Linear Recurrent Neural Networks

For time series prediction

Oliver Obst

“Prediction”

“Prediction”

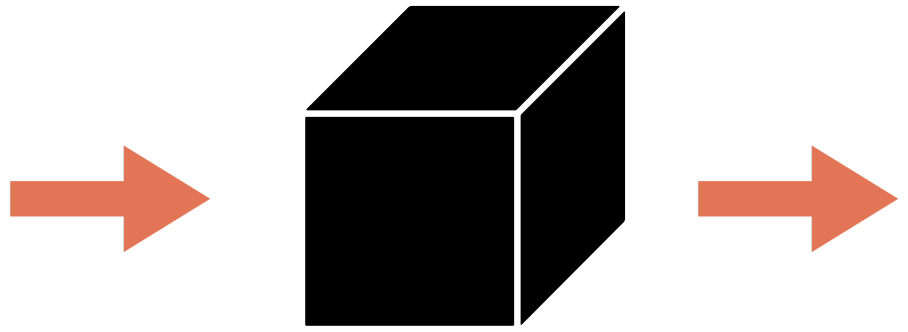
Dog

Cat

Bird

“Prediction”


```
[152,  50,  60, 223,   0, 117,
[ 16, 187,   6,  38, 190, 220,
[ 82,  36, 233, 129, 137,  58,
[237,  33, 207,   4,  76, 205,
[251, 149, 160, 177, 113, 240,
[ 79,  85, 119, 170, 159, 253,
[218, 249, 254, 196,  11,   1,
[198,  22, 211, 120, 238, 175,
[111, 246, 222,  36, 126, 236,
[195, 168,  82, 158,  17,  51,
[177, 133, 232, 113, 227, 141,
[ 84,  42,  13, 100,  17,  69,
[246, 158,  29,  92, 242,  34,
[  6, 136, 137,  55,  18,  57,
[182, 229,  49, 182, 103, 253,
[ 81,  10,  17, 209, 223, 111,
[246,  41, 242,   1, 117,  76,
[204, 205,  56,  13,   2, 222,
[209, 162, 239, 205,  62,  13,
[ 43, 219,  96, 135, 102,  14,
[ 58,  80, 236, 238, 122, 103,
[ 24, 124,  61,  94,  80,  18,
[  8, 223, 251,  22, 252,  35,
[ 18, 224, 213,  12,  76,  92,
[199, 188, 216, 156, 230, 177,
[139, 201, 188, 142,   5,  46,
[ 43, 198,  11,  58,  59, 222,
[141,  60,  51, 181, 125,  23,
[ 75, 238,  27,  90,  28, 118,
```

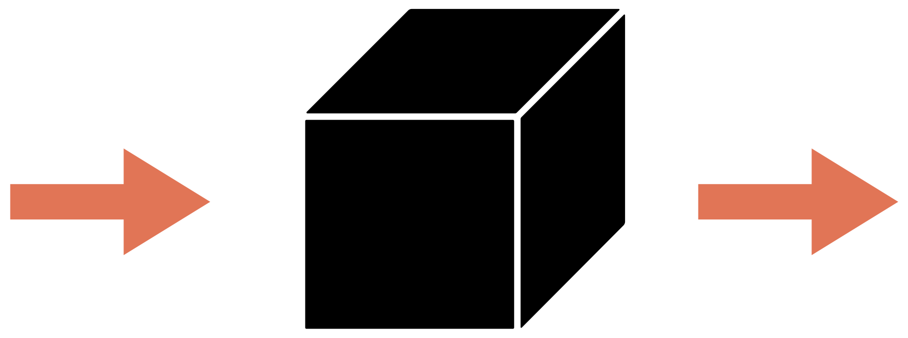
Dog

Cat

Bird

“Prediction”

```
[  4, 161, 191,  29, 202, 220,
[220,  83,  96, 130, 239,  37,
[ 42, 197,  58, 178,  69, 229,
[ 77, 178,  30, 174, 236,  97,
[126,  32,  19, 234,  59, 197,
[214, 166,  15, 105, 160,  95,
[ 25, 199,  48, 245, 169, 173,
[129,   1,  78,  68, 211,  61,
[ 69,  82,  21,  97, 223, 168,
[ 63, 184, 218, 117, 163, 218,
[248, 184,  44, 121,  80, 217,
[235, 121, 165,  80,   8,  21,
[244, 138, 163, 109, 116, 144,
[159, 250, 108,  61,   1, 255,
[ 76,  32,  48, 148,  87,  11,
[248, 129,  99, 170,  28, 232,
[216, 205, 208, 247,  11, 148,
[ 32, 209,  26, 211, 205,  19,
[ 85, 190, 243,   7,  68, 134,
[225, 115,  23, 255,  30, 227,
[ 33,  43, 232,  62, 201, 214,
[175, 168,  61, 162, 141, 112,
[234,  70,  90,  53, 123, 201,
[ 91,  26, 237,  33, 173,  14,
[  1, 229,  35, 147,  83,  91,
[192,  92, 140, 210,  80, 234,
[158, 114,  58, 241, 101,   8,
[ 54, 148,  33, 232, 240,  60,
[110, 223,  91,  25, 204, 173,
[  1, 114,  81, 238, 204, 174,
```



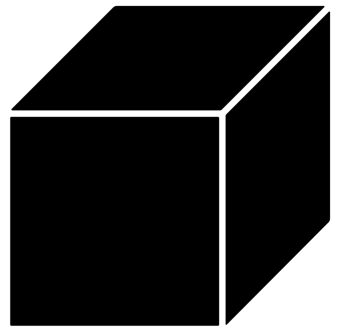
Dog

Cat

Bird

“Prediction”

```
[  4, 161, 191,  29, 202, 220,
[220,  83,  96, 130, 239,  37,
[ 42, 197,  58, 178,  69, 229,
[ 77, 178,  30, 174, 236,  97,
[126,  32,  19, 234,  59, 197,
[214, 166,  15, 105, 160,  95,
[ 25, 199,  48, 245, 169, 173,
[129,   1,  78,  68, 211,  61,
[ 69,  82,  21,  97, 223, 168,
[ 63, 184, 218, 117, 163, 218,
[248, 184,  44, 121,  80, 217,
[235, 121, 165,  80,   8,  21,
[244, 138, 163, 109, 116, 144,
[159, 250, 108,  61,   1, 255,
[ 76,  32,  48, 148,  87,  11,
[248, 129,  99, 170,  28, 232,
[216, 205, 208, 247,  11, 148,
[ 32, 209,  26, 211, 205,  19,
[ 85, 190, 243,   7,  68, 134,
[225, 115,  23, 255,  30, 227,
[ 33,  43, 232,  62, 201, 214,
[175, 168,  61, 162, 141, 112,
[234,  70,  90,  53, 123, 201,
[ 91,  26, 237,  33, 173,  14,
[  1, 229,  35, 147,  83,  91,
[192,  92, 140, 210,  80, 234,
[158, 114,  58, 241, 101,   8,
[ 54, 148,  33, 232, 240,  60,
[110, 223,  91,  25, 204, 173,
[  1, 114,  81, 238, 204, 174
```



Dog

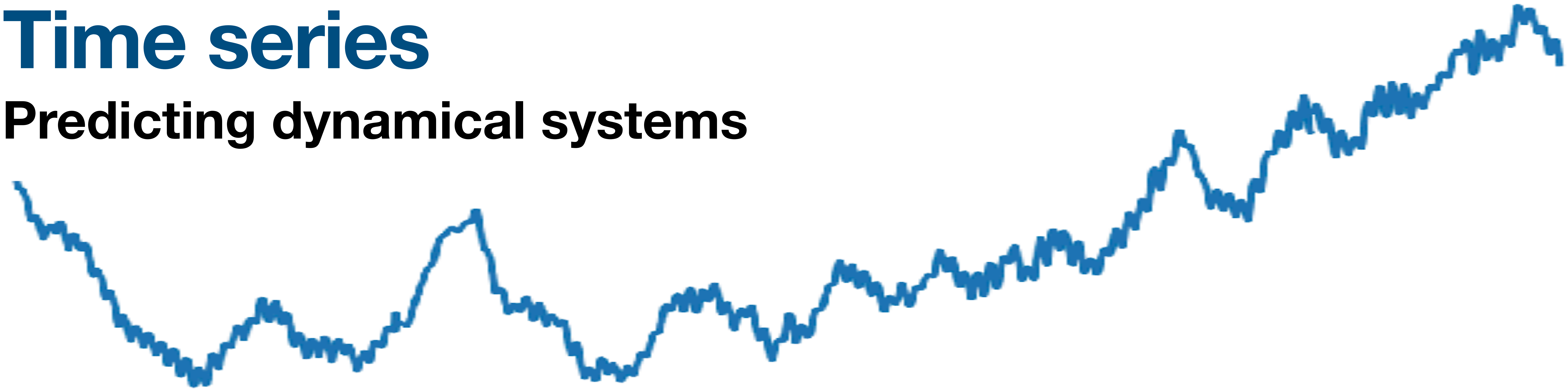
Cat

Bird

$$f(\mathbf{x}) = \hat{y}$$

Time series

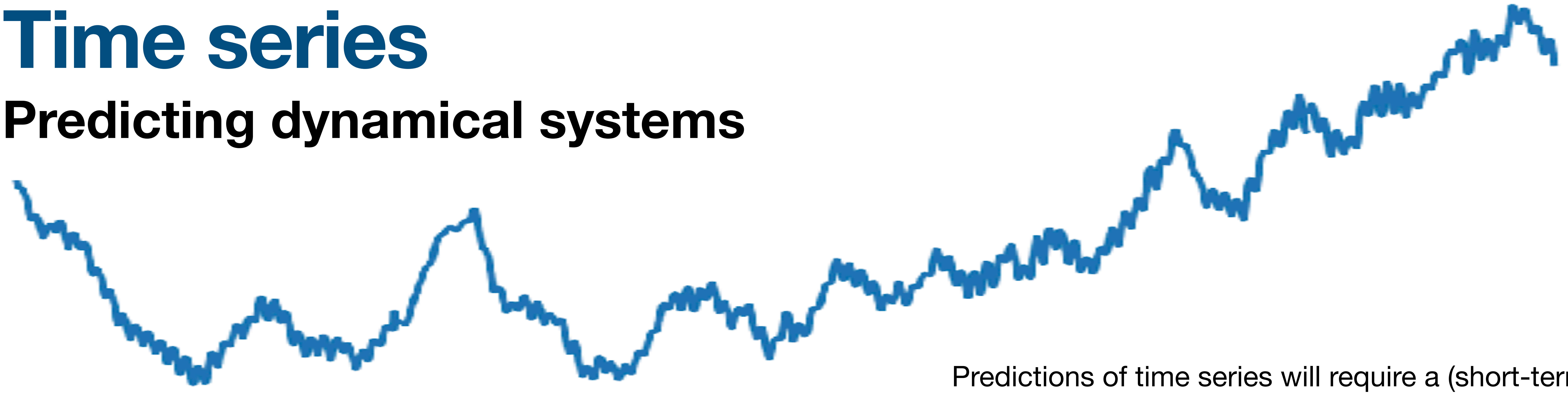
Predicting dynamical systems



$$f(\mathbf{x}) = \hat{y}$$

Time series

Predicting dynamical systems



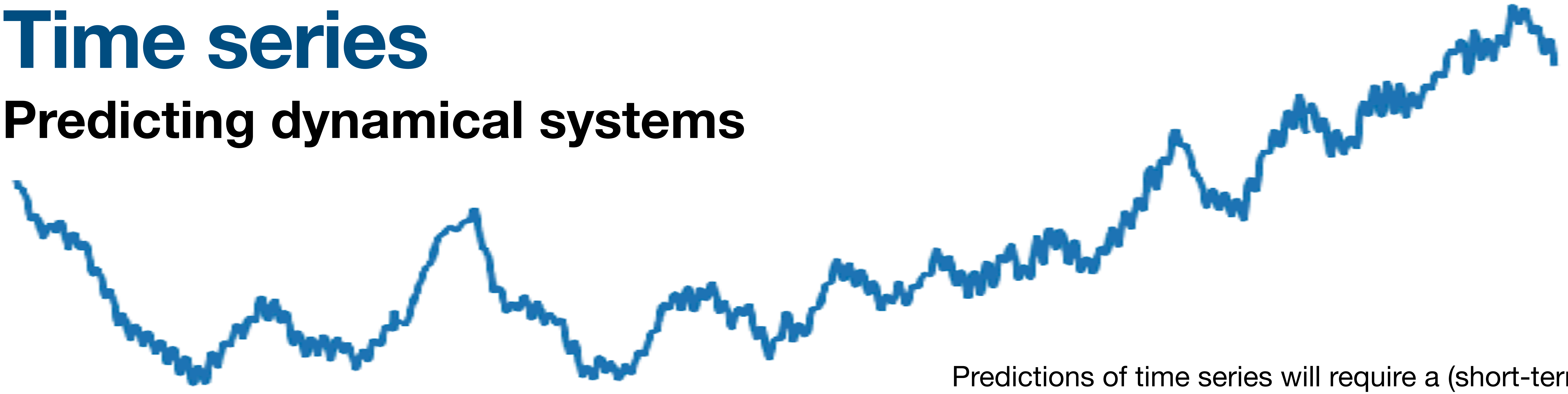
$$\cancel{f(\mathbf{x}) = \hat{y}}$$

Predictions of time series will require a (short-term) memory

This is not required or available from methods predicting functions, e.g., for classification of images.

Time series

Predicting dynamical systems



Predictions of time series will require a (short-term) memory

This is not required or available from methods predicting functions, e.g., for classification of images.

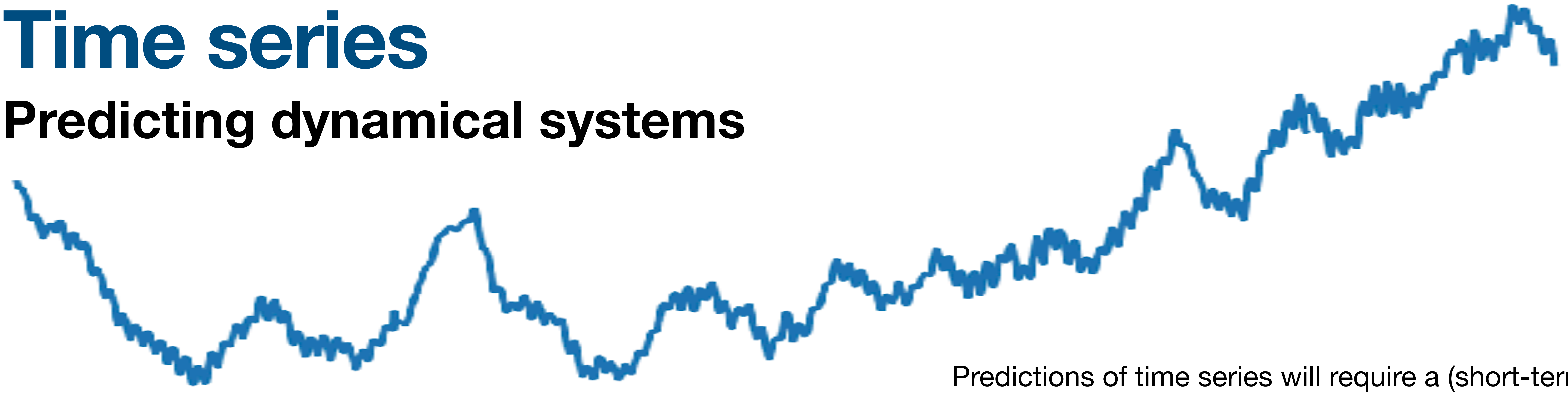
$$\cancel{f(\mathbf{x}) = \hat{y}}$$

$$x(t+1) = F(t, x(t))$$

Autonomous dynamical system

Time series

Predicting dynamical systems



Predictions of time series will require a (short-term) memory

This is not required or available from methods predicting functions, e.g., for classification of images.

