## Packages to upload

```
In [1]: #from google.colab import drive
        #from google.colab import files
        import torch
        import torch.nn as nn
        import torch.nn.functional as F
        import torch.optim as optim
        import pandas as pd
        import numpy as np
        import random
        import matplotlib.pyplot as plt
        import cvxpy as cp
        import scipy as scipy
        import cvxopt as cvxopt
        import cvxpylayers as cvxpylayers
        from numpy.linalg import matrix_rank
        from cvxpylayers.torch import CvxpyLayer
        from sklearn.model_selection import train_test_split
```

## Data

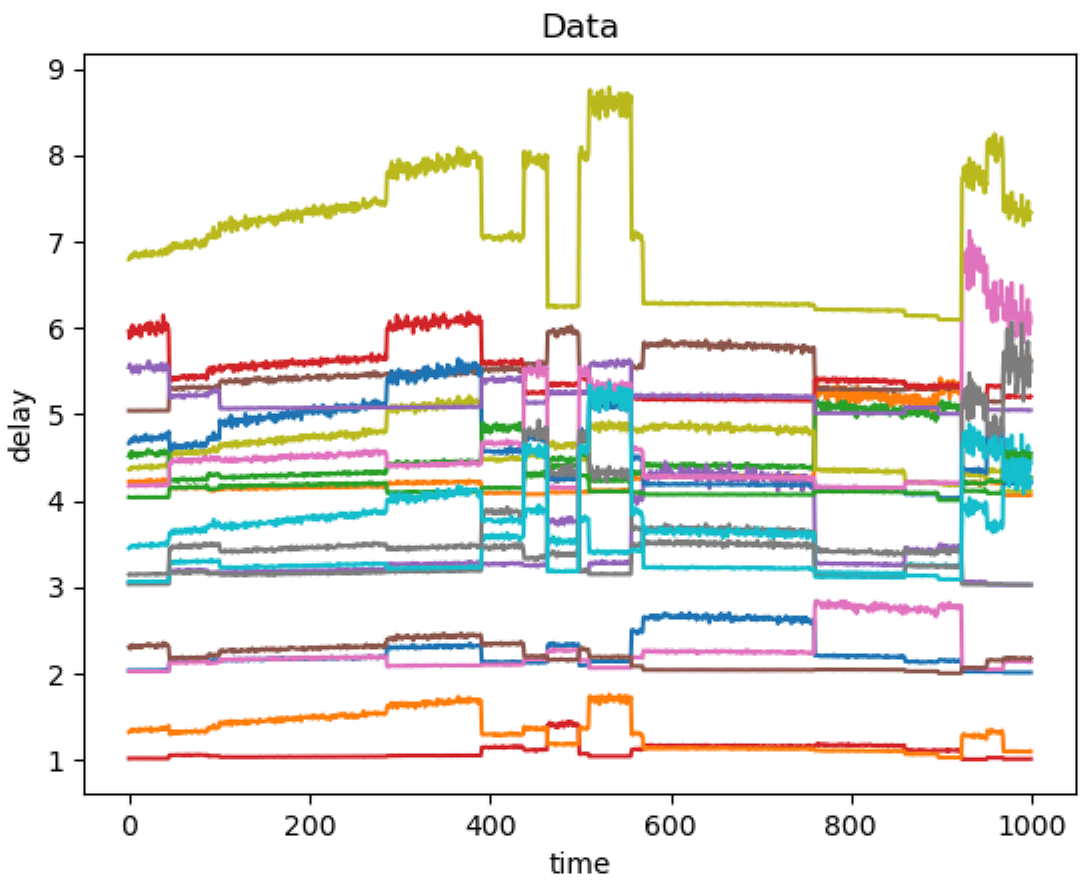
In this lab, the dataset used are the evolution of delays in a network of N nodes. The columns are the time instants and the row are the source-destination pairs.

1/ Upload the dataset and excute the cell with plt.plot. Can you list three main properties that your dataset have?

```
In [2]: df=pd.read_csv('Data/red_data1.csv', sep=',',dtype=float)
        df=np.array(df.values)
```

```
In [3]: plt.plot()
        for i in range(df.shape[0]):
          plt.plot(df[i,:])
        plt.title('Data')
        plt.xlabel('time')
        plt.ylabel('delay')
```

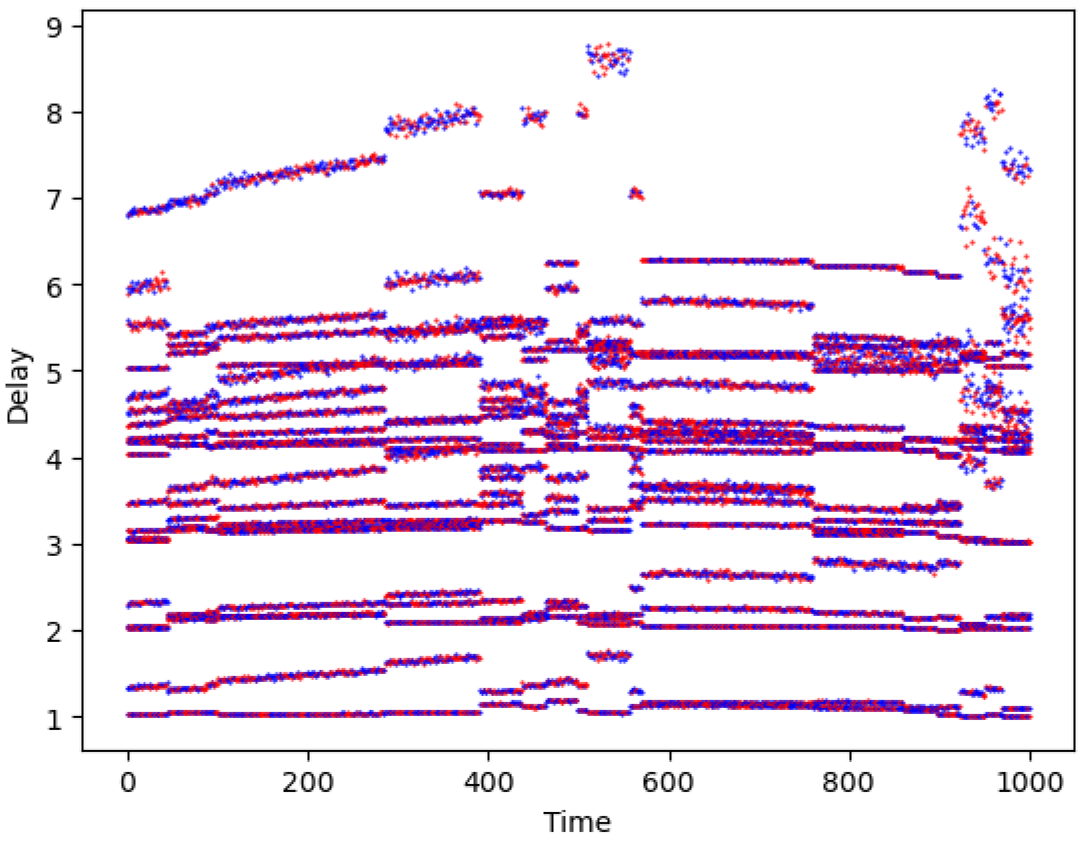
```
Out[3]: Text(0, 0.5, 'delay')
```



We observe that delay times are approximatevely constant at intervals. All signals show as well an additive noise. Also, all delays are highly correlated, with changes always taking time at the same time.

