University of Thessaly

Neuro-Fuzzy Computing

ECE447

2nd Problem Set

Alexandra Gianni ID: 3382

Nikos Stylianou ID: 2917

February 1, 2024

Problem 1

In this exercise we need to find the minimum of the given 2-dimensional function:

$$F(\mathbf{w}) = w_1^2 + w_2^2 + (0.5w_1 + w_2)^2 + (0.5w_1 + w_2)^4 \tag{1}$$

with the Conjugate Gradient (Fletcher-Reeves) method.

Initially, we can conclude that the function $F(w)$ is not in quadratic form because of the term $(0.5w_1 + w_2)^4$. A function is said to be in quadratic form if it can be expressed as a second-degree polynomial where all the terms are either squared terms or cross-products of the variables. The presence of the fourth-degree term $(0.5w_1 + w_2)^4$. makes this function a higher-degree polynomial, specifically a quartic function with respect to $(0.5w_1 + w_2)$, which means it cannot be classified as quadratic.
Also, the independent values in this function are w_1, w_2, because only with them we can manipulate the $F(w)$.

As an initial guess we have $w\,(0) = [3, 3]^T$.
The steps we have to use are specific for each iteration

FIRST ITERATION k = 0

Step1: Calculate the Gradient at $w\,(k)$

$$\nabla f(w_1, w_2) = \begin{pmatrix} \frac{\partial f}{\partial w_1} \\ \frac{\partial f}{\partial w_2} \end{pmatrix} = \begin{pmatrix} 2w_1 + (0.5w_1 + w_2) + 2(0.5w_1 + w_2)^3 \\ 2w_2 + 2(0.5w_1 + w_2) + 4(0.5w_1 + w_2)^3 \end{pmatrix} = \begin{pmatrix} 2.5w_1 + w_2 + 2(0.5w_1 + w_2)^3 \\ w_1 + 4w_2 + 4(0.5w_1 + w_2)^3 \end{pmatrix}$$

where at the point $w\,(0) = [3, 3]^T$ we have $\nabla f(x) = \begin{pmatrix} -53 \\ -19 \end{pmatrix}$

Problem 2

Problem 3

For the given neural network, we have:

- learning rate $LR = 1$,
- $w^1\,(0) = -3,\ w^2\,(0) = -1$,
- $b^1\,(0) = 2,\ b^2\,(0) = -1$ and
- input/target pair $\{p = 1,\ t = 0\}$

FIRST ITERATION

Step 1: Calculate first layer's output

$$n^1 = w^1 p + b^1 = (-3)(1) + 2 = -1$$
$$a^1 = Swish\left(n^1\right) = Swish\left(-1\right) = \frac{n^1}{1+e^{-n^1}} = \frac{-1}{1+e} = -0.2689$$

Step 2: Calculate second layer's output

$$n^2 = w^2 a^1 + b^2 = (-1)(-0.2689) + (-1) = -0.7311$$
$$a^2 = LReLU\left(n^2\right) = LReLU\left(-0.7311\right) = -0.000731$$

Step 3: Calculate error

$$e = t - a^2 = (0 - (-0.000731)) = 0.000731$$

Step 4: Calculate sensitivity on second layer

$$s^2 = -2\ LReLU'\left(n^2\right)\left(t - a^2\right) = -2\left(0.001\right)\left(0.000731\right) = -1.462\text{e}{-6}$$

LReLU's derivative is 1 *for* $x > 0$ *and* 0.001 *for* $x < 0$.

Step 5: Calculate sensitivity on first layer using back-propagation

$$s^1 = Swish'\left(n^1\right)\left(w^2\right)^T s^2 = Swish'\left(-1\right)\left(-1\right)\left(-1.462\text{e}{-6}\right) = 0.0723(-1)(-1.462\text{e}{-6})$$
$$s^1 = 1.0570\text{e}{-7}$$

Step 6: Update wheights and biases

$$w^2(1) = w^2(0) - LR\ s^2(a^1)^T = -1 - 1(-1.462\text{e}{-6})(-0.2689) \approx -1$$
$$b^2(1) = b^2(0) - LR\ s^2 = -1 - 1(-1.462\text{e}{-6}) \approx -1$$
$$w^1(1) = w^1(0) - LR\ s^1(a^0)^T = -3 - 1(1.0570\text{e}{-7})(-1) \approx -3$$
$$b^1(1) = b^1(0) - LR\ s^1 = 2 - 1(1.0570\text{e}{-7}) \approx 2$$

Since there were no changes on the biases and weights, the next iteration will not change the parameters of the given neural network, but we will calculate them anyway.

SECOND ITERATION

Step 1:

$$n^1 = w^1 p + b^1 = (-3)(1) + 2 = -1$$
$$a^1 = Swish\left(n^1\right) = Swish\left(-1\right) = \frac{n^1}{1+e^{-n^1}} = \frac{-1}{1+e} = -0.2689$$

Step 2:

$$n^2 = w^2 a^1 + b^2 = (-1)(-0.2689) + (-1) = -0.7311$$
$$a^2 = LReLU\left(n^2\right) = LReLU\left(-0.7311\right) = -0.000731$$

Step 3:

$$e = t - a^2 = (0 - (-0.000731)) = 0.000731$$

Step 4:

$$s^2 = -2\ LReLU'\left(n^2\right)\left(t - a^2\right) = -2\left(0.001\right)\left(0.000731\right) = -1.462\text{e}{-6}$$

Step 5:

$$s^1 = Swish'\left(n^1\right)\left(w^2\right)^T s^2 = Swish'\left(-1\right)\left(-1\right)\left(-1.462\text{e}{-6}\right) = 0.0723(-1)(-1.462\text{e}{-6})$$
$$s^1 = 1.0570\text{e}{-7}$$

Step 6:

$$w^2(1) = w^2(0) - LR\, s^2(a^1)^T = -1 - 1(-1.462\text{e}{-6})(-0.2689) \approx -1$$
$$b^2(1) = b^2(0) - LR\, s^2 = -1 - 1(-1.462\text{e}{-6}) \approx -1$$
$$w^1(1) = w^1(0) - LR\, s^1(a^0)^T = -3 - 1(1.0570\text{e}{-7})(-1) \approx -3$$
$$b^1(1) = b^1(0) - LR\, s^1 = 2 - 1(1.0570\text{e}{-7}) \approx 2$$

Problem 4

In this problem, we are asked to write a program that implements backpropagation algorithm for an $1-S^1-1$ network with $S = \{2, 8, 12\}$, as shown in figure 1.
The first layer has $logsig$ as activation function and the output layer has $ReLU$ as activation function. Also, every weight and bias is initialized to a uniformly random number in $(-0.5, 0.5)$. All of the above are done in order to train our network to approximate the following function:

$$g(p) = 1 + e^{p\left(\frac{3\pi}{8}\right)}, \qquad p \in [-2, 2]$$

This means that, during training, we have to train the network for multiple input data (*specifically, for all input data*).

