Learning Neural Networks: Perceptron and Backpropagation

Jaap Murre

University of Amsterdam and

University of Maastricht

jaap@murre.com

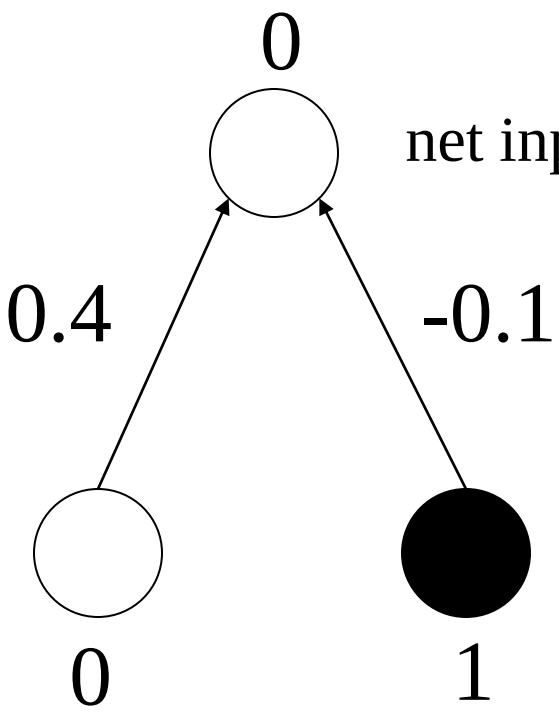
Two main forms of learning

- Associative (Hebbian) learning
- Error-correcting learning
  - Perceptron
  - Delta rule
  - Error-backpropagation
    - aka generalized delta rule
    - aka multilayer perceptron

The Perceptron by Frank Rosenblatt (1958, 1962)

- Two-layers
- binary nodes (McCulloch-Pitts nodes) that take values 0 or 1
- continuous weights, initially chosen randomly

Very simple example



net input = $0.4 \times 0 + -0.1 \times 1 = -0.1$

Learning problem to be solved

- Suppose we have an input pattern (0 1)
- We have a single output pattern (1)
- We have a net input of -0.1, which gives an output pattern of (0)
- How could we adjust the weights, so that this situation is remedied and the spontaneous output matches our target output pattern of (1)?

Answer

- Increase the weights, so that the net input exceeds 0.0
- E.g., add 0.2 to all weights
- Observation: Weight from input node with activation 0 does not have any effect on the net input
- So we will leave it alone

Perceptron algorithm in words

For each node in the output layer:

- Calculate the error, which can only take the values -1, 0, and 1
- If the error is 0, the goal has been achieved. Otherwise, we adjust the weights
- Do not alter weights from inactivated input nodes
- Increase the weight if the error was 1, decrease it if the error was -1

Perceptron algorithm in rules

- weight change = some small constant × (target activation - spontaneous output activation) × input activation
- if speak of *error* instead of the "target activation minus the spontaneous output activation", we have:
- weight change = some small constant × error × input activation

Perceptron algorithm as equation

- If we call the input node *i* and the output node *j* we have:

$$\Delta w_{ji} = \mu (t_j - a_j) a_i = \mu \delta_j a_i$$

- Δw_{ji} is the weight change of the connection from node *i* to node *j*
- a_i is the activation of node *i*, a_j of node *j*
- t_j is the target value for node *j*
- δ_j is the error for node *j*
- The learning constant μ is typically chosen small (e.g., 0.1).

Perceptron algorithm in pseudo-code

```
Start with random initial weights (e.g., uniform random in [-.3,.3])

Do
{
  For All Patterns p
  {
    For All Output Nodes j
    {
      CalculateActivation(j)

      Error_j = TargetValue_j_for_Pattern_p - Activation_j

      For All Input Nodes i To Output Node j
      {
        DeltaWeight = LearningConstant * Error_j * Activation_i
        Weight = Weight + DeltaWeight
      }
    }
  }
}
Until "Error is sufficiently small" Or "Time-out"
```

Perceptron convergence theorem

- *If* a pattern set can be represented by a two-layer Perceptron, …
- the Perceptron learning rule will always be able to find some correct weights

The Perceptron was a big hit

- Spawned the first wave in ‘connectionism’
- Great interest and optimism about the future of neural networks
- First neural network hardware was built in the late fifties and early sixties

