Neuro-Fuzzy Computing
2nd Problem Set

Ioannis Roumpos 2980
Konstantinos Vermisoglou 2988
Nikos Gkagkosis 3079

February 10, 2024

Contents

1 Problem - 01

The Fletcher-Reeves Method is distinguished by the different calculation of the β value and it is commonly used in the conjugate gradient method. Below, we can see how this method works in detail.

Algorithm 1 Fletcher-Reeves Method

```
1: Input: Objective function $f(x)$, initial guess $x_0$, tolerance $\epsilon$.
2: Set $k = 0$, $g_0 = \nabla f(x_0)$, and $d_0 = -g_0$.
3: while $||g_k|| > \epsilon$ do
4:     Choose step size $\alpha_k$ by line search.
5:     Update $x_{k+1} = x_k + \alpha_k d_k$.
6:     Compute $g_{k+1} = \nabla f(x_{k+1})$.
7:     Compute $\beta_{k+1} = \frac{||g_{k+1}||^2}{||g_k||^2}$.
8:     Update $d_{k+1} = -g_{k+1} + \beta_{k+1} d_k$.
9:     Set $k = k + 1$.
10: end while
11: Output: Optimal solution $x^*$.
```

For the given function, $f(w) = w_1^2 + w_2^2 + (0.5w_1 + w_2)^2 + (0.5w_1 + w_2)^4$ we start with $w(0) = [3, 3]^T$. Calculating the gradient $\nabla f(w)$

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

We will present the calculations for the first iteration and likewise the next iterations are calculated.

$$d_0 = -g_0 = \nabla f(w(0)) = \begin{pmatrix} -192.75 \\ -379.5 \end{pmatrix}$$

To choose the step size a_k we use backtracking line search.

$$a_0 = 0.003$$

The next value of weights is

$$w_1 = w_0 + a_0 \cdot d_0 = \begin{pmatrix} 2.2718 \\ 1.5667 \end{pmatrix}$$

$$g_1 = \nabla f(w(1)) = \begin{pmatrix} 46.70790813 \\ 87.46115099 \end{pmatrix}$$

$$b_1 = \frac{g_1^T \cdot g_1}{g_0^T \cdot g_0} = 0.0542$$

In the following lines we present the results for the first 5 iterations of the method.

1st iteration: $\mathbf{w}$ = [2.27181109 1.56628954]
2nd iteration: $\mathbf{w}$ = [1.74453665 0.56966571]
3rd iteration: $\mathbf{w}$ = [1.37544248 0.00258434]
4th iteration: $\mathbf{w}$ = [0.84503177 -0.42619371]
5th iteration: $\mathbf{w}$ = [0.21043746 -0.35644691]

Algorithm 2 Gradient Descent Method

```
1: Input: Objective function $f(x)$, initial guess $x_0$, step size $\alpha$, tolerance $\epsilon$.
2: Set $k = 0$.
3: while $||\nabla f(x_k)|| > \epsilon$ do
4:     Compute the gradient $\nabla f(x_k)$.
5:     Update $x_{k+1} = x_k - \alpha \nabla f(x_k)$.
6:     Set $k = k + 1$.
7: end while
8: Output: Optimal solution $x^*$.
```

We assume that we have unit step movement ,so computing the gradient for the first iteration as above

$$\nabla f(w(0)) = \begin{pmatrix} 192.75 \\ 379.5 \end{pmatrix}$$

$$w(1) = w(0) - a \cdot \nabla f(w(0)) = \begin{pmatrix} 2.375 \\ 1.769 \end{pmatrix}$$

So for the next iterations we have the following results:

2nd iteration

$$\nabla f(w(1)) = \begin{pmatrix} 59.371 \\ 112.782 \end{pmatrix}$$

$$w(2) = \begin{pmatrix} 1.936 \\ 0.936 \end{pmatrix}$$

3rd iteration

$$\nabla f(w(2)) = \begin{pmatrix} 19.60244 \\ 33.33179 \end{pmatrix}$$

$$w(2) = \begin{pmatrix} 1.6024482 \\ 0.3684566 \end{pmatrix}$$

<u>**4th iteration**</u>

$$\nabla f(w(3)) = \begin{pmatrix} 7.575182 \\ 9.477483 \end{pmatrix}$$

$$w(4) = \begin{pmatrix} 1.28702183 \\ -0.02618057 \end{pmatrix}$$

<u>**5th iteration**</u>

$$\nabla f(w(4)) = \begin{pmatrix} 3.661899 \\ 2.123350 \end{pmatrix}$$

$$w(5) = \begin{pmatrix} 0.8136731 \\ -0.3006515 \end{pmatrix}$$

<u>**6th iteration**</u>

$$\nabla f(w(5)) = \begin{pmatrix} 1.7359259 \\ -0.3841438 \end{pmatrix}$$

$$w(6) = \begin{pmatrix} 0.009305505 \\ -0.122652638 \end{pmatrix}$$

<u>**7th iteration**</u>

$$\nabla f(w(6)) = \begin{pmatrix} -0.1026749 \\ -0.4878772 \end{pmatrix}$$

$$w(7) = \begin{pmatrix} 0.03194666 \\ -0.01506941 \end{pmatrix}$$

<u>**8th iteration**</u>

$$\nabla f(w(7)) = \begin{pmatrix} 0.06479723 \\ -0.02833099 \end{pmatrix}$$

$$w(8) = \begin{pmatrix} -0.0003451292 \\ -0.0009506256 \end{pmatrix}$$

9th iteration

$$\nabla f(w(8)) = \begin{pmatrix} -0.001813451 \\ -0.004147637 \end{pmatrix}$$

$$w(9) = \begin{pmatrix} 5.845172e-05 \\ -2.757511e-05 \end{pmatrix}$$

10th iteration

$$\nabla f(w(9)) = \begin{pmatrix} 1.185542e-04 \\ -5.184872e-05 \end{pmatrix}$$

$$w(10) = \begin{pmatrix} -6.306587e-07 \\ -1.735908e-06 \end{pmatrix}$$

As we observe the minimum is well approximated in both algorithms but better in gradient since we have more iterations.

