6ELEN018W - Applied Robotics
Lecture 8: Robot Control - Intelligent Control Algorithms

Dr Dimitris C. Dracopoulos

What is Control and Why it is Needed

What is Control and Why it is Needed

- A robot needs to move its joints to achieve tasks

What is Control and Why it is Needed

- A robot needs to move its joints to achieve tasks
- A mobile robot moves to different locations

What is Control and Why it is Needed

- A robot needs to move its joints to achieve tasks
- A mobile robot moves to different locations

The movement of a robot (joints) is done using actuators.

What is Control and Why it is Needed

- A robot needs to move its joints to achieve tasks
- A mobile robot moves to different locations

The movement of a robot (joints) is done using actuators.

In general, everything can be considered as **control**:

- Decisions we make affect (control) our future

What is Control and Why it is Needed

- A robot needs to move its joints to achieve tasks
- A mobile robot moves to different locations

The movement of a robot (joints) is done using actuators.

In general, everything can be considered as **control**:

- Decisions we make affect (control) our future
- Decision while driving affect (control) the next position and the final location

What is Control and Why it is Needed

- A robot needs to move its joints to achieve tasks
- A mobile robot moves to different locations

The movement of a robot (joints) is done using actuators.

In general, everything can be considered as **control**:

- Decisions we make affect (control) our future
- Decision while driving affect (control) the next position and the final location
- Control theory is a big area used not only in engineering and robotics, but in computer science

What is Control and Why it is Needed

- A robot needs to move its joints to achieve tasks
- A mobile robot moves to different locations

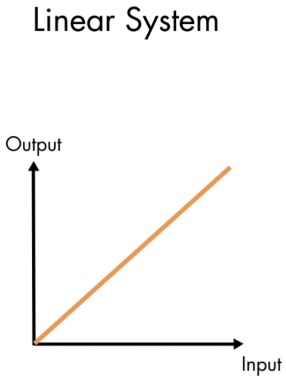
The movement of a robot (joints) is done using actuators.

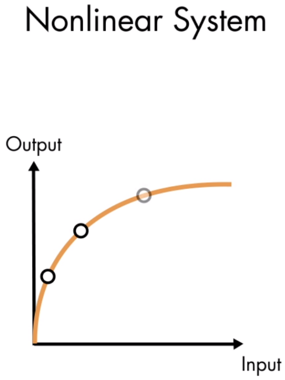
In general, everything can be considered as **control**:

- Decisions we make affect (control) our future
- Decision while driving affect (control) the next position and the final location
- Control theory is a big area used not only in engineering and robotics, but in computer science
- Can be seen as what is <u>the best next action to take</u> (given a specific state) so as to <u>achieve (optimise) specific objectives!</u>