In this lab, we will focus on the matrix completion problem. The matrix completion aims to fill in missing entries of a partially observed matrix. In the next cell, we only sample 50% of the observations of the delay matrix.

In [4]:
```
m = df.shape[0]
n = df.shape[1]
W = np.random.binomial(n=1, p=0.5, size=m * n).reshape(m, n)

plt.plot()

for i in range(df.shape[0]):
    for j in range(df.shape[1]):
        if W[i, j] == 1:
            plt.plot(j, df[i, j], marker='.', color='b',markersize=1)
        else:
            plt.plot(j, df[i, j], marker='.', color='r',markersize=1)

plt.xlabel('Time')
plt.ylabel('Delay')

plt.show()
```



## Completion using matrix factorisation

The matrix completion can be solved as follows:

Let $M$ be the partially observed matrix of size $m \times n$, and $W$ be the set of observed indices (entries). We seek to find matrices $U$ and $V$ such that $M \approx UV$, where $U$ is of size $m \times r$, $V$ is of size $r \times m$ with rank $r < m, n$. Usually to find $U$ and $V$ we solve the following optimisation problem:

$$\min_{U,V} \sum_{(i,j)\in\Omega} (M_{ij} - (UV)_{ij})^2$$

To solve such problems, one classical algorithm is the Alternating Least Squares (ALS) algorithm :

- Initialize $U^{(0)}$ and $V^{(0)}$ with small random values.
- Repeat until convergence:
  - Update $U$ by solving by minimizing the following objective while fixing $V$:

$$U = \arg\min_{U} \sum_{i,j:W_{ij}=1} (M_{ij} - (UV)_{ij})^2$$

  - Update $V$ by solving by minimizing the following objective while fixing $U$:

$$V = \arg\min_{U} \sum_{i,j:W_{ij}=1} (M_{ij} - (UV)_{ij})^2$$

2/ Complete the code below to implement the Alternating Least Squares (ALS) algorithm? You will need to use the cvxpy package. Try to observe the performance of the reconstruction when you change the number of points observed.

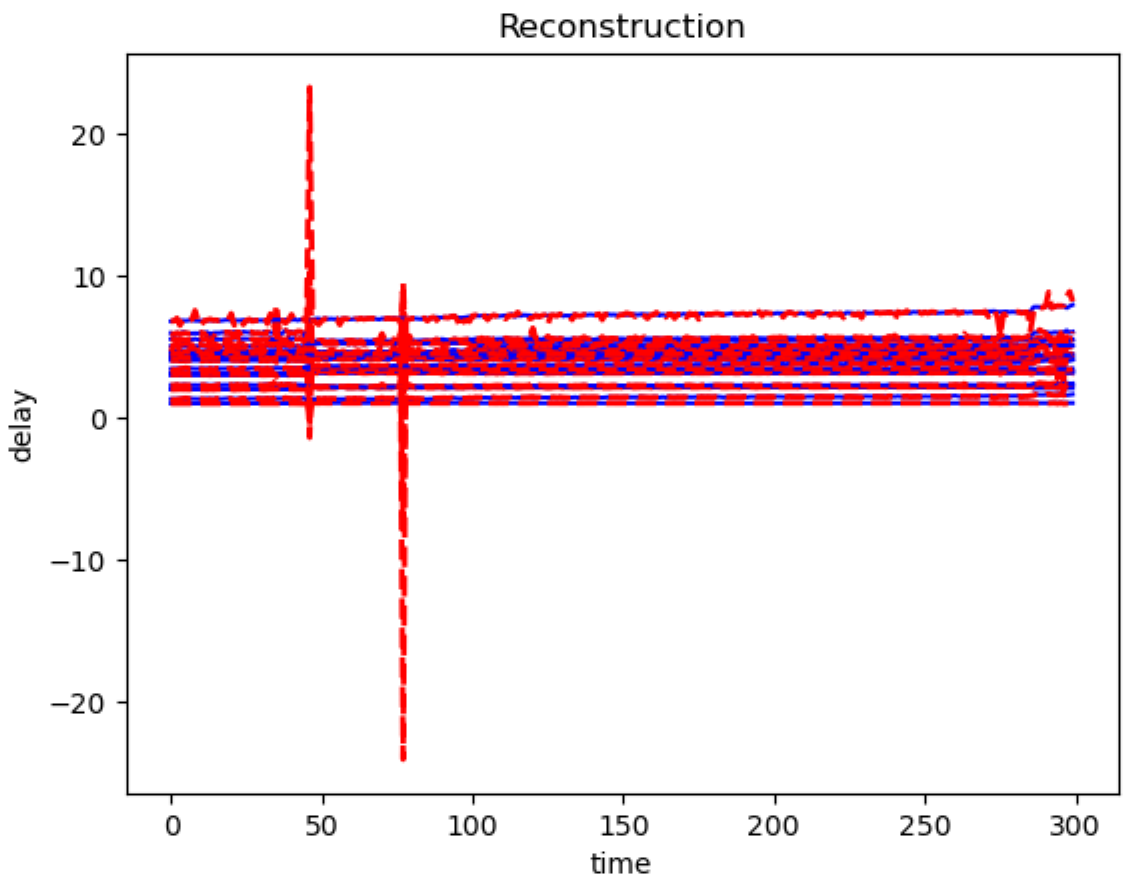
```
In [5]: # SVD
np.random.seed(1)
# Ensure same initial random Y, rather than generate new one
# when executing this cell.


A = df[:,:300]

def solve_UV(A, prob):
    m = A.shape[0]
    n = A.shape[1]
    k = 5
    W = np.random.binomial(n=1, p=prob, size=m*n).reshape(m,n)#np.random.rar
    # Initialize Y randomly.
    U_init = np.random.rand(m, k)

    U = U_init

    # Perform alternating minimization.
    MAX_ITERS = 15
    residual = np.zeros(MAX_ITERS)
    for iter_num in range(1, 1+MAX_ITERS):
        # At the beginning of an iteration, X and Y are NumPy
        #  array types, NOT CVXPY variables.

        # For odd iterations, treat Y constant, optimize over X.
        if iter_num % 2 == 1:
            V = cp.Variable((k, n))
        # For even iterations, treat X constant, optimize over Y.
        else:
            U = cp.Variable((m, k))

        # Solve the problem
        C = U@V
        cost = 0
        for i in range(m):
            for j in range(n):
                if W[i,j] ==1:
                    cost += cp.square(A[i, j] - U[i, :] @ V[:, j]) # complet
        obj = cp.Minimize(cost)
        prob = cp.Problem(obj)
        prob.solve(solver=cp.SCS, max_iters=10000)

        if prob.status != cp.OPTIMAL:
            raise Exception("Solver did not converge!")

        print('Iteration {}, residual norm {}'.format(iter_num, prob.value))
```

```python
        residual[iter_num-1] = prob.value

        # Convert variable to NumPy array constant for next iteration.
        if iter_num % 2 == 1:
            V = V.value
        else:
            U = U.value
    return U, V
```

