Lab: Nonlinear Least Squares for Modeling Materials

Nonlinear least squares (NLLS) is a widely-used method for modeling data. In NLLS, we wish to fit a model of the form,

```
yhat = g(x,w)
```

where `w` is a vector of paramters and `x` is the vector of predictors. We find `w` by minimizing a least-squares function

```
f(w) = \sum_i (y_i - g(x_i,w))^2
```

where the summation is over training samples `(x_i,y_i)`. This is similar to linear least-squares, but the function `g(x,w)` may not be linear in `w`. In general, this optimization has no closed-form expression. So numerical optimization must be used.

In this lab, we will implement gradient descent on NLLS in a problem of physical modeling of materials. Specifically, we will estimate parameters for expansion of copper as a function of temperature using a real dataset. In doing this lab, you will learn to:

- Set up a nonlinear least squares as an unconstrained optimization function
- Compute initial parameter estimates for a simple rational model
- Compute the gradients of the least squares objective
- Implement gradient descent for minimizing the objective
- Implement momentum gradient descent
- Visualize the convergence of the algorithm

We first import some key packages.

In [146]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import Ridge, LinearRegression
```

Load the Data

The NIST agency has an excellent nonlinear regression website (https://www.itl.nist.gov/div898/strd/nls/nls_main.shtml) that has several datasets for nonlinear regression problems. In this lab, we will use the data from a NIST study involving the thermal expansion of copper. The response variable is the coefficient of thermal expansion, and the predictor variable is temperature in degrees kelvin.

> Hahn, T., NIST (1979), Copper Thermal Expansion Study. (unpublished}

You can download the data as follows.

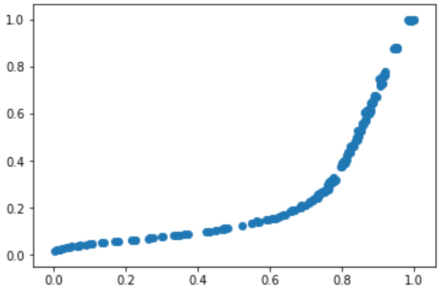
```
In [147]: url = 'https://itl.nist.gov/div898/strd/nls/data/LINKS/DATA/Hahn1.dat'
          df = pd.read_csv(url, skiprows=60, sep=' ',skipinitialspace=True, names=['x0',
          'y0','dummy'])
          df.head()
```

Out[147]:

	x0	y0	dummy
0	0.591	24.41	NaN
1	1.547	34.82	NaN
2	2.902	44.09	NaN
3	2.894	45.07	NaN
4	4.703	54.98	NaN

Extract the `x0` and `y0` into arrays. Rescale, `x0` and `y0` to values between `0` and `1` by dividing `x0` and `y0` by the maximum value. Store the scaled values in vectors x and y. The rescaling will help with the conditioning of the fitting. Plot, y vs. x.

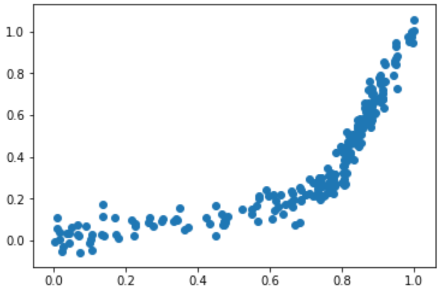
```
In [148]: # TODO
          x0 = df['x0'].values
          y0 = df['y0'].values
          x = x0/np.max(x0)
          y = y0/np.max(y0)
          plt.scatter(x, y)
          plt.show()
```



To make the problem a little more challenging, we will add some noise. Add random Gaussian noise with mean 0 and std. dev = 0.05 to y. Store the noisy results in yn. You can use the `np.random.normal()` function to add Gaussian noise. Plot yn vs. x.

In [149]:

```
# TODO
yn = y + np.random.randn(*y.shape)*0.05
plt.scatter(x, yn)
plt.show()
```



Split the data `(x,yn)` into training and test. Let `xtr,ytr` be training data and `xts,yts` be the test data. You can use the `train_test_split` function. Set `test_size=0.33` so that 1/3 of the samples are held out for test.

In [150]:

```
from sklearn.model_selection import train_test_split

# TODO
xtr, xts, ytr, yts = train_test_split(x, yn, test_size=0.33)
```

Initial Fit for a Rational Model

The NIST website (https://www.itl.nist.gov/div898/strd/nls/data/hahn1.shtml) suggests using a *rational* model of the form,