Limitations of the Perceptron

- Only binary input-output values
- Only two layers

Only binary input-output values

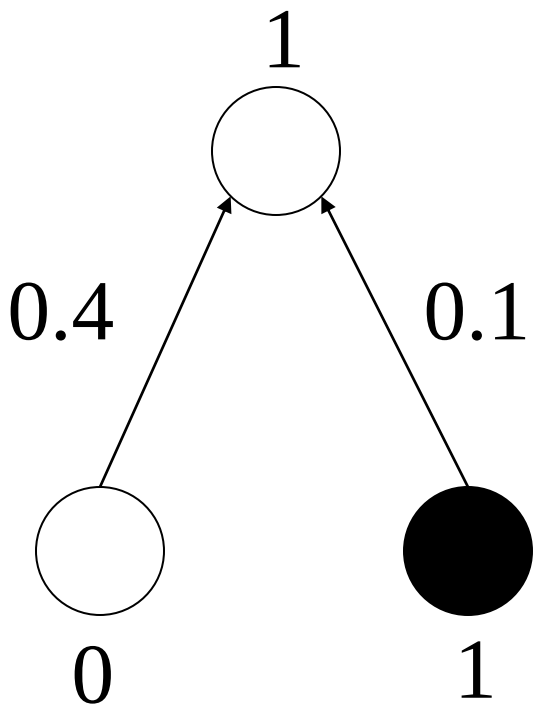
- This was remedied in 1960 by Widrow and Hoff
- The resulting rule was called the delta-rule
- It was first mainly applied by engineers
- This rule was much later shown to be equivalent to the Rescorla-Wagner rule (1976) that describes animal conditioning very well

Only two layers

- Minsky and Papert (1969) showed that a two-layer Perceptron cannot represent certain logical functions
- Some of these are very fundamental, in particular the exclusive or (XOR)
- Do you want coffee XOR tea?

Exclusive OR (XOR)

In	**Out**
0 1	1
1 0	1
1 1	0
0 0	0



An extra layer is necessary to represent the XOR

- No solid training procedure existed in 1969 to accomplish this
- Thus commenced the search for the third or hidden layer

Minsky and Papert book caused the ‘first wave’ to die out

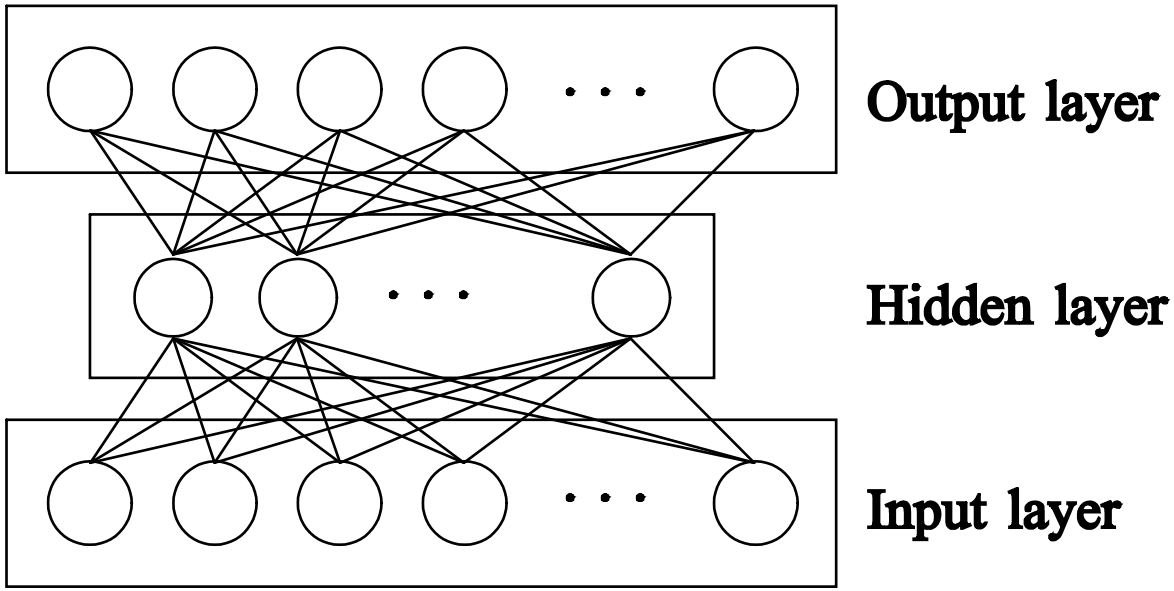
- GOOFAI was increasing in popularity
- Neural networks were very much out
- A few hardy pioneers continued
- Within five years a variant was developed by Paul Werbos that was immune to the XOR problem, but few noticed this
- Even in Rosenblatt’s book many examples of more sophisticated Perceptrons are given that can learn the XOR

Error-backpropagation

- What was needed, was an algorithm to train Perceptrons with more than two layers
- Preferably also one that used continuous activations and non-linear activation rules
- Such an algorithm was developed by
  - Paul Werbos in 1974
  - David Parker in 1982
  - LeCun in 1984
  - Rumelhart, Hinton, and Williams in 1986

