

# week08\_interactive\_MLP\_regression\_extended

March 29, 2023

## 1 MLP regression

Updated 29-Mar\_2023:

- Corrected the plot after the SGD-run to be the result from the SGD training.
- Added a run with 100 hidden nodes towards the end

```
[1]: import numpy as np
import matplotlib.pyplot as plt
import sklearn
```

### 1.1 Task

In weekly07 exercise set, we introduced a non-linear dataset and tried to fit a linear line to the the dataset.

In weekly08 we extended the dataset with polynomial features and made a better approximation. That was not so difficult as also the dataset was generated from a polynomial.

Here, the task is more challenging. We will try to approach the dataset with a simple feed-forward network with one hidden layer (a Multi-Layer Perceptron).

```
[2]: # The size of the dataset
N = 101
# The features of the dataset
xx = np.linspace(-5,5,N)
xx[:10]
```

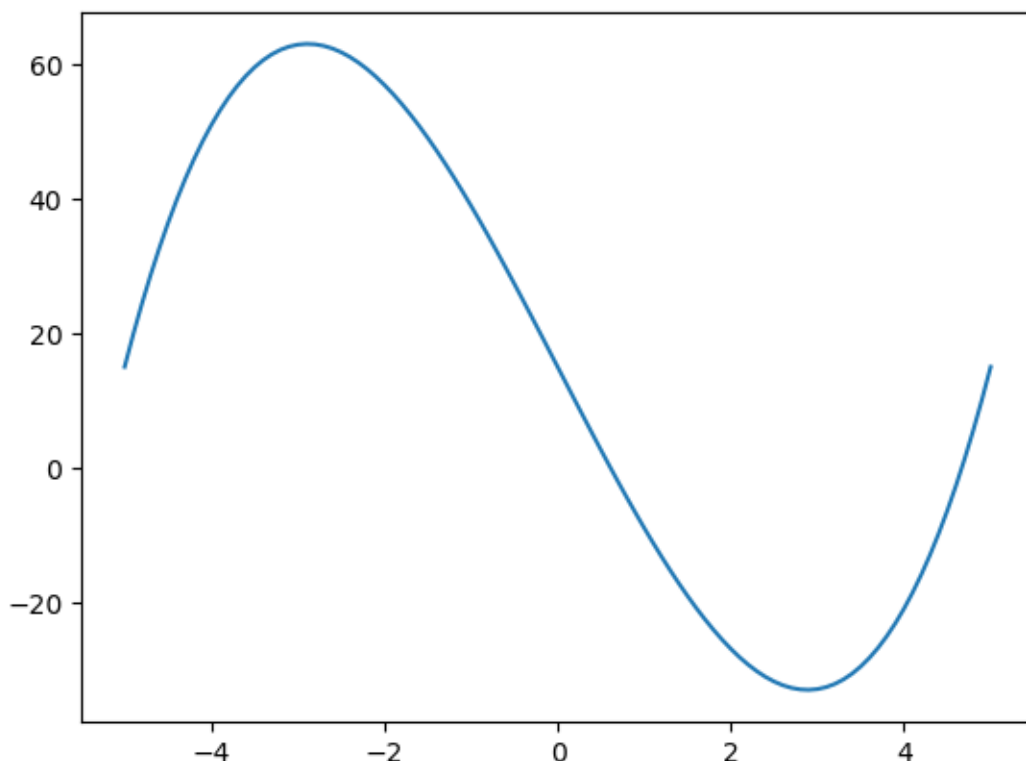
```
[2]: array([-5. , -4.9, -4.8, -4.7, -4.6, -4.5, -4.4, -4.3, -4.2, -4.1])
```

For the y-features, we start with a third degree polynomial, as in week07. We choose a polynomial which is further from linear.

```
[3]: polynom = xx**3 -25*xx +15
```

```
[4]: plt.plot(xx,polynom)
```

```
[4]: [ <matplotlib.lines.Line2D at 0x7f311a17e4c0>]
```

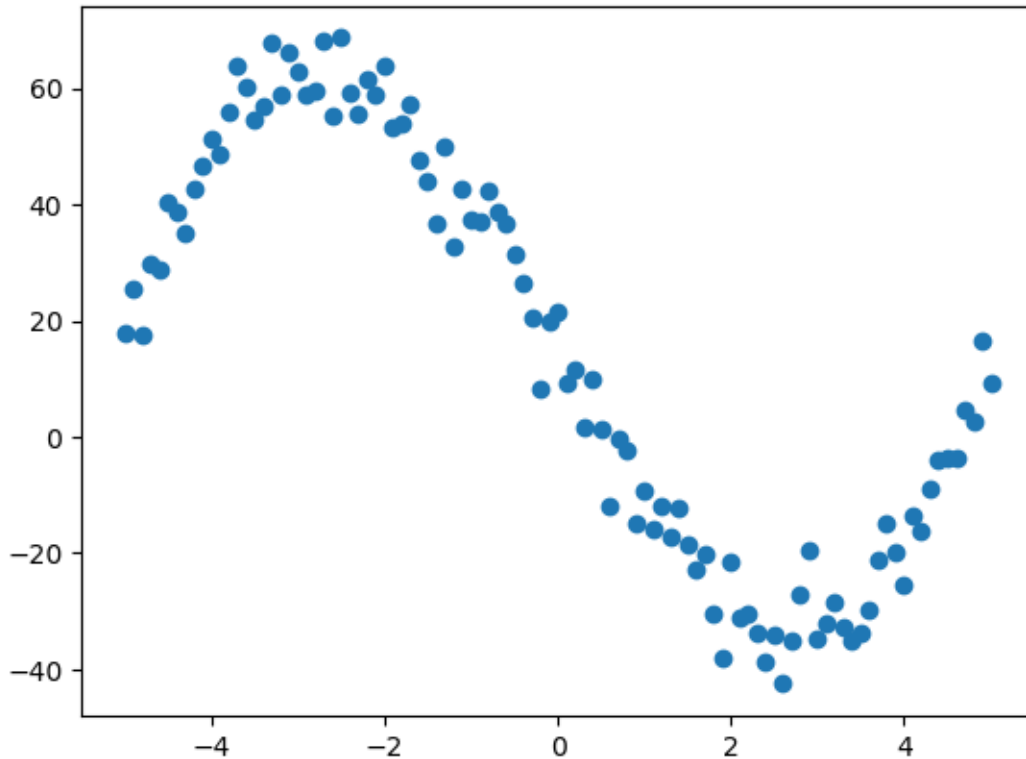


We add some “noise” to the smooth curve to generate the dataset. The underlying idea is that the smooth curve represents the function we are looking for, and that this is the best we can hope to learn. A solution which does better on the training material than the smooth curve is probably overfitted and will not generalize as well to new data as the smooth curve. We are using a normal distribution to generate noise. The numpy function `normal` will generate a vector of `size` many random points around `loc` from a distribution with standard deviation `scale`.

