Notes on Convolutional Neural Networks

Carl Binding*

March 11, 2020

Abstract

These notes describe, at an engineering level, some key points of implementing a *convolutional neural network* (CNN). Such CNNs nowadays are extensively used in various applications such as image recognition or text processing. Implementation of these networks is a relatively straightforward task once the basic principles are understood. Some of these principles are, however, not easily found in the literature and the purpose of this report is to write-up some of the fundamental operations in implementing convolutional neural networks from the ground up.

1 Introduction

Convolutional neural networks (CNN) are in high-demand for *machine-learning* applications. Although they are based on somewhat older concepts, they have gained new attraction due to the incredible increase in hardware performance where parallel computing [2], stream oriented hardware extensions [18], and highly parallelized *graphical unit processors* (GPU) [14] can be used to achieve unprecedented performance for larger size neural networks and their inputs.

Convolutional neural networks are extensions of more conventional neural networks, which are called *fully-connected neural networks* (FC). The literature on FC neural networks is abundant. A theoretical introduction to some of the concepts can be found in [1]. A detailed explanation with a corresponding Python implementation is available on-line [12].

A good introduction to CNNs is available on-line under [17], which omits however some of the implementation details. A rather mathematically oriented background book with terse notation is [8]. Chollet gives a very hands-on introduction to the use of CNNs using the popular Keras framework [7, 5], but skims over some of the intricacies of the technology. More details can be found in the concise write-up on convoluational neural networks by Bouvrie [3]. Full details on the back-propagation algorithms can be found in [9].

The basic concepts of neural networks, fully-connected or convolutional, include the use of *multi-dimensional arrays*, also known as *tensors* of which we however ignore the deeper algebraic properties. Thus, for us, a tensor simply represents a multi-dimensional array of floating point numbers. A tensor of dimension 0 is a *scalar*, of dimension 1 a *vector*, and two-dimensional tensors are known as *matrices*. Higher dimensions are used to hold image data: a three dimensional tensor typically holds the Red-Green-Blue (RGB)

*Im Gafos 19, LI-9494 Schaan, carl.binding@sunrise.ch

planes of an image, thus being a 3-D tensor with a depth of three and width and height representing the horizontal and vertical dimensions of the image. Black-and-white images are 3D tensors with a depth of 1. Tensors for video data would be 4-D adding time as a fourth dimension. See [5] for a good visualization of these data structures.

The necessary functionality for tensor manipulation can be found in packages such as Python's *numpy* [13] environment with some additional functionality typically found in signal-processing extensions such as Python's *scikit* [16] packages. These libraries are generally based on optimized C or FORTRAN [11, 6] libraries which perform the compute intensive operations. In some cases, computations are migrated to GPUs for additional performance [14] - the payoff however varies according to the size of the tensors being copied between CPU and GPU.

Implementing a tensor library is straightforward, due to the regularity of the tensor data structure. Performance however is important and careful and correct optimizations can be cumbersome and time-consuming.

As implied by the name, a key component in convolutional networks is the *convolution*. Technically correct, however, CNNs are in fact using *correlations* - convolutions are related to correlations by *flipping* the *kernel* which can be seen as a 180° rotation of a 2D, symmetric, tensor.

Mathematically a two-dimensional correlation is defined as follows. Assume we have an *image* I with a *height* and *width* of (h_I, w_I). Assume we have another 2D tensor, named the *filter* F with height and width of (h_F, w_F). Frequently we have $h_F = w_F$, i.e. filters are symmetric and some caeses must be of odd height and width, with typical values of 3 or 5. The result of the correlation shall be called C and is also a 2D tensor with however a different height and width (h_C, w_C) compared to the dimensions of the image I or the filter F and depending on zero-valued *padding* around the image as well as the motions of the filter, called *stride*.

Ignoring padding and striding, we can now formulate the equation for a 2D correlation:

$$\begin{aligned} C &= I \otimes F \\ C(i,j) &= \sum_{m=0}^{h_F-1} \sum_{n=0}^{w_F-1} I(i+m, j+n) \cdot F(m,n) \end{aligned} \tag{1.1}$$

Note that indices range from 0 to the corresponding dimension's extent minus one; following the convention of many programming languages[1].

The corresponding equation for a 2D convolution is similar:

$$\begin{aligned} C &= I * F \\ C(i,j) &= \sum_{m=0}^{h_F-1} \sum_{n=0}^{w_F-1} I(i+m, j+n) \cdot F(h_F-m, w_F-n) \end{aligned} \tag{1.2}$$

We simply have reversed ("flipped") the indices over the filter. Note that in the literature the distinction between the use of correlation and convolution is often loose. Most popular frameworks do supply both, a 2D correlation and a 2D convolution and the misuse of the term "convolutional" is probably a historical mistake. We thus use correlation consistently beyond that point (see also [8] for some discussion on the topic).

The 2D correlation process can be visualized as sliding the filter F horizontally and vertically over the image I and performing a *dot-product* between the re-arranged entries

[1] With apologies to N. Wirth.

of the filter into a vector with the overlapped, also rearranged, entries of the image. We slide the filter one pixel at a time, i.e. stride $S = 1$; the literature also describes the process when the *stride* is greater than one. See figure 1.1 for an example.

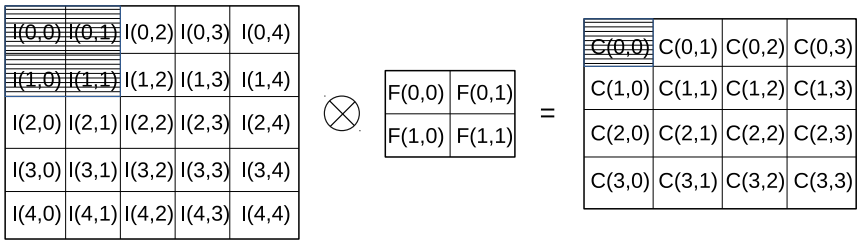

Figure 1.1: Correlation of a 5×5 image with a 2×2 filter

Note that the output C is smaller than the image I - our sample kernel only fits four times over the 5×5 image when using a stride of one. This can be remedied by applying *padding* i.e. surrounding the image with zero-valued pixels. The dimensions of the output C can be computed as follows:

$$\begin{aligned} h_C &= (h_I - h_F + 2P)/S + 1 \\ w_C &= (w_I - w_F + 2P)/S + 1 \end{aligned} \tag{1.3}$$

where P is the amount of padding around the image I and S the size of the strides, assuming that both P and S are identically horizontally and vertically. Generally, we also have $h_F = w_F$ and odd.