Problem 5

Problem 6

Problem 7

A continuous piecewise linear function is a function that is linear on every segment of its domain.
To show that a Multi-Layer Perceptron (MLP) using only the ReLU (Rectified Linear Unit) or pReLU (Parametric Rectified Linear Unit) activation functions constructs a continuous linear function, we must first review the properties of these activation functions.

Let's consider the ReLU activation function for this explanation.
The ReLU activation function is defined as:

$$\text{ReLU}(x) = \max(x, 0) = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{otherwise}, \end{cases} \tag{2}$$

We need to check if they meet the prerequisites of continuity and linearity.

- Is it Continuous?
  Yes it is, because it has no break points for the various values of x

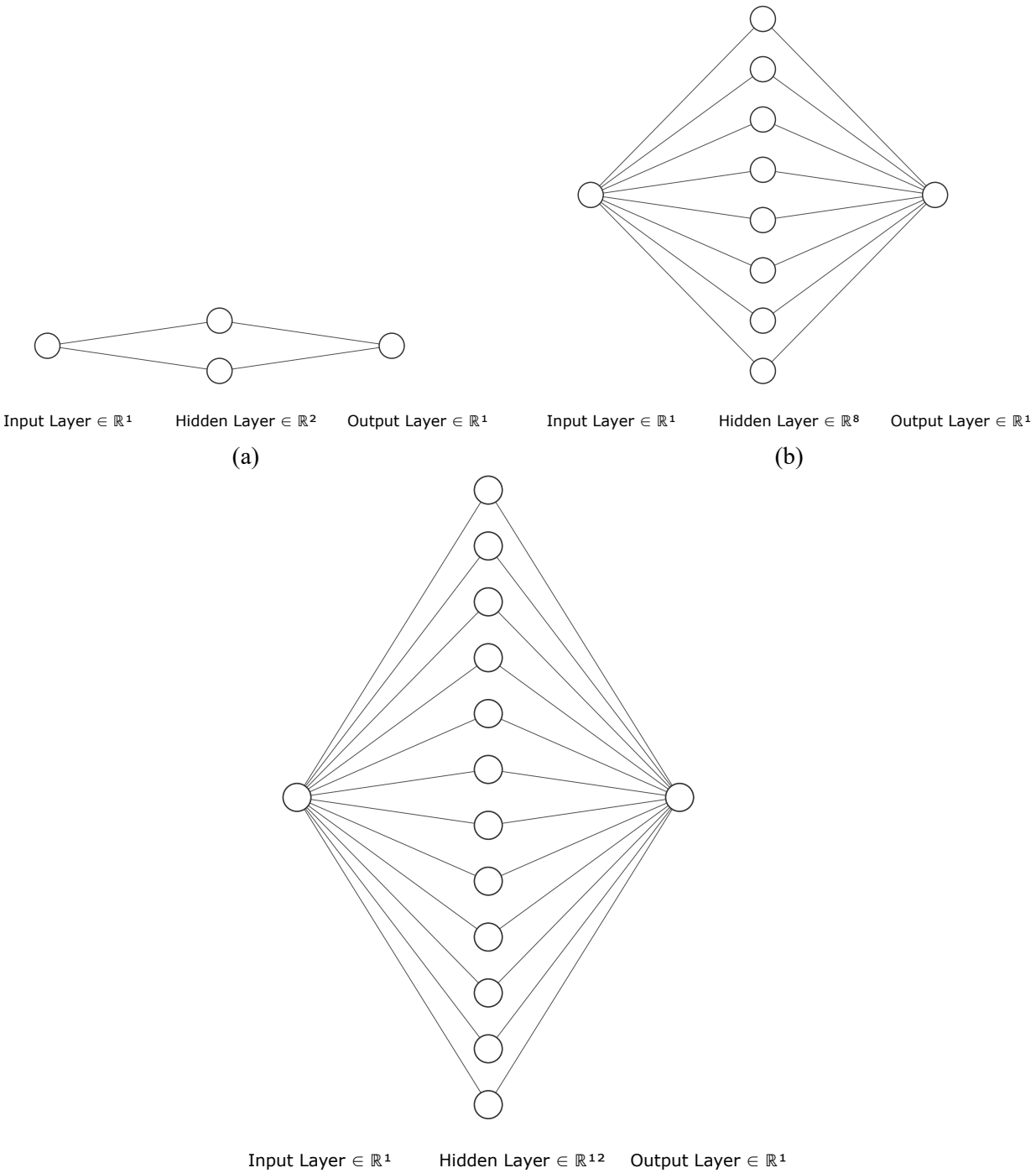

Figure 1: All the neural networks in this problem.

- Is it Linear?
  Yes it is, because it consists of only two linear parts. ReLU is linear within its segments.

In an MLP, the output of each neuron is computed by applying an affine transformation (multiplying the weights and adding the bias), followed by ReLU activation. The key property of ReLU activation is that it is a piecewise linear function. When you consider a single neuron with ReLU activation, it essentially performs two operations:

1. For inputs x where $x > 0$, the output is x.
2. For inputs x where $x \leq 0$, the output is 0

Having a closer look, the first operation $(x > 0)$ is a linear transformation with a slope of 1 (output is $y = x$), and the second operation $(x \leq 0)$ is a constant zero (output is $y = 0$).

By composing several such neurons in an MLP architecture, we effectively create a composition of linear transformations and constant zeros. Since the operations of the individual ReLU neurons are piecewise linear, the combination of these operations is naturally also a piecewise linear function.
The breakpoints in the piecewise linear function occur where the activations of the neurons go from 0 to the actual linear operation -when the input x exceeds 0 -. As you move from one layer to the next in the network, we are effectively combining multiple piecewise linear functions, resulting in a more complex piecewise linear function overall.

The activation function pReLU behaves similarly, but it introduces a learnable parameter a for the negative slope that allows a continuous range of slopes for the linear part when x is negative.

To summarize, an MLP that uses only ReLU (or pReLU) activation functions constructs a continuous piecewise linear function because the operations performed by these activation functions are individually piecewise linear and the composition of these operations across the layers results in a piecewise linear function that approximates complex mappings between inputs and outputs.

We can see also the graphical explaination here:

Problem 8

Adadelta is another variant of the AdaGrad algorithm. The main difference lies in the fact that it decreases the amount by which the learning rate is adaptive to coordinates. Moreover, traditionally it referred to as not having a learning rate since it uses the amount of change itself as calibration for future change. Adadelta is an extension to the Gradient Descent Optimization Algorithm. Although, it is better understood as an extension of the AdaGrad and RMSProp algorithms.

The idea was derived mainly from ADAGRAD in order to improve upon the two main drawbacks of the method.