2 Problem - 02

$f(w_1, w_2) = w_1^2 + w_2^2 + (0.5w_1 + w_2)^2 + (0.5w_1 + w_2)^4 \quad w(0) = (3, 3)$

Calculating the gradient $\nabla f(w)$

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

Calculating the Hessian $\nabla^2 f(w)$

$$H(w) = Hessian = \begin{pmatrix} 3(0.5w_1 + w_2)^2 + 2.5 & 6(0.5w_1 + w_2)^2 + 1 \\ 6(0.5w_1 + w_2)^2 + 1 & 12(0.5w_1 + w_2)^2 + 4 \end{pmatrix}$$

The Newton's algorithm consists of the following steps which are followed until the convergence criterion is satisfied.

Step 1:

Calculate the gradient $\nabla f(w(k))$ and the Hessian $H(w(k))$

Step 2:

Compute the minimizing direction: $s(k) = -H(k)^{-1} * \nabla f(w(k))$

Step 3:

Minimize $f(w(k) + \lambda_k * s_k)$ to find the optimal $\lambda_k, (\lambda_k \geq 0)$ but we assume that $\lambda_k = 1, \forall k$.

Step 4:

Find the next point: $w(k+1) = w(k) + \lambda * s(k)$

Step 5:

Check convergence criterion and in our case we choose the following criterion: $\|w(k+1) - w(k)\| < 10^{-6}$

An approximation to the minimum is found after **8 iterations** where the convergence criterion is met. We observe that using $\lambda_k = 1$, even though the function is non quadratic,results in more iterations before convergence.

3 Problem - 03

Performing backpropagation for the following neural network.

Activation function $Swish = \frac{x}{e^x+e^{-x}}$ and the derivative of Swish is *Swish'* $= \frac{x}{1+e^{-x}}(1-\frac{1}{1+e^{-x}})+\frac{1}{1+e^{-x}}$.

Activation function LReLU with leaky parameter 0.001:

$$LReLU = \begin{cases} x & x \geq 0 \\ 0.001x & x < 0 \end{cases} \tag{1}$$

Derivative of LReLU:

$$LReLU = \begin{cases} 1 & x \geq 0 \\ 0.001 & x < 0 \end{cases} \tag{2}$$

Starting backpropagation with the **1st (first) iteration**:

Forward Phase:

$n^1 = w^1 p + b^1 = -1$ $\quad a^1 = Swish(n^1) = -0.269$

$n^2 = a^1 w^2 + b^2 = -0.731$ $\quad a^2 = LReLU(n^2) = -0.000731$

$e = t - a^2 = 0.000731$

Backward Phase:

$s^2 = -2F'^2(n^2)(t-a) = -1.5 \cdot 10^{-6}$, where $F'^2(n^2) = 0.001$ since $n^2 < 0$.

$s^1 = F'^1(n^1)(W^2)^T s^2 = 108{\cdot}10^{-9}$, where $F'^1(n^1) = 0.0072$ the derivative of swish.

Weight Correction Phase:

$W^1(1) = W^1(0) - as^1(a^0)^T = -3 - 108 \cdot 10^{-9}$

$b^1(1) = b^1(0) - as^1 = 2 - 108 \cdot 10^{-9}$

$W^2(1) = W^2(0) - as^2(a^1)^T = -1 - 4 \cdot 10^{-7}$

$b^2(1) = b^2(0) - as^2 2 = -1 + 1.5 \cdot 10^{-6}$

2nd (second) iteration:

Forward Phase:

$n^1 = w^1(1)p + b^1(1) = -1 - 216 \cdot 10^{-9}$ $a^1 = Swish(n^1) = -0.269$

$n^2 = a^1 w^2(1) + b^2(1) = -0.731$ $a^2 = LReLU(n^2) = -0.000731$

$e = t - a^2 = 0.000731$

Backward Phase:

$s^2 = -2F'^2(n^2)(t-a) = -1.5 \cdot 10^{-6}$, where $F'^2(n^2) = 0.001$ since $n^2 < 0$.

$s^1 = F'^1(n^1)(W^2)^T s^2 = 108 \cdot 10^{-9}$, where $F'^1(n^1) = 0.0072$ the derivative of swish.

Weight Correction Phase:

