_k \times W_k}$, its corresponding output value can be computed as follows:

$$z_k = (\sum_{c=1}^{C_i} \sum_{i=1}^{H_k} \sum_{j=1}^{W_k} w_{k,c,i,j} \boldsymbol{x}'_{k,c,i,j}) + b_k, \tag{B.22}$$

where $z_k \in \mathbb{R}$ is the output value that corresponds to applying the k-th filter to the neighborhood $\boldsymbol{x}'$

In addition to filter size and the number of filters, there are a few other hyperparameters for convolutional layers. *Stride* refers to the step size with which to "slide" the filter across the input tensor. *Padding* refers to adding additional input values (*i.e.* "padding" the input). The most common value with which to pad an input tensor is 0 and is known as *zero-padding*. In the examples shown here, we set the stride to be 1 (*i.e.* we shift the filter by an increment of 1) and use no padding. In the following examples, we modified the earlier 1D example (fig. B.2) first to use zero-padding with a padding width of 1 (*i.e.* padding with 1 extra value on each side) and second to use a stride of 1.

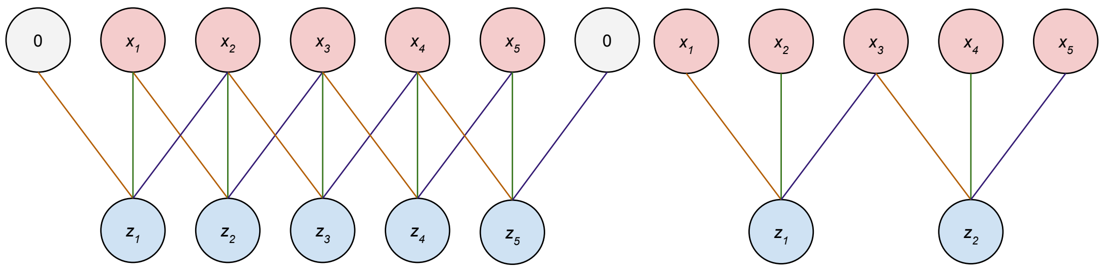

Figure B.4: More 1D convolution examples. Left: An example with padding width of 1 on each side. **Right:** An example with a stride of 2.

Now, we can write a formula for calculating the spatial size of the output tensor. To calculate the length of one output dimension O, we can use the following:

$$O = \frac{I - K + 2P}{S} + 1, \tag{B.23}$$

where I is the input size, K is the length of the filter, S is the stride step size, and P is the padding width, all along the same corresponding dimension. If the result is a fraction, round up.

B.2.2 Other layers

In addition to layers that linearly combine tensors with learned weights, there are several kinds of layers that enable a CNN to filter information such that only relevant features are propagated.

B.2.2.1 Activation layers

An activation layer typically applies a *non-linear*,[13] scalar function $g : \mathbb{R} \rightarrow \mathbb{R}$ to every element in the input tensor, thereby outputting a tensor of the same size.

The most common activation function used in modern CNNs is the **ReLU** function, which stands for *rectified linear unit*, and is defined as follows:

$$g(x) = \max(x, 0). \tag{B.24}$$

In practice, the ReLU function allows a CNN to disgard non-relevant information by setting all negative elements in an input tensor to 0. Because activation layers

[13] A linear relationship is one in which changes in the output are directly proportional to changes in the input. For instance, the function $f(x) = 2x$ is linear while the function $f(x) = \max(x, 0)$ is non-linear. This is because there are times when x varies (*i.e.* when $x < 0$) and the variation in the input is not reflected in the output.

typically immediately follow linear layers, a CNN can adapt filters in the preceding linear layer to fire positively when they recognize relevant features. Then, the following ReLU layer could "forget" neurons that captured irrelevant features.

Now, we can take a look at a simple example. Given an input tensor

$$\boldsymbol{x} = \begin{bmatrix} -2 & 1 & 2 \\ 0 & -1 & 1 \\ 3 & -1 & 2 \end{bmatrix}, \tag{B.25}$$

the result of passing it through a ReLU layer would be as follows:

$$\boldsymbol{z} = \begin{bmatrix} 0 & 1 & 2 \\ 0 & 0 & 1 \\ 3 & 0 & 2 \end{bmatrix}. \tag{B.26}$$

B.2.2.2 Pooling layers

A pooling layer forces information to be compressed spatially. Similar to 2D convolutions, which operate on spatial neighborhoods of elements in an input tensor, most pooling layers operate on neighborhoods in the input.

The **max pooling** operation chooses the maximum value from a set of inputs (*i.e.* spatial neighorhood) as the output value, while **average pooling** sets the output value as the average (*i.e.* mean) of a set of input values.

Because pooling layers are typically used to compress information *spatially*, they usually leave the channel dimension unchanged and apply pooling functions (*i.e.* max or average pooling) to spatial neighborhoods of tensor slices, that is, they are applied to every channel independently.

Now, we can define the pooling functions precisely. Given $\boldsymbol{x}' \in \mathbb{R}^{H_k \times W_k}$, a tensor slice representing a spatial neighborhood, max pooling and average pooling are defined as

$$g_{\text{maxpool}}(\boldsymbol{x}') = \max_{i=1}^{H_k} \max_{j=1}^{W_k} x'_{i,j} \text{ and } g_{\text{avgpool}}(\boldsymbol{x}') = \frac{1}{H_k \cdot W_k} \sum_{i=1}^{H_k} \sum_{j=1}^{W_k} x'_{i,j}. \tag{B.27}$$

Finally, let's work through a simple example.