Depending on the size of the paddings and assuming $S = 1$, we have the following correlation *modes*[2]:

- *valid*: No padding of the image I. The output C dimensions are the corresponding dimensions of the image minus the filter's dimension plus one: $h_C = h_I - h_F + 1$ and $w_C = w_I - w_F + 1$.
- *same*: The padding is (roughly) half of the filter's width and height of zero valued pixels around the image. In that case, the output C would have the same height and width as the image I when $S = 1$, i.e. $h_C = h_I$ and $w_C = w_I$. For identical filter height and width, we have $P = (h_F - 1)/2 = (w_F - 1)/2$ which yields integer values when h_F and w_F are odd.
- *full*: The image is padded with a frame of $h_F - 1$ and $w_F - 1$ zero valued pixels. The output C has dimensions $h_C = h_I + h_F/2 + 1$ and $w_C = w_I + w_F/2 + 1$ and thus larger than the input I.

As visualized in figure 1.1 the filter F when positioned at position $(0, 0)$ of the image I generates an output pixel $C(0, 0)$:

$$\begin{aligned} C(0,0) &= I(0,0) \cdot F(0,0) + I(0,1) \cdot F(0,1) + I(1,0) \cdot F(1,0) + I(1,1) \cdot F(1,1) \\ &= \sum_{m=0}^{1} \sum_{n=0}^{1} I(0+m, 0+n) \cdot F(m,n) \\ &= [I(0,0), I(0,1), I(1,0), I(1,1)] \cdot [F(0,0), F(0,1), F(1,0), F(1,1)]^T \end{aligned} \tag{1.4}$$

[2]To borrow Matlab terminology which appears to be widely accepted.

in which the last line represents the dot-product between a row-vector based on the image I and the column-vector based on the filter F. Implementations of this dot-product are key to the performance of CNNs during training - during which the filters (weights) of convolutional layers are tuned - and classifications which then use the trained filters.

Note that correlation and convolution are well-known techniques and various implementation speed-ups are known: the *im2col* mapping [17], the Fast Fourier Transforms (FTT) based implementation [15] or the Winograd convolution [10, 19]. However, performance also depends on the sizes of the images and filters and the underlying hardware characteristics, i.e. memory bandwidths and performance and precision of the arithmetic operations. It is thus no wonder that dedicated hardware is being built to optimize performance of 2D correlations, also knowing that over 60 % of computation time during CNN training is spent for correlation.

Similar to FC neural networks, CNNs are organized in *layers*. Layers, numbered from $0..(L-1)$ with $L-1$ indexing the *output* layer, can be of different nature:

- *Correlation layer*: These layers take an 3D tensor of size (d_I, h_I, w_I)[3] and apply correlation against c channels where each channel represents one filter F of size (d_F, h_F, w_F). Note that in general $h_F = w_F$ and that $d_F = d_I$, i.e. the depth of the filters F match the depth of the input I. For each channel, we obtain one output feature map C of depth 1 and, using *valid* correlation, of height $h_C = h_I - h_F + 1$ and width $w_C = w_I - w_F + 1$. This is illustrated in figure 1.2.

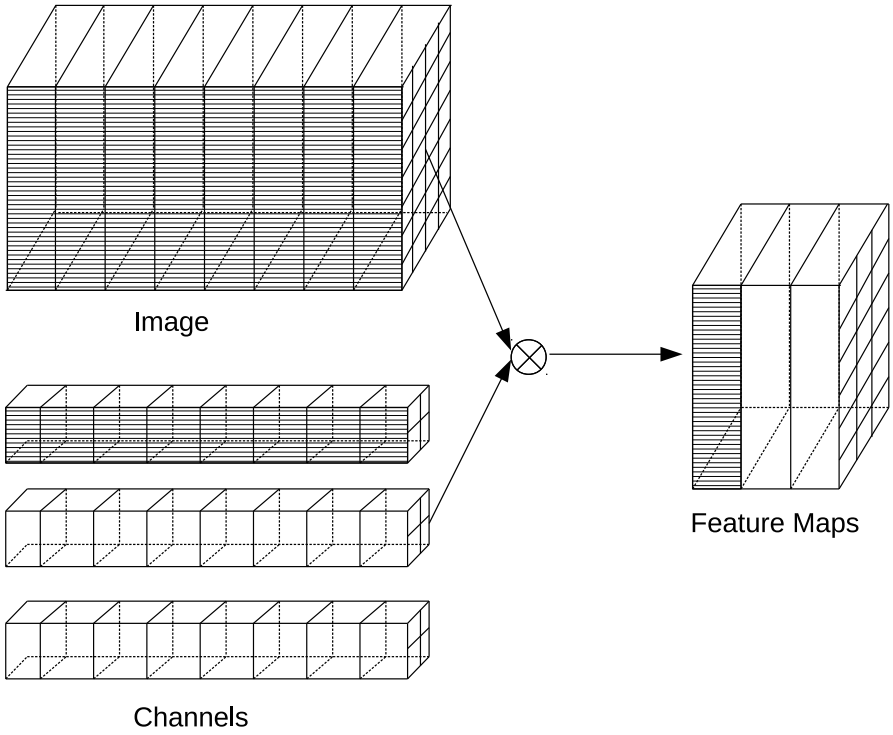

Figure 1.2: Correlation layer in neural network

[3]We use 3D shapes in the (z, y, x) dimensions for depth, height and width of a 3D tensor.

The output map is then run through an *activation* function which is one of the well-known mathematical functions such as *sigmoid, tanh, RELU*. These range between $0 \ldots 1$ or $-1 \ldots 1$ and their derivatives are mostly well defined[4]. The size of the tensor does not change during activation; it is determined by the correlation.

- *Pooling layer*: The pooling layer does a 2D reduction in the height and width dimensions of a 3D input tensor; the depth remains unaffected (see figure 1.3). Pooling choses a square area of an input slice of dimensions $P \times P$ and selects either the maximum value over this square, the average value over the pooling square, or an L2-norm value over the pooling area. The output size of the pooling layer is thus a height of $h_C = h_I/P$ and a width $w_C = w_I/P$. (Here we assume that the input height and width dimensions are even.)