Observe the `rng` prefix. This fix the seed such that we get the same dataset each time we run the notebook.

```
[5]: from numpy.random import normal
     rng = np.random.default_rng(2023)
     t = polynom + rng.normal(loc=0, scale=5, size=N)
     plt.scatter(xx, t)
```

```
[5]: <matplotlib.collections.PathCollection at 0x7f3119877190>
```



Our training data now consist of pairs  $(xx[i], t[i])$ , where  $xx[i]$  is the datapoint and  $t[i]$  the target value. So far, both  $xx$  and  $t$  are vectors. Check their shapes, e.g., `xx.shape`. The goal is to make an implementation for linear regression which works with an arbitrary number of input variables and not just one. We will therefore transform  $xx$  to a matrix of dimension  $N \times m$  where each row represents one datapoint, and  $m$  is the number of input variables (or features). Check the shape of  $X$  after the transform.

```
[6]: X = xx.reshape(-1,1)
```

## 1.2 Part A: Linear regression from weekly07

We will implement our own linear regression model. Our aim is to find an approximate function that fits the data generated above.

Since we are dealing with only one input variable, we start with a simple linear function,  $f(x_1) = w_0 + w_1 x_1$ .

### 1.2.1 Exercise 7.1: MSE

We wonder if our  $f$  fits the data well, and what parameters will give us the best approximation. We will estimate this using the Mean Squared Error:

$$\frac{1}{N} \sum_{j=1}^N (t_j - \sum_{i=0}^m w_i x_{ji})^2$$

Write a function calculating MSE of our approximation.

```
[7]: # Vector form solution
def mse_2(x,y):
    return np.sum((x - y)**2) /x.shape[0]
```

```
[8]: # Vector form solution
def mse_3(x,y):
    return np.mean((x - y)**2)
```

### 1.2.2 Exercise 7.2: testing the MSE

To test our implementation, we can take the function  $f(x) = 0$  as a baseline and calculate the MSE for this  $f$ . Also calculate the Root Means Square Error which provides a more natural measure for how good the fit is.

**solution**

```
[9]: hypothesis = np.array([0 for x in xx])
mse = mse_2(hypothesis, t)
print("MSE: ", mse)
```

MSE: 1417.5522855383565

```
[10]: print("RMSE: ", np.sqrt(mse))
```

RMSE: 37.65039555620042

**end of solution** What is the MSE and RMSE for the theoretical distribution?

```
[11]: error = mse_2(t, polynom)
print("MSE: ", error)
```

MSE: 26.496434406097755

```
[12]: print("RMSE :", np.sqrt(error))
```

RMSE : 5.147468737748463

The RMSE is as expected. (cf., the standard deviation of the noise).

### 1.2.3 Exercise 7.3: Adding bias

We will implement linear regression with gradient descent and test it on the data. To make it simple, we will add a  $x_0 = 1$  to all our datapoints, and consider  $f(x_1) = w_0 + w_1x_1$  as  $f(x_0, x_1) = w_0x_0 + w_1x_1$ . Make a procedure that does this.

**solution** There are many ways that can be done in numpy. This is one of them.

```
[13]: def add_bias(X, bias=1):
        """X is a Nxm matrix: N datapoints, m features
        Return a Nx(m+1) matrix with added bias in position zero"""
```

```

N = X.shape[0]
biases = np.ones((N,1))*bias # Make a N*1 matrix of bias-s
# Concatenate the column of bias in front of the columns of X.
return np.concatenate((biases, X), axis = 1)

```

end of solution

#### 1.2.4 Exercise 7.4: Gradient Descent

We will implement the linear regression in a class as we did with the classifiers earlier. The fit method will run the gradient descent step a number of times to train the classifier. The predict method should take a matrix containing several data points and predict the outcome for all of them. Fill in the methods.

Assume that the matrix of training data are not extended with bias features. Hence, make adding bias a part of your methods.

After training there should be an attribute with learned coefficients (weights) which is applied by the predict method.

solution

```

[14]: class NumpyLinReg():

    def fit(self, X_train, t_train, eta = 0.1, epochs=10):
        """X_train is a Nxm matrix, N data points, m features
        t_train is a vector of length N,
        the targets values for the training data"""

        (N, m) = X_train.shape
        X_train = add_bias(X_train)

        self.weights = weights = np.zeros(m+1)

        for e in range(epochs):
            weights -= eta / N * X_train.T @ (X_train @ weights - t_train)

    def predict(self, X):
        """X is a Kxm matrix for some K>=1
        predict the value for each point in X"""
        Z = add_bias(X)
        return Z @ self.weights

```

end of solution

#### 1.2.5 Exercise 7.5: Train and test the model

Fit the model to the training data. Report the coefficients. Plot the line together with the observations. Calculate the RMSE. Is the result a better fit than the baseline constant function

$$f(x) = 0?$$

**solution**

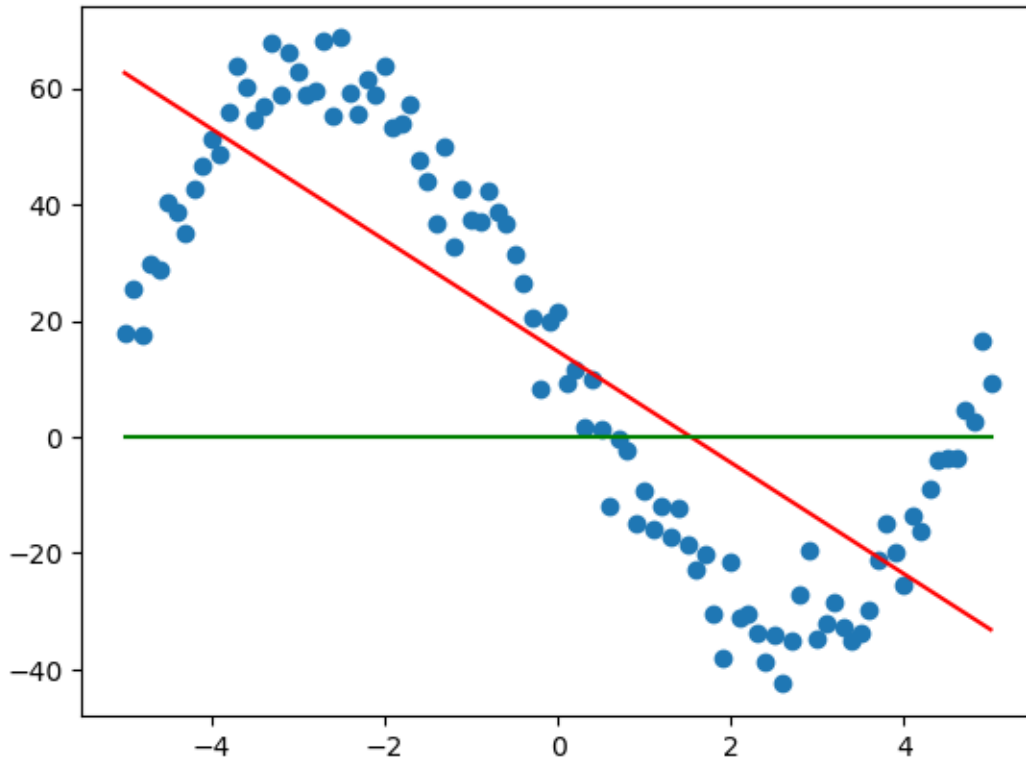
```
[15]: reg = NumpyLinReg()
      reg.fit(X,t, epochs=100)
      print("The coefficients: ", reg.weights)
```

