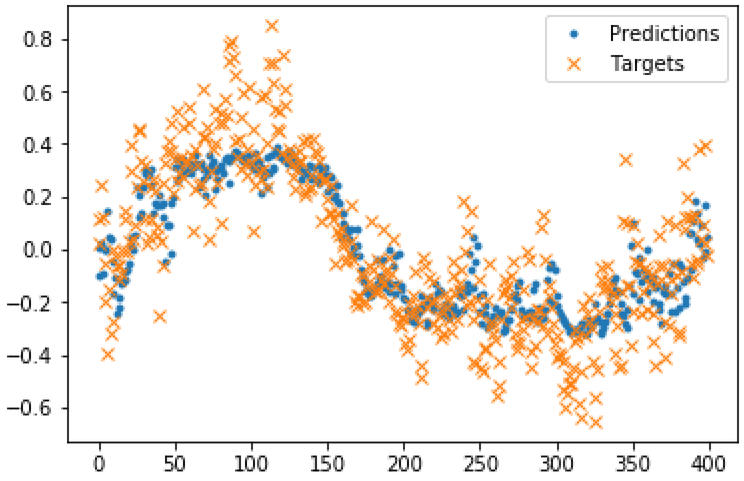
Lab 3 – Neural Network - MLP

1. Study Section 4.4.4. Download the dataset PNOz.dat and run the MLP. See whether you can get something similar to Fig. 4.16.

Answer: We were able to get a figure like Fig. 4.16, as showed below:



We used the following code:

--- File PNOz.py ---

```
from pylab import *
from numpy import *

PNoz = loadtxt('/Users/Sebastian/Documents/MACHINE LEARNING/Lab3/PNOz.dat',
delimiter=',')
ion()
plot(arange(shape(PNoz)[0]), PNoz[:, 2], '.')
xlabel('Time (Days)')
ylabel('Ozone (Dobson units)')

# Normalise data
PNoz[:, 2] = PNoz[:, 2] - PNoz[:, 2].mean()
PNoz[:, 2] = PNoz[:, 2] / PNoz[:, 2].max()

# Assemble input vectors
t = 2
k = 3

lastPoint = shape(PNoz)[0] - t * (k + 1) - 1
inputs = zeros((lastPoint, k))
targets = zeros((lastPoint, 1))
for i in range(lastPoint):
    inputs[i, :] = PNoz[i:i + t * k:t, 2]
    targets[i] = PNoz[i + t * (k + 1), 2]

test = inputs[-400:, :]
testtargets = targets[-400:]

# Randomly order the data
```

```
inputs = inputs[:-400, :]
targets = targets[:-400]
change = list(range(shape(inputs)[0]))
random.shuffle(change)
inputs = inputs[change, :]
targets = targets[change, :]

train = inputs[::2, :]
traintargets = targets[::2]
valid = inputs[1::2, :]
validtargets = targets[1::2]

# Train the network
import mlp

net = mlp.mlp(train, traintargets, 3, outtype='linear')
net.earlystopping(train, traintargets, valid, validtargets, 0.25)

test = concatenate((test, -ones((shape(test)[0], 1))), axis=1)
testout = net.mlpfwd(test)

figure()
plot(arange(shape(test)[0]), testout, '.')
plot(arange(shape(test)[0]), testtargets, 'x')
legend(('Predictions', 'Targets'))
print((0.5 * sum((testtargets - testout) ** 2)))
show()
```

