Slides with the course Reinforcement Learning: Neural Networks M. Wiering, March 2004.

Neural Networks

Goals of this course:

- To know what neural networks are
- To know the Delta learning-rule
- To be able to compute changes in a (linear) neural network given a learning example
- To know the backpropagation learning rule
- To know how neural networks can be combined with Reinforcement learning

---

Neural networks

Artificial neural networks (ANNs) consist of a set of neurons (computing units) which are connected in networks (McCulloch en Pitts, 1943).

They posess useful computational properties (e.g. they can exactly approximate all continuous functions with 1 hidden layer)

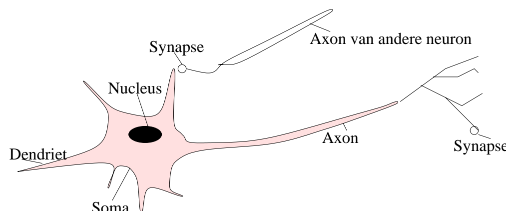
They offer us the possibility to learn how the brain works

A **neuron** or nervous cell is the fundamental element of the brain

A neuron consists of a celbody: the **soma**

Out of the celbody **dendrites** and an **axon** branch. An axon connects with the dendrites of other neurons through **synapses**, the connection points.

A biological neural network



Chemical transmitter fluids are released in the synapses and flow in the dendrites

This causes the **action-potential** in the soma to increase or decrease

When the action-potential is higher than a particular threshold, an electrical pulse is passed to the axon (the neuron **fires**).

Synapses which make the action-potential to increase are called **excitatory**.

Synapses which make the action-potential to decrease are called **inhibitory**.

Artificial neural networks

A neural network consists of a number of **neurons** (units) and connections between the neurons

Each connection has a scalar weight value associated to it.

Learning happens by adjusting the weights

Each neuron has a number of incoming connections from other neurons, a number of outgoing connections, and an **activation-level**

The idea is that each neuron makes a local computation, using its incoming connections.

To build a neural network, one has to specify the topology of the network (how are the neurons connected)

Weights are usually random initialized

Comparison between ANNs and Biological NNs

Examine humans:

- Neuron switch time: .001 second
- Number of neurons: 10^{10-11}
- Connections per neuron: 10^{4-5}
- Visual recognition time: 0.1 second
- 100 inference steps do not look to be enough $\rightarrow$
  Lots of parallel computation

Properties of artificial neural networks

- Many neuron-like threshold switch units

- Many weighted connections between units
- To large extent, parallel, distributed process
- Automatic learning of the weights

When can Neural Networks be used?

- Input is high-dimensional discrete or continuous
  (e.g. raw sensor data as input)
- Output is discrete or continuous
- Output is a vector of (target) values
- Possibly noisy data
- Shape of the targetfunction is unknown
- Readabiliy of found solution is unimportant

Examples:

- Speech recognition
- Image classification (face recognition)
- Financial forecasting
- Pattern recognition (zipcodes)

Example: car-driving

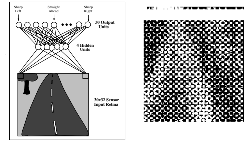
Feedforward neural networks

Topology is a layered structure; all connections go from one layer to the next one.

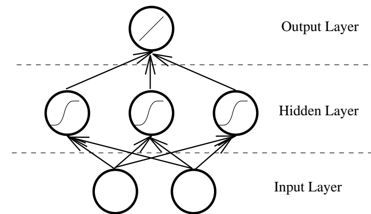
We distinct the following layers:

- The **Input** layer: here the inputs for the network are stored
- The **Hidden** layer: here the internal (non-linear) computations are made.
- The **Output** layer: here the values of the target-outputs are computed and stored.

ALVINN drives 70 mph on highways



77 lecture slides for textbook *Machine Learning*, T. Mitchell, McGraw Hill, 1997



Neuron

In a network an individual neuron looks as follows:

Learning is done by adjusting the weights on a set of training examples

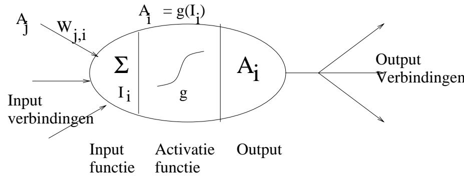
Example: the output of a network is 0.9. The desired output is 1.0. Increase the weights which make the output higher. Decrease the weights which make the output lower.

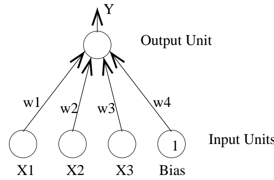
A linear neural network

The simplest neural network is a linear neural network. This network consists only of an input and output layer.

A linear neural network looks as follows:

An additional bias-unit is used to be able to represent all linear functions. This can be done by adding an additional value (1) to the input-vector.



