Deep Feedforward Networks

Master-Seminar: Introduction to Deep Learning

Bodo Burger

2016-01-13

Ludwig-Maximilians-Universität

Overview

1. Basics of artificial neural networks
    - Historical notes
2. Architecture design
    - Output units
    - Hidden units
3. Gradient-based learning
    - Back-propagation
    - Other differentiation algorithms
4. Application
    - Odd/Even
    - “multiple of”

Basics of Artificial Neural Networks

Deep feedforward networks?

What are deep feedforward networks?

- *the* basic deep learning model
- names: feedforward neural networks, multilayer perceptrons (MLPs)

What is a network?

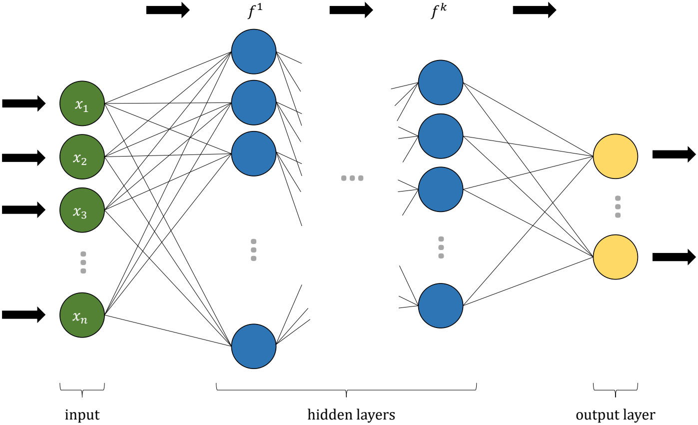
- composing together many different functions

$$f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$$

  $f^{(1)}$ would be the first layer, $f^{(2)}$ the second layer, and so on …

- length of the chain: depth of the network
- **input:** $\boldsymbol{x}$
  output layer: last layer
  hidden layers: layers in between

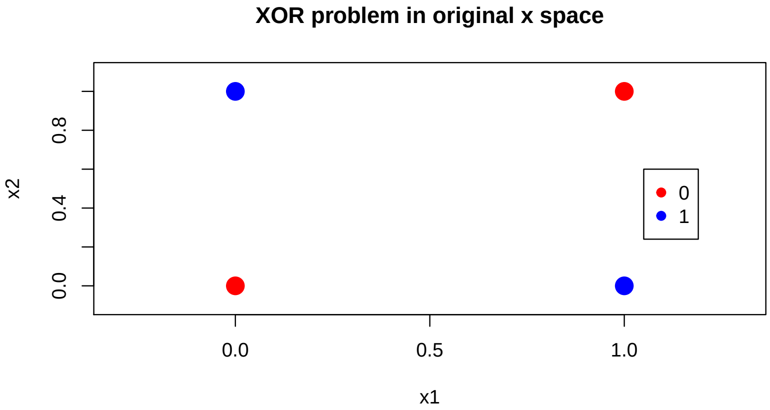
Graphical representation of a neural network



Function approximation

- *given:* noisy example $\boldsymbol{x}$, corresponding label y
- The goal of the neural network is to approximate a function $f^*(\boldsymbol{x})$ ($y \approx f^*(\boldsymbol{x})$).
- Find optimal parameters $\boldsymbol{\theta}$ so that a function f maps $(\boldsymbol{x}, \boldsymbol{\theta})$ on $\boldsymbol{y}$.

Example: XOR



How to overcome the limitations of a linear model?

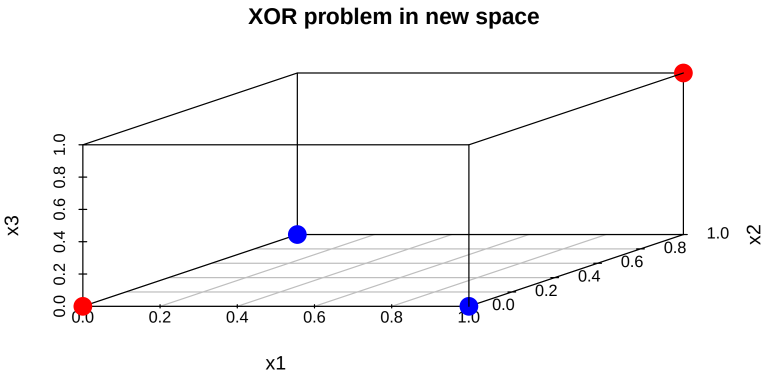
Transform the input $\boldsymbol{x}$ by a nonlinear transformation ϕ.

Possible solutions for ϕ:

1. kernel trick, SVM
2. manually engineer ϕ
3. learn ϕ

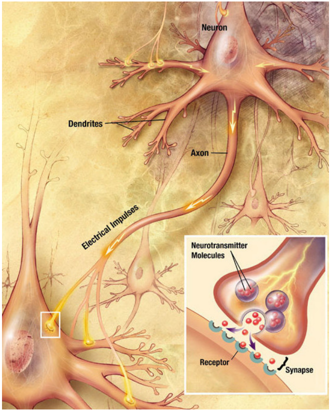
Example: XOR

Adding a third input $x_3 = x_1 \cdot x_2$:



Why neural?

Human brain: around 85 billion neurons

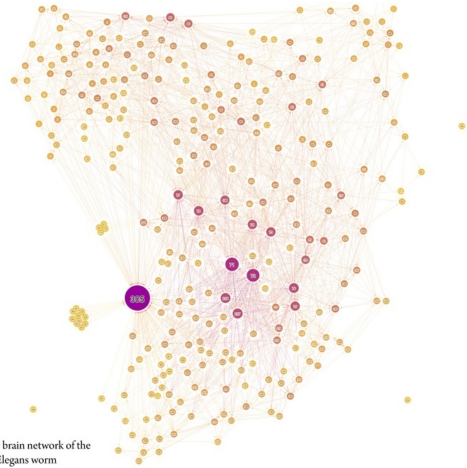


left: `https://commons.wikimedia.org/wiki/File:Purkinje_cell_by_Cajal.png`

right: `https://commons.wikimedia.org/wiki/File:Chemical_synapse_schema_cropped.jpg`

Why neural?

C. elegans: worm with around 300 neurons



The brain network of the
C. Elegans worm

Why neural?

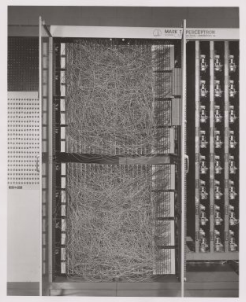
- **More precise:** artificial neural networks (ANN)
- architecture is inspired by biological neural networks
- **but:** not an exact model of the function of a brain
- development of ANN shaped by maths / computer science

Basics of Artificial Neural Networks

Historical Notes

Historical notes

- 1943: McCulloch / Pitts create first computational model for neural networks
- late 1940s: Hebbian learning
- 1951: Minsky develops SNARC, possibly the first ANN
- 1958: Rosenblatt creates the Mark I perceptron



https://upload.wikimedia.org/wikipedia/en/5/52/Mark_I_perceptron.jpeg

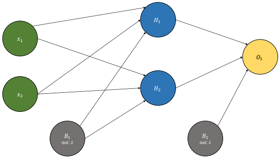
Historical notes

- 1969: Minsky/Papert "Perceptrons: An Introduction to Computational Geometry"
- limited processing power of computers at the time
- period of defunding of AI research
- 1986: Rumelhart, Hinton and Williams on MLP and back-propagation
- Schmidhuber: "Deep learning since 1991"

New developments

- since 2000s: many breakthroughs because of increasing processing power and massive data availability
- Google Translate
- Voice recognition in smartphones
- Amazon Echo / Alexa speech recognition
- DeepMind's AlphaGo
- Self-driving cars

Architecture Design



What is the structure of the neural network?

- How many units?
- Which units?
- How do you connect these units?

Chain structure

Typical design approach: multiple layers that consist of multiple units are linked in a chain structure.

First layer is a function of the inputs:

$$\boldsymbol{h}^{(1)} = g^{(1)}\left(\boldsymbol{W}^{(1)T}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right)$$

Second layer is a function of the first layer's output:

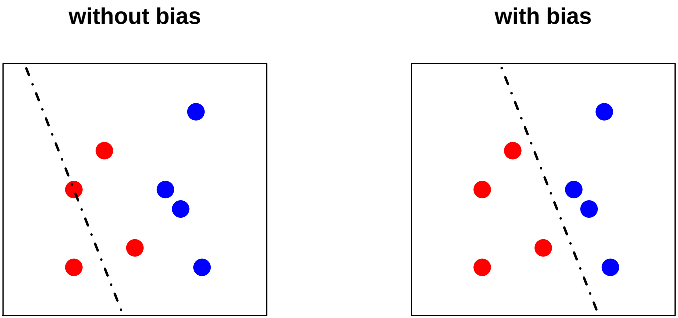
$$\boldsymbol{h}^{(2)} = g^{(2)}\left(\boldsymbol{W}^{(2)T}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}\right)$$

...

$g^{()}$ is called activation function.

Bias

Bias puts the decision boundary at the correct position in the learned space.



Depth and width

Depth of the network

number of layers of the network

Width of a layer

number of units in a layer

Deeper networks often work with thiner layers which means less parameters.

Ideal architecture has to be found via experimentation.

Universal approximation theorem

"In the mathematical theory of artificial neural networks, the universal approximation theorem states that a feed-forward network with a single hidden layer containing a finite number of neurons (i.e., a multilayer perceptron), can approximate continuous functions on compact subsets of $\mathbb{R}^n$, under mild assumptions on the activation function. The theorem thus states that simple neural networks can represent a wide variety of interesting functions when given appropriate parameters; however, it does not touch upon the algorithmic learnability of those parameters."

https://en.wikipedia.org/wiki/Universal_approximation_theorem

Why may learning a function fail?