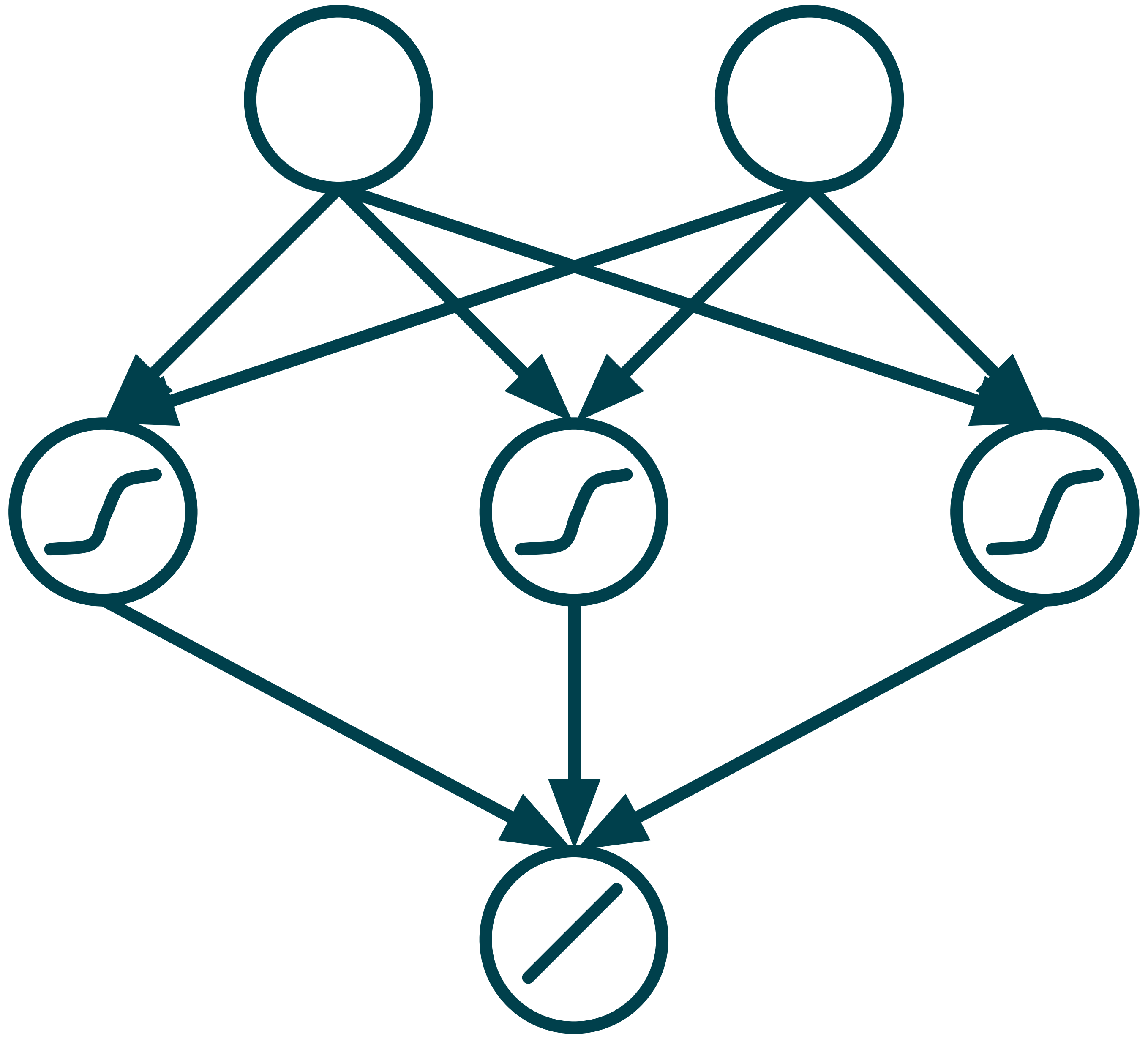
$$\cancel{f(\mathbf{x}) = \hat{y}}$$

$$x(t+1) = F(t, x(t))$$

Autonomous dynamical system

$$x(t+1) = F(t, x(t), u(t))$$

Non-autonomous dynamical system

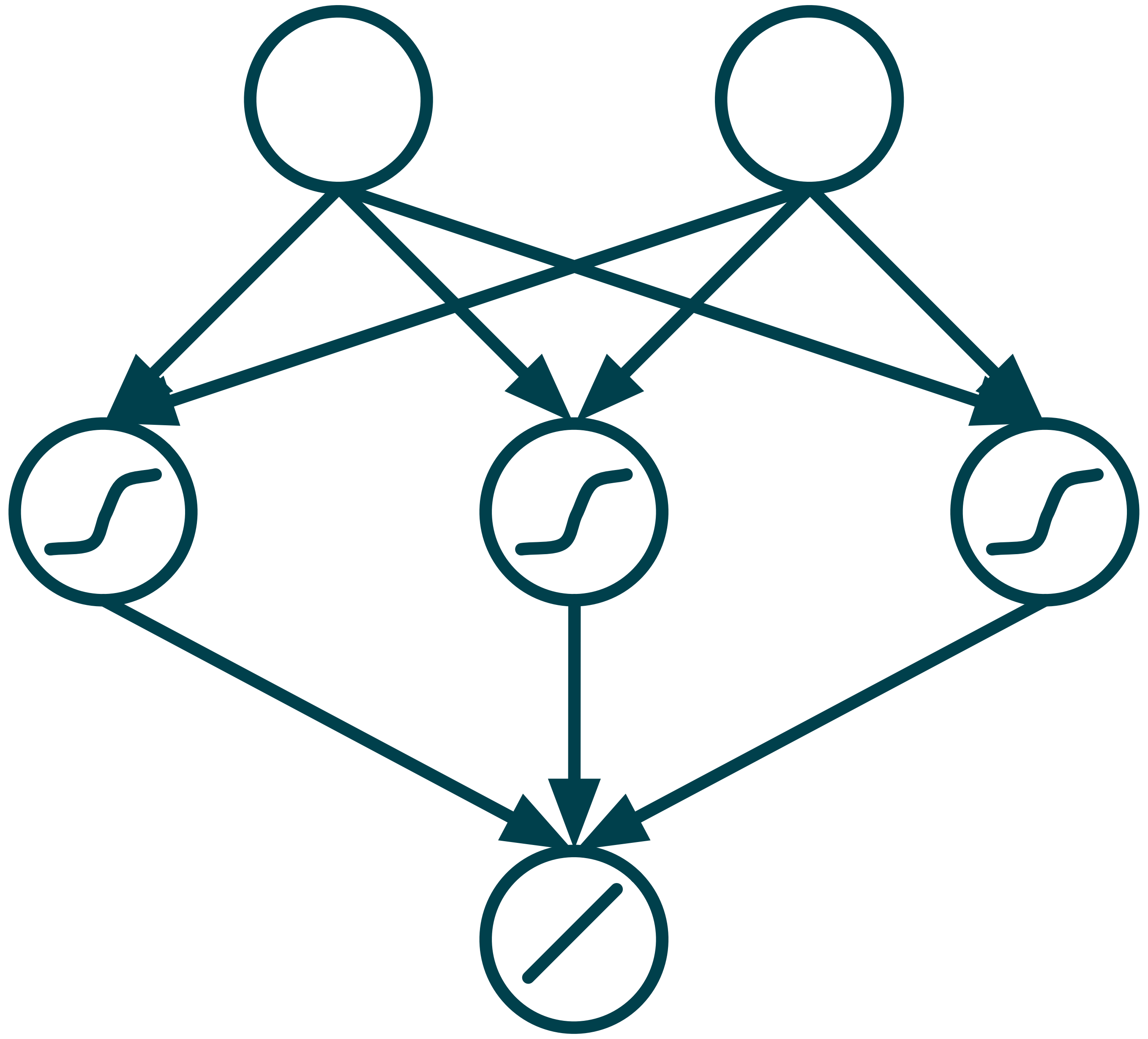


Feedforward neural network

- can represent continuous functions with arbitrary accuracy ("universal approximation theorem")

- "Training": optimisation

- **no (persistent) internal state:** output is a function of the input



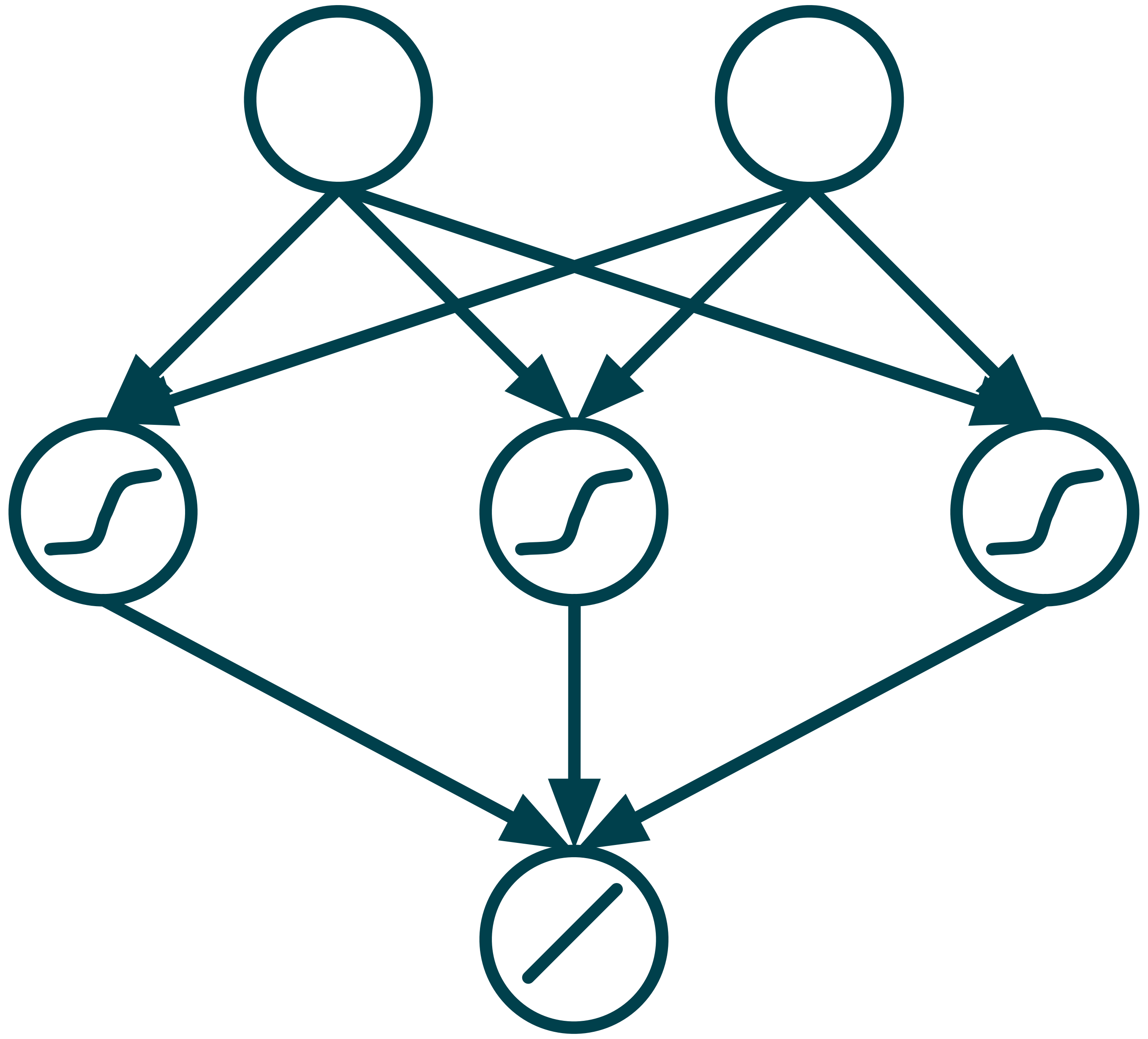
u

Feedforward neural network

- can represent continuous functions with arbitrary accuracy ("universal approximation theorem")

- "Training": optimisation

- **no (persistent) internal state:** output is a function of the input

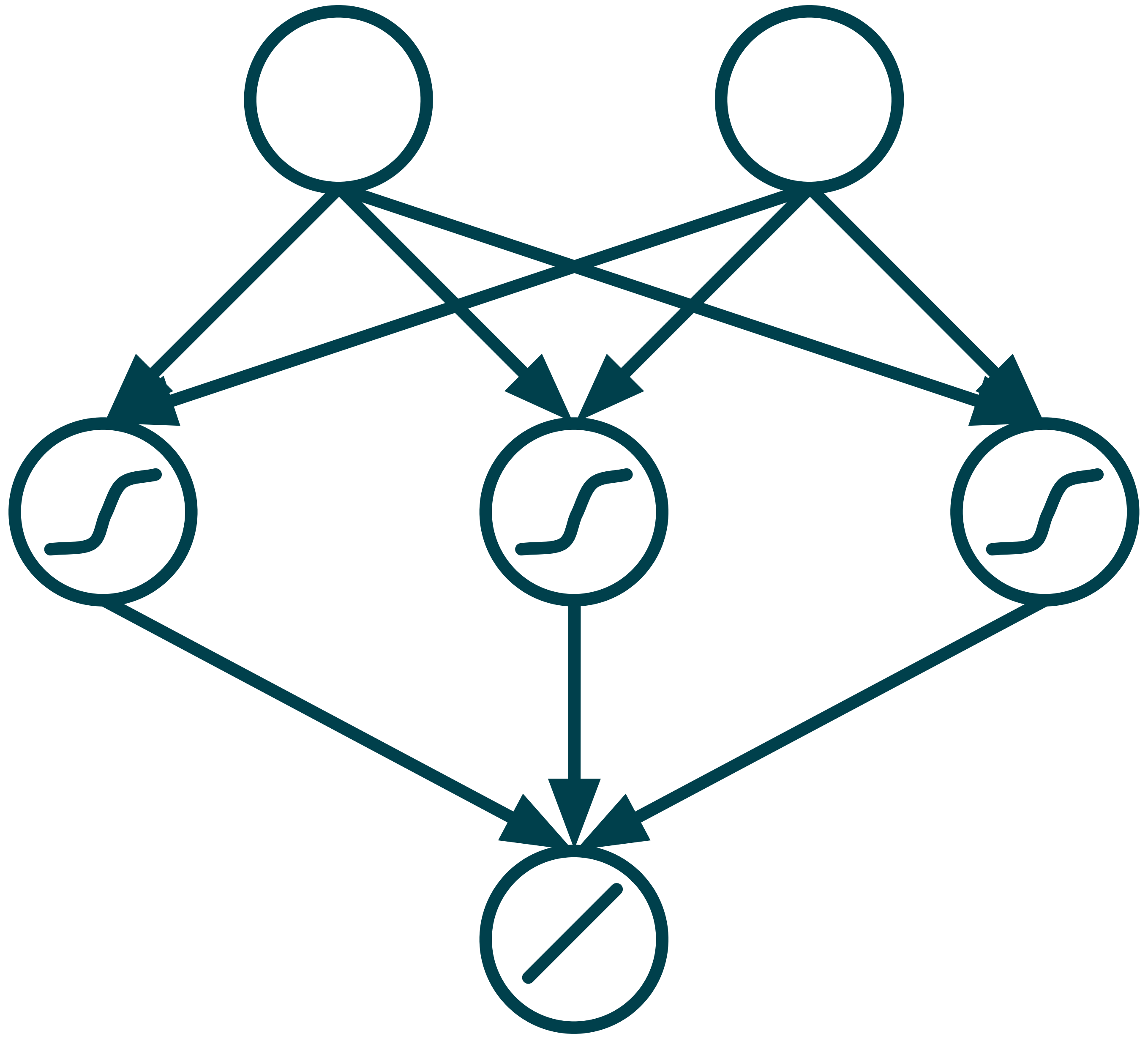


$\mathbf{u}$

$$\mathbf{x} = \tanh(\mathbf{w}^{\text{in}}\,\mathbf{u})$$

Feedforward neural network

- can represent continuous functions with arbitrary accuracy ("universal approximation theorem")
- "Training": optimisation
- **no (persistent) internal state:** output is a function of the input