In [6]:
```python
U, V = solve_UV(A, 0.5)
times_series = np.matmul(U,V)
plt.plot()
for i in range(times_series.shape[0]):
  plt.plot(A[i,:],color = "blue")
  plt.plot(times_series[i,:],'--', color = "red",linewidth=2)


plt.title('Reconstruction')
plt.xlabel('time')
plt.ylabel('delay')
```

```
/home/gonzalo/anaconda3/envs/NN/lib/python3.11/site-packages/cvxpy/problems/
problem.py:158: UserWarning: Objective contains too many subexpressions. Con
sider vectorizing your CVXPY code to speed up compilation.
  warnings.warn("Objective contains too many subexpressions. "
Iteration 1, residual norm 5383.530285665022
Iteration 2, residual norm 997.6639821173919
Iteration 3, residual norm 128.989554380791
Iteration 4, residual norm 43.944191488140206
Iteration 5, residual norm 18.660523869068545
Iteration 6, residual norm 11.29379390282354
Iteration 7, residual norm 7.2333743410698235
Iteration 8, residual norm 5.056599628189673
Iteration 9, residual norm 3.633657922466384
Iteration 10, residual norm 2.7473352910512348
Iteration 11, residual norm 2.181044523565017
Iteration 12, residual norm 1.7078867725227513
Iteration 13, residual norm 1.401966400930903
Iteration 14, residual norm 1.1749664985585169
Iteration 15, residual norm 1.023648384339188
```

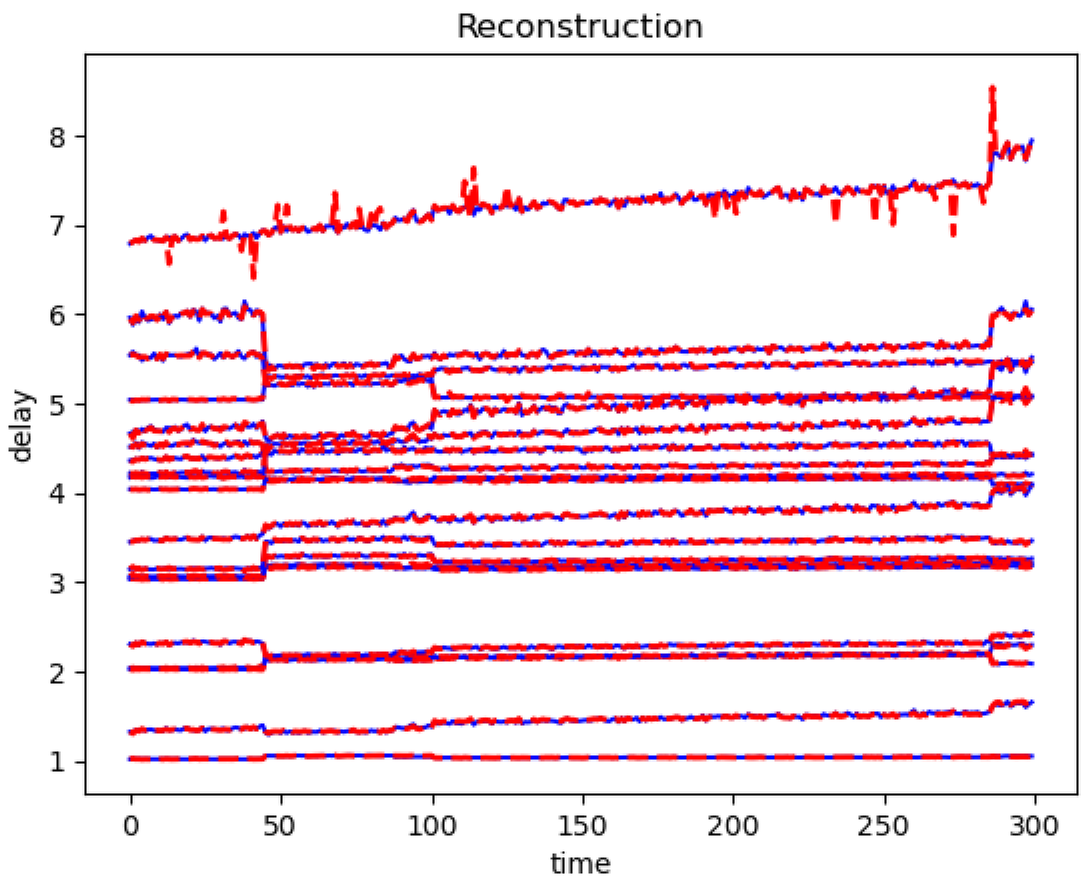
Out[6]:
```
Text(0, 0.5, 'delay')
```



```
In [7]: U, V = solve_UV(A, 0.9)
        times_series = np.matmul(U,V)
        plt.plot()
        for i in range(times_series.shape[0]):
          plt.plot(A[i,:],color = "blue")
          plt.plot(times_series[i,:],'--', color = "red",linewidth=2)


        plt.title('Reconstruction')
        plt.xlabel('time')
        plt.ylabel('delay')
```

```
Iteration 1, residual norm 14321.22137107051
Iteration 2, residual norm 171.7905422583908
Iteration 3, residual norm 24.726751428283606
Iteration 4, residual norm 6.047193587206831
Iteration 5, residual norm 1.8026566145139313
Iteration 6, residual norm 1.344740735081575
Iteration 7, residual norm 1.2120838993278937
Iteration 8, residual norm 1.0867687700203374
Iteration 9, residual norm 0.9624793494309382
Iteration 10, residual norm 0.807510815730563
Iteration 11, residual norm 0.6864809085361058
Iteration 12, residual norm 0.6075277803262302
Iteration 13, residual norm 0.5702621226025357
Iteration 14, residual norm 0.5538841167765504
Iteration 15, residual norm 0.5463645672772229
```

```
Out[7]: Text(0, 0.5, 'delay')
```



As expected, as the number of observed points increases, the precision of the prediction improves.

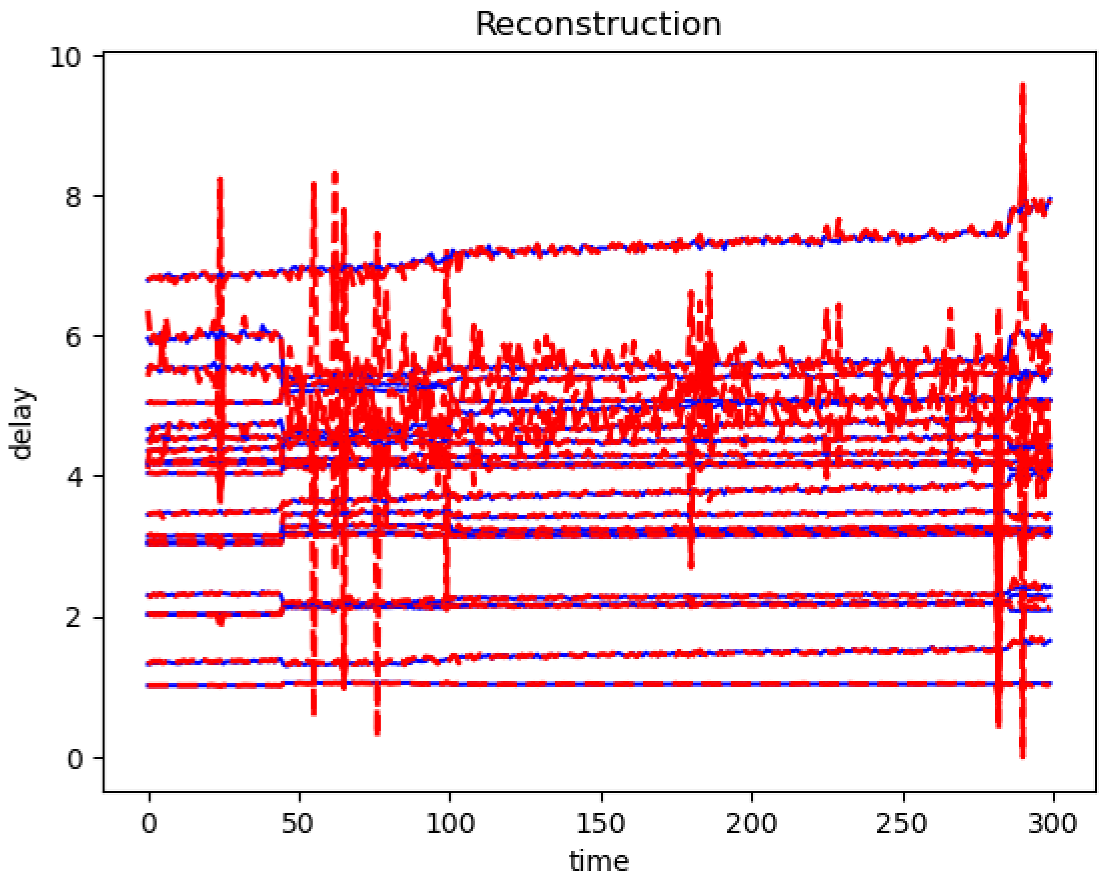
3/ One main property of delays is that they are positive. How to modify the ALS to ensure positiveness? Implement your solution and try it on the dataset with different sampling rate.

