Contents

Chapter 1

Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are widely used algorithms in the field of neural networks. They are particularly known for their capability in feature extraction, weight sharing, and dimension reduction. CNNs leverage the mathematical properties of convolution to extract valuable information from input images and utilize the propagation characteristics of neural networks to produce results through a classifier.

This algorithm empowers CNNs with self-machine learning capabilities, making them indispensable in deep learning. Notably, CNNs serve as crucial mathematical structures for analyzing datasets and images, excelling at rapidly extracting features from samples with diverse requirements.

A Convolutional Neural Network (CNN) comprises multiple layers, resembling a long shelf, with each layer consisting of numerous computing units or elements that process data simultaneously. Within the same layer, these computing units or elements perform identical functions on the input data. Despite its complexity, a CNN generally consists of three main types of layers: convolutional layers, pooling layers, and output layers.

1.1 Layers

Convolutional layers serve as the initial step in a CNN. They extract features from the input data and simultaneously reduce its size. As the data progresses through multiple convolutional layers, it undergoes further feature extraction.

Following the convolutional layers, the data is passed to pooling layers. The primary purpose of pooling layers is to enhance the extracted features while reducing the spatial dimensions of the data. This process involves truncating or downsampling the data, allowing for more efficient computation.

Ultimately, the results generated by the CNN are obtained from the output layers. These layers provide the final classification or regression outputs based on the learned features and patterns from the previous layers.

In summary, the layers of a CNN work together to extract features from the input data, progressively enhancing and reducing its dimensions. The convolutional layers extract features, pooling layers enhance them, and the output layers produce the final results of the CNN.

1.1.1 Convolutional Layers

The convolution operation involves applying a set of learnable filters, also known as convolutional kernels or weights, to the input data in order to extract meaningful features.
In the context of image processing, the input data is typically a 3D tensor representing an image, with dimensions of width, height, and channels (e.g., RGB channels). The convolutional layer convolves these filters across the input data by sliding them spatially.

The convolutional window, also referred to as the filter size or kernel size, determines the receptive field of the convolution operation. It represents the spatial extent of the filters and defines the area of the input data that the filters cover at a time. The size of the convolutional window is typically square, with a specified width and height.

The stride, on the other hand, determines the step size or the amount by which the convolutional window slides across the input data. It defines the spatial movement of the window during the convolution operation. A larger stride value results in a larger spatial downsampling or compression of the output data.

Visually, the convolution operation can be represented as follows:

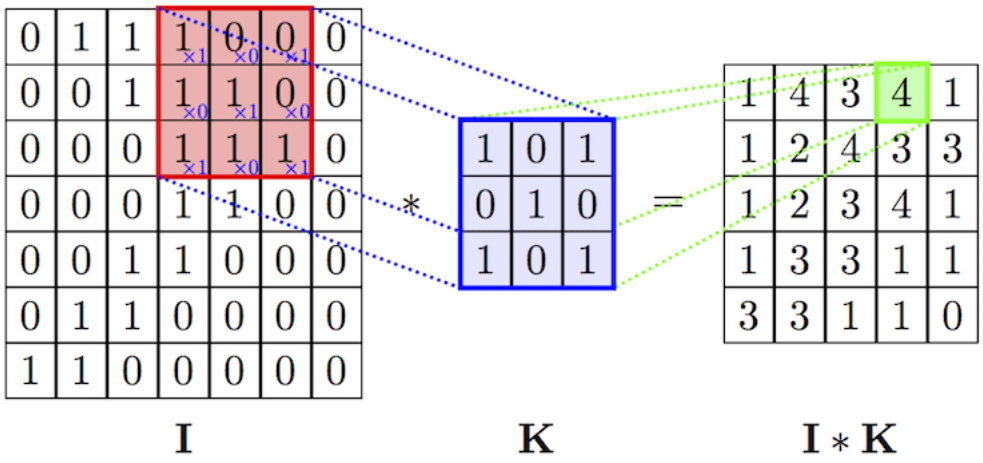


Figure 1.1: An Example of Convolution with 3x3 Kernel and Stride = 1

Analytically, we can express the convolution as follows, starting from a single unit of a convolutional layer:

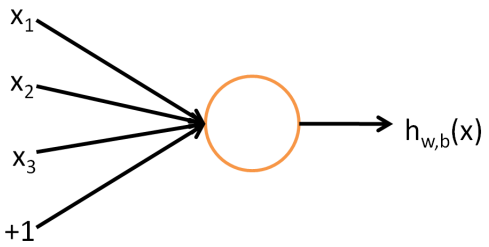


Figure 1.2: Single Computing Unit in Convolutional Layer, single Kernel

This "neuron" is a computational unit that takes as input x1,x2,x3 (and a +1 bias term), and outputs:

$$h_{W,b}(x) = f\left(W^T x\right) = f\left(\sum_{i=1}^{3} W_i x_i + b\right) \tag{1.1}$$

which is called the activation function. In a two-layer structure, the first layer can be interpreted as the input items, while the second layer corresponds to a convolutional layer.

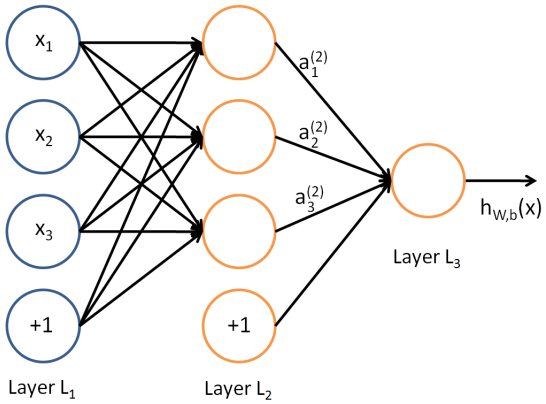


Figure 1.3: Two-Layer Basic Convolutional Network

In the second layer, each circle is associated with a unique weighted function. Consequently, each input data element undergoes computations with all the weighted functions present in layer 2. The resulting outputs are then transmitted to the subsequent layer, which is referred to as the fully connected convolutional layer. The detailed expressions are as follows:

$$\begin{aligned}
a_1^{(2)} &= f\left(W_{11}^{(1)} x_1 + W_{12}^{(1)} x_2 + W_{13}^{(1)} x_3 + b_1^{(1)}\right) \\
a_2^{(2)} &= f\left(W_{21}^{(1)} x_1 + W_{22}^{(1)} x_2 + W_{23}^{(1)} x_3 + b_2^{(1)}\right) \\
a_3^{(2)} &= f\left(W_{31}^{(1)} x_1 + W_{32}^{(1)} x_2 + W_{33}^{(1)} x_3 + b_3^{(1)}\right) \\
h_{W,b}(x) = a_1^{(3)} &= f\left(W_{11}^{(2)} a_1^{(2)} + W_{12}^{(2)} a_2^{(2)} + W_{13}^{(2)} a_3^{(2)} + b_1^{(2)}\right)
\end{aligned} \tag{1.2}$$

1.1.2 Pooling Layers

Max and Average pooling are the two types of pooling layers and are common operations used in convolutional neural networks (CNNs) for down-sampling or reducing the spatial dimensions of feature maps. Both pooling layers are typically inserted between convolutional layers to extract important features and reduce the computational complexity of subsequent layers.

Max pooling operates by dividing the input feature map into non-overlapping rectangular regions (or "pools") and taking the maximum value within each region. The output of each pool is the maximum value, effectively representing the most prominent feature present in that region. Max pooling aids in feature detection by selecting the strongest activation within a local neighborhood.

Mean pooling, also known as average pooling, computes the average value of each pool in the input feature map. It divides the feature map into non-overlapping pools and calculates the mean value within each pool. Average pooling reduces the spatial dimensions of the feature map in a similar way to max pooling, and can be useful when preserving the spatial structure of the input is more important than capturing specific local maxima.
It provides a slightly smoother representation of the input compared to max pooling, as it takes into account all values within each pool.

In both max pooling and mean pooling, the pooling operation is performed independently on each feature map/channel of the input. The pooling layers have hyperparameters such as the size of the pooling window and the stride, which determine the pool's size and the amount of overlap between pools.
Overall, max pooling and mean pooling layers help to down-sample the input feature maps, reducing their spatial dimensions while retaining important information. The two operations are visually represented as follows:

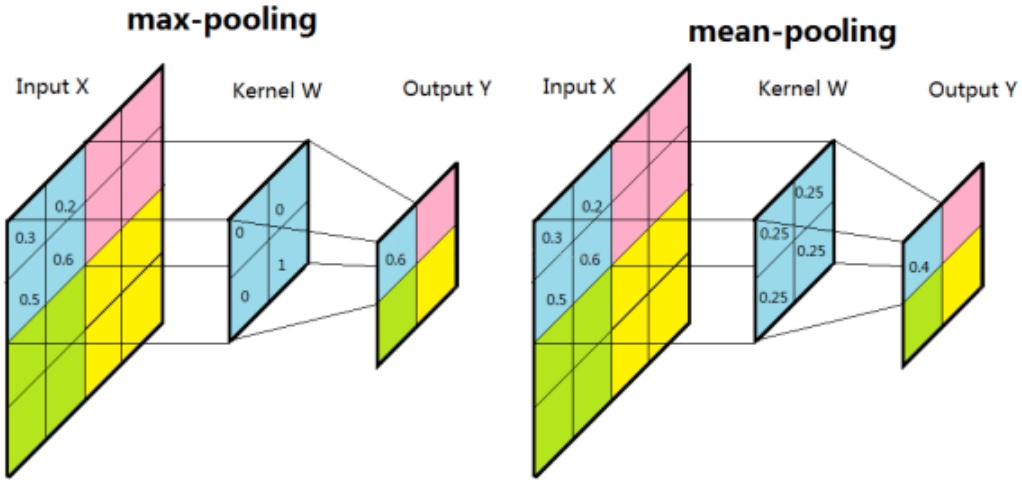


Figure 1.4: Max and Average pooling operations

1.1.3 Output Layers

Convolutional layers and pooling layers play crucial roles in feature extraction and dimensionality reduction within a convolutional neural network (CNN). While these layers focus on extracting important features and reducing unnecessary information, they alone are not sufficient for achieving accurate categorization. To produce meaningful and diverse outputs, a fully connected layer is commonly employed as the output layer.

The fully connected output layer serves as the final stage in the CNN architecture, responsible for generating a suitable quantity and variety of results. It acts as a classifier, employing computational units to calculate a loss function. The loss function plays a vital role in adjusting the weighted factors of the convolutional layers.

During the forward pass of the CNN, the input data propagates through the layers, and once it reaches the output layer, the loss function comes into play. The loss function assesses the disparity between the predicted outputs and the actual targets, enabling the network to learn and adjust its weights accordingly. This process is known as the backward pass or backpropagation.

By leveraging the loss function during the backward pass, the CNN can identify and rectify errors and losses incurred during the weighted computations. This iterative feedback loop enables the network to refine its predictions and optimize its performance over subsequent training iterations.

1.2 Gradient Descent and Backpropagation

There are two passes in the CNNs: the forward pass and the backward pass that is also known as Error Backpropagation. The latter will be covered in this section.
Because of the random initialization of the convolutional layers filters' values, this initial setting will be repeatedly fixed again and again during the CNNs training process. Indeed, the BackPropagation (BP) is for transmitting the fixing error to the lower layers from the upper layers. How can we find "the fixed errors"? Here comes the *Gradient Descent* method.[1]

1.2.1 Gradient descent

Gradient Descent (GD) method is one of the most common optimization algorithms in machine learning field. From a general point of view we know that most of the mathematical models have errors due to estimating factors. The total error between samples and the function output is described by a notion known as loss function. Assuming we are using the following mathematical model:

$$h_w(x_1, x_2, \ldots, x_n) = w_0 + w_1 x_1 + \ldots + w_n x_n \qquad w_i = (i = 1, 2, \ldots, n) \tag{1.3}$$

where w_0, w_1, …, w_n are the equation's components. By adding a feature value$x_0 = 1$, which

refers to the bias, we can simplify the function as:

$$h_w(x_1, x_2, \ldots, x_n) = \frac{1}{2m} \sum_{i=0}^{m} (w_i x_i) \tag{1.4}$$

Thus, the relative loss function should be:

$$J(w_0, w_1, \ldots, w_n) = \frac{1}{2m} \sum_{i=0}^{m} (h_w(x_0, x_1, \ldots, x_n) - y_i)^2 \tag{1.5}$$

Its sample is (x_i, y_i) (i=1,2,. . .,n) where every x_i corresponds to a y_i. By modifying the weights of the original function and lowering the value of the loss function, the gradient descent technique tries to optimize both the mathematical model and the loss function.
As the name suggests, the gradient is used. It measures the trend of the loss function and by considering the partial derivative of w_i to the loss function as a gradient, the equation is:

$$\frac{\delta}{\delta w_i} J(w_0, w_1, \ldots, w_n) \tag{1.6}$$

A distance threshold parameter ϵ, usually the distance between the output of the mathematical model and the desired sample output, is used to define the function precision: when this difference is less than or equal to the distance threshold, the weighted functions in the model are satisfied and the training process stops. A learning rate parameter η, also known as step size, is a factor that says which is the portion of the gradient used to update the new weight w_i, it's a number in [0,1] and need to be set before the process starts.
The direction of the weight update is computed by taking a step in the opposite direction of the gradient. Here is the updating formula that uses the gradient descent:

$$w_{i_{new}} = w_{i_{old}} - \eta \frac{\delta}{\delta w_i} J(w_0, w_1, \ldots, w_n) \tag{1.7}$$