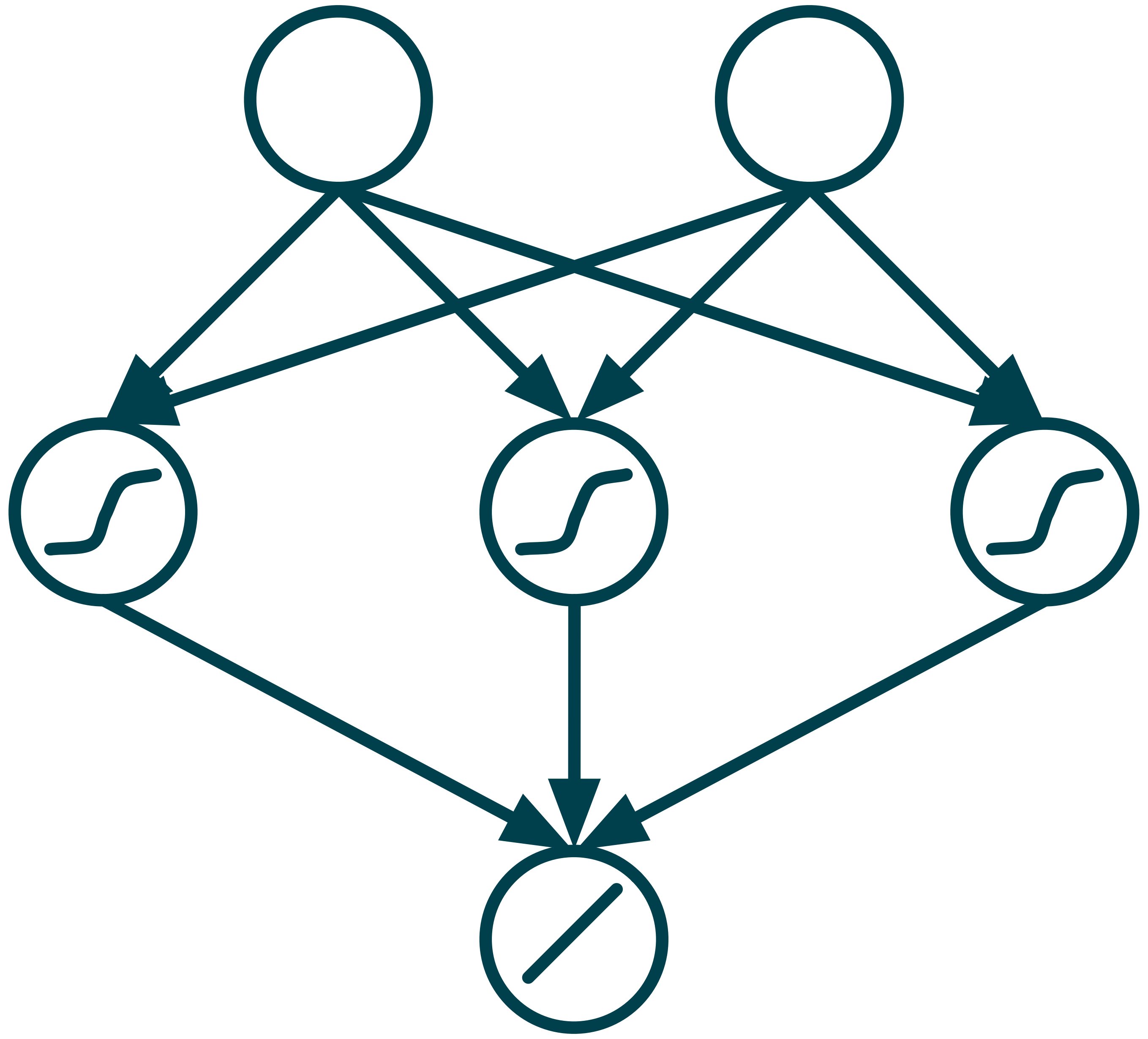
$\mathbf{u}$

$$\mathbf{x} = \tanh(\mathbf{w}^{\text{in}}\,\mathbf{u})$$

$$\hat{y} = \mathbf{w}^{\text{out}}\,\mathbf{x}$$

Feedforward neural network

- can represent continuous functions with arbitrary accuracy ("universal approximation theorem")
- "Training": optimisation
- **no (persistent) internal state:** output is a function of the input



$$\hat{y} = f_\theta(\mathbf{u})$$

$$\theta = \{\mathbf{w}^{\text{in}}, \mathbf{w}^{\text{out}}\}$$

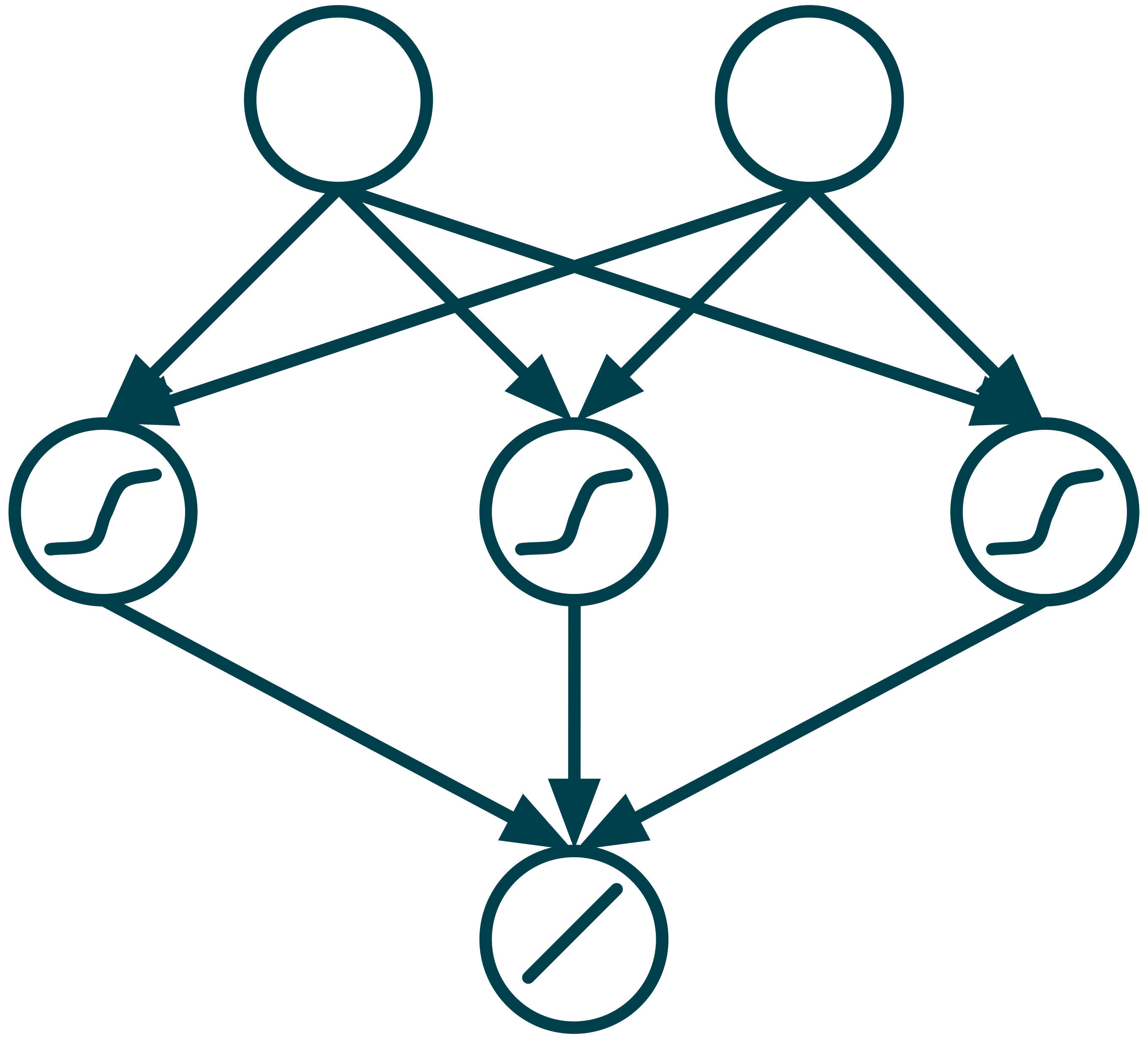
$\mathbf{u}$

$$\mathbf{x} = \tanh(\mathbf{w}^{\text{in}}\,\mathbf{u})$$

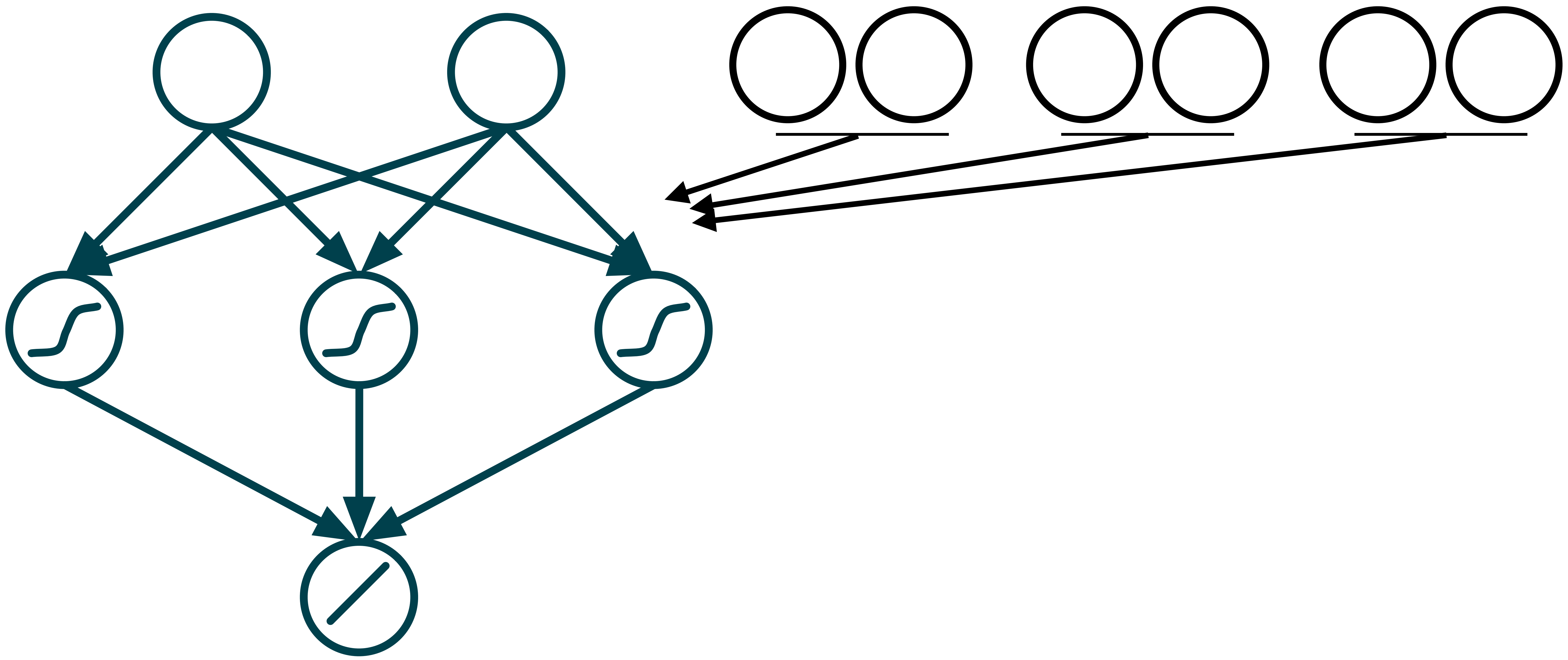
$$\hat{y} = \mathbf{w}^{\text{out}}\,\mathbf{x}$$

Feedforward neural network

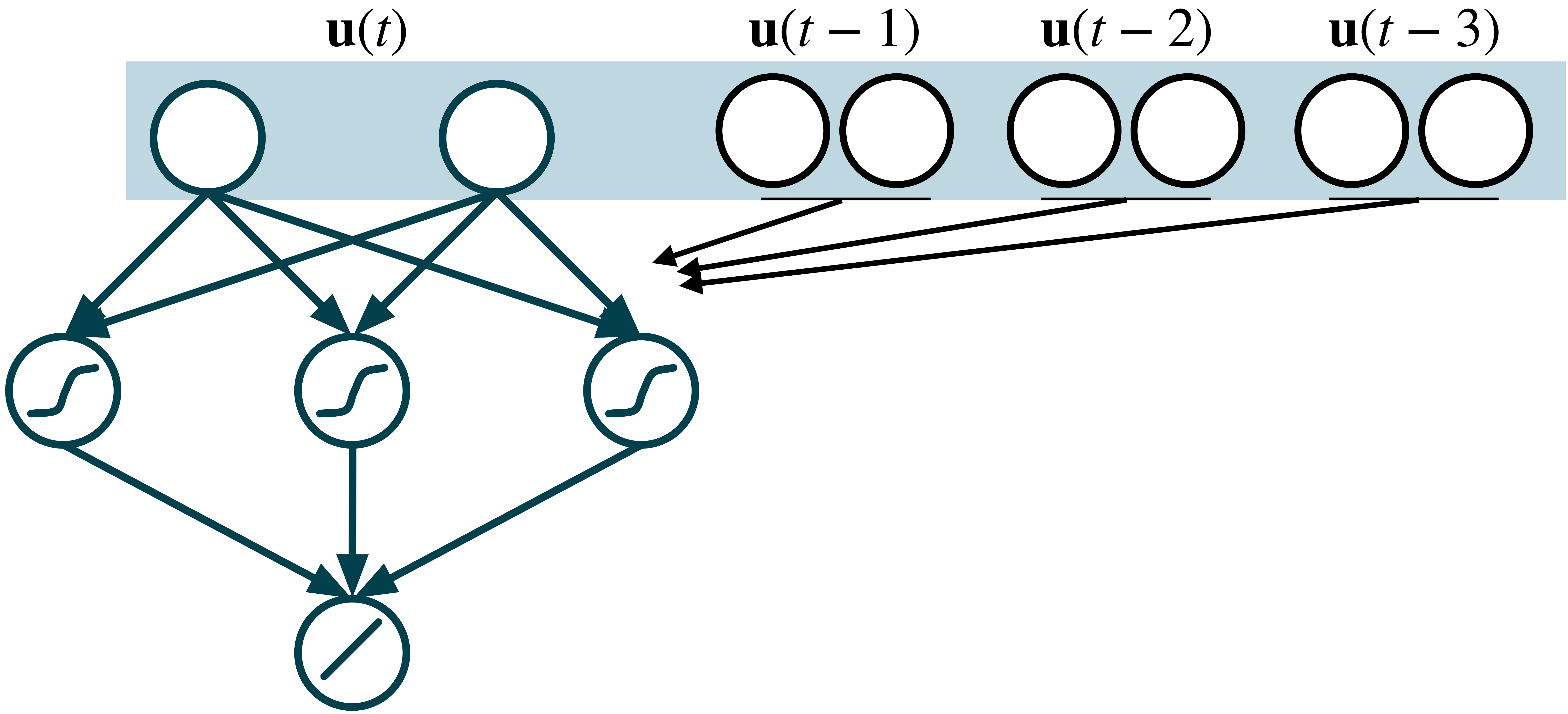
- can represent continuous functions with arbitrary accuracy (“universal approximation theorem”)
- “Training”: optimisation
- **no (persistent) internal state:** output is a function of the input



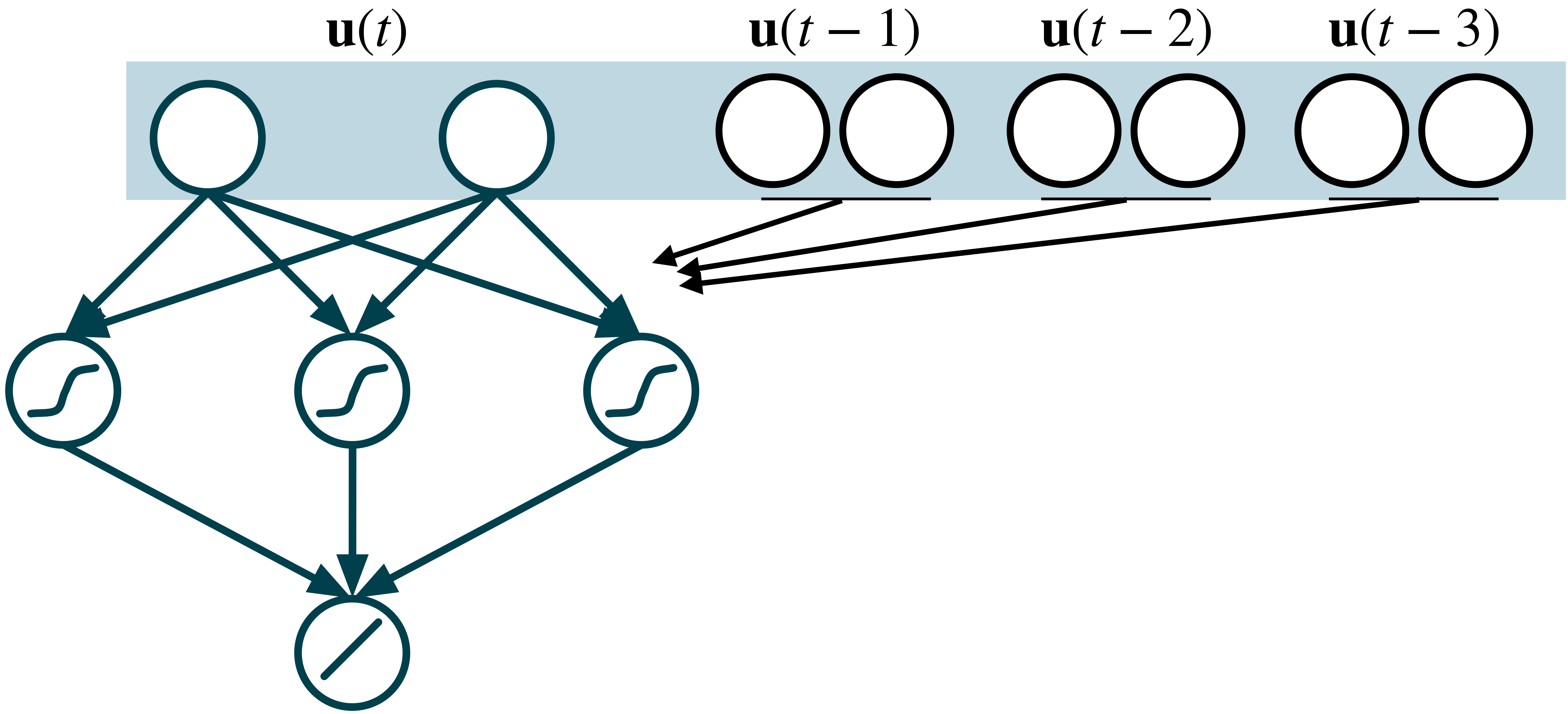
See also O. Obst and M. Riedmiller, "Taming the reservoir: Feedforward training for recurrent neural networks," 2012, doi: 10.1109/IJCNN.2012.6252506.