1. optimization algorithm may not find the parameters of the function
2. overfitting

“No free lunch” theorem

“There is no strategy - applied on all possible problems - that is better than pure guessing.”

https://de.wikipedia.org/wiki/No-free-Lunch-Theoreme

Alternative network designs

Depth and height do not need to be the only parameters of the architecture of a neural network.

Possible modifications:

- Add connections that skip a layer and connect layer i to layer $i+2$
- Do not connect every input of a layer with every output of the preceding layer.

The choice of modifications is very dependent on the actual application!

Alternative network designs: examples

- Convolutional networks (for computer vision)
- Recurrent neural networks (for sequence processing)
    - Long short-term memories (Google voice, Siri)
- Autoencoder (dimensionality reduction)
- Deep belief network
- ...

Architecture Design

Output Units

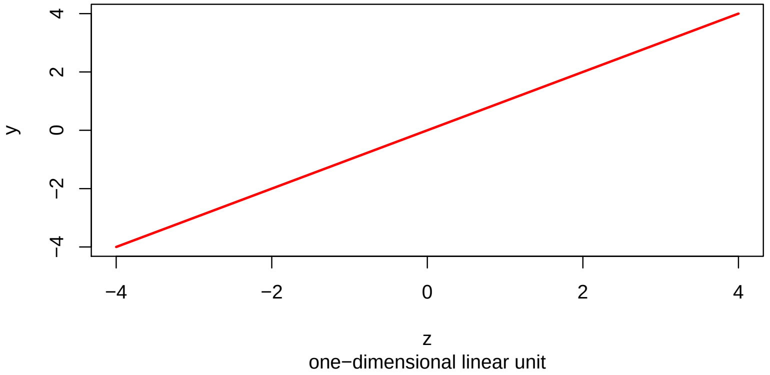
Output Units

The output unit defines what happens to the output $\boldsymbol{h}$ of the last hidden layer. So it is the last transformation on $\boldsymbol{x}$.

The choice of the output unit also determines the cost function.

Linear units

$$\hat{\boldsymbol{y}} = \boldsymbol{z} = \boldsymbol{W}^T \boldsymbol{h} + \boldsymbol{b}$$



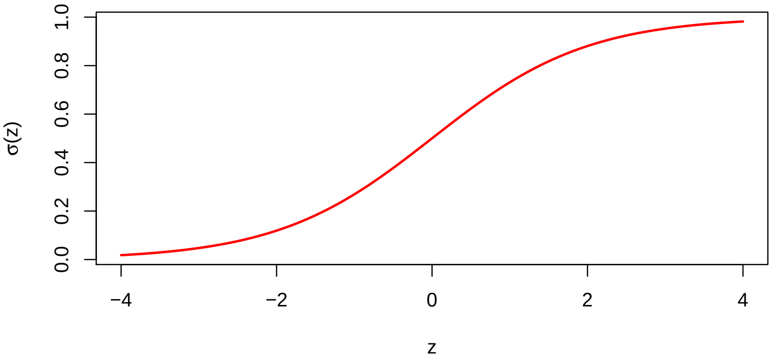
Linear units

$$\hat{\boldsymbol{y}} = \boldsymbol{z} = \boldsymbol{W}^T \boldsymbol{h} + \boldsymbol{b}$$

- used for predicting the mean of a Gaussion distribution
- easy to handle for optimization algorithms

Sigmoid units

Sigmoid function: $\sigma(z) = \frac{1}{1+e^{-z}}$.



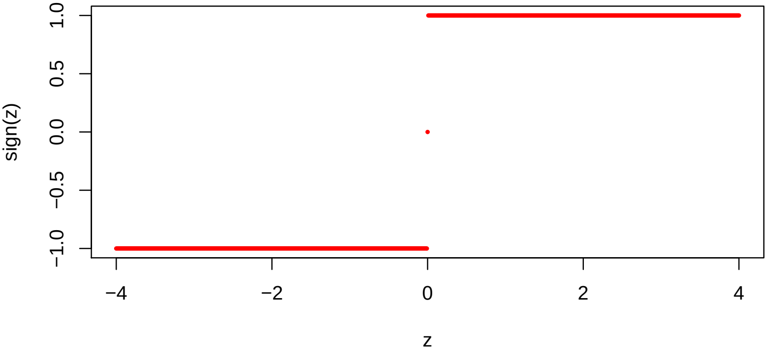
Sigmoid units

Sigmoid function: $\sigma(z) = \frac{1}{1+e^{-z}}$.

$$\hat{y} = \sigma\left(\boldsymbol{w}^T\boldsymbol{h} + b\right)$$

- applying the sigmoid function on a linear unit
- used for predicting binary outputs
- problematic: saturation for large absolute values of z

Signum function



Softmax units

Softmax function: $\text{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$.

- generalization of sigmoid function for multiclass problem
- used for predicting discrete output with k possible values

Architecture Design

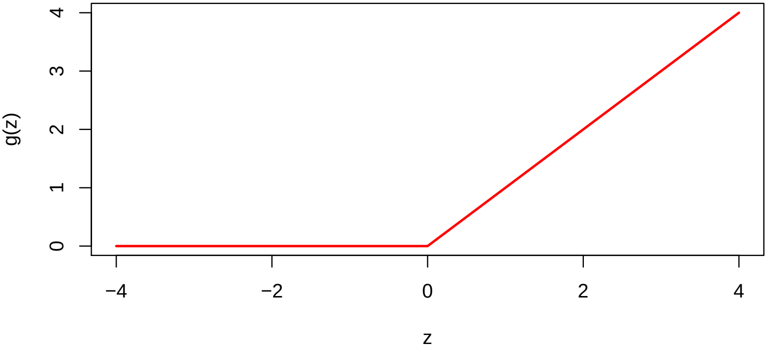
Hidden Units

Design of hidden units

- active area of research, not many definitive theoretical guidelines
- Try out different kinds of units and look which work best.
- good default choice: Rectified linear units
- Hidden units do not have to be differentiable.

Rectified linear units (ReLU)

$$g(z) = \max\{0, z\}$$



Rectified linear units (ReLU)

- typically used with a bias $\boldsymbol{b}$ (set to small, positive value)
- not differentiable at zero, but some generalizations of a ReLU are

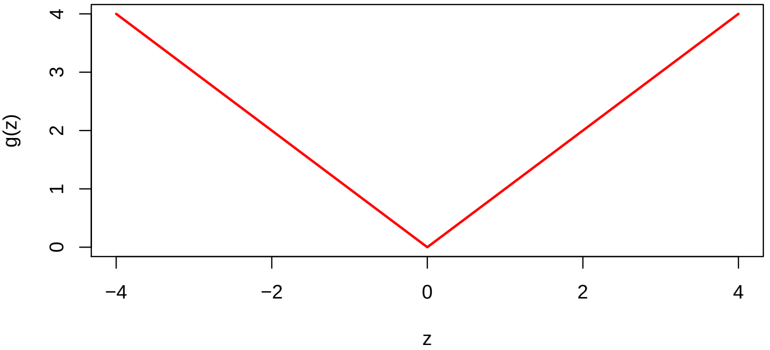
Generalizations of ReLU

There are various generalizations of ReLU:

- absolute value rectification
- leaky ReLU
- parametric ReLU (PReLU)

$$h_i = g(\boldsymbol{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i)$$

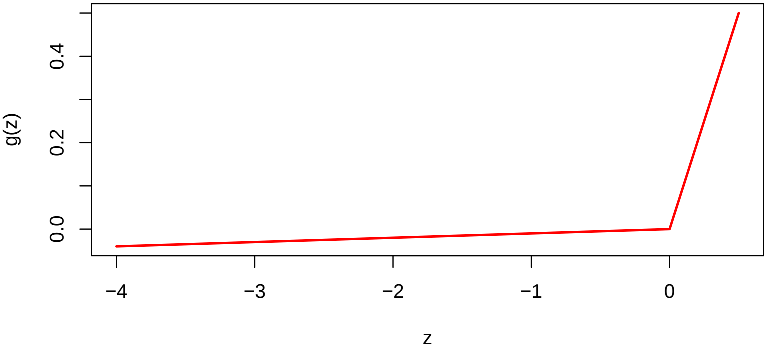
Absolute value rectification: $\alpha_i = -1 \Rightarrow g(z) = |z|$.



Generalizations of ReLU

$$h_i = g(\boldsymbol{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i)$$

leaky ReLU: $\alpha_i = 0.01$.



Generalizations of ReLU

$$h_i = g(\boldsymbol{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i)$$

parametric ReLU: treat α_i as learnable parameter.

Maxout units

$$g(\boldsymbol{z})_i = \max z_j$$

- divide $\boldsymbol{z}$ into groups of k values
- learning a piecewise linear function
- Maxout units avoid catastrophic forgetting.

Logistic sigmoid and hyperbolic tangent

Sigmoid function:

$$g(z) = \sigma(z)$$

Hyperbolic tangent function:

$$g(z) = tanh(z)$$

Closely related, because $tanh(z) = 2\sigma(2z) - 1$.

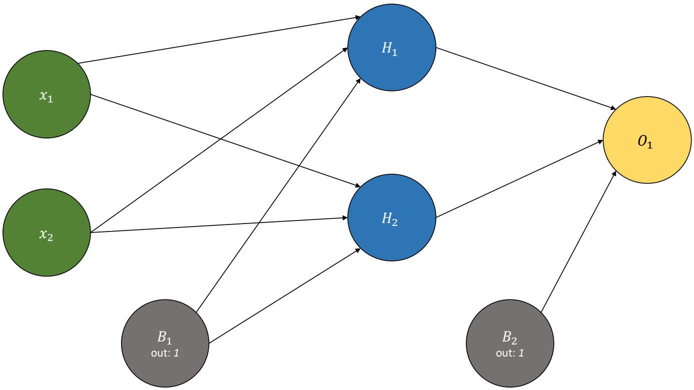
Used e.g. for recurrent networks or autoencoders.