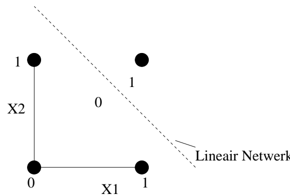


The linear network is a function (mapping) from the inputs $X_1, \ldots, X_N$ to the output Y:

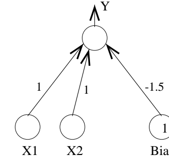
$$Y = \sum_i w_i X_i$$

Representing functions

A linear network can for example represent the AND function.



The following network (Perceptron) does this (if the output > 0 then $Y = 1$, else $Y = 0$):



Learning

An initial network is made with random weights (e.g. between -0.3 and 0.3)

Learning is done as follows:

- Definine the error as the quadratic difference between the desired outcome D and the obtained outcome Y on an example : $X_1, \ldots, X_N \rightarrow D$:

$$E = \frac{1}{2}(D - Y)^2$$

- We want to compute the derivative of the error E to the weights $w_1, \ldots, w_N$:

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial Y}\frac{\partial Y}{\partial w_i} = -(D - Y)X_i$$

- Now we "update" the weights with a learning speed $\alpha > 0$ to decrease the error.
  The **Delta learningrule** is as follows:

$$w_i = w_i + \alpha(D - Y)X_i$$

- We stop when the error over all training examples is small enough

---

Example

Given the learning example $(0.5, 0.5 \rightarrow 1)$.

We make an initial linear network with weights: 0.3 and 0.5 and 0.0 (for the bias).

We choose a learningspeed, eg.: $\alpha = 0.5$

We adjust now the weights:

$Y = 0.3 * 0.5 + 0.5 * 0.5 + 0.0 * 1.0 = 0.4.$
$E = 1/2(1.0 - 0.4)^2 = 0.18$
$w_1 = 0.3 + 0.5 * 0.6 * 0.5 = 0.45$
$w_2 = 0.5 + 0.5 * 0.6 * 0.5 = 0.65$
$w_3 = 0.0 + 0.5 * 0.6 * 1.0 = 0.30$

For a next presentation of the learning example, the output will be:
$Y' = 0.45 * 0.5 + 0.65 * 0.5 + 0.3 * 1.0 = 0.85.$

Question: Suppose that the learning example is presented again: compute the new weights

---

Batch vs Stochastic Gradient Descent

There are in principle two ways to deal with the training data

- Batch-learning: tries to minimize the error in 1 time for all learning examples. For this the total derivative is computed and

the weights are adjusted a single time per iteration :

$$E = \frac{1}{2}\sum_p (D^p - Y^p)^2$$

Dus:

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial Y}\frac{\partial Y}{\partial w_i} = \sum_p -(D^p - Y^p)X_i^p$$

- Online-learning: adjusts the error after each individual learning example. Thus it makes stochastic steps in the error landscape (the total error can be decreases or increases):

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial Y^p}\frac{\partial Y^p}{\partial w_i} = -(D^p - Y^p)X_i^p$$

Usually online learning is used. It can also result in 10 to 100 times faster convergence.

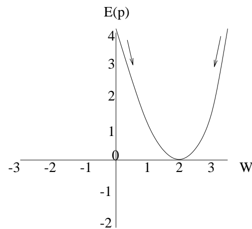
Intuition of the learning process

Error of the learning example has a derivative to each weight. Minimalise the error by going downwards along the derivative (gradient).

Example: goalfunction $Y = 2X$.

Learningexample $p = (1, 2)$.

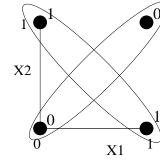
Errorlandscape example:



Limitations of linear networks

A linear network can not represent the X-OR function. The examples of the X-OR function is non linear-separable.

After the appearance of the book Perceptrons of Minsky and Papert (1969) in which these problems were indicated, the initial enthusiasm for neural networks disappeared.



A small group of researchers continued. This led to a number of different neural networks.

In 1986 neural networks became popular again after the invention of the **backpropagation** algorithm, in which by using the chain-rule for derivatives, also non-linear (multi-layer) feedforward neural networks could be trained.

Representation in multi-layer feedforward neural networks

We represent the network in a directed graph. The optimal representation can have an arbitrary small error for a particular target function.

Usually the topology of the network is chosen a-priori.

This causes a representation error (even the optimal weights in the chosen representation have a particular error).

It is also very difficult to learn the optimal weights (learning error) due to local minima.

