



Machine learning techniques for design and optimization of communication systems, Winter 2025

Linear and nonlinear equalization

Professor: *Darko Zibar*, dazi@dtu.dk

Student: *Enrico Leonardi*, *s222721*

January 24, 2025

Assignment 4

The entire assignment has been addressed resorting on 4 Python scripts uploaded within [Appendix A](#).

1 Linear discrete-time communication system model

The objective of this exercise is to implement linear discrete-time communication system model from the lecture slides, and show that the linear adaptive linear equalizer is effective in combating distortions induced by the channel. Linear discrete-time communication systems and the linear adaptive equalizer are shown on slides 29-31.

The generated input data is represented by [Figure 1](#).

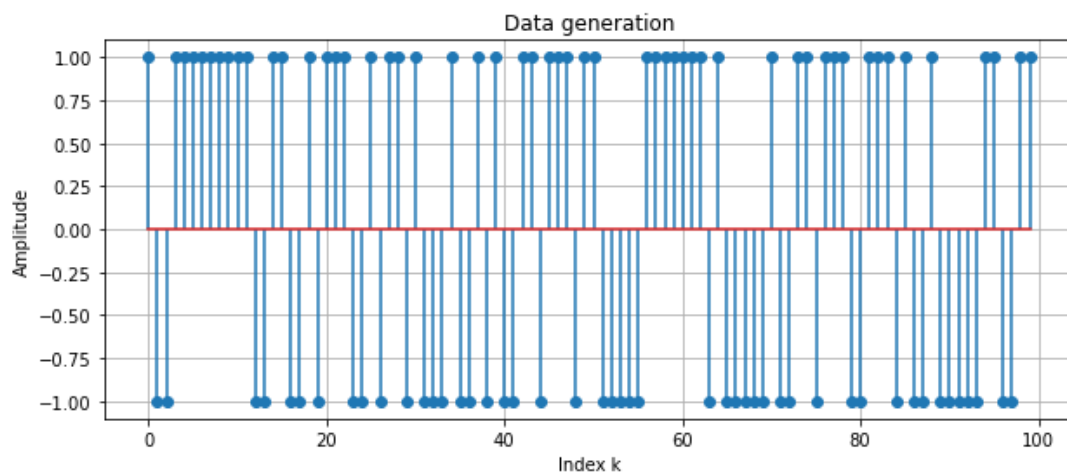


Figure 1: Generated input data

By convolving the input $x[k]$ with the channel's impulse response $h[k]$ we obtain the output of the channel (see [Figure 2](#)).

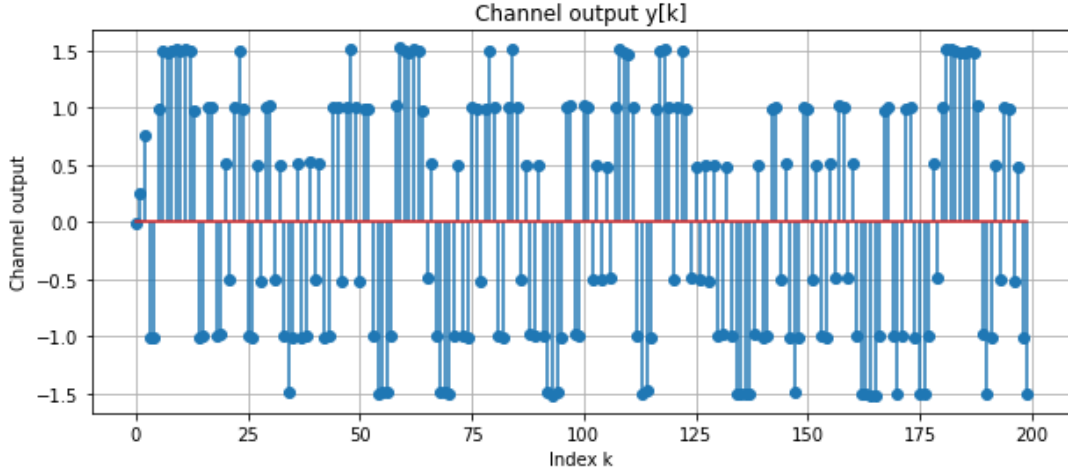


Figure 2: Linear channel output

By implementing and applying the linear adaptive equalizer with $M = 11$ taps (gradient descent based learning), we can compensate for the distortions introduced by the linear channel (see Figure 3). This process will introduce some amount of delay D we will take care to calculate using cross-correlation on later stages.

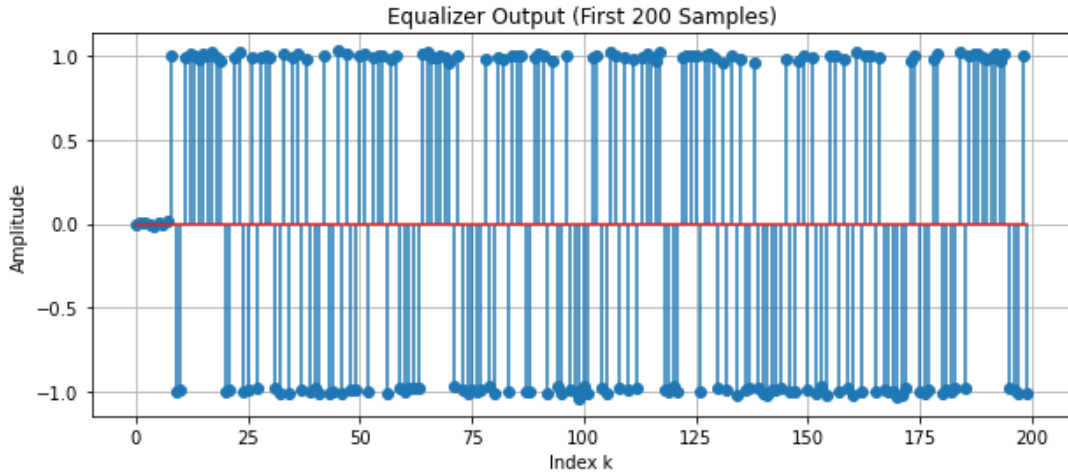


Figure 3: Linear equalizer output

The error signal used to update the linear equalizer weights is defined as the difference between the equalizer output and the symbols. If we plot it as a function of number of iterations we obtain Figure 4.



Figure 4: Linear equalizer output

We can see that it took ≈ 100 iterations for the equalizer to converge and learn its weights.

By taking into account the delay D , we want now to count the number of errors between the equalized signal and the original data sequence $x[k]$. For clarity, a comparison between the two plots has been provided within Figure 5. N.B. the equalized signal has not been quantized, but it will be in the following exercises.

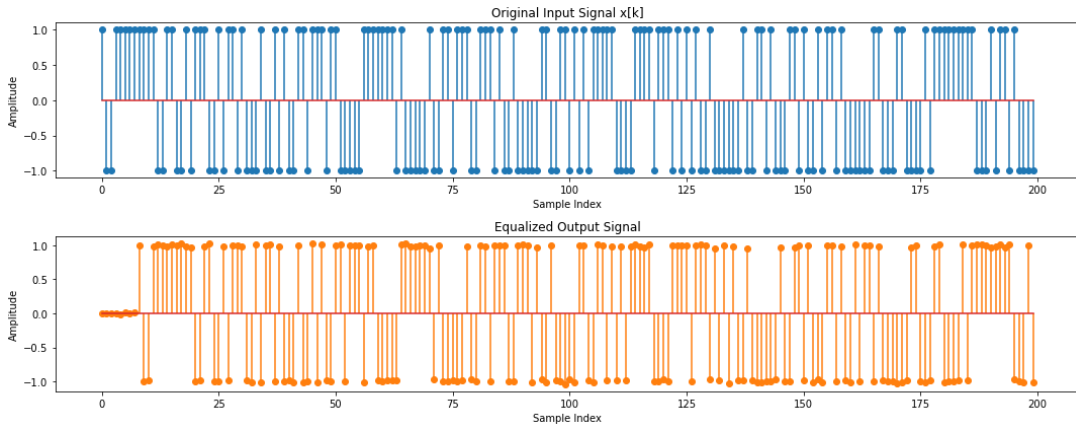


Figure 5: Input sequence vs Equalized sequence

Finally, by resorting on cross-correlation to detect the number of delayed samples between the two sequences, we can properly count the number of errors and compute the error rate between equalized and original data sequences.

