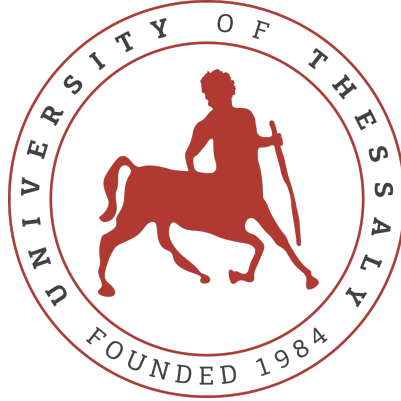


UNIVERSITY OF THESSALY



NEURO-FUZZY COMPUTING

ECE447

---

## 2<sup>nd</sup> Problem Set

---

Alexandra Gianni   Nikos Stylianou

ID: 3382

ID: 2917

February 5, 2024

## Problem 1

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

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

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

$$n^1 = w^1 p + b^1 = (-3)(1) + 2 = -1$$

$$a^1 = \text{Swish}(n^1) = \text{Swish}(-1) = \frac{n^1}{1 + e^{-n^1}} = \frac{-1}{1 + e} = -0.2689$$

Step 2: Calculate second layer's output

$$n^2 = w^2 a^1 + b^2 = (-1)(-0.2689) + (-1) = -0.7311$$

$$a^2 = \text{LReLU}(n^2) = \text{LReLU}(-0.7311) = -0.000731$$

Step 3: Calculate error

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

Step 4: Calculate sensitivity on second layer

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

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

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

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

$$s^1 = 1.0570\text{e} - 7$$

Step 6: Update weights 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 ITERATIONStep 1:

$$n^1 = w^1 p + b^1 = (-3)(1) + 2 = -1$$

$$a^1 = \text{Swish}(n^1) = \text{Swish}(-1) = \frac{n^1}{1 + e^{-n^1}} = \frac{-1}{1 + e} = -0.2689$$

Step 2:

$$n^2 = w^2 a^1 + b^2 = (-1)(-0.2689) + (-1) = -0.7311$$

$$a^2 = \text{LReLU}(n^2) = \text{LReLU}(-0.7311) = -0.000731$$

Step 3:

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

Step 4:

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

Step 5:

$$s^1 = Swish'(n^1) (w^2)^T s^2 = Swish'(-1) (-1) (-1.462e - 6) = 0.0723(-1)(-1.462e - 6)$$

$$s^1 = 1.0570e - 7$$

Step 6:

$$w^2(1) = w^2(0) - LR s^2 (a^1)^T = -1 - 1(-1.462e - 6)(-0.2689) \approx -1$$

$$b^2(1) = b^2(0) - LR s^2 = -1 - 1(-1.462e - 6) \approx -1$$

$$w^1(1) = w^1(0) - LR s^1 (a^0)^T = -3 - 1(1.0570e - 7)(-1) \approx -3$$

$$b^1(1) = b^1(0) - LR s^1 = 2 - 1(1.0570e - 7) \approx 2$$

So, we conclude that there's no change in weights and biases after two iterations of back-propagation algorithm.

## Problem 4

In this problem, we are asked to write a program that implements backpropagation algorithm for an  $1 - S^1 - 1$  network with  $S^1 = \{2, 8, 12\}$ , 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:

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

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

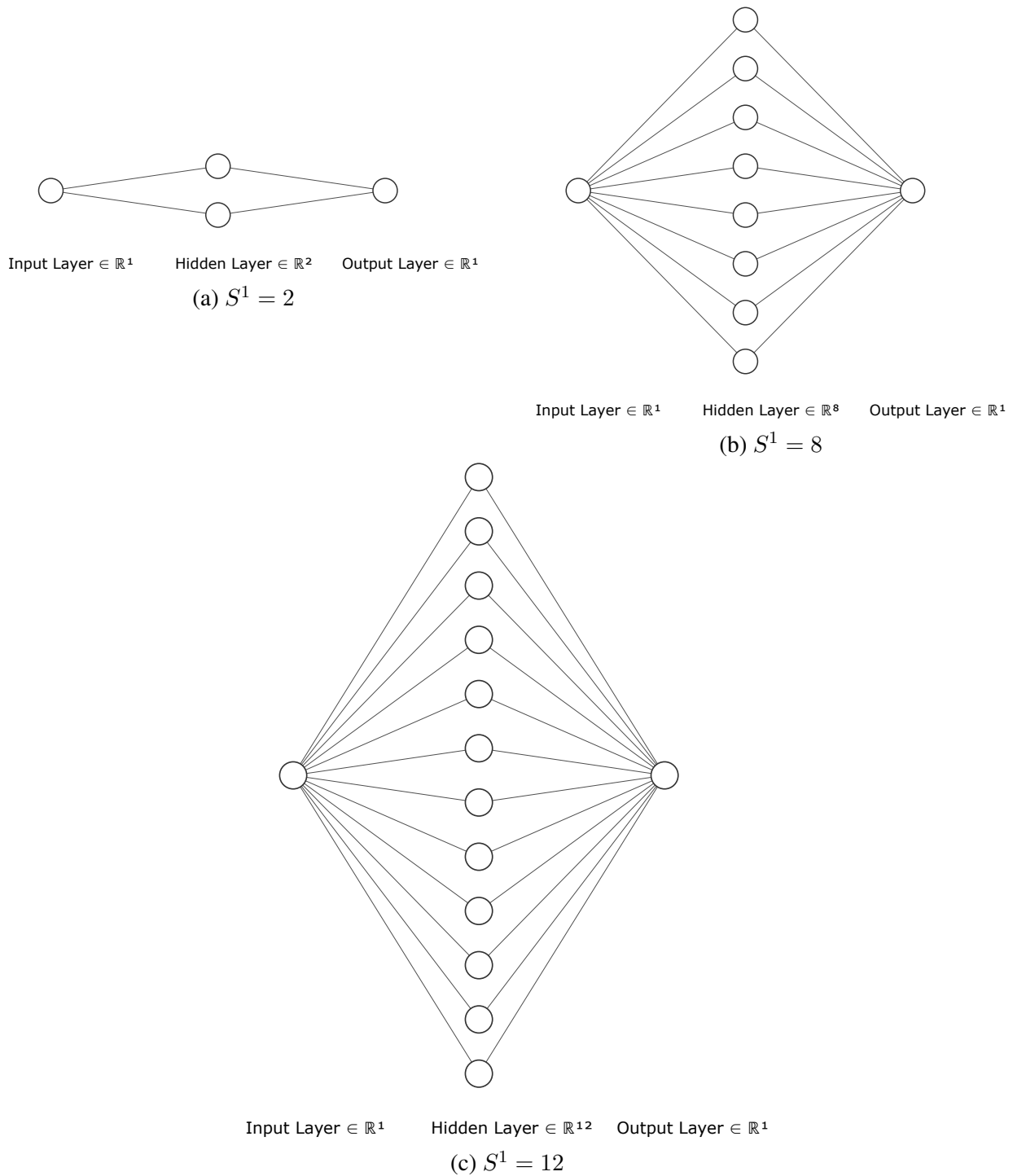


Figure 1: All the neural networks in this problem.

We chose to write this program in MATLAB, despite the majority of these problems being written in Python. This ensures that we are going to use matrix operations for initialization of all weights and biases, as we're mostly familiar with this language. The file "*backpropagation.m*" contains all of the necessary code.

In order to see what difference  $S$  and learning rate does to our network, we defined the following learning rates  $[0.1, 0.01, 0.03, 0.001]$  and generated the network's output and error throughout training. From the very first run ( $S = 2$ ,  $\alpha = 0.001$ ), we can see in figure 2 that the back-propagation algorithm didn't converge at all, keeping the error to a constant value throughout the epochs.