Some consideration about the learning rate η need to be done since it covers a significant role during the programming. When η is too small, starting from the initial random set of values, the training process move slowly towards the minimum error $J_{min}(w_0, w_1, \ldots, w_n)$, but it takes too many iterations to reach the optimal because it slowers the convergence speed and makes the process inefficient; on the other hand, a large value can extremely improve the convergence speed, but while approaching the optimal it can start oscillating around that point and it can even diverge from this converge space.
The setting of an appropriate learning rate value it's a matter of experimentation and typically can vary over time.
In the figure below we can see a comparison among different learning rate's loss functions convergence situations:

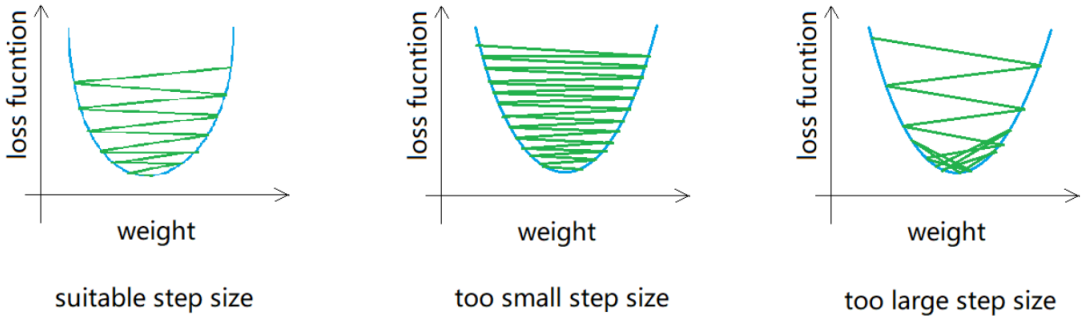


Figure 1.5: Comparison of different step size

The general gradient descent algorithm is as follows:

1. Initialise w_0, w_1, …, w_n in the cost function, a distance threshold parameter ϵ, a maximum number of epochs and the learning rate η.
2. Calculate the gradient, i.e. the partial derivative of the loss function: $\frac{\delta}{\delta w_i} J(w_0, w_1, \ldots, w_n)$.
3. If $J(w_0, w_1, \ldots, w_n) \leqslant \epsilon$ or maximum number of epochs is reached stop the algorithm, otherwise, continue the process.
4. Use the gradient multiplying with step size η.
5. Renew all the weights w by $w_i = w_i - \eta \frac{\delta}{\delta w_i} J(w_0, w_1, \ldots, w_n)$, and then back to step 2.

Summing up, the way the gradient descent algorithm works is to repeatedly compute the gradient, and then to move in the opposite direction, "falling down" the slope of the valley. We can easily visualize it for a cost function with a single weight w:

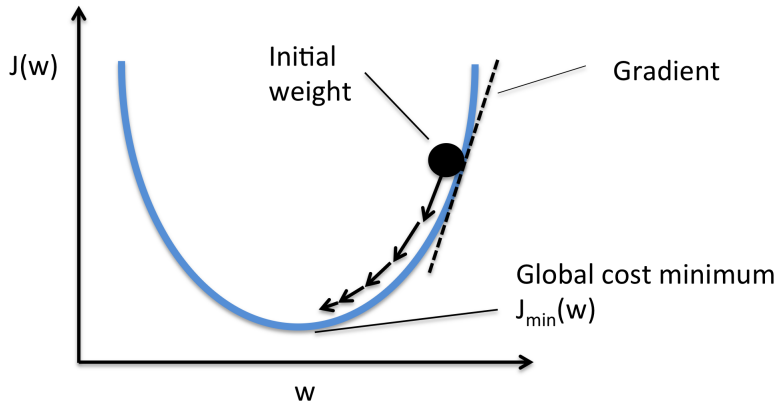


Figure 1.6: Gradient descent procedure

1.2.2 Backpropagation

Besides updating weight functions, the updating fixed errors should be passed backward through the layers in the CNNs. Below we can see an example of backpropagation in a single neuron:

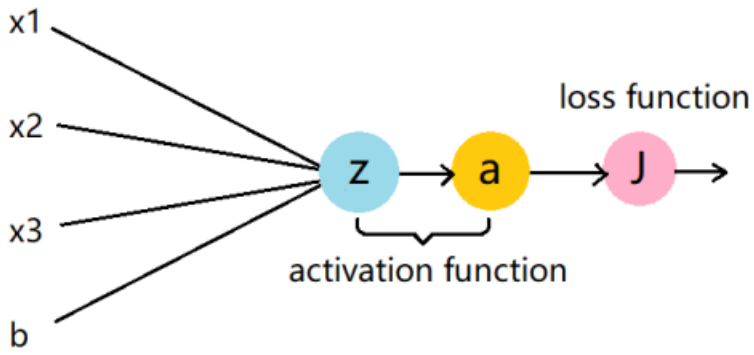


Figure 1.7: Example of backpropagation in a single calculating unit

Assuming that the training sample is (x, y), where x is the input which passes through the activation function $h_{w,b}(x)$, the neuron gets an output a that goes through the loss function whose output is the cost J. The expression of the activation function is:

$$h_{w,b}(x) = a = sigmoid(z) = sigmoid\left(\sum_{i=0}^{n}(x_i W_i + b)\right) \tag{1.8}$$

where the activation function is the sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$. Other functions can also be used as activation function to introduce nonlinearity into neural networks. However, a reason why the sigmoid function is so popular is because it satisfies such a property:

$$\frac{\delta\, sig(z)}{\delta z} = sig(z)(1 - sig(z)) \tag{1.9}$$

that is a simple polynomial relationship between the derivative and the function itself that, from a programming perspective, is simple to implement.
The equation of the loss function is:

$$J(W, b, x, y) = \frac{1}{2}||y - h_{w,b}(x)||^2 \tag{1.10}$$

where y is the target output of the training sample, while the result of the activation function $h_{w,b}(x)$ is the predicted output.

During the training there are two gradients of the cost J to compute by partial derivatives: one is for the factor W representing the weights and the other one is for the bias b. However, we cannot calculate their partial derivative directly from the cost J. The chain rule would help to in this case. It can viewed as $\frac{\delta}{\delta W} J = \frac{\delta J}{\delta z}\frac{\delta z}{\delta W}$. Now, we should figure out the partial derivative from cost J to intermediate variable a and z at first:

$$\delta^{(a)} = \frac{\delta}{\delta a} J(W, b, x, y) = -(y - a) \tag{1.11}$$

$$\delta^{(b)} = \frac{\delta}{\delta z} J(W, b, x, y) = \frac{\delta J}{\delta a} \frac{\delta a}{\delta z} = \delta^{(a)} a(1-a) \tag{1.12}$$

Now, according to the chain rule, we can obtain W and b 's gradients:

$$\nabla_W J(W, b, x, y) = \frac{\delta}{dW} J = \frac{\delta J}{\delta z} \frac{\delta z}{\delta W} = \delta^{(z)} x^T \tag{1.13}$$

$$\nabla_b J(W, b, x, y) = \frac{\delta}{db} J = \frac{\delta J}{\delta z} \frac{\delta z}{\delta b} = \delta^{(z)} \tag{1.14}$$

During the above processes, we take $\frac{\delta J}{\delta a}$ firstly, then figure out $\frac{\delta J}{\delta z}$, finally we get $\frac{\delta J}{\delta W}$ and $\frac{\delta J}{\delta b}$. Therefore, we can find that this is a procedure which propagates the increment δJ of the cost function from the back to the forward, it's called backpropagation.

1.2.3 CNN backpropagation observations

The error backpropagation algorithm as it is shown above is used when dealing with a Multi-Layer Perceptron (MLP), a DNN that can include more than one hidden layer.
Due to the different structural layers of a CNN, to apply DNN's backpropagation algorithm to CNN, there are several problems to be solved:

1. There is no activation function in the pooling layer. We can make the activation function of the pooling layer as $\sigma(z) = z$, that is, it's itself after activation. In this way, the derivative of the pooling layer activation function is 1.

2. The pooling layer compresses the input during forward propagation, so we now need to deduce δ^{l-1} forward and backward. This derivation method is completely different from DNNs.

3. The output of the convolutional layer is obtained as a linear tensor convolution between the input and the current layer that consists in one or more kernels. This is very different from DNN. Indeed, the DNN fully connected layer directly performs matrix multiplication to obtain the current layer output. In this way, when the convolutional layer is backpropagating, the δ^{l-1} recursive calculation method of the previous layer must be different. The recursive calculation method is definitely different.

4. For the convolutional layer, since the operation used by W is convolution, the way of deriving W, b of all convolution kernels of this layer from δ^l is also different.

It can be seen that problem 1 is relatively easy to solve, but problems 2, 3, and 4 need to be brainstormed and are also the key to solve the CNN backpropagation algorithm. In addition, everyone should note that inputs and outputs in DNN are just vectors, while in CNN they are all three-dimensional tensors, that is, composed of several input composed of sub-matrixes [2].

Chapter 2

Dual Numbers

Dual numbers are a mathematical construct that extends the real numbers by introducing an additional element called the dual unit, denoted by ϵ. Each dual number has two components: a real part and a dual part. A dual number is represented as $a + b\epsilon$, where a and b are real numbers and ϵ represents the dual unit, which is an additional element introduced to extend the real numbers. It is defined such that ϵ^2 equals zero ($\epsilon^2 = 0$).

Basic operations definitions for Dual Numbers:

- Addition:
$$(a + b\epsilon) + (c + d\epsilon) = (a + c) + (b + d)\epsilon$$
- Subtraction:
$$(a + b\epsilon) - (c + d\epsilon) = (a - c) + (b - d)\epsilon$$
- Multiplication:
$$(a + b\epsilon) \cdot (c + d\epsilon) = ac + (ad + bc)\epsilon + bd\epsilon^2$$
(Note: ϵ^2 is taken to be zero, as $\epsilon^2 = 0$)
- Division:
$$\frac{(a + b\epsilon)}{(c + d\epsilon)} = \frac{ac + bd\epsilon}{c^2}$$
(Note: Division by dual part is not allowed)

2.1 Automatic Differentiation

One of the most important application for Dual Numbers is Automatic differentiation, a key feature that leverages their properties. In this approach, each variable in a function is represented as a dual number, where the real part corresponds to the value of the variable, and the dual part represents its derivative. By using dual numbers, the derivatives are automatically computed as a byproduct of evaluating the function.

The dual unit ϵ is a key element in automatic differentiation using dual numbers. It allows

for the computation of derivatives as a byproduct of evaluating functions symbolically. The dual part, when multiplied by ϵ, captures the derivative information, allowing for accurate and efficient derivative calculations without the need for numerical approximations.

It's important to note that ϵ^2 is taken to be zero in dual numbers. This property ensures that higher-order derivatives are automatically accounted for, simplifying the computation of derivatives.

In general, any analytic real function can be extended to the Dual Numbers by leveraging at the Taylor series of that function:

$$f(a+be) = \sum_{i=1}^{\infty} \frac{f(n)(a)b^n\varepsilon^n}{n!} = f(a) + bf'(a)\varepsilon \tag{2.1}$$

For instance, applying this property to a real polynomial P(x)

$$P(a+b\varepsilon) = p_0 + p_1(a+b\varepsilon) + \cdots + p_n(a+b\varepsilon) \tag{2.2}$$

$$= p_0 + p_1a + p_2a^2 + \cdots + p_na^n + p_1b\varepsilon + p_2ab\varepsilon + \cdots + p_na^{n-1}b\varepsilon \tag{2.3}$$

$$= P(a) + bP'(a)\varepsilon \tag{2.4}$$

Thus, we obtained P' which is the derivative of P.

Automatic differentiation using dual numbers can be highly advantageous in the context of neural network learning, specifically for the backpropagation algorithm. By collecting derivatives during the forward pass, we can avoid the need to compute derivatives explicitly during the backward pass.