See also O. Obst and M. Riedmiller, "Taming the reservoir: Feedforward training for recurrent neural networks," 2012, doi: 10.1109/IJCNN.2012.6252506.

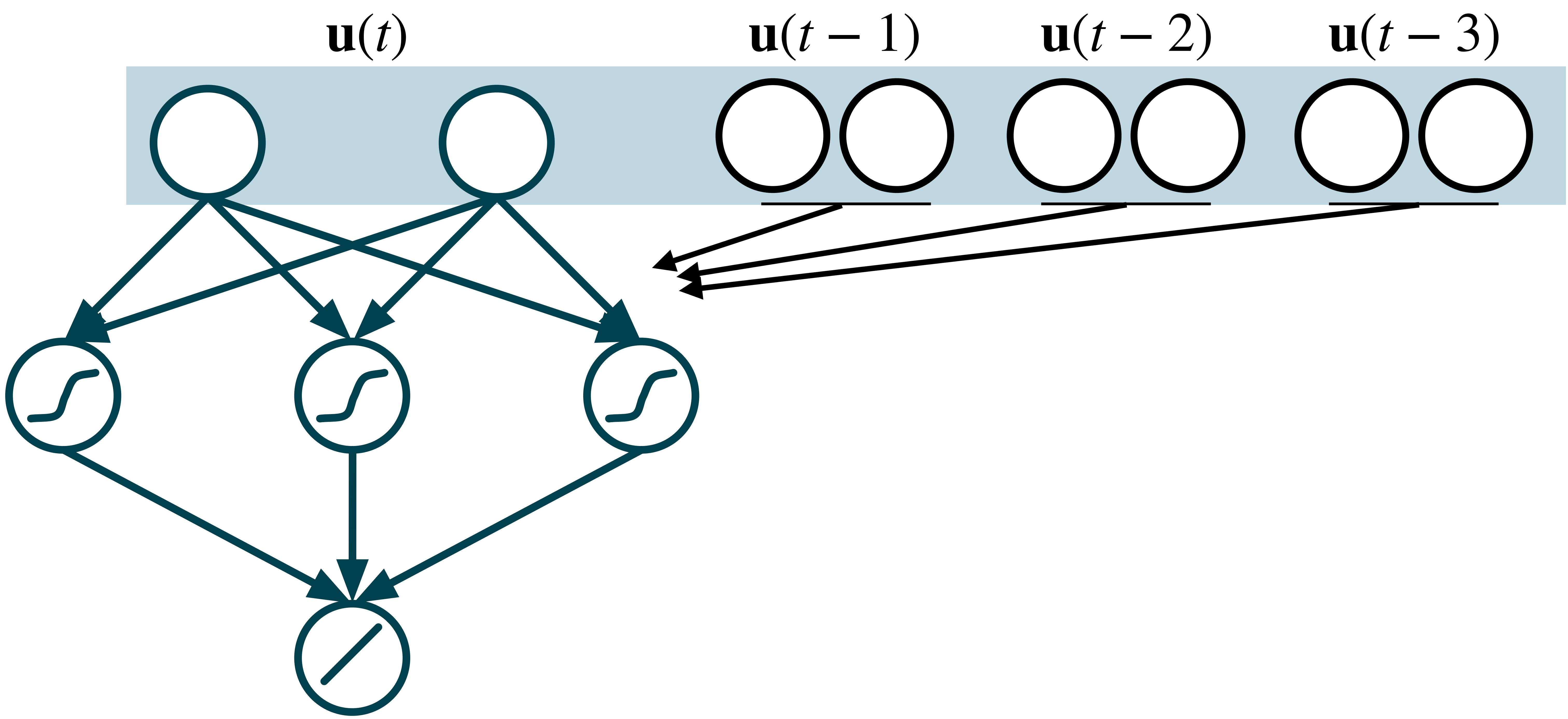


See also O. Obst and M. Riedmiller, "Taming the reservoir: Feedforward training for recurrent neural networks," 2012, doi: 10.1109/IJCNN.2012.6252506.



$$\mathbf{u}_t = [\mathbf{u}(t), \mathbf{u}(t-1), \mathbf{u}(t-2), \mathbf{u}(t-3)]$$

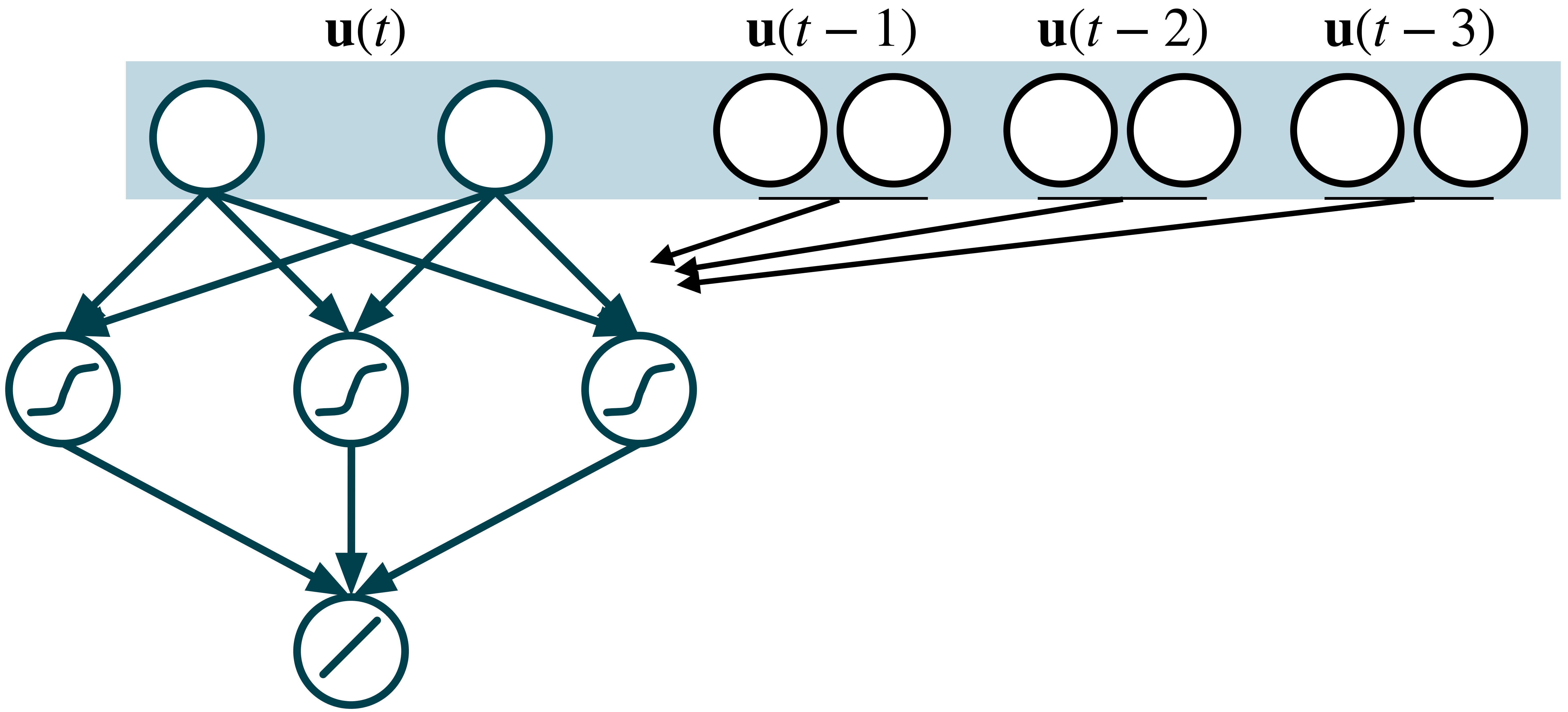
See also O. Obst and M. Riedmiller, "Taming the reservoir: Feedforward training for recurrent neural networks," 2012, doi: 10.1109/IJCNN.2012.6252506.



$$\mathbf{u}_t = [\mathbf{u}(t), \mathbf{u}(t-1), \mathbf{u}(t-2), \mathbf{u}(t-3)]$$

$$\mathbf{x}_t = \tanh(\mathbf{w}^{\text{in}}\,\mathbf{u}_t)$$

See also O. Obst and M. Riedmiller, "Taming the reservoir: Feedforward training for recurrent neural networks," 2012, doi: 10.1109/IJCNN.2012.6252506.

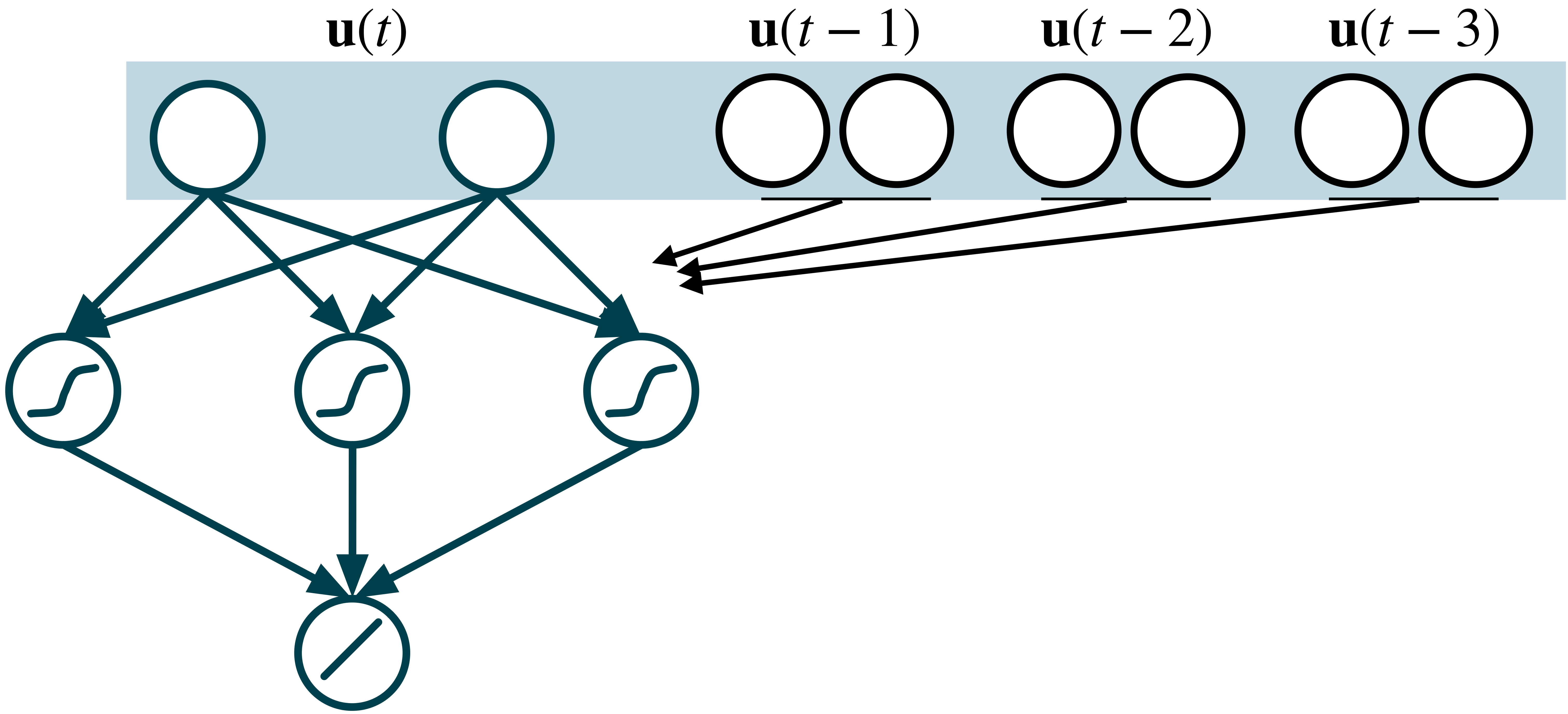


$$\mathbf{u}_t = [\mathbf{u}(t), \mathbf{u}(t-1), \mathbf{u}(t-2), \mathbf{u}(t-3)]$$

$$\mathbf{x}_t = \tanh(\mathbf{w}^{\text{in}}\,\mathbf{u}_t)$$

$$\hat{y}_t = \mathbf{W}^{\text{out}}\,\mathbf{x}_t$$

See also O. Obst and M. Riedmiller, "Taming the reservoir: Feedforward training for recurrent neural networks," 2012, doi: 10.1109/IJCNN.2012.6252506.



$$\mathbf{u}_t = [\mathbf{u}(t), \mathbf{u}(t-1), \mathbf{u}(t-2), \mathbf{u}(t-3)]$$