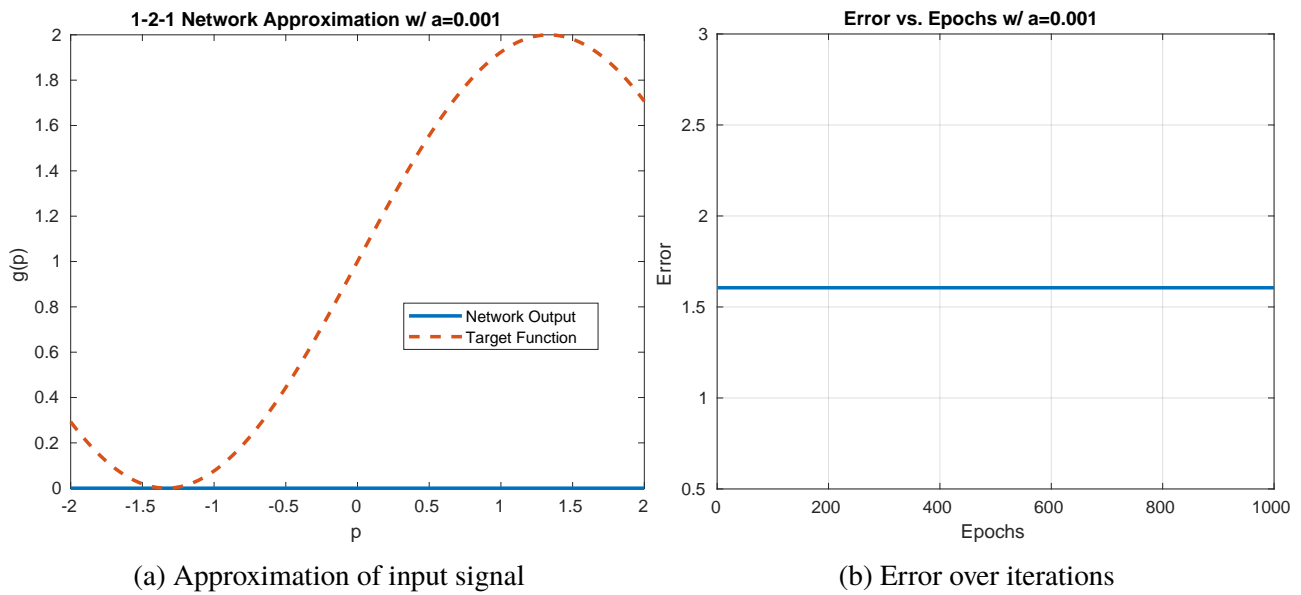


Figure 2: A failed attempt of approximating the input signal.

The reason this approximation failed derives from the uniformly random initial guess for the weights and biases.

After taking a closer look in the code, we can see the point of failure of the algorithm. Below, there's a snippet of the code that calculates the back-propagation.

```
e = g(i) - a2;
s2 = -2*e*relu_derivative(n2);
s1 = logsig_derivative(n1) .* W2' .* s2;
```

We can see that both sensitivities depends on  $s_2$ , a value that itself depends on  $ReLU$ 's derivative, which has the following expression:

$$\frac{d ReLU(x)}{dx} = \begin{cases} 1, & x > 0 \\ 0, & x < 0 \end{cases}$$

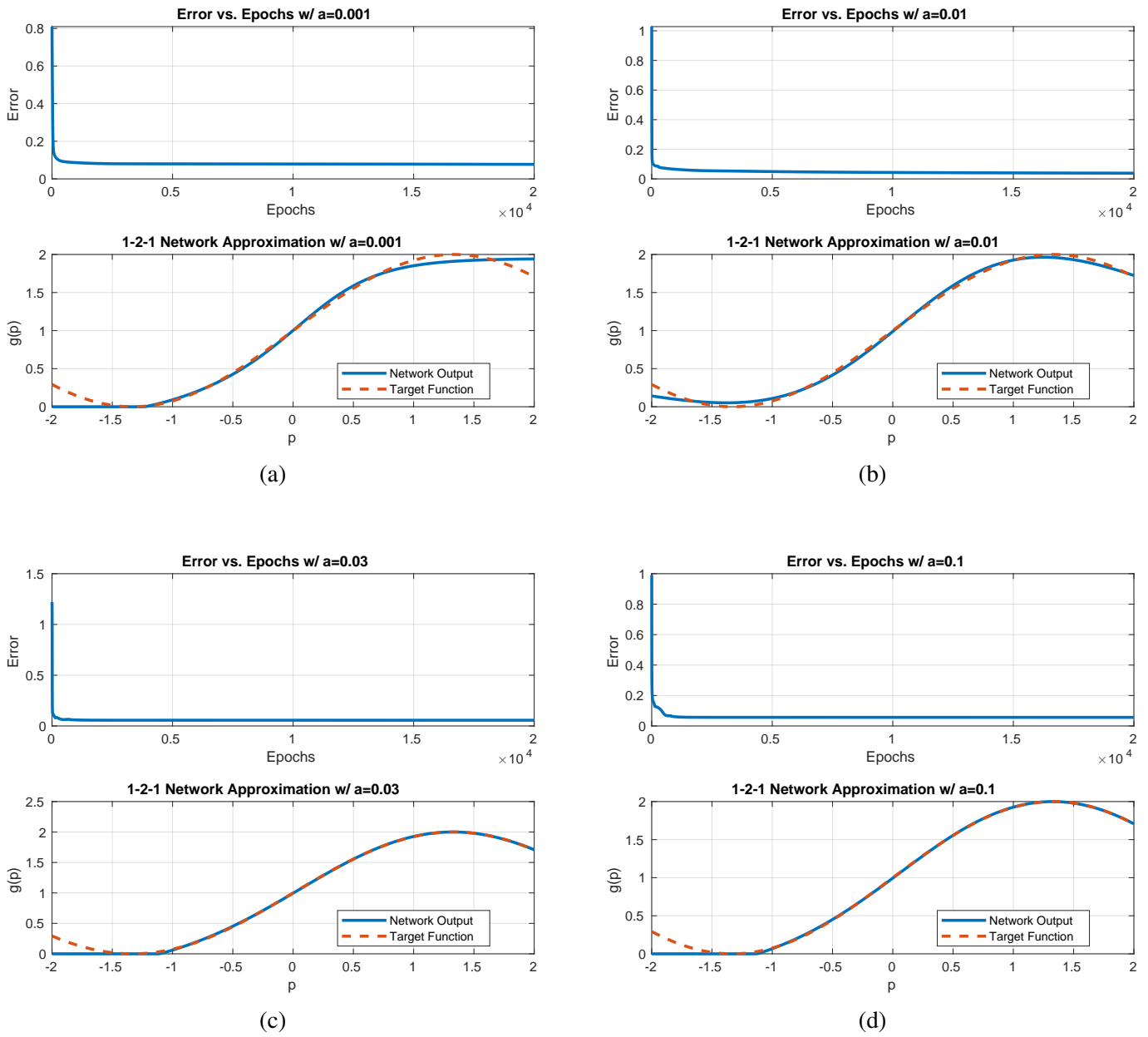
So, if  $n^2$  is a negative number,  $ReLU$ 's derivative will be 0, zeroing out both sensitivities and effectively stalling the back-propagation algorithm. Thus, in order for the algorithm to make progress,  $n^2$  must be a positive number. So, we altered the code and added an if statement that stops the algorithm if variable  $s_2=0$  and epoch=1. In this way, we stop unnecessary calculations and start over, hoping that the random initialization of the weights and biases will not give us another configuration like the one above.

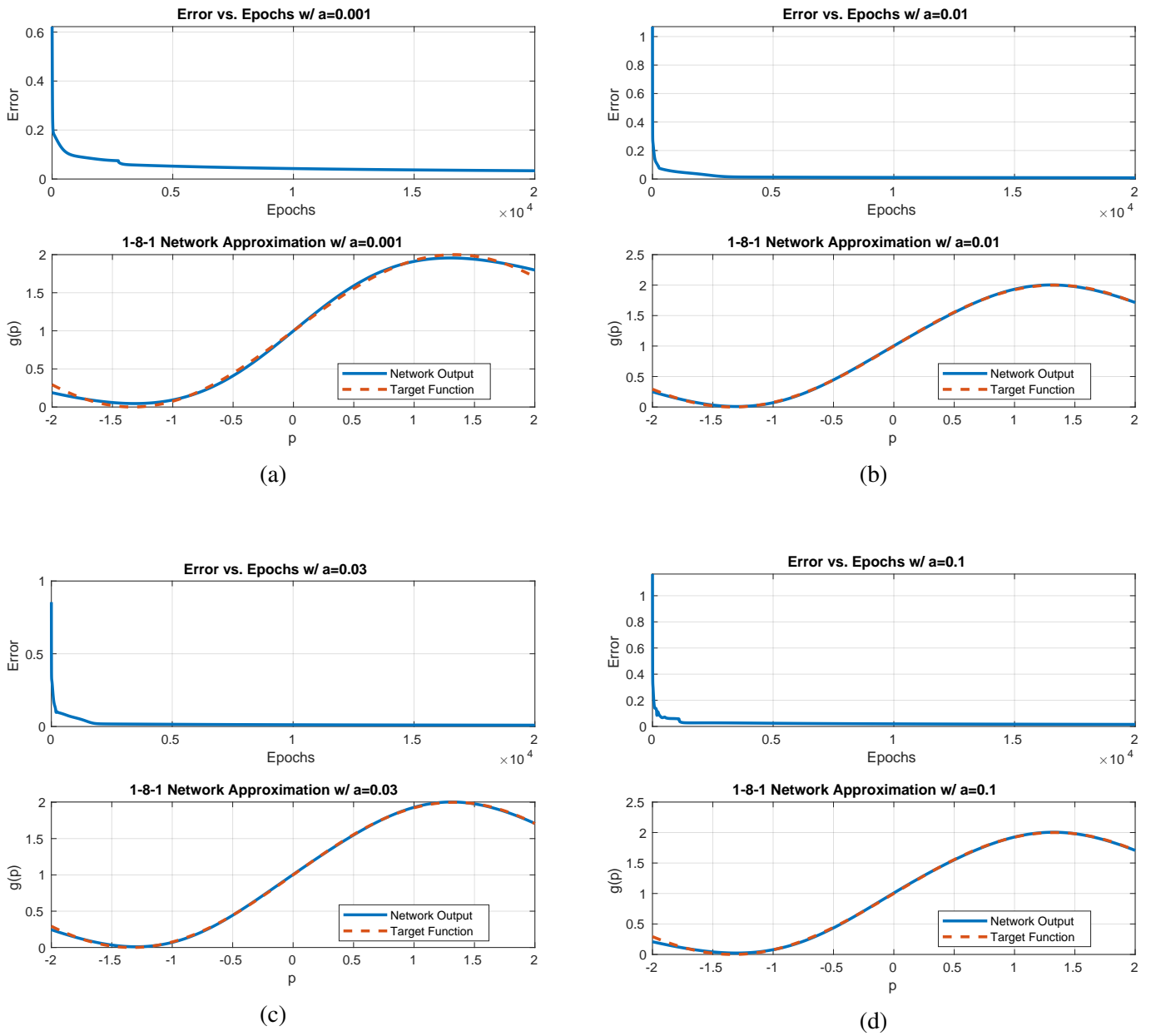
After describing the unexpected issue and its solution, we can focus on the real matter which is the accuracy of the algorithm across different learning rates and number of neurons in the first hidden layer. Figures 3, 4 and 5 contains every figure for every  $S^1$  and learning rate  $\alpha$ .