Chapter 3

CNN Implementation

We started from a convolutional neural network (CNN) implementation in Matlab [6.1] to study the feasibility of the application of Dual Numbers. The CNN will be trained in order to recognize hand-written digit using the MNIST dataset.
The author uses a custom MNIST dataset where the size of the images has been halved ($28 \times 28 \rightarrow 14 \times 14$).
There are two MAT files (mnist_train.mat and mnist_test.mat), where each file includes im_* and label_* variables:

- **im_***: is a matrix (196 × n) storing vectorized image data (196 = 14 × 14)
- **label_***: is n × 1 vector storing the label for each image data.

n corresponds to the number of images. It is possible to visualize the i^{th} image as follows:

$$imshow(uint8(reshape(im\_train(:, i), [14, 14])))$$

3.1 CNN structure

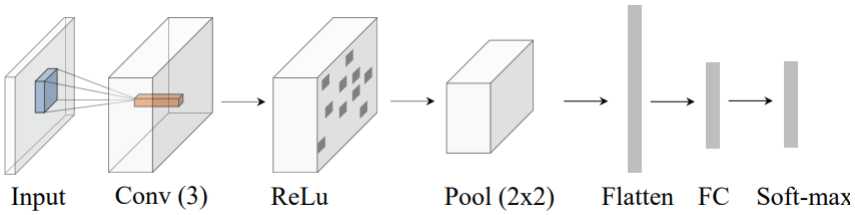


Figure 3.1: CNN structure

As shown in Figure 3.1, the network is composed of:

- Single channel input (14×14×1)
- Convolutional layer (3×3 convolution with 3 channel output and stride 1)

- ReLU layer
- Max-pooling layer (2×2 with stride 2)
- Flattening layer (147 units)
- Fully Connected (FC) layer (10 units)
- Softmax layer

3.2 CNN Functions

Below we reported the list of the implemented functions and their definitions:

- main_cnn.m
- TrainCNN.m
- GetMiniBatch.m
- FC.m
- FC_backward.m
- Loss_cross_entropy_softmax.m
- ReLu.m
- ReLu_backward.m
- Conv.m
- Conv_backward.m
- Pool2x2.m
- Pool2x2_backward.m
- Flattening.m
- Flattening_backward.m

Function: `main_cnn()`
Description: This is the main script for training and testing a convolutional neural network (CNN) on the MNIST dataset. It performs the following steps:

1. Load the training and test data.
2. Preprocess the image data by normalizing the inputs.
3. Get mini-batches of training data using the `GetMiniBatch` function.
4. Train the CNN using the `Train`CNN function.
5. Test the CNN by iterating over the test images and computing predictions.
6. Save the trained network into the `Outputs` folder in a MAT file named `cnn.mat`. The MAT file contains the trained weights and biases of the network.

Function: `[w_conv, b_conv, w_fc, b_fc] = TrainCNN(mini_batch_x, mini_batch_y)`

Output:

- `w_conv` $\in \mathbb{R}^{3\times3\times1\times3}$
- `b_conv` $\in \mathbb{R}^{3}$
- `w_fc` $\in \mathbb{R}^{10\times147}$
- `b_fc` $\in \mathbb{R}^{147}$

are the trained weights and biases of the CNN.

Description: Use the following functions to train a convolutional neural network using a stochastic gradient descent method: `Conv`, `Conv_backward`, `Pool2x2`, `Pool2x2_backward`, `Flattening`, `Flattening_backward`, `FC`, `FC_backward`, `ReLu`, `ReLu_backward`, `Loss_cross_entropy_softmax`.

Function: `[mini_batch_x, mini_batch_y] = GetMiniBatch(im_train, label_train, batch_size)`

Input:

- `im_train` and `label_train` - a set of images and labels, respectively
- `batch_size` - the size of the mini-batch for stochastic gradient descent

Output:

- `mini_batch_x` and `mini_batch_y` - cells that contain a set of batches (images and labels, respectively)
- Each batch of images is a matrix with size $194 \times$ batch_size
- Each batch of labels is a matrix with size $10 \times$ batch_size (one-hot encoding)
- Note that the number of images in the last batch may be smaller than `batch_size`

Description:

- The order of images is permuted when building the batch
- Whole sets of `mini_batch_*` must span all training data

Function: `y = FC(x, w, b)`

Input:

- $\mathtt{x} \in \mathbb{R}^m$ is the input to the fully connected layer
- $\mathtt{w} \in \mathbb{R}^{n \times m}$ is the weight matrix
- $\mathtt{b} \in \mathbb{R}^n$ is the bias vector

Output:

- $\mathtt{y} \in \mathbb{R}^n$ is the output of the linear transform (fully connected layer)

Description:

- FC is a linear transform of $\mathtt{x}$, i.e., $y = wx + b$

Function: `[dLdx, dLdw, dLdb] = FC_backward(dLdy, x, w, b, y)`
Input:

- $\mathtt{dLdy} \in \mathbb{R}^{1 \times n}$ is the loss derivative with respect to the output y
- $\mathtt{x} \in \mathbb{R}^{1 \times m}$ is the input to the fully connected layer
- $\mathtt{w} \in \mathbb{R}^{n \times m}$ is the weight matrix
- $\mathtt{b} \in \mathbb{R}^{1 \times n}$ is the bias vector
- $\mathtt{y} \in \mathbb{R}^{1 \times n}$ is the output of the linear transform (fully connected layer)

Output:

- $\mathtt{dLdx} \in \mathbb{R}^{1 \times m}$ is the loss derivative with respect to the input x
- $\mathtt{dLdw} \in \mathbb{R}^{1 \times (n \times m)}$ is the loss derivative with respect to the weights
- $\mathtt{dLdb} \in \mathbb{R}^{1 \times n}$ is the loss derivative with respect to the bias

Description: The partial derivatives with respect to the input, weights, and bias will be computed. `dLdx` will be back-propagated, and `dLdw` and `dLdb` will be used to update the weights and bias.

Function: `[L, dLdy] = Loss_cross_entropy_softmax(x, y)`
Input:

- $\mathtt{x} \in \mathbb{R}^m$ is the input to the softmax
- $\mathtt{y} \in \{0, 1\}^m$ is the ground truth label

Output:

- $\mathtt{L} \in \mathbb{R}$ is the loss

- dLdy is the loss derivative with respect to x

Description: Loss_cross_entropy_softmax measures the cross-entropy between two distributions given by $L = \sum_i y_i \log \tilde{y}_i$, where $\tilde{y}_i$ is the softmax output that approximates the max operation by clamping x to the range [0, 1].

The softmax output $\tilde{y}_i$ is calculated as follows:

$$\tilde{y}_i = \frac{e^{x_i}}{\sum_i e^{x_i}},$$

where x_i represents the i-th element of x.

Function: [y] = ReLu(x)
Input:

- x is a general tensor, matrix, or vector.

Output:

- y is the output of the Rectified Linear Unit (ReLU) with the same input size.

Description: ReLu is an activation unit defined as $y_i = \max(0, x_i)$. It applies an element-wise operation, setting negative values to zero and keeping non-negative values unchanged.

Function: [dLdx] = ReLu_backward(dLdy, x, y)
Input:

- dLdy $\in \mathbb{R}^{1\times z}$ is the loss derivative with respect to the output y $\in \mathbb{R}^z$, where z is the size of the input (which can be a tensor, matrix, or vector).

Output:

- dLdx $\in \mathbb{R}^{1\times z}$ is the loss derivative with respect to the input x.

Description: ReLu_backward computes the derivative of the loss with respect to the input x, given the loss derivative dLdy with respect to the output y. It performs an element-wise operation, propagating the gradient only to the non-negative elements of x, while setting the gradient to zero for the negative elements.

Function: [y] = Conv(x, w_conv, b_conv)
Input:

- x $\in \mathbb{R}^{H\times W\times C1}$ is an input to the convolutional operation.
- w_conv $\in \mathbb{R}^{H\times W\times C1\times C2}$ is the weight tensor of the convolutional operation.

- b_conv $\in \mathbb{R}^{C2}$ is the bias vector of the convolutional operation.

Output:

- y $\in \mathbb{R}^{H \times W \times C2}$ is the output of the convolutional operation.

Description: The Conv function performs a convolutional operation on the input x using the weights w_conv and bias b_conv. The input x has dimensions $H \times W \times C1$, where H and W represent the height and width of the input image respectively, and $C1$ represents the number of input channels. The weights w_conv have dimensions $H \times W \times C1 \times C2$, where $C2$ represents the number of output channels. The output y has dimensions $H \times W \times C2$.

Note that in order to obtain the output with the same size as the input, zero-padding can be applied at the boundary of the input image.

Function: `[dLdw, dLdb] = Conv_backward(dLdy, x, w_conv, b_conv, y)`

Input:

- dLdy is the loss derivative with respect to the output y.

Output:

- dLdw and dLdb are the loss derivatives with respect to the convolutional weights w_conv and bias b_conv, respectively.

Description: The Conv_backward function computes the derivatives of the loss with respect to the convolutional weights w_conv and bias b_conv, given the loss derivative dLdy with respect to the output y. This operation can be simplified using the MATLAB built-in function im2col. Note that for our implementation the a single convolutional layer placed immediately after the inputs, the derivative $\frac{\partial L}{\partial x}$ is not needed as we reached the end of the backpropagation phase.

Function: `[y] = Pool2x2(x)`

Input:

- x $\in \mathbb{R}^{H \times W \times C}$ is a general tensor and matrix.

Output:

- y $\in \mathbb{R}^{\frac{H}{2} \times \frac{W}{2} \times C}$ is the output of the 2×2 max-pooling operation with stride 2.

Function: `[dLdx] = Pool2x2_backward(dLdy, x, y)`

Input:

- dLdy is the loss derivative with respect to the output y.

Output:

- `dLdx` is the loss derivative with respect to the input `x`.

Function: `[y] = Flattening(x)`
Input:

- $\mathtt{x} \in \mathbb{R}^{H \times W \times C}$ is a tensor.

Output:

- $\mathtt{y} \in \mathbb{R}^{HWC}$ is the vectorized tensor (column major).

Function: `[dLdx] = Flattening_backward(dLdy, x, y)`
Input:

- `dLdy` is the loss derivative with respect to the output `y`.

Output:

- `dLdx` is the loss derivative with respect to the input `x`.

3.3 Reverse Engineering of Convolution Operations

Our work was particularly focused on the convolution operation since it is the one that characterizes CNNs.
In particular, since the author had not provided any kind of theoretical explanation in the implementation about the used formulas and tricks, we had to trace back via reverse engineering it in various matlab functions (conv.m, conv_backward.m).

Our aim was to provide an accurate theoretical documentation for this CNN implementation, as it was lacking appropriate explanation of the used formulas and was initially hard to understand. After discovering various tricks used in the code and deriving formulas by hand, the code became clearer and hopefully usable for anyone whoever will approach this implementation for future works.

3.3.1 Convolution Forward

The "naive" implementation of the convolution operation [1.1.1] would have been quite trivial to implement, as we would had to simply traverse the input matrix and pull out "windows" that are equal to the shape of the kernel. For each window, perform element-wise multiplication with the kernel and sum up all the values. Finally, before returning the result we would have added the bias term to each element of the output.

Basically, in this naive version of convolution while multiplying each window with the kernel we would have done 2 operations:

1. Multiplied the terms
2. Added them all together

and would have done this for each window in the input matrix.
So the important question to ask here is: Can we vectorize this entire operation?
The answer is Yes and that's exactly what im2col helps us do (which stands for Image Block to Column).

Here in fact the code implements the convolution operation in a smarter way that avoids the use of two for loops ($O(n^2)$), for efficiency reasons.
Below is reported the forward convolution function:

```
function [y] = Conv(x, w_conv, b_conv)
    [height, width,  ] = size(x); % height = 14, width = 14
    [f,  ,  , c2] = size(w_conv); % f = 3 [filter size], c2 = 3 [# of
   kernels]
    x_reshaped = reshape(x, [height width]); % Dim: 14 x 14

    lr_pad = zeros(height, 1); tb_pad = zeros(1, width+2);
    x_padded = [tb_pad; [lr_pad x_reshaped lr_pad]; tb_pad]; % Dim: 16 x 16

    x_row_version = im2col(x_padded, [f f]); % Dim: (3x3) x (14x14)

    w_reshaped = reshape(w_conv, [f*f c2]); % Dim: (3x3) x 3

    % Dim: (14x14) x 3
    y = transpose(x_row_version) * w_reshaped + transpose(b_conv);

    y = reshape(y, [height width c2]);
end
```

As we can notice, the $im2col$ is used for linearizing the convolution function in MATLAB; $im2col$ is a function that converts the input image into a matrix representation. It rearranges the image patches within the input image into columns of a matrix, where each column corresponds to a sliding window or kernel position in the original image.

By using im2col [3], the convolution operation can be transformed into a matrix multiplication operation, where the input image is represented as a matrix and the convolutional kernel as a matrix of weights. This allows for efficient computation using matrix operations, which is often more computationally optimized than nested loops when working with large datasets.