--- File mlp.py ---

```
from numpy import *


class mlp:
    """ A Multi-Layer Perceptron"""

    def __init__(self, inputs, targets, nhidden, beta=1, momentum=0.9,
outtype='logistic'):
        """ Constructor """
        # Set up network size
        self.nin = shape(inputs)[1]
        self.nout = shape(targets)[1]
        self.ndata = shape(inputs)[0]
        self.nhidden = nhidden

        self.beta = beta
        self.momentum = momentum
        self.outtype = outtype

        # Initialise network
        self.weights1 = (random.rand(self.nin + 1, self.nhidden) - 0.5) * 2 /
sqrt(self.nin)
        self.weights2 = (random.rand(self.nhidden + 1, self.nout) - 0.5) * 2 /
sqrt(self.nhidden)

    def earlystopping(self, inputs, targets, valid, validtargets, eta, niterations=100):

        valid = concatenate((valid, -ones((shape(valid)[0], 1))), axis=1)

        old_val_error1 = 100002
        old_val_error2 = 100001
        new_val_error = 100000
        count = 0
        while (((old_val_error1 - new_val_error) > 0.001) or ((old_val_error2 -
old_val_error1) > 0.001)):
            count += 1
            print(count)
            self.mlptrain(inputs, targets, eta, niterations)
            old_val_error2 = old_val_error1
            old_val_error1 = new_val_error
            validout = self.mlpfwd(valid)
            new_val_error = 0.5 * sum((validtargets - validout) ** 2)

        print("Stopped", new_val_error, old_val_error1, old_val_error2)
        return new_val_error

    def mlptrain(self, inputs, targets, eta, niterations):
        """ Train the thing """
        # Add the inputs that match the bias node
        inputs = concatenate((inputs, -ones((self.ndata, 1))), axis=1)
        change = list(range(self.ndata))

        updatew1 = zeros((shape(self.weights1)))
        updatew2 = zeros((shape(self.weights2)))

        for n in range(niterations):

            self.outputs = self.mlpfwd(inputs)

            error = 0.5 * sum((targets - self.outputs) ** 2)
            if (mod(n, 100) == 0):
                print("Iteration: ", n, " Error: ", error)

                # Different types of output neurons
            if self.outtype == 'linear':
                deltao = (targets - self.outputs) / self.ndata
            elif self.outtype == 'logistic':
                deltao = (targets - self.outputs) * self.outputs * (1.0 - self.outputs)
            elif self.outtype == 'softmax':
                # deltao = (targets-self.outputs)*self.outputs/self.ndata
                deltao = (targets - self.outputs) / self.ndata
            else:
                print("error")

            deltah = self.hidden * (1.0 - self.hidden) * (dot(deltao,
transpose(self.weights2)))

            updatew1 = eta * (dot(transpose(inputs), deltah[:, :-1])) + self.momentum *
updatew1
            updatew2 = eta * (dot(transpose(self.hidden), deltao)) + self.momentum *
updatew2
            self.weights1 += updatew1
            self.weights2 += updatew2

            # Randomise order of inputs
            random.shuffle(change)
            inputs = inputs[change, :]
            targets = targets[change, :]

    def mlpfwd(self, inputs):
        """ Run the network forward """

        self.hidden = dot(inputs, self.weights1);
        self.hidden = 1.0 / (1.0 + exp(-self.beta * self.hidden))
        self.hidden = concatenate((self.hidden, -ones((shape(inputs)[0], 1))), axis=1)
        outputs = dot(self.hidden, self.weights2);

        # Different types of output neurons
        if self.outtype == 'linear':
            return outputs
        elif self.outtype == 'logistic':
            return 1.0 / (1.0 + exp(-self.beta * outputs))
        elif self.outtype == 'softmax':
            normalisers = sum(exp(outputs), axis=1) * ones((1, shape(outputs)[0]))
            return transpose(transpose(exp(outputs)) / normalisers)
        else:
            print("error")

    def confmat(self, inputs, targets):
        """Confusion matrix"""

        # Add the inputs that match the bias node
        inputs = concatenate((inputs, -ones((shape(inputs)[0], 1))), axis=1)
        outputs = self.mlpfwd(inputs)

        nclasses = shape(targets)[1]

        if nclasses == 1:
            nclasses = 2
            outputs = where(outputs > 0.5, 1, 0)
        else:
            # 1-of-N encoding
            outputs = argmax(outputs, 1)
            targets = argmax(targets, 1)

        cm = zeros((nclasses, nclasses))
        for i in range(nclasses):
            for j in range(nclasses):
                cm[i, j] = sum(where(outputs == i, 1, 0) * where(targets == j, 1, 0))

        print("Confusion matrix is:")
        print(cm)
        print("Percentage Correct: ", trace(cm) / sum(cm) * 100)
```

2. Use the knowledge you gained from #1 to solve the following problem:

Suppose that the local power company wants to predict electricity demand for the next 5 days. They have the data about daily demand for the last 5 years. Typically, the demand will be a number between 80 and 400.

a. Describe how you could use an MLP to make the prediction. What parameters would you have to choose, and what do you think would be sensible values for them?