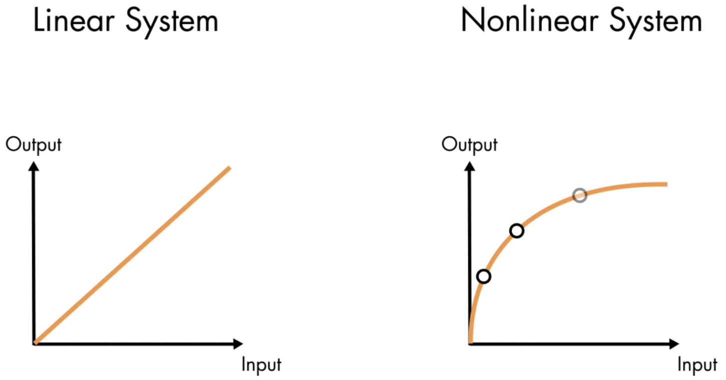
Linear vs Nonlinear Systems

Linear vs Nonlinear Systems



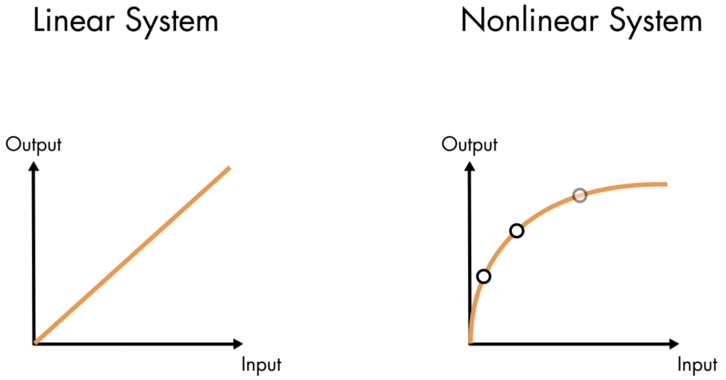


Linear vs Nonlinear Systems



- In real life all systems are nonlinear, however many of them can be linearised about their operation point.

Linear vs Nonlinear Systems



- In real life all systems are nonlinear, however many of them can be linearised about their operation point.

Linear systems are easier to analyse and prove mathematically their behaviour and properties.

Classical (Traditional) Robot Control

Classical (Traditional) Robot Control

- Manipulators and fixed robots are very good in their operation!

Classical (Traditional) Robot Control

- Manipulators and fixed robots are very good in their operation!
- Kinematics, inverse kinematics and dynamics are well understood for fixed mechanics and robots working in the same environment.

Classical (Traditional) Robot Control

- Manipulators and fixed robots are very good in their operation!
- Kinematics, inverse kinematics and dynamics are well understood for fixed mechanics and robots working in the same environment.
- Their actuators are very well controlled.

Classical (Traditional) Robot Control

- Manipulators and fixed robots are very good in their operation!
- Kinematics, inverse kinematics and dynamics are well understood for fixed mechanics and robots working in the same environment.
- Their actuators are very well controlled.
- Linearity assumptions (or operation near points where their dynamics is linearised) makes it possible to analyse their behaviour and stability.

Classical (Traditional) Robot Control

- Manipulators and fixed robots are very good in their operation!
- Kinematics, inverse kinematics and dynamics are well understood for fixed mechanics and robots working in the same environment.
- Their actuators are very well controlled.
- Linearity assumptions (or operation near points where their dynamics is linearised) makes it possible to analyse their behaviour and stability.

BUT

Complex Systems

Real Life systems are complex.

Complex Systems

Real Life systems are complex.

- Robots which do complex work are non-linear (and their behaviour/response cannot be linearised). Impossible to control using classical methods.

Complex Systems

Real Life systems are complex.

- Robots which do complex work are non-linear (and their behaviour/response cannot be linearised). Impossible to control using classical methods.
- Robots need to be adaptive and be able to cope with unknown environments and unseen situations similarly to how humans do.

Complex Systems

Real Life systems are complex.

- Robots which do complex work are non-linear (and their behaviour/response cannot be linearised). Impossible to control using classical methods.
- Robots need to be adaptive and be able to cope with unknown environments and unseen situations similarly to how humans do.
  - Robots have to be adaptive knowing what that they are going to do if encountering a partially unknown environment, a completely unknown environment or unknown difficulties.

Complex Systems

Real Life systems are complex.

- Robots which do complex work are non-linear (and their behaviour/response cannot be linearised). Impossible to control using classical methods.
- Robots need to be adaptive and be able to cope with unknown environments and unseen situations similarly to how humans do.
  - Robots have to be adaptive knowing what that they are going to do if encountering a partially unknown environment, a completely unknown environment or unknown difficulties.
  - We send robots to space.

Complex Systems

Real Life systems are complex.

- Robots which do complex work are non-linear (and their behaviour/response cannot be linearised). Impossible to control using classical methods.
- Robots need to be adaptive and be able to cope with unknown environments and unseen situations similarly to how humans do.
  - Robots have to be adaptive knowing what that they are going to do if encountering a partially unknown environment, a completely unknown environment or unknown difficulties.
  - We send robots to space.
- Reconfigurable Robots

Complex Systems

Real Life systems are complex.