The coefficients: [14.71749146 -9.58055389]

```
[16]: plt.plot(X, reg.weights[0]+reg.weights[1]*X, color="r")
      plt.plot(X, [0 for i in X], 'g')

      # The generated dataset
      plt.scatter(X, t)
```

[16]: <matplotlib.collections.PathCollection at 0x7f3119810160>



```
[17]: error = mse_2(t, add_bias(X) @ reg.weights)
      print("MSE: ", error)
```

MSE: 420.7466149361421

```
[18]: print("RMSE :", np.sqrt(error))
```

RMSE : 20.51210898313828

It has improved

end of solution

### 1.3 End of week 07

## 2 MLP regression

### 2.1 One item, one pass

We will add bias, and use bias=-1

```
[19]: Xb = add_bias(X, -1)
```

```
[20]: Xb[:10]
```

```
[20]: array([[ -1. , -5. ],
            [ -1. , -4.9],
            [ -1. , -4.8],
            [ -1. , -4.7],
            [ -1. , -4.6],
            [ -1. , -4.5],
            [ -1. , -4.4],
            [ -1. , -4.3],
            [ -1. , -4.2],
            [ -1. , -4.1]])
```

We will follow one datapoint through the model. First prediction, then update weights (backpropagation).

We have chosen observation 25.

```
[21]: obs = 25
      Xb[25]
```

```
[21]: array([ -1. , -2.5])
```

```
[22]: t[25]
```

```
[22]: 68.64520507640746
```

```
[ ]:
```

When you start training you should initialize the weights to random numbers small in absolute value.

We started to train the networks in this way. We trained the network for small period. The weights in the matrices  $V$ ,  $W$ , are the weights after this short training period. They are chosen as an example of weight matrices that may occur (early) during training.

The current weights in matrix form.

```
[23]: V = np.array([[0.4, 0.2, 0.9],
                  [-1.5, 2.0, -2.0]])
V
```

```
[23]: array([[ 0.4,  0.2,  0.9],
          [-1.5,  2. , -2. ]])
```

```
[24]: W = np.array([6, 3.0, -3, 4.2]).reshape(-1,1)
W
```

```
[24]: array([[ 6. ],
          [ 3. ],
          [-3. ],
          [ 4.2]])
```

```
[ ]:
```

We have to be aware whether we represent our observation as a vector of `shape=(2,)` or a row matrix of `shape=(1,2)`. Either of them can work. We have to be certain regarding the chosen format, however.

To use a row matrix will have two advantages:

- We reuse the code for batch processing
- One of the steps is most easily done by matrices.

Hence, we will use a row matrix from the beginning.

```
[25]: obs = 25 # The observation we will consider
x_obs = Xb[obs]
# A vector
x_obs
```

```
[25]: array([-1. , -2.5])
```

```
[26]: X_obs = Xb[obs:obs+1]
# A row matrix
X_obs
```

```
[26]: array([[ -1. , -2.5]])
```

```
[27]: print(x_obs.shape)
print(X_obs.shape)
```

```
(2,)
(1, 2)
```

```
[ ]:
```

### 2.1.1 Forward

#### Hidden layer

```
[28]: hidden_in = X_obs @ V
      hidden_in
```

```
[28]: array([[ 3.35, -5.2 ,  4.1 ]])
```

```
[29]: def logistic(x):
      return 1/(1+np.exp(-x))
```

```
[30]: hidden_out = logistic(hidden_in)
      hidden_out
```

```
[30]: array([[0.96610484, 0.0054863 , 0.9836975 ]])
```

```
[ ]:
```

#### Final layer

```
[31]: hidden_out = add_bias(hidden_out, -1)
      hidden_out
```

```
[31]: array([[ -1.          ,  0.96610484,  0.0054863 ,  0.9836975 ]])
```

```
[32]: final_in = hidden_out @ W
      final_in
```

```
[32]: array([[1.01338511]])
```

```
[33]: Y_obs = final_out = final_in
      final_out
```

```
[33]: array([[1.01338511]])
```

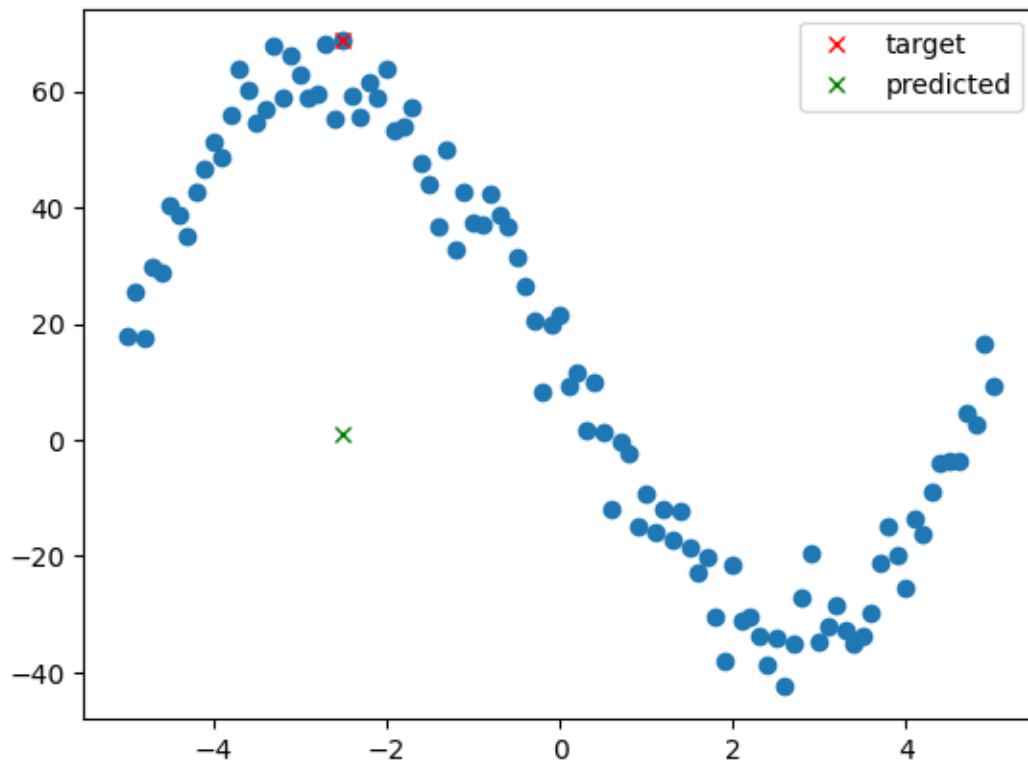
```
[ ]:
```

#### The current weights

```
[34]: # The generated dataset
      plt.scatter(X, t)
      plt.plot(X[obs], t[obs], 'rx', label='target')
      plt.plot(X[obs], Y_obs, 'gx', label='predicted')
```

```
plt.legend()
```