Detected delay: 8
Number of Errors: 0
Error Rate: 0.00%

For further exploration, setting $M = 3.5$ and $\mu = 0.01$ affects the total amount of errors as follows.

Detected delay: 8
Number of Errors: 2
Error Rate: 0.02%

2 Single-layer neural network trained by gradient descent

In this exercise, we would like to demonstrate the capabilities of a single-layer neural network, trained by gradient descent (backpropagation), to model various functions. Functions:

1. $f(x) = x^2$
2. $f(x) = x^3$
3. $f(x) = \sin(x)$
4. $f(x) = |x|$

In synthesis, we have to implement a single hidden layer neural network, including biases, with one input and one output.

To implement the training of the neural network, use the approach based the gradient descent described in the slides. In the same plot, we show the training data (our functions) and the modeling capabilities of the neural network implemented on the test data.

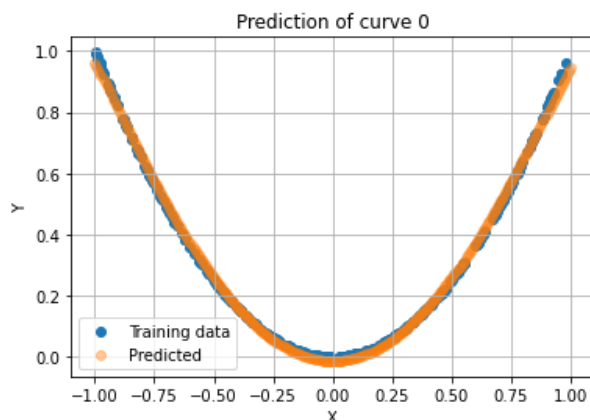


Figure 6: Prediction of training data $f(x) = x^2$

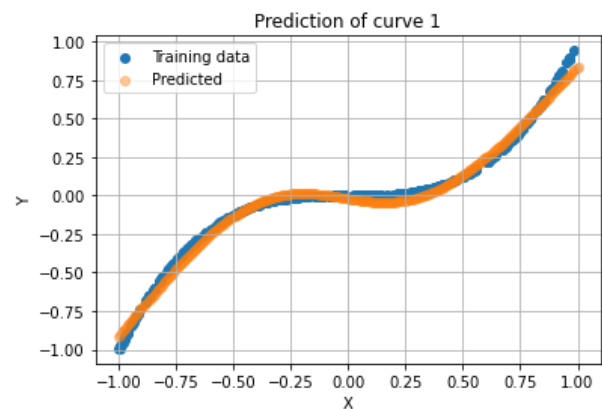


Figure 7: Prediction of training data $f(x) = x^3$

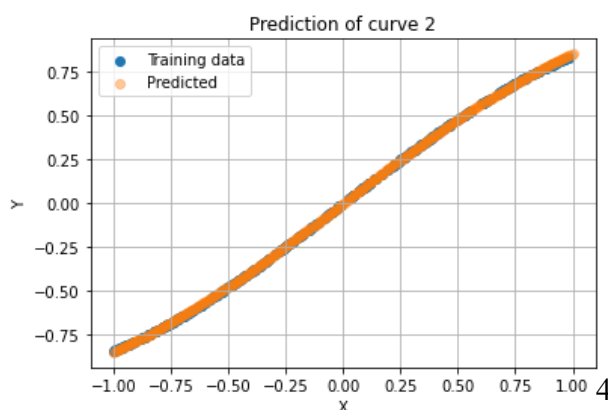


Figure 8: Prediction of training data $f(x) = \sin(x)$

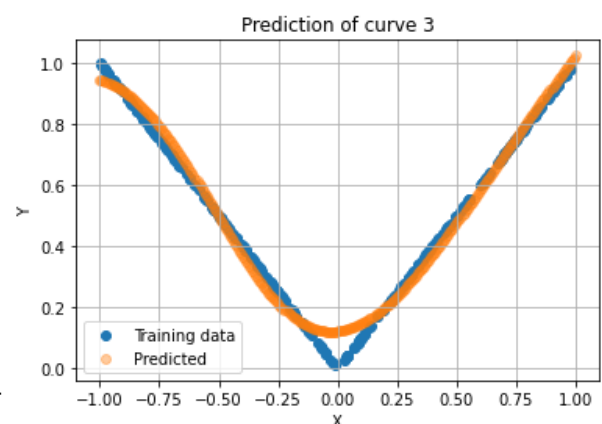


Figure 9: Prediction of training data $f(x) = |x|$

By checking the code in the Appendix, we can see a single-layer neural network with 15 nodes has been implemented to achieve the predictions displayed in Figures 6, 7, 8, 9. Unfortunately, better results were not achievable with a smaller number of nodes, being the functions too complex to be captured. In Figure 9 we can still see how the NN struggles in predicting data points of the function for $x \rightarrow 0$, being it a non-derivable area. To double check the effectiveness of the predictions, let's plot the mean square error, MSE, as a function of number of iterations.

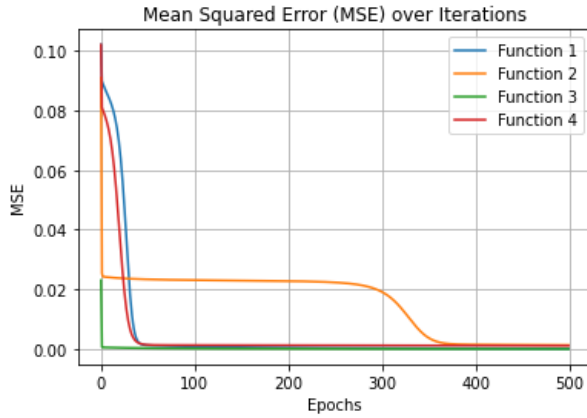


Figure 10: MSE per iteration

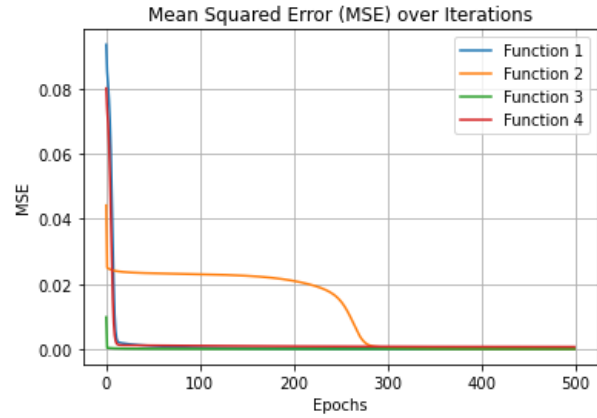


Figure 11: MSE with increased data points

Of course, the number of nodes is not the only hyper-parameter whose tuning would affect the quality of the predictions. Indeed, increasing the database size N would achieve very good predictions: the weights converge faster with more data points (see Figures 10 and 11). On the other hand, decreasing the number of nodes or an improper learning rate choice μ won't guarantee weight convergence for all the functions (see Figures 12 and 13).