So we take each window, flatten it out and stack them as columns in a matrix. Then, if we flatten out the kernel into a row vector and do matrix multiplication between the two, we should get the exact same result after reshaping the output.
Visually the entire convolution can be seen as follows:

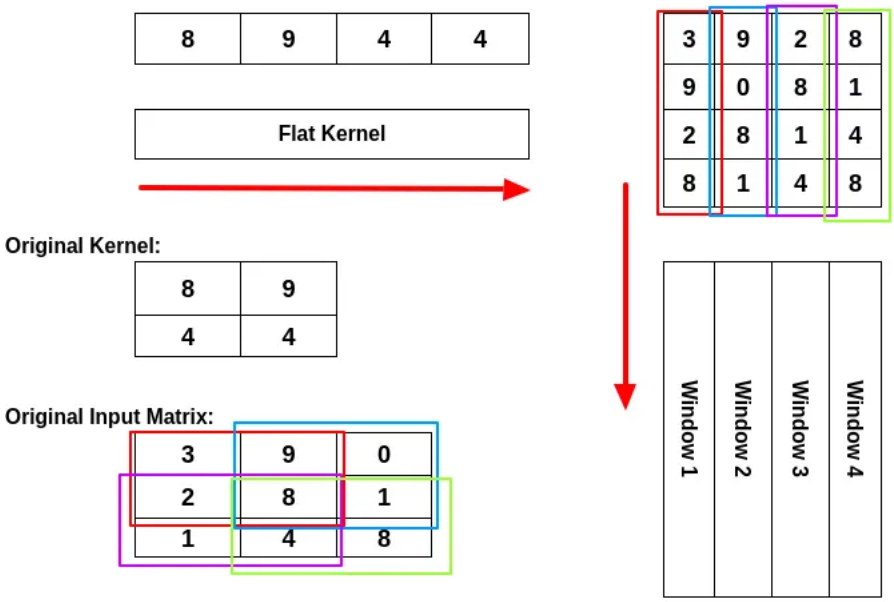


Figure 3.2: im2col to implement convolution

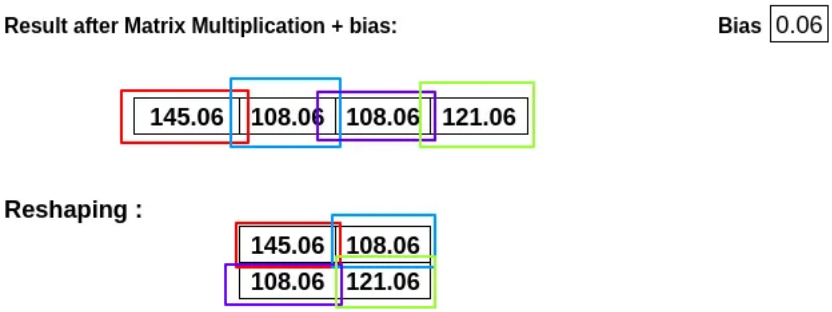


Figure 3.3: im2col resizing to original dimensions

Also note that at lines 6-7 in the code is performed the zero padding onto the input image, which helps to maintain symmetry in the convolution operation. Since the center of the kernel aligns with the center of the input image, padding ensures that the kernel is applied evenly to all pixels, including those at the borders.

3.3.2 Convolution Backward

Here we report the code corresponding to the conv_backward.m which implements the convolution during the backpropagation:

```
1 function [dLdw, dLdb] = Conv_backward(dLdy, x, w_conv, b_conv)
2     [height, width, -] = size(x); % height = 14, width = 14
3     [f, -, -, c2] = size(w_conv); % f = 3, c2 = 3
4
5     lr_pad = zeros(height, 1);
6     tb_pad = zeros(1, width+2);
```

```
    x_padded = [tb_pad; [lr_pad x lr_pad]; tb_pad]; % Dim: 16 x 16

    x_row_version = im2col(x_padded, [f f]); % Dim: (3x3) x (14x14)

    % Dim: 3 x (14x14)
    dLdy_reshaped = transpose(reshape(dLdy, [height*width c2]));

    dLdw = dLdy_reshaped * transpose(x_row_version); % Dim: 3 x (3x3)
    dLdw = reshape(dLdw, [1 c2 * f * f]); % Dim: 1 x 27

    dLdb = transpose(sum(dLdy_reshaped, 2)); % Dim.: 1 x 3
end
```

To understand the formulas that were used during the backward propagation at the convolutional phase, we had to go deeper into theory (if some of the steps below are unclear, please refer to the same theoretical steps in a hand-written form at Appendix C [6.1]). Let's start from scratch to obtain the gradients needed to backpropagate and to update weights and bias:

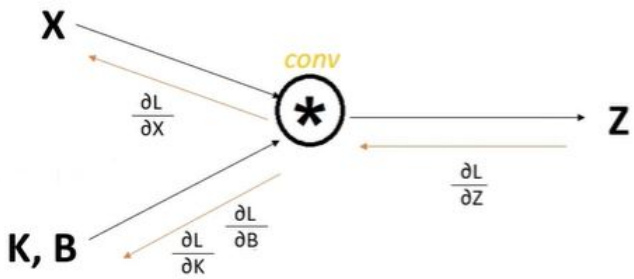


Figure 3.4: Convolution gradients to be found $\frac{\partial L}{\partial X}$, $\frac{\partial L}{\partial K}$, $\frac{\partial L}{\partial B}$

We can rewrite the above image as Convolution in matrix form, assuming 3x3 Inputs and 2x2 Kernel:

$$\begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix} \circledast \begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix} + B = \begin{bmatrix} Z_{11} & Z_{12} \\ Z_{21} & Z_{22} \end{bmatrix}$$

Where $Z_{i,j}$ can be rewritten as:

$$\begin{aligned} \mathrm{Z_{11}} &= \mathrm{X_{11}\,K_{11} + X_{12}\,K_{12} + X_{21}\,K_{21} + X_{22}\,K_{22} + B} \\ \mathrm{Z_{12}} &= \mathrm{X_{12}\,K_{11} + X_{13}\,K_{12} + X_{22}\,K_{21} + X_{23}\,K_{22} + B} \\ \mathrm{Z_{21}} &= \mathrm{X_{21}\,K_{11} + X_{22}\,K_{12} + X_{31}\,K_{21} + X_{32}\,K_{22} + B} \\ \mathrm{Z_{22}} &= \mathrm{X_{22}\,K_{11} + X_{23}\,K_{12} + X_{32}\,K_{21} + X_{33}\,K_{22} + B} \end{aligned} \tag{3.1}$$

Now, $\frac{\partial L}{\partial Z}$ is known as it was backpropagated from the next layer (in our network architecture from the pooling layer), and we need to find out three gradients:

1. $\frac{\partial L}{\partial X}$: corresponds to the gradient to be backpropagated from the conv layer to the previous one. Note that in our case, as the conv layer is just one and is at the beginning of our network structure, this therm will not be computed in the code. Anyway, we derived its mathematical formula as shown below.

2. $\frac{\partial L}{\partial K}$: corresponds to the loss derivative w.r.t kernels' weights
3. $\frac{\partial L}{\partial B}$: corresponds to the loss derivative w.r.t bias

The weights values K of the filters and of the bias B will be randomly initialized at first, then will be updated using the weights and bias update rules in back propagation:

$$K_{i+1} = K_i - \alpha \cdot \frac{dL}{dK_i} \tag{3.2}$$

$$B_{i+1} = B_i - \alpha \cdot \frac{dL}{dB_i} \tag{3.3}$$

where K_i in 3.2 represents the current kernels' weights, α is the learning rate, $\frac{dL}{dK_i}$ denotes the derivative of the loss function with respect to the kernels' weights, and K_{i+1} represents the updated kernels' weights after applying the weight update rule.
Similarly, B_i in 3.3 represents the current bias and $\frac{dL}{dB_i}$ denotes the derivative of the loss function with respect to the bias, and B_{i+1} represents the updated bias after applying the bias update rule.

So, starting from obtaining $\frac{\partial L}{\partial K}$, we can apply the chain rule:

$$\frac{\partial L}{\partial K_{mn}} = \sum \frac{\partial L}{\partial Z_{ij}} * \frac{\partial Z_{ij}}{\partial K_{mn}}$$

Expanding the formula explicitly:

$$\frac{\partial L}{\partial K_{11}} = \left(\frac{\partial L}{\partial Z_{11}} * \frac{\partial Z_{11}}{\partial K_{11}}\right) + \left(\frac{\partial L}{\partial Z_{12}} * \frac{\partial Z_{12}}{\partial K_{11}}\right) + \left(\frac{\partial L}{\partial Z_{21}} * \frac{\partial Z_{21}}{\partial K_{11}}\right) + \left(\frac{\partial L}{\partial Z_{22}} * \frac{\partial Z_{22}}{\partial K_{11}}\right)$$
$$\frac{\partial L}{\partial K_{12}} = \left(\frac{\partial L}{\partial Z_{11}} * \frac{\partial Z_{11}}{\partial K_{12}}\right) + \left(\frac{\partial L}{\partial Z_{12}} * \frac{\partial Z_{12}}{\partial K_{12}}\right) + \left(\frac{\partial L}{\partial Z_{21}} * \frac{\partial Z_{21}}{\partial K_{12}}\right) + \left(\frac{\partial L}{\partial Z_{22}} * \frac{\partial Z_{22}}{\partial K_{12}}\right)$$
$$\frac{\partial L}{\partial K_{21}} = \left(\frac{\partial L}{\partial Z_{11}} * \frac{\partial Z_{11}}{\partial K_{21}}\right) + \left(\frac{\partial L}{\partial Z_{12}} * \frac{\partial Z_{12}}{\partial K_{21}}\right) + \left(\frac{\partial L}{\partial Z_{21}} * \frac{\partial Z_{21}}{\partial K_{21}}\right) + \left(\frac{\partial L}{\partial Z_{22}} * \frac{\partial Z_{22}}{\partial K_{21}}\right)$$
$$\frac{\partial L}{\partial K_{22}} = \left(\frac{\partial L}{\partial Z_{11}} * \frac{\partial Z_{11}}{\partial K_{22}}\right) + \left(\frac{\partial L}{\partial Z_{12}} * \frac{\partial Z_{12}}{\partial K_{22}}\right) + \left(\frac{\partial L}{\partial Z_{21}} * \frac{\partial Z_{21}}{\partial K_{22}}\right) + \left(\frac{\partial L}{\partial Z_{22}} * \frac{\partial Z_{22}}{\partial K_{22}}\right)$$

Now, as $\frac{\partial L}{\partial Z}$ are known from next layer, we need to find $\frac{\partial Z}{\partial K}$.
Looking at $\frac{\partial Z_{11}}{\partial K_{11}}$, given $\mathrm{Z}_{11} = \mathrm{X}_{11}\ \mathrm{K}_{11} + \mathrm{X}_{12}\ \mathrm{K}_{12} + \mathrm{X}_{21}\ \mathrm{K}_{21} + \mathrm{X}_{22}\ \mathrm{K}_{22} + \mathrm{B}$ from 3.1, we obtain:

$$\frac{\partial Z_{11}}{\partial K_{11}} = X_{11}$$

We therefore get:

$$\frac{\partial L}{\partial K_{11}} = \frac{\partial L}{\partial Z_{11}} * X_{11} + \frac{\partial L}{\partial Z_{12}} * X_{12} + \frac{\partial L}{\partial Z_{21}} * X_{21} + \frac{\partial L}{\partial Z_{22}} * X_{22}$$
$$\frac{\partial L}{\partial K_{12}} = \frac{\partial L}{\partial Z_{11}} * X_{12} + \frac{\partial L}{\partial Z_{12}} * X_{13} + \frac{\partial L}{\partial Z_{21}} * X_{22} + \frac{\partial L}{\partial Z_{22}} * X_{23}$$
$$\frac{\partial L}{\partial K_{11}} = \frac{\partial L}{\partial Z_{11}} * X_{21} + \frac{\partial L}{\partial Z_{12}} * X_{22} + \frac{\partial L}{\partial Z_{21}} * X_{31} + \frac{\partial L}{\partial Z_{22}} * X_{32}$$
$$\frac{\partial L}{\partial K_{11}} = \frac{\partial L}{\partial Z_{11}} * X_{22} + \frac{\partial L}{\partial Z_{12}} * X_{23} + \frac{\partial L}{\partial Z_{21}} * X_{32} + \frac{\partial L}{\partial Z_{22}} * X_{33}$$

We notice from the system of equation above that $\frac{\partial L}{\partial K}$ corresponds to the convolution between X and $\frac{\partial L}{\partial Z}$:

$$\frac{\partial L}{\partial K} = \begin{bmatrix} \frac{\partial L}{\partial K_{11}} & \frac{\partial L}{\partial K_{12}} \\ \frac{\partial L}{\partial K_{21}} & \frac{\partial L}{\partial K_{22}} \end{bmatrix} = \begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix} \circledast \begin{bmatrix} \frac{\partial L}{\partial Z_{11}} & \frac{\partial L}{\partial Z_{12}} \\ \frac{\partial L}{\partial Z_{21}} & \frac{\partial L}{\partial Z_{22}} \end{bmatrix}$$

Thus, we can finally derive the equation for $\frac{\partial L}{\partial K}$:

$$\boxed{\frac{\partial L}{\partial K} = CONV(X, \frac{\partial L}{\partial Z})} \tag{3.4}$$

Next on, we can obtain the definition of $\frac{\partial L}{\partial B}$. As before we start by leveraging the chain rule:

$$\frac{\partial L}{\partial B} = \sum \frac{\partial L}{\partial Z_{ij}} * \frac{\partial Z_{ij}}{\partial B}$$

$$\Rightarrow \frac{\partial L}{\partial B} = \left(\frac{\partial L}{\partial Z_{11}} * \frac{\partial Z_{11}}{\partial B}\right) + \left(\frac{\partial L}{\partial Z_{12}} * \frac{\partial Z_{12}}{\partial B}\right) + \left(\frac{\partial L}{\partial Z_{21}} * \frac{\partial Z_{21}}{\partial B}\right) + \left(\frac{\partial L}{\partial Z_{22}} * \frac{\partial Z_{22}}{\partial B}\right)$$