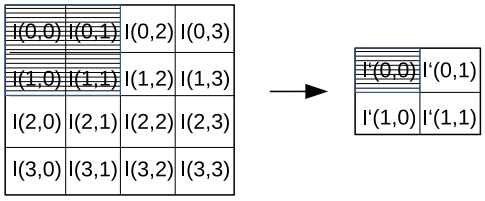

Figure 1.3: Pooling operation

We do not use activation functions associated with pooling layers - although some authors have proposed such.

- *Flattening layer*: The flattening layer maps a 3D tensor resulting from either a correlation or a pooling layer onto a 1D tensor (e.g. *resize*). This 1D tensor is then fed into the fully connected layers via a weight matrix W as for regular fully-connected layers..

- *Fully-connected layers* (FC): These map the output of their upstream layer onto a pre-activation value a using a weight matrix (2D tensor) W^{l-1} which we associate with the up-stream layer. We have $a^l = W^{l-1} \cdot y^{l-1} + b^l$ where l is the index of the layer in the network, $y^{l-1} = x^l$ is the output of the upstream layer (1D tensor) and thus the input x to layer l and b^l the layer's bias. Applying activation to a^l, we have $y^l = f(a^l)$ where f is one of the well-known activation functions.

  Multiple fully-connected layers can be present until finally reaching the output layer with index $(L-1)$ and $y^{L-1} = f(a^{L-1}) = W^{L-2} \cdot y^{L-2} + b^{L-1}$. Truth values are then compared against y^{L-1} and the *loss* (or its inverse, the *cost*) function is evaluated.

  In terms of notation we use y to denote an *output* of a layer, x for *input*, and a for the *pre-activation* values which are linear combinations of weights and input.

Correlation layers can alternate with pooling layers; these are separated from the fully-connected layers via a single flattening layer.

Given the indexing of layers, we also talk about *upstream* and *downstream* layers, following the data-flow of the *forward* training phase, i.e. data flowing from lower to higher numbered indices.

[4]The exception being RELU at 0, for which the derivative is defined to be equal to 0.

The *cost* function reflects the performance of the neural network. How close is the output value, which is a 1D tensor, to the true value? There are various cost functions used in practice. Most common is to use an L2 norm based difference between output and true value: this can be differentiated and optimized conveniently.

During training we use N samples for which the truth values belong to some pre-defined classes k, i.e. t_{nk} to indicate the truth value of sample n belonging to class k. This is thus typically a scalar ($\mathbb{N}_0$) which for a fixed set of classes K can be encoded as so-called *one-hot* vectors $\mathbf{t}_{nk}$ in which bit position k is set to 1 and all other $K-1$ bit positions are set to 0, i.e. in the vector $\mathbf{t}_{nk}$ bit position $k = 0 \dots K-1$ is 0 or 1.

The network's output at layer $L-1$ is a vector $\mathbf{y}_n \in \mathbb{R}^K$ of length K in which the value at positionn $k = 0 \dots K-1$ can be considered as a probability that the input belongs to class k. A high value of y_{nk} indicates that with some higher probability (or likelihood) the input belongs to class k.

The total error $E_N \in \mathbb{R}$ after all samples have been processed is defined as

$$E_N = \frac{1}{2} \sum_{n=0}^{N-1} \sum_{k=0}^{C-1} (t_{nk} - y_{nk})^2 \tag{1.5}$$

and for every sample n the corresponding error is

$$E_n = \frac{1}{2} \sum_{k=0}^{C-1} (t_{nk} - y_{nk})^2 = \frac{1}{2} \|(\mathbf{t}_n - \mathbf{y}_n\|_2^2 \tag{1.6}$$

i.e. the squared norm of the differences between truth value $\mathbf{t}_n$ and the network's outcome $\mathbf{y}_n$.

The trainable values in such a CNN are the values of filters in the correlation layers and the weight matrices in the fully-connected layers. Depending on the input dimensions and the number of channels, as well as the number of nodes in the FC layers, large numbers of weights must be tuned during the training of a CNN.

The *forward* phase of the training is thus relatively straightforward, once the layers are interconnected and initial random values for filters and weight matrices are chosen: correlations, activations, poolings, and matrix multiplications in the FC layers are the key operations using multi-dimensional arrays as the underlying data structure.

2 Backpropagation

Backpropgation is concerned with using the discrepancy between the network's output $\mathbf{y}_n$ and the truth value associated with sample n, $\mathbf{t}_n$. The aim is to adjust the network's weights such that the error is minimized.

The error minimization is based on *gradient descent* [4], that is weights are adjusted along the gradient of the error or cost function versus the weights $\frac{\delta E}{\delta W}$. Successively adjusting the weights will lead us to the minimum of the error function, i.e. where its gradient becomes zero when assuming the error function is concave and has a minimum.

2.1 Fully connected layers

Assume a network layer l in which the output y_k^l at position k is a linear combination of the layer's input variables x_i^l

$$y_k^l = \sum_i w_{ki}^{l-1} x_i^l \tag{2.1}$$

where w_{ki}^{l-1} is the weight connecting input x_i^l with the output y_k^l [5]. Note that we have $x_i^l = y_i^{l-1}$. i.e. the input at layer l is the output of layer $l-1$.

In matrix form we can write

$$\mathbf{y}^l = \mathbf{W^{l-1}}\mathbf{y^{l-1}} = \mathbf{W^{l-1}}\mathbf{x^l} \tag{2.2}$$

The weight matrix $\mathbf{W^{l-1}}$ is of shape $(K \times I)$ with K the number of output nodes at layer l and I the number of output nodes at layer $l-1$. Column vector $\mathbf{y^{l-1}}$ has length I and column vector $\mathbf{y^l}$ is of length K.

Using the error function 1.6 and the chain derivation rule at the final output layer $L-1$, we have

$$\frac{\delta E_n}{\delta w_{ki}^{L-2}} = (y_{nk}^{L-1} - t_{nk})x_{ni}^{L-1} \tag{2.3}$$

for a weight w_{ki}^{l-2} connecting input node x_i^{l-1} with output value y_k^{l-1}, n indicating the n-th sample.

This can be split into an *error term* $y_{nk}^{L-1} - t_{nk}$ and the input variable x_{ni}^{L-1} linked by the weight w_{ki}^{L-2}.