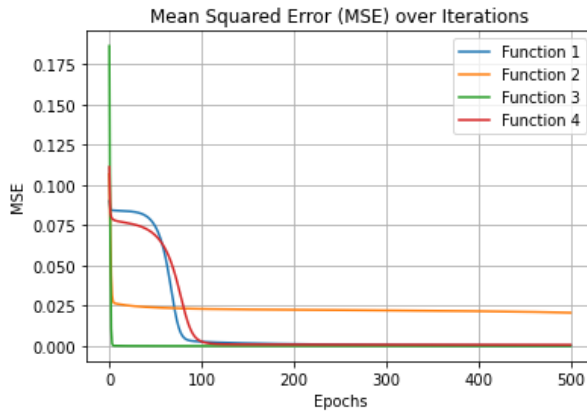


Figure 12: MSE with a 2 nodes NN

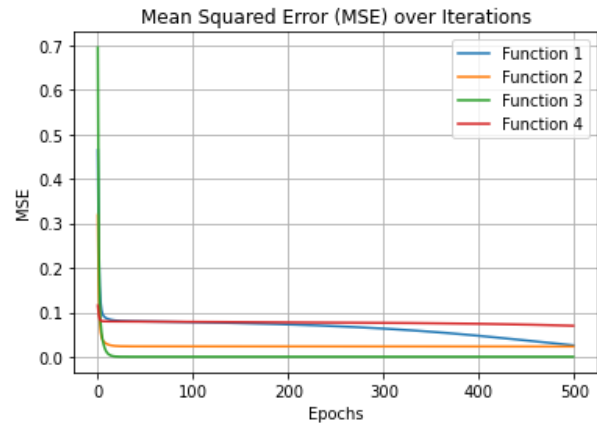


Figure 13: MSE with $\mu = 0.0005$

3 Nonlinear discrete-time communication channel

The objective of this exercise is to implement the nonlinear discrete-time communication channel with the nonlinear equalizer from the lecture slide 43. We shall demonstrate that the nonlinear equalizer, implemented as a multiple input single layer neural network, is effective in

combating distortions induced by the nonlinear channel.

Analogously to Exercise 1, we display how input data (Figure 14) and "channeled" data (Figure 15) look like.

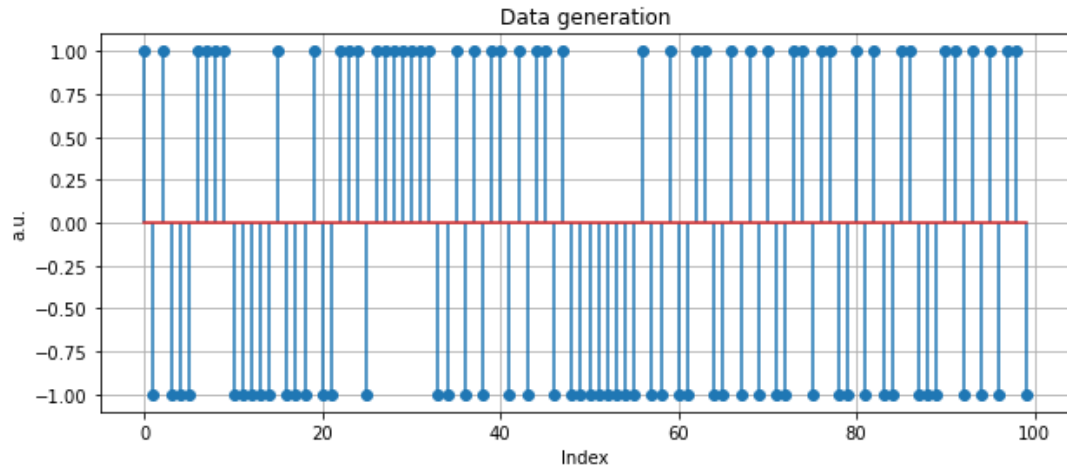


Figure 14: Generated input data $x[k]$

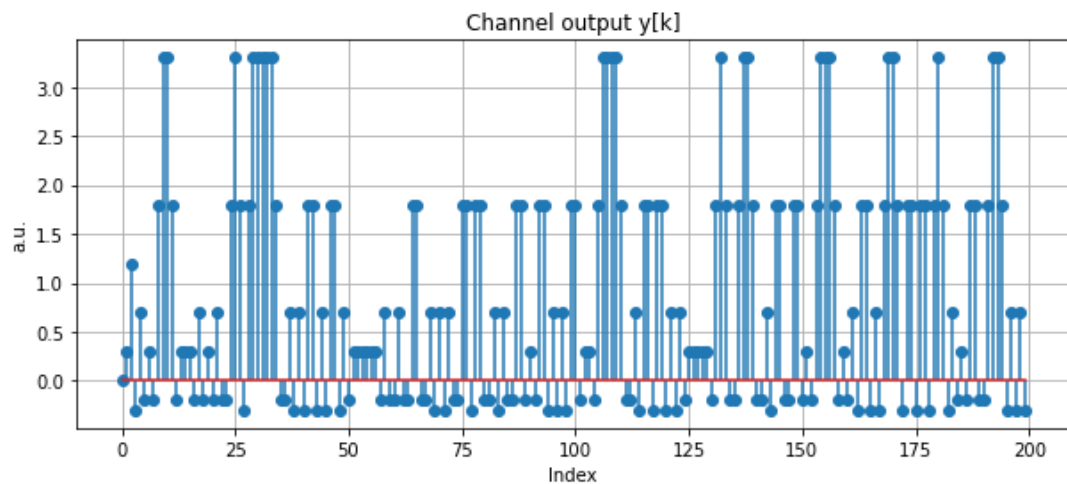


Figure 15: Nonlinear channel output

By implementing the nonlinear adaptive equalizer, based on the multiple input neural network, with $M = 11$, using the approach based the gradient descent having the error signal used to update the linear equalizer weights defined as the difference between the equalizer output and the symbols:

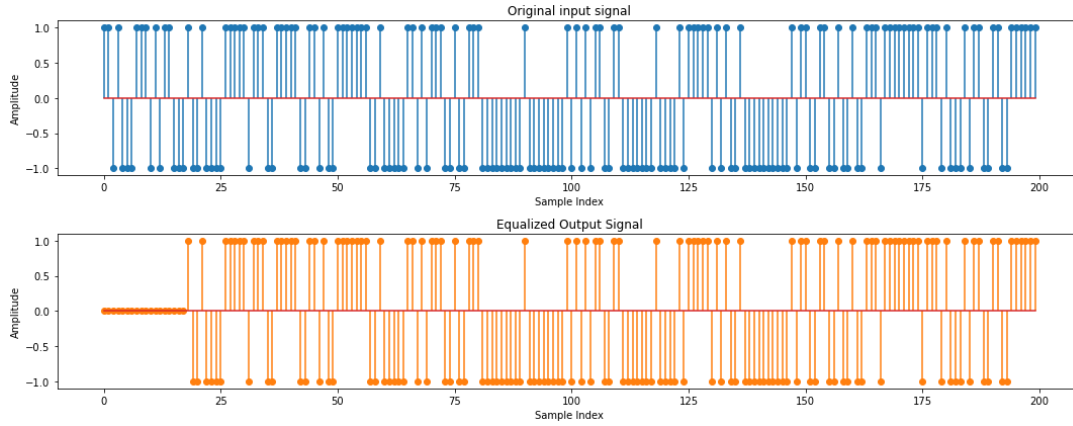


Figure 16: Input sequence vs Nonlinearly-equalized output

We can compare the original input data sequence with the (quantized) equalized signal (see [Figure 16](#)). We can see that an amount of delay D is being introduced both by the channel and the equalizer, and accounting for it using cross-correlation we are able to calculate the number of errors between the original signal and the equalized one.

Number of Errors: 117
Error Rate: 0.23%

Employing the previous-case linear adaptive equalizer to compensate for the distortion of the current-case nonlinear channel would result in worst results (in other words, a higher number of more errors). Once again, the performance is dependent to a number of hyper-parameters, including the number of hidden nodes, the initialization of the network and the learning rate μ . Weight initialization is particularly important in avoiding weight convergence toward local minima, which is the reason why each iteration initializes the weight vectors with different noise seed.

4 Convolutional nonlinear equalizer

This time the goal is implementing the convolutional nonlinear equalizer from slide 45, and evaluate its performance on the nonlinear channel specified by the parameters from Exercise 3.