Given an input tensor slice as follows:[14]

$$\boldsymbol{x} = \begin{bmatrix} -2 & 1 & -3 & 4 \\ 3 & -2 & 0 & -1 \\ 2 & 1 & -1 & -2 \\ 0 & -2 & 1 & -3 \end{bmatrix}. \tag{B.28}$$

[14]For simplicity, we are considering an input tensor with only one channel. For tensors with more than one channel, the same procedure would be applied to every channel in the input tensor.

The result of passing it through a max pooling layer that uses a 2×2 neighborhood is as follows:

$$\boldsymbol{z} = \begin{bmatrix} 3 & 1 & 4 \\ 3 & 1 & 0 \\ 2 & 1 & 1 \end{bmatrix}. \tag{B.29}$$

B.2.2.3 Regularization layers

There are also a number of layers that regularize the activation tensor (*e.g.* via dropout, batch normalization, etc.). For brevity, we only describe dropout.

A **dropout layer** randomly "drops" (*i.e.* sets to 0) input values and can be described as applying the following function $g : \mathbb{R} \rightarrow \mathbb{R}$ element-wise (*i.e.* to every input value):

$$g(x) = \begin{cases} x & p \leq 0.5, \\ 0 & \text{otherwise}, \end{cases} \tag{B.30}$$

where p is a unique random number generated afresh every time the function is applied. The result is that approximately half of the activation tensor is "dropped". This simple technique forces the model not to be too dependent on any one feature, as it will be dropped half of the time. This tends to improve a model's robustness (*i.e.* ability to perform well under a variety of conditions) and overall performance.

B.2.3 Putting it all together

Now that we've discussed various components of a CNN and aspects of a deep learning set-up, let's put this knowledge all together.

B.2.3.1 Model architecture

A CNN **architecture** refers to the specific configuration of layers and their settings (*i.e.* filter size, number of output channels, stride, padding).

In fig. B.5, we show a diagram detailing the AlexNet (Krizhevsky et al., 2012) architecture, which was one of the earliest demonstrations of the power of CNNs on object classification. This model uses around 61 million parameters (*i.e.* total number of scalars in all weight and bias tensors). Because a CNN is typically highly-parameterized, it is not feasible to understand a model simply by examining its parameters, due to the sheer volume of them.

Other popular CNN architectures include VGG networks (*e.g.* VGG16) (Simonyan et al., 2015), GoogLeNet (Szegedy et al., 2015), and residual networks (*e.g.* ResNet50) (He et al., 2016).

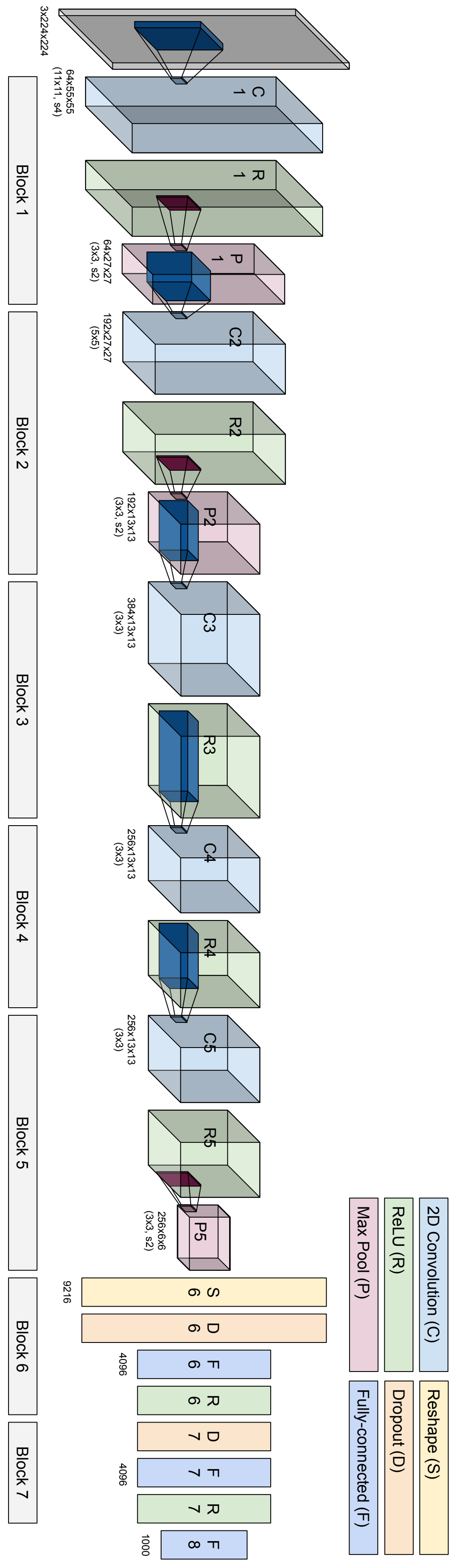

Figure B.5: AlexNet architecture.