[34]: <matplotlib.legend.Legend at 0x7f31197a2910>

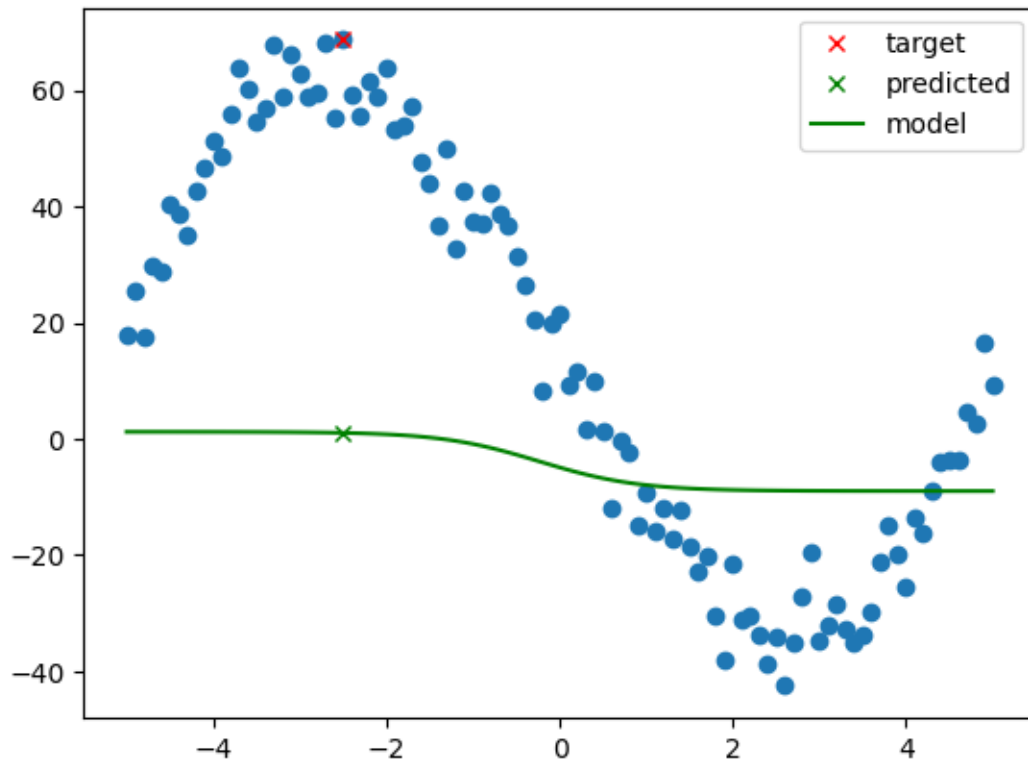


[ ]:

We can also check the prediction for other points.

```
[35]: # The generated dataset
plt.scatter(X, t)
plt.plot(X[obs], t[obs], 'rx', label='target')
plt.plot(X[obs], Y_obs, 'gx', label='predicted')
Y = add_bias(logistic(Xb @ V), -1) @ W
plt.plot(X, Y, 'g', label='model')
plt.legend()
```

[35]: <matplotlib.legend.Legend at 0x7f311972cfa0>



[ ]:

### 2.1.2 Backpropagation

[36]: `t[obs]`

[36]: 68.64520507640746

We use  $0.5 * \text{MSE}$  as Loss

[37]: `loss = 0.5*(Y_obs-t[obs])**2`  
`loss`

[37]: `array([[2287.03153575]])`

[ ]:

### The last layer

[38]: `final_delta = (Y_obs - t[obs])`  
`final_delta`

```
[38]: array([[ -67.63181996]])
```

We have what is needed for updating the  $W$ s. But we have to wait until we have found the delta terms for the first layer.

Observe that  $A = \text{hidden\_out}$  has shape (1, 4), a row matrix, while  $W$  has shape (4, 1), a column matrix. Hence we have to reshape  $A$ .

The form `reshape((-1,1))` will reshape a matrix of shape (m, n) to matrix of shape (m\*n, 1).

```
[39]: """
      W -= eta * final_delta * hidden_out.reshape((-1,1))
      """
```

```
[39]: '\nW -= eta * final_delta * hidden_out.reshape((-1,1))\n'
```

```
[40]: hidden_out.shape
```

```
[40]: (1, 4)
```

```
[41]: hidden_out.reshape((-1,1))
```

```
[41]: array([[ -1.          ],
          [ 0.96610484 ],
          [ 0.0054863  ],
          [ 0.9836975  ]])
```

```
[ ]:
```

### The hidden layer

```
[42]: hidden_out_delta = final_delta[0,0] * W.T
      hidden_out_delta
```

```
[42]: array([[ -405.79091978, -202.89545989,  202.89545989, -284.05364384]])
```

```
[43]: hidden_out_delta = final_delta @ W.T
      hidden_out_delta
```

```
[43]: array([[ -405.79091978, -202.89545989,  202.89545989, -284.05364384]])
```

```
[ ]:
```

We remove the part which corresponds to the bias in the hidden layer when we are going backwards through the hidden layer. It is important to remember where you put the bias - first or last in the vectors.

```
[44]: a = hidden_out[:,1:]
      hidden_in_delta = a * (1 - a) * hidden_out_delta[:,1:]
```

```
hidden_in_delta
```

```
[44]: array([[ -6.64407195,  1.10703809, -4.55529099]])
```

```
[ ]:
```

**The update** First, we repeat the current weights

```
[45]: V = np.array([[0.4, 0.2, 0.9],  
                  [-1.5, 2.0, -2.0]])  
V
```

```
[45]: array([[ 0.4,  0.2,  0.9],  
          [-1.5,  2. , -2. ]])
```

```
[46]: W = np.array([6, 3.0, -3, 4.2]).reshape(-1,1)  
W
```

```
[46]: array([[ 6. ],  
          [ 3. ],  
          [-3. ],  
          [ 4.2]])
```

We choose a learning rate, eta.

```
[47]: eta = 0.01
```

The updated weights

```
[48]: W -= eta * final_delta * hidden_out.reshape((-1,1))  
W
```

```
[48]: array([[ 5.3236818 ],  
          [ 3.65339428],  
          [-2.99628952],  
          [ 4.86529252]])
```

```
[49]: V -= eta * X_obs.T @ hidden_in_delta  
V
```

```
[49]: array([[ 0.33355928,  0.21107038,  0.85444709],  
          [-1.6661018 ,  2.02767595, -2.11388227]])
```

```
[ ]:
```

## 2.2 Stochastic gradient descent

We gather the code from the forward phase and backpropagation for one observation in one cell. To see that we have not copied everything correctly, we reset the weights, run everything, and see that the updated weights are the same.

```
[50]: V = np.array([[0.4, 0.2, 0.9],
                  [-1.5, 2.0, -2.0]])
W = np.array([6, 3.0, -3, 4.2]).reshape(-1,1)

eta = 0.01
obs=25
X_obs = Xb[obs:obs+1]

hidden_in = X_obs @ V

hidden_out = logistic(hidden_in)

hidden_out = add_bias(hidden_out, -1)

final_in = hidden_out @ W

Y_obs = final_out = final_in

### Backprop

t[obs]

final_delta = (Y_obs - t[obs])

# hidden_out_delta = final_delta @ W.T

hidden_out_delta = final_delta[0,0] * W.T

a = hidden_out[:,1:]

hidden_in_delta = a * (1 - a) * hidden_out_delta[:,1:]

W -= eta * final_delta * hidden_out.reshape((-1,1))
print("W: ")
print(W)

V -= eta * X_obs.T @ hidden_in_delta
print("\nV: ")
print(V)
```

```
W:
[[ 5.3236818 ]
 [ 3.65339428]
 [-2.99628952]
 [ 4.86529252]]
```

```
V:
[[ 0.33355928  0.21107038  0.85444709]
 [-1.6661018   2.02767595 -2.11388227]]
```

```
[ ]:
```

We turn did into stochastic gradient descent by running the code repeatedly. For each iterations, we choose a random point from the dataset.

