# 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.

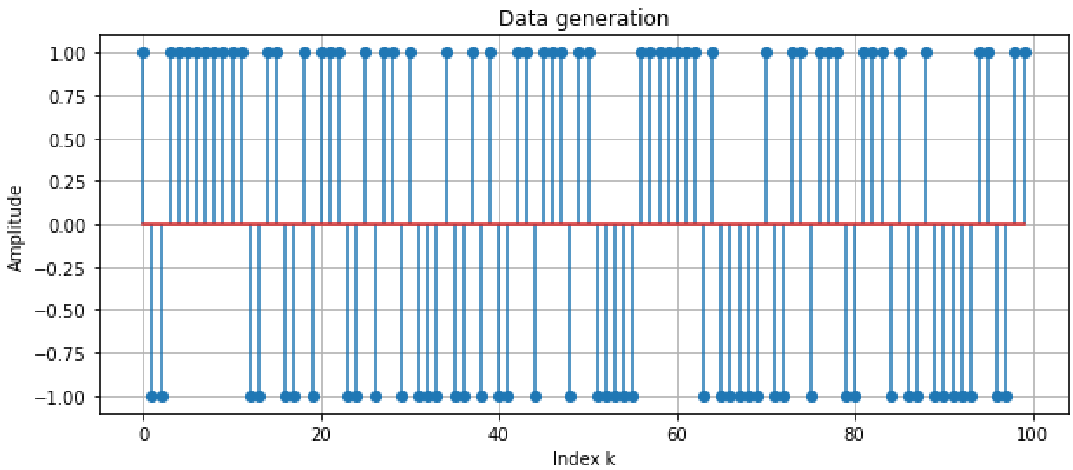


**Figure 1:** Generated input data

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

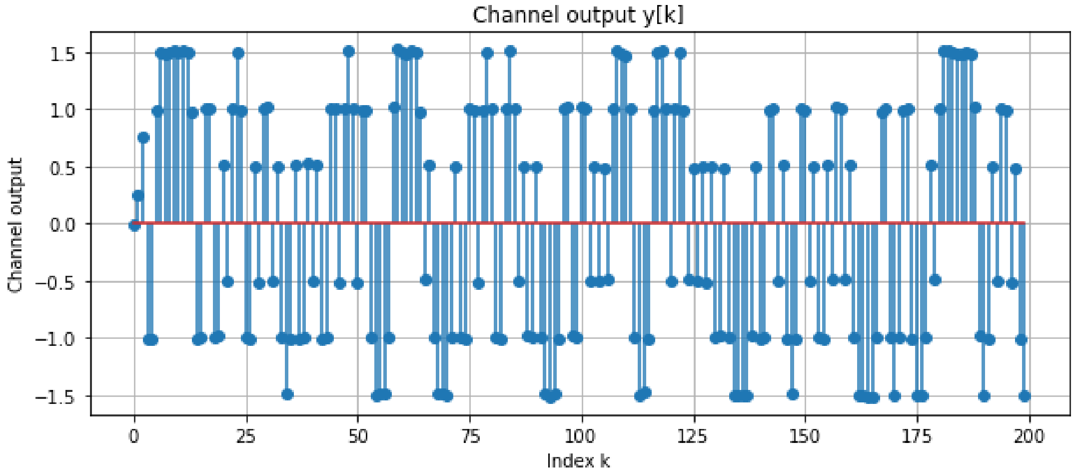


**Figure 2:** Linear channel output

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.

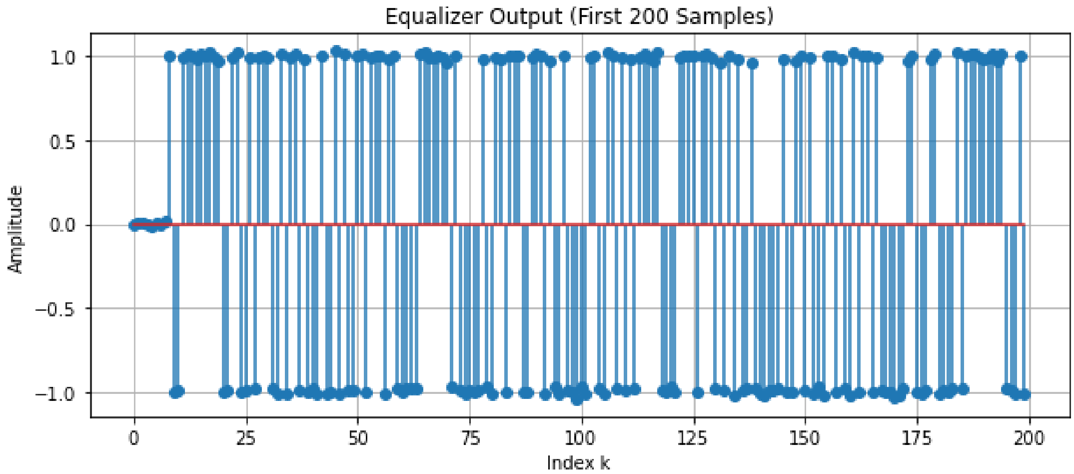


**Figure 3:** Linear equalizer output

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.

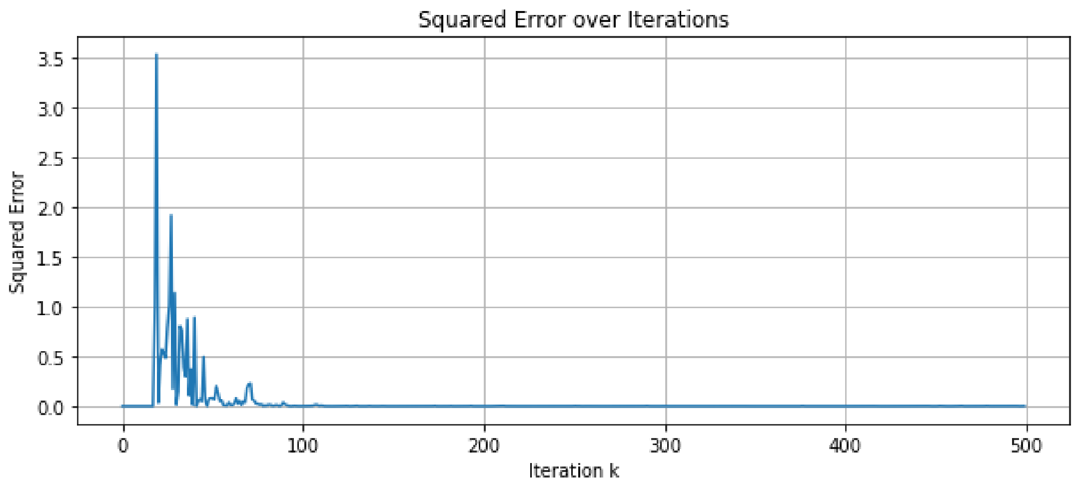


**Figure 4:** Linear equalizer output

We can see that it took $\approx 100$ 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.