The error plotted in every figure is the mean square approximation error described below:

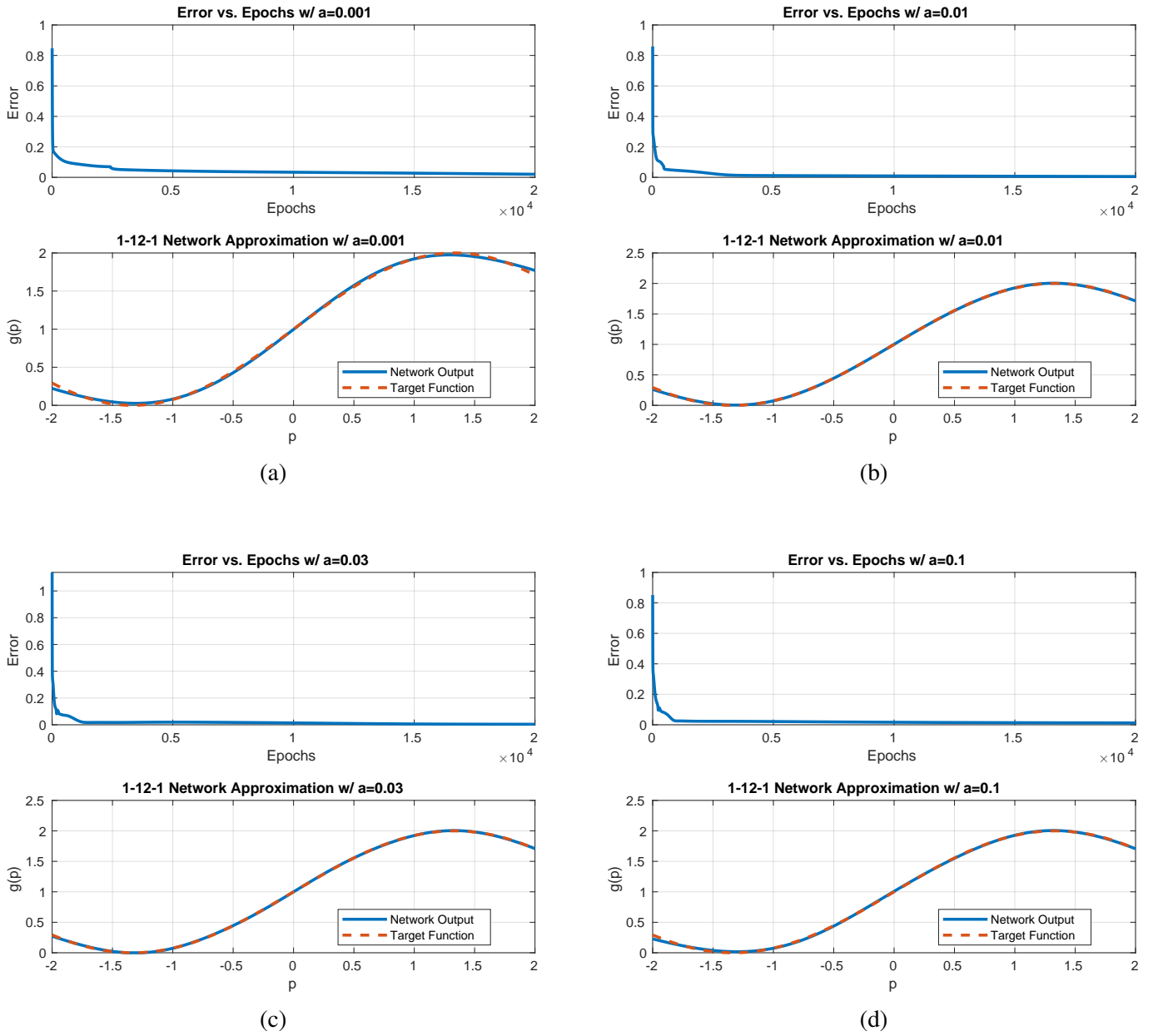
```
err = abs(a2_error' - g).^2;
err_plt(epoch) = sqrt(mean(err));
```

As far as the learning rate  $\alpha$  is concerned, we can observe that decreasing  $\alpha$  results in a smoother but slower convergence. This indicates a more stable learning but requires a lot more iterations to achieve optimal accuracy. On the other hand, higher  $\alpha$  speed up convergence initially, but the end result is not great when comparing to smaller  $\alpha$  values. Looking at the approximated functions, we can see that for  $\alpha = 0.1$  (which is a relatively big value) there's a big approximation error for every  $S^1$ , thus accuracy is compromised.

Figure 3: Error over iterations and approximation,  $S^1 = 2$

Figure 4: Error over iterations and approximation,  $S^1 = 8$



Figure 5: Error over iterations and approximation,  $S^1 = 12$

We tried increasing  $\alpha$  beyond 0.1 but as it turned out, the algorithm failed each time either from the start or later. Regardless, we couldn't get any remarkable results for the report.

Moving on to the capacity of the hidden layer, there's a big change in the accuracy of the neural network as it's increased. Looking at figure 3, we can see that there's a fairly large area at the start and end of  $p$ , where the approximation is way off the input signal. As  $S^1$  increases, the function's approximation on the areas described above is getting smaller as seen in figures 4 and 5. Particularly, in the latter the error drops very low for  $\alpha = 0.03$  and the approximation is almost the same as the original signal. Also, this can be seen in the "Error vs Epochs" plot where the error drops very low.

In conclusion, our experiments reveal a subtle relationship between the learning rate,  $\alpha$ , and the capacity of the neural network, as defined by the number of hidden neurons,  $S^1$ . A lower  $\alpha$  ensures stable and smooth convergence, albeit at the cost of increased iterations for optimal accuracy. Conversely, a higher  $\alpha$  expedites convergence but at the expense of accuracy, particularly evident at  $\alpha = 0.1$  where the approximation error significantly increases across all  $S^1$  values. Furthermore, our attempts to increase  $\alpha$  beyond 0.1 resulted in algorithmic failure, underscoring the delicate balance required in tuning  $\alpha$ . On the capacity front, enhancing  $S^1$  markedly improves the neural network's accuracy, particularly for complex function approximations as the neural network can process much more information. This is demonstrated through improved approximation closeness to the original signal with increased  $S^1$ , highlighting the critical role of neural network capacity in achieving high-fidelity approximations.

## Problem 5

This problem relies on the parametric neural network from the previous problem, only that this time we have  $S^1 = 12$  and learning rate  $\alpha = 0.1$ . Despite the use of MATLAB programming language in Problem 4, this time we opted to use python-TensorFlow. It still uses all of the matrix operations that we used earlier but they are abstracted from the end user who just calls an API. The engine does the rest.

Looking at figure 6, we can clearly see that there's no difference between it and figure 5. This means that we're on the same level as before (*same constraints, weight and bias initialization etc.*) and we can continue on this problem. The code for this is located at Problem 5/keras\_backprop.py.