```
yhat = (a[0] + a[1]*x + ... + a[d]*x^d)/(1 + b[0]*x + ... + b[d-1]*x^d)
```

with d=3. The model parameters are `w = [a[0],...,a[d],b[0],...,b[d-1]]` so there are 2d+1 parameters total. Complete the function below that takes vectors `w` and `x` and predicts a set of values `yhat` using the above model.

In [151]:

```
def predict(w,x):

    # Get the length
    d = (len(w)-1)//2

    # TODO.  Extract a and b from w
    a = w[:d+1]
    b = w[d+1:]
    a = a[::-1]
    b = b[::-1]

    b = np.append(b, 1)

    # TODO.  Compute yhat.  You may use the np.polyval function3
    # But, remember you must flip the order the a and b
    yhat = np.polyval(a, x)/np.polyval(b, x)
    return yhat
```

When we fit with a nonlinear model, most methods only get convergence to a local minima. So, you need a good initial condition. For a rational model, one way to get is to realize that if:

```
y ~= (a[0] + a[1]*x + ... + a[d]*x^d)/(1 + b[0]*x + ... + b[d-1]*x^d)
```

Then:

```
y ~= a[0] + a[1]*x + ... + a[d]*x^d - b[0]*x*y + ... - b[d-1]*x^d*y.
```

So, we can solve for the the parameters `w = [a,b]` from linear regression of the predictors,

```
Z[i,:] = [ x[i], ... , x[i]**d, y[i]*x[i], ... , y[i}*x[i]**d ]
```

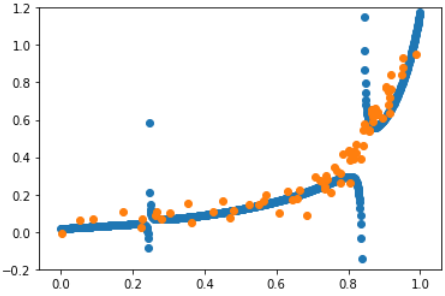
In [152]:

```
d = 3

# TODO.  Create the transformed feature matrix
xd = np.power(x.reshape(-1,1), (np.arange(d)+1).reshape(1,-1))
Z = np.hstack((xd, -xd*yn.reshape(-1,1)))

# TODO.  Fit with parameters with linear regression
regr = LinearRegression()
regr.fit(Z, yn)

# TODO
# Extract the parameters from regr.coef_ and regr.intercept_ and store the para
meter vector in winit
winit = np.hstack((regr.intercept_, regr.coef_))
```

Now plot the predicted values of the `yhat` vs. `x` using your estimated parameter `winit` for 1000 values `x` in `[0,1]`. On the same plot, plot `yts` vs. `xts`. You will see that you get a horrible fit.

In [153]:

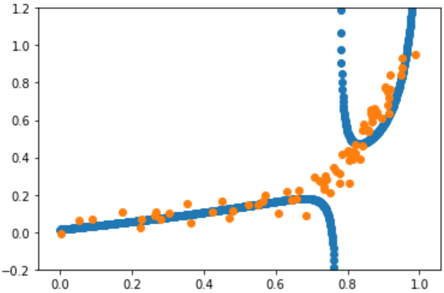
```
# TODO
xp = np.linspace(0,1, num=1000)
yhat = predict(winit, xp)

plt.scatter(xp, yhat)
plt.scatter(xts, yts)
plt.ylim(-0.2, 1.2)
plt.show()
```



The reason the previous fit is poor is that the denominator in `yhat` goes close to zero. To avoid this problem, we can use Ridge regression, to try to keep the parameters close to zero. Re-run the fit above with `Ridge` with `alpha = 1e-3`. You should see you get a reasonable, but not perfect fit.

In [154]:

```
# TODO.  Fit with parameters with linear regression
regr = Ridge(alpha=1e-3)
regr.fit(Z, yn)

# TODO
# Extract the parameters from regr.coef_ and regr.intercept_
winit_ridge = np.hstack((regr.intercept_, regr.coef_))

# TODO
# Plot the results as above.
yhat_ridge = predict(winit_ridge, xp)

plt.scatter(xp, yhat_ridge)
plt.scatter(xts, yts)
plt.ylim(-0.2, 1.2)
plt.show()
```



Creating a Loss Function

We can now use gradient descent to improve our initial estimate. Complete the following function to compute

```
f(w) = 0.5*\sum_i (y[i] - yhat[i])^2
```

and `fgrad`, the gradient of `f(w)`.

In [155]:

```
def feval(w,x,y):