In a more general feed-forward network, we have the *activation* functions to consider. We denote the pre-activation values with a_k which are the linear combination of the up-stream layer's output y_i^{l-1} using weights w_{ki} at layer $l-1$ connecting nodes i to k:

$$a_k^l = \sum_i w_{ki}^{l-1} y_i^{l-1} \tag{2.4}$$

The pre-activation value a_k at layer l is transformed using an activation function $h(\cdot)$ i.e.

$$y_k^l = h(a_k^l) = h(\sum_i w_{ki}^{l-1} y_i^{l-1}) \tag{2.5}$$

We can use the chain rule for derivations to write

$$\frac{\delta E_n}{\delta w_{ki}^{l-1}} = \frac{\delta E_n}{\delta a_k^l}\frac{\delta a_k^l}{\delta w_{ki}^{l-1}} = \delta_k^l \frac{\delta a_k^l}{\delta w_{ki}^{l-1}} \tag{2.6}$$

introducing the error value

$$\delta_k^l = \frac{\delta E_n}{\delta a_k^l} \tag{2.7}$$

i.e. the derivative of the n-th sample error E_n against the pre-activation value a_k^l.

We note that $\frac{\delta a_k^l}{\delta w_{ki}^{l-1}} = y_i^{l-1}$ since a_k^l is a linear combination of y_i^{l-1} and w_{ki}^{l-1}s. Thus we can write

$$\frac{\delta E_n}{\delta w_{ki}^{l-1}} = \delta_k^l y_i^{l-1} \tag{2.8}$$

a simple formula for the derivative $\frac{\delta E_n}{\delta w_{ki}^{l-1}}$.

We can vectorize the notation for the Jacobian ∇E_w

$$\nabla E_w = \delta^l \cdot \mathbf{y}^{l-1} \tag{2.9}$$

using a matrix multiplication between the $(K \times 1)$ column vector δ^l and the $(1 \times I)$ row vector $\mathbf{y}^{l-1}$ yielding the $(K \times I)$ Jacobian matrix ∇E_w.

[5] The first index indicating the weight's target, the second index indicates the weight's source node.

For the output layer $L-1$ and node k we have

$$\delta_k^{L-1} = y_k^{L-1} - t_k \tag{2.10}$$

For hidden, fully-connected layers, we make use of the chain rule for partial derivatives:

$$\delta_i^l = \frac{\delta E_n}{\delta a_i^l} = \sum_k \frac{\delta E_n}{\delta a_k^{l+1}} \frac{\delta a_k^{l+1}}{\delta a_i^l} \tag{2.11}$$

where the summation index k runs over all the units k at down-stream layer $l+1$ to which unit i at layer l sends its value (see appendix A for a detailed derivation of δ in hidden layers.).

Using $\frac{\delta E_n}{\delta a_k^{l+1}} = \delta_k^{l+1}$ and

$$\frac{\delta a_k^{l+1}}{\delta a_i^l} = \frac{\delta(\sum_k w_{ki}^l h(a_i^l))}{\delta a_i^l} = w_{ki}^l h'(a_i^l) \tag{2.12}$$

since in the sum over k only the term for a_i^l is relevant, we find the following backpropagation formula

$$\delta_i^l = h'(a_i^l) \sum_k w_{ki}^l \delta_k^{l+1} \tag{2.13}$$

with w_{ki}^l connecting node i at layer l to down-stream node k layer $l+1$.

For multiple nodes in a layer, we can write a vectorized notation of this formula:

$$\delta^{\mathbf{l}} = ((\mathbf{W^l})^{\mathbf{T}} \delta^{\mathbf{l+1}}) \circ \mathbf{h}'(\mathbf{a^l}) \tag{2.14}$$

with $\mathbf{W^l}$ the weight matrix connecting layer l to layer $l+1$ and of dimensions $(K \times I)$ and thus dimensions $(I \times K)$ for $(\mathbf{W^l})^{\mathbf{T}}$ where I is the number of output nodes in layer l and K the number of output nodes in layer $l+1$. $\delta^{\mathbf{l+1}}$ has the output dimension of layer $l+$ and thus is of length K. The result of $(\mathbf{W^l})^{\mathbf{T}} \delta^{\mathbf{l+1}}$ is then a $(I \times 1)$ column vector which is element-wise multiplied (e.g. Hadamard product) with $h'(\mathbf{a^l})$ also of length I.

Evidently these operations are straightforward to implement using tensor operations.

2.2 Correlation Layers

In the correlation layer of a CNN the input feature maps are correlated with one or multiple kernels (filters) - which hold the trainable weights - and run through some activation function h to generate the output feature maps. The input feature maps are of the same depth as the learnable kernels. Each kernel generates a single output feature map, see figure 1.2.

We can write

$$\mathbf{y}_j^l = h\left(\sum_{i=0}^{d_I} \mathbf{y}_i^{l-1} \otimes \mathbf{k}_{i,j}^l + b_j^l\right) = h\left(\mathbf{a}_j^l\right) \tag{2.15}$$

where the index j indicates the j-th channel (filter) k generating the output plane $\mathbf{y}_j^l$ at network layer l. j ranges over all chanels i.e. $0 \ldots (c-1)$. $\mathbf{y}_i^{l-1}$ is the i-th output plane of the up-stream layer $l-1$ which is correlated with mode *valid* with the j-th kernel $\mathbf{k}_{i,j}^l$ at layer l, with i ranging across the depth of the image d_I which equals the depth $d_K$of the c kernels. Note that we write $\mathbf{a}_j^l$ for the pre-activation tensor at correlation layer l, which is the result of the correlation and addition of the bias b_j^l. We denote the shape of the output feature map $\mathbf{y}_j^l$ as $(1, h_{\mathbf{y}^l}, w_{\mathbf{y}^l})$. Note that the activation function h is applied on the output of the correlation.

2.2.1 Computing the Gradients in Correlation Layers

We assume that each correlation layer is followed by either a down-sampling (pooling) layer or a flattening layer. To compute the error terms δ_j^l in case of an up-stream pooling layer, we need to consider the errors at layer $l+1$ which are *up-sampled* to match the dimension of layer l's output feature maps.

Note that we associate one plane of the δ tensor with each of the c kernels (equivalently with each output plane) and its shape matches the shape of the correlation's output feature maps and, in the z dimensio, has depth c.

The up-sampling operation is implemented depending on the precise nature of the down-sampling (pooling) and can be accelerated by using appropriate caching of indices (for max-pooling) or values (for L2 pooling) during the forward pass. In the case of max-pooling, the down-stream error is back-propagated into the cell corresponding to the maximum value in the output feature map at layer l, with other values set to 0. For average-pooling, the down-stream error is evenly distributed into all the up-stream cells. For L2-norm pooling, the error is distributed proportionally to each up-stream cell's contribution.