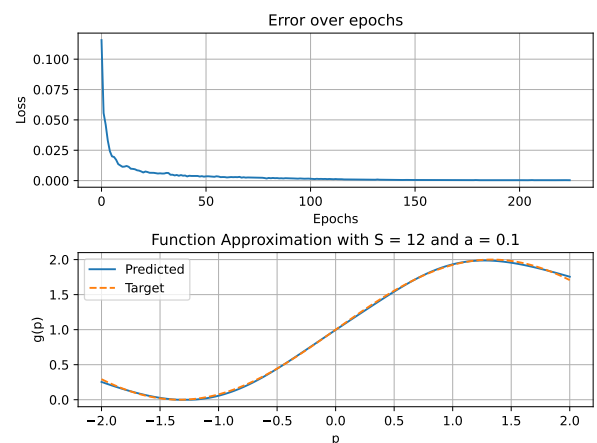


Figure 6: Approximated function and error over iterations

In order for us to not have unnecessary computations, we implemented Early Stopping. This is a method that stops a neural network's train if there aren't significant changes in approximation error.

Backdrop implementation was fairly easy. We just added the Dropout layer between the first and last layer with the appropriate probability  $\theta$  every time. TensorFlow assures us that the dropout layer we just added will not be used during testing.

After the model's been created, we run the program and produce the images in figure complex 7.

We can see from the figures that, the approximation is not working at all, despite the loss function being so low. The approximated function “crosses” the input one 2 or 3 times at most, making the approximation a low fidelity one. Also, the loss function over iterations jumping all over the place as epochs progresses.

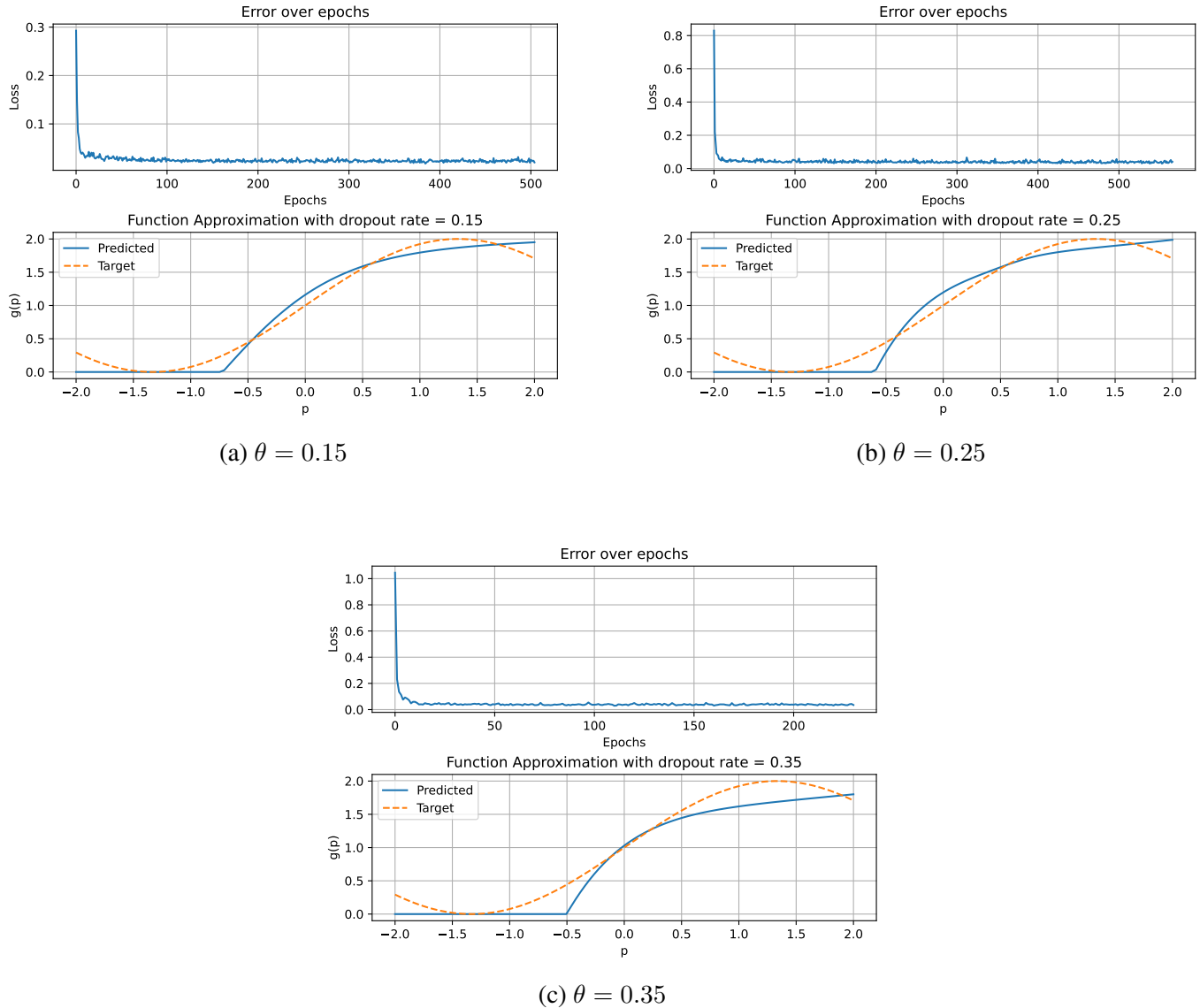


Figure 7: Loss and approximation of input for multiple  $\theta$  values.

Specifically, for  $\theta = \{0.15, 0.25\}$ , the loss decreases sharply at the beginning and then plateaus, suggesting that the model is quickly learning the function but then reaches a point where additional learning is marginal. For  $\theta = 0.35$ , the loss decreases more slowly, indicating that a higher dropout rate might be slowing down the learning process or improving generalization by preventing overfitting.

As far as the approximation of the input signal is concerned, for  $\theta = 0.15$ , the predicted function seems to fit the target quite well, except for slight deviations at the boundaries ( $p = -2$  and  $p = 2$ ). For  $\theta = 0.25$ , the fit is less accurate, with noticeable divergence from the target, especially for values of  $p$  greater than 1. Lastly, for  $\theta = 0.35$ , the predicted function deviates significantly from the target across almost the entire range of  $p$ , indicating that the model may be underfitting, possibly due to too much regularization from the dropout.

## 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:

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

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  $x$

- 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.

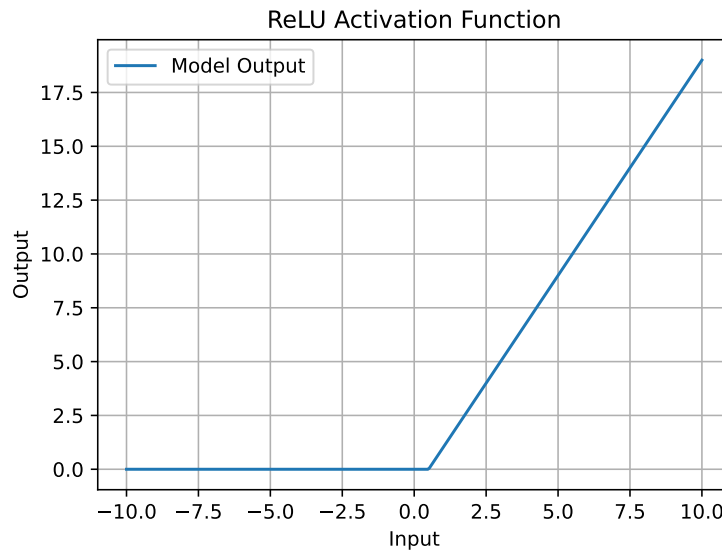


Figure 8: Plot of the MLP using the ReLU activation function

We can see also the graphical explanation 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

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

$$F(w) = 0.1w_1^2 + 2w_2^2 \quad (3)$$

### (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. Adadelta computes a moving average of the parameter updates and the learning rate is adjusted based on this moving average.

The Adadelta algorithm has two main parameters:  $\rho$  and  $\epsilon$ . We will set:

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

$\epsilon$  is a small value which is added to maintain numerical stability.

The  $\rho$  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,  $\mathbf{s}_t$  to store a leaky average of the second moment of the gradient and  $\Delta \mathbf{x}_t$  to store a leaky average of the second moment of the change of parameters in the model itself.

Given the parameter  $\rho$ , we obtain the following leaky updates:

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

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

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

where the rescaled gradient is calculated as:

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

Here,  $\Delta \mathbf{x}_{t-1}$  is the leaky average of the squared rescaled gradient  $\mathbf{g}'_t$ . We initialize  $\Delta \mathbf{x}_0 = \mathbf{0}$  and update each step with  $\mathbf{g}'_t$  as this equation shows:

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

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  $\rho$ , Constant  $\epsilon$ , Learning rate  $\alpha$

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

    Apply Update:  $x_{t+1} = x_t - (\alpha * g'_t)$

**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

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

into

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

To conclude, to point out the operation of the AdaDelta algorithm, we will plot the algorithm's trajectory on a contour plot of  $F(x)$ .