Other hidden units

- radial basis function
- sofplus
- hard tanh

Example: A neural net for the XOR example

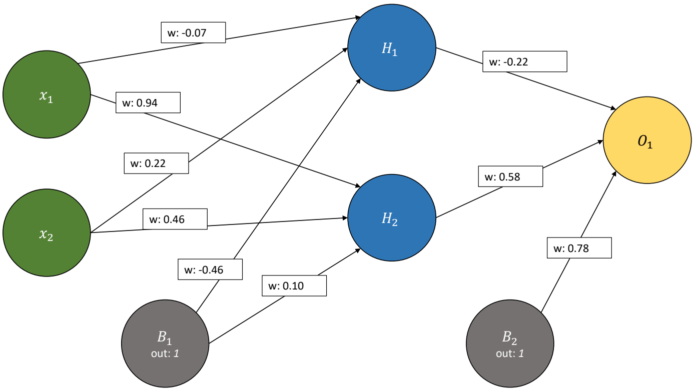
two inputs, one hidden layer with two units



sigmoid activation function for hidden layer and output

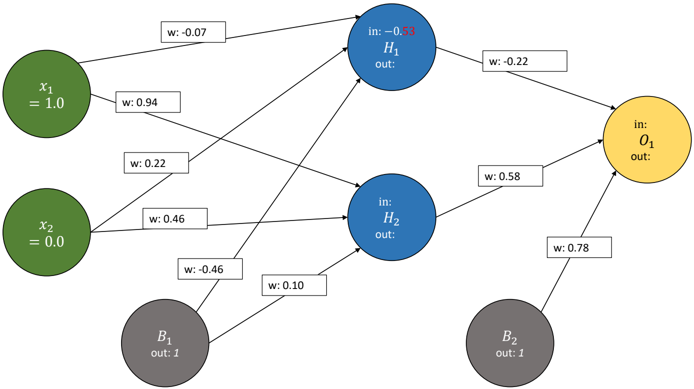
Weight initialisation

initialising random weights for all connections



Forward propagation

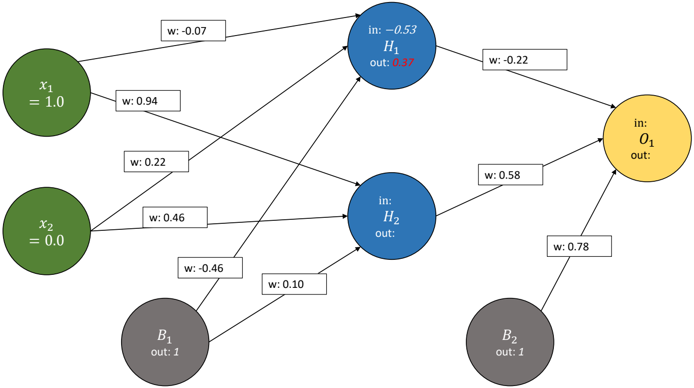
$x_1 = 1, x_2 = 0$:



H_1: in $= -0.07 \cdot 1 + 0.22 \cdot 0 + -0.46 \cdot 1 = -0.53$

Forward propagation

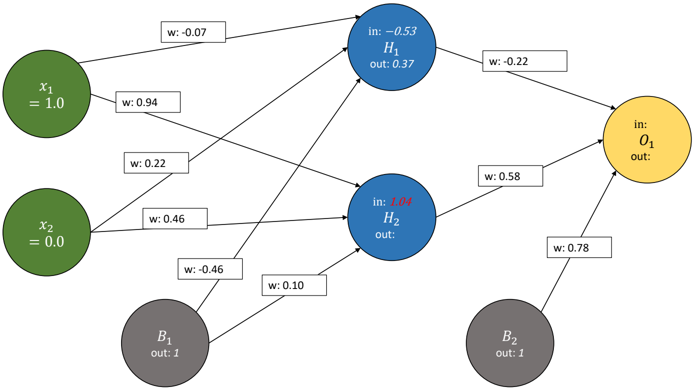
$x_1 = 1, x_2 = 0$:



H_1: out $= \frac{1}{1-e^{-0.53}}$

Forward propagation

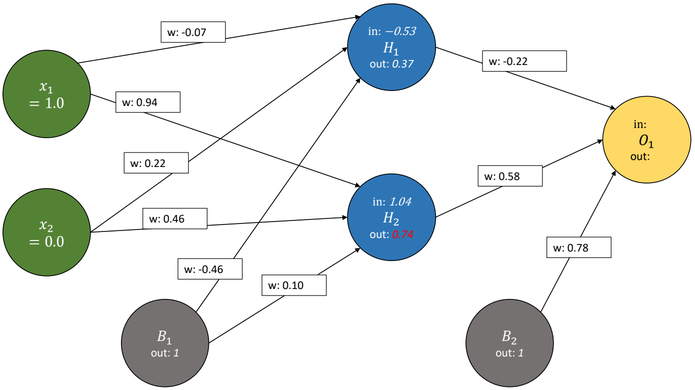
$x_1 = 1, x_2 = 0$:



H_2: in $= -0.07 \cdot 1 + 0.22 \cdot 0 + -0.46 \cdot 1 = -0.53$

Forward propagation

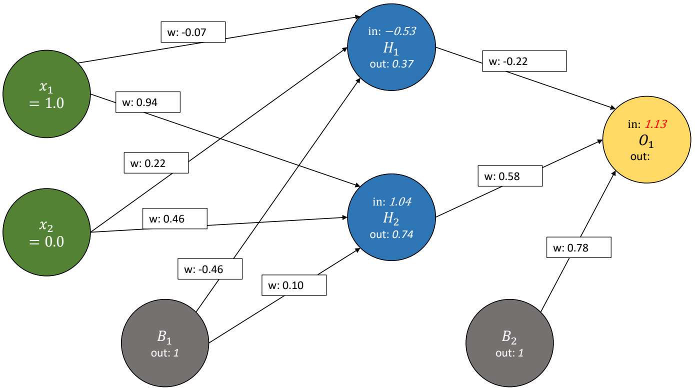
$x_1 = 1, x_2 = 0$:



H_2: out $= \frac{1}{1-e^{-0.53}}$

Forward propagation

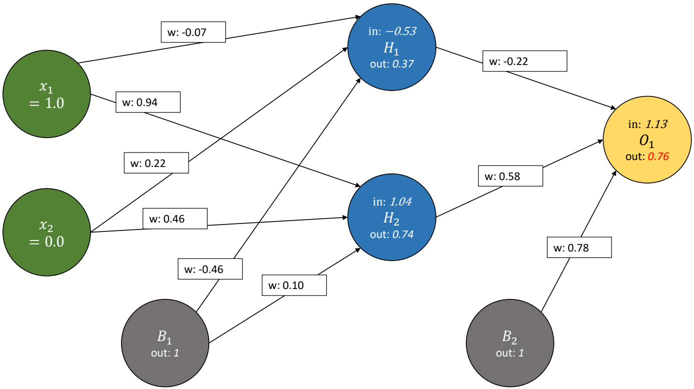
$x_1 = 1, x_2 = 0$:



$$O_1\text{: in} = -0.07 \cdot 1 + 0.22 \cdot 0 + -0.46 \cdot 1 = -0.53$$

Forward propagation

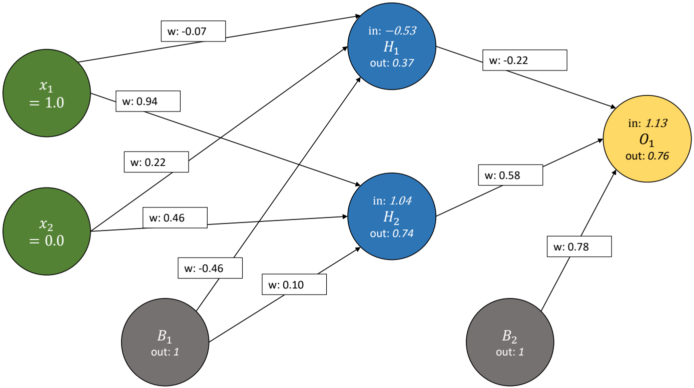
$x_1 = 1, x_2 = 0$:



O_1: out $= \frac{1}{1 - e^{-0.53}}$

Error of the output

Error: $E = f(\boldsymbol{x}) - y = \hat{y} - y$



$E = 0.76 - 1 = -0.24$

Total error

Total error can be defined as

$$E = \frac{1}{2}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2.$$

For our example:

$$\begin{aligned} E &= \frac{1}{2}\left[(0.73-1)^2 + (0.74-1)^2 + (0.76-1)^2 + (0.76-1)^2\right] \\ &= 0.12785 \end{aligned}$$

Gradient-Based Learning

Why gradient-based learning?

- Nonlinearity of neural nets causes loss functions to become non-convex.
- Convex optimization algorithms do not work anymore.
- Use iterative, gradient-based optimization.
- Does not guarantee convergence and results may depend heavily on initial parameters.
- For *very large* data sets, it does make sense to train a linear model or SVM by gradient descent, too.

Gradient-Based Learning

Back-Propagation

Forward propagation and back-propagation

Forward propagation

The information of the inputs $\boldsymbol{x}$ flows through the hidden units to finally produce $\hat{\boldsymbol{y}}$ and the cost $J(\boldsymbol{\theta})$.

Back-propagation

The information of the cost $J(\boldsymbol{\theta})$ flows backwards through the hidden units to calculate the gradient.