Answer: To make the prediction, we could use the same strategy used in the last question creating a:

- Training set
- Validation set
- Test set

And then, training an MLP. We would have to use parameters such as k = 4 and tau = 2.

b. If the weather forecast for the next day, being the estimated temperatures for daytime and nighttime, was available, how would you add that into your system?

Answer: We could add this information as a new input to the network. We also would need to relate estimated temperatures for daytime and nighttime with the power used, creating data if it were not available.

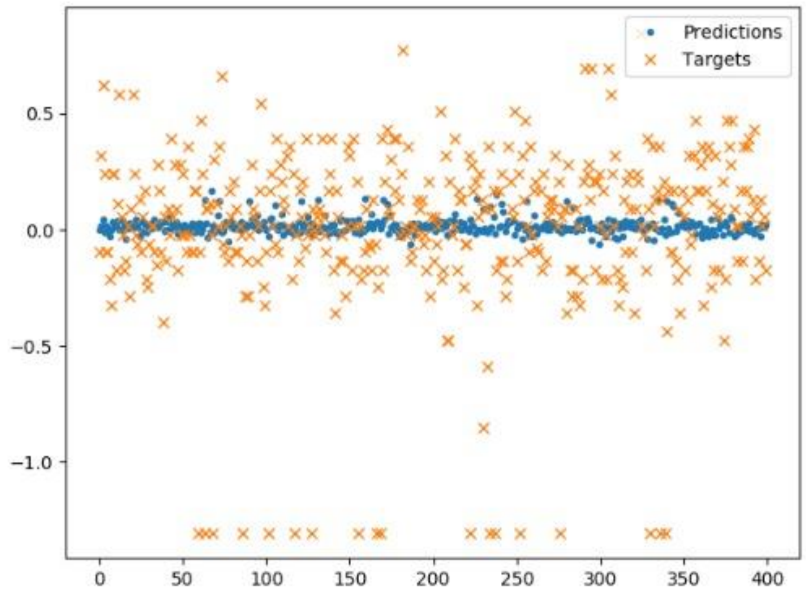
c. Do you think that this system would work well for predicting power consumption? Are there demands that it would not be able to predict?

Answer: Yes, we think that it would work well. The system would not be able to predict, for example, trends within a specific season.

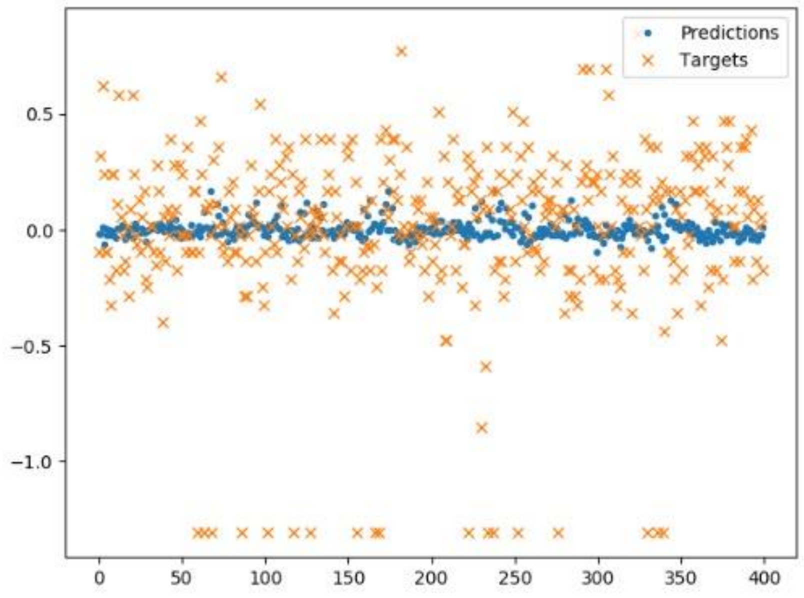
3. Modify the code to allow another hidden layer to be used. You will have to work out the gradient as well in order to compute the weight updates for the extra layer of weights. Test this new network on the Pima Indian dataset that was described in Section 3.4.4.