Error-backpropagation by Rumelhart, Hinton, and Williams

Meet the hidden layer

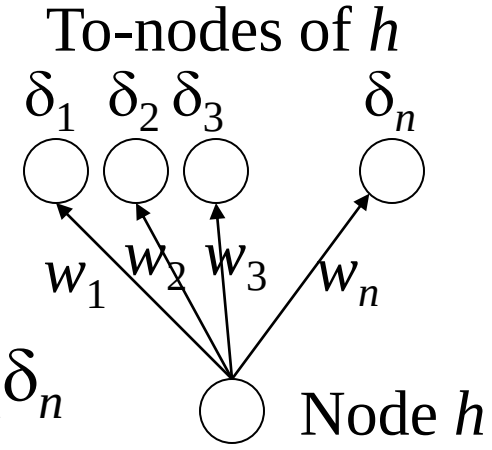


The problem to be solved

- It is straightforward to adjust the weights to the output layer, using the Perceptron rule
- But how can we adjust the weights to the hidden layer?

The backprop trick

- To find the error value for a given node *h* in a hidden layer, …
- Simply take the weighted sum of the errors of all nodes connected from node *h*
- i.e., of all nodes that have an incoming connection from node *h*:



This is backpropgation of errors

$$\delta_h = w_1\delta_1 + w_2\delta_2 + w_3\delta_3 + \ldots + w_n\delta_n$$

Characteristics of backpropagation

- Any number of layers
- Only feedforward, no cycles (though a more general versions does allow this)
- Use continuous nodes
  - Must have differentiable activation rule
  - Typically, *logistic*: S-shape between 0 and 1
- Initial weights are random
- Total error never increases (gradient descent in error space)

The gradient descent makes sense mathematically

- It does not guarantee high performance
- It does not prevent local minima
- The learning rule is more complicated and tends to slow down learning unnecessary when the logistic function is used

Logistic function

- S-shaped between 0 and 1
- Approaches a linear function around $x = 0$
- Its rate-of-change (derivative) for a node with a given activation is:

$$\text{activation} \times (1 - \text{activation})$$

Backpropagation algorithm in rules

- weight change = some small constant × error × input activation
- For an output node, the error is:

  error = (target activation - output activation) × output activation × (1 - output activation)

- For a hidden node, the error is:

  error = weighted sum of to-node errors × hidden activation × (1 - hidden activation)

Weight change and momentum

- backpropagation algorithm often takes a long time to learn
- So, the learning rule is often augmented with a so called momentum term
- This consist in adding a fraction of the old weight change
- The learning rule then looks like:

  weight change = some small constant × error × input activation + momentum constant × old weight change

Backpropagation in equations I

- If j is a node in an output layer, the error δ_j is:

  $$\delta_j = (t_j - a_j)\, a_j(1-a_j)$$

- where a_j is the activation of node j
- t_j is its target activation value, and
- δ_j its error value

Backpropagation in equations II

- If j is a node in a hidden layer, and if there are k nodes 1, 2, …, k, that receive a connection from j, the error δ_j is:
- $\delta_j = (w_{1j}\,\delta_1 + w_{2j}\,\delta_1 + \ldots + w_{kj}\delta_k)\,a_j\,(1\text{-}a_j)$
- where the weights w_{1j}, w_{2j}, …, w_{kj} belong to the connections from hidden node j to nodes 1, 2, …, k.

Backpropagation in equations III

- The backpropagation learning rule (applied at time *t*) is:

$$\Delta w_{ji}(t) = \mu \, \delta_j a_i + \beta \Delta w_{ji}(t\text{-}1)$$

- where $\Delta w_{ji}(t)$ is the change in the weight from node *i* to node *j* at time *t*,
- The learning constant μ is typically chosen rather small (e.g., 0.05).
- The momentum term β is typically chosen around 0.5.

NetTalk: Backpropagation's 'killer-app'

- Text-to-speech converter
- Developed by Sejnowski and Rosenberg (1986)
- Connectionism's answer to DECTalk
- Learned to pronounce text with an error score comparable to DECTalk
- Was trained, not programmed
- Input was letter-in-context, output phoneme

Despite its popularity backpropagation has some disadvantages

- Learning is slow
- New learning will rapidly *overwrite* old representations, unless these are interleaved (i.e., repeated) with the new patterns
- This makes it hard to keep networks up-to-date with new information (e.g., dollar rate)
- This also makes it very implausible from as a psychological model of human memory

Good points

- Easy to use
  - Few parameters to set
  - Algorithm is easy to implement
- Can be applied to a wide range of data
- Is very popular
- Has contributed greatly to the ‘new connectionism’ (second wave)

Conclusion

- Error-correcting learning has been very important in the brief history of connectionism
- Despite its limited plausibility as a psychological model of learning and memory, it is nevertheless used widely (also in psychology)