Total error is defined as: $E = \frac{1}{2}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$

Gradient of connection from unit i to output O_1:

$$\frac{\partial E}{\partial w_{O_1,i}} = \frac{\partial E}{\partial out_{O_1}} \cdot \frac{\partial \text{out}_{O_1}}{\partial \text{in}_{O_1}} \cdot \frac{\partial \text{in}_{O_1}}{\partial w_{O_1,i}}$$

If we define $\delta_{O_1} = (\hat{y}_i - y_i) \cdot \sigma'_{O_1}$, then we get

$$\frac{\partial E}{\partial w_{O_1,i}} = \delta_{O_1}\text{out}_i,$$

with $\text{out}_{O_1} = \hat{y}_i$ and $\frac{\partial \text{out}_{O_1}}{\partial \text{in}_{O_1}} = \sigma'_{O_1}$.

Calculating the gradient (hidden layer)

Gradient of connection from unit i to unit j:

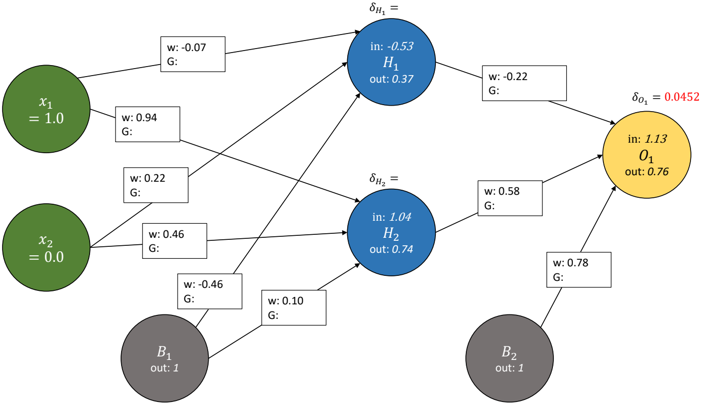
$$\frac{\partial E}{\partial w_{j,i}} = \frac{\partial E}{\partial out_j} \cdot \frac{\partial \text{out}_j}{\partial \text{in}_j} \cdot \frac{\partial \text{in}_j}{\partial w_{j,i}}$$

Node delta is now defined as $\delta_j = \sigma_j' \sum_k w_{k,j} \cdot \delta_k$. We get

$$\frac{\partial E}{\partial w_{j,i}} = \delta_j \text{out}_i.$$

Back-propagation for the XOR-example

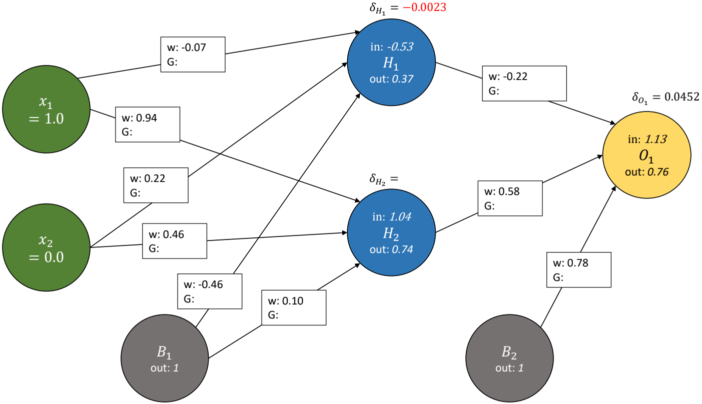
Delta of output node:



$$\delta_{O_1} = -(-0.24) \cdot d\sigma(1.13) = 0.24 \cdot \sigma(1.13)(1 - \sigma(1.13)) = 0.0452$$

Back-propagation for the XOR-example

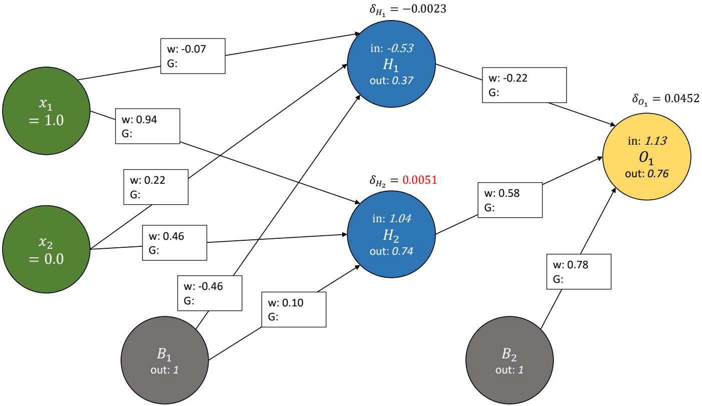
Delta of hidden node 1:



$$\delta_{H_1} = d\sigma(-0.53) \cdot (-0.22 \cdot 0.0452) = -0.0023$$

Back-propagation for the XOR-example

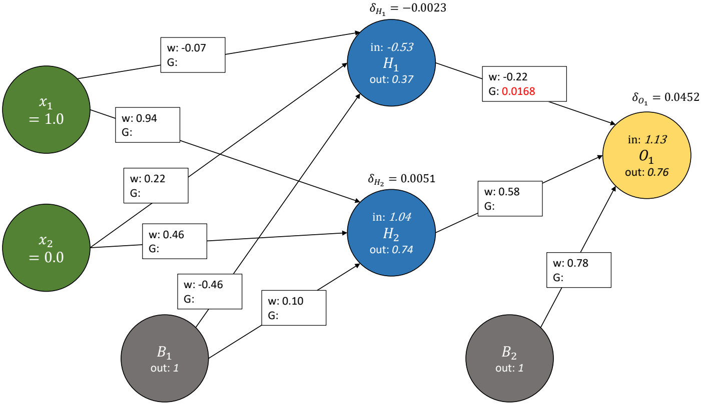
Delta of hidden node 2:



$$\delta_{H_2} = d\sigma(1.04) \cdot (0.58 \cdot 0.0452) = 0.0051$$

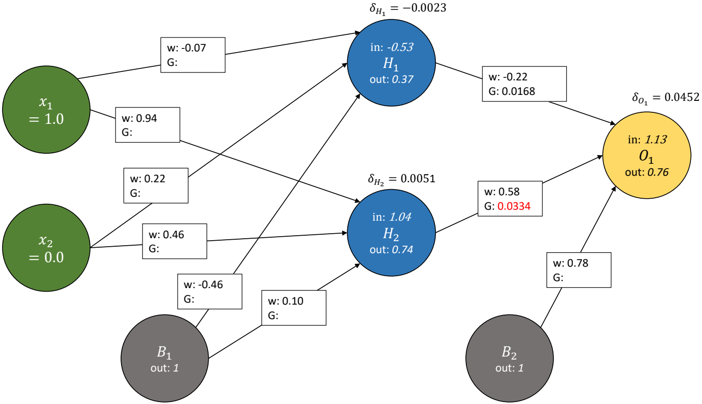
Back-propagation for the XOR-example

Gradient from H_1 to O_1:



$$G_{O_1,H_1} = 0.37 \cdot 0.0452 = 0.0168$$

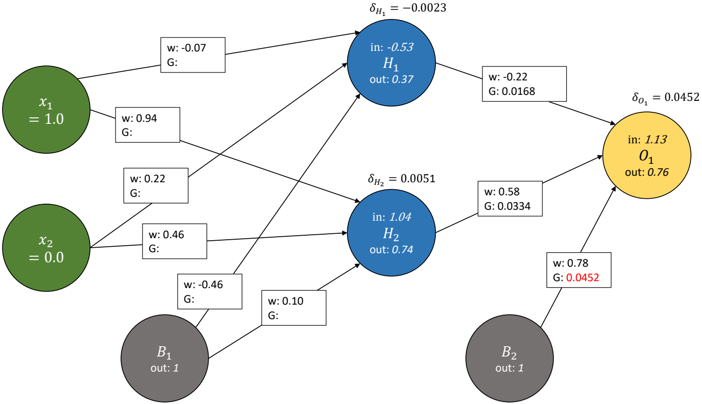
Gradient from H_2 to O_1:



$$G_{O_1,H_2} = 0.74 \cdot 0.0452 = 0.0334$$

Back-propagation for the XOR-example

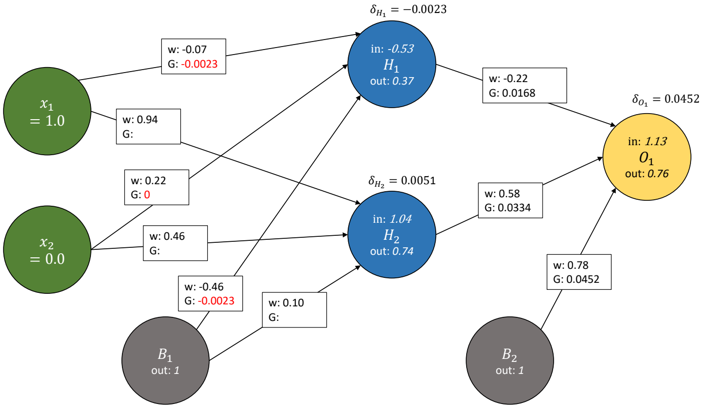
Gradient from B_2 to O_1:



$$G_{O_1,B_2} = 1 \cdot 0.0452 = 0.0452$$

Back-propagation for the XOR-example

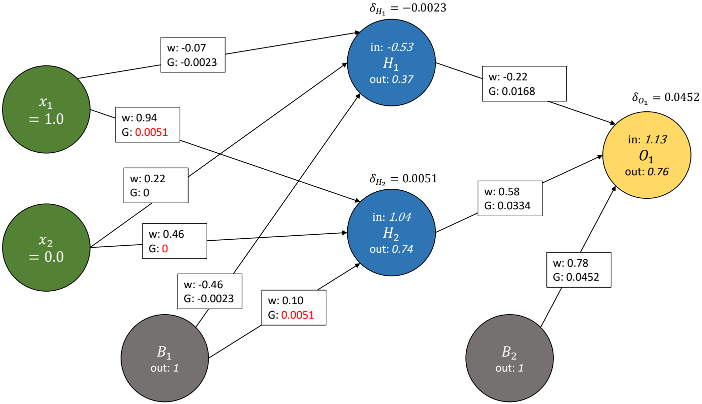
Gradient from x_1, x_2, B_1 to H_1:



$$G_{H_1,\cdot} = \begin{pmatrix} -0.0023 & 0 & -0.0023 \end{pmatrix}$$

Back-propagation for the XOR-example

Gradient from x_1, x_2, B_1 to H_2:



$$G_{H_2,\cdot} = \begin{pmatrix} 0.0051 & 0 & 0.0051 \end{pmatrix}$$

Weight update

$$\Delta w^{(t)} = -\epsilon \frac{\partial E}{\partial w^{(t)}} + \alpha \Delta w^{(t-1)}$$