Taking into account the derivative of the activation function h' applied to the pre-activation values $\mathbf{a}_j^l$, we can write

$$\delta_j^l = h'(\mathbf{a}_j^l) \circ \text{up}(\delta_j^{l+1}) \tag{2.16}$$

The shape of sub-tensor δ_j^l is $(1, h_{\mathbf{y}^l}, w_{\mathbf{y}^l})$ with $j \in [0 \dots (c-1)]$, i.e. it is identical with the output shape of the correlation layer.

For the bias b_j associated with channel j we have the gradient

$$\frac{\delta E}{\delta b_j} = \sum_{u,v} (\delta_j^l)_{uv} \tag{2.17}$$

with $u \in [0 \dots (h_{\mathbf{y}^l} - 1)]$ and $v \in [0 \dots (w_{\mathbf{y}^l} - 1)]$.

The gradient for the filter weights can be expressed (see appendix B) as

$$\frac{\delta E}{\delta \mathbf{k}_{ij}^l} = \text{corr2}(\mathbf{x}_i^l, \delta_j^l, \text{'valid'}) \tag{2.18}$$

that is we perform a 2-dimensional *valid* correlation[6] between the i-th input plane $\mathbf{x}_i^l$, corresponding to the i-th output plane $\mathbf{y}_i^{l-1}$ at up-stream layer $l-1$, with the j-th error plane δ_j^l to obtain the gradients for the j-th channel's i-th plane. (Recall that the kernels have the same depth, indexed with i, as the correlation layer's input x and we have c kernels (or channels) corresponding to the number of output planes, indexed with j.)

2.3 Pooling layers

The pooling layer does a 2D reduction in the height and width dimensions of a 3D input tensor; the depth remains unaffected. Formally we can write

$$\mathbf{y}_j^l = \text{down}(\mathbf{y}_j^{l-1}) \tag{2.19}$$

if we do not associate any activation function or bias with a pooling layer l. The number of planes in the output tensor remains unchanged, the heights and widths are affected as described on page 5.

[6] Similar to Python SciPy *correlate2d()* function or Matlab *xcorr2*.

2.3.1 Computing the Error

There are no learnable parameters for a pooling layer, which we assume to be surrouned up-stream and down-stream with a correlation layer. In the case of a flattening layer as the down-stream layer, the backpropagation approach of the fully-connected layers can be used (equation 2.14) where the derivation of the identity activation is 1.

The errors δ_j^l are the errors for the pooling layer l which has an output tensor depth equal to the number of channels in the up-stream correlation layer c which is unchanged by the pooling layer.

We have

$$\delta_i^l = \sum_{j=0}^{c} \text{corr2}(\delta_j^{l+1}, \text{rot180}(\mathbf{k}_{i,j}^{l+1}), \text{'full'}) \tag{2.20}$$

where $l+1$ indicates the down-stream correlation layer.

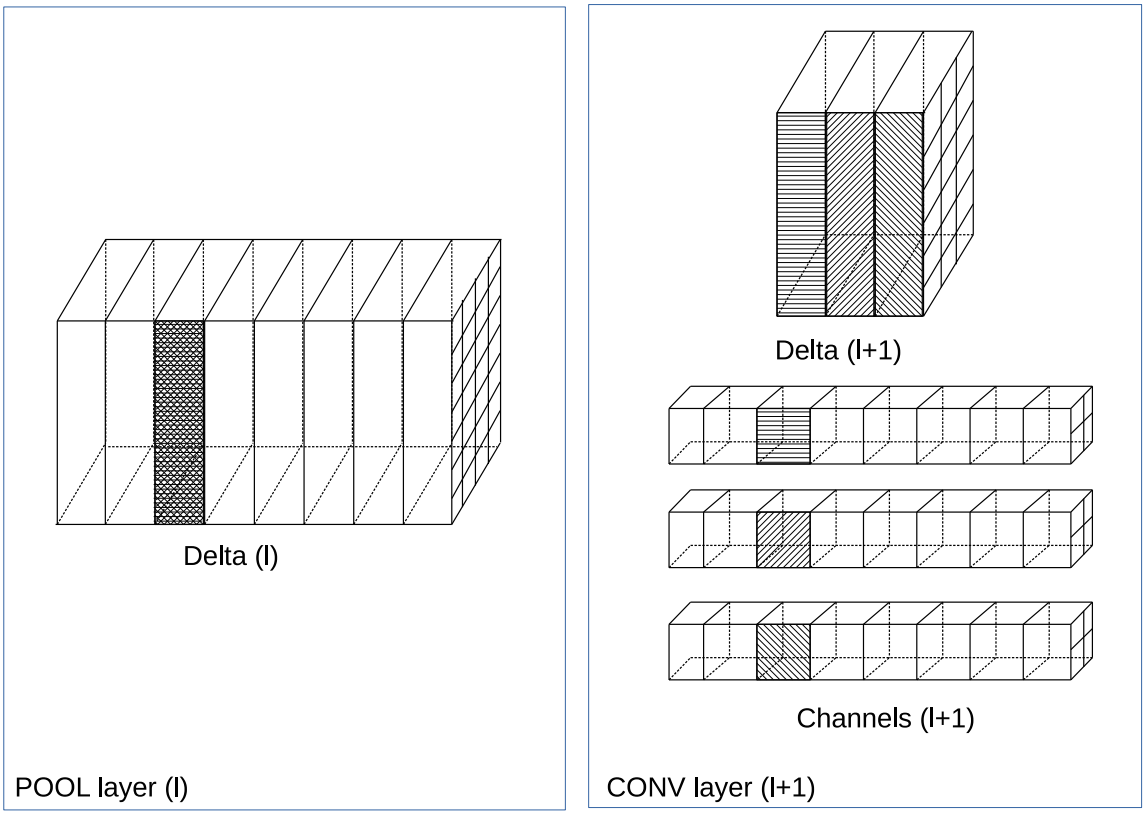

Figure 2.1: Backpropagation of error in pooling layer

We take the j-th error plane δ_j^{l+1} of the down-stream correlation layer $l+1$, having depth 1, and *fully* correlate it with the rotated by 180° i-th planes of all the c filters of the down-stream correlation layer to yield the i-th plane in δ_i^l.

This reflects the fact that the i-th input plane is connected via the i-th plane of the j-th channel/filter to the j-th output plane.