The idea is to combine the approaches adopted in Exercise 1 (linear adaptive equalizer first) and Exercise 3 (nonlinear adaptive equalizer secondly). Generated input data ([Figure 17](#)) and nonlinear channeled ([Figure 18](#)) output look like

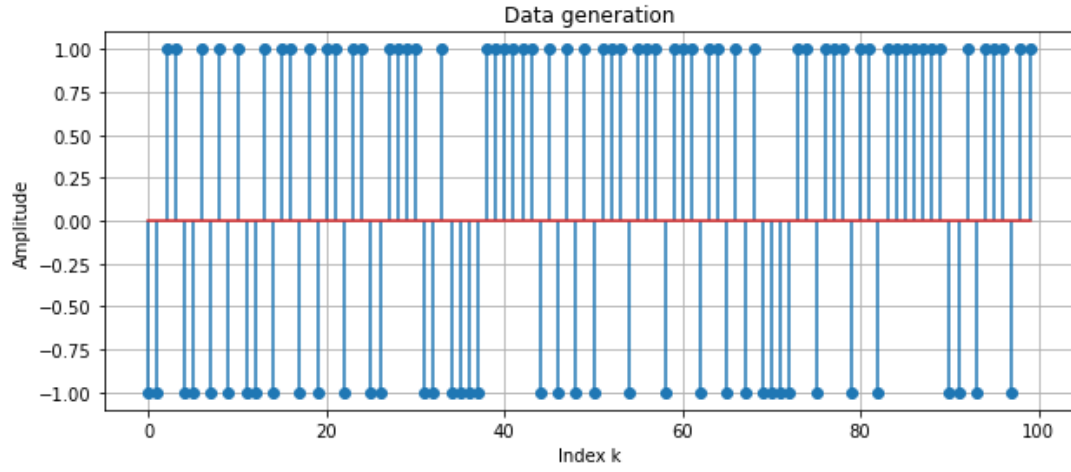


Figure 17: Generated input data $x[k]$

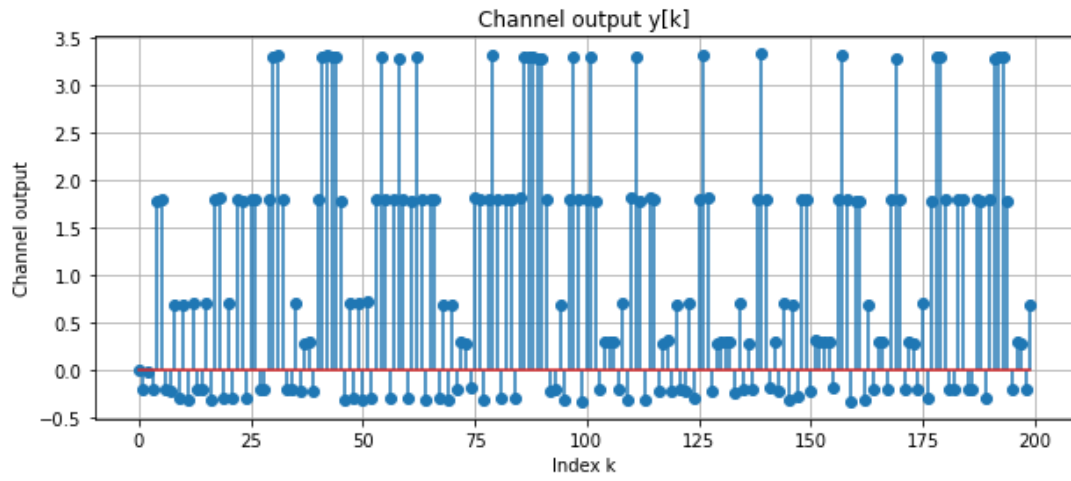


Figure 18: Nonlinear channel output

By resorting on the adaptive linear equalizer, we obtain a first (quantized) attempt of equalization of the distorted signal in [Figure 19](#).

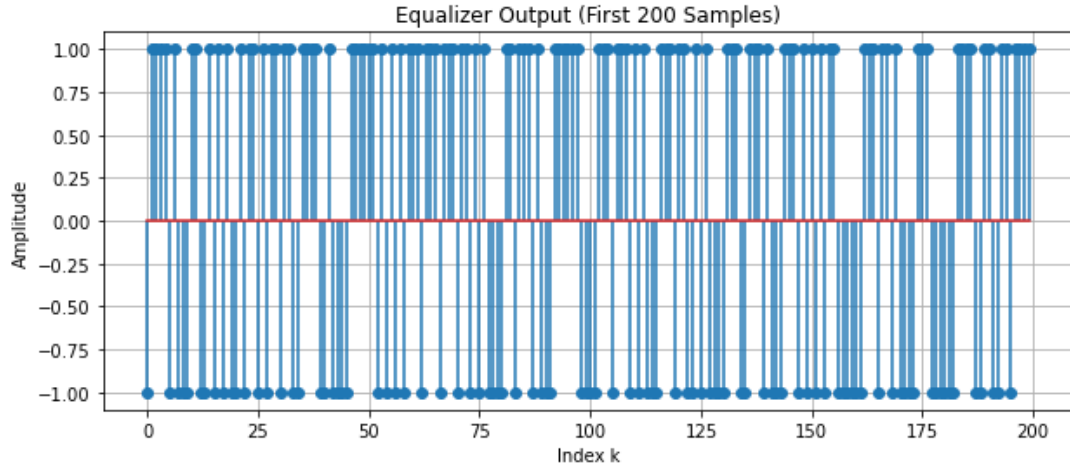


Figure 19: Linearly-equalized quantized signal

To ease its comparison with the original input data, the signals can be displayed closely attached as in [Figure 20](#).

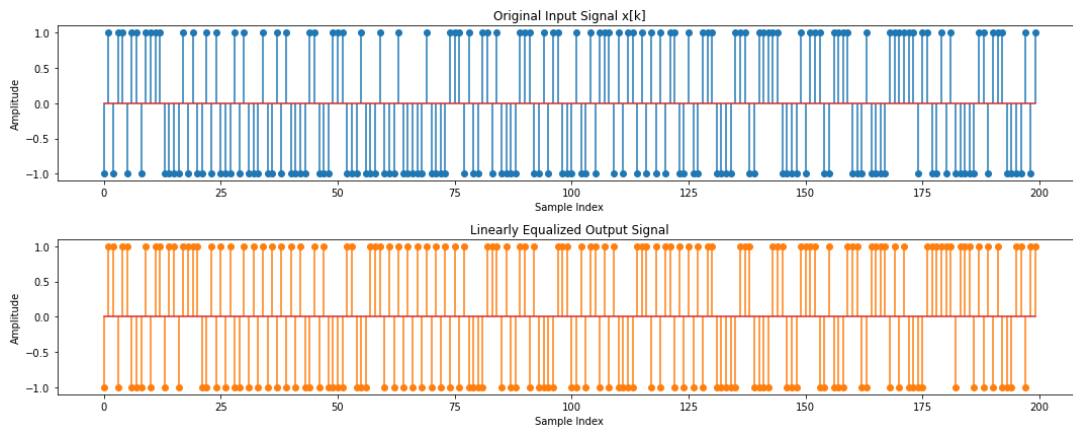


Figure 20: Input data vs Linearly-equalized signal

Which, as expected, results in a conspicuous amount of errors, even after achieving weight convergence (see [Figure 22](#)).

Detected delay: 8
 Number of Errors: 4280
 Error Rate: 8.56%

However, if we sequentially input the output of the linear equalizer into the single-layer NN-based nonlinear equalizer, we obtain a (delayed) equalized signal which closely resembles the original (see [Figure 21](#)).