1. The continual decay of learning rates throughout training
2. The need for a manually selected global learning rate

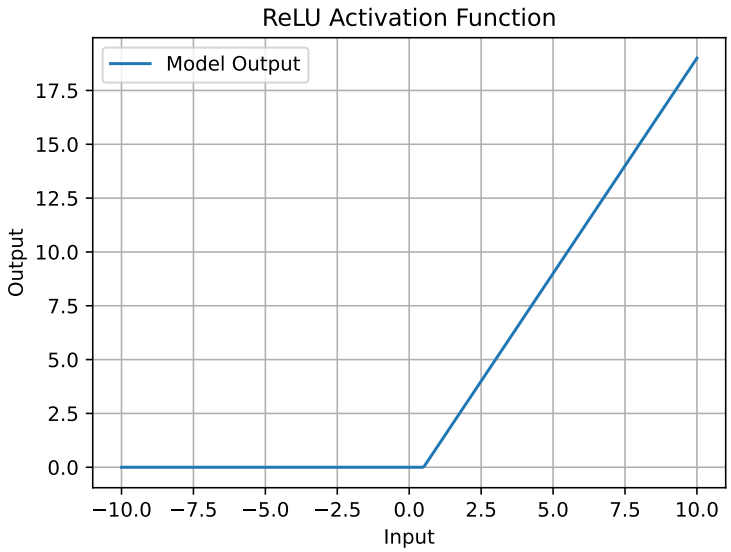

Figure 2: Plot of the MLP using the ReLU activation function

All things considered, in this exercise we are given the following function:

$$F(w) = 0.1w_1^2 + 2w_2^2 \tag{3}$$

(a) Question A

To find the minimum of the Function using the AdaDelta optimizer, instead of the gradient descent, we need to iteratively update the weights w_1 and w_2 based on the optimizer's rules. ADADELTA is an adaptive learning rate optimization algorithm that adjust the learning rate during training.

The Adadelta algorithm has two main parameters: ρ and e. We will set:

- Decay rate $\rho = 0.9$
- Constant $\epsilon = 10^{-6}$

ϵ is a small value which is added to maintain numerical stability.
The ρ variable is a hyperparameter that controls the decay rate of the running averages of the squared gradients, $\mathbf{s}_t$, and squared parameter updates, $\Delta\mathbf{x}_t$.

Based on these values, we will briefly explain the algorithm of AdaDelta. In a nutshell, AdaDelta uses two state variables, $\boldsymbol{s_t}$ to store a leaky average of the second moment of the gradient and $\boldsymbol{\Delta x_t}$ to store a leaky average of the second moment of the change of parameters in the model itself.

Given the parameter ρ, we obtain the following leaky updates:

$$\mathbf{s}_t = \rho\mathbf{s}_{t-1} + (1-\rho)\mathbf{g}_t^2 \tag{4}$$

A crucial point and difference from the RMSProp algorithm is that we perform updates with the rescaled gradient $\boldsymbol{g_t'}$. So, each time the weights are updated as:

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \mathbf{g}_t'. \tag{5}$$

where the rescaled gradient is calculated as:

$$\mathbf{g}'_t = \frac{\sqrt{\Delta \mathbf{x}_{t-1} + \epsilon}}{\sqrt{\mathbf{s}_t + \epsilon}} \, \mathbf{g}_t \tag{6}$$

Here, $\boldsymbol{\Delta x_{t-1}}$ is the leaky average of the squared rescaled gradient $\mathbf{g}'_t$. We initialize $\boldsymbol{\Delta x_0} = \mathbf{0}$ and update each step with g'_t as this equation shows:

$$\Delta \mathbf{x}_t = \rho \Delta \mathbf{x}_{t-1} + (1 - \rho) \mathbf{g}'^2_t, \tag{7}$$

For a better understanding of the topic, we will provide a pseudo-code of the ADADELTA algorithm.

Algorithm 1 Computing ADADELTA update at time t

Require: Decay rate ρ, Constant ϵ, Learning rate α
Require: Initial parameter x_1
  Initialize accumulation variables $S_t\_0 = 0$, $\Delta x_t\_0 = 0$
  for $t = 1$ **to** T **do**
    Compute Gradient: g_t
    Accumulate Gradient: $S_t = \rho S_{t-1} + (1 - \rho) g_t^2$
    Compute Update: $\mathbf{g}'_t = \frac{\sqrt{\Delta \mathbf{x}_{t-1} + \epsilon}}{\sqrt{\mathbf{s}_t + \epsilon}} \, \mathbf{g}_t$
    Accumulate Updates: $\Delta \mathbf{x}_t = \rho \Delta \mathbf{x}_{t-1} + (1 - \rho) \mathbf{g}'^2_t$
    Apply Update: $x_{t+1} = x_t - (a * g'_t)$
  end for

it is very important to emphasize on the tricky part of the provided pseudo-code, because we have slightly changed it. For this exercise we want to include the learning rate in our optimizer. However, as we mentioned before, AdaDelta is an optimizer that doesn't use a learning rate. Taking everything into account, we modified the traditional update

$$x_{t+1} = x_t - g'_t. \tag{8}$$

into

$$x_{t+1} = x_t - (a \cdot g'_t) \tag{9}$$

To conclude, to point out the operation of the AdaDelta algorithm, we will plot the algorithm's trajectory on a contour plot of F(x) for *number of iterations* = 300.

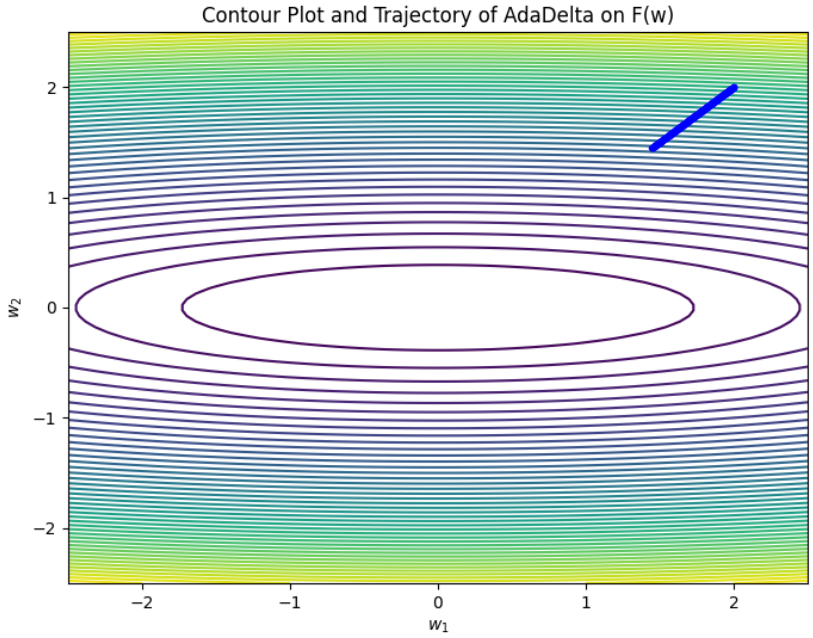

Figure 3: AdaDelta algorithm and trajectory with learning rate $\alpha = 0.4$

(b) Question B

Problem 9

In figure 4, we are given a contour plot and we are asked to draw one gradient step for the three following algorithms:

- Gradient Descent,
- Natural Gradient (*Newton's method*) and
- Adagrad or RMSprop.

(a) Gradient Descent

Gradient Descent calculates the next step as follows. First, the gradient is calculated using a *mathematical expression*. Then, it progresses through the contour plot in the opposite direction of gradient.
This is done repeatedly, until a local minimum is found and converges there.