$W^1(2) = W^1(1) - as^1(a^0)^T = -3 - 108 \cdot 10^{-9}$

$b^1(2) = b^1(1) - as^1 = 2 - 108 \cdot 10^{-9}$

$W^2(2) = W^2(1) - as^2(a^1)^T = -1 - 4 \cdot 10^{-7}$

$b^2(2) = b^2(1) - as^2 2 = -1 + 1.5 \cdot 10^{-6}$

4 Problem - 04

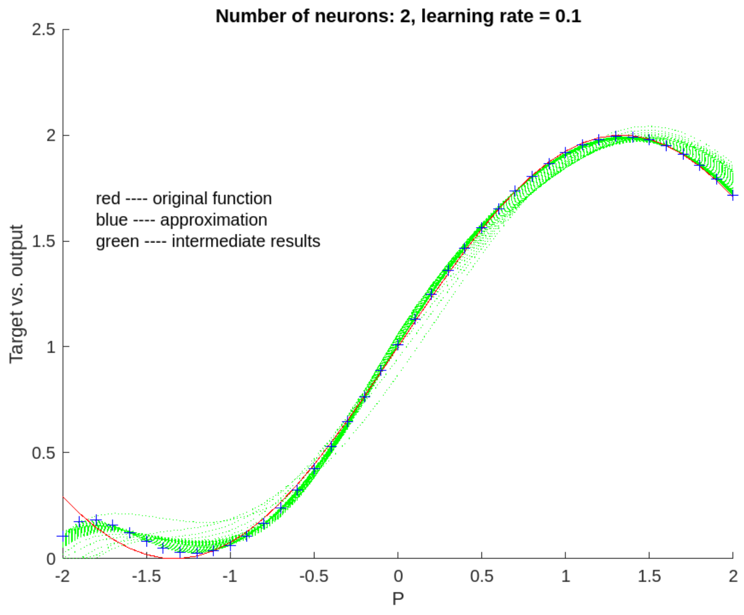

Figure 1: 2 neurons and a = 0.1

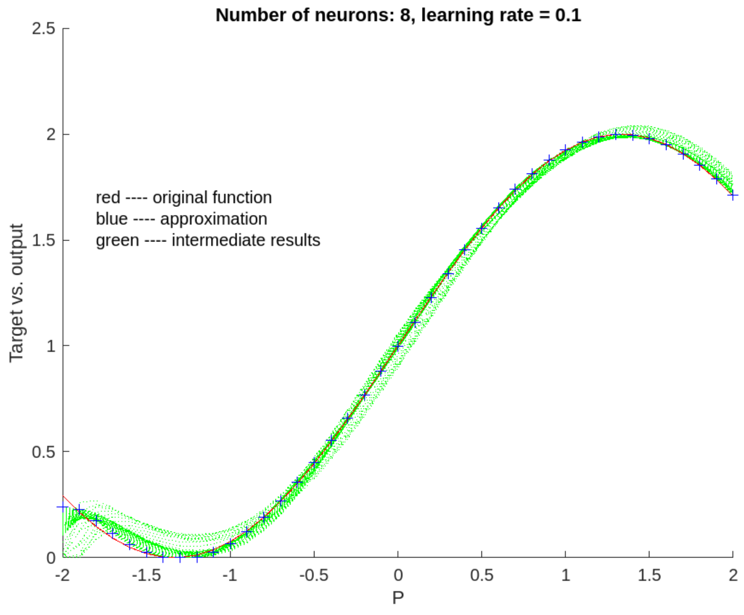

Figure 2: 8 neurons and a = 0.1

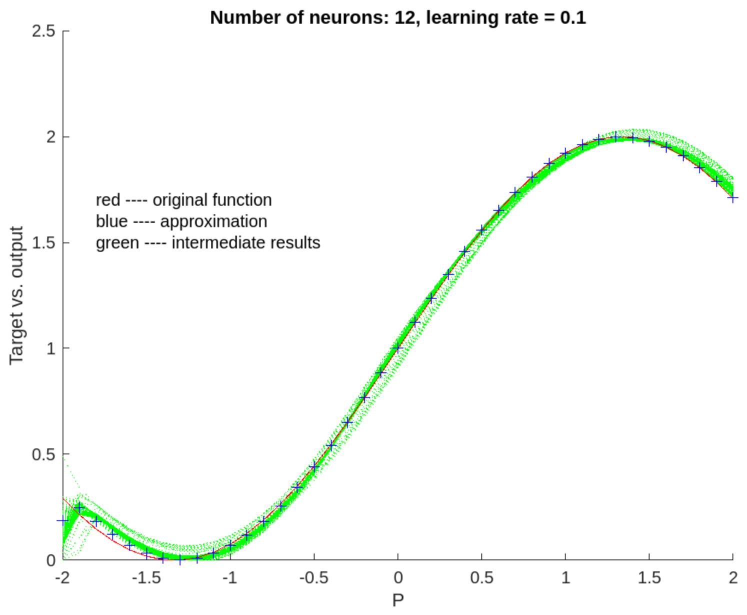

Figure 3: 12 neurons and a = 0.1

As we can observe from the figures above the optimal function approximation is succeeded when the number of neurons equals to 8. For neurons = 2 the NN cannot approximate the function as well as with 8 because the capabilities of the network were inherently limited by the number of hidden neurons it contained. On the other way, using 12 neurons gives a very good approximation but not as good as the one with 8 neurons due to the high complexity introduced by the number of neurons used.

Convergence on different setups

We experimented we different values for the learning rate(a = 0.4,0.6,0.8) and different initial conditions for different number of neurons in the hidden layer(S = 2,8,12). We can conclude from the figures that are displayed below that as the learning rate increases and in combination with the increased number of neurons and hyperparemeters the NN cannot approximate the original function. Also, for higher number of neurons(8,12) the approximation seem not to be good when $a \geq 0.6$ when for smaller S(= 2) this applies for higher a (= 0.8).