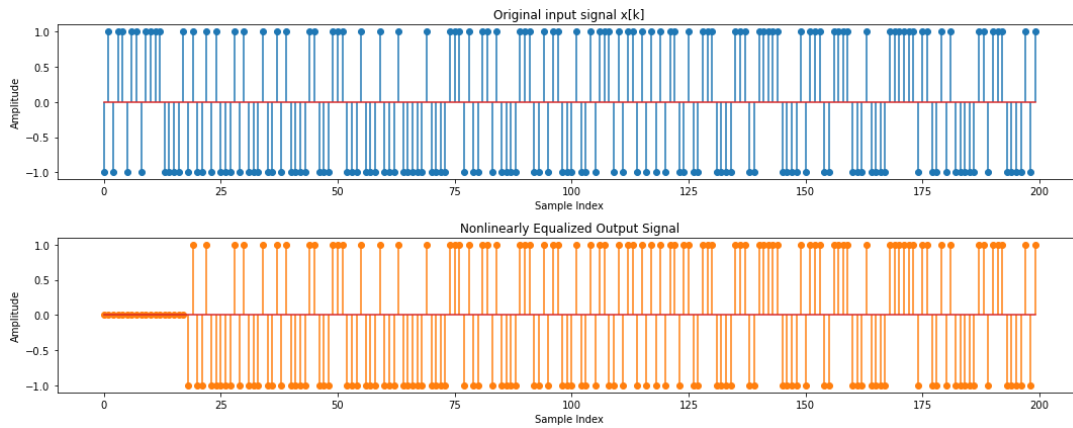


Figure 21: Input data vs Nonlinearly-equalized signal

Similarly to the third exercise, by taking into account the delay of the entire chain of devices, we can calculate the number of errors, which is now relatively small.

Number of Errors: 178

Error Rate: 0.36%

As a good practice, and evaluate weight convergence happening within both the linear and nonlinear case, we can examine the squared error, as done in Exercise 1. The squared error resulting from the linear and nonlinear equalizers can be inspected respectively within Figures 22 and 23.

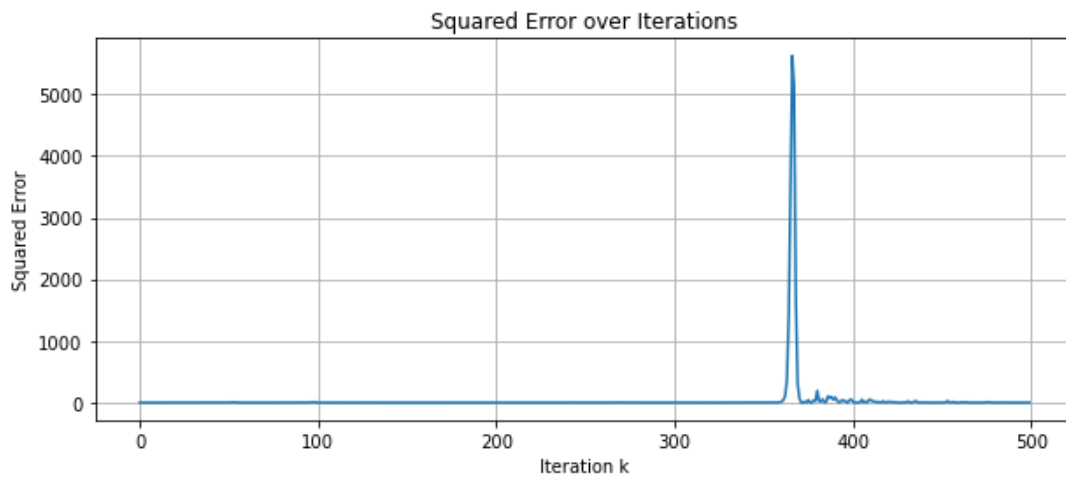


Figure 22: Squared error linear equalizer

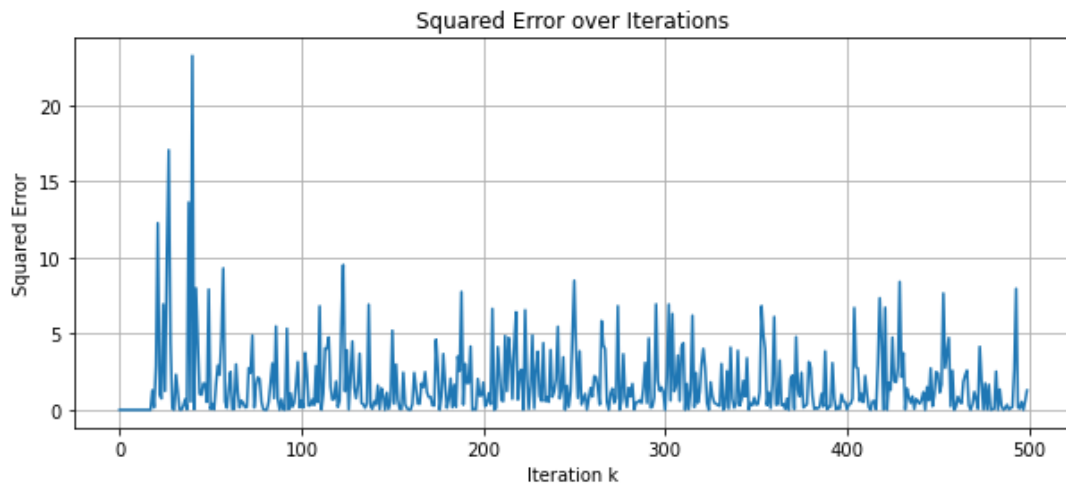


Figure 23: Squared error nonlinear equalizer

Figure 23 clearly suggests that the nonlinear equalizer can be tuned better to achieve an even smaller amount of errors.

A Appendix

```

1 # ASSIGNMENT 4 EXERCISE 1
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 # Exercise 3.1.1
6 # Implement linear discrete-time communication system model
7
8 L = 10000 # number of samples
9 x = np.random.choice([1, -1], size=L) # random sequence of +1 and -1
10
11 # Plot the data sequence
12 plt.figure(figsize=(10, 4))
13 plt.stem(x[:100]) # plotting only the first 100
14 plt.title('Data generation')
15 plt.xlabel('Index k')
16 plt.ylabel('Amplitude')
17 plt.grid(True)
18 plt.show()
19
20 # Exercise 3.1.2
21 # Use the convolution to obtain the output of the channel
22 W = 3
23 sigma_n = 10e-3
24
25 # channel impulse response
26 h = np.array([0.5 * (1 + np.cos(2 * np.pi * (k - 2) / W)) if k in [1, 2, 3]
27               else 0 for k in range(0, 5)])
28 n = np.random.normal(0, sigma_n, L) # Gaussian noise n
29 # Convolve the data sequence x[k] with the channel impulse response h[k]
30 y = np.convolve(x, h, mode='full')[:L] + n

```

```

30
31 # Plot the output of the channel
32 plt.figure(figsize=(10, 4))
33 plt.stem(y[:200])
34 plt.title('Channel output y[k]')
35 plt.xlabel('Index k')
36 plt.ylabel('Channel output')
37 plt.grid(True)
38 plt.show()
39
40 # Exercise 3.1.3
41 # Implement the linear adaptive equalizer
42
43 # Parameters
44 M = 11 # Number of taps in the equalizer
45 mu = 0.075 # Learning rate for gradient descent
46 D = 7 # Delay in samples to align the equalizer output with the desired
    signal
47 w = np.zeros(M)
48 errors = np.zeros(L)
49
50 # Output initialization
51 x_hat = np.zeros(L)
52
53 # The input signal [ ] should be delayed by
54 # samples because the channel and equalizer
55 # will take some time to respond. The equalizer
56 # should predict  $\hat{x}[k]$  based on a delayed version of
57 # [ ] and the current weights.
58 # This implies that you need to compute the error
59 # signal between the predicted output  $\hat{x}[k]$  (from
60 # the equalizer) and the actual signal delayed by samples.
61
62 for k in range(M + D, L): # Implementing slide 32 update algorithm
63     y_window = y[k - M:k] # Select M recent samples from y
64     x_hat[k] = w.T @ y_window
65     errors[k] = x[k - D] - x_hat[k] # Error between delayed input and
    predicted output
66     w += mu * errors[k] * y_window
67
68 # Convolve the channel output with the equalizer weights
69 eq_output = np.convolve(y, w, mode='full')[:L]
70
71 # Plot the equalizer output
72 plt.figure(figsize=(10, 4))
73 plt.stem(eq_output[:200])
74 plt.title('Equalizer Output (First 200 Samples)')
75 plt.xlabel('Index k')
76 plt.ylabel('Amplitude')
77 plt.grid(True)
78 plt.show()
79
80 # Exercise 3.1.4
81 # Plot the error squared as a function of number of iterations.
82