A Backpropagation in fully connected layers

Based on figure A.1 , we derive the formula for backpropagation of the error term.

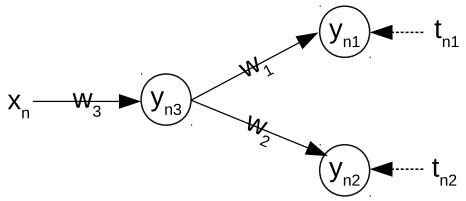

Figure A.1: Backpropagation in fully connected network

The error for the$n$-th sampleE_n is defined as

$$\begin{aligned} E_n &= \frac{1}{2}\sum_{k=1}^{2}(y_{nk} - t_{nk})^2 \\ &= \frac{1}{2}\left[(y_{n1} - t_{n1})^2 + (y_{n2} - t_{n2})^2\right] \end{aligned} \tag{A.1}$$

Note that E_n is a scalar and, in some cases, can be computed using one-hot encoding of truth values t_n.

For gradient descent optimization of the error we want to adjust the weights which influence the output of the network, given the input sample x_n. We have

$$\begin{aligned} y_{n1} &= w_1 y_{n3} = w_1 w_3 x_n \\ y_{n2} &= w_2 y_{n3} = w_2 w_3 x_n \end{aligned} \tag{A.2}$$

Note that we have no activation function for simplicity.

First we compute the derivative of the error E_n against the weight w_1 using the chain rule for derivation:

$$\begin{aligned} \frac{\delta E_n}{\delta w_1} &= (y_{n1} - t_{n1})\frac{\delta y_{n1}}{\delta w_1} \\ &= (y_{n1} - t_{n1})y_{n3} \\ &= (w_1 w_3 x_n - t_{n1})w_3 x_n \\ &= \delta_{n1} w_3 x_n \end{aligned} \tag{A.3}$$

The derivation for $\frac{\delta E_n}{\delta w_2}$ is similar.

For w_3 we also apply the chain rule and first write out the derivation of the error E_n against w_3 which influences both, y_1 and y_2:

$$\begin{aligned} \frac{\delta E_n}{\delta w_3} &= (y_{n1} - t_{n1})\frac{\delta y_{n1}}{\delta w_3} + (y_{n2} - t_{n2})\frac{\delta y_{n2}}{\delta w_3} \\ &= (w_1 w_3 x_n - t_{n1})w_1 x_n + (w_1 w_2 x_n - t_{n2})w_2 x_n \\ &= \delta_{n1} w_1 x_n + \delta_{n2} w_2 x_n \\ &= \left(\sum_{k=1}^{2}\delta_{nk} w_k\right) x_n \end{aligned} \tag{A.4}$$

B Backpropagation in correlation layers

Details on the backpropagation across correlation layers are hard to find. Some confusion stems from the use of correlations versus convolutions in CNNs: the two are equivalent when the filter operand in the convolution is rotated by 180°.

In this section, we are deriving in detail the formulas for backpropagation across correlational[7] layers. This material is largely based on [9].

B.1 One dimensional case

Assume an *input vector* $\mathbf{x} = [x_0, x_1, x_2, x_3]$ correlated with a *filter* (aka. the weights) $\mathbf{w} = [w_0, w_1]$ and a scalar b as our bias. The result of a *valid* correlation with no padding and a stride of 1 is an *output vector* $\mathbf{y} = [y_0, y_1, y_2]$

$$\begin{aligned} y_0 &= w_0 x_0 + w_1 x_1 + b \\ y_1 &= w_0 x_1 + w_1 x_2 + b \\ y_2 &= w_0 x_2 + w_1 x_3 + b \end{aligned} \tag{B.1}$$

We assume that $\frac{\delta E}{\delta y_j} = \delta_j, j = \{0, 1, 2\}$ are known from the down-stream network layer; starting backwards with the output layer $L - 1$ and using the error function introduced in 2.7. We are seeking the gradients of the error function against the bias $\frac{\delta E}{\delta b}$, against the filter coefficients (weights) $\frac{\delta E}{\delta w_k}, k = \{0, 1\}$ and against the input values $x_i, i = \{0, 1, 2, 3\}$, i.e. $\frac{\delta E}{\delta x_i} = \bar{\delta}_i$. Note that $\bar{\delta}_i$ becomes the error propagated to the up-stream layer when the influence of the activation function is included.

Using the chain rule we can write

$$\frac{\delta E}{\delta b} = \sum_{j=0}^{2} \frac{\delta E}{\delta y_j} \frac{\delta y_j}{\delta b} = [\delta_0, \delta_1, \delta_2][1, 1, 1]^T = \sum_{j=0}^{2} \frac{\delta E}{\delta y_j} = \sum_{j=0}^{2} \delta_j \tag{B.2}$$

For the filter weights we have

$$\frac{\delta E}{\delta w_k} = \sum_{j=0}^{2} \frac{\delta E}{\delta y_j} \frac{\delta y_j}{\delta w_k} = \sum_{j=0}^{2} \delta_j \frac{\delta y_j}{\delta w_k} \tag{B.3}$$

that is, given the linear dependencies againt w_j in equations B.1,

$$\begin{aligned} \frac{\delta E}{\delta w_0} &= x_0 \delta_0 + x_1 \delta_1 + x_2 \delta_2 \\ \frac{\delta E}{\delta w_1} &= x_1 \delta_0 + x_2 \delta_1 + x_3 \delta_2 \end{aligned} \tag{B.4}$$

Since

$$\frac{\delta y}{\delta w} = \begin{bmatrix} x_0 & x_1 \\ x_1 & x_2 \\ x_2 & x_3 \end{bmatrix} \tag{B.5}$$

we can write

[7] Recall we consistently use correlation instead of convolution.

$$\begin{bmatrix} \frac{\delta E}{\delta w_0} \\ \frac{\delta E}{\delta w_1} \end{bmatrix} = [\delta_0, \delta_1, \delta_2] \begin{bmatrix} x_0 & x_1 \\ x_1 & x_2 \\ x_2 & x_3 \end{bmatrix} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \otimes \begin{bmatrix} \delta_0 \\ \delta_1 \\ \delta_2 \end{bmatrix} \tag{B.6}$$

That is to say that the gradient of the filter weights $[\frac{\delta E}{\delta w_0}, \frac{\delta E}{\delta w_1}]$ is nothing but a correlation between the inputs x_l with the δ_js. Note that the input node index l ranges over the dimension of the input $\mathbf{x}$ and j over the dimension of output $\mathbf{y}$ after the correlation with mode *valid*, i.e. without any padding.