In this problem, we don't have any mathematical expression in order to calculate the gradient, so we will approximate it visually from the contour plot.
Looking at the starting point in figure 4 (*the rectangle*), the gradient at this point is perpendicular to the contour line and points in the direction of the steepest increase in function value, which is the area where contour lines are closer to each other.

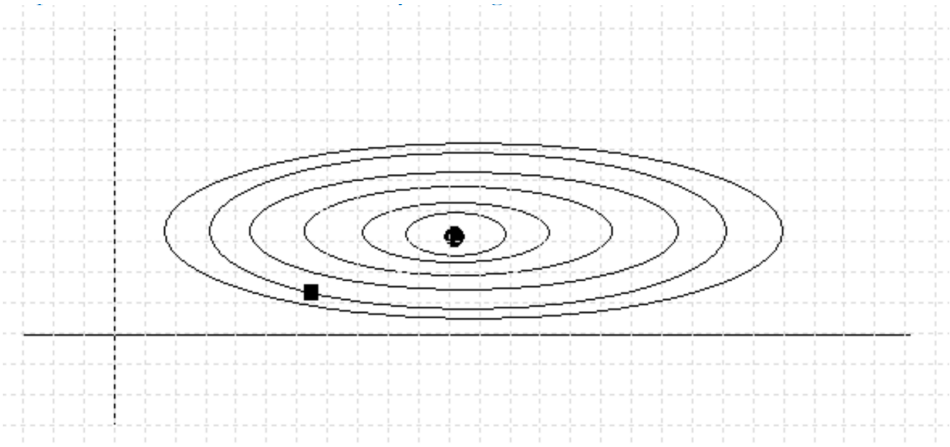

Figure 4: Given contour plot.

Following the algorithm, gradient descent moves in the *opposite direction of the gradient*, which is the direction of the steepest decrease in function value.
Gradient and direction of movement are shown in figure 5.

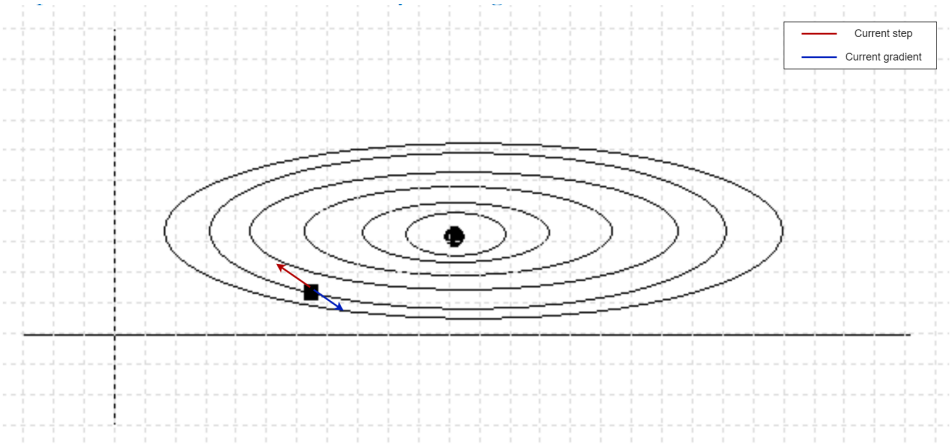

Figure 5: First step of gradient descent. Blue arrow represents the **gradient** on the first step and red arrow represents the **step** of the algorithm.

(b) Newton's Method

Newton's method is an optimization method that takes into account the information about the curvature of the function, which allows it to make a more informed step towards the minimum.
The natural gradient is adjusted by taking into consideration the inverse of the Hessian, a matrix of all second-order partial derivatives of the function. The steps are more direct and perhaps longer, pointing straight toward the minimum because these methods are designed to take the most direct route in the parameter space considering the curvature.

Again, we don't have any expression for the function in order to calculate directly the gradient (*and every other factor*), thus we are going to approximate it visually.
The first step for Newton's method is to calculate the gradient, which can be obtained from **Gradient Descend** calculations.

Next step is to factor in the curvature of the space. Because of the high curvature in this area, step's size is going to be smaller than Gradient Descent's one.

So, the approximated step of Netwon's Method is shown in figure 6.

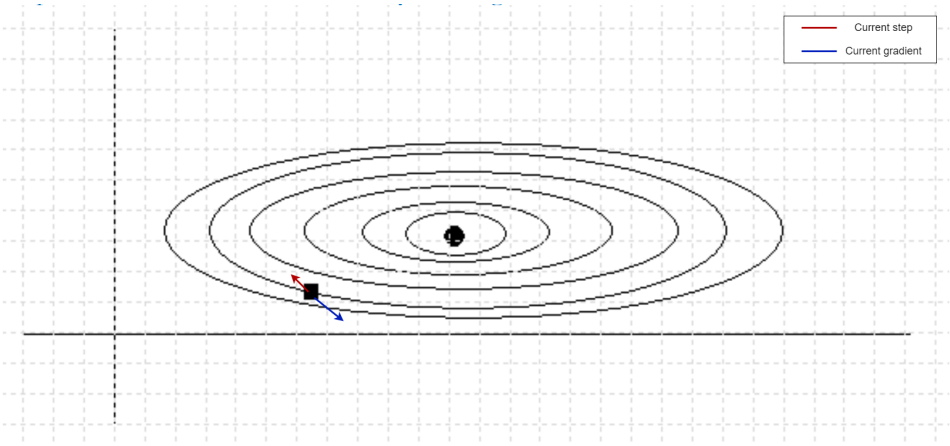

Figure 6: First step of Newton's Method. Blue arrow represents the **gradient** on the first step and red arrow represents the **step** of the algorithm.

(c) Adagrad / RMSprop

Adagrad is an optimization algorithm designed to adapt the learning rate to the parameters, performing smaller updates for parameters associated with frequently occurring features and larger updates for parameters associated with infrequent features.
It offers an adaptive learning rate where each parameter has its own learning rate, improving performance on problems with sparse gradients.

RMSprop is also an adaptive learning rate method that solves a weakness of Adagrad. RMSProp addresses Adagrad's radically diminishing learning rates by using a moving average of squared gradients. This ensures that the learning rate does not decrease too rapidly and is adapted for each weight.

For both Adagrad and RMSProp, the initial direction of movement is opposite to the gradient at the current point. After this point, the two algorithms differentiate. Adagrad decreases the learning rate for each parameter based on the sum of the squares of past gradients for that parameter but this can lead to very small step sizes.

RMSprop modifies Adagrad's approach by using a moving average of the squared gradients instead, which prevents the learning rate from diminishing too rapidly. This means that, compared to Adagrad, RMSProp can maintain a larger step size in areas where Adagrad's steps might become excessively small.
If we assume that they have run for a while to accumulate gradient information, then the step of Adagrad will be a lot smaller than that of RMSprop.
Thus, the approximated steps for Adagrad and RMSprop are shown in figure 7.