This is an example of training with stochastic gradient descent.

We will choose a smaller learning rate than in the step-by-step example. Try to run it with various numbers of iterations and various learning rates.

```
[51]: eta = 0.0001
iterations = 1000000

V = np.array([[0.4, 0.2, 0.9],
              [-1.5, 2.0, -2.0]])
W = np.array([6, 3.0, -3, 4.2]).reshape(-1,1)

for i in range(iterations):
    obs = rng.integers(101, size=1)[0]

    X_obs = Xb[obs:obs+1]

    hidden_in = X_obs @ V

    hidden_out = logistic(hidden_in)

    hidden_out = add_bias(hidden_out, -1)

    final_in = hidden_out @ W

    Y_obs = final_out = final_in

    ### Backprop

    t[obs]

    final_delta = (Y_obs - t[obs])
```

```

# hidden_out_delta = final_delta @ W.T

hidden_out_delta = final_delta[0,0] * W.T

a = hidden_out[:,1:]

hidden_in_delta = a * (1 -a) * hidden_out_delta[:,1:]

W -= eta * final_delta * hidden_out.reshape((-1,1))

V -= eta * X_obs.T @ hidden_in_delta

```

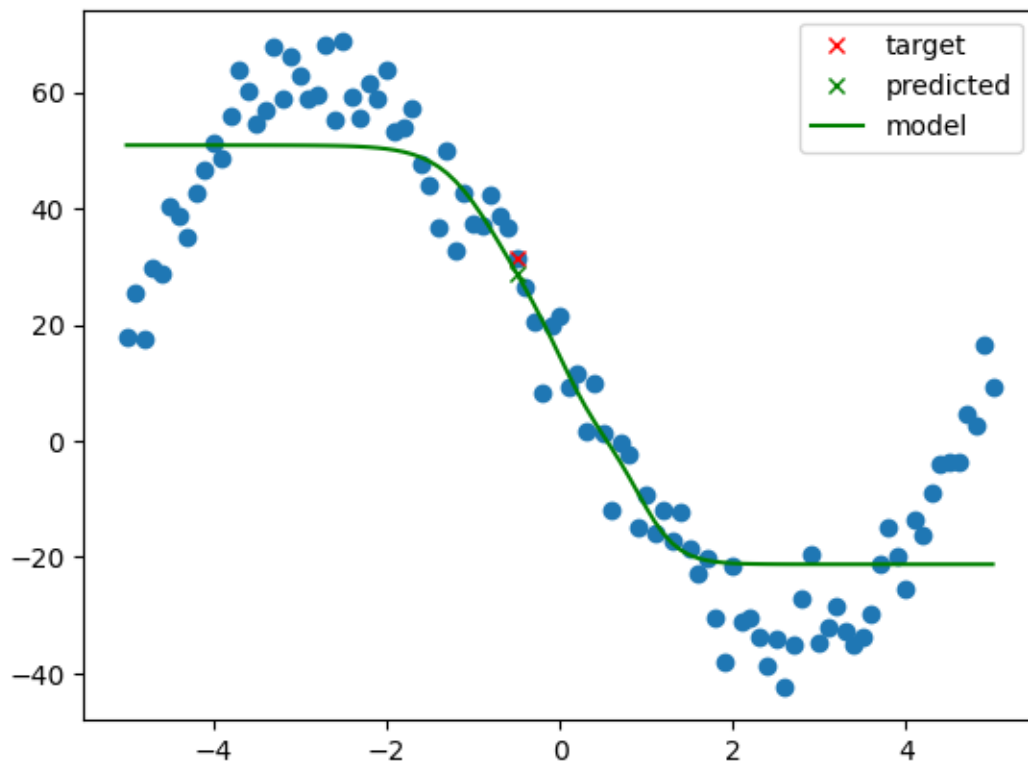
Plotting the result and evaluating.

```

[52]: plt.scatter(X, t)
plt.plot(X[obs], t[obs], 'rx', label='target')
plt.plot(X[obs], Y_obs, 'gx', label='predicted')
Y = add_bias(logistic(Xb @ V), -1) @ W
plt.plot(X,Y, 'g', label='model')
plt.legend()

```

[52]: <matplotlib.legend.Legend at 0x7f311968e640>



```
[53]: error = mse_2(t, Y[:,0])
      print("MSE: ", error)
```

MSE: 151.28694747633156

```
[54]: print("RMSE :", np.sqrt(error))
```

RMSE : 12.299875913046098

**Comment** We see that the approximation is better than the linear approximation, but it is not quite there. We also see that the predictions look suspiciously like a logistic curve. (Is this an inductive bias from the model?)

```
[ ]:
```