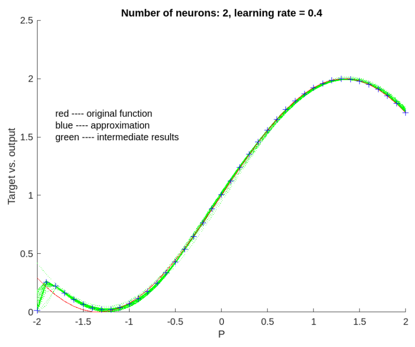

Figure 4: S = 2, a = 0.4

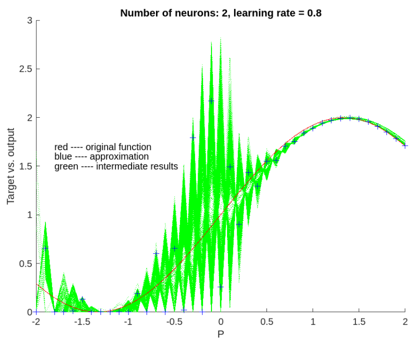

Figure 5: S = 2, a = 0.8

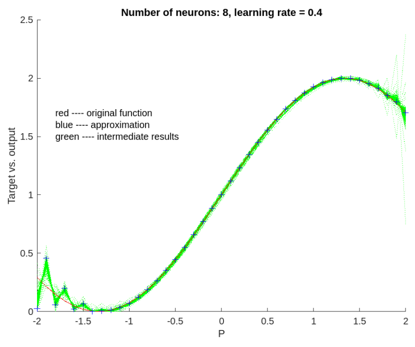

Figure 6: S = 8, a = 0.4

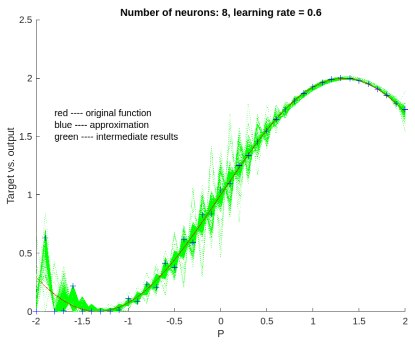

Figure 7: S = 8, a = 0.6

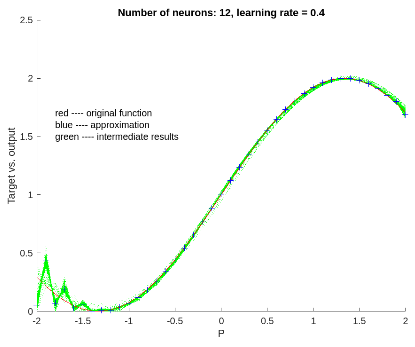

Figure 8: S = 12, a = 0.4

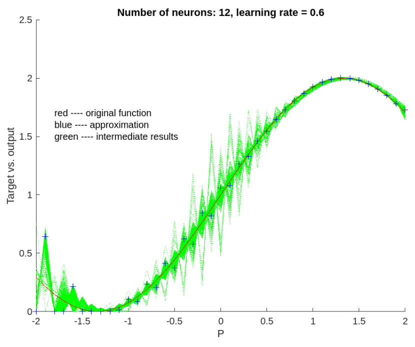

Figure 9: S = 12, a = 0.6

5 Problem - 05

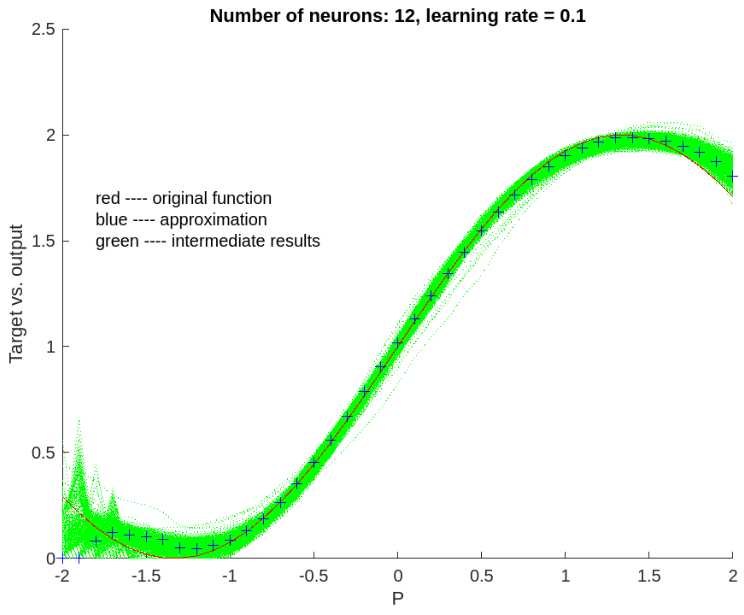

Figure 10: Dropout prob = 0.15

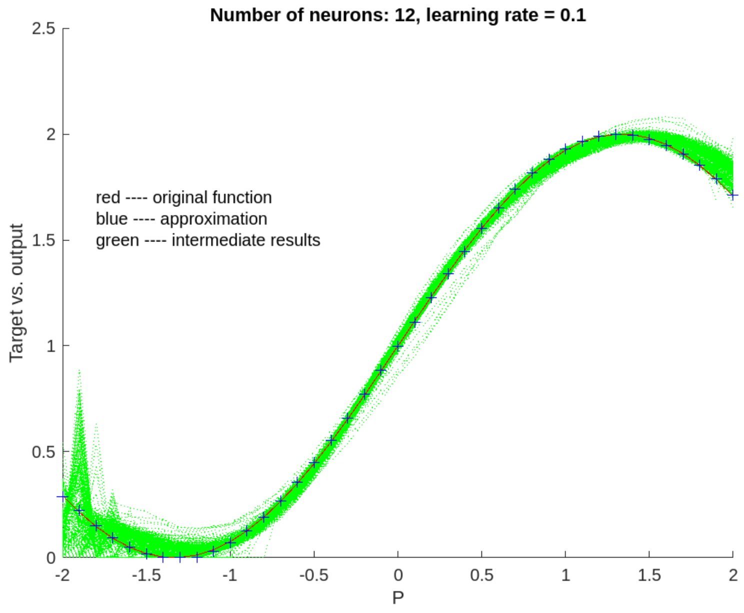

Figure 11: Dropout prob = 0.25

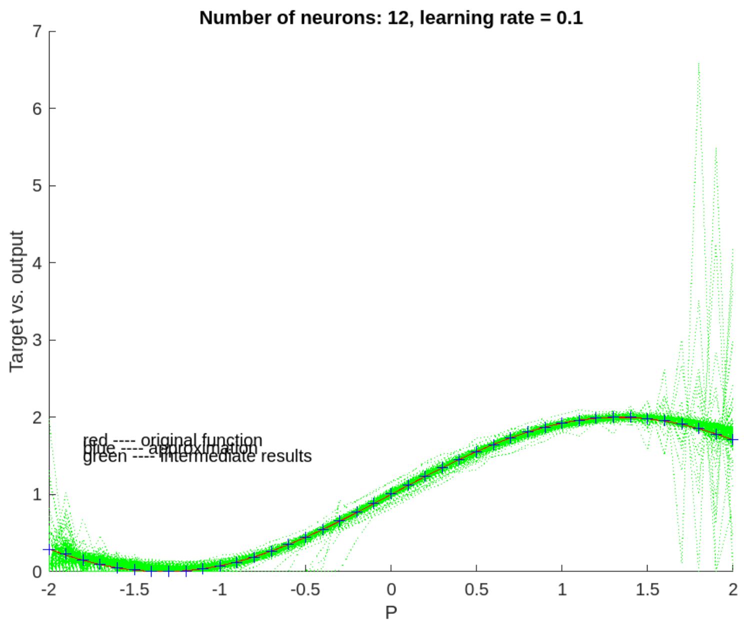

Figure 12: Dropout prob = 0.35

In the figures above we can see the how the function is approximated using the dropout technique. As we can observe the best result is when we define **the dropout probability = 0.35**. The reason why we see these results lie in the fact that by randomly deactivating a proportion of neurons during training, dropout encourages robustness and generalization, enhancing the network's ability to perform well on unseen data. Also, dropout acts as an ensemble learning approach within a single model, effectively training multiple architectures simultaneously. This diversification reduces model variance and improves performance. Moreover, dropout facilitates faster convergence during training by preventing the dominance of certain neurons, which encourages more evenly distributed learning across the network. Lastly, dropout often leads to simpler and more interpretable models by discouraging complex co-adaptations, thus enhancing model transparency and understandability, which is crucial in many real-world applications.