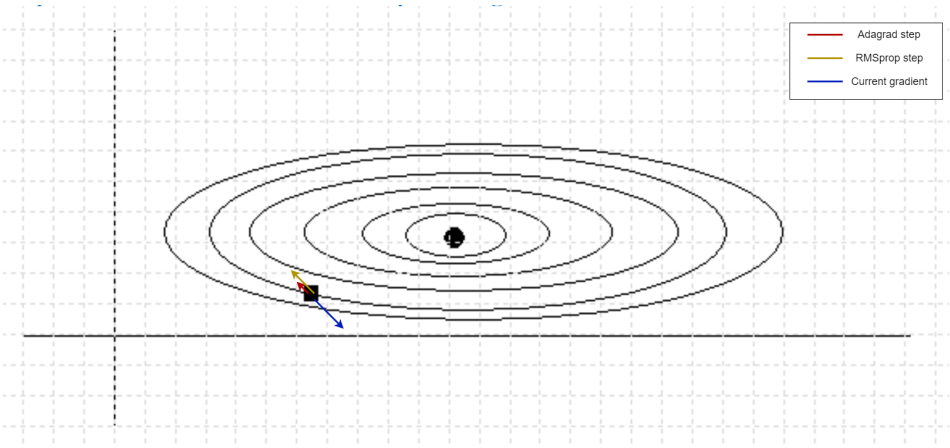

Figure 7: First step of Adagrad and RMSProp (*). Blue arrow represents the **gradient** on the first step and red arrow represents the **step** of the algorithm.
(*): Here, it is assumed that the algorithms have learned about the gradients a bit.
Also, both steps are parallel.

Problem 10

We are given the following optimization method:

$$\begin{aligned} g_{t+1} &\leftarrow \beta \cdot g_t + (1-\beta) \cdot \nabla \hat{L}_t(\theta_t) \\ \theta_{t+1} &\leftarrow \theta_t - \alpha \left[(1-\nu) \cdot \nabla \hat{L}_t(\theta_t) + \nu \cdot g_{t+1} \right] \end{aligned} \tag{10}$$

, where $\alpha,\ \beta, \nu \in \Re$ and α is the learning rate. $\hat{L}_t\left(\theta_t\right)$ represents a loss function which is minimized via θ.

When examining the equation, we start to see a relation with SGD, but with some extra elements on the equation. If we set $(\beta, \nu) = (\mathbf{0}, \mathbf{1})$ in order to eliminate some of the elements, we get the following expression:

$$\left\{ \begin{array}{l} g_{t+1} \leftarrow \nabla \hat{L}_t\left(\theta_t\right) \\ \theta_{t+1} \leftarrow \theta_t - \alpha \cdot g_{t+1} \end{array} \right\} \ = \ \theta_{t+1} \leftarrow \theta_t - \alpha \nabla \hat{L}_t\left(\theta_t\right)$$

This is exactly the update rule of SGD (*Stochastic Gradient Descent*) found in our lectures, parameterized by α. We can get the SGD's update rule with one more pair of (β, ν) values. By setting $(\beta, \nu) = (\mathbf{0}, \mathbf{0})$, we effectively eliminate term g_{t+1}. So, we get:

$$\theta_{t+1} \leftarrow \theta_t - \alpha \cdot \nabla \hat{L}_t(\theta_t)$$

which is again the update rule for SGD.
Another form of the famous SGD is SGD with momentum, and has the following update rule:

$$\begin{aligned} g_{t+1} &\leftarrow \beta \cdot g_t + (1-\beta) \cdot \nabla \hat{L}_t(\theta_t) \\ \theta_{t+1} &\leftarrow \theta_t - \alpha \cdot g_{t+1} \end{aligned}$$

SGD with momentum is an extension of the basic stochastic gradient descent algorithm, designed to accelerate learning, especially in the context of high curvature, small but consistent gradients, or noisy gradients.

In this extension, instead of using only the gradient of the current step to guide the learning process, we also take into account the gradient of the previous steps. This is typically done by keeping a running average of the gradients.

By looking the equation 10, we can clearly obtain the update rule of SGD with momentum easily. We just need to remove the term $(1-\nu)\cdot\nabla\hat{L}_t(\theta_t)$ from θ_{t+1} and we will get the update rule of SGD with momentum. This term is zeroed only when $\nu = 1$ and term β is necessary in the update rule, thus a pair of values $(\beta, \nu) = (\beta, \mathbf{1})$.

Unfortunately, we cannot extract any other familiar optimization method because they introduce sums and other complex operations in the update rule but the given method does not contain any of those. This formula represents a single, unified optimization method that is a variation of SGD with momentum, rather than multiple distinct methods that can be extracted.

Problem 11

Convolutional Neural Networks (CNNs) have revolutionized in the field of image processing and computer vision and are widely utilized.

In this exercise we are considering a $6x6$ image I, where each entry represents the intensity of a pixel.The values are typically normalized , and the CNN would perform operations on this matrix to learn features and perform tasks like classification, detection, or segmentation. We will apply various layers and filters, so that we can extract higher-level features.

$$I = \begin{bmatrix} 20 & 35 & 35 & 35 & 35 & 20 \\ 29 & 46 & 44 & 42 & 42 & 27 \\ 16 & 25 & 21 & 19 & 19 & 12 \\ 66 & 120 & 116 & 154 & 114 & 62 \\ 74 & 216 & 174 & 252 & 172 & 112 \\ 70 & 210 & 170 & 250 & 170 & 110 \end{bmatrix} \tag{11}$$

Given the input matrix we can understand that it represents a grayscale image. In a grayscale image, each pixel is represented by a single intensity value, typically on a scale $[0, 255]$. The 2D input array contains such intensity values for each pixel in the image.

(a) Question A

The output of a convolution layer is a new matrix that's the result of the convolution operation. The convolution operation involves sliding the kernel over the input matrix, with a given stride $(1, 1)$, and for each position, computing the sum of elementwise multiplications.

The use of a stride in a convolutional layer is important, because it determines how much the filter or kernel moves across the input matrix. In our case, a stride of $(1, 1)$ means that the kernel moves one step at a time horizontally and vertically. This will result in an output matrix that is smaller than the input matrix by one less than the kernel size in each dimension. So, in our case the output will be a $4x4$. Also,the output's matrix size is smaller than the original because of the "valid" mode on our code. The "valid" mode means that the

convolution product is only given for points where the kernels overlap completely with the input array. It doesn't add any padding to the input image.

In addition, the kernel we have defined is a $3x3$ matrix with a zero in the center. This means that the convolution operation will sum up the values of the eight surrounding pixels and ignore the center pixel for each position in the input image.

So, in conclusion, with a

- $stride = (1, 1)$ and
- $kernel = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$

The result of the convolution is a $4x4$ matrix

$$result = \begin{bmatrix} 225 & 258 & 250 & 209 \\ 458 & 566 & 552 & 472 \\ 708 & 981 & 887 & 802 \\ 1000 & 1488 & 1320 & 1224 \end{bmatrix} \tag{12}$$

The resulting matrix, represents the features in the input image that the kernel was able to detect. In this case, the kernel seems to act like a filter that emphasizes the surrounding context of each pixel. The exact interpretation would depend on the specific values in the input image and the kernel.