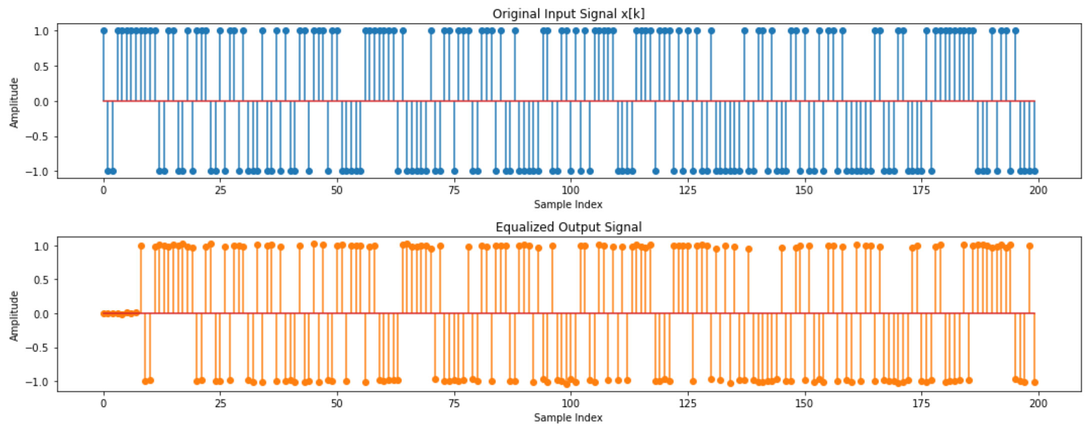


**Figure 5:** Input sequence vs Equalized sequence

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.

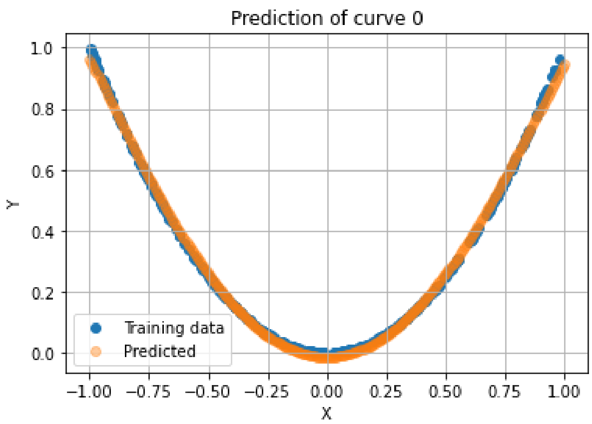


**Figure 6:** Prediction of training data $f(x) = x^2$

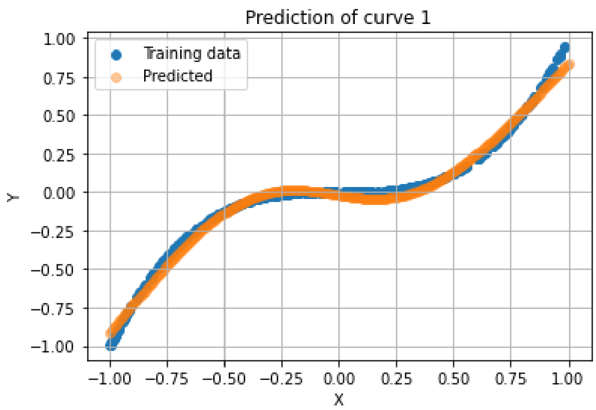


**Figure 7:** Prediction of training data $f(x) = x^3$

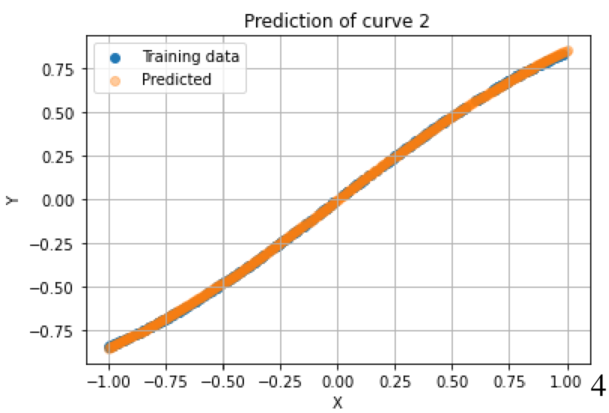


**Figure 8:** Prediction of training data $f(x) = sin(x)$

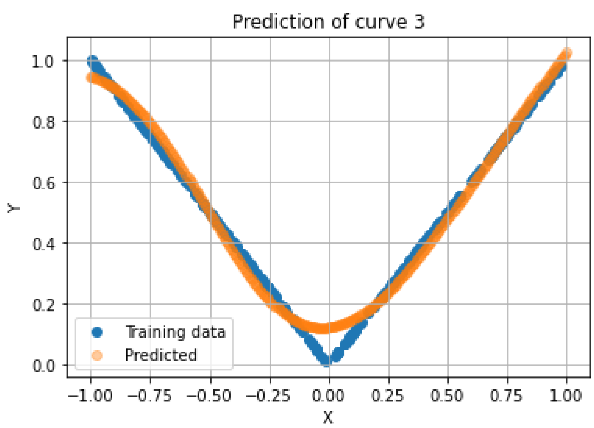


**Figure 9:** Prediction of training data $f(x) = |x|$

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 $x \to 0$, 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.