We distinguish 2 steps: forward propagation (use) and backward propagation / backprop (learning)

Forward propagation in multi-layer feedforward networks

- Clamp input vector $\vec{a}$
- Compute weighted sum for all hidden units:

$$i_i = \sum_j w_{ji}a_j + b_i$$

- Compute activation of hidden units (here also denoted as $\vec{a}$):

$$a_i = F_i(i_i) = \frac{1}{1 + e^{-i_i}}$$

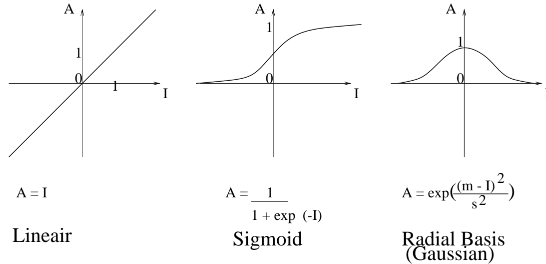
- Compute activation of output units s:

$$s_i = F_i(\sum_j w_{ji}a_j + b_i)$$

Activation Functions

One can use a multitude of activation functions.

Different activation functions are useful for representing different functions (a-priori knowledge)



The hidden layer uses usually sigmoid functions or Radial Basis functions (more local)

The output layer uses mostly a linear activation functions (so that all functions can be learned)

Backpropagation

Minimalise error function:

$$E = \frac{1}{2} \sum_i (d_i - s_i)^2$$

By adjusting the weights through gradient descent:

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial i_i} \frac{\partial i_i}{\partial w_{ji}}$$

- Learningrule with learningrate α :

$$\Delta w_{ji} = -\alpha \frac{\partial E}{\partial w_{ji}} = \alpha \delta_i a_j$$

- Output unit:

$$\delta_i = -\frac{\partial E}{\partial i_i} = (d_i - s_i) F_i'(i_i)$$

- Hidden unit:

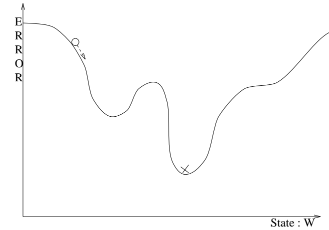
$$\delta_i = F_i'(i_i) \sum_{o \in Outputs} \delta_o w_{io}$$

Here $F_i'(i_i) = (1 - a_i) a_i$, if F is the sigmoid function.

And $F_i'(i_i) = 1$, if F is the linear function.

Learning as search

Gradient descent on the errorlandscape works as follows:



Problems:

- **Local minima**. If the network is trapped in a local minima, it cannot escape anymore with gradient descent.
- **Long plain areas.** If the errorlandscape is very flat in some area, learning becomes very slow (the gradient is very small).
- Learning an optimal network is an NP-hard problem.

More about backpropagation

- Gradient descent over whole network's weight vector
- Easily generalizable to different directed graphs.
- Will find local minimum and not necessarily global minimum. Can still work well with multiple restarts.
- Sometimes use momentum term (batch learning):

$$\Delta w_{ij}(t) = \gamma \delta_j x_i + \mu \Delta w_{ij}(t-1)$$

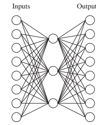
- Minimalises error over all training examples, will it generalize well to new unseen examples?
  - Be careful with too many hidden units → overfitting
  - Works well with enough training examples

- Learning can take thousands of iterations $\rightarrow$ slow!
- Use of previously trained network is fast.

Representation of hidden units

Learning Hidden Layer Representations

A network:



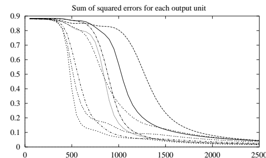
Learned hidden layer representation:

Input		Hidden Values				Output
10000000	$\rightarrow$	.89	.04	.08	$\rightarrow$	10000000
01000000	$\rightarrow$	.01	.11	.88	$\rightarrow$	01000000
00100000	$\rightarrow$	.01	.97	.27	$\rightarrow$	00100000
00010000	$\rightarrow$	.99	.97	.71	$\rightarrow$	00010000
00001000	$\rightarrow$	.03	.05	.02	$\rightarrow$	00001000
00000100	$\rightarrow$	.22	.99	.99	$\rightarrow$	00000100
00000010	$\rightarrow$	.80	.01	.98	$\rightarrow$	00000010
00000001	$\rightarrow$	.60	.94	.01	$\rightarrow$	00000001

94 lecture slides for textbook *Machine Learning*, T. Mitchell, McGraw Hill, 1997

Evolution of leerning process

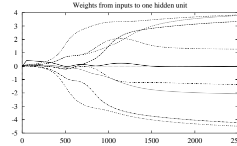
Training



95 lecture slides for textbook *Machine Learning*, T. Mitchell, McGraw Hill, 1997

Evolution of learning process (2)

Training



97 lecture slides for textbook *Machine Learning*, T. Mitchell, McGraw Hill, 1997

Generalization in RL

Until now we have used **lookup-table** representations for storing the Q- and V-functions.