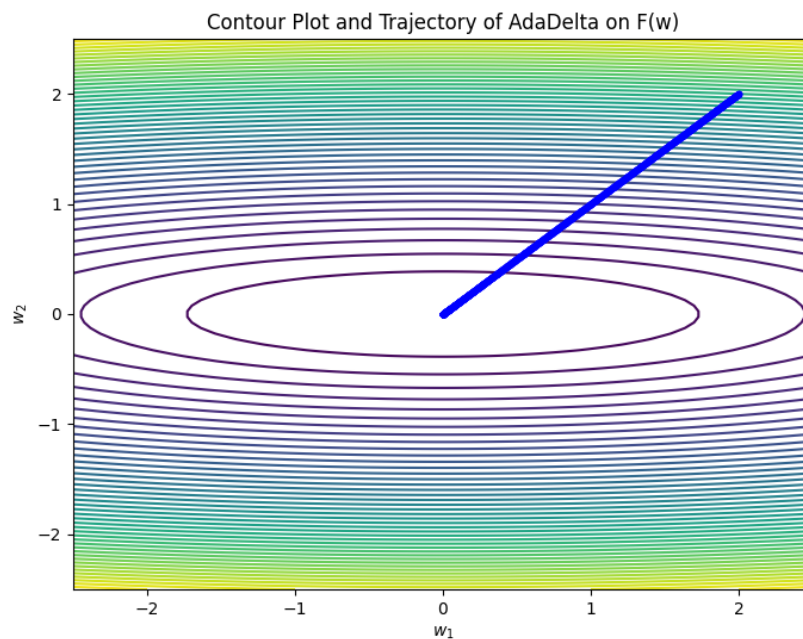


Figure 9: AdaDelta algorithm and trajectory with learning rate  $\alpha = 0.4$

## (b) Question B

As previously mentioned, in the AdaDelta optimization algorithm, the learning rate is adaptively changed based on the running average of the recent gradients and the recent parameter updates. This is different from many other optimization algorithms where the learning rate is a fixed hyperparameter.

But, in this example we will show once again how by implementing some tricks we can still define it and observe how it can affect the algorithm's trajectory.

On the previous question, we plotted the algorithm's trajectory for learning rate  $\alpha = 0.4$ . When we change it to  $\alpha = 3$  we can remark some crucial points.

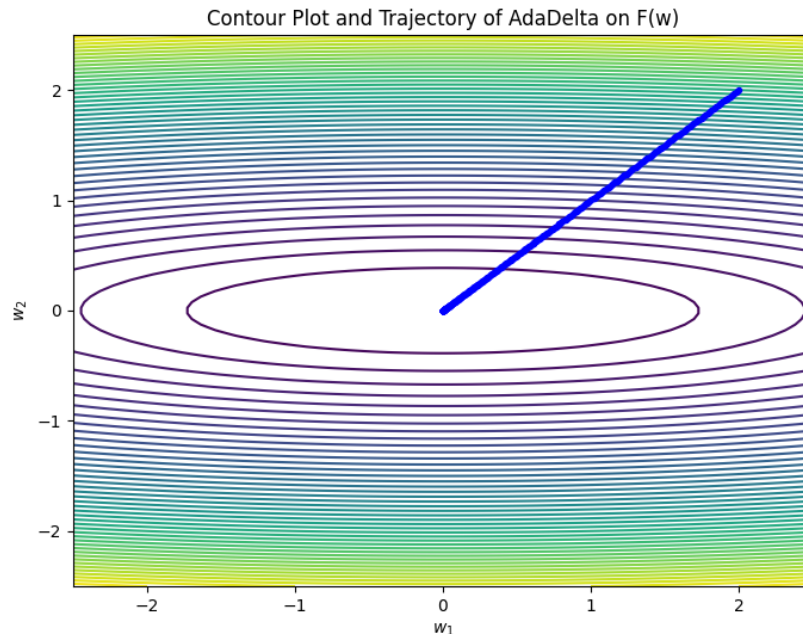


Figure 10: AdaDelta algorithm and trajectory with learning rate  $\alpha = 3$

In drawing things to a close, based on Figure 9 and Figure 10 we can evaluate the impact of the learning rate and to take a closer look on the algorithm.

In both plots, the contours represent levels of the function  $F(w)$  with each line indicating the points where the function has the same value. The closer the lines, the steeper the gradient at that point. The central region where the lines are more widely spaced indicates the minimum of the function.

The observation that we can highlight is the speed of convergence. A larger learning rate generally means that the optimizer makes larger steps in the parameter space. If the learning rate is set appropriately, this can lead to faster convergence to the minimum of the function. Though, there is a big risk of overshooting the minimum, which means that the optimizer could oscillate around the minimum, or in the worst case, to diverge.

Dissimilar to it, in Figure 10, it appears that the AdaDelta optimizer with a higher learning rate has discovered a trajectory that approaches the function's central minimum more closely. This suggests that effective parameter space exploration has been made possible by the higher learning rate without leading to instability in the optimization procedure.

With the help of our code we discovered that for the same *number of iterations* = 2000, the optimizer with learning rate  $lr = 0.4$  converges after **1236 iterations** at  $[1.219\ 18e - 5, 4.015\ 99e - 31]$  and with learning rate  $lr = 3$  after **315 iterations** at  $[2.375\ 91e - 6, 6.354\ 76e - 7]$ .

### (c) Question C

As a last note, we would like to test AdaDelata with the same objective function rotated by  $45^\circ$ . Hence, now we have the following objective function:

$$F(w) = 0.1(w_1 + w_2)^2 + 2(w_1 - w_2)^2 \quad (10)$$



We may export our final observations by comparing the code for the original function and the rotated one.

Similar to other gradient-based optimization techniques, the AdaDelta approach is invariant to orthogonal transformations like rotations. This is because orthogonal transformations maintain the angles and distances between vectors. Rotating an objective function by 45 degrees, or any other angle, in its variable space doesn't change the intrinsic properties of the function such as its minima, maxima, or saddle points. What changes is the coordinate system in which you are expressing the function.

To observe the behavior of the AdaDelta optimization algorithm on the rotated objective function, we can compare its convergence with the original objective function and see if they behave differently in terms of convergence. We notice that both objective functions ultimately converge to a similar minimum value with similar number of iterations, which suggests that the rotated function behaves similarly to the original function in terms of convergence.

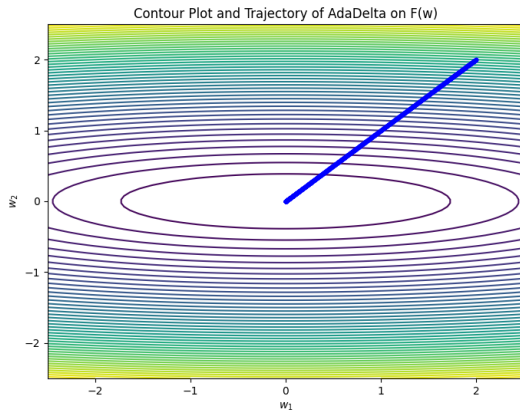
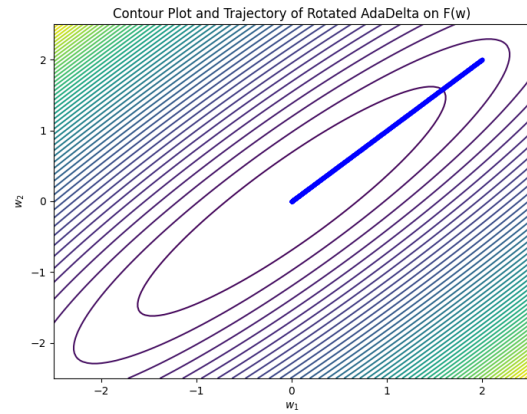
In addition to convergence behavior, we can notice other similarities and differences between the original and rotated function.

By geometry, the original function is an elliptical bowl with its major axis aligned with the coordinate axes, while the rotated one has a more circular or isotropic shape, as it combines  $(w_1 + w_2)$  and  $(w_1 - w_2)$  terms, resulting into a more balanced sensitivity to changes of  $w_1, w_2$ . The shape of the contour of the function will appear rotated by 45°, but the relative distances between the contours will remain the same, which means the steepness or flatness of the function does not change.

By gradient behavior, the gradients of the original function have larger components in the  $w_2$  direction due to the higher coefficient for  $w_2^2$ . As a result, during optimization, the algorithm will place a greater emphasis on updating  $w_2$ . On the other hand, the rotated function exhibits more balanced gradients in both  $w_1$  and  $w_2$  directions due to the equal contributions of the  $(w_1 + w_2)$  and  $(w_1 - w_2)$  terms. The convergence route and the relative significance of the  $w_1$  and  $w_2$  updates may be impacted by this balanced gradient behavior.

On a final note, by curvature, the Hessian matrix of the original function will have larger eigenvalues along the  $w_2$  direction, indicating stronger curvature in that direction. In the rotated function, the Hessian matrix will have more balanced eigenvalues due to the isotropic shape.