- Robots which do complex work are non-linear (and their behaviour/response cannot be linearised). Impossible to control using classical methods.
- Robots need to be adaptive and be able to cope with unknown environments and unseen situations similarly to how humans do.
  - Robots have to be adaptive knowing what that they are going to do if encountering a partially unknown environment, a completely unknown environment or unknown difficulties.
  - We send robots to space.
- Reconfigurable Robots
  - What happens if damage happens in one of the actuators with the robot or one of their thumbs might hit an obstacle and it might lose part of it?

Complex Systems

Real Life systems are complex.

- Robots which do complex work are non-linear (and their behaviour/response cannot be linearised). Impossible to control using classical methods.
- Robots need to be adaptive and be able to cope with unknown environments and unseen situations similarly to how humans do.
  - Robots have to be adaptive knowing what that they are going to do if encountering a partially unknown environment, a completely unknown environment or unknown difficulties.
  - We send robots to space.
- Reconfigurable Robots
  - What happens if damage happens in one of the actuators with the robot or one of their thumbs might hit an obstacle and it might lose part of it?

$\Longrightarrow$ A new challenge: **Intelligent Control**

Complex Systems

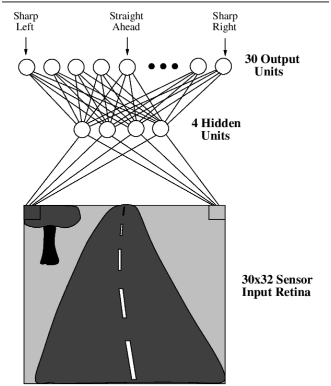
Real Life systems are complex.

- Robots which do complex work are non-linear (and their behaviour/response cannot be linearised). Impossible to control using classical methods.
- Robots need to be adaptive and be able to cope with unknown environments and unseen situations similarly to how humans do.
  - Robots have to be adaptive knowing what that they are going to do if encountering a partially unknown environment, a completely unknown environment or unknown difficulties.
  - We send robots to space.
- Reconfigurable Robots
  - What happens if damage happens in one of the actuators with the robot or one of their thumbs might hit an obstacle and it might lose part of it?

$\Longrightarrow$ A new challenge: **Intelligent Control** based on intelligent algorithms.

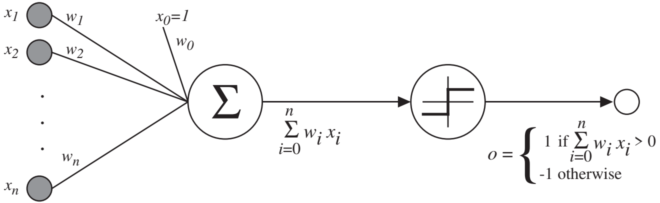
Robot Driving a Car - Autonomous Driving

ALVINN [Pomerleau 1989] drives 70 mph on highways



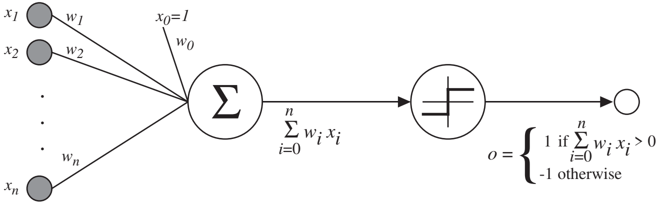
https://www.youtube.com/watch?v=IaoIqVMd6tc&t=71s

Perceptron



$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Perceptron

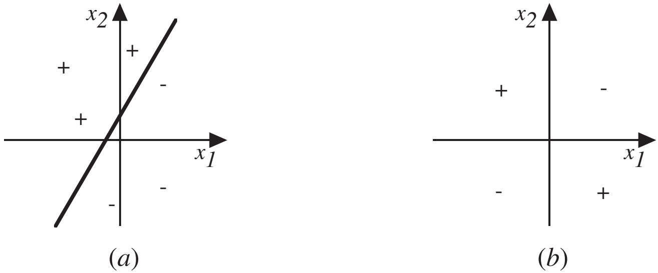


$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

A simpler vector notation can be used:

$$o(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} > 0 \\ -1 & \text{otherwise.} \end{cases}$$

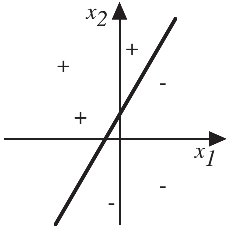
Perceptron (cont'd)



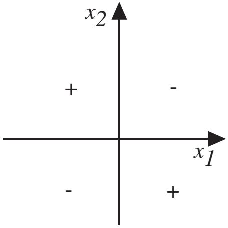
Represents some useful functions

- What weights represent $g(x_1, x_2) = AND(x_1, x_2)$?