In [8]:
```
def solve_UV_positive(A, prob):
    m = A.shape[0]
    n = A.shape[1]
    k = 5
    W = np.random.binomial(n=1, p=prob, size=m*n).reshape(m,n)#np.random.rar
    # Initialize Y randomly.
    U_init = np.random.rand(m, k)
    U = np.abs(U_init)

    # Perform alternating minimization.
    MAX_ITERS = 15
    residual = np.zeros(MAX_ITERS)
    for iter_num in range(1, 1+MAX_ITERS):
        # At the beginning of an iteration, X and Y are NumPy
        #  array types, NOT CVXPY variables.

        # For odd iterations, treat Y constant, optimize over X.
        if iter_num % 2 == 1:
            V = cp.Variable((k, n))
        # For even iterations, treat X constant, optimize over Y.
        else:
            U = cp.Variable((m, k))

        # Solve the problem
        C = U@V
        cost = 0
        for i in range(m):
            for j in range(n):
                if W[i,j] ==1:
                    cost += cp.square(A[i, j] - U[i, :] @ V[:, j]) # complet

        constraints = [U @ V >= 0]

        obj = cp.Minimize(cost)
        prob = cp.Problem(obj, constraints)
        prob.solve(solver=cp.SCS, max_iters=10000)

        if prob.status != cp.OPTIMAL:
            raise Exception("Solver did not converge!")

        print('Iteration {}, residual norm {}'.format(iter_num, prob.value))
        residual[iter_num-1] = prob.value

        # Convert variable to NumPy array constant for next iteration.
        if iter_num % 2 == 1:
            V = V.value
        else:
            U = U.value
    return U, V
```

In [9]:
```
U, V = solve_UV_positive(A, 0.5)
times_series = np.matmul(U,V)
plt.plot()
for i in range(times_series.shape[0]):
  plt.plot(A[i,:],color = "blue")
  plt.plot(times_series[i,:],'--', color = "red",linewidth=2)


plt.title('Reconstruction')
plt.xlabel('time')
plt.ylabel('delay')
```

```
Iteration 1, residual norm 3599.677502126938
Iteration 2, residual norm 530.3268602278229
Iteration 3, residual norm 53.83018606361047
Iteration 4, residual norm 16.391122486526342
Iteration 5, residual norm 6.255598071194362
Iteration 6, residual norm 4.012303550740388
Iteration 7, residual norm 2.61946437849281
Iteration 8, residual norm 1.8575584769198523
Iteration 9, residual norm 1.3703434283576517
Iteration 10, residual norm 1.076168696469205
Iteration 11, residual norm 0.8856994326435731
Iteration 12, residual norm 0.7310460085466507
Iteration 13, residual norm 0.6272884955047264
Iteration 14, residual norm 0.5415997244031893
Iteration 15, residual norm 0.49036406916612263
```

```
Out[9]: Text(0, 0.5, 'delay')
```



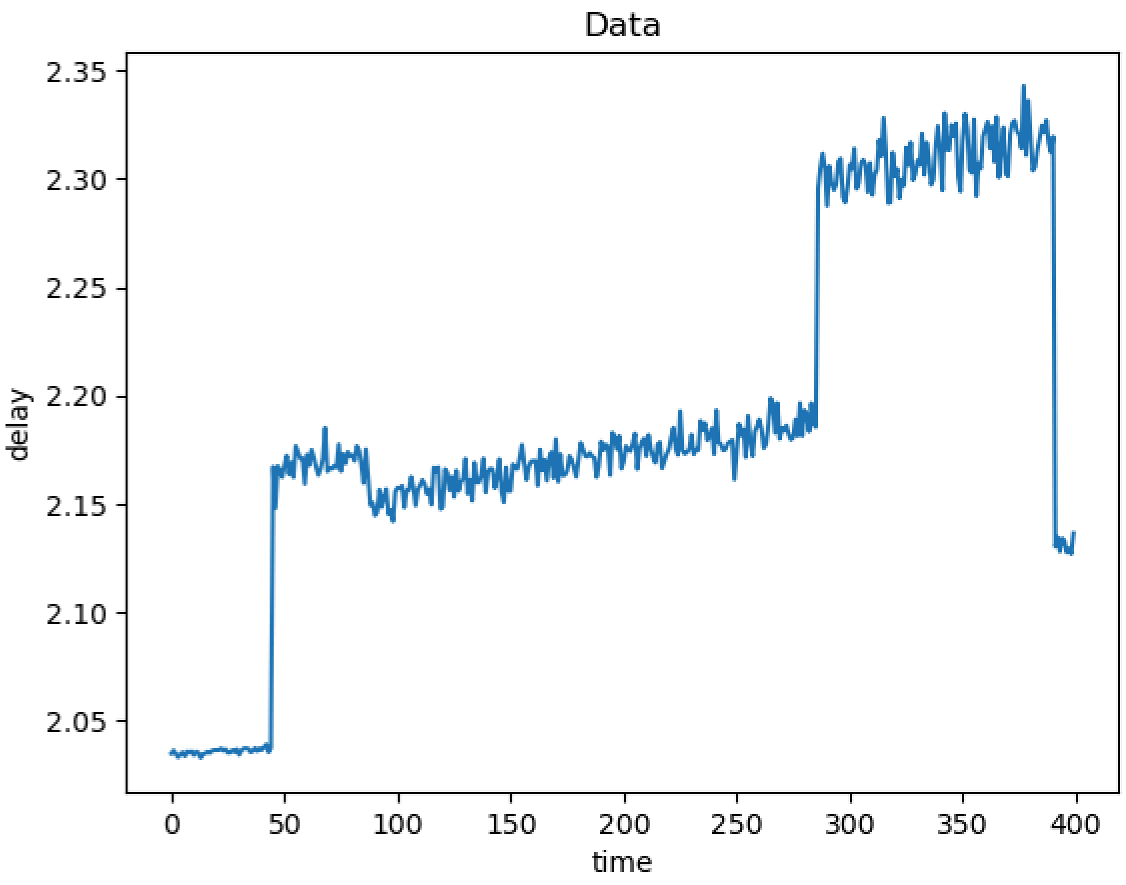
The prediction keeps on showing a weird behavior, but at least now all values are positive :)

## Neural Network

We will now do the completion of one time-serie.