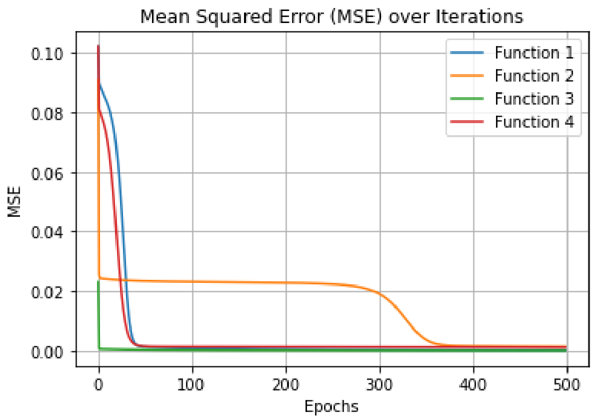


**Figure 10:** MSE per iteration

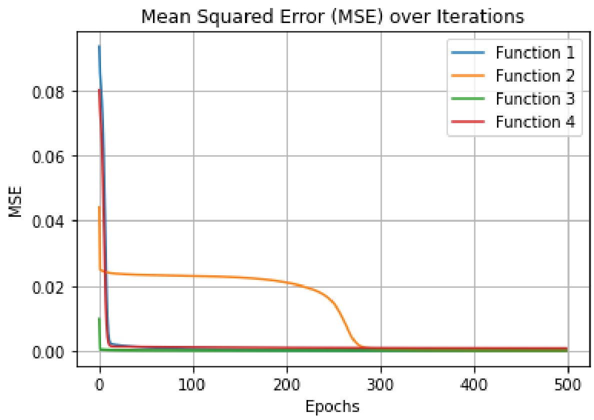


**Figure 11:** MSE with increased data points

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 $\mu$ won't guarantee weight convergence for all the functions (see Figures 12 and 13).

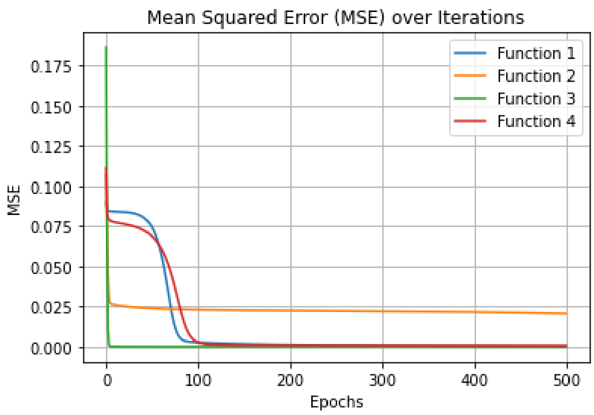


**Figure 12:** MSE with a 2 nodes NN

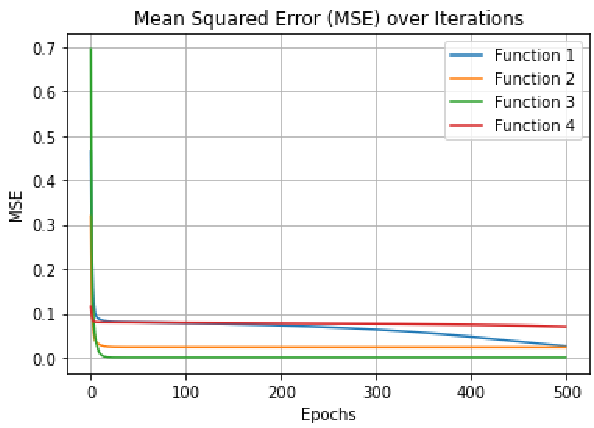


**Figure 13:** MSE with $\mu = 0.0005$

## 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.

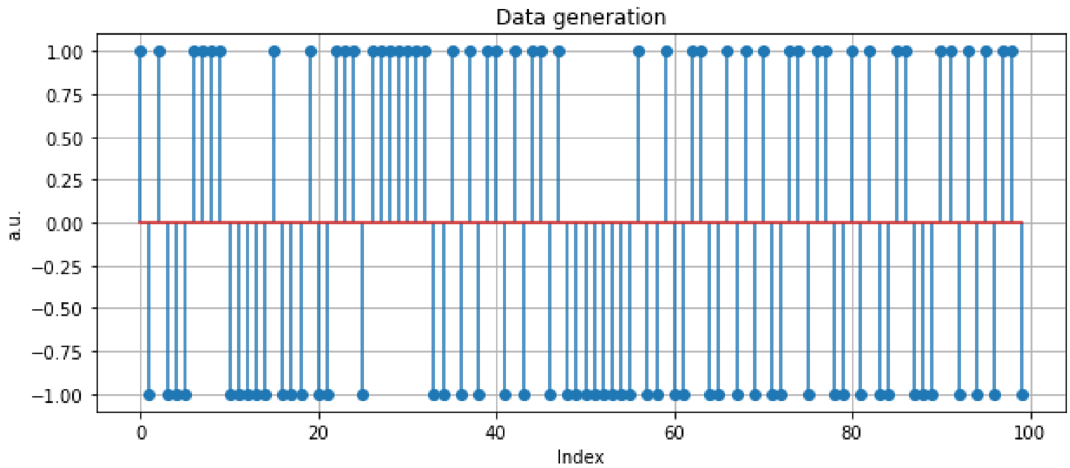


**Figure 14:** Generated input data $x[k]$

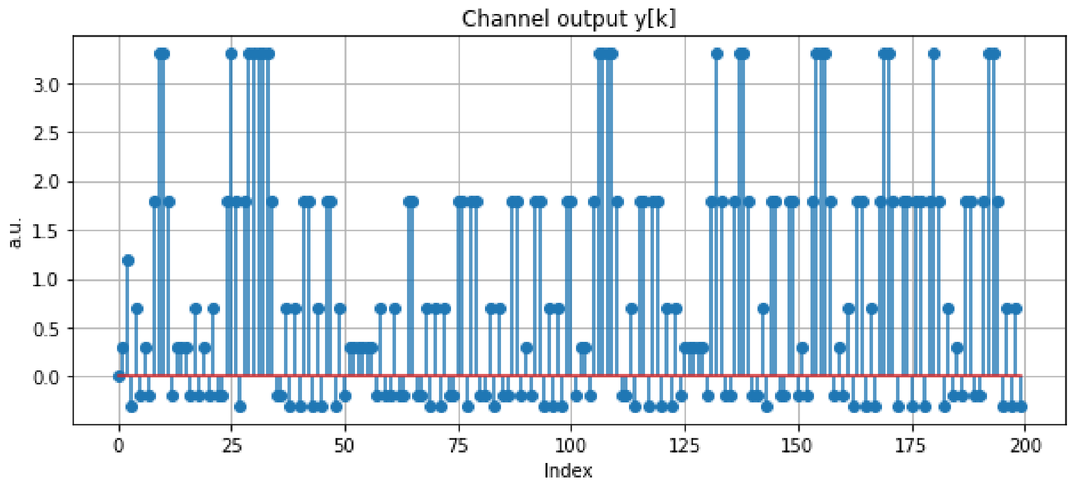


**Figure 15:** Nonlinear channel output

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:

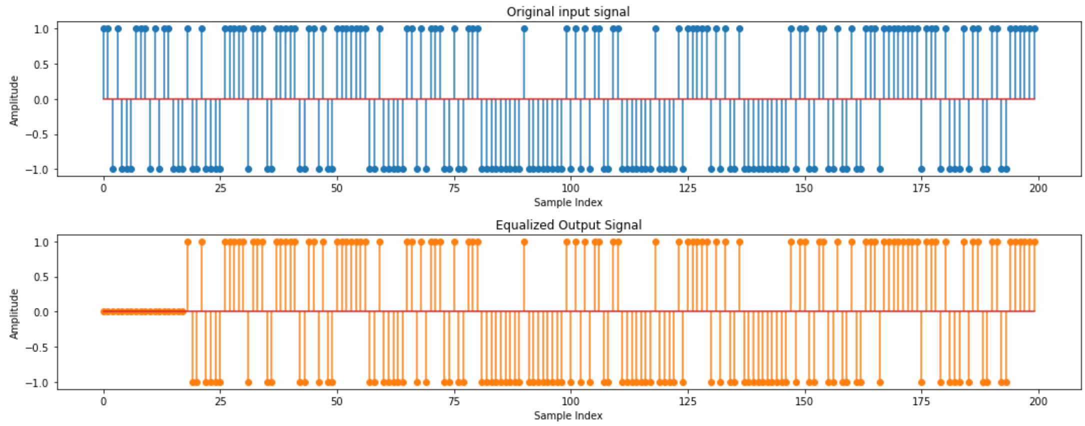


**Figure 16:** Input sequence vs Nonlinearly-equalized output

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 $\mu$. 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