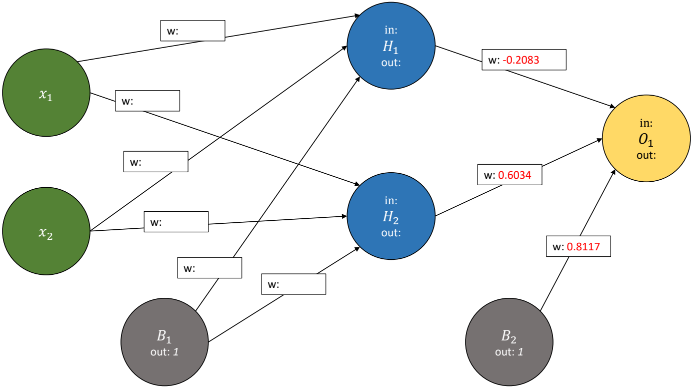
- learning rate ϵ
  - too high: may fail to converge
  - too low: very slow learning
- momentum α
  - $0 \leq \alpha < 1$
  - can speed up learning
  - can help to avoid local minima
- ϵ and α have to be determined by trial and error or by experience

Stochastic vs batch learning

- stochastic learning:
    - random select a training set element, modify weights and repeat
    - can help to avoid local minima, but slow
- batch learning:
    - gradients for each training set element are summed up, then weights are updated
    - faster as it incorporates the average error of all elements
- mini-batch: select several training set elements and compute a weight update

Weight update

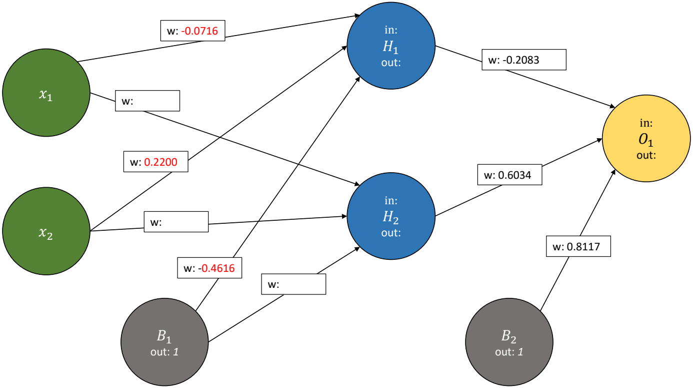
New weights $w_{O_1,\cdot}$:



$$w_{O_1,\cdot} = \begin{pmatrix} -0.22 & 0.58 & 0.78 \end{pmatrix} + 0.7 \cdot \begin{pmatrix} 0.0168 & 0.0334 & 0.0452 \end{pmatrix} + 0.3 \cdot 0$$

Weight update

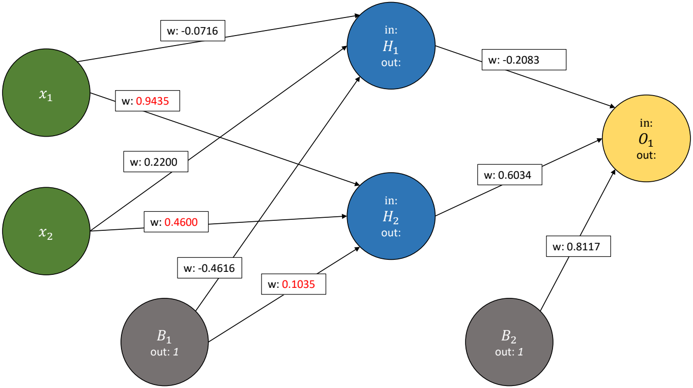
New weights $w_{H_1,\cdot}$:



$$w_{H_1,\cdot} = \begin{pmatrix} -0.07 & 0.22 & -0.46 \end{pmatrix} + 0.7 \cdot \begin{pmatrix} -0.0023 & 0 & -0.0023 \end{pmatrix} + 0.3 \cdot 0$$

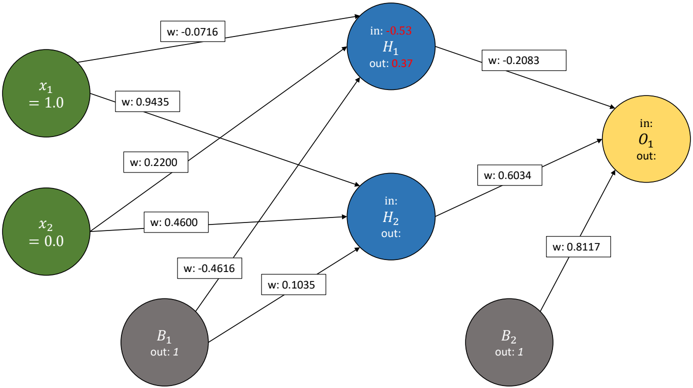
Weight update

New weights $w_{H_2,\cdot}$:

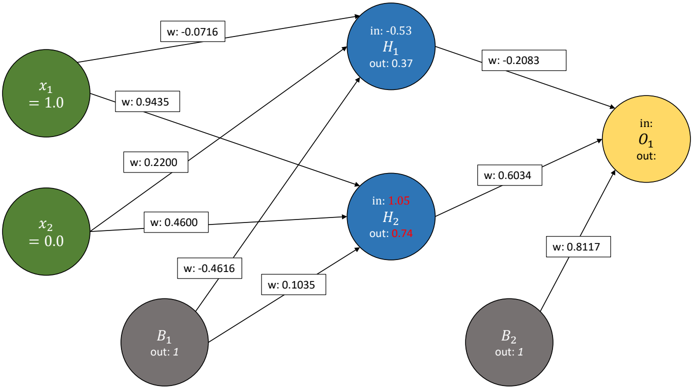


$$w_{H_2,\cdot} = \begin{pmatrix} 0.94 & 0.46 & 0.10 \end{pmatrix} + 0.7 \cdot \begin{pmatrix} 0.0051 & 0 & 0.0051 \end{pmatrix} + 0.3 \cdot 0$$

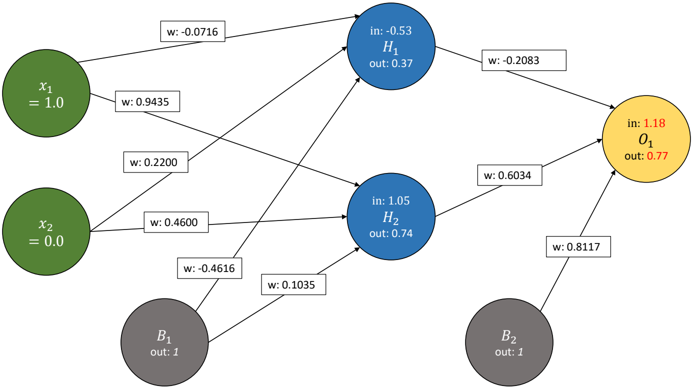
Test: forward propagation with new weights



Test: forward propagation with new weights



Test: forward propagation with new weights



Total error for new weights

Total error before adjusting weights was 0.12785.

Now:

$$
\begin{aligned}
E &= \frac{1}{2}\left[(0.74-1)^2 + (0.75-1)^2 + (0.77-1)^2 + (0.77-1)^2\right] \\
&= 0.11795
\end{aligned}
$$

Alternative cost function

Cross-entropy error function:

$$J = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log \hat{y}_i + (1 - y_i)\log(1 - \hat{y}_i)\right]$$

The node delta for the output layer becomes: $\delta_O = \hat{y} - y$.

Cross-entropy error function often performs better than MSE for learning a classification task.

Gradient-Based Learning

Other Algorithms

Resilient back-propagation (Rprop)

Rprop weight change:

$$\Delta w_{j,i}^{(t)} = \begin{cases} -\Delta_{j,i}^{(t)}, \text{if } c > 0 \\ +\Delta_{j,i}^{(t)}, \text{if } c < 0 \\ 0, otherwise \end{cases}$$

Modify update values:

$$\Delta_{j,i}^{(t)} = \begin{cases} \eta^{+} \cdot \Delta_{j,i}^{(t-1)}, \text{if } c > 0 \\ \eta^{-} \cdot \Delta_{j,i}^{(t-1)}, \text{if } c < 0 \\ \Delta_{j,i}^{(t-1)}, otherwise \end{cases}$$

Levenberg-Marquardt algorithm

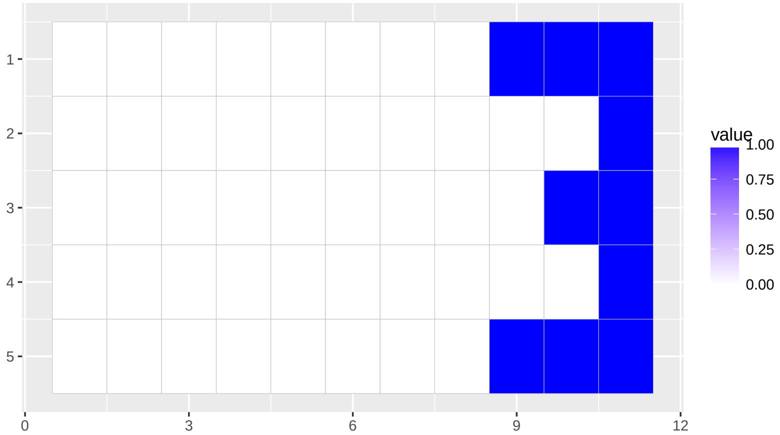
LMA is a very efficient training method for neural networks.