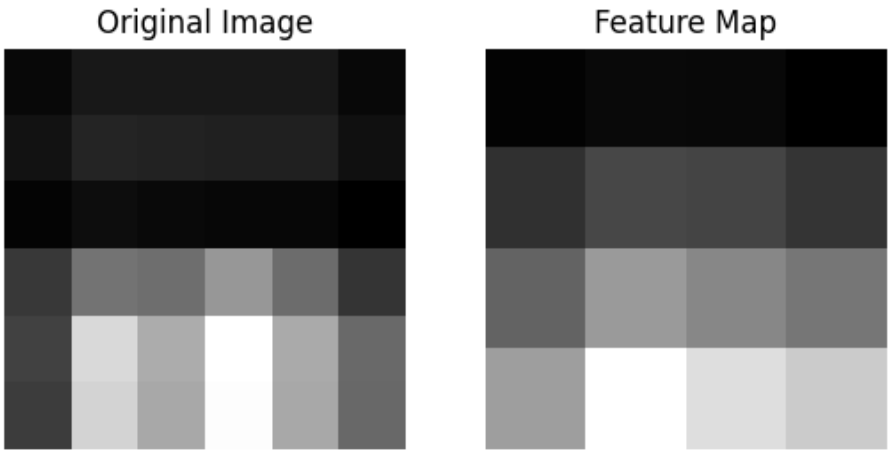

Figure 8: The Original Image and the Image after the convolution

(b) Question B

Now, using the output of the convolution of the input image we are going to apply a max pooling layer with the following properties:

- $stride = (2, 2)$ and
- $window\ \ shape = (2, 2)$

In general, a max pooling layer performs a downsampling operation along the spatial dimensions, width and height, of the input data. The main goal is to reduce the dimensionality of the input, which helps to control overfitting and reduces computational complexity for subsequent layers.
In our exercise, the size of the input matrix is reduced from $4x4 \rightarrow 2x2$.

The max pooling operation works by defining a spatial neighborhood, in our case a $2x2\ \ window$ and taking the maximum element from the rectified feature map within the window. This window is slid over the input data with a certain stride to produce a new matrix where each element is the maximum of a neighborhood from the input. This process effectively reduces the spatial dimensions of the feature map.

The result of the max pooling layer is a $2x2$ matrix of the same image

$$max\_pooling = \begin{bmatrix} 566 & 552 \\ 1488 & 1320 \end{bmatrix} \tag{13}$$

We can conclude that the max pooling operation only reduces the size of the feature map while preserving the most important and prominent features. It gives a more abstract and compressed representation of the input image.

(c) Question C

As we have seen in the previous questions, the use of kernels, also known as filters, is a fundamental tool for image processing. They are essential for the efficient extraction of different features, the reduction of the number of parameters and optimal processing. In this exercise, we will emphasize the importance of kernels for extracting different features from the same input image.

So, for the input image (Matrix 11) we have the following results:

- Filter $F1$

$$F1 = \begin{bmatrix} -10 & -10 & -10 \\ 5 & 5 & 5 \\ -10 & -10 & -10 \end{bmatrix} \tag{14}$$

We can conclude that this filter is a type of edge detection filter, specifically designed to detect edges running horizontally in an image.
In more detail, the negative values on the top and bottom rows will respond strongly to intensity changes in those directions, while the positive values in the middle row will respond to the opposite. Areas with strong horizontal edges will result in high absolute values in the convolved feature map. This means that this filter will highlight areas of the image where there is a strong intensity change from dark to light or light to dark in a horizontal direction, effectively detecting horizontal edges. To be precise, the positive values in the middle row of the kernel will align with the lighter part of a

horizontal edge, while the negative values will align with the darker part.

After applying the kernel $F1$ to our input image I, we obtain the following matrix:

$$I\_F1 = \begin{bmatrix} -925 & -1040 & -1000 & -845 \\ -3900 & -4895 & -4825 & -4160 \\ -3750 & -5120 & -4650 & -4210 \\ -5200 & -6990 & -6750 & -5920 \end{bmatrix} \tag{15}$$

As we can observe, the size of the matrix is reduced, due to the ”valid” mode in the convolution operation on our code, as it only computes the convolution where the kernel fits entirely within the image boundaries.

The values in this feature map represent the strength and location of horizontal edges detected in the input image. High absolute values, whether positive or negative, indicate strong edges, while values close to zero indicate regions with little or no horizontal edge presence.
In this case, the large negative values indicate strong horizontal edges where there is a transition from light to dark pixels. This is consistent with the design of the kernel, which is tailored to detect such features in the image.

To understand the topic, i will provide a graphical explanation.

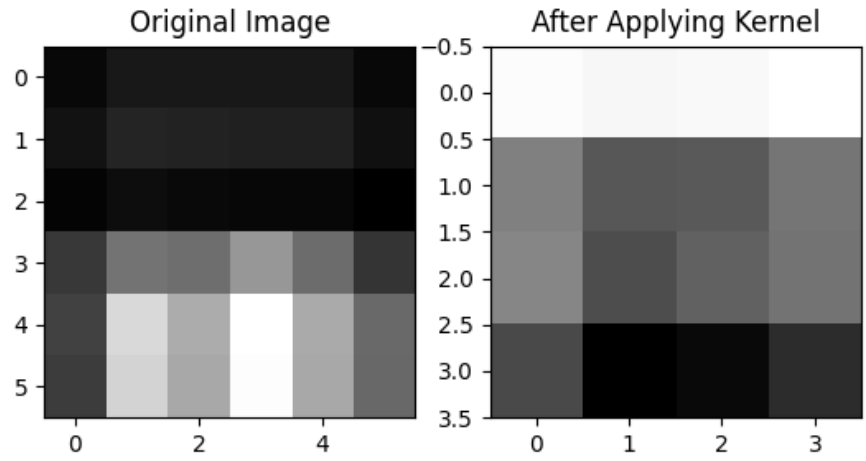

Figure 9: Before and After of the input image

- Filter $F2$

$$F2 = \begin{bmatrix} 2 & 2 & 2 \\ 2 & -12 & 2 \\ 2 & 2 & 2 \end{bmatrix} \tag{16}$$

The filter $F2$ is a $3x3$ matrix with a negative value in the center and positive values surrounding it. This type of kernel is often used for edge detection, most likely to highlight the edges of objects in the image.

This configuration indicates that the kernel is likely designed to detect points in the image where there is a central pixel that is significantly different from its surrounding pixels. Exemplifying, when this kernel is convoluted with an image, it computes a difference between the center pixel and its neighbors. If the image has a region where pixel intensity changes rapidly, the convolution operation will yield a high absolute value. In contrast, in regions of the image where pixel intensity changes slowly, the convolution operation will yield values close to zero.

To have a better understanding, we will apply this kernel to our image I.
After applying the kernel $F2$ to our image, the result is:

$$I\_F2 = \begin{bmatrix} -102 & -12 & -4 & -86 \\ 616 & 880 & 876 & 716 \\ -24 & 570 & -74 & 236 \\ -592 & 888 & -384 & 384 \end{bmatrix} \tag{17}$$

That being said, the positive and negative values in the feature map correspond to areas where this contrast is detected. Particularly, positive values indicate regions where the central pixel is much darker than its surroundings and negative values indicate regions where the central pixel is not significally different from its surroundings as in regions with higher positive values.