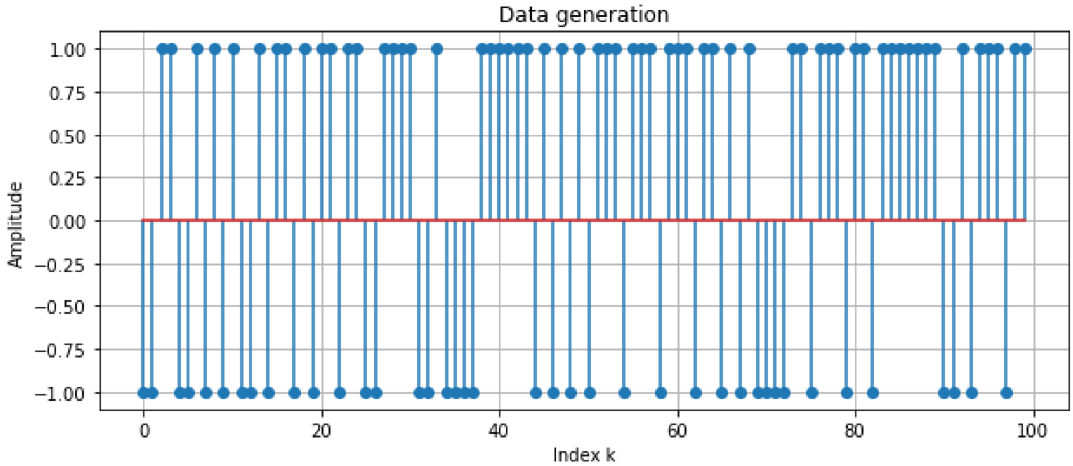


**Figure 17:** Generated input data $x[k]$

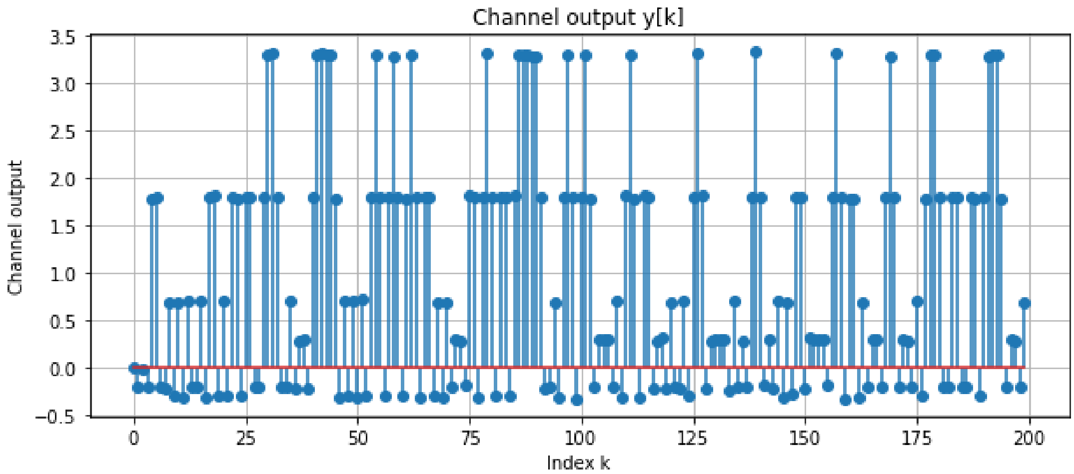


**Figure 18:** Nonlinear channel output

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

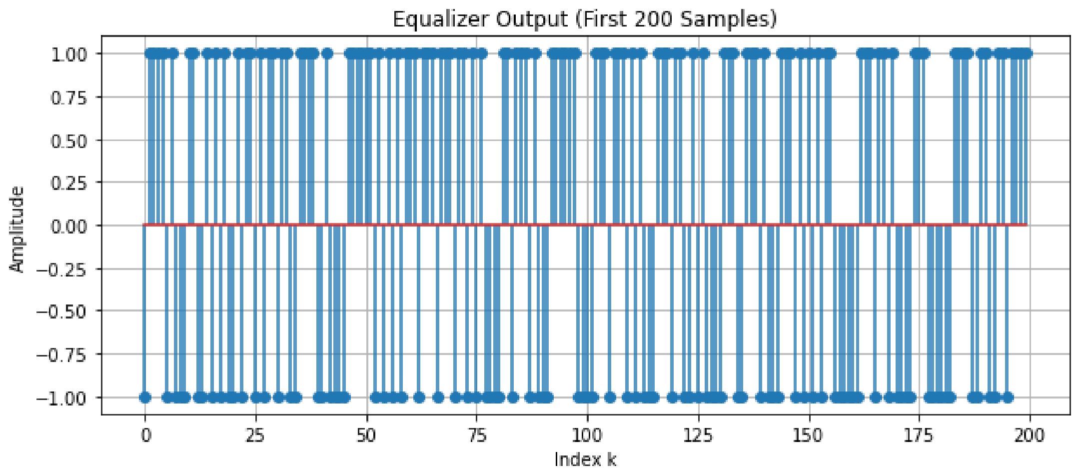


**Figure 19:** Linearly-equalized quantized signal

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

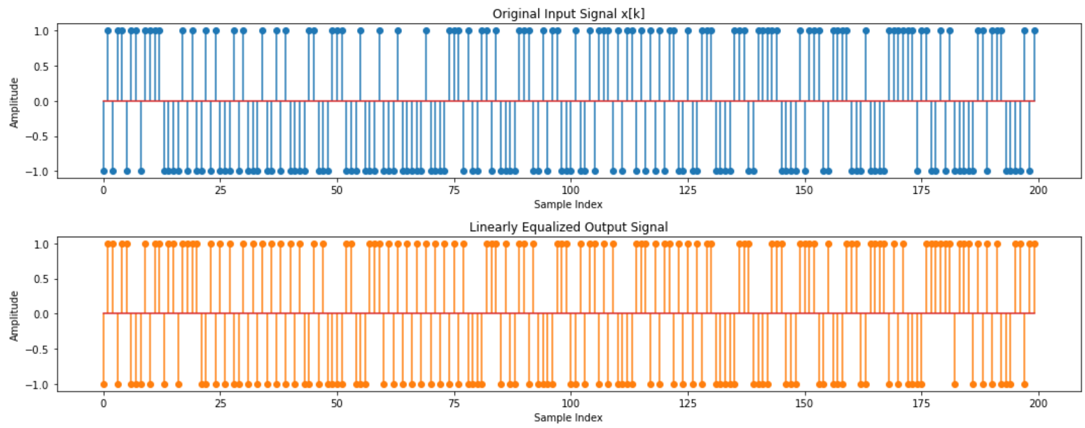


**Figure 20:** Input data vs Linearly-equalized signal

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).

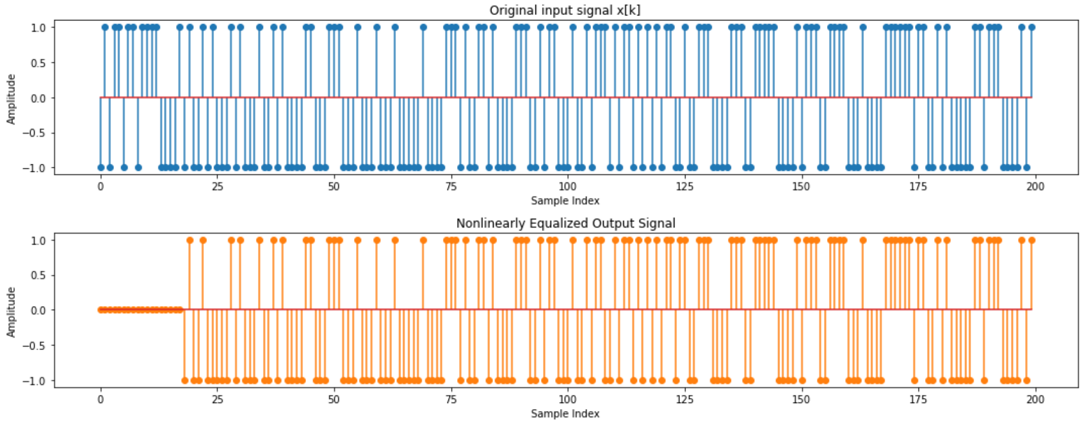


**Figure 21:** Input data vs Nonlinearly-equalized signal

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

```
# ASSIGNMENT 4 EXERCISE 1
import numpy as np
import matplotlib.pyplot as plt

# Exercise 3.1.1
# Implement linear discrete-time communication system model

L = 10000  # number of samples
x = np.random.choice([1, -1], size=L) # random sequence of +1 and -1

# Plot the data sequence
plt.figure(figsize=(10, 4))
plt.stem(x[:100])  # plotting only the first 100
plt.title('Data generation')
plt.xlabel('Index k')
plt.ylabel('Amplitude')
plt.grid(True)
plt.show()

# Exercise 3.1.2
# Use the convolution to obtain the output of the channel
W = 3
sigma_n = 10e-3

# channel impulse response
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)])
n = np.random.normal(0, sigma_n, L) # Gaussian noise n
# Convolve the data sequence x[k] with the channel impulse response h[k]
y = np.convolve(x, h, mode='full')[:L] + n
```

```python

# Plot the output of the channel
plt.figure(figsize=(10, 4))
plt.stem(y[:200])
plt.title('Channel output y[k]')
plt.xlabel('Index k')
plt.ylabel('Channel output')
plt.grid(True)
plt.show()

# Exercise 3.1.3
# Implement the linear adaptive equalizer

# Parameters
M = 11 # Number of taps in the equalizer
mu = 0.075  # Learning rate for gradient descent
D = 7 # Delay in samples to align the equalizer output with the desired signal
w = np.zeros(M)
errors = np.zeros(L)

# Output initialization
x_hat = np.zeros(L)

# The input signal     [    ] should be delayed by
# samples because the channel and equalizer
# will take some time to respond. The equalizer
# should predict     ^[    ] based on a delayed version of
#      [    ] and the current weights.
# This implies that you need to compute the error
# signal between the predicted output     ^[    ] (from
# the equalizer) and the actual signal delayed by      samples.

for k in range(M + D, L): # Implementing slide 32 update algorithm
    y_window = y[k - M:k]  # Select M recent samples from y
    x_hat[k] = w.T @ y_window
    errors[k] = x[k - D] - x_hat[k] # Error between delayed input and predicted output
    w += mu * errors[k] * y_window

# Convolve the channel output with the equalizer weights
eq_output = np.convolve(y, w, mode='full')[:L]

# Plot the equalizer output
plt.figure(figsize=(10, 4))
plt.stem(eq_output[:200])
plt.title('Equalizer Output (First 200 Samples)')
plt.xlabel('Index k')
plt.ylabel('Amplitude')
plt.grid(True)
plt.show()

# Exercise 3.1.4
# Plot the error squared as a function of number of iterations.