```

```

83 # Plot the squared errors
84 plt.figure(figsize=(10, 4))
85 plt.plot(errors[:500] ** 2)
86 plt.title('Squared Error over Iterations')
87 plt.xlabel('Iteration k')
88 plt.ylabel('Squared Error')
89 plt.grid(True)
90 plt.show()
91
92 # Exercise 3.1.5
93 # Once the equalizer has converged and the weights of the equalizer has
    been learned,
94 # show that the distorted signal after the channel output can be equalized
95
96 # Plot the original input and the equalized output
97 plt.figure(figsize=(15, 6))
98
99 # Plot original input signal
100 plt.subplot(2, 1, 1) # two rows, one column, first subplot
101 plt.stem(x[:200], linefmt='C0-', markerfmt='C0o')
102 plt.title('Original Input Signal x[k]')
103 plt.xlabel('Sample Index')
104 plt.ylabel('Amplitude')
105
106 # Plot equalized output signal
107 plt.subplot(2, 1, 2) # two rows, one column, second subplot
108 plt.stem(eq_output[:200], linefmt='C1-', markerfmt='C1o')
109 plt.title('Equalized Output Signal')
110 plt.xlabel('Sample Index')
111 plt.ylabel('Amplitude')
112
113 plt.tight_layout()
114 plt.show()
115
116 # Exercise 3.1.6
117 # Compute the number of errors between the equalized signal and the
    original data sequence x[k]
118 cross_corr = np.correlate(x, eq_output, mode='full')
119 # Find the index of the maximum correlation value
120 delay = np.argmax(cross_corr) - (len(x) - 1)
121
122 print(f'Detected delay: {-delay}')
123
124 # Calculate the error signal
125 error_signal = x[:len(x)+delay] - eq_output[-delay:]
126
127 # Count the number of mismatches (errors)
128 number_of_errors = np.sum(np.abs(error_signal) >= 0.1)
129
130 # Calculate the error rate
131 error_rate = number_of_errors / len(error_signal)
132
133 print(f"Number of Errors: {number_of_errors}")
134 print(f"Error Rate: {error_rate * 100:.2f}%")

```

1 # ASSIGNMENT 4 EXERCISE 2

```

2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 # Initialize
6 sigma2 = 0.09
7 N = 300
8 n_nodes = 15
9 mu = 0.01 # Learning rate for gradient descent
10
11 x_train = np.random.uniform(-1, 1, N)
12
13 # Define the functions
14 y1 = x_train ** 2
15 y2 = x_train ** 3
16 y3 = np.sin(x_train)
17 y4 = np.abs(x_train)
18
19 # Stack them into a single 2D array
20 Y = np.column_stack((y1, y2, y3, y4)) # each column is a different function
21
22 mse_history = []
23
24 for i in range(Y.shape[1]):
25     Y_train = Y[:, i]
26     Y_train = Y_train[:, np.newaxis]
27     # Add bias column to X_train
28     X_train = np.column_stack((x_train, np.ones((len(Y_train), 1))))
29
30     # Initialize weights
31     W1 = np.random.normal(0, np.sqrt(sigma2), (2, n_nodes))
32     W2 = np.random.normal(0, np.sqrt(sigma2), (n_nodes + 1, 1))
33
34     # Track the MSE for the current function
35     mse_per_iteration = []
36
37     for epoch in range(500):
38         y_pred = []
39         for k in range(len(X_train[:,0])):
40
41             # Forward pass to hidden layer
42             A = X_train[k,:] @ W1 # (n_nodes,)
43
44             # Apply activation function np.tanh() and add bias column
45             Z = np.append(np.tanh(A), 1) # (n_nodes + 1, 1)
46
47             # Backward pass: Compute the gradient and update the weights
48             # Gradient of the error with respect to W2
49             dE_dW2 = Z * (Z @ W2 - Y_train[k,:]) # (n_nodes + 1, 1)
50             W2 -= mu * dE_dW2[:, np.newaxis] # Update rule for W2 (
n_nodes + 1, 1)
51
52             # Gradient of the error with respect to W1
53             dE_dW1 = X_train[k,:] * ((1 - np.tanh(A)**2)[:, np.newaxis] * (
Z @ W2 - Y_train[k,:]) * W2[0:-1]) # (n_nodes, 2)
54             W1 -= mu * dE_dW1.T # (2, n_nodes)

```

```

55         # Compute predictions from output layer
56         y_pred.append(Z @ W2) # (1, )
57         # Calculate MSE for this iteration
58         mse = np.mean((y_pred - Y_train) ** 2)
59         mse_per_iteration.append(mse)
60
61     # Store the MSE for the current function
62     mse_history.append(mse_per_iteration)
63
64     # After training, use the final updated weights to predict. TESTING
65     x_test = np.linspace(-1, 1, N)
66     X_test = np.column_stack((x_test, np.ones((len(Y_train), 1))))
67     A_final = X_test @ W1
68     Z_final = np.column_stack((np.tanh(A_final), np.ones((len(A_final), 1))
69     ))
70     y_pred_final = Z_final @ W2
71
72     # Plot
73     plt.figure()
74     plt.scatter(x_train, Y_train, label='Training data')
75     plt.scatter(x_test, y_pred_final, label='Predicted', alpha=0.4)
76     plt.title(f"Prediction of curve {i}")
77     plt.xlabel("X")
78     plt.ylabel("Y")
79     plt.grid()
80     plt.legend()
81     plt.show()
82
83 # Plot MSE history for each function
84 plt.figure()
85 for idx, mse in enumerate(mse_history):
86     plt.plot(mse, label=f'Function {idx+1}')
87 plt.title("Mean Squared Error (MSE) over Iterations")
88 plt.xlabel("Epochs")
89 plt.ylabel("MSE")
90 plt.legend()
91 plt.grid()
92 plt.show()

```



```

1 # ASSIGNMENT 4 EXERCISE 3
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 L = 50000 # number of samples
6 W = 3
7 # channel impulse response
8 h = np.array([0.5 * (1 + np.cos(2 * np.pi * (k - 2) / W)) if k in [1, 2, 3]
9               else 0 for k in range(0,5)])
9 #sigma_n = 10e-3
10 n = 0#np.random.normal(0, sigma_n, L) # Gaussian noise n
11 a = 0.8
12
13 # Exercise 3.3.1
14 # Generate the data sequence x[k].
15 x_train = np.random.choice([1, -1], size=L) # random sequence of +1 and -1