As before $\frac{\partial L}{\partial Z}$ are known from next layer, so we need to find $\frac{\partial Z}{\partial B}$.
Looking at $\frac{\partial Z_{11}}{\partial B}$, given $\mathrm{Z_{11}} = \mathrm{X_{11}\ K_{11} + X_{12}\ K_{12} + X_{21}\ K_{21} + X_{22}\ K_{22} + B}$ from 3.1, we obtain:

$$\frac{\partial Z_{11}}{\partial B} = 1$$

We therefore get:

$$\frac{\partial L}{\partial B} = \left(\frac{\partial L}{\partial Z_{11}} * 1\right) + \left(\frac{\partial L}{\partial Z_{12}} * 1\right) + \left(\frac{\partial L}{\partial Z_{21}} * 1\right) + \left(\frac{\partial L}{\partial Z_{22}} * 1\right) = \frac{\partial L}{\partial Z_{11}} + \frac{\partial L}{\partial Z_{12}} + \frac{\partial L}{\partial Z_{21}} + \frac{\partial L}{\partial Z_{22}}$$

$$\Rightarrow \boxed{\frac{\partial L}{\partial B} = sum\left(\frac{\partial L}{\partial Z}\right)} \tag{3.5}$$

Lastly, we can obtain the definition of $\frac{\partial L}{\partial X}$. As before we start by leveraging the chain rule:

$$\frac{\partial L}{\partial X_{mn}} = \sum \frac{\partial L}{\partial Z_{ij}} * \frac{\partial Z_{ij}}{\partial X_{mn}}$$

Let's start by obtaining $\frac{\partial L}{\partial X_{11}}$ (note that the chain rule contain only Z_{11} because a variation in X_{11} would impact only on Z_{11}):

$$\frac{\partial L}{\partial X_{11}} = \sum \frac{\partial L}{\partial Z_{11}} * \frac{\partial Z_{11}}{\partial X_{11}}$$

As before $\frac{\partial L}{\partial Z}$ are known from next layer, so we need to find $\frac{\partial Z}{\partial X}$.
Now, looking at $\frac{\partial Z_{11}}{\partial X_{11}}$, given $\mathrm{Z_{11}} = \mathrm{X_{11}}\ \mathrm{K_{11}} + \mathrm{X_{12}}\ \mathrm{K_{12}} + \mathrm{X_{21}}\ \mathrm{K_{21}} + \mathrm{X_{22}}\ \mathrm{K_{22}} + \mathrm{B}$ from 3.1, we obtain:

$$\frac{\partial Z_{11}}{\partial X_{11}} = K_{11}$$
$$\Rightarrow \frac{\partial L}{\partial X_{11}} = \frac{\partial L}{\partial Z_{11}} * K_{11}$$

Similarly, we can obtain $\frac{\partial L}{\partial X_{12}}$ (note that the chain rule this time contains Z_{11} and Z_{12} because a variation in X_{12} would impact both on Z_{11} and Z_{12}):

$$\frac{\partial L}{\partial X_{12}} = \left(\frac{\partial L}{\partial Z_{11}} * \frac{\partial Z_{11}}{\partial X_{12}}\right) + \left(\frac{\partial L}{\partial Z_{12}} * \frac{\partial Z_{12}}{\partial X_{12}}\right) = \frac{\partial L}{\partial Z_{11}} * K_{12} + \frac{\partial L}{\partial Z_{12}} * K_{11}$$

Applying the same operation for each m, n from 1,1 to 3,3:

$$\begin{aligned}
\frac{\partial L}{\partial X_{11}} &= \frac{\partial L}{\partial Z_{11}} * K_{11} \\
\frac{\partial L}{\partial X_{12}} &= \frac{\partial L}{\partial Z_{11}} * K_{12} + \frac{\partial L}{\partial Z_{12}} * K_{11} \\
\frac{\partial L}{\partial X_{13}} &= \frac{\partial L}{\partial Z_{12}} * K_{12} \\
\frac{\partial L}{\partial X_{21}} &= \frac{\partial L}{\partial Z_{11}} * K_{21} + \frac{\partial L}{\partial Z_{21}} * K_{11} \\
\frac{\partial L}{\partial X_{22}} &= \frac{\partial L}{\partial Z_{11}} * K_{22} + \frac{\partial L}{\partial Z_{12}} * K_{21} + \frac{\partial L}{\partial Z_{21}} * K_{12} + \frac{\partial L}{\partial Z_{22}} * K_{11} \\
\frac{\partial L}{\partial X_{23}} &= \frac{\partial L}{\partial Z_{12}} * K_{22} + \frac{\partial L}{\partial Z_{22}} * K_{12} \\
\frac{\partial L}{\partial X_{31}} &= \frac{\partial L}{\partial Z_{21}} * K_{21} \\
\frac{\partial L}{\partial X_{32}} &= \frac{\partial L}{\partial Z_{21}} * K_{22} + \frac{\partial L}{\partial Z_{22}} * K_{21} \\
\frac{\partial L}{\partial X_{33}} &= \frac{\partial L}{\partial Z_{22}} * K_{22}
\end{aligned} \tag{3.6}$$

At this point, we can notice that the above system 3.6 corresponds to the convolution between a padded $\frac{\partial L}{\partial Z}$ and the filter K rotated by 180 degrees:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & \frac{\partial L}{\partial Z_{11}} & \frac{\partial L}{\partial Z_{12}} & 0 \\ 0 & \frac{\partial L}{\partial Z_{21}} & \frac{\partial L}{\partial Z_{22}} & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \circledast \begin{bmatrix} K_{22} & K_{21} \\ K_{12} & K_{11} \end{bmatrix} = \begin{bmatrix} \frac{\partial L}{\partial X_{11}} & \frac{\partial L}{\partial X_{12}} & \frac{\partial L}{\partial X_{13}} \\ \frac{\partial L}{\partial X_{21}} & \frac{\partial L}{\partial X_{22}} & \frac{\partial L}{\partial X_{23}} \\ \frac{\partial L}{\partial X_{31}} & \frac{\partial L}{\partial X_{32}} & \frac{\partial L}{\partial X_{33}} \end{bmatrix}$$

We therefore get:

$$\Rightarrow \boxed{\frac{\partial L}{\partial X} = \text{CONV}\left(\text{padded}\left(\frac{\partial L}{\partial Z}\right), 180^\circ \text{ rotated filter K}\right)} \tag{3.7}$$

As already explained above, this last $\frac{\partial L}{\partial X}$ term will not be used in the code as in our network architecture the convolutional layer is just one and is placed as first layer immediately after the inputs, so there is no need to further backpropagate back.

Code and Related Theory

Let's now associate the operations performed in the code with their relative theoretical meaning.
Recalling the conv_backward.m code [Par. 3.3.2], we are able to point out the following analogies:

- At line 5, 6, 7 it simply performs the classical zero-padding to the input.
- At line 9 it reshapes the input concatenating patches of the input image into columns by using the im2col MATLAB function, figure [3.2].
- At line 12 it performs both the reshaping of the dLdy and the transpose to match the dimension with the input xrowversion.
- At line 14 it performs the actual convolution operation exactly as found in the theory [3.4].
- At line 15 it simply rearranges dLdw, came from the pooling layer, to be returned in the expected form.
- At line 17 it performs the sum over the columns of the dLdy term exactly as found in the theory [3.5].

Chapter 4

Feasibility Study of CNN with Dual Numbers

We investigated the possibility of using the Dual numbers. Investigating on the CNN state of the art, there exist two main approaches to make a CNN ”non-linear”:

1. A first method consists in applying immediately after the linear convolution layer, another ”layer” where the linear outputs of the convolution pass through a non-linear function (e.g., ReLu, Sigmoid, tanh, ...)

2. A second method consists in a variation of the classical linear convolutional operation, where the inputs undergo a nonlinear transformation instead.

4.1 CNN with non-linear activation function

As we can see from the CNN structure of reference in figure [3.1] a non-linear activation function (ReLu) has been included as a different layer after the application of the convolutional operator. Infact a non-linear activation function is used after a convolutional layer to introduce non linearity into the network. This is important because without a non-linear activation function, convolutional layers would essentially be equivalent to a single linear layer. When the convolution is applied to an input image, a linear combination of image pixels with the corresponding kernel weights is performed.
Although this linear combination operation is useful for detecting basic patterns and features in the image, it is limited in learning complex, nonlinear relationships between features. By using a non-linear activation function allows the network to learn more complex relationships between the extracted features.

4.2 CNN with non-linear filters

Typical convolutional layers operate as **linear systems**, which limits their expressive capabilities. In order to overcome this limitation, various non-linearities have been used as activation functions inside CNNs, along with various pooling strategies.

We were able to find a paper from 2017 "*Non-linear Convolution Filters for CNN-based Learning*" [4] which tackles the challenge of enhancing convolutional methods within a computational model of the visual cortex by investigating quadratic forms using **Volterra kernels**.
These forms, representing a more diverse function space, serve as approximations for the response profiles of visual cells. The proposed **second-order convolution** approach is evaluated on CIFAR-10 and CIFAR-100 datasets and they demonstrated that a network combining linear and non-linear filters in its convolutional layers outperforms networks employing solely standard linear filters of the same architecture. This approach yields competitive results with the state-of-the-art on these datasets.

Until now and to the best of our understanding, non-linearities have predominantly been utilized through activation functions and pooling operations in different layers of CNNs. However, these non-linearities may offer a way to encode inner processes of the visual system rather than those present within the area of a receptive field. So little attention has been given to the exploration of new computational models that expand the convolution technique to non-linear forms, despite the existence of research findings in neuroscience that demonstrate the presence of non-linear operations in the response of visual cells.
The complexity of the human visual cortex reveals gaps in the application of convolution operations within CNNs. One such gap involves the investigation of **higher-order models**.

In the paper, an investigation is conducted into the possibility of adopting an alternative convolution scheme to enhance the learning capacity of CNNs by employing Volterra's theory [5]. This theory has been traditionally used to examine non-linear physiological systems and is adapted to the spatial domain. Instead of solely summing linear terms when computing a filter's response on a data patch, they propose the inclusion of non-linear terms generated by multiplicative interactions between all pairs of elements in the input data patch. By transforming the inputs using a second-order form, our aim is to increase their separability.
Consequently, convolution filters with enhanced properties in terms of selectivity and invariance are generated.

4.2.1 Volterra-based convolution

The Volterra series model is a technique used to approximate continuous functions and represent the input-output relationship of non-linear dynamical systems. It utilizes a polynomial functional expansion and can involve equations with terms of infinite orders. However, in practical implementations, truncated versions are employed by retaining terms up to a specific order, denoted as "r."

Similar to linear convolution, Volterra-based convolution applies kernels to filter the input data. The first-order Volterra kernel comprises coefficients related to the linear part of the filter. On the other hand, the second-order kernel represents the coefficients associated with quadratic interactions between two elements of the input. In a general sense, the r-th order kernel represents the weights that capture non-linear interactions between r input elements and their influence on the response.

So while given an input vector $x \in \mathbb{R}^n$, a vector for the weights w_1 and bias b, the classical input-output function of a linear filter would be:

$$\mathbf{x} = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}^T$$
$$w_1^T = \begin{bmatrix} w_1^1 & w_1^2 & \cdots & w_1^n \end{bmatrix}$$

$$y(\mathbf{x}) = \sum_{i=1}^{n} \left(w_1^i x_i\right) + b \tag{4.1}$$

In the paper they instead defined the Volterra-based convolution, expanding the previous function [4.1] in the following quadratic form:

$$\boxed{y(\mathbf{x}) = \sum_{i=1}^{n} \left(w_1^i x_i\right) + \sum_{i=1}^{n}\sum_{j=i}^{n} \left(w_2^{i,j} x_i x_j\right) + b} \tag{4.2}$$

where $w_2^{i,j}$ are the weights of the filter's second-order terms.
Expressing 4.2 in a more concise manner, we have:

$$\Rightarrow \boxed{y(\mathbf{x}) = \underbrace{\mathbf{x}^T \mathbf{w}_2 \mathbf{x}}_{\text{quadratic term}} + \underbrace{\mathbf{w}_1^T \mathbf{x}}_{\text{linear term}} + b} \tag{4.3}$$

Where for the Volterra kernel we have:

$$\mathbf{w}_2 = \begin{bmatrix} w_2^{1,1} & w_2^{1,2} & \cdots & w_2^{1,n} \\ 0 & w_2^{2,2} & \cdots & w_2^{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & w_2^{n,n} \end{bmatrix}$$

containing the coefficients $w_2^{i,j}$ of the quadratic term, and:

$$w_1^T = \begin{bmatrix} w_1^1 & w_1^2 & \cdots & w_1^n \end{bmatrix}$$

containing the coefficients w_1^i of the linear term.

4.2.2 Backward pass & Dual Numbers' Benefits

Now we got to the interesting part for our Dual Numbers use case.
To train the weights of the Volterra kernels, the authors of the paper needed to compute the gradients of the layer's output $y(x)$ [4.2], with respect to the weights w_1^i and $w_2^{i,j}$:

$$\begin{aligned} \frac{\partial y}{\partial w_1^i} &= x_i \\ \frac{\partial y}{\partial w_2^{i,j}} &= x_i x_j \end{aligned} \tag{4.4}$$

while to propagate back the error, they had to compute the gradients of the layer's output $y(x)$ with respect to the inputs x_i:

$$\frac{\partial y}{\partial x_i} = w_1^i + \sum_{k=1}^{i} \left(w_2^{k,i} x_k \right) + \sum_{k=i}^{n} \left(w_2^{i,k} x_k \right) \tag{4.5}$$

The gradients now depend on the inputs x and not only on the kernels' weights w like in the linear convolution. Also note that the gradient for the backward pass of the Volterra-based convolution were obtained explicitly **by hand** through derivation of the equation 4.2. So the authors adapted the classic backpropagation scheme to the aforementioned input-output function calculating the **ad-hoc derivative**. The computation of those derivatives were almost trivial to get, as $y(x)$ [4.2] is just a second-order polynomial refactoring of the original convolution operation [4.1].