```

```python
# Plot the squared errors
plt.figure(figsize=(10, 4))
plt.plot(errors[:500] ** 2)
plt.title('Squared Error over Iterations')
plt.xlabel('Iteration k')
plt.ylabel('Squared Error')
plt.grid(True)
plt.show()

# Exercise 3.1.5
# Once the equalizer has converged and the weights of the equalizer has been learned,
# show that the distorted signal after the channel output can be equalized

# Plot the original input and the equalized output
plt.figure(figsize=(15, 6))

# Plot original input signal
plt.subplot(2, 1, 1)  # two rows, one column, first subplot
plt.stem(x[:200], linefmt='C0-', markerfmt='C0o')
plt.title('Original Input Signal x[k]')
plt.xlabel('Sample Index')
plt.ylabel('Amplitude')

# Plot equalized output signal
plt.subplot(2, 1, 2)  # two rows, one column, second subplot
plt.stem(eq_output[:200], linefmt='C1-', markerfmt='C1o')
plt.title('Equalized Output Signal')
plt.xlabel('Sample Index')
plt.ylabel('Amplitude')

plt.tight_layout()
plt.show()

# Exercise 3.1.6
# Compute the number or errors between the equalized signal and the original data sequence  x[k]
cross_corr = np.correlate(x, eq_output, mode='full')
# Find the index of the maximum correlation value
delay = np.argmax(cross_corr) - (len(x) - 1)

print(f'Detected delay: {-delay}')

# Calculate the error signal
error_signal = x[:len(x)+delay] - eq_output[-delay:]

# Count the number of mismatches (errors)
number_of_errors = np.sum(np.abs(error_signal) >= 0.1)

# Calculate the error rate
error_rate = number_of_errors / len(error_signal)

print(f"Number of Errors: {number_of_errors}")
print(f"Error Rate: {error_rate * 100:.2f}%")
```

```python
# ASSIGNMENT 4 EXERCISE 2
```

```
import numpy as np
import matplotlib.pyplot as plt

# Initialize
sigma2 = 0.09
N = 300
n_nodes = 15
mu = 0.01  # Learning rate for gradient descent

x_train = np.random.uniform(-1, 1, N)

# Define the functions
y1 = x_train ** 2
y2 = x_train ** 3
y3 = np.sin(x_train)
y4 = np.abs(x_train)

# Stack them into a single 2D array
Y = np.column_stack((y1, y2, y3, y4)) # each column is a different function

mse_history = []

for i in range(Y.shape[1]):
    Y_train = Y[:, i]
    Y_train = Y_train[:, np.newaxis]
    # Add bias column to X_train
    X_train = np.column_stack((x_train, np.ones((len(Y_train), 1))))

    # Initialize weights
    W1 = np.random.normal(0, np.sqrt(sigma2), (2, n_nodes))
    W2 = np.random.normal(0, np.sqrt(sigma2), (n_nodes + 1, 1))

    # Track the MSE for the current function
    mse_per_iteration = []

    for epoch in range(500):
        y_pred = []
        for k in range(len(X_train[:,0])):

            # Forward pass to hidden layer
            A = X_train[k,:] @ W1 # (n_nodes,)

            # Apply activation function np.tanh() and add bias column
            Z = np.append(np.tanh(A), 1) # (n_nodes + 1, 1)

            # Backward pass: Compute the gradient and update the weights
            # Gradient of the error with respect to W2
            dE_dW2 = Z * (Z @ W2 - Y_train[k,:]) # (n_nodes + 1, 1)
            W2 -= mu * dE_dW2[:, np.newaxis]   # Update rule for W2 (n_nodes + 1, 1)

            # Gradient of the error with respect to W1
            dE_dW1 = X_train[k,:] * ((1 - np.tanh(A)**2)[:, np.newaxis] * (Z @ W2 - Y_train[k,:]) * W2[0:-1]) # (n_nodes, 2)
            W1 -= mu * dE_dW1.T # (2, n_nodes)
```

```python

            # Compute predictions from output layer
            y_pred.append(Z @ W2) # (1, )
        # Calculate MSE for this iteration
        mse = np.mean((y_pred - Y_train) ** 2)
        mse_per_iteration.append(mse)

    # Store the MSE for the current function
    mse_history.append(mse_per_iteration)

    # After training, use the final updated weights to predict. TESTING
    x_test = np.linspace(-1, 1, N)
    X_test = np.column_stack((x_test, np.ones((len(Y_train), 1))))
    A_final = X_test @ W1
    Z_final = np.column_stack((np.tanh(A_final), np.ones((len(A_final), 1))))
    y_pred_final = Z_final @ W2

    # Plot
    plt.figure()
    plt.scatter(x_train, Y_train, label='Training data')
    plt.scatter(x_test, y_pred_final, label='Predicted', alpha=0.4)
    plt.title(f"Prediction of curve {i}")
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.grid()
    plt.legend()
    plt.show()

# Plot MSE history for each function
plt.figure()
for idx, mse in enumerate(mse_history):
    plt.plot(mse, label=f'Function {idx+1}')
plt.title("Mean Squared Error (MSE) over Iterations")
plt.xlabel("Epochs")
plt.ylabel("MSE")
plt.legend()
plt.grid()
plt.show()
```

```python
# ASSIGNMENT 4 EXERCISE 3
import numpy as np
import matplotlib.pyplot as plt

L = 50000  # number of samples
W = 3
# channel impulse response
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)])
#sigma_n = 10e-3
n = 0#np.random.normal(0, sigma_n, L) # Gaussian noise n
a = 0.8

# Exercise 3.3.1
# Generate the data sequence x[k].
x_train = np.random.choice([1, -1], size=L) # random sequence of +1 and -1
```

```
# Plot the data sequence
plt.figure(figsize=(10, 4))
plt.stem(x_train[:100])  # plotting only the first 100
plt.title('Data generation')
plt.xlabel('Index')
plt.ylabel('a.u.')
plt.grid(True)
plt.show()