We now derive the formula used in the backpropagation of the δs between layers. Assume correlation layer has 3 output nodes and thus $[\delta_0, \delta_1, \delta_2]$. How is the error propagated backwards across the filter $\mathbf{w} = [w_0, w_1]$ into an upstream layer?

For the one-dimensional case, figure B.1 illustrates the case.

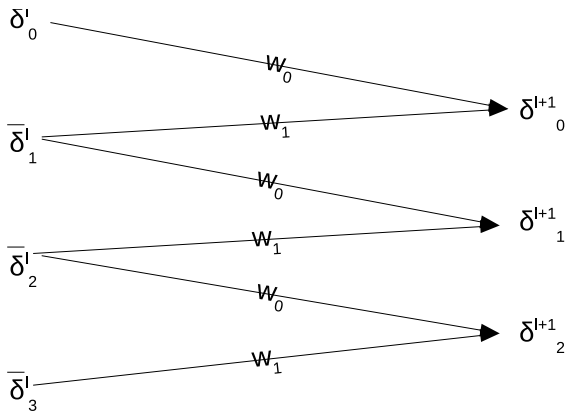

Figure B.1: Backpropagation of δ

We can write

$$\begin{aligned} \bar{\delta}_0 &= w_0 \delta_0^l \\ \bar{\delta}_1 &= w_1 \delta_0^l + w_0 \delta_1^l \\ \bar{\delta}_2 &= w_1 \delta_1^l + w_0 \delta_2^l \\ \bar{\delta}_3 &= w_1 \delta_2^l \end{aligned} \tag{B.7}$$

which can be written as

$$\begin{bmatrix} \bar{\delta}_0 \\ \bar{\delta}_1 \\ \bar{\delta}_2 \\ \bar{\delta}_3 \end{bmatrix} = \begin{bmatrix} 0 \\ \delta_0 \\ \delta_1 \\ \delta_2 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} w_1 \\ w_0 \end{bmatrix} \tag{B.8}$$

i.e. a *full* correlation between δ and a flipped filter $\mathbf{w_{180}} = [w_1, w_0]$ to obtain $\bar{\delta}$ for the up-stream layer.

B.2 Two-dimensional case

We now illustrate the case for two dimensions. We have

$$\mathbf{X} = \begin{bmatrix} x_{00} & x_{01} & x_{02} & x_{03} \\ x_{10} & x_{11} & x_{12} & x_{13} \\ x_{20} & x_{21} & x_{22} & x_{23} \\ x_{30} & x_{31} & x_{32} & x_{33} \end{bmatrix} \tag{B.9}$$

as *input matrix* and a *filter matrix*

$$\mathbf{W} = \begin{bmatrix} w_{00} & w_{01} \\ w_{10} & w_{11} \end{bmatrix} \tag{B.10}$$

The forward *output matrix* $\mathbf{Y}$ is

$$\begin{aligned} y_{00} &= w_{00}x_{00} + w_{01}x_{01} + w_{10}x_{10} + w_{11}x_{11} + b \\ y_{01} &= w_{00}x_{01} + w_{01}x_{02} + w_{10}x_{11} + w_{11}x_{12} + b \\ y_{03} &= w_{00}x_{02} + w_{01}x_{03} + w_{10}x_{12} + w_{11}x_{13} + b \\ &\dots \end{aligned} \tag{B.11}$$

i.e. the two-dimensional correlation $\mathbf{Y} = \mathbf{X} \otimes \mathbf{W} + b$ which is written out as

$$y_{ij} = \left(\sum_{k=0}^{1} \sum_{l=0}^{1} w_{kl} x_{i+k,j+l} \right) + b, i \in [0 \dots 2], j \in [0 \dots 2] \tag{B.12}$$

Note that the bias b is a scalar.

Again, we know $\delta_{ij} = \frac{\delta E}{\delta y_{ij}}$ from down-stream layers.

For the bias b we have

$$\frac{\delta E}{\delta b} = \sum_{i=0}^{2} \sum_{j=0}^{2} \frac{\delta E}{\delta y_{ij}} \frac{\delta y_{ij}}{\delta b} = \sum_{i=0}^{2} \sum_{j=0}^{2} \delta_{ij} \tag{B.13}$$

since $\frac{\delta y_{ij}}{\delta b} = 1$.

For the weights we have

$$\frac{\delta E}{\delta w_{mn}} = \frac{\delta E}{\delta y_{ij}} \frac{\delta y_{ij}}{\delta w_{mn}} = \delta_{ij} \frac{\delta y_{ij}}{\delta w_{mn}} \tag{B.14}$$

Using equation B.12 we have

$$\frac{\delta y_{ij}}{\delta w_{mn}} = \frac{\delta \left(\sum_{k=0}^{1} \sum_{l=0}^{1} w_{kl} x_{i+k,j+l} \right)}{\delta w_{mn}} \tag{B.15}$$

Because of the linear dependencies between weights w and inputs x, all terms disappear except when $(k, l) = (m, n)$, thus

$$\frac{\delta y_{ij}}{\delta w_{mn}} = x_{i+m,j+n} \tag{B.16}$$

and summing over (i, j) to obtain

$$\frac{\delta E}{\delta w_{mn}} = \sum_{i=0}^{2}\sum_{j=0}^{2} \delta_{ij} x_{i+m,j+n} \tag{B.17}$$

In matrix notation we can write

$$\frac{\delta E}{\delta w} = \begin{bmatrix} x_{00} & x_{01} & x_{02} & x_{03} \\ x_{10} & x_{11} & x_{12} & x_{13} \\ x_{20} & x_{21} & x_{22} & x_{23} \\ x_{30} & x_{31} & x_{32} & x_{33} \end{bmatrix} \otimes \begin{bmatrix} \delta_{00} & \delta_{01} & \delta_{02} \\ \delta_{10} & \delta_{11} & \delta_{12} \\ \delta_{20} & \delta_{21} & \delta_{22} \end{bmatrix} \tag{B.18}$$

which is nothing but a *valid* correlation of matrix $\mathbf{x}$ with filter matrix δ.