6 Problem - 06

a)
$n = p_1 w_1 + p_1 p_2 w_{12} + p_2 w_2 + b$

$a = \tanh(n)$

Performance index: $\hat{F}(x) = (t(k) - a(k))^2 = (e(k))^2$

Wright updates: $\Delta_{w_i} = -a \cdot \frac{\partial}{\partial w_i} \hat{F}(x)$

$\frac{\partial}{\partial w_i} \hat{F}(x) = \frac{\partial}{\partial w_i} (t(k) - a(k))^2 = 2\,(t(k) - a(k)) \cdot \left\{ -\frac{\partial a(k)}{\partial w_i} \right\}$

- $\frac{\partial a(k)}{\partial w_1} = \frac{\partial \tanh(n)}{\partial w_1} = \left[1 - \tan^2 h(n)\right](n)' = \left[1 - \tan^2 h(n)\right](p_1)$
- $\frac{\partial a(k)}{\partial w_{1,2}} = \frac{\partial \tanh(n)}{\partial w_{1,2}} = \left[1 - \tan^2 h(n)\right](p_1 \cdot p_2)$
- $\frac{\partial a(k)}{\partial w_2} = \frac{\partial \tanh(n)}{\partial w_2} = \left[1 - \tan^2 h(n)\right](p_2)$

For bias: $\Delta b = -a \cdot \frac{\partial}{\partial b} \hat{f}(x) = 2\,(t(k) - a(k)) \cdot \left\{ -\frac{\partial a(k)}{\partial b} \right\}$

$\frac{\partial a(k)}{\partial b} = \left[1 - \tan^2 h(n)\right]$

b)

$n = p_1 \cdot \omega_1 + p_{12} \cdot \omega_{12} + p_2 \cdot \omega_2 + 1 = 0 \cdot 1 + 0 \cdot 1 \cdot 0.5 + (-1) + 1 = 0$

$a = \tanh(n) = 0$

- $\Delta W_1 = -2(t(k) - a(k)) \cdot \left[1 - \tanh^2(n)\right] \cdot p_1 = 0$
- $\Delta W_{1,2} = -2(t(k) - a(k)) \cdot \left[1 - \tanh^2(n)\right] \cdot p_1 \cdot p_2 = 0$
- $\Delta W_2 = -2 \cdot (t(k) - a(k)) \cdot \left[1 - \tanh^2(n)\right] \cdot p_2 = -2 \cdot (0.75) \cdot 1 \cdot 1 = -1.5$
- $\Delta b = -2 \cdot (t(k) - a(k)) \cdot \left[1 - \tanh^2(n)\right] = -1.5$

$W_1(1) = W_1(0) + \Delta W_1 = 1 - 0 = 1$

$W_{1,2}(1) = W_{1,2}(0) + \Delta W_{1,2} = 0.5 - 0 = 0.5$

$W_2(1) = W_2(0) + \Delta W_2 = -1 - 1,5 = -2.5$

$b(1) = b(0) + \Delta b = 1 - 1.5 = -0.5$

7 Problem - 07

To show that a multi-layer perceptron (MLP) using only Rectified Linear Unit (ReLU) activation functions constructs a continuous piecewise linear function, we need to understand the behavior of ReLU and how it impacts the linearity of the function.

ReLU activation function is defined as:

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

It returns 0 if the input is negative and returns the input itself if it's positive. This function is linear for $x \geq 0$ and 0 for $x < 0$.

Now, let's consider a simple MLP architecture with ReLU activation functions. Suppose we have L layers, and let's denote the output of the i-th layer as $h_i(x)$, where x is the input to the network. Then the output of the network, denoted as $f(x)$, is given by:

$$f(x) = h_L(h_{L-1}(\dots(h_2(h_1(x)))))$$

Each $h_i(x)$ is a composition of affine transformations (weights and biases) followed by the ReLU activation function. Mathematically, if W_i and b_i represent the weights and biases of the i-th layer, respectively, then:

$$h_i(x) = \text{ReLU}(W_i \cdot h_{i-1}(x) + b_i)$$

Because ReLU is piecewise linear, each layer's output $h_i(x)$ is piecewise linear as well. The composition of piecewise linear functions is also a piecewise linear function. Hence, $f(x)$ is a piecewise linear function.

Additionally, since each ReLU unit is continuous, the composition of continuous functions remains continuous. Therefore, $f(x)$ is continuous everywhere.

In the case of parametric ReLU (pReLU), the function is similar, with the addition of a learnable parameter to determine the slope of the negative part of the function. This doesn't change the piecewise linearity property.

So, an MLP using only ReLU (or pReLU) activation functions constructs a continuous piecewise linear function.

8 Problem - 08

For this problem we used the optimizer Adadelta from the library of **Pytorch**. In the next plots we experimented with different learning rates. In our experiments we used **number of iterations = 1000** and as initial point $\mathbf{w}$ = (3,2).