## 2.3 The number of hidden nodes

### 2.3.1 Batch processing

We will need whether we can get a better approximation with a larger number of nodes in the hidden layer. To not have to write new code, we switch to batch processing and the code we already have (in mandatory 2) where the number of hidden nodes is parameter

```
[55]: class MLPBinaryLinReg():
      """A multi-layer neural network with one hidden layer"""

      def __init__(self, bias=-1, dim_hidden = 6):
          """Initialize the hyperparameters"""
          self.bias = bias
          self.dim_hidden = dim_hidden

          def logistic(x):
              return 1/(1+np.exp(-x))
          self.activ = logistic

          def logistic_diff(y):
              return y * (1 - y)
          self.activ_diff = logistic_diff

      def fit(self, X_train, t_train, eta=0.001, epochs = 100):
          """Initialize the weights. Train *epochs* many epochs.

          X_train is a Nxm matrix, N data points, m features
          t_train is a vector of length N of targets values for the training
          data,
          where the values are 0 or 1.
          """
          self.eta = eta
```

```

T_train = t_train.reshape(-1,1)

dim_in = X_train.shape[1]
dim_out = T_train.shape[1]

# Itilaize the wights
self.weights1 = (np.random.rand(
    dim_in + 1,
    self.dim_hidden) * 2 - 1)/np.sqrt(dim_in)
self.weights2 = (np.random.rand(
    self.dim_hidden+1,
    dim_out) * 2 - 1)/np.sqrt(self.dim_hidden)
X_train_bias = add_bias(X_train, self.bias)

for e in range(epochs):
    # One epoch
    hidden_outs, outputs = self.forward(X_train_bias)
    # The forward step
    out_deltas = (outputs - T_train)
    # The delta term on the output node
    hiddenout_diffs = out_deltas @ self.weights2.T
    # The delta terms at the output of the jidden layer
    hiddenact_deltas = (hiddenout_diffs[:, 1:] *
        self.activ_diff(hidden_outs[:, 1:]))
    # The deltas at the input to the hidden layer
    self.weights2 -= self.eta * hidden_outs.T @ out_deltas
    self.weights1 -= self.eta * X_train_bias.T @ hiddenact_deltas
    # Update the weights

def forward(self, X):
    """Perform one forward step.
    Return a pair consisting of the outputs of the hidden_layer
    and the outputs on the final layer"""
    hidden_activations = self.activ(X @ self.weights1)
    hidden_outs = add_bias(hidden_activations, self.bias)
    outputs = hidden_outs @ self.weights2
    return hidden_outs, outputs

def predict(self, X):
    """Predict the class for the members of X"""
    Z = add_bias(X, self.bias)
    forw = self.forward(Z)[1]
    return forw[:, 0]

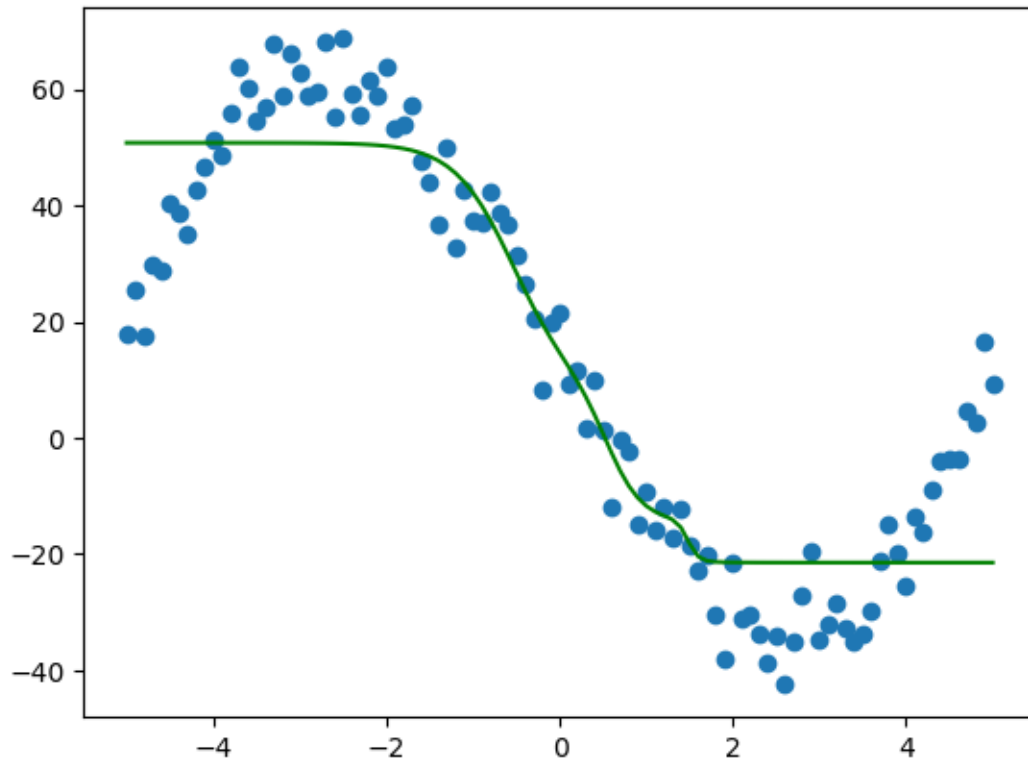
```

**3 hidden nodes** First, we try to train with three hidden nodes, and compare to the stochastic gradient descent algorithm.

```
[56]: regr = MLPBinaryLinReg(bias=-1, dim_hidden=3)
regr.fit(X, t, epochs=50000, eta=0.001)
Y=regr.predict(X)

plt.scatter(X, t)
plt.plot(X, Y, 'g')
```

```
[56]: [<matplotlib.lines.Line2D at 0x7f31195e0f70>]
```



```
[57]: error = mse_2(t, Y)
print("MSE: ", error)
```

MSE: 150.78797370219

```
[58]: print("RMSE :", np.sqrt(error))
```

RMSE : 12.279575469135324

We get comparable results to the s.g.d.

```
[ ]:
```

### 2.3.2 12 hidden nodes

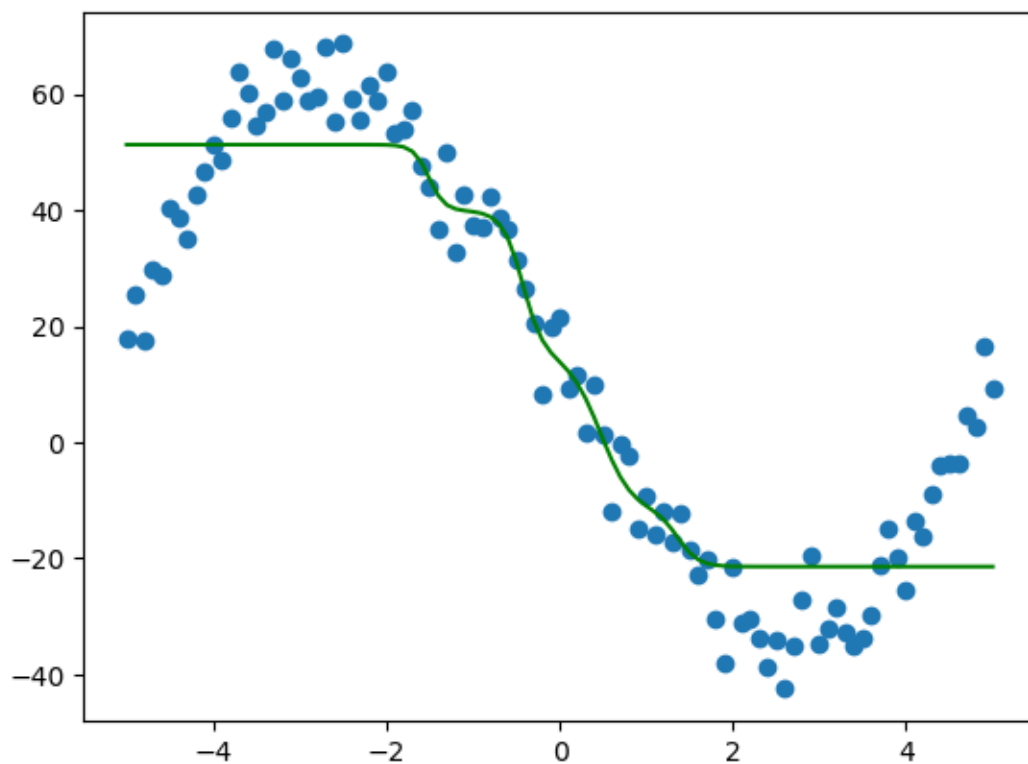
```
[59]: %%time

regr = MLPBinaryLinReg(bias=-1, dim_hidden=12)
regr.fit(X, t, epochs=50000, eta=0.0001)
Y=regr.predict(X)

plt.scatter(X, t)
plt.plot(X, Y, 'g')
```

