University of Thessaly

Neuro-Fuzzy Computing

ECE447

2nd Problem Set

Alexandra Gianni ID: 3382

Nikos Stylianou ID: 2917

January 29, 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

Problem 5

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
- 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:

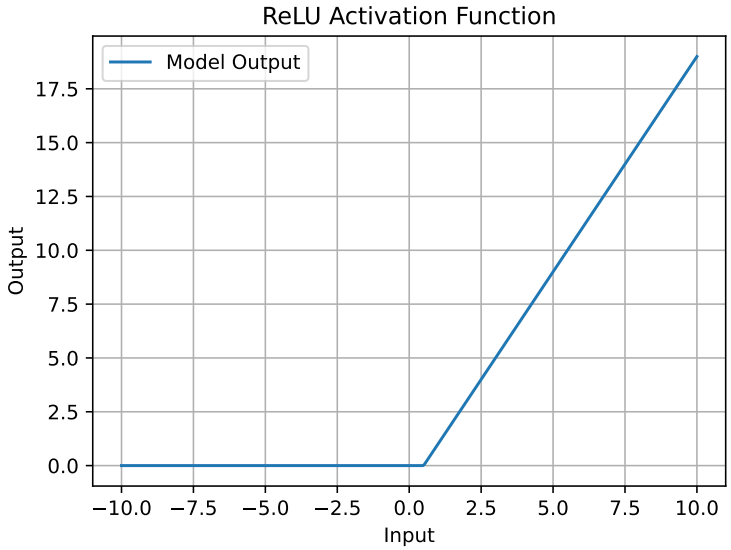

Figure 1: Plot of the MLP using the ReLU activation function

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.

$$I = \begin{bmatrix} 20 & 35 & 35 & 35 & 35 & 20 \\ 29 & 46 & 44 & 42 & 42 & 27 \\ 16 & 25 & 21 & 19 & 19 & 12 \\ 66 & 120 & 116 & 154 & 114 & 62 \\ 74 & 216 & 174 & 252 & 172 & 112 \\ 70 & 210 & 170 & 250 & 170 & 110 \end{bmatrix} \tag{3}$$

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

$$result = \begin{bmatrix} 225 & 258 & 250 & 209 \\ 458 & 566 & 552 & 472 \\ 708 & 981 & 887 & 802 \\ 1000 & 1488 & 1320 & 1224 \end{bmatrix} \tag{4}$$

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.

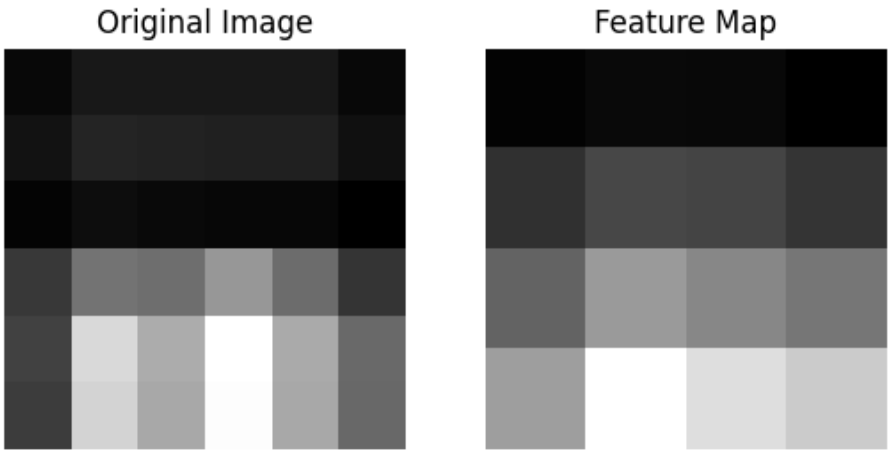

Figure 2: The Original Image and the Image after the convolution

(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

$$max\_pooling = \begin{bmatrix} 566 & 552 \\ 1488 & 1320 \end{bmatrix} \tag{5}$$

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 3) we have the following results:

- Filter $F1$

$$F1 = \begin{bmatrix} -10 & -10 & -10 \\ 5 & 5 & 5 \\ -10 & -10 & -10 \end{bmatrix} \tag{6}$$

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:

$$I\_F1 = \begin{bmatrix} -925 & -1040 & -1000 & -845 \\ -3900 & -4895 & -4825 & -4160 \\ -3750 & -5120 & -4650 & -4210 \\ -5200 & -6990 & -6750 & -5920 \end{bmatrix} \tag{7}$$

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.