    # TODO.  Parse w
    d = (len(w)-1)//2

    # TODO.  Extract a and b from w
    a = w[:d+1]
    b = w[d+1:]
    # if we use it in this way, not need to reverse

    # TODO.  Znum[i,j] = x[i]**j
    # 0, 1, 2, 3
    Znum = np.power(x.reshape(-1,1), (np.arange(d+1)).reshape(1,-1))

    # TODO.  Zden[i,j] = x[i]**(j+1)
    # 1, 2, 3
    Zden = np.power(x.reshape(-1,1), (np.arange(d)+1).reshape(1,-1))

    # TODO.  Compute yhat
    # Compute the numerator and denominator
    yhat = np.dot(Znum, a.reshape(-1,1))/(1 + np.dot(Zden, b.reshape(-1,1)))

    # TODO.  Compute loss
    f = 0.5*np.sum((y.reshape(-1,1)-yhat)**2)

    # TODO.  Compute gradients
    dyhat = (yhat-y.reshape(-1, 1))

    da_aid = ((Znum)/(1 + np.dot(Zden, b.reshape(-1,1)))).T
    da = np.dot (da_aid ,dyhat)

    db_aid = -(np.dot(Znum, a.reshape(-1,1))/((1 + np.dot(Zden, b.reshape(-1,1
)))**2)).T* Zden.T
    db = np.dot(db_aid, dyhat)

    da = da.reshape(-1)
    db = db.reshape(-1)
    fgrad = np.hstack((da, db))

    return f, fgrad
```

Test the gradient function:

- Take `w0=winit` and compute `f0,fgrad0 = feval(w0,xtr,ytr)`
- Take `w1` very close to `w0` and compute `f1,fgrad1 = feval(w1,xtr,ytr)`
- Verify that `f1-f0` is close to the predicted value based on the gradient.

In [156]:

```
# TODO
for i in range (winit.shape[0]):
    w0 = winit
    f0, fgrad0 = feval(w0,xtr,ytr)
    w1 = winit
    w1[i] = w0[i] + 1e-6
    f1, fgrad1 = feval(w1,xtr,ytr)
    grad_num = (f1-f0)/1e-6
    print("relative gradient error of {0} element of winit is {1}".format(i, np
.abs((grad_num-fgrad0[i])/grad_num)))
```

```
relative gradient error of 0 element of winit is 0.0003552886494037278
relative gradient error of 1 element of winit is 0.0002994248134771475
relative gradient error of 2 element of winit is 0.0002525393618567955
relative gradient error of 3 element of winit is 0.00021309594197117267
relative gradient error of 4 element of winit is 0.00130775273643947
relative gradient error of 5 element of winit is 0.0011007533267015153
relative gradient error of 6 element of winit is 0.0009266106551808558
```

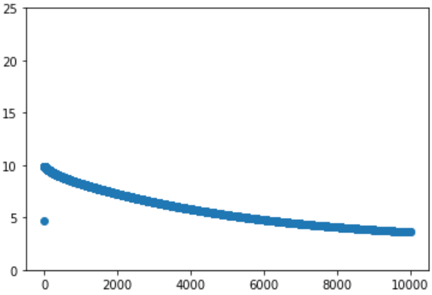
Implement gradient descent

We will now try to minimize the loss function with gradient descent. Using the function `feval` defined above, implement gradient descent. Run gradient descent with a step size of `alpha=1e-6` starting at `w=winit`. Run it for `nit=10000` iterations. Compute `fgd[it]`= the objective function on iteration `it`. Plot `fgd[it]` vs. `it`.

You should see that the training loss decreases, but it still hasn't converged after 10000 iterations.

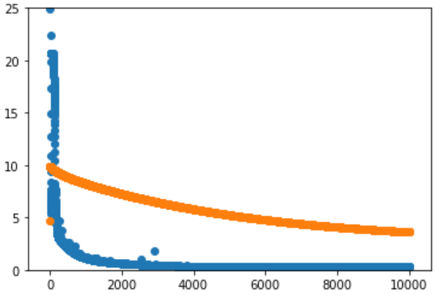
In [159]:

```
# TODO
nit = 10000
step = 1e-6
fgd = np.zeros(nit)

w = np.copy (winit)

for i in range (nit):
    fgd[i], fgrad = feval(w,xtr,ytr)
    w -= step*fgrad

plt.scatter(np.arange(nit), fgd)
plt.ylim(0, 25)
plt.show()
```



Now, try to get a faster convergence with adaptive step-size using the Armijo rule. Implement the gradient descent with adaptive step size. Let `fadapt[it]` be the loss function on iteration `it`. Plot `fadapt[it]` and `fgd[it]` vs. `it` on the same graph. You should see a slight improvement, but not much.

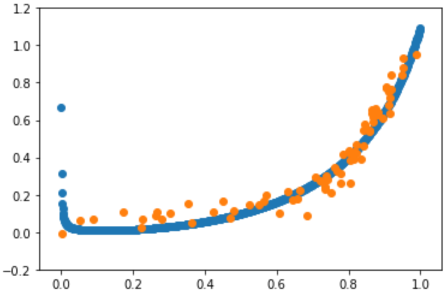
```
In [166]: # TODO
          nit = 10000
          step = 1e-6  # Initial step
          fadapt = np.zeros(nit)

          w = np.copy (winit)
          i = 0
          beta = 1.95

          while i<nit:

              fadapt[i], fgrad = feval(w,xtr,ytr)
              # fadapt[i] = f(w)
              w_previous = w
              w -= step*fgrad
              # w = w+1

              armijo = lambda x : feval(x, xtr, ytr)[0] <= fadapt[i] - 0.5*step*(np.sum (
          fgrad**2))

              if armijo(w) == True:
                  # pass armijo rule
                  step = step*1.2
                  i += 1
              else:
                  # reject armijo rule
                  step = step/2
                  w = w_previous
                  #reject this point and reiter!

          plt.scatter(np.arange(nit), fadapt)
          plt.scatter(np.arange(nit), fgd)

          plt.ylim(0, 25)
          plt.show()
```



Using he final estimate for `w` from the adaptive step-size plot the predicted values of the `yhat` vs. `x` usfor 1000 values `x` in `[0,1]`. On the same plot, plot `yhat` vs. `x` for the initial parameter `w=winit`. Also, plot `yts` vs. `xts`. You should see that gradient descent was able to improve the estimat slightly, although the initial estimate was not too bad.

In [161]:

```
# TODO
xp = np.linspace(0,1, num=1000)

yhat = predict(w, xp)

plt.scatter(xp, yhat)
plt.scatter(xts, yts)
plt.ylim(-0.2, 1.2)
plt.show()
```



Momentum Gradient Descent

This section is bonus.

One way to improve gradient descent is to use *momentum*. In momentum gradient descent, the update rule is:

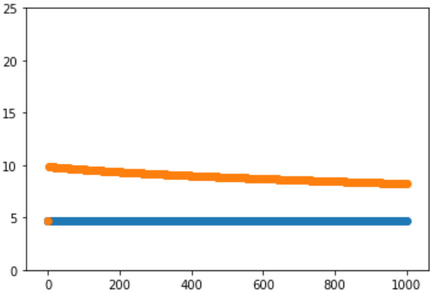
```
f, fgrad = feval(w,...)
z = beta*z + fgrad
w = w - step*z
```

This is similar to gradient descent, except that there is a second order term on the gradient. Implement this algorithm with `beta = 0.99` and `step=1e-5`. Compare the convergence of the loss function with gradient descent.

In [168]:

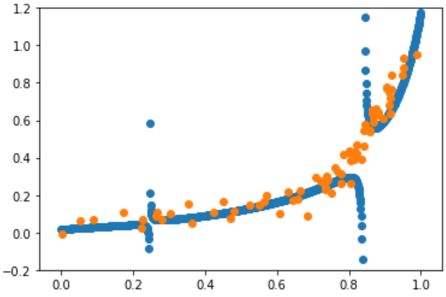
```
# TODO
nit = 1000
step = 1e-5
beta = 0.99
sdgm = np.zeros(nit)
v = 0
w = np.copy (winit)
print(w)
for i in range (nit):
    sdgm[i], fgrad = feval(w,xtr,ytr)
    z = beta*z + fgrad
    w =w - step*v

plt.scatter(np.arange(nit), sdgm)
plt.scatter(np.arange(nit), fgd[:nit])
plt.ylim(0, 25)
plt.show()
```

```
[ 1.76335663e-02  3.62098544e-04 -4.16391319e-01  4.67652991e-01
 -6.15592378e+00  9.56293536e+00 -4.34820936e+00]
```



In [163]:

```
# TODO
# plot yhat vs. x
xp = np.linspace(0,1, num=1000)

yhat = predict(w, xp)

plt.scatter(xp, yhat)
plt.scatter(xts, yts)
plt.ylim(-0.2, 1.2)

plt.show()
```



Beyond This Lab

In this lab, we have just touched at some of the ideas in optimization. There are several other important algorithms that you can explore:

- Levenberg-Marquardt (https://en.wikipedia.org/wiki/Levenberg%E2%80%93Marquardt_algorithm) method for non-linear least squares
- Newton's method
- More difficult non-linear least squares problems.