Summing up, both functions show variances in their geometry, gradient behaviour, sensitivity to initialization, and curvature, but ultimately converge to a similar minimum value. A more balanced and isotropic shape is obtained by rotating the objective function by 45 degrees, and this can have an impact on the optimization behavior and relative relevance of various factors.

(a) Initial Function with  $\alpha = 0.4$ (b) Rotated Function with  $\alpha = 0.4$ 

The initial function, for learning rate  $\alpha = 0.4$ , converges after 1236 iterations at  $[1.219\,18e - 5, 4.015\,99e - 31]$   
 The Rotated function, for the same learning rate, converges after 1213 iterations at  $[3.916\,12e - 6, 3.916\,12e - 6]$

## Problem 9

In figure 12, 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.

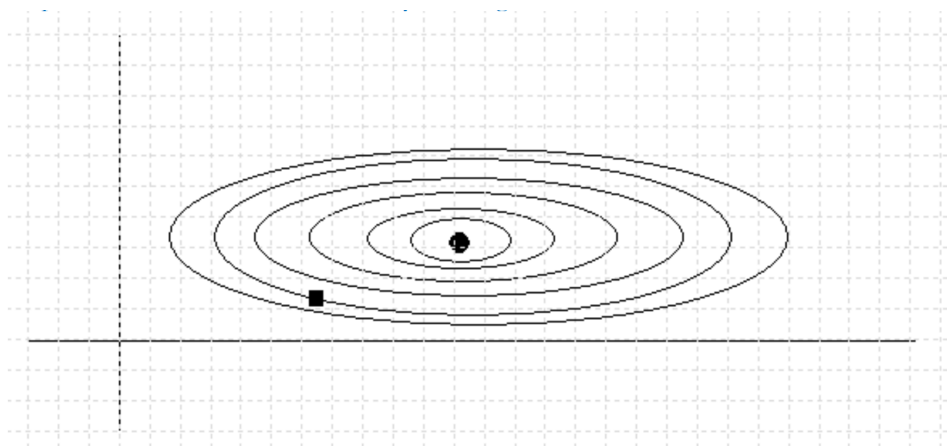


Figure 12: Given contour plot.

### (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 12 (*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.

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 13.

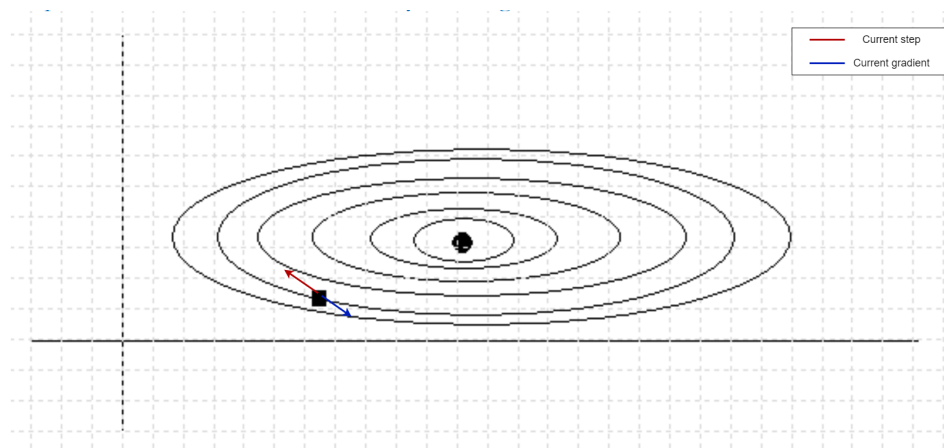


Figure 13: First step of gradient descent. Blue arrow represents the **gradient** on the first step and red arrow represents the **step** of the algorithm.

### (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 Descent** 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 Newton's Method is shown in figure 14.

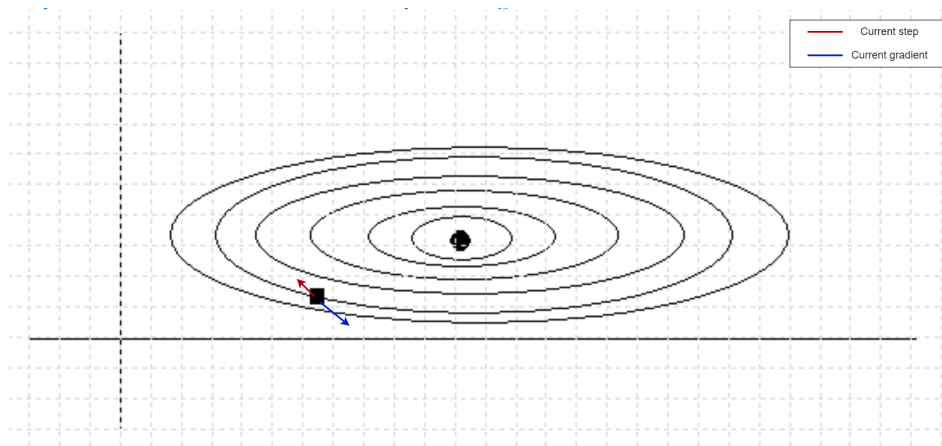


Figure 14: First step of Newton's Method. Blue arrow represents the **gradient** on the first step and red arrow represents the **step** of the algorithm.

### (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 15.

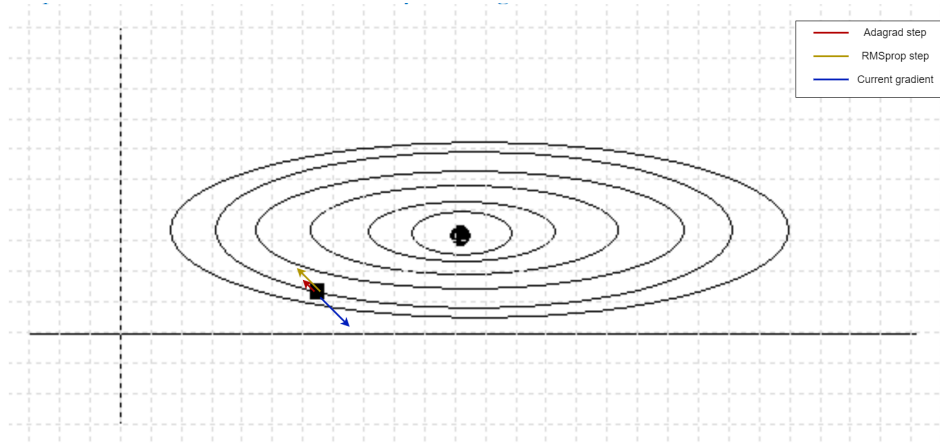


Figure 15: First step of Adagrad and RMSProp (\*). Blue arrow represents the **gradient** on the first step and red arrow represents the **step** of the algorithm.

(\*): 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:

$$\begin{aligned} g_{t+1} &\leftarrow \beta \cdot g_t + (1 - \beta) \cdot \nabla \hat{L}_t(\theta_t) \\ \theta_{t+1} &\leftarrow \theta_t - \alpha \left[ (1 - \nu) \cdot \nabla \hat{L}_t(\theta_t) + \nu \cdot g_{t+1} \right] \end{aligned} \quad (11)$$

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

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:

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

This is exactly the update rule of SGD (*Stochastic Gradient Descent*) found in our lectures, parameterized by  $\alpha$ . We can get the SGD's update rule with one more pair of  $(\beta, \nu)$  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 11, 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  $\theta_{t+1}$  and we will get the update rule of SGD with momentum. This term is zeroed only when  $\nu = 1$  and term  $\beta$  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  $6 \times 6$  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.

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

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  $4 \times 4$ . 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  $3 \times 3$  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  $4 \times 4$  matrix

$$result = \begin{bmatrix} 225 & 258 & 250 & 209 \\ 458 & 566 & 552 & 472 \\ 708 & 981 & 887 & 802 \\ 1000 & 1488 & 1320 & 1224 \end{bmatrix} \quad (13)$$

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.