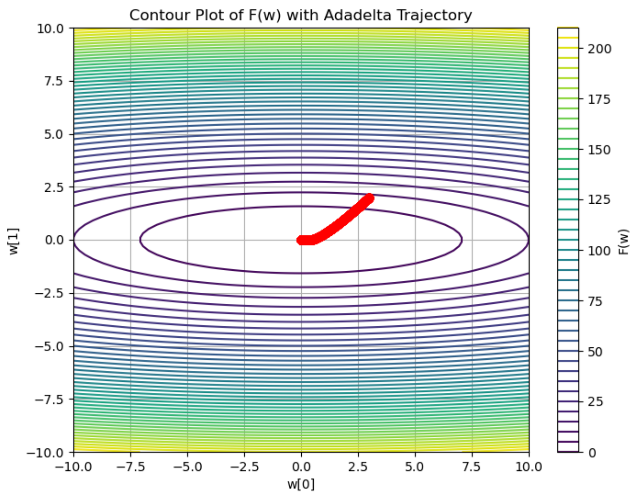

Figure 13: Trajectory with learning rate 0.4

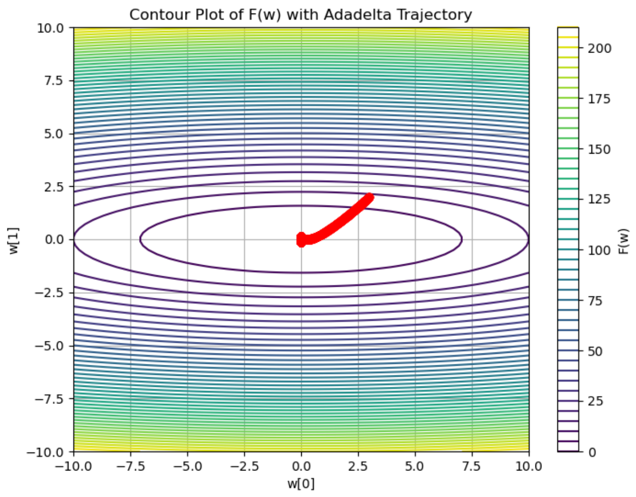

Figure 14: Trajectory with learning rate 3

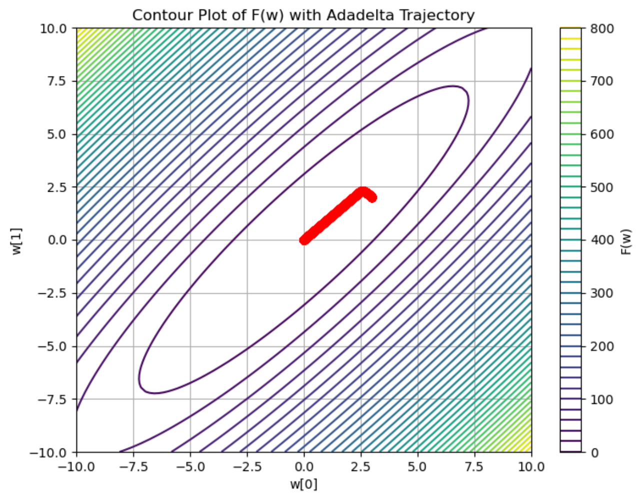

Figure 15: Trajectory with rotated function

a,b)
As we can see from the figures above the minimum point is found whatever the learning rate is but in the case of higher learning rate as we expected the trajectory is not straight and at the end point it seems to be stucked in a region before it finally found the minimum.
c)
Regarding the last figure, the plot is rotated by 45 degrees and rotating the function will also rotate the gradient vectors at each point. This means that the direction of steepest ascent/descent will be different from the original function. Gradients along the new coordinate axes will now influence the optimization process differently compared to before the rotation. The rotated contours align better with the axes of the optimization algorithm, thus the convergence is faster.

9 Problem - 09

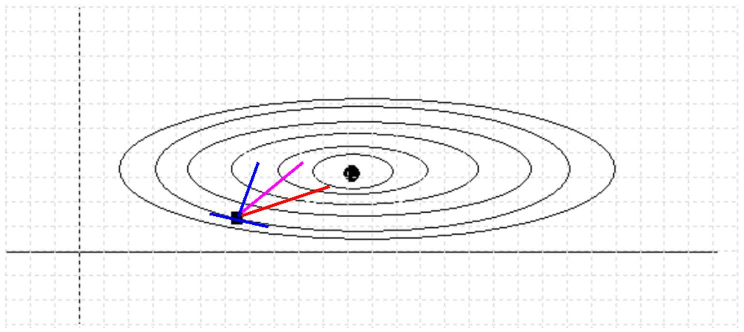

Figure 16: Directions of first iteration

- The blue line represents the direction of the first iteration for **standard gradient**. The line points perpendicular the contours of the function and that why we draw and the tangent line of that point.
- The red line represents the direction of the first iteration for **natural gradient**. The line is in the direction of negative Hessian inverse of the gradient. In other words, it is in a direction orthogonal to the sphered contours' lines direct path but not necessarily in a straight line.
- The magentian line represents the direction of the **Adagrad**. Based on the average root-mean squared gradient in a region the direction is calculated. In our function, the ratio of the axis seems to be 3:1 so the angle is ≈ 45 degrees.

10 Problem - 10

Original Form :

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

For $\beta = 0$ and v $= 0$ the original form transform into Gradient Descent.

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

For $b > 1, \quad b \to \mu$ and $b - 1 \to \varepsilon > 0$, v $= 1$, the original form transform into Momentum.

$g_{t+1} \leftarrow \mu \cdot g_t + \varepsilon \cdot \nabla \hat{L}(\theta_t)$
$\theta_{t+1} \leftarrow \theta_t - a \cdot \nabla \hat{L}(\theta_t)$

11 Problem - 11

A.

Size of the Output Tensor (Image) of a Conv Layer

Define

- O = Size (width) of output image
- I = Size (width) of input image
- K = Size (width) of kernels used in the Conv Layer
- N = Number of kernels
- S = Stride of the convolution operation
- P = Padding

The size of the output image is given by

$$O = \frac{I - K + 2P}{S} + 1 \tag{3}$$

So in our case

$$O = \frac{6 - 3 + 0}{1} + 1$$
$$O = 4$$

The number of channels in the output image is equal to the number of kernels N.
Hence, the output image is 4x4x1
The output of the convolution I $*$ kernel is:

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

B.