We now proceed to the gradient

$$\frac{\delta E}{\delta x_{mn}} = \bar{\delta}_{mn} = \frac{\delta E}{\delta y_{ij}} \frac{\delta y_{ij}}{\delta x_{mn}} \tag{B.19}$$

Using equation B.12, we have

$$\begin{aligned} \frac{\delta y_{ij}}{\delta x_{mn}} &= \frac{\delta \left(\sum_{k=0}^{1} \sum_{l=0}^{1} w_{kl} x_{i+k,j+l} \right)}{\delta x_{mn}} \\ &= \sum_{k=0}^{1}\sum_{l=0}^{1} w_{kl} \frac{\delta x_{i+k,j+l}}{\delta x_{mn}} \end{aligned} \tag{B.20}$$

The above conditions can be rewritten for the indices k, l

$$\frac{\delta x_{i+k,j+l}}{\delta x_{mn}} = \begin{cases} 1 & \text{if } k = m - i \text{ and } l = n - j \\ 0 & \text{otherwise} \end{cases} \tag{B.21}$$

We thus have

$$\frac{\delta y_{ij}}{\delta x_{mn}} = w_{m-i,n-j} \tag{B.22}$$

which combined with equation B.19 yields

$$\bar{\delta}_{mn} = \sum_{i=0}^{2}\sum_{j=0}^{2} \delta_{ij} w_{m-i,n-j} \tag{B.23}$$

For instance with $m = 0, n = 0$ we have

$$\begin{aligned} \bar{\delta}_{00} &= \sum_{i=0}^{2}\sum_{j=0}^{2} \delta_{ij} w_{0-i,0-j} \\ &= \sum_{i=0}^{2} \delta_{i0} w_{0-i,0} + \delta_{i1} w_{0-i,-1} + \delta_{i2} w_{0-i,-2} \\ &= \delta_{00} w_{0,0} + \delta_{01} w_{0,-1} + \delta_{02} w_{0,-2} \\ &+ \delta_{10} w_{-1,0} + \delta_{11} w_{-1,-1} + \delta_{12} w_{-1,-2} \\ &+ \delta_{20} w_{-2,0} + \delta_{21} w_{-2,-1} + \delta_{22} w_{-2,-2} \end{aligned} \tag{B.24}$$

Note the negative indices on some of the filter weights. We set the corresponding values to 0 and thus can write

$$\bar{\delta}_{00} = \begin{bmatrix} \delta_{00} & \delta_{01} & \delta_{02} \\ \delta_{10} & \delta_{11} & \delta_{12} \\ \delta_{20} & \delta_{21} & \delta_{22} \end{bmatrix} \otimes \begin{bmatrix} w_{00} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{B.25}$$

Eventually we can write

$$\begin{bmatrix} \bar{\delta}_{00} & \bar{\delta}_{01} & \bar{\delta}_{02} & \bar{\delta}_{03} \\ \bar{\delta}_{10} & \bar{\delta}_{11} & \bar{\delta}_{12} & \bar{\delta}_{13} \\ \bar{\delta}_{20} & \bar{\delta}_{21} & \bar{\delta}_{22} & \bar{\delta}_{23} \\ \bar{\delta}_{30} & \bar{\delta}_{31} & \bar{\delta}_{32} & \bar{\delta}_{33} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & \delta_{00} & \delta_{01} & \delta_{02} & 0 \\ 0 & \delta_{10} & \delta_{11} & \delta_{12} & 0 \\ 0 & \delta_{20} & \delta_{21} & \delta_{22} & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \otimes \begin{bmatrix} w_{11} & w_{10} \\ w_{01} & w_{00} \end{bmatrix} \tag{B.26}$$

In matrix notation we write

$$\bar{\delta} = \delta_{\mathbf{0}} \otimes \mathbf{w_{180}} \tag{B.27}$$

where $\delta_{\mathbf{0}}$ is a zero-padded matrix to implement a *full* correlation of the (unpadded) matrix δ with the filter $\mathbf{w}$ rotated[8] by 180°.

References

[1] Bishop, C. M. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[2] Blaise Barney. POSIX Threads Programming. `https://computing.llnl.gov/tutorials/pthreads/`, 2020. [Online; accessed 1-January-2020].

[3] Bouvrie, Jake. Notes on convolutional neural networks. `http://cogprints.org/5869/1/cnn_tutorial.pdf`, 2006. [Online; accessed 1-February-2020].

[4] Cauchy, A. Methode générale pour la résolution des systèmes d'équations simultanées. *Comptes rendus de l´ Académie des sciences 25* (1847), 536–538.

[5] Chollet, F. *Deep Learning with Python*. Manning Publications Co., 2018.

[6] Dongarra, J. BLAS: Basic Linear Algebra Subprograms. `http://www.netlib.org/blas/`, 9 2017. [Online; accessed 1-January-2020].

[7] Francois Chollet et al. Keras: the Python Deep-Learning Library. `https://keras.io/`, 2020. [Online; accessed 1-January-2020].

[8] Goodfellow, I., Bengio, Y., and Courville, A. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[9] Jaumier, Pierre. Backpropagation in a convolutional layer. `https://towardsdatascience.com/backpropagation-in-a-convolutional-layer-24c8d64d8509`, 2019. [Online; accessed 1-February-2020].

[8] This is also known as *flipped*.

[10] Lavin, A., and Gray, S. Fast algorithms for convolutional neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition* (2106).

[11] Lawson, P., Hanson, R., Kincaid, D., and Krogh, F. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software 5*, 3 (9 1979), 308–323.

[12] Nielsen, M. A. *Neural Networks and Deep Learning*. Determination Press, 2015. `http://neuralnetworksanddeeplearning.com`.

[13] numpy.org. NumPy: Python package for scientific computing. `https://numpy.org/doc/`, 2020. [Online; accessed 1-January-2020].

[14] Nvidia Corporation. Cuda c++ programming guide. Tech. Rep. 2, Nvidia, Santa Clara, California, 11 2019. `https://developer.nvidia.com/cuda-zone`.

[15] Oppenheim, A. V., and Schafter, R. W. *Discrete-Time Signal Processing*. Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[16] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12* (2011), 2825–2830.

[17] Stanford University. CS231n: Convolutional Neural Networks for Visual Recognition. `http://cs231n.github.io/`, 2018. [Online; accessed 1-January-2020].

[18] Wikipedia. Streaming SIMD Extension. `https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions`, 2020. [Online; accessed 1-January-2020].

[19] Winograd, S. On multiplication of polynomials modulo a polynomial. *SIAM J. Comput. 9*, 2 (1980), 225–229.