In [24]:
```
y_df = df[0,:400]
plt.plot()
plt.plot(y_df)
plt.title('Data')
plt.xlabel('time')
plt.ylabel('delay')
```

Out[24]:
```
Text(0, 0.5, 'delay')
```

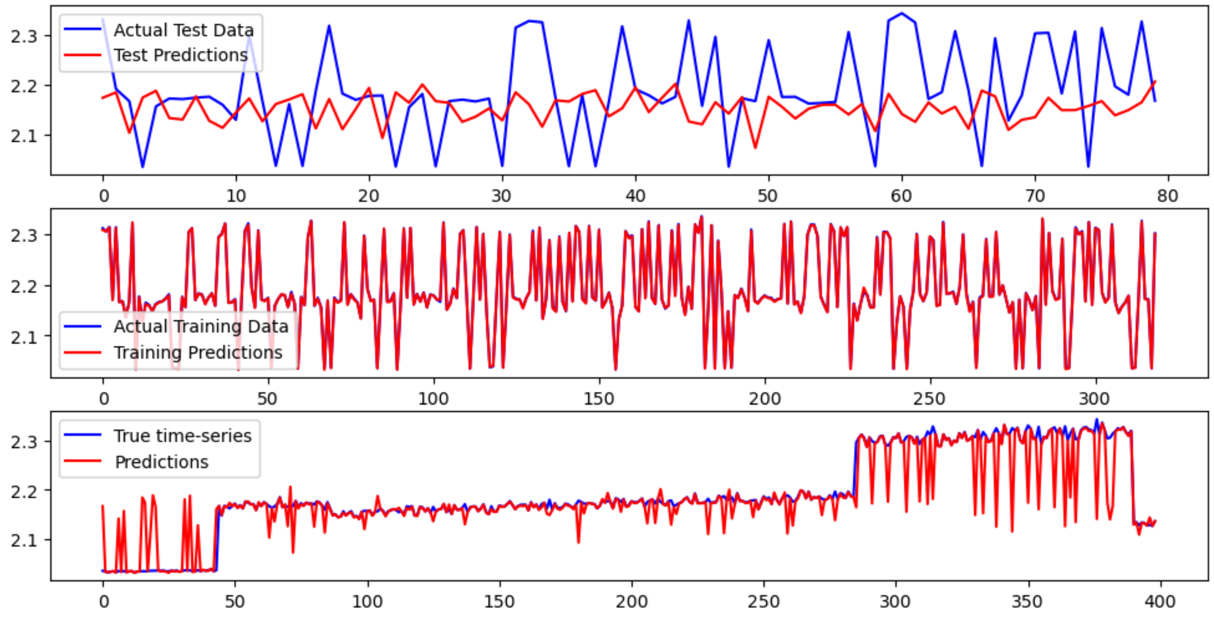


4/ You will find below the code of the creation and the training of a neural network dedicated to completion. Comment the code and describe what is the input of the neural network. Comment about the performance of the network. What do you suggest to improve its performance?

In [25]:
```
# Hyperparameters
input_size =len(y_df)-1 # size of the input signal
output_size = 1 # sampled point of signal for each input delta at time t
hidden_size = 300 # Number of neurons in hidden layer
num_epochs = 100 # Number of times the dataset is processed
learning_rate = 0.01 # How much to modify parameter estimates based on the l
weight_decay = 0 # Penalize large weights (i.e. the parameters)

# Define a class for a Feed Forward (FF) neural network
class SimpleNN(nn.Module):
    def __init__(self):
        # Initialize object of class SimpleNN
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size) # First layer
        self.relu = nn.ReLU() # Rectified Linear Unit (ReLU) activation func
        self.fch1 = nn.Linear(hidden_size, hidden_size) # Second layer
        self.fch2 = nn.Linear(hidden_size, hidden_size) # Third layer
        self.fc3 = nn.Linear(hidden_size, output_size) # Fourth and last lay

    def forward(self, x):
    # Forward input data along the NN
    # Sequence is output -> weights -> Activation function (ReLU) -> output
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fch1(out)
        out = self.relu(out)
        out = self.fch2(out)
        out = self.relu(out)
        out = self.fc3(out)
        return out

model = SimpleNN() # NN object
criterion = nn.MSELoss() # Define a loss function (mean square error)
optimizer = optim.Adam(model.parameters(), lr=learning_rate, weight_decay=we

X = np.eye(len(y_df) - 1) # Input is an identity matrix i.e. deltas at all t
y = y_df[1:] # Output is the time-series sampled at time t_i
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shu

# Define variables as PyTorch tensors
X_train = torch.FloatTensor(X_train)
X_test = torch.FloatTensor(X_test)
y_train = torch.FloatTensor(y_train).view(-1, 1) # Reshape
y_test = torch.FloatTensor(y_test).view(-1, 1)


batch_size = 32 # Number of samples to work through before updating parament
train_data = torch.utils.data.TensorDataset(X_train, y_train) # Create a tra
train_loader = torch.utils.data.DataLoader(dataset=train_data, batch_size=ba

# Training the model
for epoch in range(num_epochs):
    for inputs, targets in train_loader:
        outputs = model(inputs) # Forward pass
        loss = criterion(outputs, targets) # Compute loss
        optimizer.zero_grad() # Reset gradients
        loss.backward() # Backward pass, compute gradients of the loss wrt p
        optimizer.step() # Update parameters

    if (epoch + 1) % 10 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')

# Plot the original data and model predictions on the test set
model.eval()
with torch.no_grad():
    predictions_test = model(X_test)
    predictions_train = model(X_train)
    predictions_all = model(torch.FloatTensor(X))
```

```
# Plot the original data and model predictions on the test set
plt.figure(figsize=(12, 6))
plt.subplot(3, 1, 1)
plt.plot(y_test.numpy(), label='Actual Test Data', color='blue')
plt.plot(predictions_test.numpy(), label='Test Predictions', color='red')
plt.legend()

# Plot the original data and model predictions on the training set
plt.subplot(3, 1, 2)
plt.plot(y_train.numpy(), label='Actual Training Data', color='blue')
plt.plot(predictions_train.numpy(), label='Training Predictions', color='rec
plt.legend()

plt.subplot(3, 1, 3)
plt.plot(y, label='True time-series', color='blue')
plt.plot(predictions_all.numpy(), label='Predictions', color='red')
plt.legend()
plt.show()
```

```
Epoch [10/100], Loss: 0.0015
Epoch [20/100], Loss: 0.0000
Epoch [30/100], Loss: 0.0000
Epoch [40/100], Loss: 0.0000
Epoch [50/100], Loss: 0.0007
Epoch [60/100], Loss: 0.0000
Epoch [70/100], Loss: 0.0010
Epoch [80/100], Loss: 0.0000
Epoch [90/100], Loss: 0.0000
Epoch [100/100], Loss: 0.0000
```



The second property of a delay time-serie is that it is piecewise stationnary. We will see now how to enforce this property in the neural network using Total Variation (TV) regularization. TV is a technique commonly used in image processing and signal denoising. It's designed to preserve edges and reduce noise by penalizing the total variation or total change in the values of a signal. In the context of time-series data, such as in your neural network, applying TV regularization can help in smoothing the temporal variations while preserving important features.

5/ Use the following code to filter your time-series when everything is observed. What do you observe when you modify vlambda ?