We can figure it out by providing also this figure:

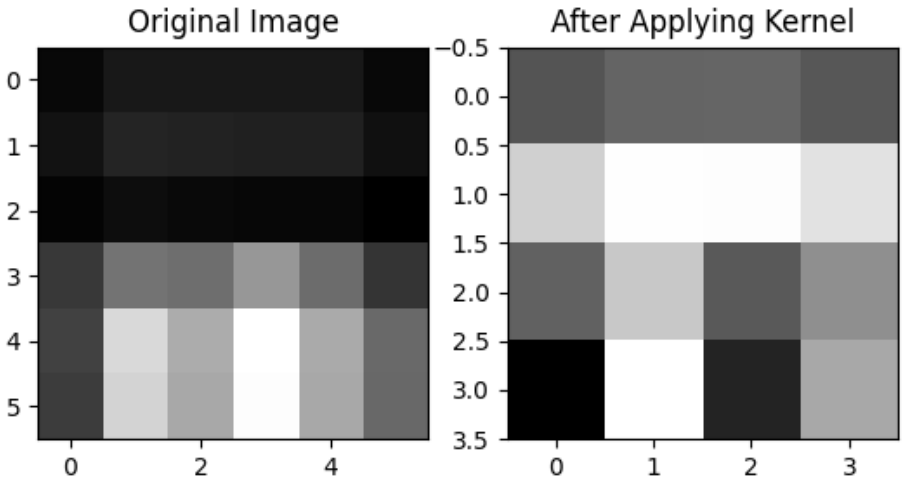

Figure 10: Before and After of the input image

The visualization in the image seems to reflect the application of such a kernel, because areas of the original image that had a central pixel with lower intensity compared to the neighbors would stand

out as brighter spots in the resulting image.

To sum up, this kind of kernel is useful for detecting features such as sharp edges, corners, or small isolated features where there is a notable contrast between a central point and its neighboring pixels. The high values (both positive and negative) in the feature map highlight these contrasting areas in the image.

- Filter $F3$

$$F3 = \begin{bmatrix} -20 & -10 & 0 & 5 & 10 \\ -10 & 0 & 5 & 10 & 5 \\ 0 & 5 & 10 & 5 & 0 \\ 5 & 10 & 5 & 0 & -10 \\ 10 & 5 & 0 & -10 & -20 \end{bmatrix} \tag{18}$$

We can presume that this kernel appears to be a type of edge detection filter, specifically designed to detect diagonal edges in an image.

To elaborate, the negative values in the top-left and bottom-right corners will respond strongly to intensity changes in those directions, while the positive values in the top-right and bottom0left corners will respond to the opposite. This means that this filter will highlight areas of the image where there is a strong intensity change from dark to light or light to dark in diagonal direction, effectively detecting diagonal edges.

By applying the kernel to our image (Matrix 11) we get a $2x2$ feature map:

$$I\_F3 = \begin{bmatrix} -2405 & 1000 \\ -120 & 3915 \end{bmatrix} \tag{19}$$

The negative values on the left and top, transitioning to positive values on the right and bottom, indicate that this kernel might be sensitive to edges that go from dark to light in both vertical and horizontal directions.

Graphically, this can be shown by figure 11.

Problem 12

To compute the number of weights and biases for each convolutional layer, we need to take into consideration the size of kernels and the number of input/output channels for each layer.

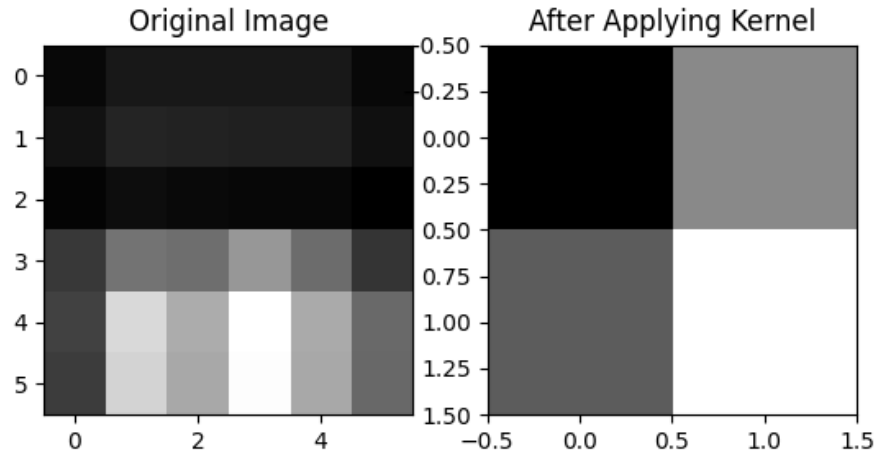

Figure 11: Before and After of the input image

(a) First hidden layer

For the first hidden layer we have:

- **Input Channels:** 3
- **Kernel Size:** 3
- **Output Channels:** 4

Each filter in a convolutional layer has a weight for each entry in the kernel in the kernel for each input channel, and there is one bias per filter.

The number of weights for a single filter in the first layer is the kernel size multiplied by the number of input channels:

$$\textit{Weights per filter} = \textit{Kernel size} \times \textit{input channels} = 3 \times 3 = 9$$

Since there are 4 filters, the total number of weights for the first layer is:

$$\textit{Total weights} = \textit{Weights per filter} \times \textit{Filters} = 9 \times 4 = 36.$$

There's only one bias per filter, so the total number of biases is 4.

(b) Second hidden layer

For the second hidden layer we have:

- **Input Channels:** 4 (*from the previous layer*)
- **Kernel Size:** 5

- **Output Channels:** 10

The number of weights for a single filter in the second layer is the kernel size multiplied by the number of input channels from the first layer:

$$\textit{Weights per filter} = \textit{Kernel size} \times \textit{Input channels from previous layer} = 5 \times 4 = 20$$

Since there are 10 filters, the total number of filters on the second layer is $20 \times 10 = 200$.
As far as the biases are concerned, there's only one bias per filter, so for 10 filters, the total number is 10.

Summarizing, for the two convolutional layers, we need a total of 36 weights and 4 biases for the first layer and 200 weights and 10 biases for the second layer.

Problem 13

Max-pooling is a process used for downsapling the input or reducing its dimensionality. It works by sliding a window across the input and taking the maximum value within that window as an output.

(a) Question A

Max pooling can be accomplished using ReLU operations and in this Question we will show it and we will express max(a, b) by using them.

To begin with, ReLU, as we have mentioned before, is an activation function that:

- For inputs x where $x > 0$, the output is x.
- For inputs x where $x \leq 0$, the output is 0

Mathematically it is defined as: ReLU$(x) =$ max$(x, 0)$

Taking this into account, to express max(a, b) by using only ReLU operations, we can consider the following expression:

$$max(a, b) = a \cdot ReLU(a - b) + b \cdot ReLU(b - a). \tag{20}$$

Therefore, now we need to prove it:

- $\boldsymbol{For\ \ a > b}$**:**
  $a - b > 0$ and $b - a < 0$.
  ReLU$(a - b) = a - b$ and ReLU$(b - a) = 0$

  By replacing these values into Equation 20 we will have:
  $max(a, b) = a \cdot (a - b) + b \cdot 0 = a \cdot (a - b)$.