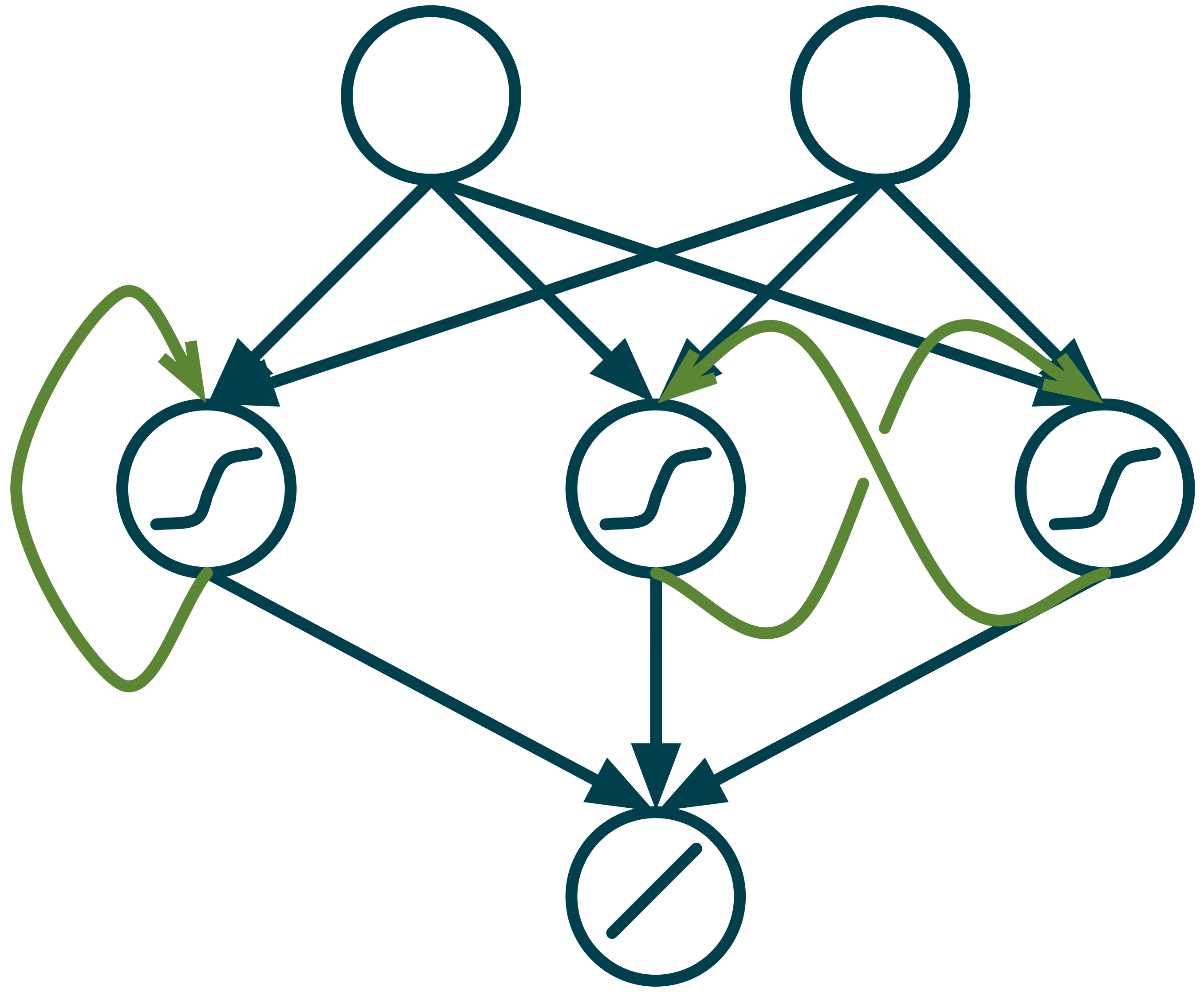
$$\mathbf{x}_t = \tanh(\mathbf{w}^{\text{in}}\, \mathbf{u}_t)$$

$$\hat{y}_t = \mathbf{w}^{\text{out}}\, \mathbf{x}_t$$

$$\hat{y}_t = f_\theta(\mathbf{u}_t)$$

$$\theta = \{\mathbf{w}^{\text{in}}, \mathbf{w}^{\text{out}}\}$$

See also O. Obst and M. Riedmiller, "Taming the reservoir: Feedforward training for recurrent neural networks," 2012, doi: 10.1109/IJCNN.2012.6252506.



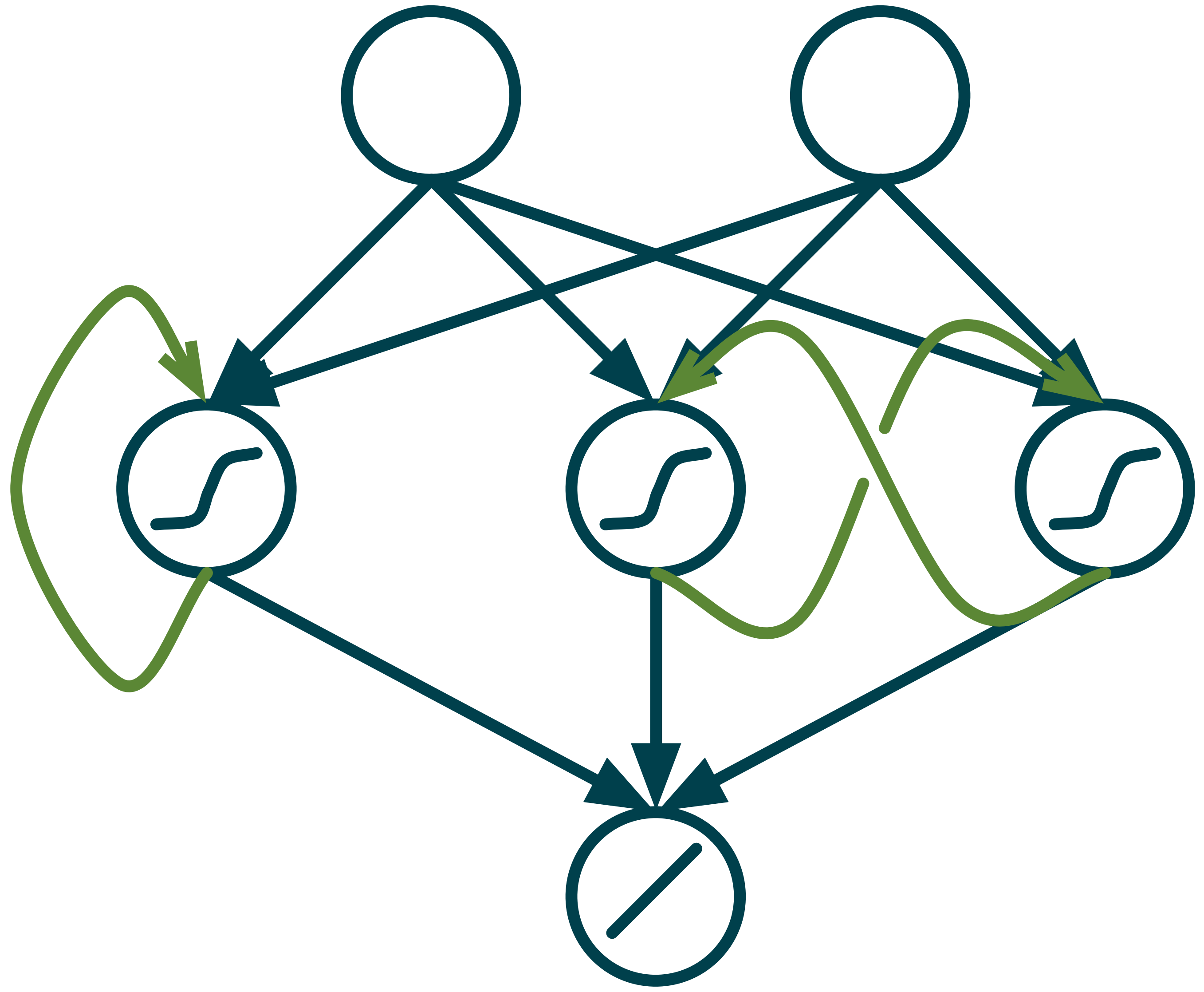
Recurrent neural network

- can represent dynamical systems

- **internal state:**

  a fading memory, dependent on previous input

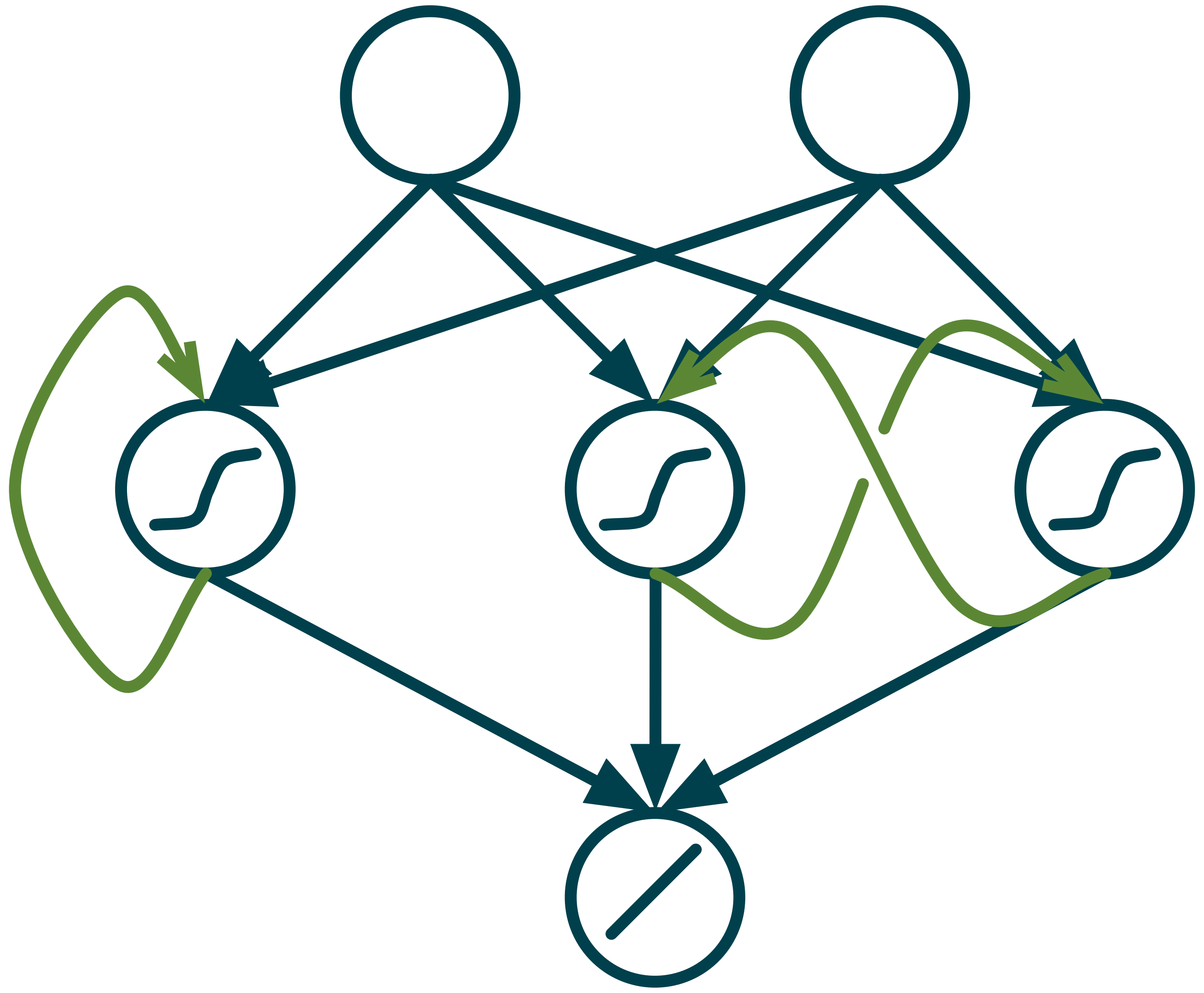
- “Training”: optimisation — same as before, but now: slow and issues with fading / exploding gradients



$\mathbf{u}(t)$

Recurrent neural network

- can represent dynamical systems
- **internal state:**
  a fading memory, dependent on previous input
- “Training”: optimisation — same as before, but now: slow and issues with fading / exploding gradients

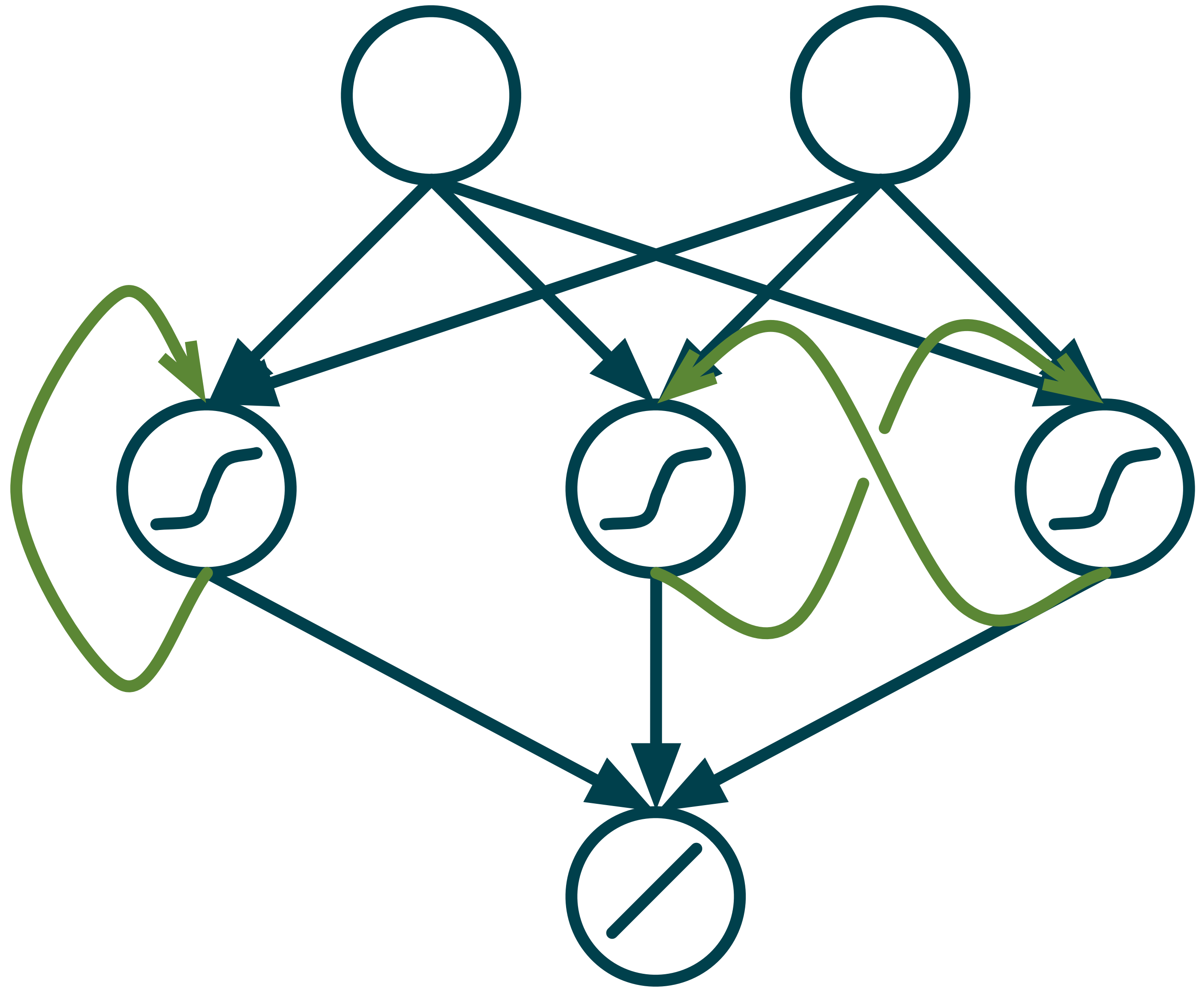


$\mathbf{u}(t)$

$$\mathbf{x}(t) = \tanh(\mathbf{W}\,\mathbf{x}(t-1) + \mathbf{w}^{\text{in}}\,\mathbf{u}(t))$$

Recurrent neural network

- can represent dynamical systems
- **internal state:**
  a fading memory, dependent on previous input
- “Training”: optimisation — same as before, but now: slow and issues with fading / exploding gradients