CPU times: user 3.55 s, sys: 7.85 ms, total: 3.56 s  
Wall time: 3.58 s

```
[59]: [<matplotlib.lines.Line2D at 0x7f31195617f0>]
```



```
[60]: error = mse_2(t, Y)
print("MSE: ", error)
```

MSE: 147.87465689731155

```
[61]: print("RMSE :", np.sqrt(error))
```

RMSE : 12.160372399614724

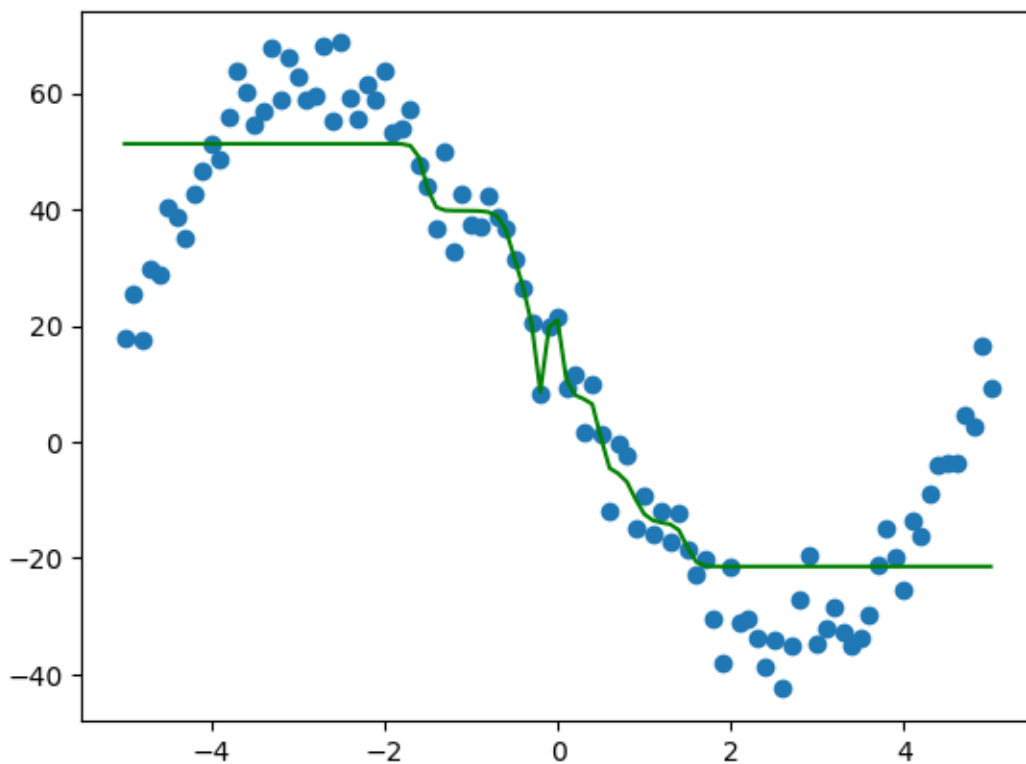
```
[62]: %%time

regr = MLPBinaryLinReg(bias=-1, dim_hidden=12)
regr.fit(X, t, epochs=500000, eta=0.0001)
Y=regr.predict(X)

plt.scatter(X, t)
plt.plot(X, Y, 'g')
```

CPU times: user 35.1 s, sys: 5.91 ms, total: 35.1 s  
Wall time: 35.3 s

```
[62]: [<matplotlib.lines.Line2D at 0x7f31194d7730>]
```



```
[63]: error = mse_2(t, Y)
print("MSE: ", error)
```

MSE: 145.13554328707946

```
[64]: print("RMSE :", np.sqrt(error))
```

RMSE : 12.04722139279757

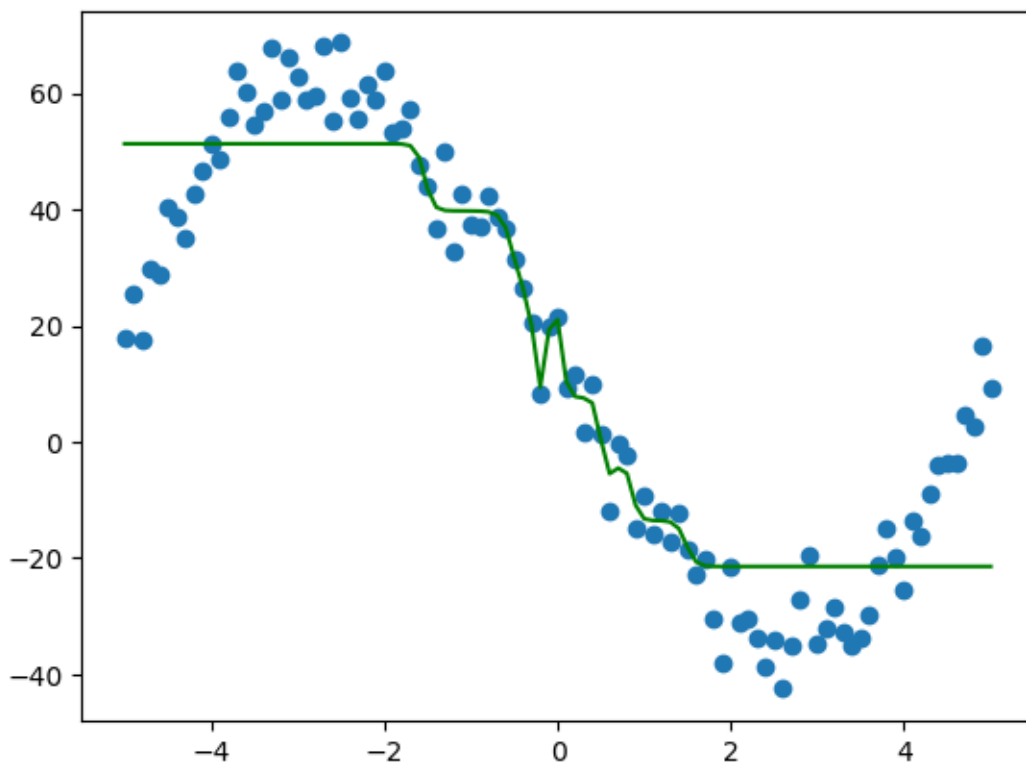
```
[65]: %%time

regr = MLPBinaryLinReg(bias=-1, dim_hidden=12)
regr.fit(X, t, epochs=50000, eta=0.001)
Y=regr.predict(X)

plt.scatter(X, t)
plt.plot(X, Y, 'g')
```

CPU times: user 3.63 s, sys: 7.93 ms, total: 3.64 s  
Wall time: 3.65 s

```
[65]: [<matplotlib.lines.Line2D at 0x7f3119445dc0>]
```



```
[66]: error = mse_2(t, Y)
print("MSE: ", error)
```