It combines the Gauss-Newton algorithm and the method of gradient descent.

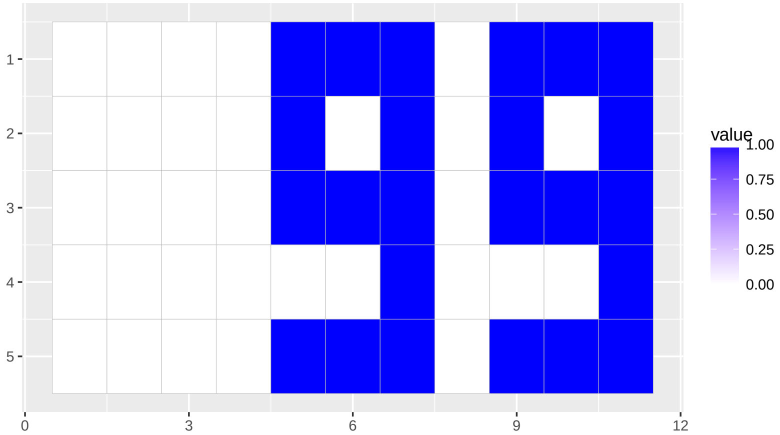
$$W_{min} = W_0 - (H + \lambda I)^{-1} g$$

Example Odd/Even

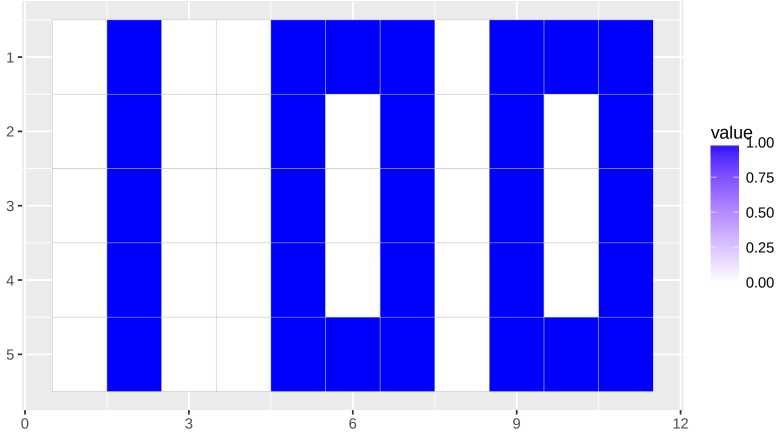
Pixel grid



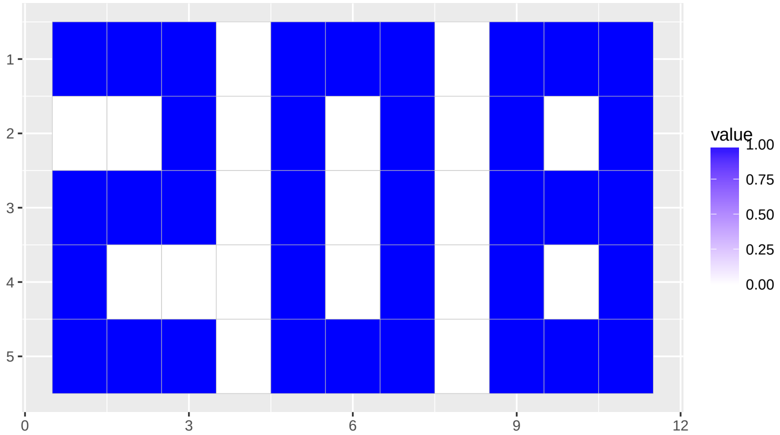
Pixel grid



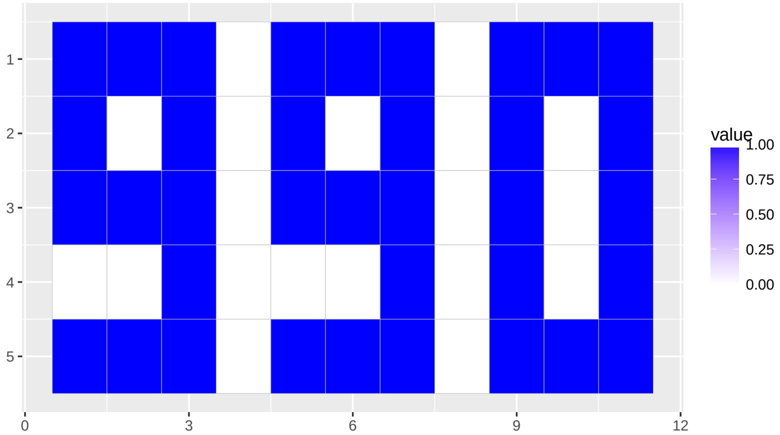
Pixel grid



Pixel grid



Pixel grid



The data

```
print(X[100,1:11])
print(X[100,12:22])
print(X[100,23:33])
print(X[100,34:44])
print(X[100,45:55])

##  [1] 0 1 0 0 1 1 1 0 1 1 1
##  [1] 0 1 0 0 1 0 1 0 1 0 1
##  [1] 0 1 0 0 1 0 1 0 1 0 1
##  [1] 0 1 0 0 1 0 1 0 1 0 1
##  [1] 0 1 0 0 1 1 1 0 1 1 1
```

Target vector for Odd/Even

```r
y <- rep(c(-1,1), 500)[-1000]
y[5]
y[56]
y[998]
y[999]

## [1] -1
## [1] 1
## [1] 1
## [1] -1
```

Perceptron function

```
perceptron(X = X, y = y, w = w, eta = 0.1,
           max_reps = 80, max_error_rate = 0.05)
```

- a single neuron
- uses signum function
- stochastic learning
- weight adjustment: $w \leftarrow w + \eta \cdot x \cdot y$

Perceptron function

```
perceptron(X = X, y = y, w = w, eta = 0.1,
             max_reps = 80, max_error_rate = 0.05)
```

- X: feature matrix
- y: target vector
- w: weights vector
- eta: learning rate
- max_reps: maximum weight update iterations
- max_error_rate: stop learning, if missclassification rate $\leq$ max_error_rate

Initialising the weights

```r
# number of features and number of obs
m = ncol(X)

# initial weights:
w = rnorm(m, 1)/10
w = rep(0.01, m)
```

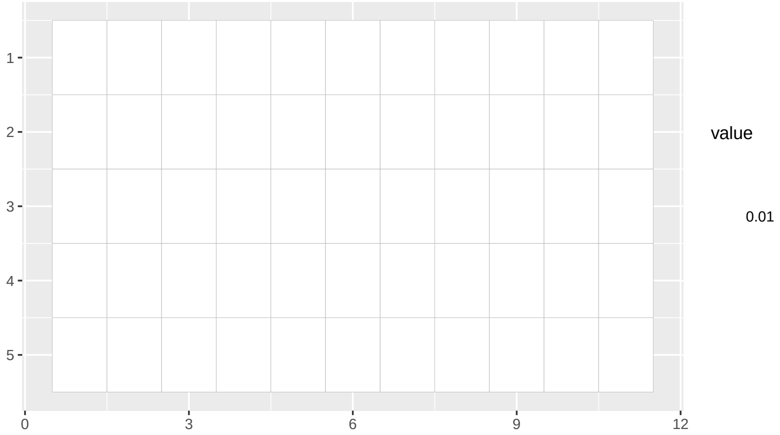
Fitting

```
set.seed(909)
fit = perceptron(X, y, w, eta = 0.1,
      max_reps =  80, max_error_rate = 0)

## [1] "At least 100% of all points correctly class
## [1] "Number of iterations:"
## [1] 70
```

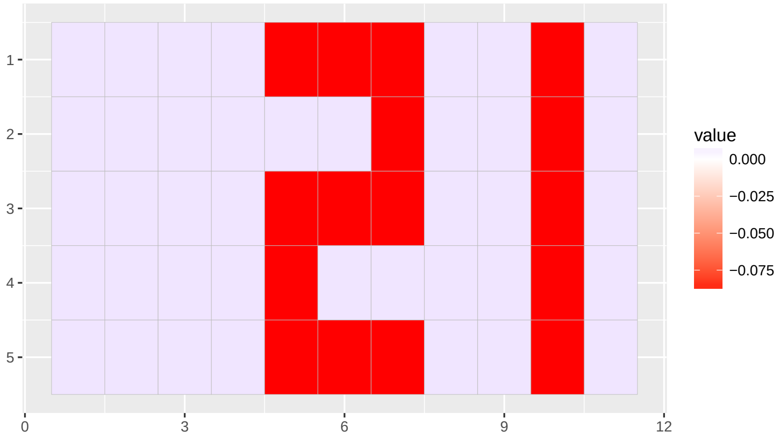
Weight changes over iterations

Initial weights:



Weight changes over iterations

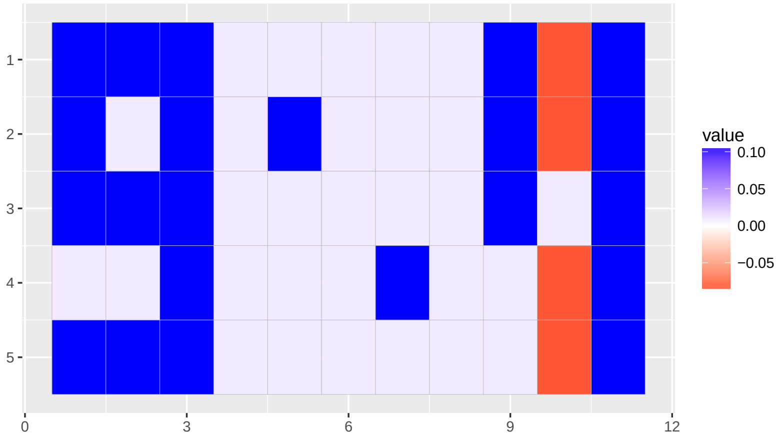
1 iteration:



Used number: 21

Weight changes over iterations

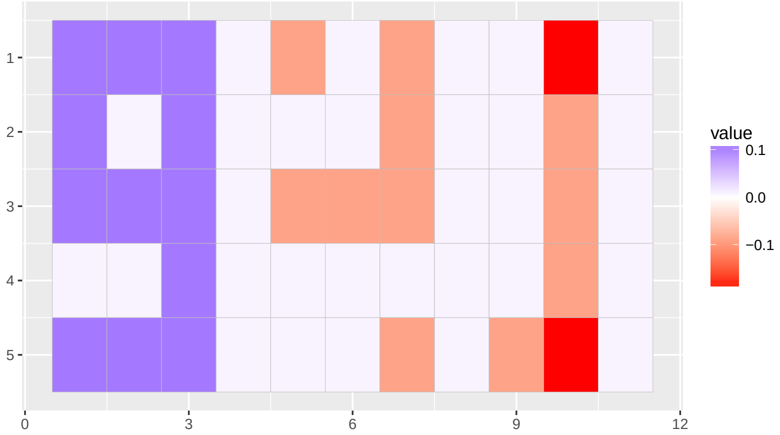
2 iterations:



Used numbers: 21, 984

Weight changes over iterations

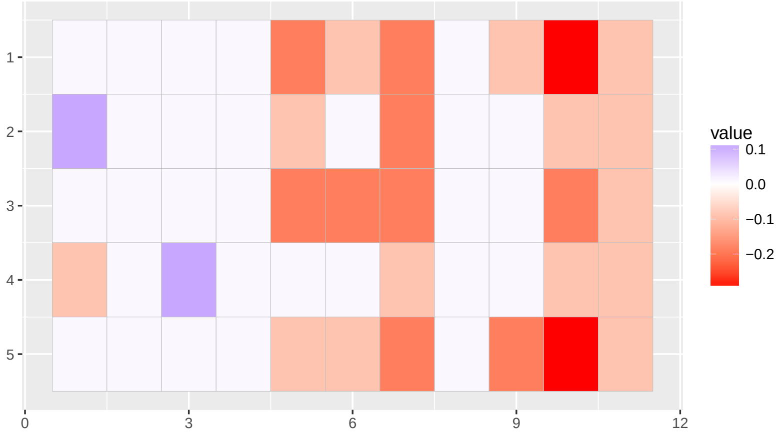
3 iterations:



Used numbers: 21, 984, 49

Weight changes over iterations

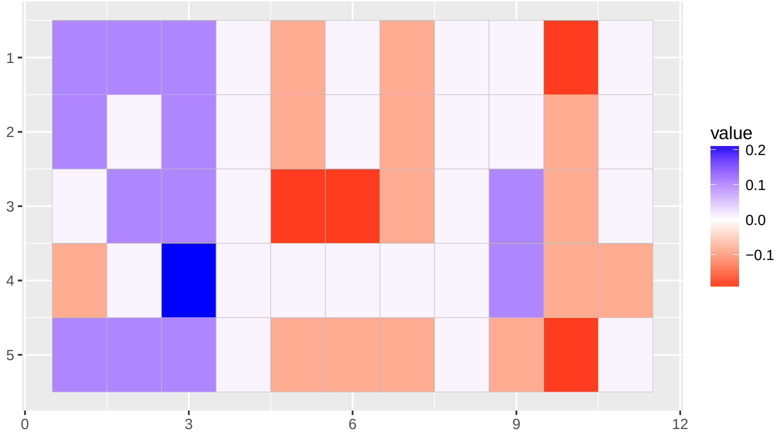
4 iterations:



Used numbers: 21, 984, 49, 293

Weight changes over iterations

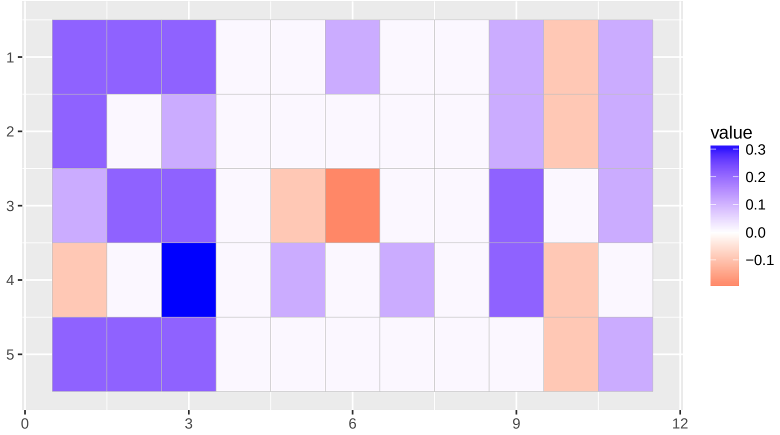
5 iterations:



Used numbers: 21, 984, 49, 293, 372

Weight changes over iterations

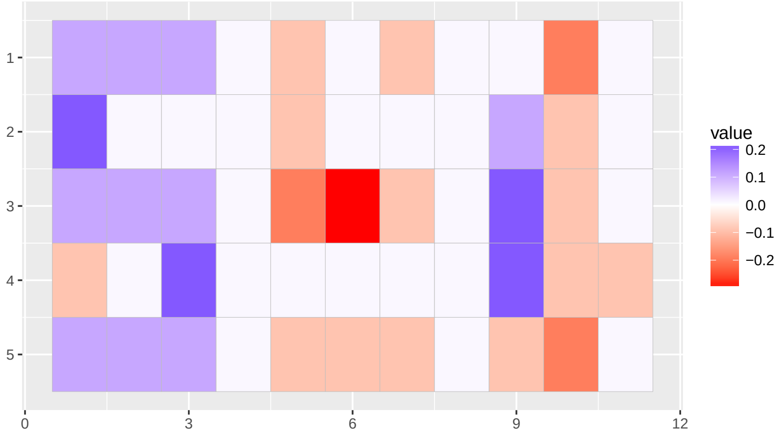
6 iterations:



Used numbers: 21, 984, 49, 293, 372, 508

Weight changes over iterations

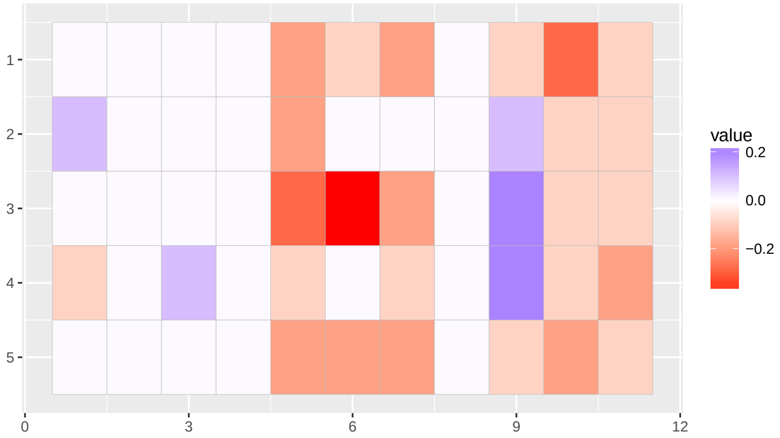
7 iterations:



Used numbers: 21, 984, 49, 293, 372, 508, 363

Weight changes over iterations

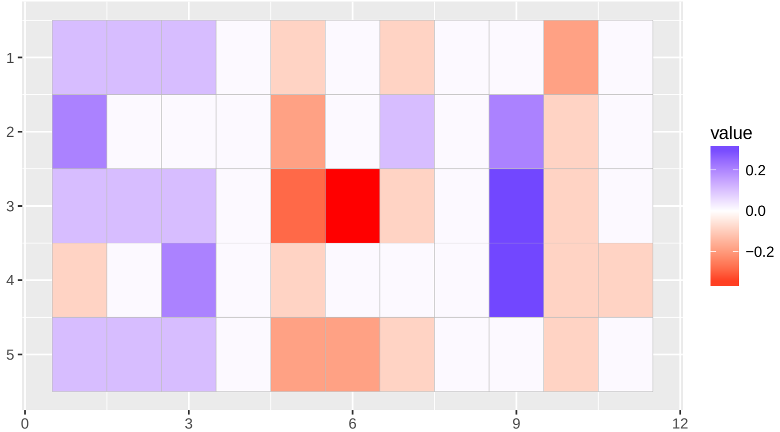
8 iterations:



