A basic network (without hidden layer)

\[ z = w_1 y_1 + w_2 y_2 + b \]
A basic network (without hidden layer)
Processing batches: Many samples in parallel

Avoid loops! (slow)
Processing batches: Many samples in parallel

one sample:
  vector \( (N_{\text{in}}) \) \( y \)

many samples:
  matrix \( (N_{\text{samples}} \times N_{\text{in}}) \) \( y \)

Apply matrix/vector operations to operate on all samples simultaneously!

Avoid loops! (slow)

Note: Python interprets \( M = A + b \) as:

\[
M_{ij} = A_{ij} + b_j
\]

First index of \( b \) is ‘expanded’ to size indicated by \( A \)
one sample:

\[ z = \text{dot}(w, y) + b \]

vector \( (N_{\text{in}}) \)

matrix \( (N_{\text{out}} \times N_{\text{in}}) \)

vector \( (N_{\text{out}}) \)

many samples:

\[ z = \text{dot}(y, w) + b \]

matrix \( (N_{\text{samples}} \times N_{\text{in}}) \)

matrix \( (N_{\text{samples}} \times N_{\text{out}}) \)

matrix \( (N_{\text{in}} \times N_{\text{out}}) \)

vector \( (N_{\text{out}}) \)

becomes

\( N_{\text{samples}} \times N_{\text{out}} \)
We can create complicated functions...

...but can we create arbitrary functions?
Approximating an arbitrary nonlinear function

$F(y)$
Approximating an arbitrary nonlinear function
\[ y_{\text{out}} = \delta F_1 f(w \cdot (y - Y_1)) + \delta F_2 f(w \cdot (y - Y_2)) \]

\[ (f = \text{sigmoid} = \text{smooth step}) \]
\[ y_{out} = \delta F_1 f(w \cdot (y - Y_1)) + \delta F_2 f(w \cdot (y - Y_2)) \]

(f = sigmoid = smooth step)

use biases:
\[ b_1 = -wY_1 \]
\[ b_2 = -wY_2 \]
Approximating an arbitrary 2D nonlin. function
Approximating an arbitrary 2D nonlin. function

First step: create quarter-space “step function”
Trick: “AND” operation in a neural network

\[ y_{\text{out}} = f(w \cdot (y_1 + y_2 - 1.5)) \]

for large \( w \):

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Trick: “AND” operation in a neural network

for large $w$:

<table>
<thead>
<tr>
<th></th>
<th>$y_2$</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Figure out how to implement the following operations using a neural network:

**OR**

**XOR** (gives 1 only if inputs are different, i.e. for 10 and 01)
Approximating an arbitrary 2D nonlin. function

\[ y_{\text{out}} = f(w \cdot (y_1 - \bar{y}_1)) \land f(w \cdot (y_2 - \bar{y}_2)) \]

- Step in \( y_1 \): \( f(w \cdot (y_1 - \bar{y}_1)) \)
- Step in \( y_2 \): \( f(w \cdot (y_2 - \bar{y}_2)) \)

Diagram:
- \( y_{\text{out}} \)
- \( y_1 \)
- \( y_2 \)
- \( \bar{y}_1 \)
- \( \bar{y}_2 \)
- Axes: \( y_1 \) and \( y_2 \)
- Colors:
  - Blue: \( 0 \)
  - Dark blue: \( 1 \)

Graphical representation of the AND function for \( y_1 \) and \( y_2 \).
(superimposing two such quarter-space functions)
Approximating an arbitrary 2D nonlin. function

\[ \delta F_1, \delta F_2, \delta F_3 \]

\( y_1, y_2 \)
Approximating an arbitrary 2D nonlin. function
Approximating an arbitrary 2D nonlin. function

\[ \delta F_1, \quad \delta F_2, \quad \delta F_3 \]

\[ \text{AND} \quad \text{AND} \quad \text{AND} \]

\[ y_1, \quad y_2 \]

\[ y_2 \]

\[ y_1 \]
Approximating an arbitrary 2D nonlin. function

$\delta F_1$, $\delta F_2$, $\delta F_3$, AND

$y_1$, $y_2$
Approximating an arbitrary 2D nonlin. function

\[ \delta F_1 \quad \delta F_2 \quad \delta F_3 \]

\[ \text{AND} \quad \text{AND} \quad \text{AND} \]

\[ y_1 \quad y_2 \]
Approximating an arbitrary 2D nonlin. function

\[ F_1 \]
\[ F_2 \]
\[ F_3 \]
Any arbitrary (smooth) function (with vector input and vector output) can be approximated as well as desired by a neural network with a single (!) hidden layer.

(as long as we allow for sufficiently many neurons)
Figure out how to implement a 2D function that produces a (smoothened) square

Bonus version: how to get an arbitrary convex shape (approximately)?

Implement them on the computer and play around...
Extra * bonus version:

We have indicated how to approximate arbitrary functions in 2D using 2 hidden layers (with our AND construction, and summing up in the end)

Can you do it with a **single** hidden layer?
A neural network

\[ y_{\text{out}} = F_w(y_{\text{in}}) \]

Note: When we write “\( w \)” as subscript of \( F \), we mean all the weights and also biases

Note: When we write “\( y_{\text{out}} \)”, we mean the whole vector of output values
How to choose the weights (and biases)?

By “training” with thousands of examples!
This is essentially nonlinear curve fitting!

Example for one output neuron and one input neuron

\[ y^{\text{out}} = F_w(y^{\text{in}}) \]

curve depends on parameters \( w \)

adjust \( w \)!

training examples = known data points
Challenge:

Curve fitting with a million parameters!

maybe 1000s of input neurons (dimension of $y^{\text{in}}$)
many 1000s of hidden layer neurons

millions of weights

need at least tens of thousands (or more) examples
change this weight!
Goal: Adapt weights to get closer to the “correct” answer (provided by the trainer)
A neural network

Complicated nonlinear function that depends on all the weights and biases

\[ y^{\text{out}} = F_w(y^{\text{in}}) \]

Note: When we write “\(w\)” as subscript of \(F\), we mean all the weights and also biases
Note: When we write “\(y^{\text{out}}\)”, we mean the whole vector of output values
We have:

\[ y^{\text{out}} = F_w(y^{\text{in}}) \]

neural network (\(w\) here also stands for the biases)

We would like:

\[ y^{\text{out}} \approx F(y^{\text{in}}) \]

desired “target” function

**Cost function** measures deviation:

\[
C(w) = \frac{1}{2} \langle \| F_w(y^{\text{in}}) - F(y^{\text{in}}) \|^2 \rangle
\]

vector norm

average over all training samples
Approximate version, for $N$ samples:

$$C(w) \approx \frac{1}{2} \frac{1}{N} \sum_{s=1}^{N} \left\| F_w(y^{(s)}) - F(y^{(s)}) \right\|^2$$

$s$ = index of sample

Minimizing $C$ for this case: “least-squares fitting”!
Method: “Sliding down the hill”
(“gradient descent”)
\[ \dot{w} \sim -\nabla_w C(w) \]

physicist would say:
motion of an overdamped particle (velocity set by force)
Problem: Evaluating $C$ would mean averaging over ALL training samples

Solution: Only average over a few samples, get approximate $C$

Discrete steps: for each step evaluate a few samples and update weights according to:

$$w_j \rightarrow w_j - \eta \frac{\partial \tilde{C}(w)}{\partial w_j}$$

(approximate version of $C$)
(take different samples in each step!)

(Note: just as before, the biases $b$ are included here, think of them as extra parameters $w$)
For sufficiently small steps: sum over many steps approximates true gradient (because it is an additional average)