Although this can result in very accurate approximations of the optimal Q- and V-functions, they have a number of **disadvantages**:

- For continuous state/action spaces we cannot directly use it.
- The number of states can be very huge. Note that the number of states grows exponentially with the number of state variables (dimensions).
- We have to visit every state to find out the optimal action in that state.

For more complex problems we have to make use of a **function approximator** which can **generalise**.

Function approximation

A **function approximator** learns a mapping from the state s and action a to the Q-value $Q(s, a)$.

Here, s is usually a vector with N variable values.

Examples:

Feedforward Neural networks: Can generalize very well and are good in pattern recognition.

Tesauro used neural networks with backpropagation for TD-gammon. The system was very successful, also thanks to the "Smooth evaluation functions".

A disadvantage of neural networks is that they are not suited for model-based RL and usually learn quite slow.

RL and Neural Networks

An often used function approximator for RL is a neural network.

Now, we do not work anymore with discrete states, but with a continuous approximation of the value functions.

Neural networks with Q-learning can use 1 big network in which output units represent the Q-values of the different (discrete) actions.

But they can also use 1 neural network for each individual action. Then we get networks f_a which map the input s_t to $Q(s_t, a)$.

Temporal difference learning with neural networks

We are interested in learning $V(\vec{s})$ where $\vec{s}$ is the state-vector or input-vector which the agent receives as information.

We create an input-layer with k inputs: $(I_1, I_2, \ldots, I_k)$.

We create a hidden-layer with l hidden units: $(H_1, H_2, \ldots, H_l)$

Finally we create one output unit for representing the V-value given an input: O_1.

We also make weights w_{IH} for the weights from input to hidden units, and weights w_{HO} for weights between hidden units and the output unit.

Finally we use a bias-units for all hidden-units: b_h and a bias for the output unit: b_o.

We initialize all weights and biases to values between -0.3 and 0.3

Computing V(s) given a state-vector

The first step is to forward-propagate the input to the output unit:

1. Copy state-vector s to input-layer I
2. Compute activations of all hidden units (for $i = 1 \ to \ l$):

   $$H_i = F(\sum_j w_{IH}(j,i)I_j + b_h(i))$$

   Where F is e.g. the sigmoid function.
3. Compute activation of output unit

   $$O_1 = F(\sum_j w_{HO}(j,1)H_j + b_o(1))$$

Now the value V(s) is represented by $O_1 = V(s)$.

After each step of the agent, the agent receives the information: s_t, r_{t+1}, s_{t+1}.

Now the agent can compute the desired value for state s_t as the value: $d = r_{t+1} + \gamma V(s_{t+1})$. Here $V(s_{t+1})$ needs the forward propagation step.

Backprop with TD-learning

Now we have a training-pattern s_t, d.

1. First we use forward-propagation on s_t to obtain $O_1 = V(s_t)$.
2. Then we compute the delta values of the output unit (for linear activation function):

   $$\delta_O(1) = (d - O_1)$$
3. Then we compute the delta values of all hidden units $i = 1 \ to \ l$ (for signoid activation function):

   $$\delta_H(i) = \delta_O(1)w_{HO}(i,1)H_i(1 - H_i)$$
4. Then we change the hidden-output weights for $i = 1 \ to \ l$:

   $$w_{HO}(i,1) = w_{HO}(i,1) + \alpha\delta_O(1) * H_i$$
5. then we change the input-hidden weights for $i = 1 \ to \ k$ and $j = 1 \ to \ l$

   $$w_{IH}(i,j) = w_{IH}(i,j) + \alpha\delta_H(j)I_i$$

Adapting the Neural networks with Q-learning

The agent receives information: $(s_t, a_t, r_{t+1}, s_{t+1})$.

We want to adapt the output of network for the actions $a = a_t$.

For this we have to know the desired output value:

$$\hat{d} = \gamma \max_{a'} f_{a'}(s_{t+1}) + r_{t+1}$$

Then we use $(s_t, \hat{d})$ as a training example for the action-network f_a. We can use backpropagation to adapt the neural network.

This we repeat constantly until the system hardly changes anymore.

Applications: Backgammon, robot control, chess, go, elevator control

---

Disadvantages of Neural networks

- Often become trapped in local minima.
- No direct way to handle missing values.
- Sometimes very slow learning process
- Sometimes the network forgets previously learned knowledge if it is trained on a different function (learning-interference)
- Can cost lots of experimental time to find right topology and learning parameter(s)
- It is not simple to use a-priori knowledge in the network
- Solution is not readable by humans

Advantages of Neural networks

- Can represent all functions
- Is well able to deal with noise
- Handles redundancy very well
- Can deal well with high dimensional input spaces
- Is robust against removing neurons → graceful degradation