Size of Output Tensor (Image) of a MaxPool Layer

Define

- O = Size (width) of output image
- I = Size (width) of input image
- S = Stride of the convolution operation
- P_s = Pool size

The size of the output image is given by

$$O = \frac{I - P_s}{S} + 1 \tag{4}$$

The number of channels in the maxpool layer's output is unchanged.
In our case

$$O = \frac{4-2}{2} + 1$$
$$O = 2$$

Hence, the output image is 2x2x1. The ouput of the max pool layer is:

$$\begin{bmatrix} 566 & 552 \\ 1488 & 1320 \end{bmatrix}$$

C.
Intuition
The negative values in kernels correspond to darker pixel intensity values than the positive ones.
F1
The values of the kernel change by rows, so the tranistion from dark pixels to bright pixels and then again dark will detect horizontal edge. The resulting feature will have high activation at each position where a horizontal edge is seen.
F2
All the elements of F2 have high values except the central element. Hence, the resulting feature will have high activation when circles are seen.
F3
The elements in the right diagonal are higher than the remaining values of F3. A right diagonal edge will detect.

12 Problem - 12

Number of Parameters of a Convolutional Layer
Define

- W = Number of weights of the Convolutional Layer.
- B = Number of biases of the Conv Layer.
- P = Number of parameters of the Convolutional Layer.
- K = Size (width) of kernels used in the Convolutional Layer.
- N = Number of kernels.
- C = Number of channels of the input image.

$$W = K \cdot K \cdot C \cdot N \tag{5}$$
$$B = N \tag{6}$$
$$P = W + B \tag{7}$$

In a Convolutional Layer, the depth of every kernel is always equal to the number of channels in the input image. So every kernel has $K^2 \cdot C$ parameters, and there are N such kernels.

First hidden layer

The size of the kernel is 3x1, the input channels are 3 and the number of kernels are 4. From equations (5) (6) (7) we obtain:

$$W = 36$$
$$B = 4$$
$$P = 40$$

Second hidden layer

4 kernels produce 4 feature maps so the number of channels of the input (first hidden layer) are C = 4. Kernel size is 5x1 and there are 10 kernels so:

$$W = 200$$
$$B = 10$$
$$P = 210$$

13 Problem-13

A.

$$ReLU(z) = max(0, z)$$

Expressing max(a, b) by using ReLU operations:

$$max(a, b) = a + ReLU(b - a) \tag{8}$$

$$max(a, b) = ReLU(a) - ReLU(-a) + ReLU(b - a) \tag{9}$$

- If b-a is positive, then b is greater than a. ReLU return b-a and you get in equation (8) max(a,b) = b
- If b-a is zero or negative, then a is greater or equal to b. ReLU return 0 and you get in equation (8) max(a,b) = a

B.
Sample an input image 6x6. A kernel 2x2 with stride = 2 is used where the values of the kernel are $a_{11} = 1$ and all other elements are zero.In each layer the kernel switches 1 to the next position having all the other elements again zero.So for a kernel 2x2 the valid positions of 1 are 4 and this number is the number of layers. Also the convolution operation is having valid padding.

$$\mathrm{I} = \begin{bmatrix} 1.9 & 1.5 & 0.9 & -2.1 & 0.7 & -1.2 \\ 0 & -1.6 & -0.8 & 1.6 & -0.4 & -1.4 \\ -0.7 & -0.6 & -0.8 & 0.8 & 1.6 & -0.2 \\ -0.5 & 0.4 & 0.3 & -0.4 & 0.3 & -0.8 \\ 0 & 0.3 & 1.6 & -0.8 & -1.3 & 2.1 \\ -1.2 & -0.5 & -1.4 & 0.9 & 0 & 2.3 \end{bmatrix}$$

Layer 1

$$K_1 = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

$$\text{Convolution of I } *\mathrm{K}_1 = \begin{bmatrix} 1.9 & 0.9 & 0.7 \\ -0.7 & -0.8 & 1.6 \\ 0 & 1.6 & -1.3 \end{bmatrix} = \text{conv1}$$

The output of the convolution would be according to (3):

$$out = \frac{6-2}{2} + 1$$
$$out = 3$$

The output will be 3x3 and we initialize it before the result with large negative values in order to get the produced results of conv1 using the expression we obtained from A.

$$\text{out} = \begin{bmatrix} -1000 & -1000 & -1000 \\ -1000 & -1000 & -1000 \\ -1000 & -1000 & -1000 \end{bmatrix}$$

$max(conv1, out) = conv1 + ReLU(out - conv1)$ will take the element-wise maximum of the conv1 and the output.

$$\text{out} = \begin{bmatrix} 1.9 & 0.9 & 0.7 \\ -0.7 & -0.8 & 1.6 \\ 0 & 1.6 & -1.3 \end{bmatrix}$$

Layer 2

$$K_2 = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

$$\text{Convolution of I} * \text{K}_2 = \begin{bmatrix} 1.5 & -2.1 & -1.2 \\ -0.6 & 0.8 & -0.2 \\ 0.3 & -0.8 & 2.1 \end{bmatrix} = \text{conv2}$$

$max(conv2, out) = conv2 + ReLU(out - conv2)$ will give as result:

$$\begin{bmatrix} 1.9 & 0.9 & 0.7 \\ -0.6 & 0.8 & 1.6 \\ 0.3 & 1.6 & 2.1 \end{bmatrix}$$

Layer 3

$$K_3 = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$$

$$\text{Convolution of I} * \text{K}_3 = \begin{bmatrix} 0 & -0.8 & -0.4 \\ -0.5 & 0.3 & 0.3 \\ -1.2 & -1.4 & 0 \end{bmatrix} = \text{conv3}$$

$max(conv3, out) = conv3 + ReLU(out - conv3)$ will give as result:

$$\begin{bmatrix} 1.9 & 0.9 & 0.7 \\ -0.5 & 0.8 & 1.6 \\ 0.3 & 1.6 & 2.1 \end{bmatrix}$$

Layer 4

$$K_4 = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

$$\text{Convolution of I} * \text{K}_4 = \begin{bmatrix} -1.6 & 1.6 & -1.4 \\ 0.4 & -0.4 & -0.8 \\ -0.5 & 0.9 & 2.3 \end{bmatrix} = \text{conv4}$$