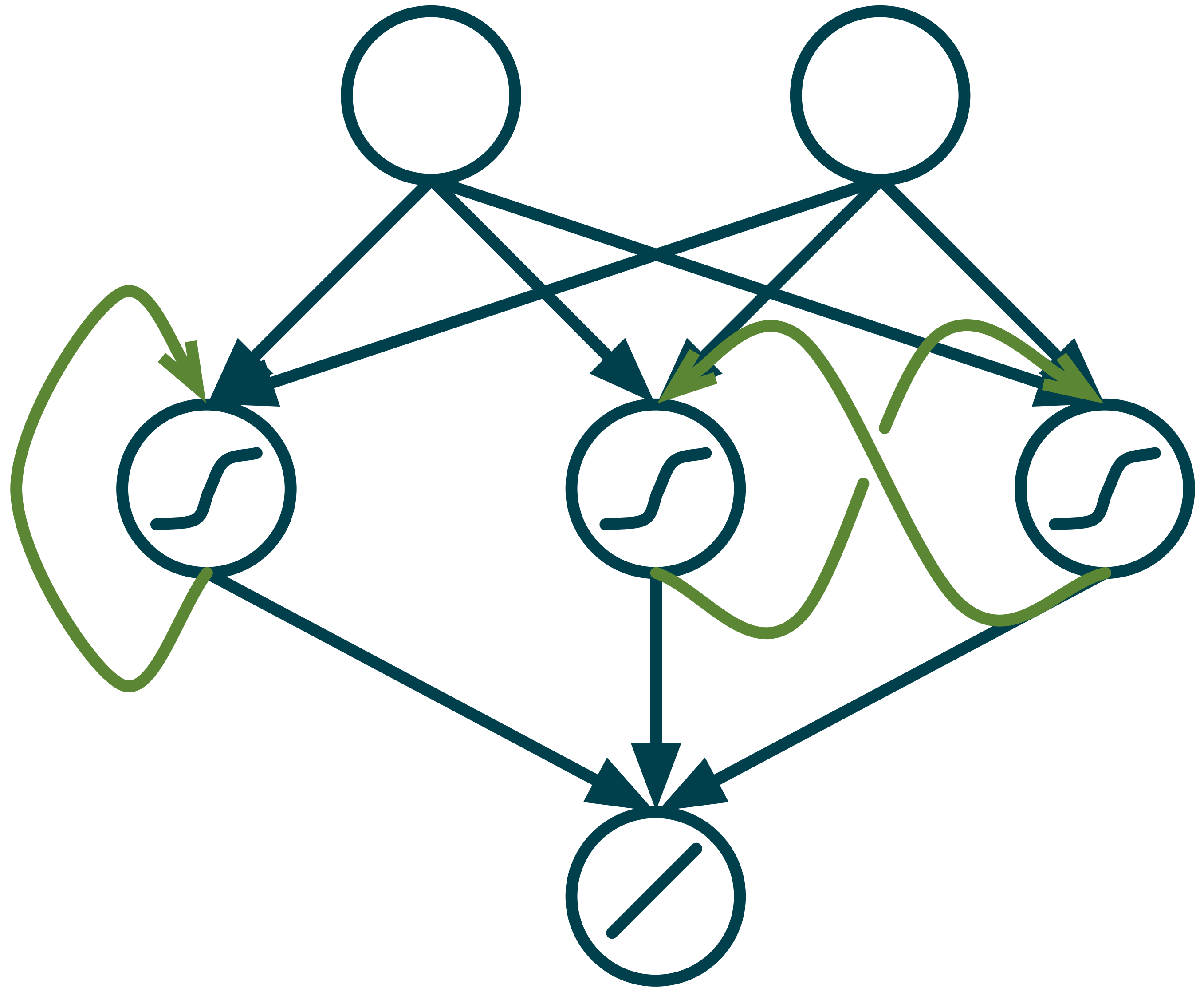
$\mathbf{u}(t)$

$$\mathbf{x}(t) = \tanh(\mathbf{W}\,\mathbf{x}(t-1) + \mathbf{w}^{\text{in}}\,\mathbf{u}(t))$$

$$\hat{y}(t) = \mathbf{w}^{\text{out}}\,\mathbf{x}(t)$$

Recurrent neural network

- can represent dynamical systems
- **internal state:**
  a fading memory, dependent on previous input
- “Training”: optimisation — same as before, but now: slow and issues with fading / exploding gradients



$$\mathbf{u}(t)$$

$$\mathbf{x}(t) = \tanh(\mathbf{W}\,\mathbf{x}(t-1) + \mathbf{w}^{\text{in}}\,\mathbf{u}(t))$$

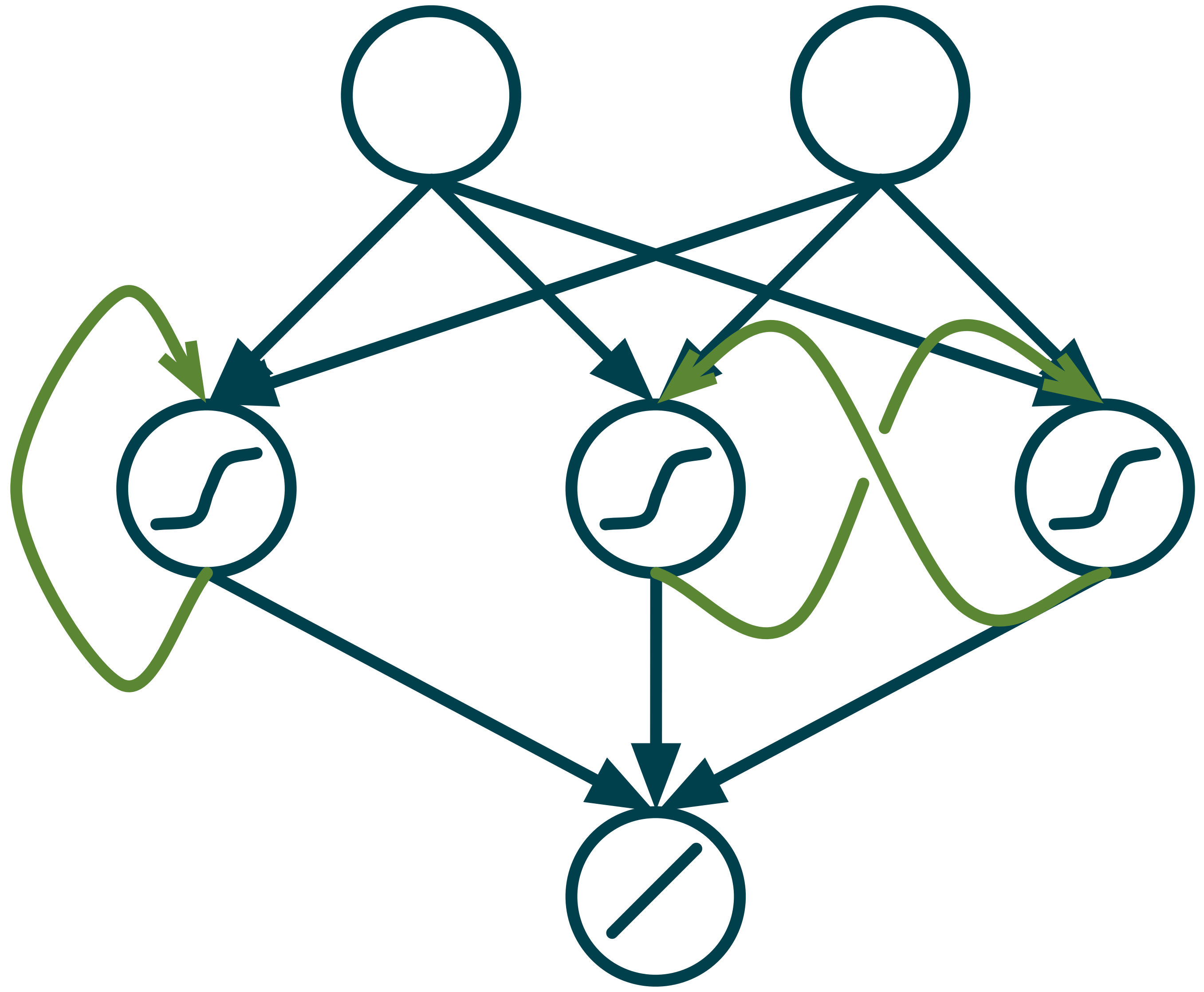
$$\hat{y}(t) = \mathbf{w}^{\text{out}}\,\mathbf{x}(t)$$

$$\hat{y}_t = F_\theta(\mathbf{u}_t, t)$$

$$\theta = \{\mathbf{w}^{\text{in}}, \mathbf{W}, \mathbf{w}^{\text{out}}\}$$

Recurrent neural network

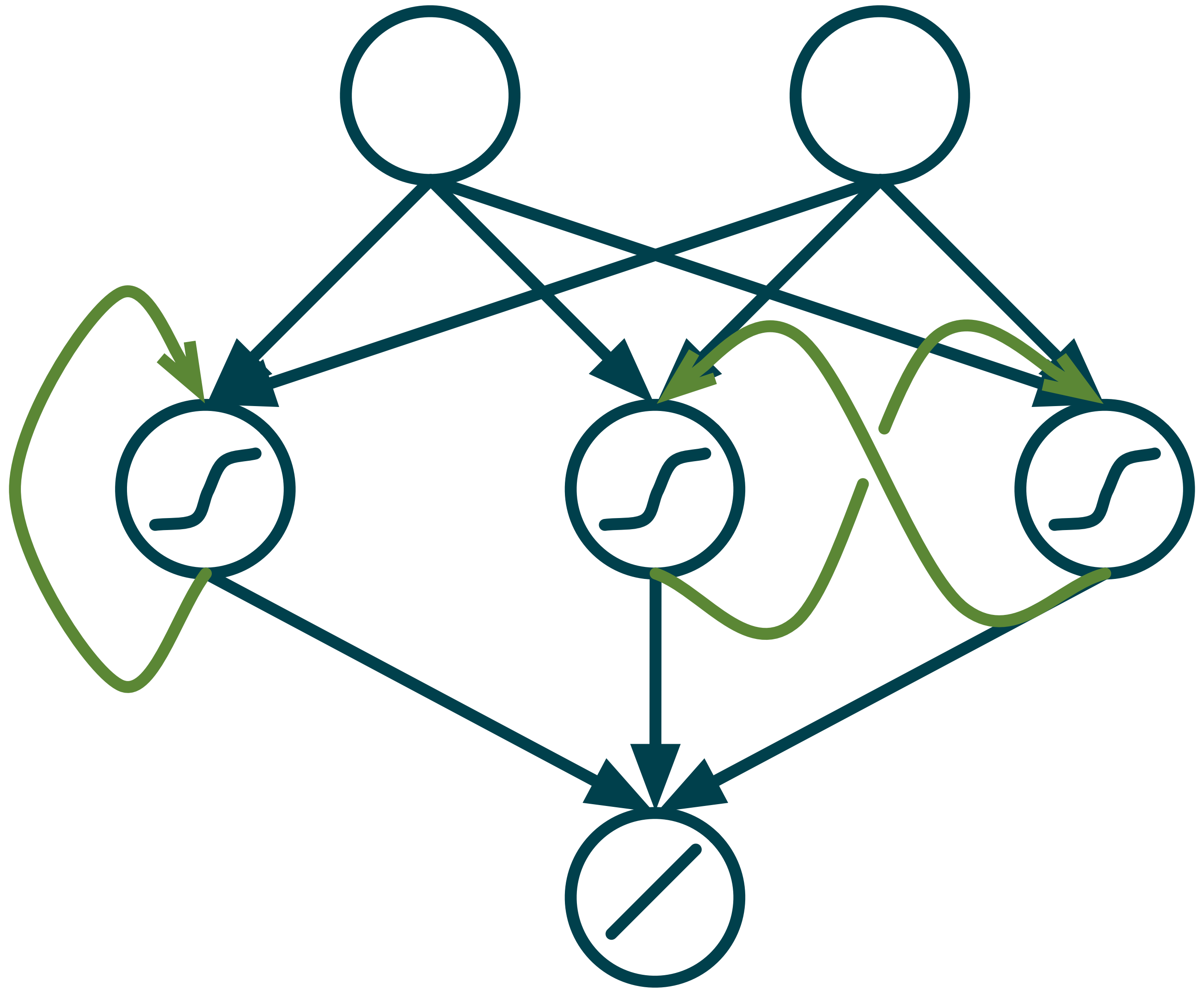
- can represent dynamical systems
- **internal state:**
  a fading memory, dependent on previous input
- “Training”: optimisation — same as before, but now: slow and issues with fading / exploding gradients



$\mathbf{u}(t)$

$\mathbf{x}(t) = \tanh(\mathbf{W}\,\mathbf{x}(t-1) + \mathbf{w}^{\text{in}}\,\mathbf{u}(t))$

$\hat{y}(t) = \mathbf{w}^{\text{out}}\,\mathbf{x}(t)$



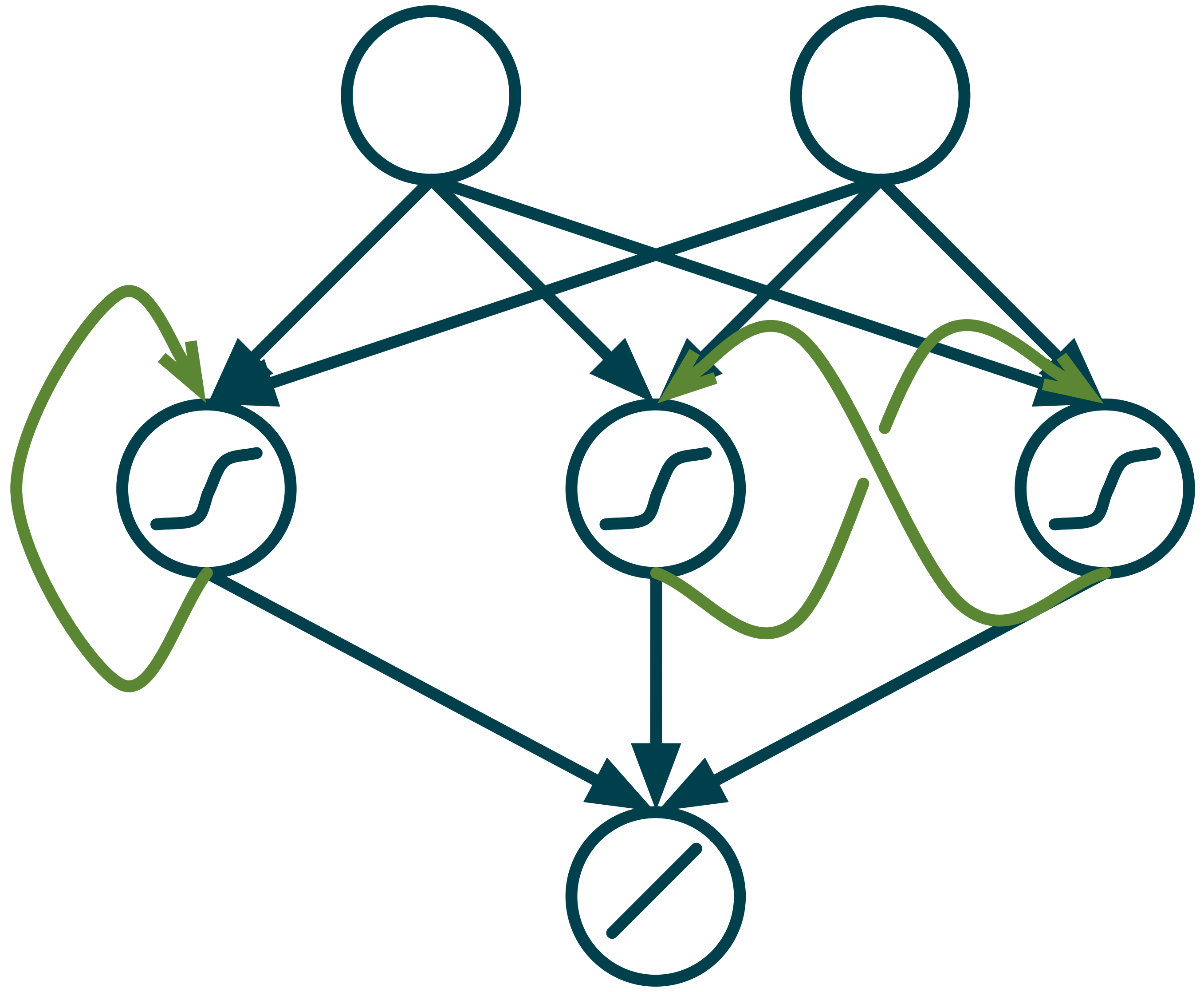
$$\mathbf{u}(t)$$

$$\mathbf{x}(t) = \tanh(\mathbf{W}\,\mathbf{x}(t-1) + \mathbf{w}^{\text{in}}\,\mathbf{u}(t))$$

$$\hat{y}(t) = \mathbf{w}^{\text{out}}\,\mathbf{x}(t)$$

“Echo state” network training:

(Jaeger & Haas, Science, 2004)