  In this result, $a > b$, so it is the maximum value between these two and $(a - b) > 0$. Thus, the product is positive and the possible maximum.

  Hence, the expression max(a,b) evaluates to the maximum of a, b.

- ***For $a < b$:***
  $a - b < 0$ and $b - a > 0$.
  ReLU$(a - b) = 0$ and ReLU$(b - a) = b - a$

  By replacing these values into Equation 20 we will have:

  $max(a, b) = a \cdot 0 + b \cdot (b - a) = b \cdot (b - a)$.

  Similarly, $b > a$, so it is the maximum value between these two and $(b - a) > 0$. Thus, the product is positive and the possible maximum.

  Additionally, the expression max(a,b) evaluates to the maximum of a, b too.

Everything considered, we can express max(a, b) as $max(a, b) = a \cdot ReLU(a - b) + b \cdot ReLU(b - a)$.

(b) Question B

In relation to our theory, we know that pooling is a technique used in Convolutional Neural Networks to reduce the spatial dimensions, width and height, of a volume. It is mainly used to reduce computational complexity, control overfitting and manage the number of parameters in an network. Another important purpose of pooling is to increase the receptive field while reducing the spatial extent of the layer by using strides larger than 1.

Max pooling is one of the most common pooling techniques. This operation calculates the maximum valuer in each field of the input matrix within a given window. It also introduces a form of translation invariance as the exact position of the features becomes less important.

However, it has been recently suggested that pooling is not always necessary. One can design a network consisting only of convolutional and ReLU operations and achieve the expansion of the receptive field by using larger steps within the convolutional operations.

That's why in this question, by using the previous expression 20, we will try to implement the max-pooling operation by means of convolutions and ReLU Layers. For this implemention we will need:

- Convolutional Kernel: We can use a $2x2$ convolutional kernel with a stride of 2 and no padding. The purpose of the convolutional kernel is to slide over the input feature map and perform the pooling operation.
- Stride of 2: The stride of 2 means that the convolutional kernel will move by 2 pixels horizontally and vertically at each step. This results in downsampling the feature map by a factor of 2 in both width and height. The pooling operation will select the maximum value within each $2x2$ window.
- No padding: Without padding, the convolutional kernel will only be applied to valid positions of the input. This means that the output feature map will have reduced spatial dimensions compared to the input.
- ReLU activation function: After the convolution operation, we apply the ReLU activation function to the output feature map. The ReLU sets all negative values to zero and keeps the positive values unchanged. This introduces non-linearity and helps the network learn complex patterns and features.

By combining the 2×2 convolution with a stride of 2 and no padding and afterwards applying the ReLU activation function, we achieve the effect of max-pooling. The convolution operaton reduces the spacial dimensions of the feature map, while the ReLU introduces non-linearity.

Overall, this implementation of max-pooling using convolutions and ReLU layers allows for downsampling the feature maps and retaining the maximum values within each pooling window, which is the essence of the Max pooling operation.

(c) Question C

In general, an $n \times n$ convolution needs n^2 channels and layers.
In a standard convolutional layer, each filter processes the entire input and produces a single output channel. But, to mimic the behavior of max-pooling with convolutions, we need to ensure that each element in the $n \times n$ pooling window can be independently compared with the others. This requires a unique filter for each position in the pooling window.

Also, a $n \times n$ window naturally has n^2 elements, with each element being compared with every other element to determine the maximum. Therefore, you need exactly n^2 different channels (*or filters*) where each channel is responsible for one of the n^2 positions in the pooling window.
So, for a 2×2 convolution, $\underline{2^2 = 4}$ layers and channels are needed. For a 3×3 convolution, $\underline{3^2 = 9}$ channels and layers are used.

Problem 14

We are given an abstract of a CNN that classifies images into two classes. Its structure is as follows:

- **Input**: 100×100 grayscale images.
- **Layer 1**: Convolutional layer with 100 5×5 convolutional filters.
- **Layer 2**: Convolutional layer with 100 5×5 convolutional filters.
- **Layer 3**: Max Pooling layer with reduction of 2.
- **Layer 4**: Dense layer with 100 units.
- **Layer 5**: Dense layer with 100 units.
- **Layer 6**: Single output unit.

In order to calculate all the weights in this CNN, we have to consider each layer separately:

Layer 1:

- Input size: 100×100.
- Filter size: 5×5.
- Number of filters: 100.
- Weights: Each filter has 5×5 weights and there's a bias per filter.
  - Weights per filter: $5 \times 5 = 25$.
  - Total weights: $25 \times 100 = 2500$.
  - Total biases: 100 (1 per filter).

So, this layer produces shape $(96, 96, 100)$ and in total we have $2500 + 100 = 2600$ weights.
Output shape is calculated as: input num $-$ kernel size $+1$

Layer 2:

- Input channels: 100 (from layer 1).
- Filter size: 5×5.
- Number of filters: 100.
- Weights: Each filter has 5×5 weights for each input channel.
  - Weights per filter: $5 \times 5 \times 100 = 2500$.
  - Total weights: $2500 \times 100 = 250\,000$.
  - Total biases: 100 (1 per filter).

In total, we have $250\,000 + 100 = 250\,100$ weights and it creates an output shape of $(92, 92, 100)$.

Moving on to **Layer 3**, it's important to note that this layer doesn't have any weights or biases because it's a pooling layer. Pooling layers downsample the output from the previous layer. In this case, Layer's 2 output is reduces to $46 \times 46 \times 100$ from $92 \times 92 \times 100$.

Layer 4 is a dense (*fully connected*) layer with an input that of the max pool layer. Before it connects with the max pool layer, the data must be converted from multi-dimensional array into a one-dimensional array. After this is done, the input data of layer 4 have a size of $46 \times 46 \times 100 = 211\,600$. So, in order to calculate the weights and biases, we only need two information: the number of neurons (100) and the number of neurons of the previous layer (211 600).
The equation for total weights of this layer is:
number of neurons $\times$ *number of neurons of the previous layer* $+$ *number of neurons* $= 100 \times 211600 + 100 = 21160100$.

Layer 5 is also a dense layer and the procedure for calculating the weights is the same as above. We have 100 units in this layer and 100 in the last one, so total weights are: $100 \times 100 + 100 = 10\,100$.

Moving on to the **output layer**, weights are equal to the input units and bias is only 1, so this layer's weights number is $100 + 1 = 101$.

The total number of weights is:

$$\begin{gathered}\text{Total Weights} = \text{Layer 1} + \text{Layer 2} + \text{Layer 3} + \text{Layer 4} + \text{Layer 5} + \text{Layer 6} = \\ = 2600 + 250\,100 + 21\,160\,100 + 10\,100 + 101 = \\ = 21\,423\,001 \text{ weights}\end{gathered}$$

Problem 15