Perceptron (cont'd)



(a)



(b)

Represents some useful functions

- What weights represent $g(x_1, x_2) = AND(x_1, x_2)$?

But some functions are not representable with a single layer of neurons.

- e.g., not linearly separable (such as the XOR function)

Perceptron training rule

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = \eta(t - o)x_i$$

Where:

- $t = c(\vec{x})$ is target value
- o is perceptron output
- η is small constant (e.g., .1) called *learning rate*

Perceptron training rule

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = \eta(t - o)x_i$$

Where:

- $t = c(\vec{x})$ is target value
- o is perceptron output
- η is small constant (e.g., .1) called *learning rate*

Can prove it will converge:

- If training data is linearly separable
- and η sufficiently small

Gradient Descent

To understand, consider simpler *linear unit*, where

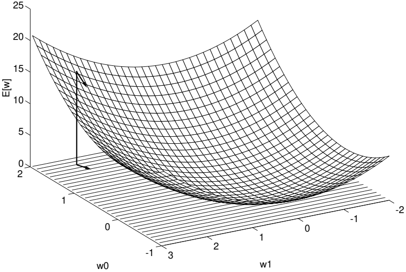
$$o = w_0 + w_1 x_1 + \cdots + w_n x_n$$

Let's learn w_i's that minimise the squared error

$$E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

where D is set of training examples

Gradient Descent (cont'd)



Gradient

$$\nabla E[\vec{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots \frac{\partial E}{\partial w_n} \right]$$

Training rule:

$$\Delta \vec{w} = -\eta \nabla E[\vec{w}]$$

i.e.,

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Calculating the Derivative

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_d (t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_d (t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x_d}) \\
\frac{\partial E}{\partial w_i} &= \sum_d (t_d - o_d)(-x_{i,d})
\end{aligned}$$

Application of Gradient Descent

Each training example is a pair of the form $\langle \vec{x}, t \rangle$, where $\vec{x}$ is the vector of input values, and t is the target output value. η is the learning rate (e.g., .05).

Application of Gradient Descent

Each training example is a pair of the form $\langle \vec{x}, t \rangle$, where $\vec{x}$ is the vector of input values, and t is the target output value. η is the learning rate (e.g., .05).

- Initialise each w_i to some small random value
- Until the termination condition is met, Do
  - Initialise each Δw_i to zero.
  - For each $\langle \vec{x}, t \rangle$ in *training_examples*, Do
    - Input the instance $\vec{x}$ to the unit and compute the output o
    - For each linear unit weight w_i, Do

$$\Delta w_i \longleftarrow \Delta w_i + \eta (t - o) x_i$$

  - For each linear unit weight w_i, Do

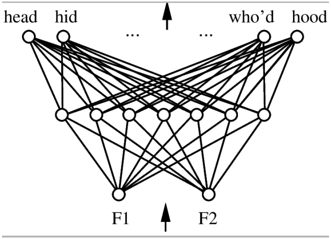
$$w_i \longleftarrow w_i + \Delta w_i$$

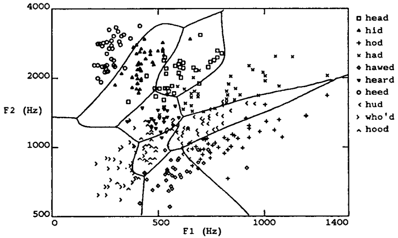
Multilayer Perceptrons with Hidden Layers and Sigmoid Units

To overcome the limitations of the single layer Perceptron (linear separability):