# Plot the data sequence after it has passed the nonlinear channel.
# Obtain y[k]
convolved = np.convolve(x_train, h, mode='full')[:L]
Y_train = (convolved + a * (convolved ** 2) + n)

# Plot the output of the channel
plt.figure(figsize=(10, 4))
plt.stem(Y_train[:200])
plt.title('Channel output y[k]')
plt.xlabel('Index')
plt.ylabel('a.u.')
plt.grid(True)
plt.show()

# Exercise 3.3.2
# Implement the nonlinear adaptive equalizer
M = 11
D = 7
n_nodes = 100
sigma2 = 0.09
mu = 0.01  # Learning rate for gradient descent
w = np.zeros(M)
y_hat = np.zeros(L)
y_pred_final = np.zeros(L)
errors = np.zeros(L)

for epoch in range(200):
    # Run the nonlinear equalizer several times, each time with different weights initialization seed
    W1 = np.random.normal(0, np.sqrt(sigma2), (M + 1, n_nodes))
    W2 = np.random.normal(0, np.sqrt(sigma2), (n_nodes + 1, 1))
    # We feed the NN with the output of the channel Y_train
    # The goal is to perform equalization and get back
    # An estimation of the original input signal x_train
    for k in range(M + D, L): # Implementing slide 32 update algorithm
        # Add bias column to Y_train
        y_window = np.append(Y_train[:, np.newaxis][k - M:k], 1) # Select M recent samples from Y_train (M + 1, 1)

        # Forward pass to hidden layer
        A = y_window.T @ W1 # (1, n_nodes)

        # Apply activation function np.tanh() and add bias column
        Z = np.append(np.tanh(A), 1) # (n_nodes + 1, 1)

        # Compute predictions from output layer
```

```python
        y_hat[k] = Z @ W2 # (L, )

        # Backward pass: Compute the gradient and update the weights
        # Gradient of the error with respect to W2
        dE_dW2 = Z * (y_hat[k] - x_train[k]) # (1, n_nodes+1)
        W2 -= mu * dE_dW2[:, np.newaxis]  # Update rule for W2 (n_nodes+1, 1)

        # Gradient of the error with respect to W1
        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)
        W1 -= mu * dE_dW1.T # (2, n_nodes)

        errors[k] = y_hat[k] - x_train[k - D] # Error between delayed input and predicted output

# After training, use the final updated weights to predict. TESTING
x_test = np.random.choice([1, -1], size=L) # (L,1)
convolved = np.convolve(x_test, h, mode='full')[:L]
Y_test = (convolved + a * (convolved ** 2) + n)
for k in range(M + D, L): # Implementing slide 32 update algorithm
    # Add bias column to Y_test
    y_window = np.append(Y_test[:, np.newaxis][k - M:k], 1) # Select M recent samples from Y_est (M + 1, 1)

    # Forward pass to hidden layer
    A = y_window.T @ W1 # (1, n_nodes)

    # Apply activation function np.tanh() and add bias column
    Z = np.append(np.tanh(A), 1) # (n_nodes + 1, 1)

    # Compute predictions from output layer
    y_hat[k] = Z @ W2 # (L, )

    # Backward pass: Compute the gradient and update the weights
    # Gradient of the error with respect to W2
    dE_dW2 = Z * (y_hat[k] - x_test[k]) # (1, n_nodes+1)
    W2 -= mu * dE_dW2[:, np.newaxis]  # Update rule for W2 (n_nodes+1, 1)
    y_pred_final[k] = Z @ W2 # (L,)

    # Gradient of the error with respect to W1
    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)
    W1 -= mu * dE_dW1.T # (2, n_nodes)

    #errors[k] = y_hat[k] - Y_train[k - D] # Error between delayed input and predicted output

# Plot the original input and the equalized output
plt.figure(figsize=(15, 6))

# Plot original input signal
plt.subplot(2, 1, 1)  # two rows, one column, first subplot
plt.stem(x_test[:200], linefmt='C0-', markerfmt='C0o')
plt.title('Original input signal')
```

```python
plt.xlabel('Sample Index')
plt.ylabel('Amplitude')

y_pred_final = np.sign(y_pred_final)
# Plot equalized output signal
plt.subplot(2, 1, 2)  # two rows, one column, second subplot
plt.stem(y_pred_final[:200], linefmt='C1-', markerfmt='C1o')
plt.title('Equalized Output Signal')
plt.xlabel('Sample Index')
plt.ylabel('Amplitude')
plt.tight_layout()
plt.show()

# Compute the number or errors between the equalized signal and the
    original data sequence  x[k]
cross_corr = np.correlate(x_test, y_pred_final, mode='full')
# Find the index of the maximum correlation value
delay = np.argmax(cross_corr) - (len(x_test) - 1)

print(f'Detected delay: {-delay}')

# Calculate the error signal
error_signal = x_test[:len(x_test)+delay] - y_pred_final[-delay:]

# Count the number of mismatches (errors)
#number_of_errors = np.sum(y_pred_final != x_test)
number_of_errors = np.sum(np.abs(error_signal) >= 0.1)

# Calculate the error rate
error_rate = number_of_errors / len(error_signal)

print(f"Number of Errors: {number_of_errors}")
print(f"Error Rate: {error_rate * 100:.2f}%")
```

```python
# ASSIGNMENT 4 EXERCISE 4
import numpy as np
import matplotlib.pyplot as plt

# Implement linear discrete-time communication system model
L = 50000  # number of samples
x = np.random.choice([1, -1], size=L) # random sequence of +1 and -1

# Plot the data sequence
plt.figure(figsize=(10, 4))
plt.stem(x[:100])  # plotting only the first 100
plt.title('Data generation')
plt.xlabel('Index k')
plt.ylabel('Amplitude')
plt.grid(True)
plt.show()

# Use the convolution to obtain the output of the channel
W = 3
a = 0.8
sigma_n = 10e-3