```

```

16 # Plot the data sequence
17 plt.figure(figsize=(10, 4))
18 plt.stem(x_train[:100]) # plotting only the first 100
19 plt.title('Data generation')
20 plt.xlabel('Index')
21 plt.ylabel('a.u.')
22 plt.grid(True)
23 plt.show()
24
25 # Plot the data sequence after it has passed the nonlinear channel.
26 # Obtain y[k]
27 convolved = np.convolve(x_train, h, mode='full')[:L]
28 Y_train = (convolved + a * (convolved ** 2) + n)
29
30 # Plot the output of the channel
31 plt.figure(figsize=(10, 4))
32 plt.stem(Y_train[:200])
33 plt.title('Channel output y[k]')
34 plt.xlabel('Index')
35 plt.ylabel('a.u.')
36 plt.grid(True)
37 plt.show()
38
39 # Exercise 3.3.2
40 # Implement the nonlinear adaptive equalizer
41 M = 11
42 D = 7
43 n_nodes = 100
44 sigma2 = 0.09
45 mu = 0.01 # Learning rate for gradient descent
46 w = np.zeros(M)
47 y_hat = np.zeros(L)
48 y_pred_final = np.zeros(L)
49 errors = np.zeros(L)
50
51 for epoch in range(200):
52     # Run the nonlinear equalizer several times, each time with different
    # weights initialization seed
53     W1 = np.random.normal(0, np.sqrt(sigma2), (M + 1, n_nodes))
54     W2 = np.random.normal(0, np.sqrt(sigma2), (n_nodes + 1, 1))
55     # We feed the NN with the output of the channel Y_train
56     # The goal is to perform equalization and get back
57     # An estimation of the original input signal x_train
58     for k in range(M + D, L): # Implementing slide 32 update algorithm
59         # Add bias column to Y_train
60         y_window = np.append(Y_train[:, np.newaxis][k - M:k], 1) # Select M
        # recent samples from Y_train (M + 1, 1)
61
62         # Forward pass to hidden layer
63         A = y_window.T @ W1 # (1, n_nodes)
64
65         # Apply activation function np.tanh() and add bias column
66         Z = np.append(np.tanh(A), 1) # (n_nodes + 1, 1)
67
68         # Compute predictions from output layer

```



```

69     y_hat[k] = Z @ W2 # (L, )
70
71     # Backward pass: Compute the gradient and update the weights
72     # Gradient of the error with respect to W2
73     dE_dW2 = Z * (y_hat[k] - x_train[k]) # (1, n_nodes+1)
74     W2 -= mu * dE_dW2[:, np.newaxis] # Update rule for W2 (n_nodes+1,
1)
75
76     # Gradient of the error with respect to W1
77     dE_dW1 = (y_hat[k] - x_train[k]) * (1 - np.tanh(A)**2).T[:, np.
newaxis] * W2[:-1] * y_window # (n_nodes, n_nodes + 2)
78     W1 -= mu * dE_dW1.T # (2, n_nodes)
79
80     errors[k] = y_hat[k] - x_train[k - D] # Error between delayed input
and predicted output
81
82 # After training, use the final updated weights to predict. TESTING
83 x_test = np.random.choice([1, -1], size=L) # (L,1)
84 convolved = np.convolve(x_test, h, mode='full')[:L]
85 Y_test = (convolved + a * (convolved ** 2) + n)
86 for k in range(M + D, L): # Implementing slide 32 update algorithm
87     # Add bias column to Y_test
88     y_window = np.append(Y_test[:, np.newaxis][k - M:k], 1) # Select M
recent samples from Y_est (M + 1, 1)
89
90     # Forward pass to hidden layer
91     A = y_window.T @ W1 # (1, n_nodes)
92
93     # Apply activation function np.tanh() and add bias column
94     Z = np.append(np.tanh(A), 1) # (n_nodes + 1, 1)
95
96     # Compute predictions from output layer
97     y_hat[k] = Z @ W2 # (L, )
98
99     # Backward pass: Compute the gradient and update the weights
100    # Gradient of the error with respect to W2
101    dE_dW2 = Z * (y_hat[k] - x_test[k]) # (1, n_nodes+1)
102    W2 -= mu * dE_dW2[:, np.newaxis] # Update rule for W2 (n_nodes+1, 1)
103    y_pred_final[k] = Z @ W2 # (L,)
104
105    # Gradient of the error with respect to W1
106    dE_dW1 = (y_hat[k] - x_test[k]) * (1 - np.tanh(A)**2).T[:, np.newaxis]
* W2[:-1] * y_window # (n_nodes, n_nodes + 2)
107    W1 -= mu * dE_dW1.T # (2, n_nodes)
108
109    #errors[k] = y_hat[k] - Y_train[k - D] # Error between delayed input
and predicted output
110
111 # Plot the original input and the equalized output
112 plt.figure(figsize=(15, 6))
113
114 # Plot original input signal
115 plt.subplot(2, 1, 1) # two rows, one column, first subplot
116 plt.stem(x_test[:200], linefmt='C0-', markerfmt='C0o')
117 plt.title('Original input signal')

```

```

118 plt.xlabel('Sample Index')
119 plt.ylabel('Amplitude')
120
121 y_pred_final = np.sign(y_pred_final)
122 # Plot equalized output signal
123 plt.subplot(2, 1, 2) # two rows, one column, second subplot
124 plt.stem(y_pred_final[:200], linefmt='C1-', markerfmt='C1o')
125 plt.title('Equalized Output Signal')
126 plt.xlabel('Sample Index')
127 plt.ylabel('Amplitude')
128 plt.tight_layout()
129 plt.show()
130
131 # Compute the number of errors between the equalized signal and the
    original data sequence x[k]
132 cross_corr = np.correlate(x_test, y_pred_final, mode='full')
133 # Find the index of the maximum correlation value
134 delay = np.argmax(cross_corr) - (len(x_test) - 1)
135
136 print(f'Detected delay: {-delay}')
137
138 # Calculate the error signal
139 error_signal = x_test[:len(x_test)+delay] - y_pred_final[-delay:]
140
141 # Count the number of mismatches (errors)
142 #number_of_errors = np.sum(y_pred_final != x_test)
143 number_of_errors = np.sum(np.abs(error_signal) >= 0.1)
144
145 # Calculate the error rate
146 error_rate = number_of_errors / len(error_signal)
147
148 print(f"Number of Errors: {number_of_errors}")
149 print(f"Error Rate: {error_rate * 100:.2f}%")

1 # ASSIGNMENT 4 EXERCISE 4
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 # Implement linear discrete-time communication system model
6 L = 50000 # number of samples
7 x = np.random.choice([1, -1], size=L) # random sequence of +1 and -1
8
9 # Plot the data sequence
10 plt.figure(figsize=(10, 4))
11 plt.stem(x[:100]) # plotting only the first 100
12 plt.title('Data generation')
13 plt.xlabel('Index k')
14 plt.ylabel('Amplitude')
15 plt.grid(True)
16 plt.show()
17
18 # Use the convolution to obtain the output of the channel
19 W = 3
20 a = 0.8
21 sigma_n = 10e-3
22

```

```

23 # channel impulse response
24 h = np.array([0.5 * (1 + np.cos(2 * np.pi * (k - 2) / W)) if k in [1, 2, 3]
                else 0 for k in range(0,5)])
25 n = np.random.normal(0, sigma_n, L) # Gaussian noise n
26 # Convolve the data sequence x[k] with the channel impulse response h[k]
27 convolved = np.convolve(x, h, mode='full')[:L]
28 y = (convolved + a * (convolved ** 2) + n)
29
30 # Plot the output of the channel
31 plt.figure(figsize=(10, 4))
32 plt.stem(y[:200])
33 plt.title('Channel output y[k]')
34 plt.xlabel('Index k')
35 plt.ylabel('Channel output')
36 plt.grid(True)
37 plt.show()
38
39 # ----- Implement the linear adaptive equalizer
40 # Parameters
41 M = 11 # Number of taps in the equalizer
42 mu = 0.033 # Learning rate for gradient descent
43 D = 7 # Delay in samples to align the equalizer output with the desired
        signal
44 w = np.zeros(M)
45 errors = np.zeros(L)
46
47 # Output initialization
48 x_hat = np.zeros(L)
49
50 for k in range(M + D, L): # Implementing slide 32 update algorithm
51     y_window = y[k - M:k] # Select M recent samples from y
52     x_hat[k] = w.T @ y_window
53     errors[k] = x[k - D] - x_hat[k] # Error between delayed input and
        predicted output
54     w += mu * errors[k] * y_window
55
56 # Plot the error squared as a function of number of iterations.
57 plt.figure(figsize=(10, 4))
58 plt.plot(errors[:500] ** 2)
59 plt.title('Squared Error over Iterations')
60 plt.xlabel('Iteration k')
61 plt.ylabel('Squared Error')
62 plt.grid(True)
63 plt.show()
64
65 # Convolve the channel output with the equalizer weights
66 eq_output = np.convolve(y, w, mode='full')[:L]
67 eq_output = np.sign(eq_output)
68
69 # Plot the equalizer output
70 plt.figure(figsize=(10, 4))
71 plt.stem(eq_output[:200])
72 plt.title('Equalizer Output (First 200 Samples)')
73 plt.xlabel('Index k')
74 plt.ylabel('Amplitude')