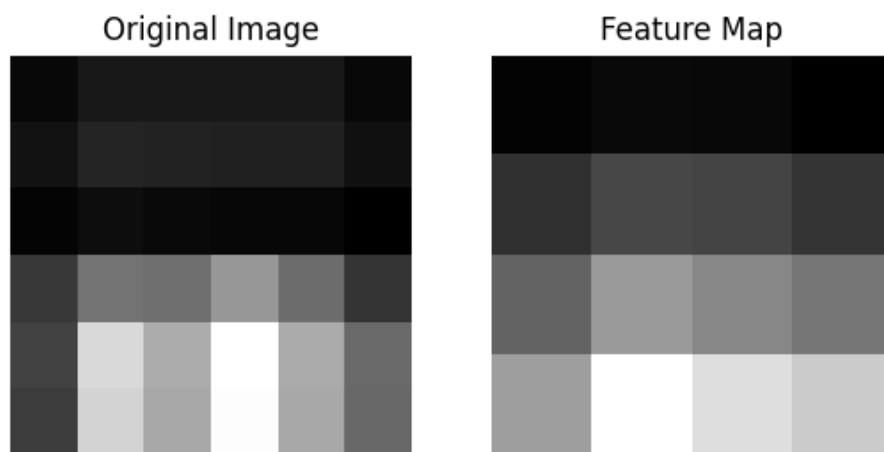


Figure 16: The Original Image and the Image after the convolution

## (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  $4 \times 4 \rightarrow 2 \times 2$ .

The max pooling operation works by defining a spatial neighborhood, in our case a  $2 \times 2$  *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  $2 \times 2$  matrix of the same image

$$max\_pooling = \begin{bmatrix} 566 & 552 \\ 1488 & 1320 \end{bmatrix} \quad (14)$$

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 12) we have the following results:

- Filter  $F1$

$$F1 = \begin{bmatrix} -10 & -10 & -10 \\ 5 & 5 & 5 \\ -10 & -10 & -10 \end{bmatrix} \quad (15)$$

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:

$$I_{F1} = \begin{bmatrix} -925 & -1040 & -1000 & -845 \\ -3900 & -4895 & -4825 & -4160 \\ -3750 & -5120 & -4650 & -4210 \\ -5200 & -6990 & -6750 & -5920 \end{bmatrix} \quad (16)$$

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.

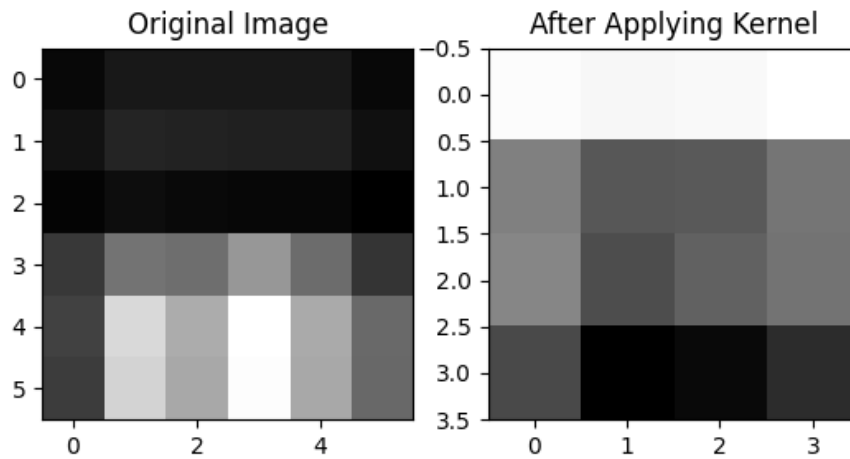


Figure 17: Before and After of the input image

- Filter  $F2$

$$F2 = \begin{bmatrix} 2 & 2 & 2 \\ 2 & -12 & 2 \\ 2 & 2 & 2 \end{bmatrix} \quad (17)$$

The filter  $F2$  is a  $3 \times 3$  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:

$$I_{F2} = \begin{bmatrix} -102 & -12 & -4 & -86 \\ 616 & 880 & 876 & 716 \\ -24 & 570 & -74 & 236 \\ -592 & 888 & -384 & 384 \end{bmatrix} \quad (18)$$

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 significantly different from its surroundings as in regions with higher positive values.

We can figure it out by providing also this figure:

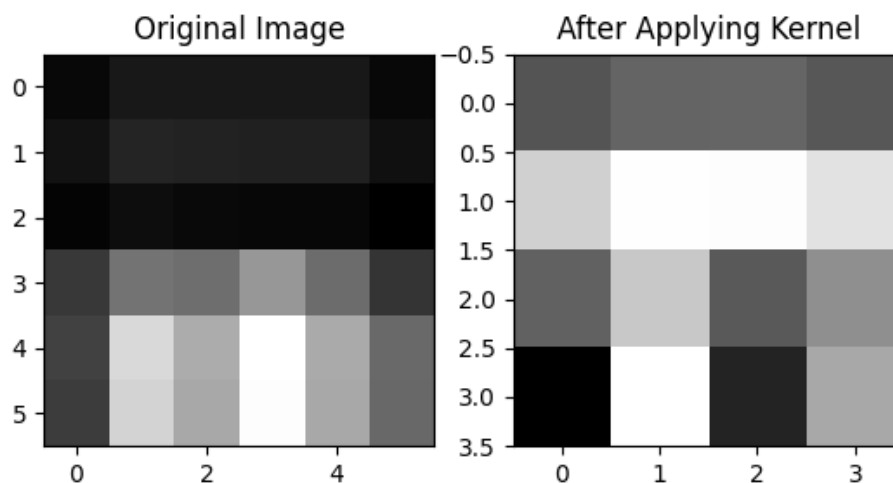


Figure 18: Before and After of the input image

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  $F3$

$$F3 = \begin{bmatrix} -20 & -10 & 0 & 5 & 10 \\ -10 & 0 & 5 & 10 & 5 \\ 0 & 5 & 10 & 5 & 0 \\ 5 & 10 & 5 & 0 & -10 \\ 10 & 5 & 0 & -10 & -20 \end{bmatrix} \quad (19)$$

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 bottom-left 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 12) we get a  $2 \times 2$  feature map:

$$I_{F3} = \begin{bmatrix} -2405 & 1000 \\ -120 & 3915 \end{bmatrix} \quad (20)$$

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 19.

## 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.

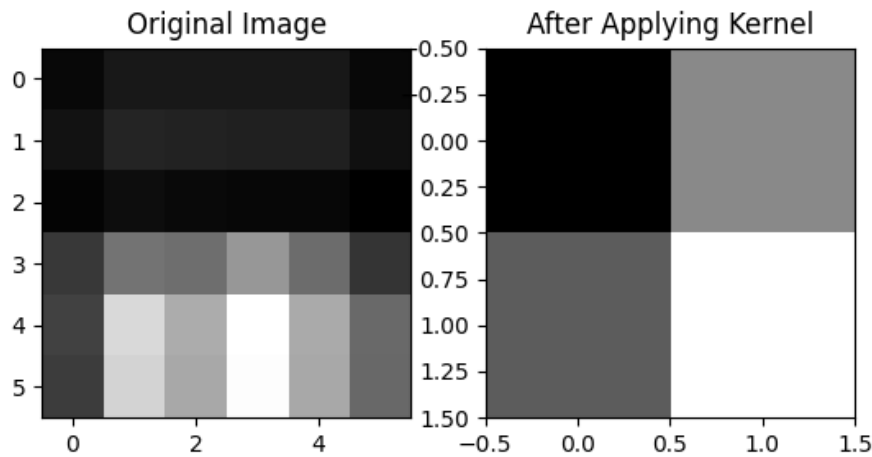


Figure 19: Before and After of the input image

**(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:

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

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

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

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:

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

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 downsampling 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:  $\text{ReLU}(x) = \max(x, 0)$

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

$$\max(a, b) = a \cdot \text{ReLU}(a - b) + b \cdot \text{ReLU}(b - a). \quad (21)$$

Therefore, now we need to prove it:

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

By replacing these values into Equation 21 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  $a < b$ :**

$$a - b < 0 \text{ and } b - a > 0.$$

$$\text{ReLU}(a - b) = 0 \text{ and } \text{ReLU}(b - a) = b - a$$

By replacing these values into Equation 21 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  $\max(a, b) = a \cdot \text{ReLU}(a - b) + b \cdot \text{ReLU}(b - a)$ .

## (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 a 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 value 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 21, we will try to implement the max-pooling operation by means of convolutions and ReLU Layers. For this implementation we will need:

- Convolutional Kernel: We can use a  $2 \times 2$  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  $2 \times 2$  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  $2 \times 2$  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 operation reduces the spatial 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  $2 \times 2$  convolution,  $2^2 = 4$  layers and channels are needed. For a  $3 \times 3$  convolution,  $3^2 = 9$  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:**  $100 \times 100$  grayscale images.
- **Layer 1:** Convolutional layer with 100  $5 \times 5$  convolutional filters.
- **Layer 2:** Convolutional layer with 100  $5 \times 5$  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:  $100 \times 100$ .
- Filter size:  $5 \times 5$ .
- Number of filters: 100.
- Weights: Each filter has  $5 \times 5$  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

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 reduced 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:

$$\text{number of neurons} \times \text{number of neurons of the previous layer} + \text{number of neurons} = 100 \times 211\,600 + 100 = 21\,160\,100.$$

**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:

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

**Layer 2:**

- Input channels: 100 (from layer 1).
- Filter size:  $5 \times 5$ .
- Number of filters: 100.
- Weights: Each filter has  $5 \times 5$  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).