```

```
# channel impulse response
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)])
n = np.random.normal(0, sigma_n, L) # Gaussian noise n
# Convolve the data sequence x[k] with the channel impulse response h[k]
convolved = np.convolve(x, h, mode='full')[:L]
y = (convolved + a * (convolved ** 2) + n)

# Plot the output of the channel
plt.figure(figsize=(10, 4))
plt.stem(y[:200])
plt.title('Channel output y[k]')
plt.xlabel('Index k')
plt.ylabel('Channel output')
plt.grid(True)
plt.show()

# ---------------- Implement the linear adaptive equalizer
# Parameters
M = 11 # Number of taps in the equalizer
mu = 0.033  # Learning rate for gradient descent
D = 7 # Delay in samples to align the equalizer output with the desired
    signal
w = np.zeros(M)
errors = np.zeros(L)

# Output initialization
x_hat = np.zeros(L)

for k in range(M + D, L): # Implementing slide 32 update algorithm
    y_window = y[k - M:k]  # Select M recent samples from y
    x_hat[k] = w.T @ y_window
    errors[k] = x[k - D] - x_hat[k] # Error between delayed input and
    predicted output
    w += mu * errors[k] * y_window

# Plot the error squared as a function of number of iterations.
plt.figure(figsize=(10, 4))
plt.plot(errors[:500] ** 2)
plt.title('Squared Error over Iterations')
plt.xlabel('Iteration k')
plt.ylabel('Squared Error')
plt.grid(True)
plt.show()

# Convolve the channel output with the equalizer weights
eq_output = np.convolve(y, w, mode='full')[:L]
eq_output = np.sign(eq_output)

# Plot the equalizer output
plt.figure(figsize=(10, 4))
plt.stem(eq_output[:200])
plt.title('Equalizer Output (First 200 Samples)')
plt.xlabel('Index k')
plt.ylabel('Amplitude')
```

```python
plt.grid(True)
plt.show()

# Implement the nonlinear adaptive equalizer
#M = 11
#D = 7
n_nodes = 100
sigma2 = 0.09
mu = 0.01  # Learning rate for gradient descent
y_hat = np.zeros(L)
y_pred_final = np.zeros(L)
#errors = np.zeros(L)

for epoch in range(300):
    # Run the nonlinear equalizer several times, each time with different weights initialization seed
    W1 = np.random.normal(0, np.sqrt(sigma2), (M + 1, n_nodes))
    W2 = np.random.normal(0, np.sqrt(sigma2), (n_nodes + 1, 1))
    # We feed the NN with the output of the channel Y_train
    # The goal is to perform equalization and get back
    # An estimation of the original input signal x_train
    for k in range(M + D, L): # Implementing slide 32 update algorithm

        y_window = np.append(eq_output[:, np.newaxis][k - M:k], 1) # Select M recent samples from Y_train (M + 1, 1)

        A = y_window.T @ W1 # (1, n_nodes)

        Z = np.append(np.tanh(A), 1) # (n_nodes + 1, 1)

        y_hat[k] = Z @ W2 # (L, )

        dE_dW2 = Z * (y_hat[k] - x[k]) # (1, n_nodes+1)
        W2 -= mu * dE_dW2[:, np.newaxis]  # Update rule for W2 (n_nodes+1, 1)

        dE_dW1 = (y_hat[k] - x[k]) * (1 - np.tanh(A)**2).T[:, np.newaxis] * W2[:-1] * y_window # (n_nodes, n_nodes + 2)
        W1 -= mu * dE_dW1.T # (2, n_nodes)

        errors[k] = y_hat[k] - x[k - D] # Error between delayed input and predicted output


# Plot the error squared as a function of number of iterations.
plt.figure(figsize=(10, 4))
plt.plot(errors[:500] ** 2)
plt.title('Squared Error over Iterations')
plt.xlabel('Iteration k')
plt.ylabel('Squared Error')
plt.grid(True)
plt.show()

# After training, use the final updated weights to predict ------------ TESTING
```

```
x_test = np.random.choice([1, -1], size=L) # (L,1)
convolved = np.convolve(x_test, h, mode='full')[:L]
y_test = (convolved + a * (convolved ** 2) + n)

# Convolve the channel output with the equalizer weights
eq_output = np.convolve(y_test, w, mode='full')[:L]
eq_output = np.sign(eq_output)

for k in range(M + D, L): # Implementing slide 32 update algorithm
    # Add bias column to Y_test
    y_window = np.append(eq_output[:, np.newaxis][k - M:k], 1) # Select M recent samples from Y_est (M + 1, 1)

    # Forward pass to hidden layer
    A = y_window.T @ W1 # (1, n_nodes)

    # Apply activation function np.tanh() and add bias column
    Z = np.append(np.tanh(A), 1) # (n_nodes + 1, 1)

    # Compute predictions from output layer
    y_hat[k] = Z @ W2 # (L, )

    # Backward pass: Compute the gradient and update the weights
    # Gradient of the error with respect to W2
    dE_dW2 = Z * (y_hat[k] - x_test[k]) # (1, n_nodes+1)
    W2 -= mu * dE_dW2[:, np.newaxis]  # Update rule for W2 (n_nodes+1, 1)
    y_pred_final[k] = Z @ W2 # (L,)

    # Gradient of the error with respect to W1
    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)
    W1 -= mu * dE_dW1.T # (2, n_nodes)

# Once the equalizer has converged and the weights of the equalizer has been learned,
# show that the distorted signal after the channel output can be equalized
plt.figure(figsize=(15, 6))

# Plot original input signal
plt.subplot(2, 1, 1)  # two rows, one column, first subplot
plt.stem(x_test[:200], linefmt='C0-', markerfmt='C0o')
plt.title('Original Input Signal x[k]')
plt.xlabel('Sample Index')
plt.ylabel('Amplitude')

# Plot equalized output signal
plt.subplot(2, 1, 2)  # two rows, one column, second subplot
plt.stem(eq_output[:200], linefmt='C1-', markerfmt='C1o')
plt.title('Linearly Equalized Output Signal')
plt.xlabel('Sample Index')
plt.ylabel('Amplitude')

plt.tight_layout()
plt.show()
```

```
# Compute the number or errors between the equalized signal and the
    original data sequence  x[k]
cross_corr = np.correlate(x_test, eq_output, mode='full')
# Find the index of the maximum correlation value
delay = np.argmax(cross_corr) - (len(x_test) - 1)

print(f'Detected delay: {-delay}')

# Calculate the error signal
error_signal = x_test[:len(x)+delay] - eq_output[-delay:]

# Count the number of mismatches (errors)
number_of_errors = np.sum(np.abs(error_signal) >= 0.1)

# Calculate the error rate
error_rate = number_of_errors / len(error_signal)

print(f"Number of Errors: {number_of_errors}")
print(f"Error Rate: {error_rate * 100:.2f}%")

# Plot the original input and the equalized output
plt.figure(figsize=(15, 6))

# Plot original input signal
plt.subplot(2, 1, 1)  # two rows, one column, first subplot
plt.stem(x_test[:200], linefmt='C0-', markerfmt='C0o')
plt.title('Original input signal x[k]')
plt.xlabel('Sample Index')
plt.ylabel('Amplitude')

y_pred_final = np.sign(y_pred_final)
# Plot equalized output signal
plt.subplot(2, 1, 2)  # two rows, one column, second subplot
plt.stem(y_pred_final[:200], linefmt='C1-', markerfmt='C1o')
plt.title('Nonlinearly Equalized Output Signal')
plt.xlabel('Sample Index')
plt.ylabel('Amplitude')
plt.tight_layout()
plt.show()

# Compute the number or errors between the equalized signal and the
    original data sequence x[k]
cross_corr = np.correlate(x_test, y_pred_final, mode='full')
# Find the index of the maximum correlation value
delay = np.argmax(cross_corr) - (len(x_test) - 1)
print(f'Detected delay: {-delay}')

# Calculate the error signal
error_signal = x_test[:len(x_test)+delay] - y_pred_final[-delay:]

# Count the number of mismatches (errors)
#number_of_errors = np.sum(y_pred_final != x_test)
number_of_errors = np.sum(np.abs(error_signal) >= 0.1)

# Calculate the error rate
```

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