```

```

75 plt.grid(True)
76 plt.show()
77
78 # Implement the nonlinear adaptive equalizer
79 #M = 11
80 #D = 7
81 n_nodes = 100
82 sigma2 = 0.09
83 mu = 0.01 # Learning rate for gradient descent
84 y_hat = np.zeros(L)
85 y_pred_final = np.zeros(L)
86 #errors = np.zeros(L)
87
88 for epoch in range(300):
89     # Run the nonlinear equalizer several times, each time with different
    weights initialization seed
90     W1 = np.random.normal(0, np.sqrt(sigma2), (M + 1, n_nodes))
91     W2 = np.random.normal(0, np.sqrt(sigma2), (n_nodes + 1, 1))
92     # We feed the NN with the output of the channel Y_train
93     # The goal is to perform equalization and get back
94     # An estimation of the original input signal x_train
95     for k in range(M + D, L): # Implementing slide 32 update algorithm
96
97         y_window = np.append(eq_output[:, np.newaxis][k - M:k], 1) # Select
        M recent samples from Y_train (M + 1, 1)
98
99         A = y_window.T @ W1 # (1, n_nodes)
100
101         Z = np.append(np.tanh(A), 1) # (n_nodes + 1, 1)
102
103         y_hat[k] = Z @ W2 # (L, )
104
105         dE_dW2 = Z * (y_hat[k] - x[k]) # (1, n_nodes+1)
106         W2 -= mu * dE_dW2[:, np.newaxis] # Update rule for W2 (n_nodes+1,
        1)
107
108         dE_dW1 = (y_hat[k] - x[k]) * (1 - np.tanh(A)**2).T[:, np.newaxis] *
        W2[:-1] * y_window # (n_nodes, n_nodes + 2)
109         W1 -= mu * dE_dW1.T # (2, n_nodes)
110
111         errors[k] = y_hat[k] - x[k - D] # Error between delayed input and
        predicted output
112
113
114 # Plot the error squared as a function of number of iterations.
115 plt.figure(figsize=(10, 4))
116 plt.plot(errors[:500] ** 2)
117 plt.title('Squared Error over Iterations')
118 plt.xlabel('Iteration k')
119 plt.ylabel('Squared Error')
120 plt.grid(True)
121 plt.show()
122
123 # After training, use the final updated weights to predict -----
    TESTING

```

```

124 x_test = np.random.choice([1, -1], size=L) # (L,1)
125 convolved = np.convolve(x_test, h, mode='full')[:L]
126 y_test = (convolved + a * (convolved ** 2) + n)
127
128 # Convolve the channel output with the equalizer weights
129 eq_output = np.convolve(y_test, w, mode='full')[:L]
130 eq_output = np.sign(eq_output)
131
132 for k in range(M + D, L): # Implementing slide 32 update algorithm
133     # Add bias column to Y_test
134     y_window = np.append(eq_output[:, np.newaxis][k - M:k], 1) # Select M
    recent samples from Y_est (M + 1, 1)
135
136     # Forward pass to hidden layer
137     A = y_window.T @ W1 # (1, n_nodes)
138
139     # Apply activation function np.tanh() and add bias column
140     Z = np.append(np.tanh(A), 1) # (n_nodes + 1, 1)
141
142     # Compute predictions from output layer
143     y_hat[k] = Z @ W2 # (L, )
144
145     # Backward pass: Compute the gradient and update the weights
146     # Gradient of the error with respect to W2
147     dE_dW2 = Z * (y_hat[k] - x_test[k]) # (1, n_nodes+1)
148     W2 -= mu * dE_dW2[:, np.newaxis] # Update rule for W2 (n_nodes+1, 1)
149     y_pred_final[k] = Z @ W2 # (L,)
150
151     # Gradient of the error with respect to W1
152     dE_dW1 = (y_hat[k] - x_test[k]) * (1 - np.tanh(A)**2).T[:, np.newaxis]
    * W2[:-1] * y_window # (n_nodes, n_nodes + 2)
153     W1 -= mu * dE_dW1.T # (2, n_nodes)
154
155 # Once the equalizer has converged and the weights of the equalizer has
    been learned,
156 # show that the distorted signal after the channel output can be equalized
157 plt.figure(figsize=(15, 6))
158
159 # Plot original input signal
160 plt.subplot(2, 1, 1) # two rows, one column, first subplot
161 plt.stem(x_test[:200], linefmt='C0-', markerfmt='C0o')
162 plt.title('Original Input Signal x[k]')
163 plt.xlabel('Sample Index')
164 plt.ylabel('Amplitude')
165
166 # Plot equalized output signal
167 plt.subplot(2, 1, 2) # two rows, one column, second subplot
168 plt.stem(eq_output[:200], linefmt='C1-', markerfmt='C1o')
169 plt.title('Linearly Equalized Output Signal')
170 plt.xlabel('Sample Index')
171 plt.ylabel('Amplitude')
172
173 plt.tight_layout()
174 plt.show()
175

```

```

176 # Compute the number of errors between the equalized signal and the
    original data sequence x[k]
177 cross_corr = np.correlate(x_test, eq_output, mode='full')
178 # Find the index of the maximum correlation value
179 delay = np.argmax(cross_corr) - (len(x_test) - 1)
180
181 print(f'Detected delay: {-delay}')
182
183 # Calculate the error signal
184 error_signal = x_test[:len(x)+delay] - eq_output[-delay:]
185
186 # Count the number of mismatches (errors)
187 number_of_errors = np.sum(np.abs(error_signal) >= 0.1)
188
189 # Calculate the error rate
190 error_rate = number_of_errors / len(error_signal)
191
192 print(f"Number of Errors: {number_of_errors}")
193 print(f"Error Rate: {error_rate * 100:.2f}%")
194
195 # Plot the original input and the equalized output
196 plt.figure(figsize=(15, 6))
197
198 # Plot original input signal
199 plt.subplot(2, 1, 1) # two rows, one column, first subplot
200 plt.stem(x_test[:200], linefmt='C0-', markerfmt='C0o')
201 plt.title('Original input signal x[k]')
202 plt.xlabel('Sample Index')
203 plt.ylabel('Amplitude')
204
205 y_pred_final = np.sign(y_pred_final)
206 # Plot equalized output signal
207 plt.subplot(2, 1, 2) # two rows, one column, second subplot
208 plt.stem(y_pred_final[:200], linefmt='C1-', markerfmt='C1o')
209 plt.title('Nonlinearly Equalized Output Signal')
210 plt.xlabel('Sample Index')
211 plt.ylabel('Amplitude')
212 plt.tight_layout()
213 plt.show()
214
215 # Compute the number of errors between the equalized signal and the
    original data sequence x[k]
216 cross_corr = np.correlate(x_test, y_pred_final, mode='full')
217 # Find the index of the maximum correlation value
218 delay = np.argmax(cross_corr) - (len(x_test) - 1)
219 print(f'Detected delay: {-delay}')
220
221 # Calculate the error signal
222 error_signal = x_test[:len(x_test)+delay] - y_pred_final[-delay:]
223
224 # Count the number of mismatches (errors)
225 #number_of_errors = np.sum(y_pred_final != x_test)
226 number_of_errors = np.sum(np.abs(error_signal) >= 0.1)
227
228 # Calculate the error rate

```

```
229 error_rate = number_of_errors / len(error_signal)
230 print(f"Number of Errors: {number_of_errors}")
231 print(f"Error Rate: {error_rate * 100:.2f}%")
```