$max(conv4, out) = conv4 + ReLU(out - conv4)$ will give as result:

$$\begin{bmatrix} 1.9 & 1.6 & 0.7 \\ 0.4 & 0.8 & 1.6 \\ 0.3 & 1.6 & 2.3 \end{bmatrix}$$

We want to check in the window (2,2) of a pooling kernel all the elements and get the biggest one as a result in every slide of this window.
In layer1 we get as conv1 the elements a_{11} a_{13} a_{15}, then with stride = 2 , a_{31} a_{33} etc. and store this result.
In layer2 we get as conv2 the elements a_{12} a_{14} a_{16}, then with stride = 2 , a_{32} a_{34} etc.Then we compare element-wise conv2 and conv1 and keep the biggest ones.
In layer3 we get as conv3 the elements a_{21} a_{23} a_{25}, then with stride = 2 , a_{41} a_{43} etc. We compare element-wise conv2 and the output from the result of layer 2 and keep the biggest ones.
In layer4 we get as conv4 the elements a_{22} a_{24} a_{26}, then with stride = 2 , a_{42} a_{44} etc. Comparison of conv4 with the result from layer3.
Hence, we did compare all the elements in window (2,2) and kept the maximum values.

C.
In section B. we did 2x2 convolution and it required 4 kernels which means 4 channels and 4 layers. In 3x3 convolution we need 9 channels each one having value 1 in a different position and 9 layers.

14 Problem-14

Layer 1

From equation (3):

$$O = \frac{100-5}{1} + 1$$
$$O = 96$$

There are 100 kernels so depth = 100 of the output layer.
Output image: 96x96x100

Weights
From equations (5) (6) (7):

$$W_1 = 5 \cdot 5 \cdot 1 \cdot 100 = 2500$$
$$B_1 = 100$$
$$P_1 = W_1 + B_1 = 2600$$

Layer 2

$$O = \frac{96-5}{1} + 1$$
$$O = 92$$

100 filters were used so 100 output feature maps.
Output image: 92x92x100

Weights

$$W_2 = 5 \cdot 5 \cdot 100 \cdot 100 = 250.000$$
$$B_2 = 100$$
$$P_2 = W_2 + B_2 = 250.100$$

Layer 3

A pooling layer with 2x2 filter and stride = 2 will downsample layer 2 from 92x92 $\rightarrow$ 46x46.
This can be proved by (4):

$$O = \frac{I - P_s}{S} + 1$$
$$P_s = I + S - O \cdot S$$
$$P_s = 92 + 2 - 46 \cdot 2$$
$$P_s = 2$$

which means that pooling size of the kernel has to be 2x2.
The depth of the previous layer is the same with this layer.
Output image: 46x46x100

Weights
There are no weights in max pooling kernel, the values in the kernel are fixed.

Layer 4

Output vector: 100x1

Neurons in Layer 3

$$F_{-1} = Neurons = 46 \cdot 46 \cdot 100$$
$$Neurons = 211600$$

Neurons in Layer 4

$$F = Neurons = units = 100$$

where F_{-1} and F are the neurons in previous and current layer.Each neuron has its own bias.

Weights

$$W_4 = F_{-1} \cdot F$$
$$W_4 = 21.160.000$$
$$B_4 = 100$$
$$P4 = W_4 + B_4$$
$$P4 = 21.160.100$$

Layer 5

Output vector: 100x1

Neurons in Layer 4

$$F_{-1} = Neurons = units = 100$$

Neurons in Layer 5

$$F = Neurons = units = 100$$

Weights

$$W_5 = F_{-1} \cdot F$$
$$W_5 = 10.000$$
$$B_5 = 100$$
$$P5 = W_5 + B_5$$
$$P5 = 10.100$$

Layer 6

Output vector: 1x1

Neurons in Layer 5

$$F_{-1} = Neurons = units = 100$$

Neurons in Layer 6

$$F = Neurons = units = 1$$

Weights

$$W_6 = F_{-1} \cdot F$$
$$W_6 = 100$$
$$B_6 = 1$$
$$P6 = W_6 + B_6$$
$$P6 = 101$$

Total number of learning parameters

$$P = P_1 + P_2 + P_3 + P_4 + P_5 + P_6$$
$$P = 21.423.001$$

15 Problem-15

A.

The alternative approach of reading a k + Δ wide strip and computing a Δ-wide output strip is preferable for several reasons, and it can lead to more efficient convolution operations. This approach is often known as "vectorized convolution" or "strip mining.
Advantages of Alternative Approach:

- Memory Access Pattern
  - The alternative approach increases the data reuse by reading a larger strip of input data (k + Δ wide) at once. This can take better advantage of memory hierarchy and cache, reducing the number of memory accesses compared to reading k-wide strips multiple times.
- Parallelization
  - The wider strip allows for more opportunities for parallelism since you can perform computations for multiple output values simultaneously. This can be beneficial for optimizing computations on modern hardware, like GPUs or parallel CPU architectures.
- Reduced Overhead
  - Reading a larger strip of input data with fewer iterations reduces loop overhead and branching, which can lead to improved performance.

However, there is a limit to how large you should choose Δ, and this depends on factors such as memory constraints, cache size, and hardware architecture. Increasing Δ too much can lead to increased memory usage and cache thrashing,

negating the benefits of the approach. A larger Δ also requires more computation to process the wider strip, which might introduce additional overhead.
It's important to find a balance between the benefits of increased data reuse and parallelism and the potential drawbacks of increased memory usage and computation. Profiling and experimenting with different values of Δ on the specific hardware you're targeting can help you determine an optimal choice for Δ

B.
In the table below, we can verify the theoretical approach we presented in part A. As we can observe, for different Δ there is a significant reduce in time when we are using the wider stripe type. In the Code folder there is code that describes the experiments.

Kernel Size	**Strip Type**	**Execution Time (s)**
3x3	Horizontal	0.4259
3x3	Wider	0.1078
7x7	Horizontal	0.3464
7x7	Wider	0.0511
11x11	Horizontal	0.3168
11x11	Wider	0.0348

Table 1: Execution times for different kernel sizes and strip types