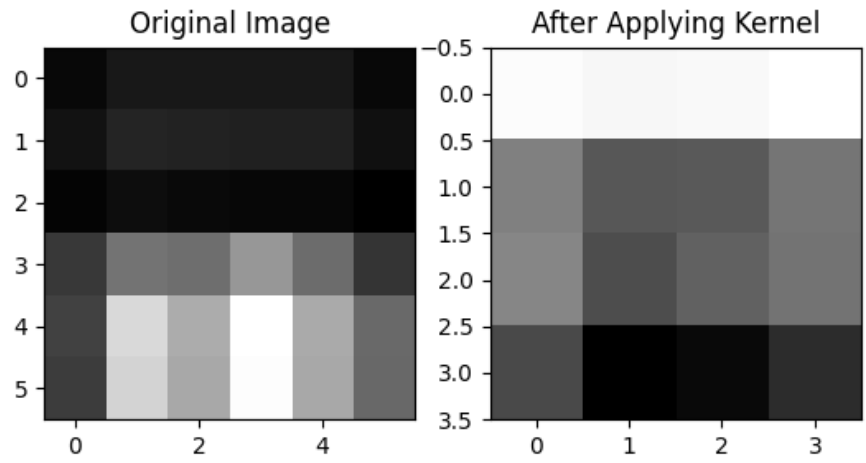

Figure 3: Before and After of the input image

- Filter $F2$

$$F2 = \begin{bmatrix} 2 & 2 & 2 \\ 2 & -12 & 2 \\ 2 & 2 & 2 \end{bmatrix} \tag{8}$$

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:

$$I\_F2 = \begin{bmatrix} -102 & -12 & -4 & -86 \\ 616 & 880 & 876 & 716 \\ -24 & 570 & -74 & 236 \\ -592 & 888 & -384 & 384 \end{bmatrix} \tag{9}$$

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:

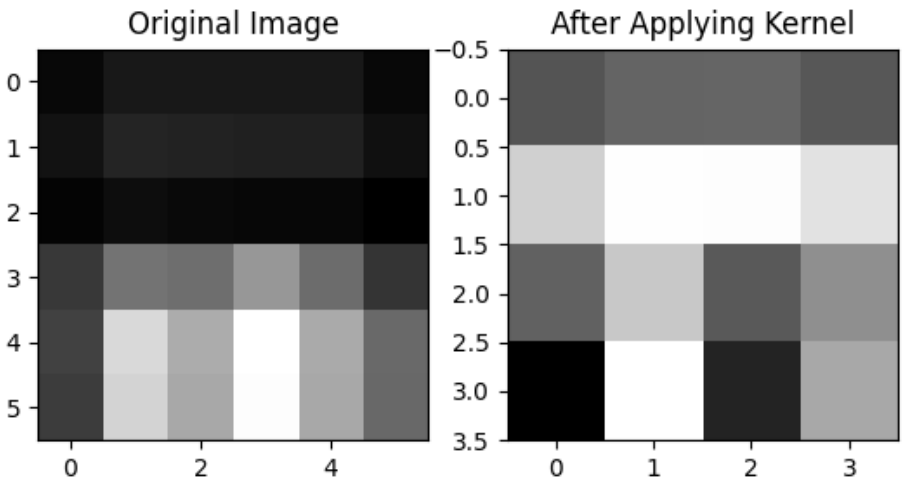

Figure 4: Before and After of the input image

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 $F3$

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 from $100 \times 100 \rightarrow 50 \times 50$.
- **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, in total we have $2500 + 100 = 2600$ weights.

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.

Moving on to **layer 3**, this layer doesn't have any weights or biases because it's a pooling layer. After max pooling, there's a reduction in layer's 2 output, from $100 \times 100 \rightarrow 25 \times 25$ and there are still 100 channels.
Layer 4 is a dense layer (*fully connected*) which has input size of $25 \times 25 \times 100 = 62500$. So, in order to calculate the total number of weights, we multiply the input size with the number of channels and add the number of total biases (*which is equal to the number of channels*), which is equal to $62500 \times 100 + 100 = 6\,250\,000 + 100 = 6\,250\,100$.
Layer 5 is also a dense layer and the procedure for calculating the weights is the same as above. We have 100 input units from layer 4 and 100 output units, so the 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:

$$\text{Total Weights} = \sum (\text{layers num}) =$$
$$= \text{Layer 1} + \text{Layer 2} + \text{Layer 3} + \text{Layer 4} + \text{Layer 5} + \text{Layer 6} =$$
$$= 2600 + 250\,100 + 6\,250\,100 + 10\,100 + 101 = \mathbf{6\,513\,001}$$