In [26]:
```
y_df = df[0,:400]
```

In [36]:
```
# Load time series data
y = y_df
n = y.size

# Form second difference matrix.
e = np.ones((1, n))
D = scipy.sparse.spdiags(np.vstack((e, -2*e, e)), range(3), n-2, n)

# Set regularization parameter.
vlambda = 1

# Solve l1 trend filtering problem.
x = cp.Variable(shape=n)
obj = cp.Minimize(0.5 * cp.sum_squares(y - x)
                  + vlambda * cp.norm(D*x, 1) )
prob = cp.Problem(obj)

# ECOS and SCS solvers fail to converge before
# the iteration limit. Use CVXOPT instead.
prob.solve(solver=cp.CVXOPT, verbose=True)
print('Solver status: {}'.format(prob.status))

# Check for error.
if prob.status != cp.OPTIMAL:
    raise Exception("Solver did not converge!")

print("optimal objective value: {}".format(obj.value))
```

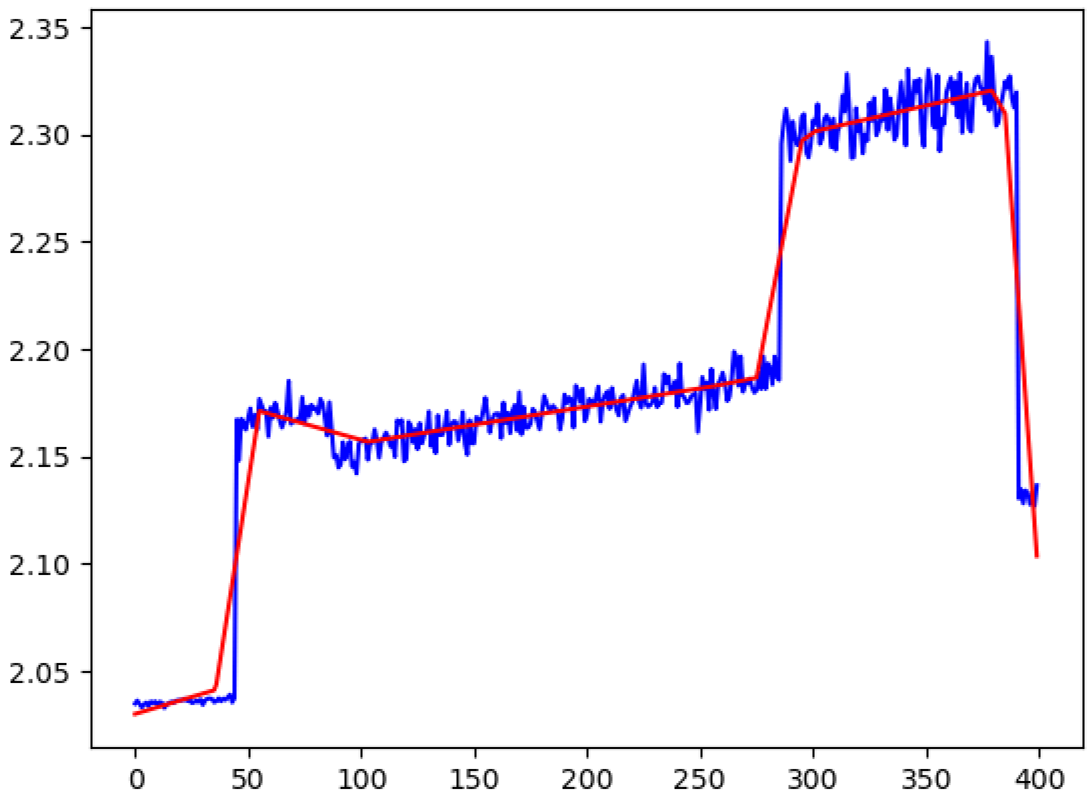
```
=============================================================================
===
                                     CVXPY
                                     v1.4.1
=============================================================================
===
(CVXPY) Jan 20 05:11:58 PM: Your problem has 400 variables, 0 constraints, a
nd 0 parameters.
(CVXPY) Jan 20 05:11:58 PM: It is compliant with the following grammars: DC
P, DQCP
(CVXPY) Jan 20 05:11:58 PM: (If you need to solve this problem multiple time
s, but with different data, consider using parameters.)
(CVXPY) Jan 20 05:11:58 PM: CVXPY will first compile your problem; then, it
will invoke a numerical solver to obtain a solution.
(CVXPY) Jan 20 05:11:58 PM: Your problem is compiled with the CPP canonicali
zation backend.
-----------------------------------------------------------------------------
---
                                  Compilation
-----------------------------------------------------------------------------
---
(CVXPY) Jan 20 05:11:58 PM: Compiling problem (target solver=CVXOPT).
(CVXPY) Jan 20 05:11:58 PM: Reduction chain: Dcp2Cone -> CvxAttr2Constr -> C
oneMatrixStuffing -> CVXOPT
(CVXPY) Jan 20 05:11:58 PM: Applying reduction Dcp2Cone
(CVXPY) Jan 20 05:11:58 PM: Applying reduction CvxAttr2Constr
(CVXPY) Jan 20 05:11:58 PM: Applying reduction ConeMatrixStuffing
(CVXPY) Jan 20 05:11:58 PM: Applying reduction CVXOPT
(CVXPY) Jan 20 05:11:58 PM: Finished problem compilation (took 4.082e-02 sec
onds).
-----------------------------------------------------------------------------
---
                                Numerical solver
-----------------------------------------------------------------------------
---
(CVXPY) Jan 20 05:11:58 PM: Invoking solver CVXOPT  to obtain a solution.
```

```
/home/gonzalo/anaconda3/envs/NN/lib/python3.11/site-packages/cvxpy/expressio
ns/expression.py:621: UserWarning:
This use of ``*`` has resulted in matrix multiplication.
Using ``*`` for matrix multiplication has been deprecated since CVXPY 1.1.
    Use ``*`` for matrix-scalar and vector-scalar multiplication.
    Use ``@`` for matrix-matrix and matrix-vector multiplication.
    Use ``multiply`` for elementwise multiplication.
This code path has been hit 4 times so far.

  warnings.warn(msg, UserWarning)
```

```
     pcost       dcost       gap    pres   dres   k/t
 0:  0.0000e+00 -1.0000e+00  1e+03  3e-01  2e+00  1e+00
 1: -2.7704e+00 -6.8401e-01  7e+01  5e-02  3e-01  2e+00
 2: -2.1146e-01 -5.2858e-04  3e+00  2e-03  1e-02  2e-01
 3: -1.2967e-02  5.6777e-02  1e+00  9e-04  5e-03  7e-02
 4:  5.5346e-02  6.7771e-02  3e-01  2e-04  1e-03  1e-02
 5:  8.4297e-02  8.6335e-02  6e-02  5e-05  3e-04  2e-03
 6:  8.8116e-02  8.8936e-02  3e-02  3e-05  2e-04  1e-03
 7:  9.0825e-02  9.1045e-02  1e-02  9e-06  5e-05  3e-04
 8:  9.1388e-02  9.1450e-02  7e-03  6e-06  3e-05  1e-04
 9:  9.1634e-02  9.1650e-02  4e-03  3e-06  2e-05  3e-05
10:  9.1960e-02  9.1960e-02  2e-03  1e-06  8e-06  8e-06
11:  9.2013e-02  9.2012e-02  5e-04  4e-07  2e-06  1e-06
12:  9.2058e-02  9.2058e-02  2e-04  1e-07  8e-07  4e-07
13:  9.2063e-02  9.2062e-02  5e-05  4e-08  2e-07  9e-08
14:  9.2065e-02  9.2065e-02  2e-05  1e-08  8e-08  3e-08
15:  9.2065e-02  9.2065e-02  5e-06  4e-09  2e-08  6e-09
16:  9.2065e-02  9.2065e-02  3e-07  2e-10  1e-09  4e-10
17:  9.2065e-02  9.2066e-02  2e-08  2e-11  1e-08  3e-11
Optimal solution found.
----------------------------------------------------------------------------
---
                                    Summary
----------------------------------------------------------------------------
---
(CVXPY) Jan 20 05:11:59 PM: Problem status: optimal
(CVXPY) Jan 20 05:11:59 PM: Optimal value: 9.207e-02
(CVXPY) Jan 20 05:11:59 PM: Compilation took 4.082e-02 seconds
(CVXPY) Jan 20 05:11:59 PM: Solver (including time spent in interface) took
1.451e+00 seconds
Solver status: optimal
optimal objective value: 0.09206546791267689
```