$\mathbf{u}(t)$

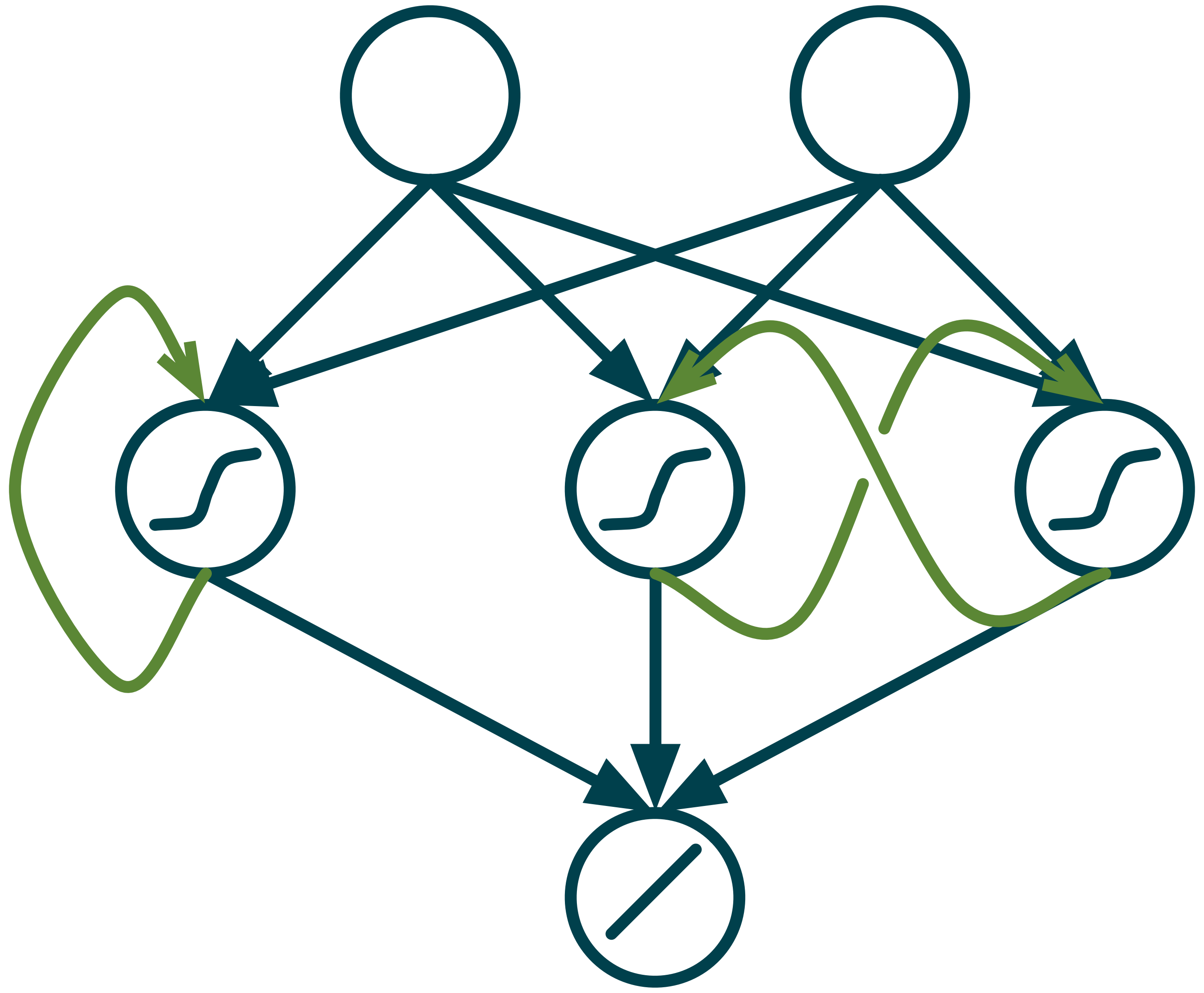
$$\mathbf{x}(t) = \tanh(\mathbf{W}\,\mathbf{x}(t-1) + \mathbf{w}^{\text{in}}\,\mathbf{u}(t))$$

$$\hat{y}(t) = \mathbf{w}^{\text{out}}\,\mathbf{x}(t)$$

"Echo state" network training:

(Jaeger & Haas, Science, 2004)

1. Create random weights $\mathbf{w}^{\text{in}}$ and $\mathbf{W}$.



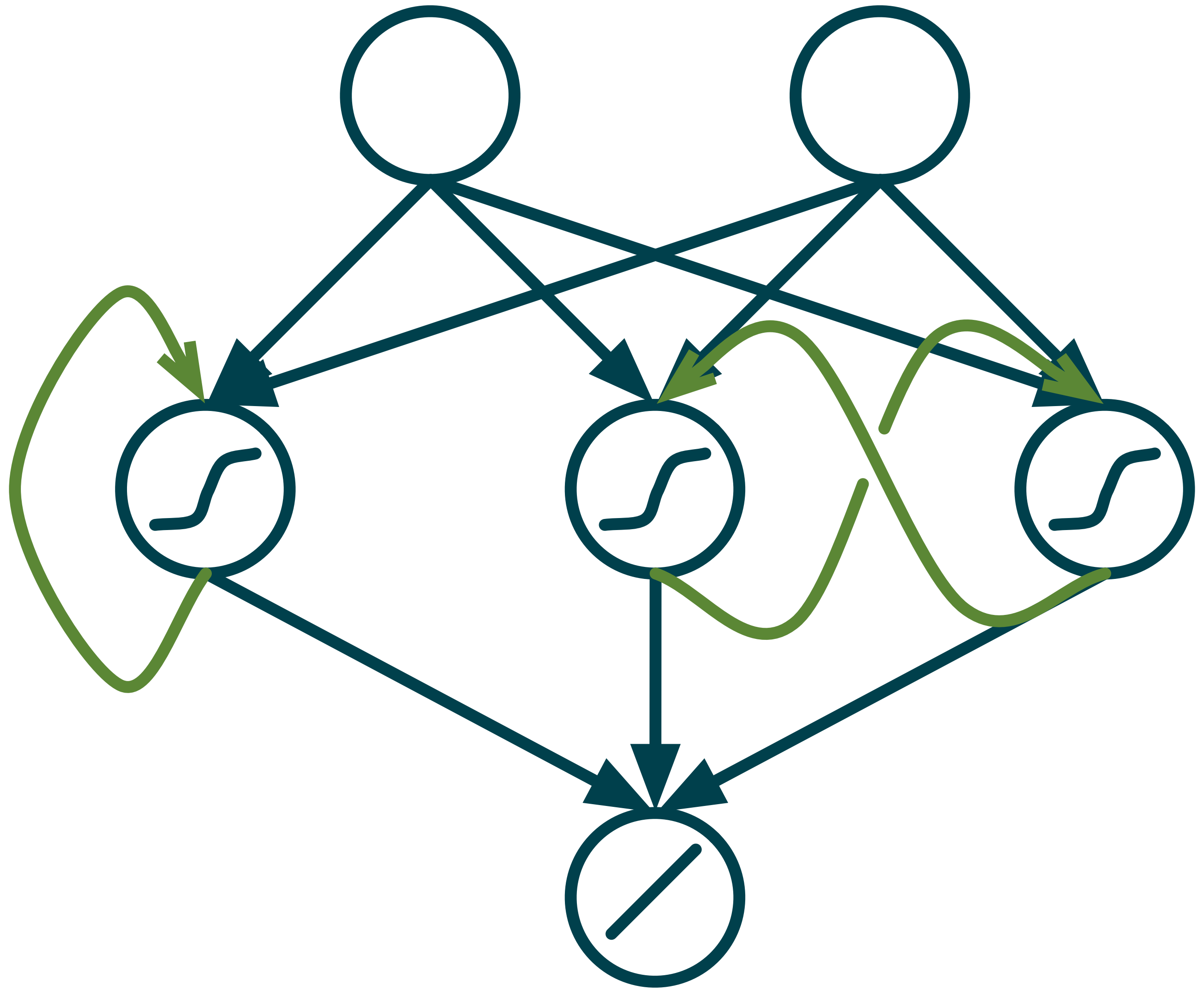
$\mathbf{u}(t)$

$\mathbf{x}(t) = \tanh(\mathbf{W}\,\mathbf{x}(t-1) + \mathbf{w}^{\text{in}}\,\mathbf{u}(t))$

$\hat{y}(t) = \mathbf{w}^{\text{out}}\,\mathbf{x}(t)$

"Echo state" network training:
(Jaeger & Haas, Science, 2004)

1. Create random weights $\mathbf{w}^{\text{in}}$ and $\mathbf{W}$.
2. Training data $(\mathbf{U}^{\text{train}}, \mathbf{y}^{\text{train}})$, with T steps. Run this through your network.



$\mathbf{u}(t)$

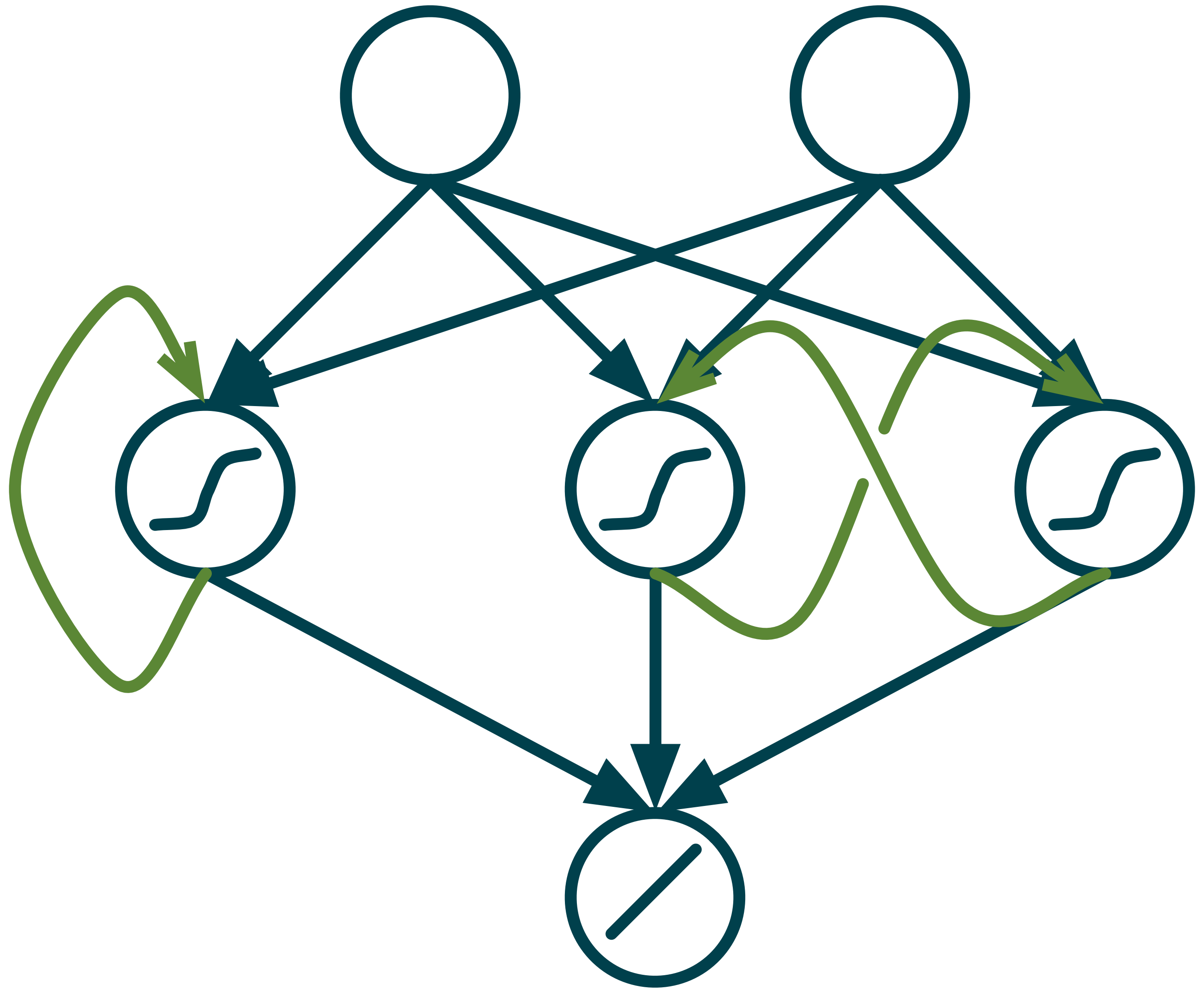
$$\mathbf{x}(t) = \tanh(\mathbf{W}\,\mathbf{x}(t-1) + \mathbf{w}^{\text{in}}\,\mathbf{u}(t))$$

$$\hat{y}(t) = \mathbf{w}^{\text{out}}\,\mathbf{x}(t)$$

"Echo state" network training:

(Jaeger & Haas, Science, 2004)

1. Create random weights $\mathbf{w}^{\text{in}}$ and $\mathbf{W}$.
2. Training data $(\mathbf{U}^{\text{train}}, \mathbf{y}^{\text{train}})$, with T steps. Run this through your network.
   a) Collect all the vectors $\mathbf{x}(t)$ in a state collection matrix $\mathbf{M}$.



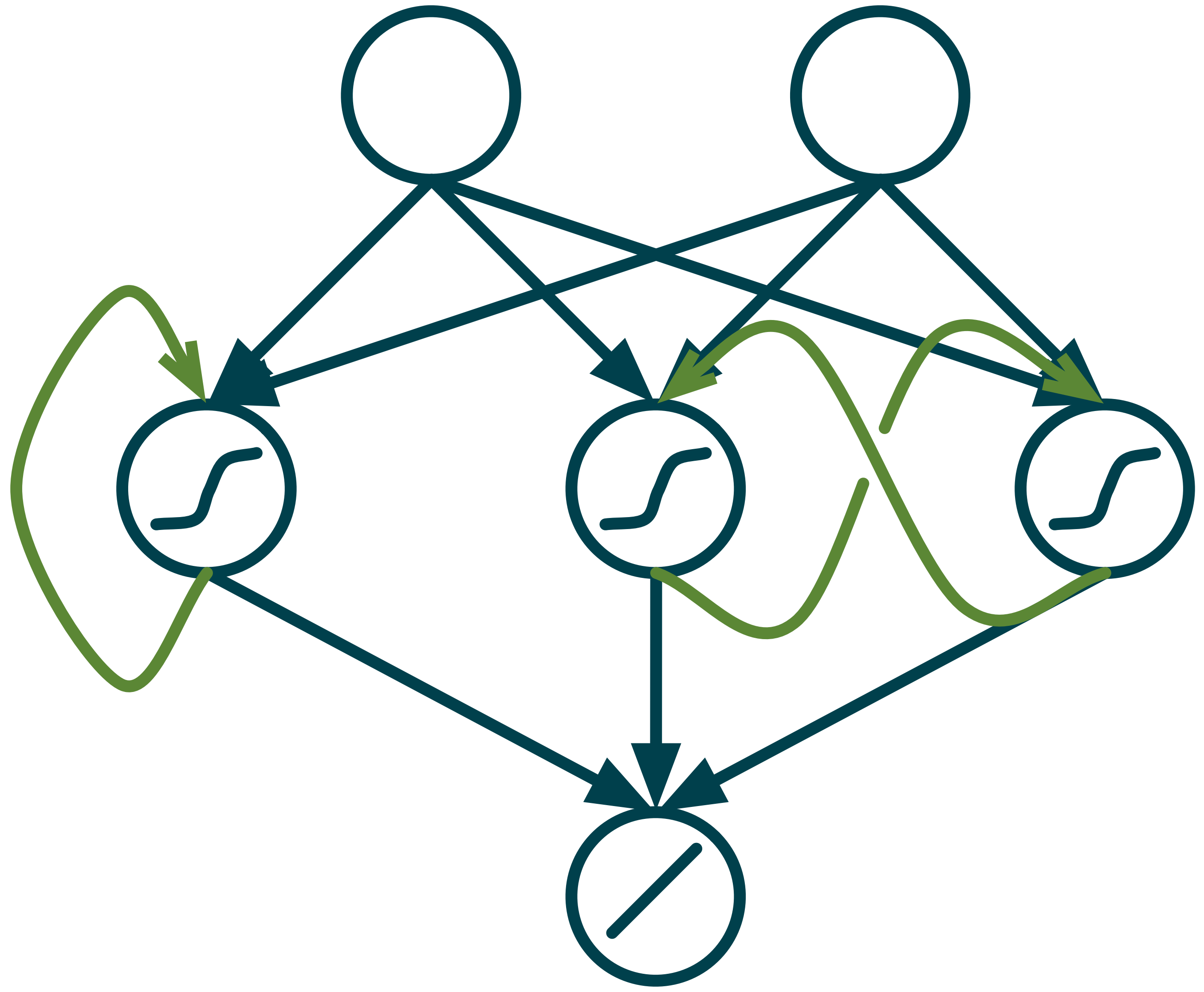
$\mathbf{u}(t)$

$\mathbf{x}(t) = \tanh(\mathbf{W}\,\mathbf{x}(t-1) + \mathbf{w}^{\text{in}}\,\mathbf{u}(t))$

$\hat{y}(t) = \mathbf{w}^{\text{out}}\,\mathbf{x}(t)$

"Echo state" network training:
(Jaeger & Haas, Science, 2004)

1. Create random weights $\mathbf{w}^{\text{in}}$ and $\mathbf{W}$.
2. Training data $(\mathbf{U}^{\text{train}}, \mathbf{y}^{\text{train}})$, with T steps. Run this through your network.
    a) Collect all the vectors $\mathbf{x}(t)$ in a state collection matrix $\mathbf{M}$.
    b) Collect all desired outputs in $\mathbf{d}$.