Used numbers: 21, 984, 49, 293, 372, 508, 363, 567

Weight changes over iterations

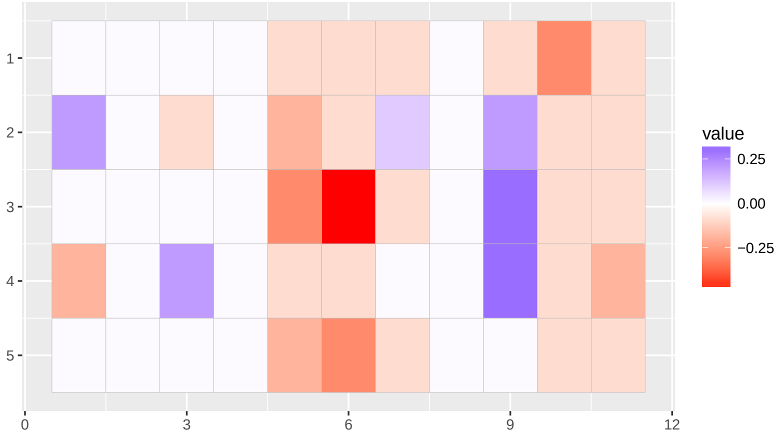
9 iterations:



Used numbers: 21, 984, 49, 293, 372, 508, 363, 567, 570

Weight changes over iterations

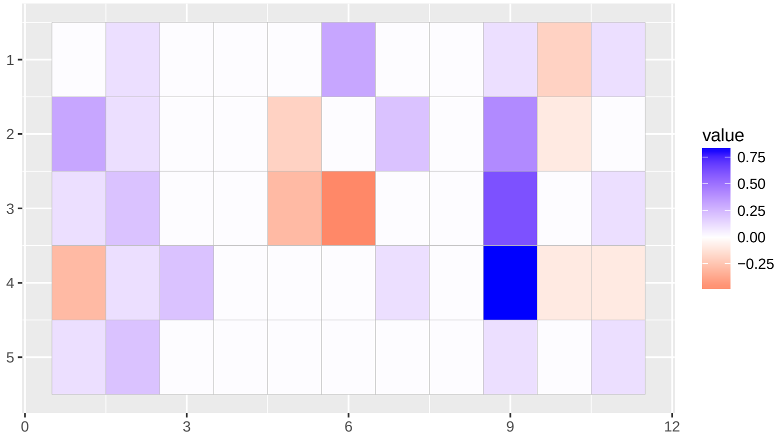
10 iterations:



Used numbers: 21, 984, 49, 293, 372, 508, 363, 567, 570, 217

Weight changes over iterations

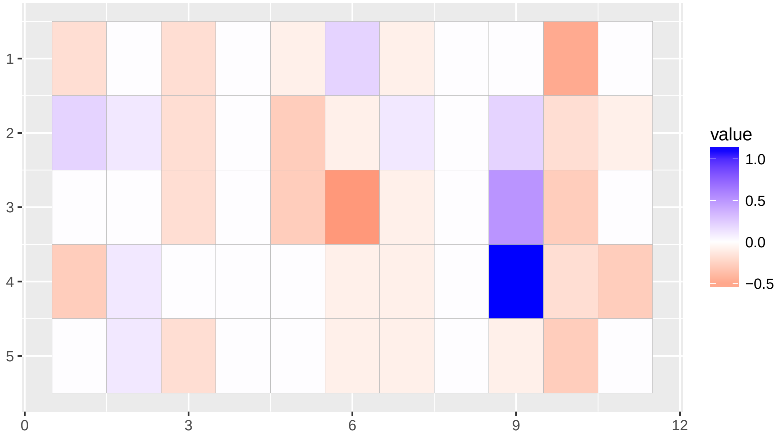
20 iterations:



Used numbers: 21, 984, 49, 293, 372, 508, 363, 567, 570, 217, 516, 745, 938, 349, 100, 533, 972, 649, 96, 344

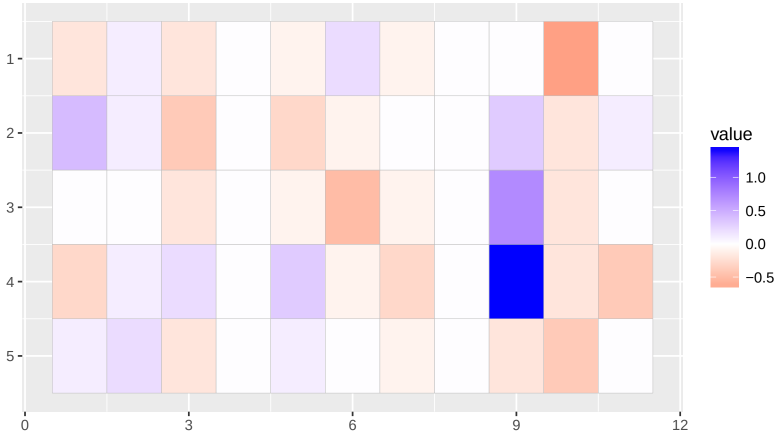
Weight changes over iterations

30 iterations:



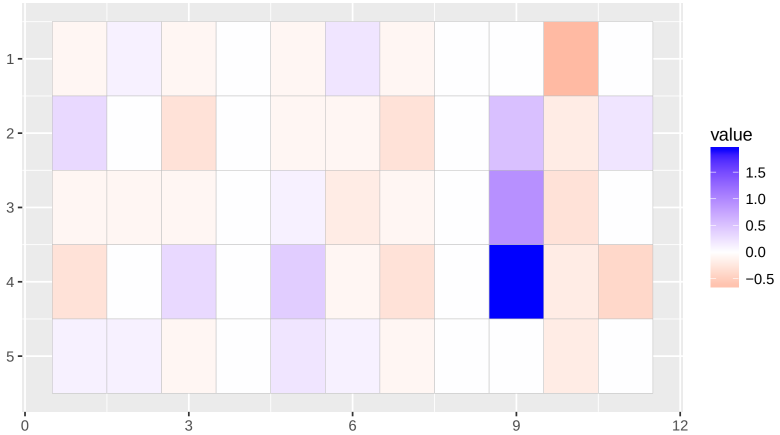
Weight changes over iterations

40 iterations:



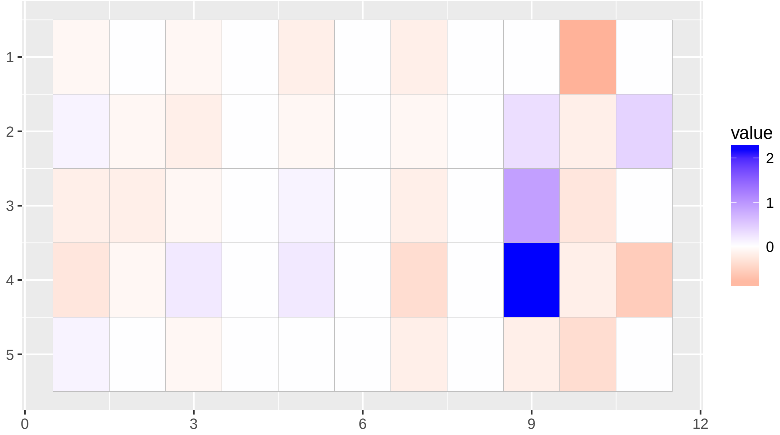
Weight changes over iterations

50 iterations:



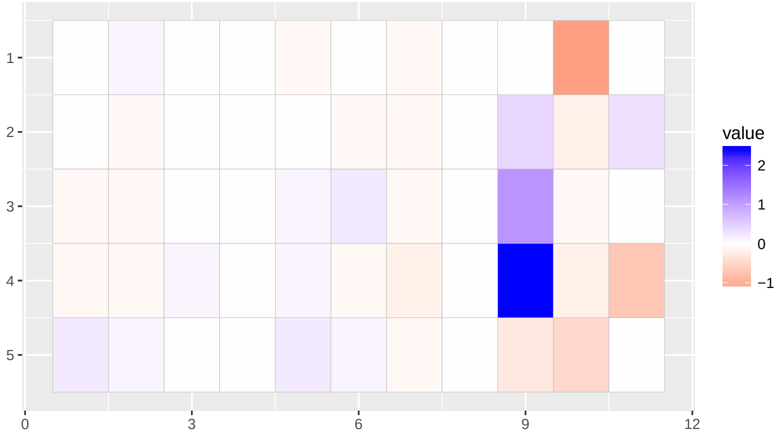
Weight changes over iterations

60 iterations:



Weight changes over iterations

Final weights after 70 iterations:



New targets

- “multiple of 5”
- “multiple of 4”
- “multiple of 3”

New target: multiple of 5

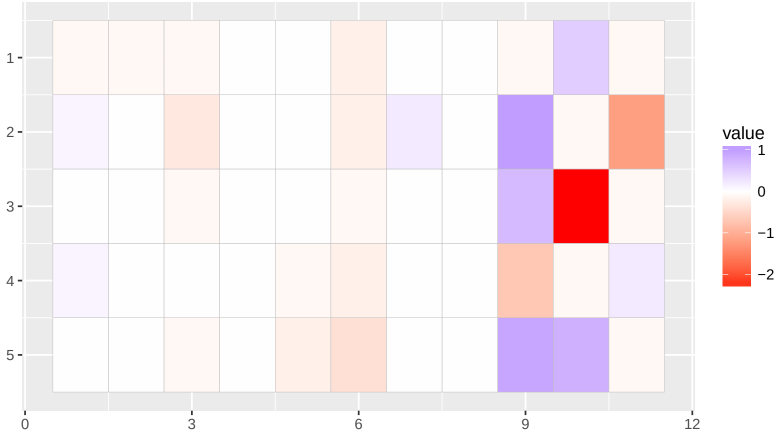
```
y = rep(-1, 999)
y[(1:999) %% 5 == 0] = 1

set.seed(909)
fit = perceptron(X, y, w, eta = 0.1,
      max_reps =  300, max_error_rate = 0.05)

## [1] "At least 95% of all points correctly classi
## [1] "Number of iterations:"
## [1] 112
```

Final weights

Final weights for "multiple of 5":



References

TODO