In [37]:
```python
plt.plot(y_df, label='True time-series', color='blue')
plt.plot(x.value, label='Predictions', color='red')
```

Out[37]:
```
[<matplotlib.lines.Line2D at 0x7f2143e60110>]
```

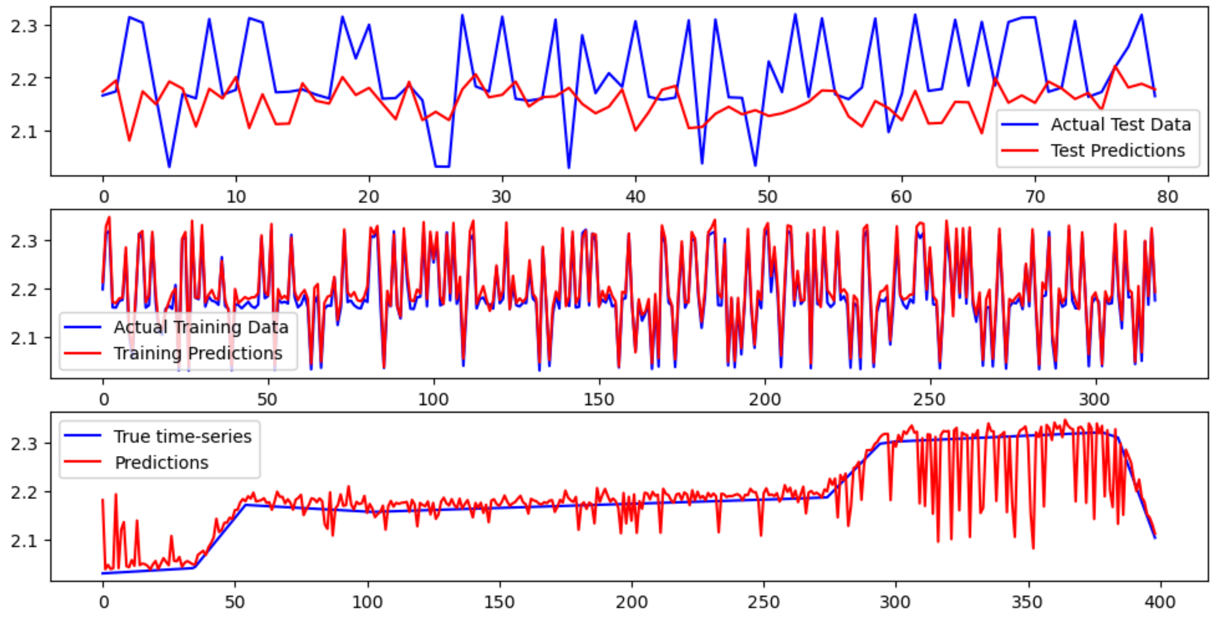


As lambda becomes lower the approximation overfits more to the signal. This is logical since the cost function is the difference of the prediction against the real signal plus a term that depends on the second derivative of the output. This means that as lambda is higher, more weight is given to the derivative so that the minimization forces the approximation to be softer (i.e. lower second derivative, so first derivative has slower variations, so the signal behaves more linear).

6/ Use the following code to smooth the prediction of the neural network. What are the performance of this new approach.

In [39]:
```
# Hyperparameters
input_size =len(y_df)-1 # size of the input signal
output_size = 1 # sampled point of signal for each input delta at time t
hidden_size = 300 # Number of neurons in hidden layer
num_epochs = 100 # Number of times the dataset is processed
learning_rate = 0.01 # How much to modify parameter estimates based on the l
weight_decay = 0 # Penalize large weights (i.e. the parameters)

# Define a class for a Feed Forward (FF) neural network
class SimpleNN(nn.Module):
    def __init__(self):
        # Initialize object of class SimpleNN
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size) # First layer
        self.relu = nn.ReLU() # Rectified Linear Unit (ReLU) activation func
        self.fch1 = nn.Linear(hidden_size, hidden_size) # Second layer
        self.fch2 = nn.Linear(hidden_size, hidden_size) # Third layer
        self.fc3 = nn.Linear(hidden_size, output_size) # Fourth and last lay
    def forward(self, x):
    # Forward input data along the NN
    # Sequence is output -> weights -> Activation function (ReLU) -> output
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fch1(out)
        out = self.relu(out)
        out = self.fch2(out)
        out = self.relu(out)
        out = self.fc3(out)
        return out

model = SimpleNN() # NN object
criterion = nn.MSELoss() # Define a loss function (mean square error)
optimizer = optim.Adam(model.parameters(), lr=learning_rate, weight_decay=we

X = np.eye(len(y_df) - 1) # Input is an identity matrix i.e. deltas at all t
y = x.value[1:] # Output is the time-series sampled at time t_i
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shu

# Define variables as PyTorch tensors
X_train = torch.FloatTensor(X_train)
X_test = torch.FloatTensor(X_test)
y_train = torch.FloatTensor(y_train).view(-1, 1) # Reshape
y_test = torch.FloatTensor(y_test).view(-1, 1)


batch_size = 32 # Number of samples to work through before updating parament
train_data = torch.utils.data.TensorDataset(X_train, y_train) # Create a tra
train_loader = torch.utils.data.DataLoader(dataset=train_data, batch_size=ba

# Training the model
for epoch in range(num_epochs):
    for inputs, targets in train_loader:
        outputs = model(inputs) # Forward pass
        loss = criterion(outputs, targets) # Compute loss
        optimizer.zero_grad() # Reset gradients
        loss.backward() # Backward pass, compute gradients of the loss wrt p
        optimizer.step() # Update parameters

    if (epoch + 1) % 10 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')

# Plot the original data and model predictions on the test set
model.eval()
with torch.no_grad():
    predictions_test = model(X_test)
    predictions_train = model(X_train)
    predictions_all = model(torch.FloatTensor(X))


# Plot the original data and model predictions on the test set
plt.figure(figsize=(12, 6))
plt.subplot(3, 1, 1)
plt.plot(y_test.numpy(), label='Actual Test Data', color='blue')
plt.plot(predictions_test.numpy(), label='Test Predictions', color='red')
plt.legend()

# Plot the original data and model predictions on the training set
plt.subplot(3, 1, 2)
plt.plot(y_train.numpy(), label='Actual Training Data', color='blue')
plt.plot(predictions_train.numpy(), label='Training Predictions', color='rec
plt.legend()

plt.subplot(3, 1, 3)
plt.plot(y, label='True time-series', color='blue')
plt.plot(predictions_all.numpy(), label='Predictions', color='red')
plt.legend()
plt.show()
```

```
Epoch [10/100], Loss: 0.0004
Epoch [20/100], Loss: 0.0000
Epoch [30/100], Loss: 0.0000
Epoch [40/100], Loss: 0.0000
Epoch [50/100], Loss: 0.0000
Epoch [60/100], Loss: 0.0000
Epoch [70/100], Loss: 0.0000
Epoch [80/100], Loss: 0.0000
Epoch [90/100], Loss: 0.0002
Epoch [100/100], Loss: 0.0004
```



7/ The code below is more efficient way of using total variation penalisation. Comment and explain the code. Compare with the performance of the different approaches.