But what if the $y(x)$ had been more complex, e.g., with an exponential/logarithm in its definition? Here is exactly the case where the automatic differentiation of the Dual Numbers would come handy; in fact, we could avoid computing explicitly the derivative of $y(x)$ during backward pass as it would come for free during the forward pass, thanks to the Dual Numbers' properties. In this way we would be independent from whatever the $y(x)$ equation is.

From what we discovered up so far, it would be ideal to have a CNN implementation where the forward pass for the convolutional operation is written in an old *C* fashion without the *im2col* trick, thus with a double for loop for iterating over the input and the kernel, something just like this (here we assumed 4x4 input, 2x2 kernel, stride=2):

```
function output = old_fashioned_convolution(input, kernel)
    [input_height, input_width] = size(input);
    kernel_height = 2;
    kernel_width = 2;

    output_height = floor((input_height - kernel_height) / 2) + 1;
    output_width = floor((input_width - kernel_width) / 2) + 1;

    output = zeros(output_height, output_width);

    for i = 1:output_height
        for j = 1:output_width
            for m = 1:kernel_height
                for n = 1:kernel_width
                    output(i, j) = output(i, j) + input(2*i+m-2, 2*j+n-2) *
    kernel(m, n);
                end
            end
        end
    end
end
```

As we can notice, in this version we have all linear operation between the inputs and kernel. It follows that if we transform the two matrices of the inputs and of the kernels' weights into

Dual Numbers Matrices, then we would end up having the derivative in the Dual part of the resulting matrix.
Finally, Volterra convolution could be implemented instead of the classical linear convolution above to exploit the non-linearity right inside the convolution operation and thus exploit the Dual Numbers' automatic differentiation.

Chapter 5

CNN with Dual Numbers Implementation

We then tried to integrate the Dual Numbers inside the original CNN implementation, in the same manner as explained in Par. 4.1, thus, applying the non-linearity with a "Sigmoid layer" right after the Convolutional layer.
To check out our implementation please refer to Appendix B [6.1].

We took the following steps:

1. First of all, we extended the *DualArray* class creating both *DualMatrix* and *DualTensor* classes as we deal with additional dimensions in CNNs (dictated by the # of kernels).

2. Redefined the Approximated Sigmoid function and used it over the original ReLu function right after the Conv layer.

```
    function [y] = Sigmoid(x)
        sz = size(x);
        y = reshape(x, [], 1);  % Reshape tensor to a column vector
        y = y./(abs(y) + 1) + 0.5;  % Apply sigmoid function element-
    wise
        y = reshape(y, sz);  % Reshape back to the original tensor
    size
        y = y.*0.5;  % Scale down the values
    end

```

3. Adapted the fully connected layer forward (*FC.m*) to be able to handle Dual Numbers. Original:

```
    function y = FC(x, w, b)
        y= w*x +b;
    end

```

   Modified:

```
function y = FC(x, w, b)
    [nRows, -] = size(w);
    y = DualArray(zeros(nRows,1),zeros(nRows,1));

    for nR = 1:nRows
        y(nR).dArr = sum(times(x, w(nR, :)')) + b(nR);
    end
end

```

4. Adjusted the **forward** pass steps inside TrainCNN to accomodate the Dual Numbers. Original:

```
%FORWARD PASS
pred1 = Conv(x, w_conv, b_conv);
pred2 = ReLu(pred1);
pred3 = Pool2x2(pred2);
pred4 = Flattening(pred3);
pred5 = FC(pred4, w_fc, b_fc);

[l, dldy] = Loss_cross_entropy_softmax(pred5, y);
loss(iIter) = loss(iIter)+l;

```

Modified:

```
% FORWARD PASS
pred1 = Conv(x, w_conv, b_conv);
pred1 = DualTensor(pred1, ones(size(pred1)));
pred2 = Sigmoid(pred1);
grad_pred2 = getDual(pred2);
pred2 = getReal(pred2);

pred3 = Pool2x2(pred2);
pred4 = Flattening(pred3);
pred4 = DualArray(pred4, ones(size(pred4)));
pred5 = FC(pred4, w_fc, b_fc);
grad_pred5 = getDual(pred5);
pred5 = getReal(pred5);

[l, dldy] = Loss_cross_entropy_softmax(pred5, y);
loss(iIter) = loss(iIter)+l;

```

5. Adjusted the *backward* pass steps inside TrainCNN to accomodate the Dual Numbers. Original:

```
% BACKWARD PASS
[dldx_fc, dldw, dldb] = FC_backward(dldy, pred4, w_fc, b_fc, pred5)
;
[dldx_flat] = Flattening_backward(dldx_fc, pred3, pred4);
[dldx_pool] = Pool2x2_backward(dldx_flat, pred2, pred3);
```

```
    [dldx_relu] = ReLu_backward(dldx_pool, pred1, pred2);
    [dldw_conv, dldb_conv] = Conv_backward(dldx_relu, x, w_conv, b_conv
    , pred1);

```

Modified:

```
    % BACKWARD PASS
    %[dldx_fc, dldw, dldb] = FC_backward(dldy, pred4, w_fc); %NOT USED
    ANYMORE
    dldx_fc = (dldy * w_fc)' * grad_pred5';
    pred4 = getReal(pred4);
    dldw = dldy' * pred4';
    dldb = dldy;

    [dldx_flat] = Flattening_backward(dldx_fc, pred3);
    %[dldx_pool] = Pool2x2_backward(dldx_flat, pred2); %NOT USED
    ANYMORE
    %[dldx_sig] = Sigmoid_backward(dldx_pool, pred1); %NOT USED ANYMORE
    dldx_sig = reshape(grad_pred2, [4,49, 3]);
    [dldw_conv, dldb_conv] = Conv_backward(dldx_sig, x, w_conv, b_conv)
    ;

```

5.1 Gradients Goodness Evaluation

In order to check whether the adjustment of the kernel weights obtained through Automatic Differentiation contains the actual derivative, we decided to compare the correction for the 1st training sample of our implementation with the correction of the original implementation which does not use Dual Numbers.
This answers the question:

which is the first correction we make? (= are the gradients correct or not?)

To do that, we must ensure that the two implementations start from the same weights for the kernels (**w_conv**) and for the fully connected layers (**w_fc**), avoiding random initialization. Doing so, we have the same reproducibility environment and thus the two corrections are comparable and we can check how much they differ. In the ideal case, we should get two indentical ***dLdw_conv***, or at least with slight changes between them.

Specifically, analyzing the code for the updating phase inside TrainCNN.m:

```
w_conv = w_conv - gamma/currBatchSize*dLdw_conv;
b_conv = b_conv - gamma/currBatchSize*dLdb_conv;
w_fc = w_fc - gamma/currBatchSize*dLdw;
b_fc = b_fc - gamma/currBatchSize*dLdb;
```

We have:

- **w_conv**: known as generated a priori with same seed (see below)

- **w_fc**: known as generated a priori with same seed (see below)
- **gamma**: known as it is the hyperparameter for the learning rate
- **currBatchSize**: equals to 1 as we are seeking for the correction made for the 1st traning sample

$\Rightarrow$ the only unknown variable to compare is then *dLdw_conv*.

So we started generating the w_conv and w_fc in the Matlab Command Window to start with the same weights:

rng(1)
w_conv = normrnd(0, 1, [3 3 1 3])
w_fc = normrnd(0, 1, [10 147])
save w_conv_and_w_fc.mat w_conv w_fc

and we modified the *main_cnn.m* to have the same random seed while generating w_conv and w_fc, and set the batch_size equal to one:

```
...
batch_size = 1;
rng(1);
load('w_conv_and_w_fc.mat');
[w_conv, b_conv, w_fc, b_fc] = TrainCNN(mini_batch_x, mini_batch_y, w_conv,
    w_fc);
...
```

After feeding the 1st training sample through the network, the two weight variation matrix (dLdw_conv) in our implementation and in the original implementation were:

```
val(:,:,1,1) =

    5.1855    5.5111    4.0009
    5.4146    4.8821    5.7624
    1.8370    1.2470    2.9945


val(:,:,1,2) =

    2.7604    2.2752    3.1967
    4.1367    5.1818    4.9322
    1.7180    2.2418    3.4135


val(:,:,1,3) =

    2.9958    4.1769    6.1999
    4.5654    2.2582    2.2639
    3.2715    3.5420    3.9134
```

Figure 5.1: weight variation matrix (dLdw_conv) for our implementation with Approximated Sigmoid

```
val(:,:,1,1) =

    1.0673    0.9127    0.5892
    0.3990    0.1531    0.5978
    0.0206    0.1118   -0.1477


val(:,:,1,2) =

   -0.0730    0.3491   -0.0618
    0.6619    0.6012    0.6283
   -0.1266    0.5035   -0.7604


val(:,:,1,3) =

    0.3788   -0.0035    0.8397
   -0.1653    0.0599   -0.1779
    0.0987   -0.9416   -1.8113
```

Figure 5.2: weight variation matrix (dLdw_conv) for the original implementation with Sigmoid

As we can notice, unfortunately our (***dLdw_conv***) differs significantly w.r.t. the original one. The magnitude of these variations is too significant to be accounted only for the fact that we are using an approximate sigmoid.
Thus, we can conclude that there exists some problems in the network workflow which makes our matrix not compliant with the actual derivatives.

5.2 Performance Evaluations

By running our modified version, we realized that it would take a very long time for the whole execution, indeed we tried our version with the initial parameters we found in the original implementation which were batch_size = 30 in the *main_cnn* and iITer = 10,000 training iterations in *Train_CNN*. Therefore, we began to investigate by training the network setting different values for both parameters to determine which portion of the program burdened the processing.

Thanks to MATLAB's "Profiler" option we were able to provide an answer to our requirement. For this purpose, we executed the code related to main up to the call of Train_cnn, that is, commenting out the test-related part present in the main_cnn function.
First of all we executed the original version of the CNN by setting batch_size = 1 in the *main_cnn* function and, as already said, limiting to 1 the number of cycles for iIter = 1:1 in the *Train_CNN* function. That is what we got:

Profile Summary (Total time: 0.333 s)

▸ **Flame Graph**

Generated 04-giu-2023 19:21:35 using performance time.

Function Name	Calls	Total Time (s)	Self Time* (s)
main_cnn	1	0.195	0.031
UnifiedAxesInteractions.createDefaultInteractions	1	0.129	0.001
…ics.interaction.internal.UnifiedAxesInteractions.createDefaultInteractionsInSync(ax,is2dim,numDataSpaces)	1	0.127	0.001
UnifiedAxesInteractions.createDefaultInteractionsInSync	1	0.126	0.004
UnifiedAxesInteractions.createDefaultInteractionsOnAxes	1	0.120	0.005
BaseAxesInteractionContainer>BaseAxesInteractionContainer.setupInteractions	1	0.109	0.006
TrainCNN	1	0.095	0.029
BaseAxesInteractionContainer>BaseAxesInteractionContainer.createDefaultAxesInteractions	1	0.080	0.002
BaseAxesInteractionContainer>BaseAxesInteractionContainer.createDefaultSetAxesInteractions	1	0.076	0.008
GetMiniBatch	1	0.056	0.056

Figure 5.3: Original profile summary, batch size = 1, number of iters = 1

[*Self time is the time spent in a function excluding any time spent in child functions. The time includes any overhead time resulting from the profiling process.]

From the profile above we can find the total processing time of just a single iteration with one image next to the *main_cnn* function under the "Total Time" column which is in the order of 0.2 seconds.
We set the same parameters for the sake of comparison in our dual version, and the following profile has been generated:

Profile Summary (Total time: 0.528 s)

▸ **Flame Graph**

Generated 04-giu-2023 19:25:02 using performance time.

Function Name	Calls	Total Time (s)	Self Time* (s)
main_cnn	1	0.528	0.034
TrainCNN	1	0.427	0.011
Sigmoid	2	0.227	0.004
FC	1	0.116	0.003
DualTensor>DualTensor.plus	2	0.091	0.057
DualArray>DualArray.times	11	0.068	0.018
Dual2>Dual2.plus	2676	0.065	0.051
DualTensor>DualTensor.rdivide	1	0.057	0.041
GetMiniBatch	1	0.054	0.054
Dual2>Dual2.mtimes	2068	0.048	0.035
DualArray>DualArray.sum	10	0.047	0.012
Dual2>Dual2.Dual2	11479	0.039	0.039

Figure 5.4: Dual profile summary, batch size = 1, number of iters = 1

The total processing time is still low and it is about 0.5 seconds, but here we can see that among the functions/operations that take most of the processing time we can find the elementary operations of the classes we introduced such as the addition and division of DualTensor class and the addition of the Dual2 class.

So we further investigated about their role in the execution; we set the batch_size to the default value of 30 images per batch but still keeping just an iteration of training.

Profile Summary (Total time: 0.377 s)

▸ **Flame Graph**

Generated 04-giu-2023 20:22:57 using performance time.