## Problem 15

### (a) Question A

Reducing computing time and resource consumption through efficient techniques is necessary for implementing rapid convolutions in image processing or deep learning activities. Convolution of an image with a  $k \times k$  kernel is a popular technique. There are quite some techniques to implement it. In this exercise we will examine two of them.

One method is to scan horizontally across the source, reading a  $k - \text{wide}$  strip and computing the 1-wide output strip one value at a time.

Another alternative, possibly more effective approach is to read a  $k + \Delta$  wide strip and compute a  $\Delta - \text{wide}$  output strip.

Due to the fact that it can eliminate pointless calculations, enhance memory access patterns and make better use of hardware acceleration features, this alternative method is frequently chosen. The selection of  $\Delta$ , however, is crucial since an excessively high value could result in higher memory needs and lower performance gains. Thus, the ideal choice of  $\Delta$  strikes a balance between computational effectiveness and memory and hardware limitations.

In more details, this approach is often known as “vectorized convolution” or “strip mining” and it can lead to more efficient convolution operations.

Here are some advantages of the alternative approach and why the latter is preferred:

- Memory Access Pattern

This approach reads a wider strip of input data,  $k + \Delta \text{wide}$ , all at once, increasing the amount of data reuse. In comparison to reading  $k - \text{wide}$  strips repeatedly, this can make greater use of memory hierarchy and cache, reducing the amount of memory accesses.

- Parallelization

The wider strip allows for more opportunities for parallelism since we can perform computations for multiple output values simultaneously. This can be beneficial for optimizing computations on modern hardware, like GPUs or parallel CPU architectures that have vectorized instructions and perform the same operation on multiple data points in parallel. We can make better use of these vectorized processes by computing an  $\Delta - \text{wide}$  output strip, which will allow you to process several outputs in parallel and greatly improve computational performance.

- Reduced Overhead

Reading a larger strip of input data with fewer iterations, reduces loop overhead and branching, which can lead to improved performance.

- Input/Output Bandwidth Utilization

As the overhead of each read operation is reduced, reading data in bigger chunks can make better use of the memory bandwidth.

However, there is a limit to how large we should choose the value of  $\Delta$ , and it depends on various factors, such as memory constraints, cache size and hardware architecture. By increasing  $\Delta$  too much it could lead to increased memory usage and cache trashing, negating the benefits of the approach. Larger  $\Delta$  values require more memory to store the wider strips of the input image and intermediate computation results. Moreover, a larger  $\Delta$  requires more computation to process the wider strip, which might introduce additional overhead and in the end it may not yield significant performance improvements.

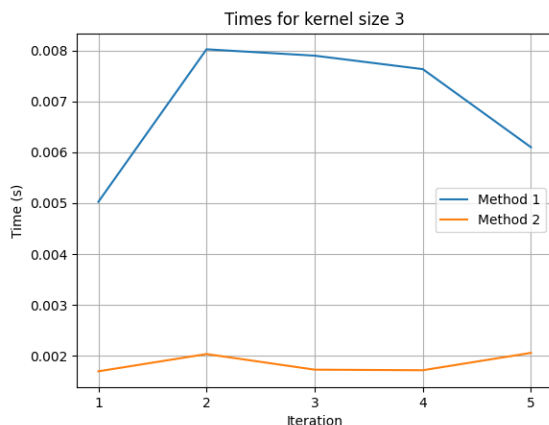
Everything considered, finding a balance between the advantages of more parallelism and data reuse and the possible disadvantages of more memory and processing is crucial. We could find the best option for the value of  $\Delta$  by profiling and experimenting with various values on the particular hardware we are targeting. In practice, finding the ideal  $\Delta$  often involves empirical testing and optimization based on the specific application and hardware configuration.

Overall, the alternative approach of reading a  $k + \Delta$  wide strip and computing a  $\Delta$  - wide output strip is preferable due to improved memory access patterns, increased parallelism, and reduced overhead. However, the choice of  $\Delta$  should be carefully considered based on hardware limitations and performance trade-offs.

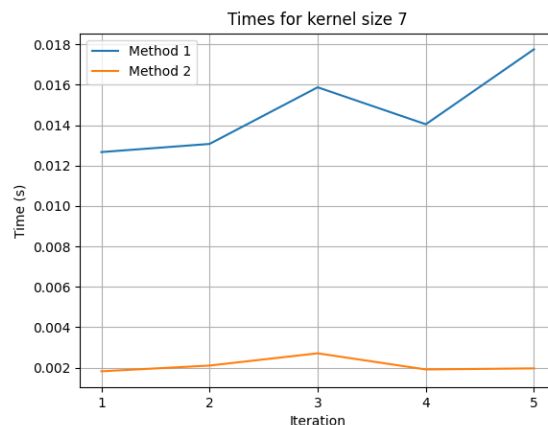
## (b) Question B

We will attempt to apply and test these two approaches in relation to one another. For this reason, we will construct a random sample image of size  $228 \times 228$ . A sequence of random  $3 \times 3$ ,  $7 \times 7$ , and  $11 \times 11$  kernels will then convolve with this image. We can have a better understanding of the two options and the reasons for the preference for the second one by comparing their execution times.

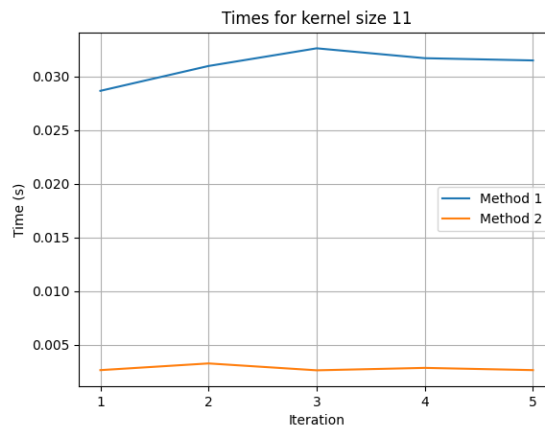
We will extract our results from the "*sample\_times.py*" code.



(a) Plot for  $3 \times 3$  kernel



(b) Plot for  $7 \times 7$  kernel



(c) Plot for  $11 \times 11$  kernel

The provided plots offer a visual representation of the performance metrics for each method, allowing us to

discern patterns and draw conclusions. With these visual data points we will analyze with more details the execution times.

### For 3x3 kernel

- Method 1: The execution times start relatively high and show a notable peak at iteration two. This could indicate that for small kernel sizes, the method's overhead associated with reading data and computing the output sequentially impacts performance negatively. This peak may suggest a temporary computational load or caching issue that affected the performance at this specific point. Following the peak, there is a slight reduction in the execution time for subsequent iterations, which could indicate some level of caching benefit as the computation progresses.
- Method 2: The consistent low execution times suggest that reading a  $k + \Delta$  wide strip and computing a  $\Delta - \text{wide}$  output is significantly more efficient. The method likely benefits from increased data locality, allowing for better utilization of cache hierarchies and minimizing the number of memory accesses required. It shows a remarkably consistent execution time across all iterations, reinforcing its efficient use of resources and possibly better memory access patterns.

### For 7x7 kernel

- Method 1: As the kernel size increases, the execution times for Method 1 also rise. There is a steady increase in execution time from the first to the fifth iteration, with the peak occurring at the fifth iteration and continuing increasing. This pattern implies that the performance of the approach deteriorates with increasing kernel size, most likely because of higher computational load and more frequent memory access conflicts.
- Method 2: Remarkably, this method shows an incremental rise in execution time but remains substantially faster than Method 1. It maintains a remarkably consistent and low execution time throughout all iterations. This resilience to increased kernel size suggests that Method 2 efficiently manages the additional computational complexity, potentially through better parallelization, without significant impact on performance..

### For 11x11 kernel

- Method 1: The continued upward pattern in execution time for larger kernels confirms the method's limited scalability. Method 1 starts with a higher execution time in the first iteration, which then remains relatively flat for subsequent iterations. The significant increase from 7x7 to 11x11 kernels suggests that the method may be approaching a computational bottleneck, possibly due to an exponential increase in the number of operations required.
- Method 2: Even at this larger kernel size, Method 2's execution time increases only slightly, maintaining a considerable lead over Method 1. Method 2 exhibits a very stable and low execution time across all five iterations, with no noticeable increase or decrease. This indicates a near-linear scalability, likely due to the method's effective amortization of the computational and memory access costs over a larger output strip.

The major differences in efficiency and scalability between the two approaches are made clear by this in-depth examination of the supplied charts. For the purpose of this exercise and in general for computational neural networks and image processing we can truly understand why Method 2 is the better choice. It is due to its consistent and better performance at all kernel sizes, its stability and low execution time at all costs. Last but not least, Method 2 can be considered more memory friendly, as it likely reduces memory bandwidth requirements and cache thrashing compared to Method 1. The consistent execution times of Method 2 across various kernel sizes support the notion of its efficient memory usage.