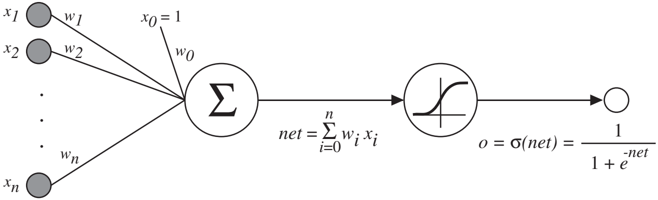
Multilayer Perceptrons with Hidden Layers and Sigmoid Units

To overcome the limitations of the single layer Perceptron (linear separability):





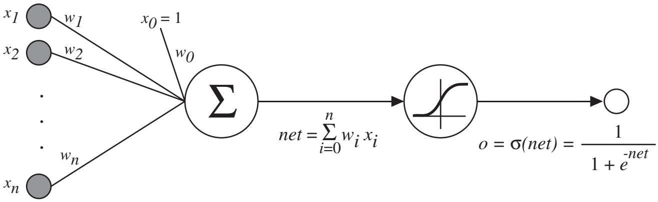
Sigmoid Unit



$\sigma(x)$ is the sigmoid function

$$\frac{1}{1 + e^{-x}}$$

Sigmoid Unit

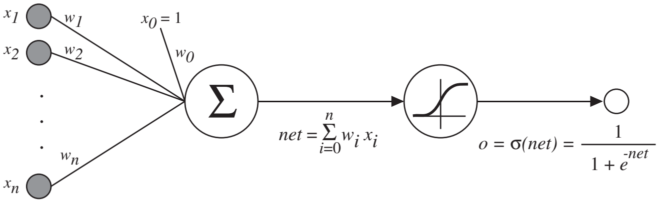


$\sigma(x)$ is the sigmoid function

$$\frac{1}{1 + e^{-x}}$$

Nice property: $\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$

Sigmoid Unit



$\sigma(x)$ is the sigmoid function

$$\frac{1}{1 + e^{-x}}$$

Nice property: $\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$
We can derive gradient descent rules to train:

- One sigmoid unit
- *Multilayer networks* of sigmoid units $\rightarrow$ Backpropagation

Backpropagation Algorithm

Initialise all weights to small random numbers.
Until satisfied, Do:

- For each training example, Do
  1. Input the training example to the network and compute the network outputs
  2. For each output unit k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

  3. For each hidden unit h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{k,h}\delta_k$$

  4. Update each network weight $w_{i,j}$

$$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$

where

$$\Delta w_{i,j} = \eta \delta_j x_i$$

More on Backpropagation

- Gradient descent over entire *network* weight vector
- Will find a local, not necessarily global error minimum
  - In practice, often works well (can run multiple times)
- Often include weight *momentum* α

$$\Delta w_{i,j}(n) = \eta \delta_j x_i + \alpha \Delta w_{i,j}(n-1)$$

- Minimises error over *training* examples
  - Will it generalise well to subsequent examples?
- Training can take thousands of iterations $\rightarrow$ slow!
- Using network after training is very fast

Convergence of Backpropagation

Gradient descent to some local minimum

- Perhaps not global minimum...
- Add momentum
- Stochastic gradient descent
- Train multiple nets with different initial weights

Expressive Capabilities of ANNs

Boolean functions:

- Every boolean function can be represented by network with single hidden layer
- but might require exponential (in number of inputs) hidden units

Expressive Capabilities of ANNs

Boolean functions:

- Every boolean function can be represented by network with single hidden layer
- but might require exponential (in number of inputs) hidden units

Continuous functions:

- Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer [Cybenko 1989; Hornik et al. 1989]

Expressive Capabilities of ANNs

Boolean functions:

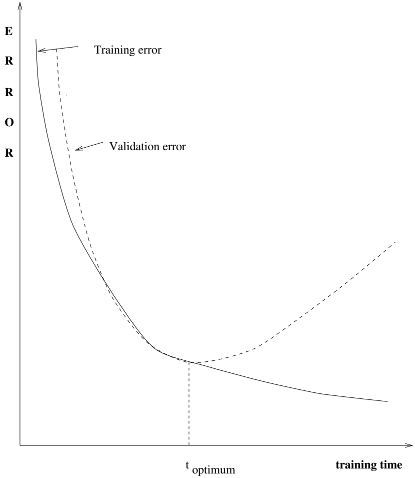
- Every boolean function can be represented by network with single hidden layer
- but might require exponential (in number of inputs) hidden units

Continuous functions:

- Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer [Cybenko 1989; Hornik et al. 1989]
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988].