Answer: After adding more hidden layers, we noticed that the algorithm continued to work fine. We just needed to use different formulas to the second hidden layer and to the input of the second hidden layer.

- Result using 3 layers:



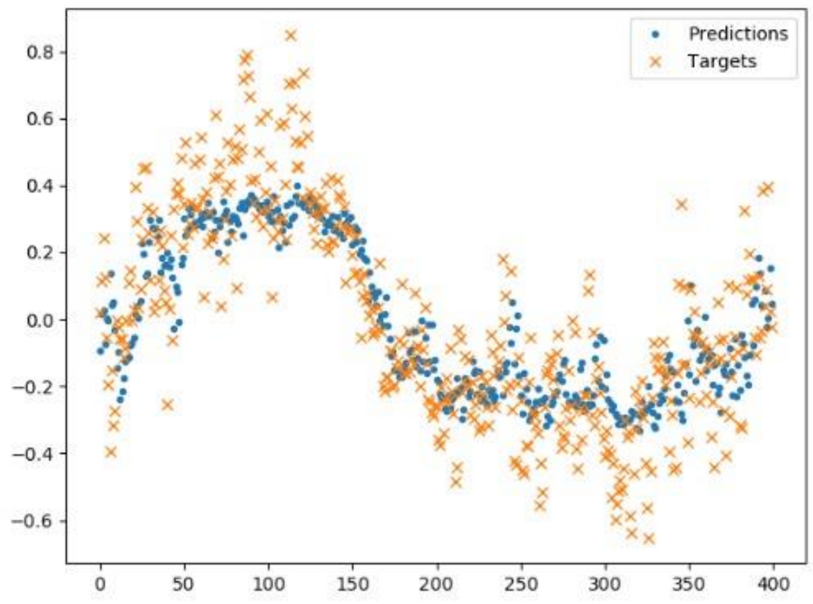
- Result using 4 layers:



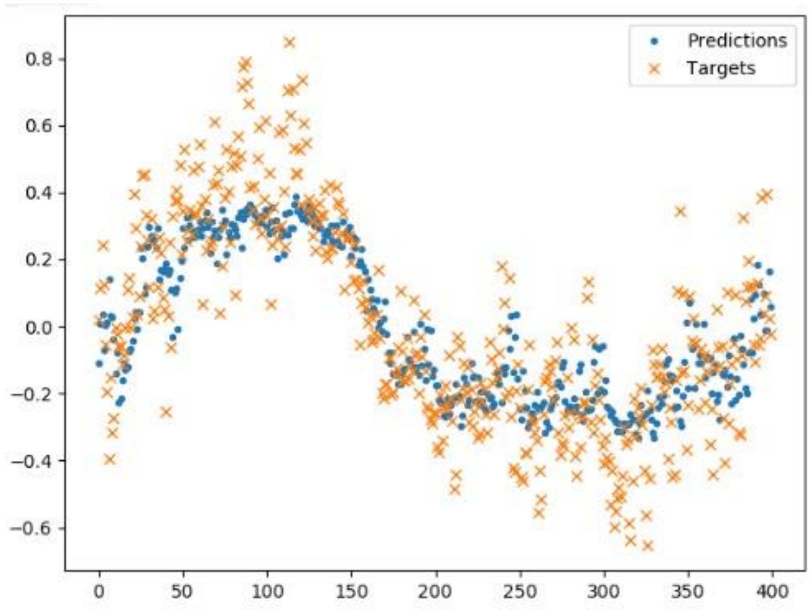
4. A recurrent network has some of its outputs connected to its own inputs, so that the outputs at time t are fed back into the network at time t + 1. This can be a different way to deal with time-series data. Modify the MLP code so that it acts as a recurrent network, and test it out on the Palmerston North ozone data on the book website.

Answer: For this problem, we only needed to modify the output node's formula. The reason is because we needed to add the last value of each output node to itself. Other equations remained the same.

- Here are the results with the original formula:



- And modifying the formula:


Predictions
Targets