MSE: 144.81100828550618

```
[67]: print("RMSE :", np.sqrt(error))
```

RMSE : 12.033744566239811

```
[68]: %%time

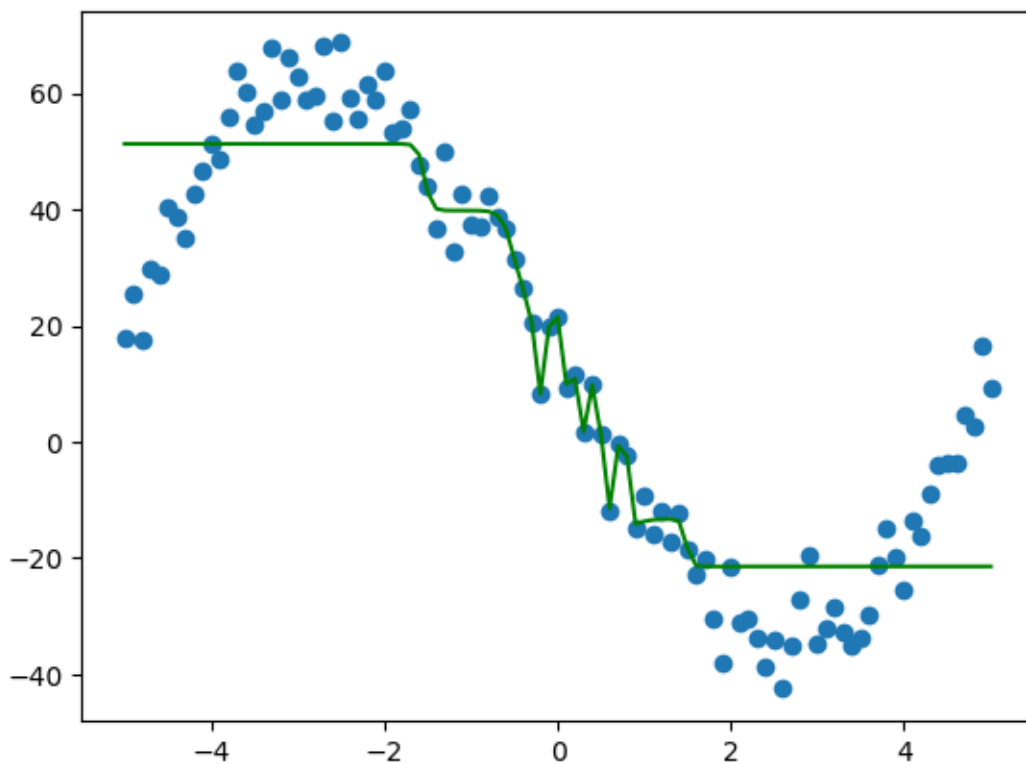
regr = MLPBinaryLinReg(bias=-1, dim_hidden=12)
regr.fit(X, t, epochs=500000, eta=0.001)
Y=regr.predict(X)

plt.scatter(X, t)
plt.plot(X, Y, 'g')
```

CPU times: user 35.3 s, sys: 9.52 ms, total: 35.3 s

Wall time: 35.5 s

```
[68]: [<matplotlib.lines.Line2D at 0x7f311961c280>]
```



```
[69]: error = mse_2(t, Y)
print("MSE: ", error)
```

MSE: 143.33236124354417

```
[70]: print("RMSE :", np.sqrt(error))
```

```
RMSE : 11.972149399483126
```

```
[ ]:
```

## 2.4 Added after class

What happens if we increase the number of hidden nodes further?

### 2.4.1 100 hidden nodes

```
[71]: %%time

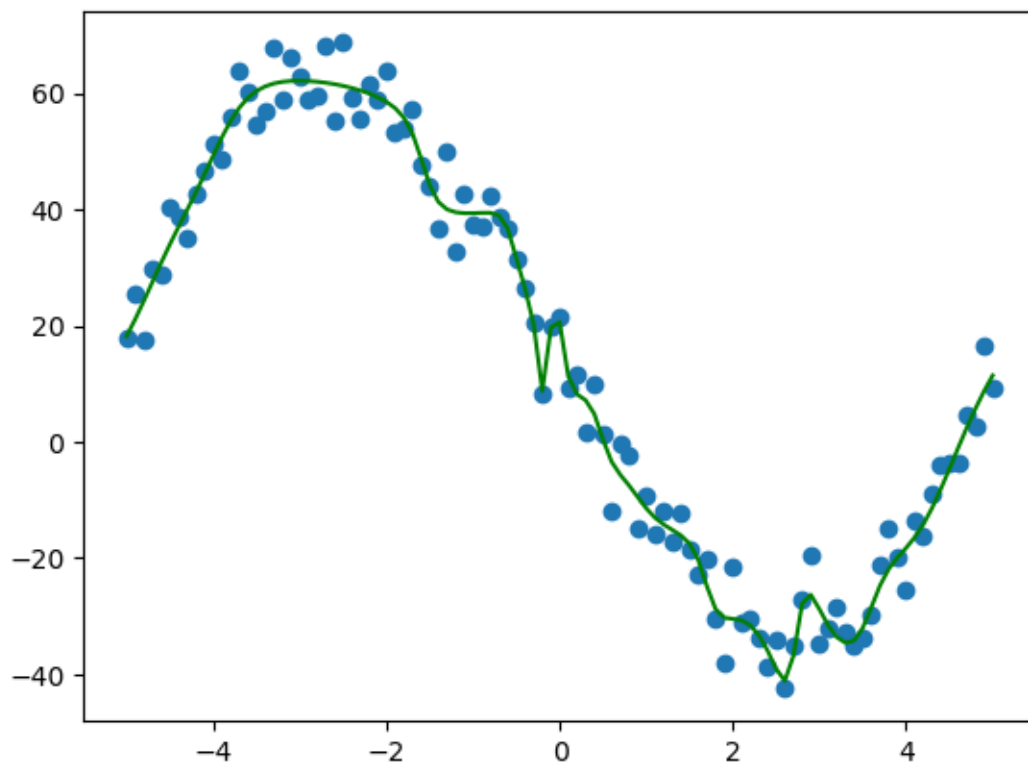
regr = MLPBinaryLinReg(bias=-1, dim_hidden=100)
regr.fit(X, t, epochs=50000, eta=0.001)
Y=regr.predict(X)

plt.scatter(X, t)
plt.plot(X, Y, 'g')
```

```
CPU times: user 9.71 s, sys: 996 µs, total: 9.71 s
```

```
Wall time: 9.77 s
```

```
[71]: [<matplotlib.lines.Line2D at 0x7f3119398ca0>]
```



```
[72]: error = mse_2(t, Y)
      print("MSE: ", error)
```

MSE: 15.836922293195377

```
[73]: print("RMSE :", np.sqrt(error))
```

RMSE : 3.979563078177726

We see we can get the predicted curve to follow the data quite closely. We have definitely entered the land of overfitting.

```
[ ]:
```