How to Avoid Overfitting and Improve Generalisation

Split the data into 3 sets, *training*, *testing* and *validation* and stop the training according to the following diagram.



Tips for Using Backpropagation

- When a sigmoid is used in the output layer, use 0.9 and 0.1 instead of 1 and 0 as the targets, to avoid the saturated parts of the sigmoid function.

Tips for Using Backpropagation

- When a sigmoid is used in the output layer, use 0.9 and 0.1 instead of 1 and 0 as the targets, to avoid the saturated parts of the sigmoid function.
- Experiment with different learning rates and number of hidden nodes and layers. Do not use more than 3 hidden layers.

Tips for Using Backpropagation

- When a sigmoid is used in the output layer, use 0.9 and 0.1 instead of 1 and 0 as the targets, to avoid the saturated parts of the sigmoid function.
- Experiment with different learning rates and number of hidden nodes and layers. Do not use more than 3 hidden layers.
- Preprocess your data by scaling them in the same range. If some features (columns) are not relevant you can discard them completely.

Using the `sklearn` Python module

To install:

- On your own computer (ideally in a virtual environment but this is not compulsory):

  ```
  pip install -U scikit-learn
  ```

- In the university labs the `sklearn` module is already installed inside Anaconda. Start the Jupyter lab application from inside Anaconda and start using it.

An Example Using `sklearn` - The Diabetes Dataset

A well-known dataset is the diabetes dataset, used to create a neural network which can predict whether someone has diabetes.

- 442 diabetes patients
- Input variables (features): age, sex, body mass index, average blood pressure, and six blood serum measurements.

An Example Using sklearn - The Diabetes Dataset (cont'd)

```
from sklearn.neural_network import MLPRegressor
from sklearn import datasets
from sklearn.model_selection import train_test_split

import matplotlib.pylab as plt
import numpy as np
from sklearn.metrics import mean_squared_error

# load all the data from the dataset
diabetes = datasets.load_diabetes()

# check the matrix shape (number of features and data for the inputs)
print(diabetes.data.shape)

# check the shape (number of data for targets)
print(diabetes.target.shape)

# feature (column) names
print(diabetes.feature_names)

X = diabetes.data
y = diabetes.target
```

An Example Using sklearn - The Diabetes Dataset (cont'd)

```
# Create training/ test data split
X_train, X_test, y_train, \
        y_test = train_test_split(X, y, test_size=0.2, random_state=1)

# Instantiate MLPRegressor
nn = MLPRegressor(
    activation='relu',
    hidden_layer_sizes=(10, 10),
    alpha=0.001,
    max_iter = 10000,
    random_state=20,
    early_stopping=False
)

# Train the model
nn.fit(X_train, y_train)

# Make prediction
pred = nn.predict(X_test)

# Calculate accuracy and error metrics
test_set_rsquared = nn.score(X_test, y_test)
test_set_rmse = np.sqrt(mean_squared_error(y_test, pred))
```

An Example Using sklearn - The Diabetes Dataset (cont'd)

```
# Print R_squared and RMSE value
print('R_squared value: ', test_set_rsquared)
print('RMSE: ', test_set_rmse)

# Predict unknown data
y_pred = nn.predict(X_test)

# plot prediction and actual data
plt.plot(y_test, y_pred, '.')

# plot a line, a perfit predict would all fall on this line
x = np.linspace(0, 330, 2)
y = x
plt.plot(x, y)
plt.show()
```

Further Material

For other details and related material see:

Dimitris C. Dracopoulos, *Evolutionary Learning Algorithms for Neural Adaptive Control*, Springer Verlag, London, August 1997, ISBN: 3-540-76161-6.