Function Name	Calls	Total Time (s)	Self Time* (s)
main_cnn	1	0.226	0.029
TrainCNN	1	0.162	0.034
UnifiedAxesInteractions.createDefaultInteractions	1	0.140	0.001
…ics.interaction.internal.UnifiedAxesInteractions.createDefaultInteractionsInSync(ax,is2dim,numDataSpaces)	1	0.138	0.000
UnifiedAxesInteractions.createDefaultInteractionsInSync	1	0.137	0.005
UnifiedAxesInteractions.createDefaultInteractionsOnAxes	1	0.129	0.005
BaseAxesInteractionContainer>BaseAxesInteractionContainer.setupInteractions	1	0.119	0.007
BaseAxesInteractionContainer>BaseAxesInteractionContainer.createDefaultAxesInteractions	1	0.085	0.002
BaseAxesInteractionContainer>BaseAxesInteractionContainer.createDefaultSetAxesInteractions	1	0.080	0.009
newplotwrapper	1	0.041	0.001
newplot	1	0.039	0.004
Pool2x2_backward	30	0.036	0.007
im2col	240	0.035	0.016

Figure 5.5: Original profile summary, batch size = 30, number of iters = 1

By increasing the number of images in the batch the original version still remains in the order of few tenths of seconds.
The same does not happen for the dual version:

Profile Summary (Total time: 7.043 s)

▸ **Flame Graph**

Generated 04-giu-2023 20:25:38 using performance time.

Function Name	Calls	Total Time (s)	Self Time* (s)
main_cnn	1	7.043	0.051
TrainCNN	1	6.956	0.072
Sigmoid	60	4.110	0.012
FC	30	2.191	0.013
DualTensor>DualTensor.plus	60	1.835	1.138
DualTensor>DualTensor.rdivide	30	1.326	0.968
DualArray>DualArray.times	330	1.307	0.273
Dual2>Dual2.plus	80280	1.298	1.011
Dual2>Dual2.mtimes	62040	0.942	0.661
DualArray>DualArray.sum	300	0.871	0.203
Dual2>Dual2.Dual2	344370	0.815	0.815
DualTensor>DualTensor.DualTensor	53910	0.485	0.405

Figure 5.6: Dual profile summary, batch size = 30, number of iters = 1

We got roughly 7 seconds. Most of the execution time is taken up by not only the elementary operations of both DualTensor and Dual2 classes, due to the fact that these operations are implemented inside nested for loops (as expected, since the DualTensor operations iterate over 2 more dimensions w.r.t. the operations of the Dual Array class) which in principle are non-optimized constructs in MATLAB, but also by the repeated calls to the class constructors.

So, considering the initial number iIter= 10,000 iterations and 30 images per batch, an estimate of the overall time that would be needed to execute a complete training of the network can be calculated: if a single iteration of a batch_size of 30 images takes about 7 seconds, multiplying to the default number of iIter it would take something like 19 hours of execution that, as one can imagine, is quite a large time.

Chapter 6

Conclusions

For our project we started looking for a basic implementation of a CNN freely available. We found the [6.1] (Appendix A) CNN implementation but however we encountered an initial challenge because of the lack of documentation of the implementation. To overcome this obstacle, we conducted reverse engineering to interpret and understand the steps. Once we gained a good understanding of the implementation, our next goal was to create a documentation that aligned with the code. We wanted to ensure that the documentation was coherent, making it easier for others to reuse and understand the code.

In our code, we have extended the DualArray class into a series of classes called DualMatrix and DualTensor that allow us to perform arithmetic operations between a dual matrix and double, a dual matrix and Dual2 number and also between dual matrices. Finally, we have modified the starting version of the CNN implementation to let the non-linear activation functions use the DualTensor arithmetic operations.
Additionally, we deepened into studying non-linearities within CNNs. During our research, we came across a paper on Volterra about Non-linear convolution filters for the CNN convolutional layer, which provided valuable insights and information on this topic.

In conclusion, we conducted a comparison for kernel weight adjustments between the correction applied to the training samples in our dual implementation and the correction applied in the original implementation. This comparison aimed to assess whether the adjustment obtained through Automatic Differentiation accurately captures the actual derivative.

6.1 Future Works

For future works, it would be best to implement a Convolutional Neural Network from scratch where the forward pass for the convolutional operation is in a traditional C-style manner without utilizing the im2col trick. It would employ a double for loop to iterate over the input and kernel matrices. With this approach, all operations between the inputs and kernel would be linear.

So, by preliminarily transforming both the input and kernel weight matrices into Dual Matrices (already implemented), we would obtain the derivative in the Dual part of the resulting matrix. To further enhance the convolution operation, Volterra-based convolution could be implemented instead of the traditional linear convolution. This would allow us to leverage the inherent non-linearity within the convolution operation and exploit the automatic differentiation capabilities of Dual Numbers (as widely discussed in Par. 4.2.2).

Appendix A - CNN implementation in Matlab, by Aditya Gaydhani

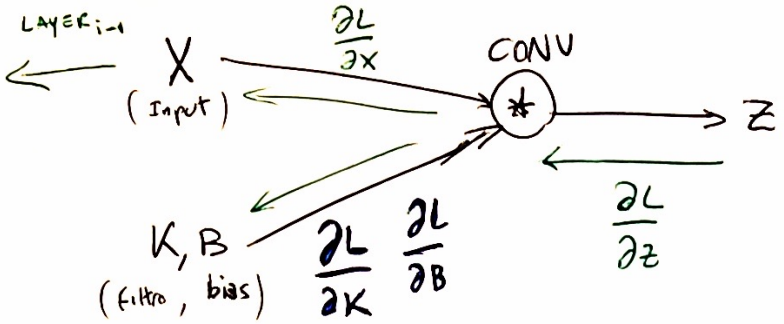
https://github.com/adityagaydhani/CSCI-5561-CNN-implementation-from-scratch

Appendix B - CNN implementation with Dual Numbers in Matlab

https://github.com/enricollen/CNN-Dual-Numbers

Appendix C - Notes for Deriving CNN Backpropagation formulas

CONVOLUZIONE NELLA BACKPROPAGATION



UPDATE

$$\begin{cases} W_{i+1} = W_i - \alpha \cdot \frac{\partial L}{\partial W} \\ K_{i+1} = K_i - \alpha \frac{\partial L}{\partial K} \\ B_{i+1} = B_i - \alpha \frac{\partial L}{\partial B} \end{cases}$$

$$\begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix} \circledast \begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix} + B = \begin{bmatrix} Z_{11} & Z_{12} \\ Z_{21} & Z_{22} \end{bmatrix}$$

$$\begin{aligned} Z_{11} &= X_{11}K_{11} + X_{12}K_{12} + X_{21}K_{21} + X_{22}K_{22} + B \\ Z_{12} &= X_{12}K_{11} + X_{13}K_{12} + X_{22}K_{21} + X_{23}K_{22} + B \\ Z_{21} &= X_{21}K_{11} + X_{22}K_{12} + X_{31}K_{21} + X_{32}K_{22} + B \\ Z_{22} &= X_{22}K_{11} + X_{23}K_{12} + X_{32}K_{21} + X_{33}K_{22} + B \end{aligned}$$

*: una variazione di K_{11} va a influire su $Z_{11}, Z_{12}, Z_{21}, Z_{22}$

Nota $\frac{\partial L}{\partial Z}$ dal layer precedente:

1) $\frac{\partial L}{\partial K} = \frac{\partial L}{\partial Z} \cdot \frac{\partial Z}{\partial K}$ (· ↑ matrici)

con $\frac{\partial L}{\partial K} = \begin{bmatrix} \frac{\partial L}{\partial K_{11}} & \frac{\partial L}{\partial K_{12}} \\ \frac{\partial L}{\partial K_{21}} & \frac{\partial L}{\partial K_{22}} \end{bmatrix}$

Einstein Notation per la Chain Rule

$$\frac{\partial L}{\partial K_{mxn}} = \frac{\partial L}{\partial Z_{ij}} \cdot \frac{\partial Z_{ij}}{\partial K_{mn}} = \sum \frac{\partial L}{\partial Z_{ij}} = \frac{\partial Z_{ij}}{\partial K_{mn}}$$

$$\Rightarrow \frac{\partial L}{\partial K_{11}} \overset{*}{=} \left(\frac{\partial L}{\partial Z_{11}} \cdot \frac{\partial Z_{11}}{\partial K_{11}}\right) + \left(\frac{\partial L}{\partial Z_{12}} \cdot \frac{\partial Z_{12}}{\partial K_{11}}\right) + \left(\frac{\partial L}{\partial Z_{21}} \cdot \frac{\partial Z_{21}}{\partial K_{11}}\right) + \left(\frac{\partial L}{\partial Z_{22}} \cdot \frac{\partial Z_{22}}{\partial K_{11}}\right)$$

$$\frac{\partial L}{\partial K_{12}} = \left(// \cdot \frac{//}{\partial K_{12}}\right) + \left(// \cdot \frac{//}{\partial K_{12}}\right) + \left(// \cdot \frac{//}{\partial K_{12}}\right) + \left(// \cdot \frac{//}{\partial K_{12}}\right)$$

$$\frac{\partial L}{\partial K_{21}} = \left(// \cdot \frac{//}{\partial K_{21}}\right) + \left(// \cdot \frac{//}{\partial K_{21}}\right) + \left(// \cdot \frac{//}{\partial K_{21}}\right) + \left(// \cdot \frac{//}{\partial K_{21}}\right)$$

$$\frac{\partial L}{\partial K_{22}} = \left(// \cdot \frac{//}{\partial K_{22}}\right) + \left(// \cdot \frac{//}{\partial K_{22}}\right) + \left(// \cdot \frac{//}{\partial K_{22}}\right) + \left(// \cdot \frac{//}{\partial K_{22}}\right)$$

$$\frac{\partial z_{11}}{\partial k_{11}} = ? \quad \text{con} \quad z_{11} = x_{11}k_{11} + x_{12}k_{12} + x_{21}k_{21} + x_{22}k_{22} + B$$

$$\Rightarrow \boxed{\frac{\partial z_{11}}{\partial k_{11}} = x_{11}}$$

Segue:

$$\frac{\partial L}{\partial k_{11}} = \frac{\partial L}{\partial z_{11}} \cdot x_{11} + \frac{\partial L}{\partial z_{12}} \cdot x_{12} + \frac{\partial L}{\partial z_{21}} \cdot x_{21} + \frac{\partial L}{\partial z_{22}} \cdot x_{22}$$

$$\frac{\partial L}{\partial k_{12}} = \frac{\partial L}{\partial z_{11}} \cdot x_{12} + \frac{\partial L}{\partial z_{12}} \cdot x_{13} + \frac{\partial L}{\partial z_{21}} \cdot x + \frac{\partial L}{\partial z_{22}} \cdot x$$

$$\frac{\partial L}{\partial k_{21}} = \frac{\partial L}{\partial z_{11}} \cdot x_{21} + \frac{\partial L}{\partial z_{12}} \cdot x_{22} + \frac{\partial L}{\partial z_{21}} \cdot x_{31} + \frac{\partial L}{\partial z_{22}} \cdot x_{32}$$

$$\frac{\partial L}{\partial k_{22}} = \frac{\partial L}{\partial z_{11}} \cdot x_{22} + \frac{\partial L}{\partial z_{12}} \cdot x_{23} + \frac{\partial L}{\partial z_{21}} \cdot x_{32} + \frac{\partial L}{\partial z_{22}} \cdot x_{33}$$

$$\Rightarrow \frac{\partial L}{\partial k} = \begin{bmatrix} \frac{\partial L}{\partial k_{11}} & \frac{\partial L}{\partial k_{12}} \\ \frac{\partial L}{\partial k_{21}} & \frac{\partial L}{\partial k_{22}} \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} \circledast \begin{bmatrix} \frac{\partial L}{\partial z_{11}} & \frac{\partial L}{\partial z_{12}} \\ \frac{\partial L}{\partial z_{21}} & \frac{\partial L}{\partial z_{22}} \end{bmatrix}$$

$$\underset{4}{X} \qquad \underset{11}{\frac{\partial L}{\partial z}}$$

$$\boxed{\frac{\partial L}{\partial k} \triangleq \text{CONV}\left(X, \frac{\partial L}{\partial z}\right)}$$

con $\frac{\partial L}{\partial z}$ = gradiente dal layer successivo ottenuto tramite backprop

② $\frac{\partial L}{\partial B} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial B} = \sum \frac{\partial L}{\partial z_{ij}} \cdot \frac{\partial z_{ij}}{\partial B}$

= nodo

$\frac{\partial L}{\partial B} = \left(\frac{\partial L}{\partial z_{11}} \cdot \frac{\partial z_{11}}{\partial B}\right) + \left(\frac{\partial L}{\partial z_{12}} \cdot \frac{\partial z_{12}}{\partial B}\right) + \left(\frac{\partial L}{\partial z_{21}} \cdot \frac{\partial z_{21}}{\partial B}\right) + \left(\frac{\partial L}{\partial z_{22}} \cdot \frac{\partial z_{22}}{\partial B}\right)$

$\frac{\partial z_{11}}{\partial B} = ?$ con $z_{11} = x_{11}k_{11} + x_{12}k_{12} + x_{21}k_{21} + x_{22}k_{22} + B$

$\Rightarrow \boxed{\frac{\partial z_{11}}{\partial B} = 1}$

Segue:

$\frac{\partial L}{\partial B} = \left(\frac{\partial L}{\partial z_{11}} \cdot 1\right) + \left(\frac{\partial L}{\partial z_{12}} \cdot 1\right) + \left(\frac{\partial L}{\partial z_{21}} \cdot 1\right) + \left(\frac{\partial L}{\partial z_{22}} \cdot 1\right)$

$\Rightarrow \frac{\partial L}{\partial B} = \frac{\partial L}{\partial z_{11}} + \frac{\partial L}{\partial z_{12}} + \frac{\partial L}{\partial z_{21}} + \frac{\partial L}{\partial z_{22}}$

$\Rightarrow \boxed{\frac{\partial L}{\partial B} = \text{sum}\left(\frac{\partial L}{\partial z}\right)}$, con $\frac{\partial L}{\partial z}$ = gradiente del layer succ. ottenuto tramite backprop.