In [54]:
```
# Parameters

D_in = input_size
e = np.ones((1, D_in))
D = scipy.sparse.spdiags(np.vstack((e, -2*e, e)), range(3), D_in-2, D_in)
lambd_ = 10
# Variables and Parameter cp
x = cp.Variable(D_in)
```

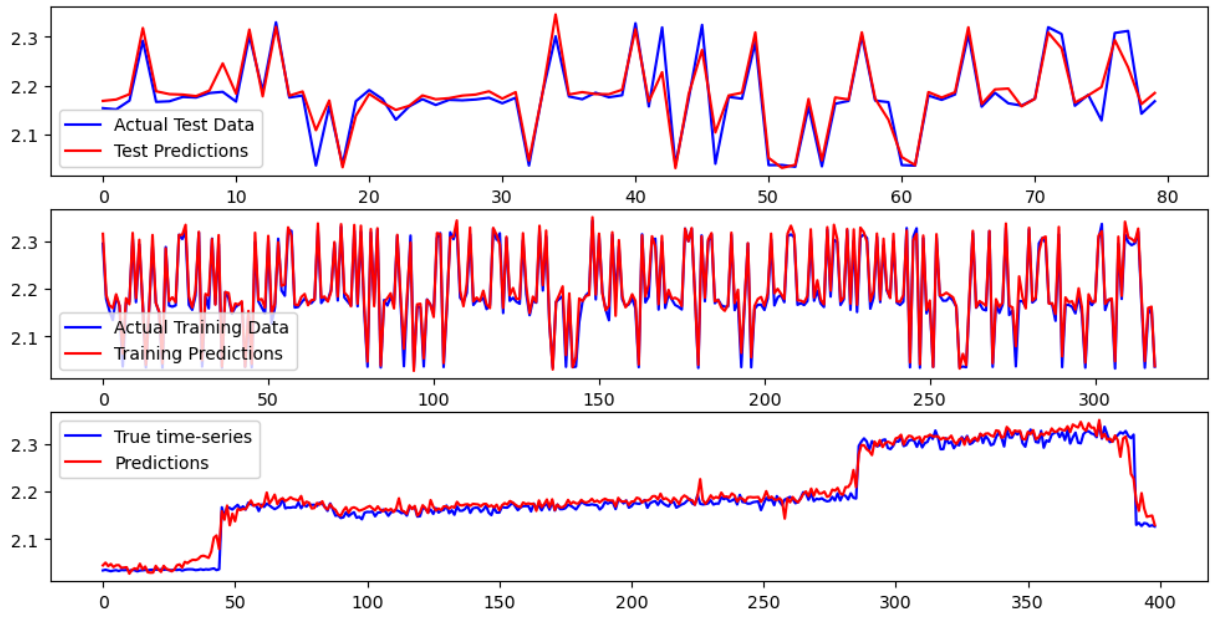
```
input = cp.Parameter(D_in)
input_model = cp.Parameter(D_in)


# Solve l1 trend filtering problem.
obj = cp.Minimize(0.5 * cp.sum_squares(input - x)+ lambd_ * cp.norm(D@x, 1)
prob = cp.Problem(obj)
TV_layer_prox = CvxpyLayer(prob, [input], [x]) # differentiable convex optim
```

In [41]:
```
# Generate synthetic time series data
# Replace this with your own dataset or data loading code

# Hyperparameters
output_size = 1  # Dimension of output data (predicted value)
hidden_size = 300  # Number of neurons in the hidden layer
num_epochs = 20
learning_rate = 0.01
weight_decay = 0
lambd_1 = 0.1

# Define a simple feedforward neural network
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.mask = nn.Linear(input_size, hidden_size)
        self.fc1 = nn.Linear(hidden_size, hidden_size)
        self.relu = nn.ReLU()
        self.fch1 = nn.Linear(hidden_size, hidden_size)
        self.fch2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        out = self.mask(x)
        out = self.fc1(out)
        out = self.relu(out)
        out = self.fch1(out)
        out = self.relu(out)
        out = self.fch2(out)
        out = self.relu(out)
        out = self.fc3(out)
        return out

# Create the model
model = SimpleNN()

# Loss and optimizer
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate, weight_decay=we

# Generate synthetic time series data

# Split the data into input (X) and output (y)
X = np.eye(input_size)  # One-hot encoding for time instant
y = y_df[:input_size]

# Split data into training (80%) and test (20%) sets
```

```python
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shu

# Convert data to PyTorch tensors
X_train = torch.FloatTensor(X_train)
X_test = torch.FloatTensor(X_test)
y_train = torch.FloatTensor(y_train).view(-1, 1)
y_test = torch.FloatTensor(y_test).view(-1, 1)

# Data loader for training data
batch_size = 32
train_data = torch.utils.data.TensorDataset(X_train, y_train)
train_loader = torch.utils.data.DataLoader(dataset=train_data, batch_size=ba

# Training the model
# At each iteration, two loss functions will be considered: The L2 norm of t
# output of the NN and the target, and the L2 norm between the filtered sigr
# So, at the end of the day, the neural network is minimizing a loss functic
# regularization term, since a component of the loss function forces the out

y_filter = torch.rand(input_size)
for epoch in range(num_epochs):
    for inputs, targets in train_loader:
        outputs = model(inputs) # Forward pass
        # Compute losses
        loss_1 = criterion(outputs, targets)
        loss_2 = criterion(y_filter,  model(torch.FloatTensor(X)).view(-1))
        loss = loss_1 + loss_2
        optimizer.zero_grad()
        loss.backward() # Backward pass
        optimizer.step()
        outputs = model(torch.FloatTensor(X))
        y_filter = TV_layer_prox(outputs.view(-1))[0] # Update filtered sigr

    if (epoch + 1) % 5 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')

# Plot the original data and model predictions on the test set
```

```
Epoch [5/20], Loss: 0.0516
/home/gonzalo/anaconda3/envs/NN/lib/python3.11/site-packages/diffcp/cone_pro
gram.py:295: UserWarning: Solved/Inaccurate.
  warnings.warn("Solved/Inaccurate.")
Epoch [10/20], Loss: 0.0113
Epoch [15/20], Loss: 0.0011
Epoch [20/20], Loss: 0.0006
```

In [42]:
```python
model.eval()
with torch.no_grad():
    predictions_test = model(X_test)
    predictions_train = model(X_train)
    predictions_all = model(torch.FloatTensor(X))


# Plot the original data and model predictions on the test set
plt.figure(figsize=(12, 6))
plt.subplot(3, 1, 1)
plt.plot(y_test.numpy(), label='Actual Test Data', color='blue')
plt.plot(predictions_test.numpy(), label='Test Predictions', color='red')
plt.legend()

# Plot the original data and model predictions on the training set
plt.subplot(3, 1, 2)
plt.plot(y_train.numpy(), label='Actual Training Data', color='blue')
plt.plot(predictions_train.numpy(), label='Training Predictions', color='rec
plt.legend()

plt.subplot(3, 1, 3)
plt.plot(y, label='True time-series', color='blue')
plt.plot(predictions_all.numpy(), label='Predictions', color='red')
plt.legend()
plt.show()
```



8/ (graph) graph neural networks can be used to do matrix completion. Adapt this code to our dataset.

```
In [ ]:
```