(3) $\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Z} \cdot \frac{\partial Z}{\partial X} \rightarrow \frac{\partial L}{\partial X_{mn}} = \sum \frac{\partial L}{\partial Z_{ij}} \cdot \frac{\partial Z_{ij}}{\partial X_{mn}}$

$\frac{\partial L}{\partial X_{11}} = ?$

*: una variazione di X_{11} andrà a incidere solo e soltanto su Z_{11}!

$$\begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix} * \begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix} = \begin{bmatrix} Z_{11} & Z_{12} \\ Z_{21} & Z_{22} \end{bmatrix}$$

$\frac{\partial L}{\partial X_{11}} \overset{*}{=} \frac{\partial L}{\partial Z_{11}} \cdot \frac{\partial Z_{11}}{\partial X_{11}}$

* = noto

**: una variaz. di X_{12} andrà a incidere su Z_{11} e Z_{12}!

$\frac{\partial Z_{11}}{\partial X_{11}} = ?$, con $Z_{11} = X_{11}K_{11} + X_{12}K_{12} + X_{21}K_{21} + X_{22}K_{22} + B$

$\frac{\partial Z_{11}}{\partial X_{11}} = K_{11}$

$\Rightarrow \frac{\partial L}{\partial X_{11}} = \frac{\partial L}{\partial Z_{11}} \cdot K_{11}$

- Similmente:

$\frac{\partial L}{\partial X_{12}} \overset{**}{=} \left(\frac{\partial L}{\partial Z_{11}} \cdot \underbrace{\frac{\partial Z_{11}}{\partial X_{12}}}_{K_{12}}\right) + \left(\frac{\partial L}{\partial Z_{12}} \cdot \underbrace{\frac{\partial Z_{12}}{\partial X_{12}}}_{K_{11}}\right)$, con $\begin{cases} Z_{11} = X_{11}K_{11} + X_{12}K_{12} + X_{21}K_{21} + X_{22}K_{22} \\ Z_{12} = X_{12}K_{11} + X_{13}K_{12} + X_{22}K_{21} + X_{23}K_{22} \end{cases}$

$= \frac{\partial L}{\partial Z_{11}} \cdot K_{12} + \frac{\partial L}{\partial Z_{12}} \cdot K_{11}$

$\frac{\partial L}{\partial X_{13}} = \frac{\partial L}{\partial Z_{12}} \cdot K_{12}$

$\vdots$

$\frac{\partial L}{\partial X_{22}} = \frac{\partial L}{\partial Z_{11}} \cdot K_{22} + \frac{\partial L}{\partial Z_{12}} \cdot K_{21} + \frac{\partial L}{\partial Z_{21}} \cdot K_{12} + \frac{\partial L}{\partial Z_{22}} \cdot K_{11}$

. se faccio $\forall\ m,n \longrightarrow 1,1 \longrightarrow 3,3$:

$$\frac{\partial L}{\partial X_{11}} = \frac{\partial L}{\partial Z_{11}} \cdot K_{11}$$

$$\frac{\partial L}{\partial X_{12}} = \frac{\partial L}{\partial Z_{11}} \cdot K_{12} + \frac{\partial L}{\partial Z_{12}} \cdot K_{11}$$

$$\frac{\partial L}{\partial X_{13}} = \frac{\partial L}{\partial Z_{12}} \cdot K_{12}$$

$$\frac{\partial L}{\partial X_{21}} = \frac{\partial L}{\partial Z_{11}} \cdot K_{21} + \frac{\partial L}{\partial Z_{21}} \cdot K_{11}$$

$$\frac{\partial L}{\partial X_{22}} = \frac{\partial L}{\partial Z_{11}} \cdot K_{22} + \frac{\partial L}{\partial Z_{12}} \cdot K_{21} + \frac{\partial L}{\partial Z_{21}} \cdot K_{12} + \frac{\partial L}{\partial Z_{22}} \cdot K_{11}$$

$$\frac{\partial L}{\partial X_{23}} = \frac{\partial L}{\partial Z_{12}} \cdot K_{22} + \frac{\partial L}{\partial Z_{22}} \cdot K_{12}$$

$$\frac{\partial L}{\partial X_{31}} = \frac{\partial L}{\partial Z_{21}} \cdot K_{21}$$

$$\frac{\partial L}{\partial X_{32}} = \frac{\partial L}{\partial Z_{21}} \cdot K_{22} + \frac{\partial L}{\partial Z_{22}} \cdot K_{21}$$

$$\frac{\partial L}{\partial X_{33}} = \frac{\partial L}{\partial Z_{22}} \cdot K_{22}$$

. Noto che le eq. sopra corrisp. alla conv. seguente:

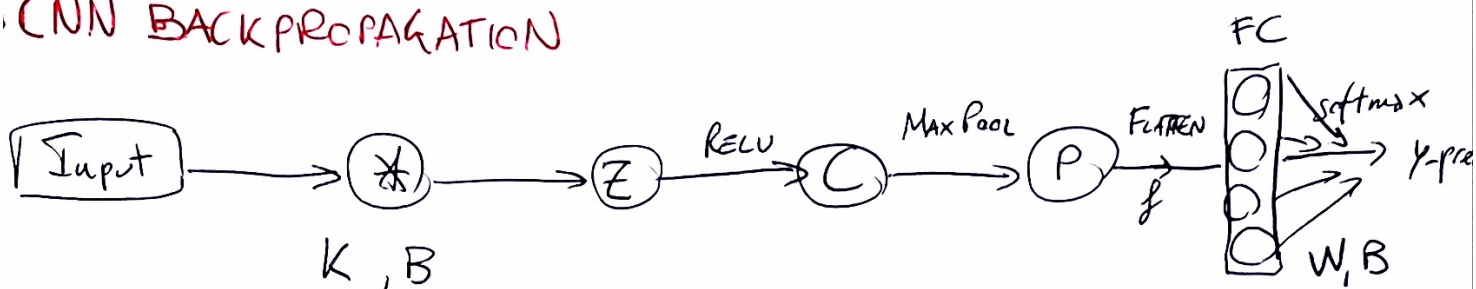
$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & \frac{\partial L}{\partial Z_{11}} & \frac{\partial L}{\partial Z_{12}} & 0 \\ 0 & \frac{\partial L}{\partial Z_{21}} & \frac{\partial L}{\partial Z_{22}} & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \circledast \begin{bmatrix} K_{22} & K_{21} \\ K_{12} & K_{11} \end{bmatrix} = \begin{bmatrix} \frac{\partial L}{\partial X_{11}} & \frac{\partial L}{\partial X_{12}} & \frac{\partial L}{\partial X_{13}} \\ \frac{\partial L}{\partial X_{21}} & \frac{\partial L}{\partial X_{22}} & \frac{\partial L}{\partial X_{23}} \\ \frac{\partial L}{\partial X_{31}} & \frac{\partial L}{\partial X_{32}} & \frac{\partial L}{\partial X_{33}} \end{bmatrix}$$

padded $\frac{\partial L}{\partial Z}$

$$\begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix} \xrightarrow{180^\circ} \begin{bmatrix} K_{22} & K_{21} \\ K_{12} & K_{11} \end{bmatrix}$$

$$\Rightarrow \boxed{\frac{\partial L}{\partial X} = \text{conv}\left(\text{padded}\left(\frac{\partial L}{\partial Z}\right), \text{180° rotated filter } K\right)}$$

CNN BACKPROPAGATION



$\frac{\partial L}{\partial K}$ $\frac{\partial L}{\partial B}$ $\frac{\partial L}{\partial W}$ $\frac{\partial L}{\partial B^{[2]}}$

FORWARD

- $Z = \text{Conv}(\text{Input}, K) + B$
- $C = \text{ReLU}(Z)$
- $P = \text{MAX POOL}(C)$
- $f = \text{FLATTEN}(P)$
- $Z^{[2]} = W \cdot f + B^{[2]}$
- $y\text{-pred} = \text{softmax}(Z^{[2]})$

OBBIETTIVO

$$\begin{cases} \frac{\partial L}{\partial W} = ? \quad , \quad \frac{\partial L}{\partial B^{[2]}} = ? \\ \frac{\partial L}{\partial K} = ? \quad , \quad \frac{\partial L}{\partial B} = ? \end{cases}$$

Weights & Bias Update:

$$\Longrightarrow \begin{cases} W_{i+1} = W_i - \alpha \cdot \frac{\partial L}{\partial W_i} \\ B^{[2]}_{i+1} = B^{[2]}_i - \alpha \frac{\partial L}{\partial B^{[2]}_i} \end{cases} \quad \begin{cases} K_{i+1} = K_i - \alpha \frac{\partial L}{\partial K_i} \\ B_{i+1} = B_i - \alpha \frac{\partial L}{\partial B_i} \end{cases}$$

BACKPROP

(per un singolo sample)

- Noto dall' MLP che:

$$\begin{cases} dZ_{out} = \frac{\partial L}{\partial Z} = (y\text{-pred} - y) \\ dW = \frac{\partial L}{\partial W} = dZ \cdot A_2 \quad \rightarrow \text{output layer prec.} \\ dB = \frac{\partial L}{\partial B} = dZ_{out} \\ dZ_2 = (W^T \cdot dZ_{out}) \cdot f_2'(Z_2) \end{cases}$$

- Ottengo:

$$\frac{\partial L}{\partial W} = dZ * f^T$$

$$\frac{\partial L}{\partial B} = dZ_{out}$$

$$\Longrightarrow \partial f = \left(\frac{\partial L}{\partial f}\right) = (W^T * dZ_{out}) \cdot 1 = W^T * dZ_{out}$$

BACKPROP FLATTEN LAYER

$\Rightarrow \frac{\partial L}{\partial P} = ?$

$f = \text{flatten}(P)$ (f e P hanno stessi elem. ma dim. diverse)

$$\frac{\partial L}{\partial P} = df.\text{reshape}(P.\text{shape})$$

$\Rightarrow \frac{\partial L}{\partial C} = ?$

BACKPROP MAX POOLING

Esempio:

$C = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \longrightarrow P = [4]$ Assumo $\longrightarrow \frac{\partial L}{\partial P} = [2]$

(max)

↳ per questi valori non cambio $P \rightarrow \frac{\partial L}{\partial C} = \emptyset$

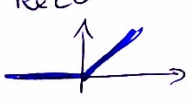
$\Rightarrow \frac{\partial L}{\partial C} = \begin{bmatrix} 0 & 0 \\ 0 & 2 \end{bmatrix}$

$$\Rightarrow \frac{\partial L}{\partial C_{mn}} = \begin{cases} \frac{\partial L}{\partial P_{xy}}, & \text{se } C_{mn} \text{ è l'elem max} \\ \emptyset, & \text{altrimenti} \end{cases}$$

BACKPROP RELU LAYER

$\Rightarrow \frac{dL}{\partial Z} = ?$

$C = ReLU(Z)$

$\frac{\partial L}{\partial Z} = \frac{\partial L}{\partial C} * \frac{\partial C}{\partial Z}$ $\longrightarrow$ 

$\longrightarrow$ 

Esempio:

$Z = \begin{bmatrix} 1.02 & -0.25 \\ -3.45 & 2.2 \end{bmatrix}$

↳ $\frac{\partial C}{\partial Z} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

$$\frac{\partial C}{\partial z_{mn}} = \begin{cases} 1 & \text{se } z_{mn} > 0 \\ 0 & \text{se } z_{mn} < 0 \end{cases}$$

$\Rightarrow \frac{\partial L}{\partial z} = \frac{\partial L}{\partial C} \cdot \frac{\partial C}{\partial z}$, con $\frac{\partial L}{\partial C}$ noto dal layer prec.

dot product

. Dalla backprop. per l'operaz. conv so che:

$dK = \frac{\partial L}{\partial K} = \text{CONV}(X, \frac{\partial L}{\partial z})$

$dB = \frac{\partial L}{\partial B} = \text{sum}(dz)$

$\Rightarrow$ Ottenuto tutti i grad., posso alla weight & bias update.

References

[1] Nan Cui. Applying gradient descent in convolutional neural networks. *Journal of Physics: Conference Series*, 1004(1):012027, apr 2018.

[2] CNN's backpropagation algorithm idea. https://www.cnblogs.com/pinard/p/6494810.html.

[3] How Are Convolutions Actually Performed Under the Hood? https://towardsdatascience.com/how-are-convolutions-actually-performed-under-the-hood-226523ce7fb

[4] Georgios Zoumpourlis, Alexandros Doumanoglou, Nicholas Vretos, and Petros Daras. Non-linear convolution filters for cnn-based learning, 2017.

[5] RE Langer. Vito volterra, theory of functionals and of integral and integro-differential equations. 1932.