$\mathbf{u}(t)$

$\mathbf{x}(t) = \tanh(\mathbf{W}\,\mathbf{x}(t-1) + \mathbf{w}^{\text{in}}\,\mathbf{u}(t))$

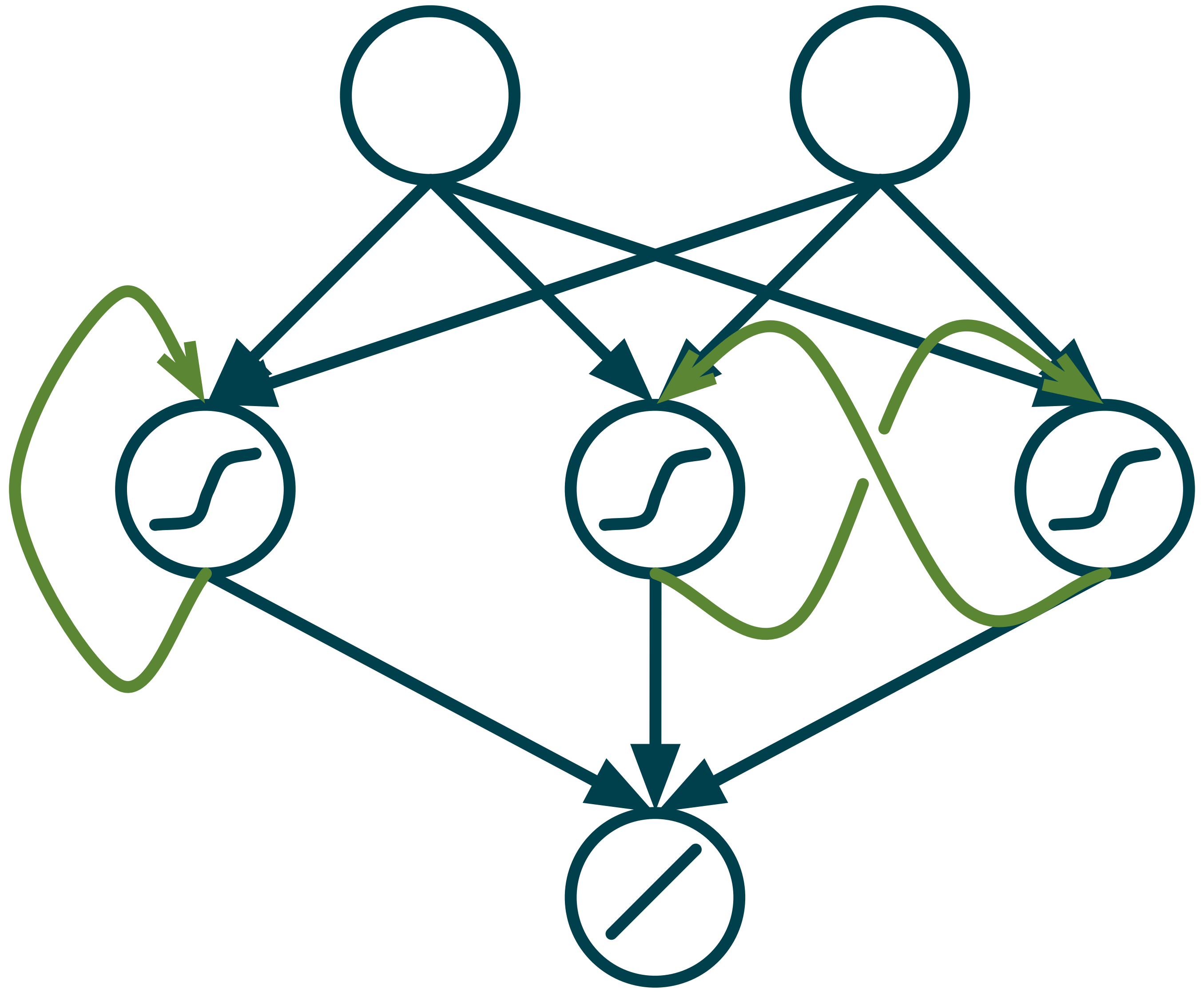
$\hat{y}(t) = \mathbf{w}^{\text{out}}\,\mathbf{x}(t)$

"Echo state" network training:

(Jaeger & Haas, Science, 2004)

1. Create random weights $\mathbf{w}^{\text{in}}$ and $\mathbf{W}$.
2. Training data $(\mathbf{U}^{\text{train}}, \mathbf{y}^{\text{train}})$, with T steps. Run this through your network.
   a) Collect all the vectors $\mathbf{x}(t)$ in a state collection matrix $\mathbf{M}$.
   b) Collect all desired outputs in $\mathbf{d}$.
3. Compute the output weights $\mathbf{w}^{\text{out}}$:

$$w^{\text{out}} = (\mathbf{M}^{-1}\mathbf{d})^{\top}$$



$\mathbf{u}(t)$

$\mathbf{x}(t) = \tanh(\mathbf{W}\,\mathbf{x}(t-1) + \mathbf{w}^{\text{in}}\,\mathbf{u}(t))$

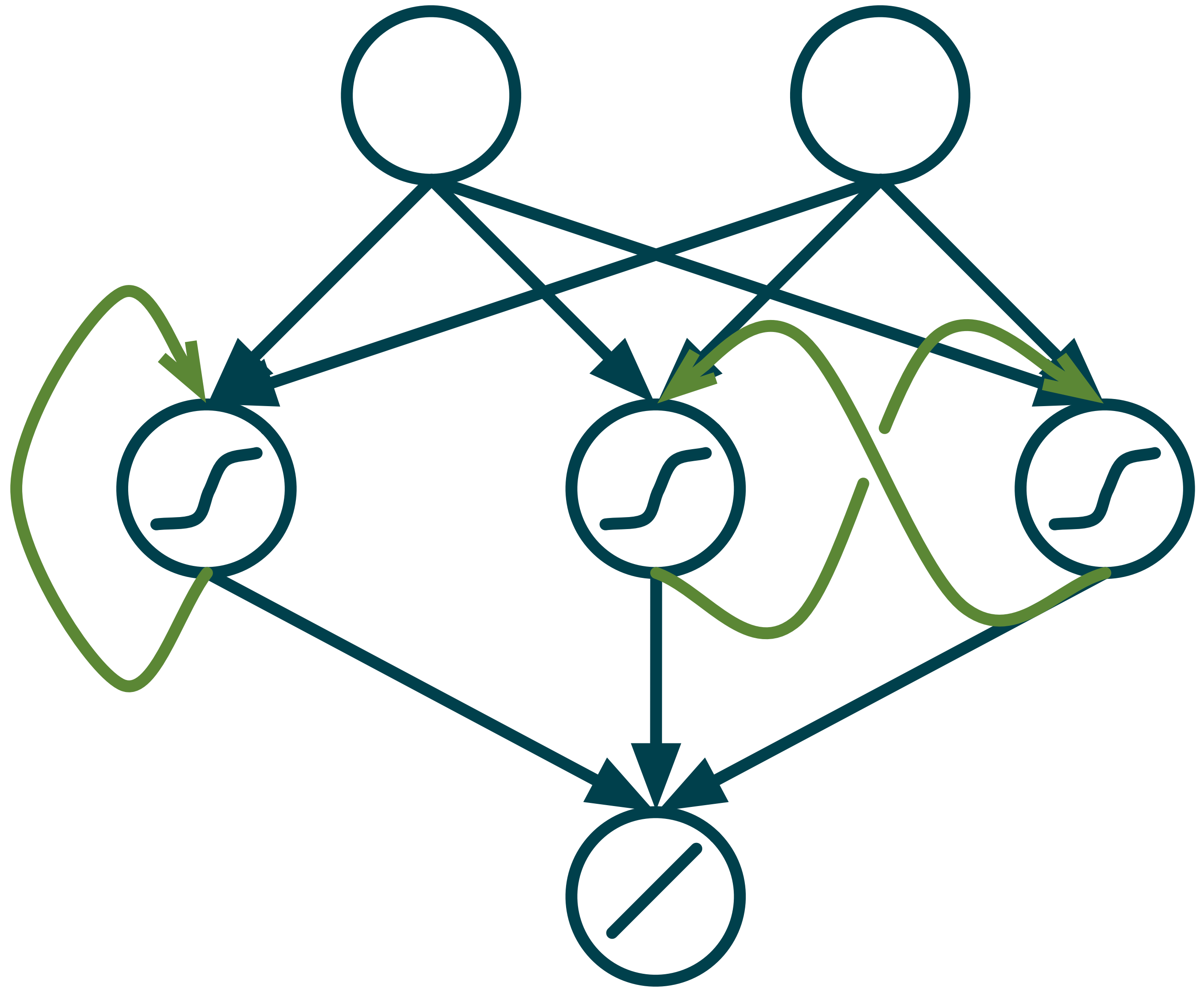
$\hat{y}(t) = \mathbf{w}^{\text{out}}\,\mathbf{x}(t)$

"Echo state" network training:
(Jaeger & Haas, Science, 2004)

1. Create random weights $\mathbf{w}^{\text{in}}$ and $\mathbf{W}$.
2. Training data $(\mathbf{U}^{\text{train}}, \mathbf{y}^{\text{train}})$, with T steps. Run this through your network.
   a) Collect all the vectors $\mathbf{x}(t)$ in a state collection matrix $\mathbf{M}$.
   b) Collect all desired outputs in $\mathbf{d}$.
3. Compute the output weights $\mathbf{w}^{\text{out}}$:

$$w^{\text{out}} = (\mathbf{M}^{-1}\mathbf{d})^{\top}$$

Notes:
- We use the pseudo-inverse, $\mathbf{M}^{\dagger}$ because $\mathbf{M}$ is usually not square



$\mathbf{u}(t)$

$\mathbf{x}(t) = \tanh(\mathbf{W}\,\mathbf{x}(t-1) + \mathbf{w}^{\text{in}}\,\mathbf{u}(t))$

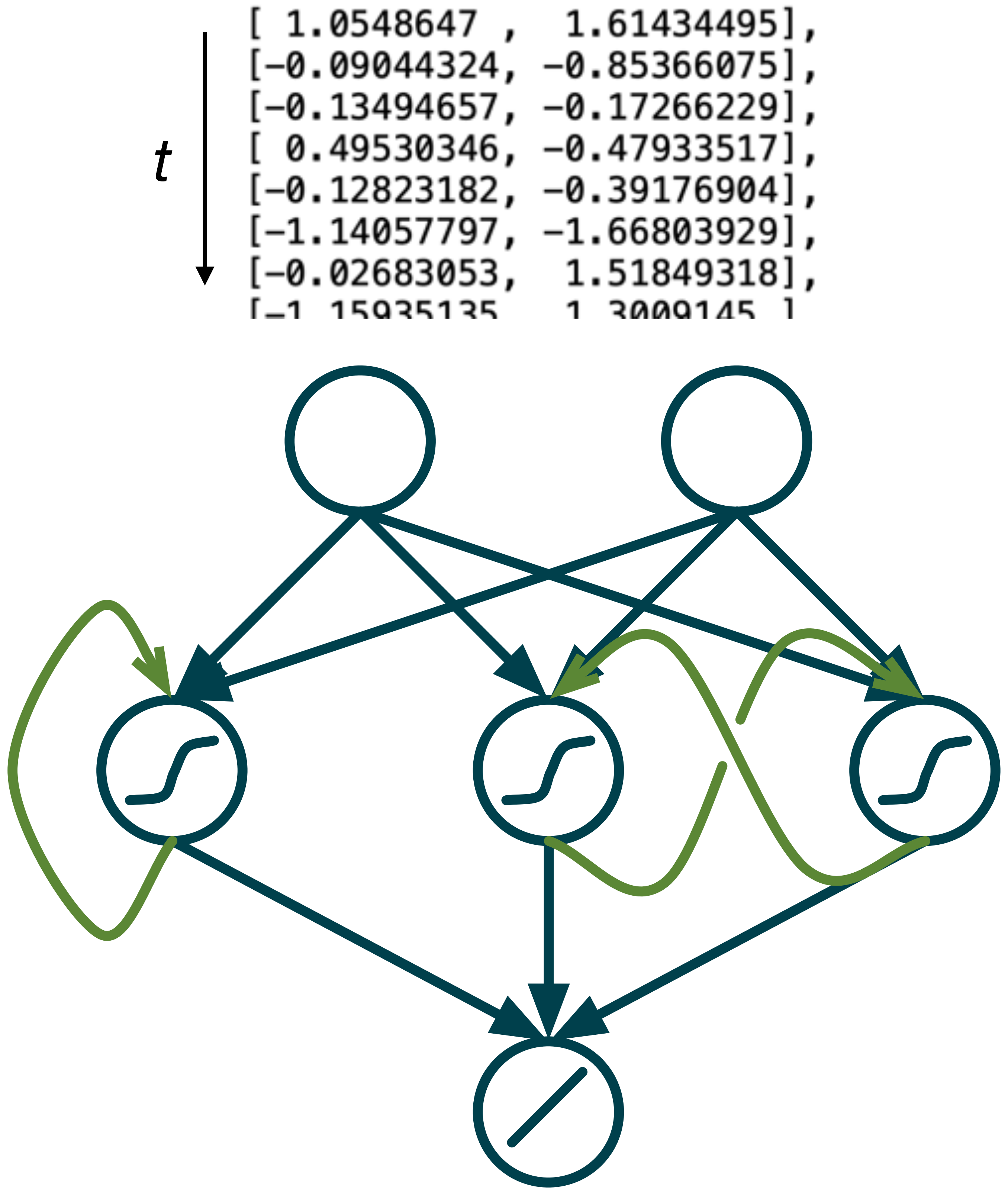
$\hat{y}(t) = \mathbf{w}^{\text{out}}\,\mathbf{x}(t)$

"Echo state" network training:
(Jaeger & Haas, Science, 2004)

1. Create random weights $\mathbf{w}^{\text{in}}$ and $\mathbf{W}$.
2. Training data $(\mathbf{U}^{\text{train}}, \mathbf{y}^{\text{train}})$, with T steps. Run this through your network.
   a) Collect all the vectors $\mathbf{x}(t)$ in a state collection matrix $\mathbf{M}$.
   b) Collect all desired outputs in $\mathbf{d}$.
3. Compute the output weights $\mathbf{w}^{\text{out}}$:

$$w^{\text{out}} = (\mathbf{M}^{-1}\mathbf{d})^{\top}$$

Notes:
- We use the pseudo-inverse, $\mathbf{M}^{\dagger}$ because $\mathbf{M}$ is usually not square
- This is solving a *linear* regression problem, with $\mathbf{x}$ as inputs.



$\mathbf{u}(t)$

$$\mathbf{x}(t) = \tanh(\mathbf{W}\,\mathbf{x}(t-1) + \mathbf{w}^{\text{in}}\,\mathbf{u}(t))$$

$$\hat{y}(t) = \mathbf{w}^{\text{out}}\,\mathbf{x}(t)$$

"Echo state" network training:

(Jaeger & Haas, Science, 2004)

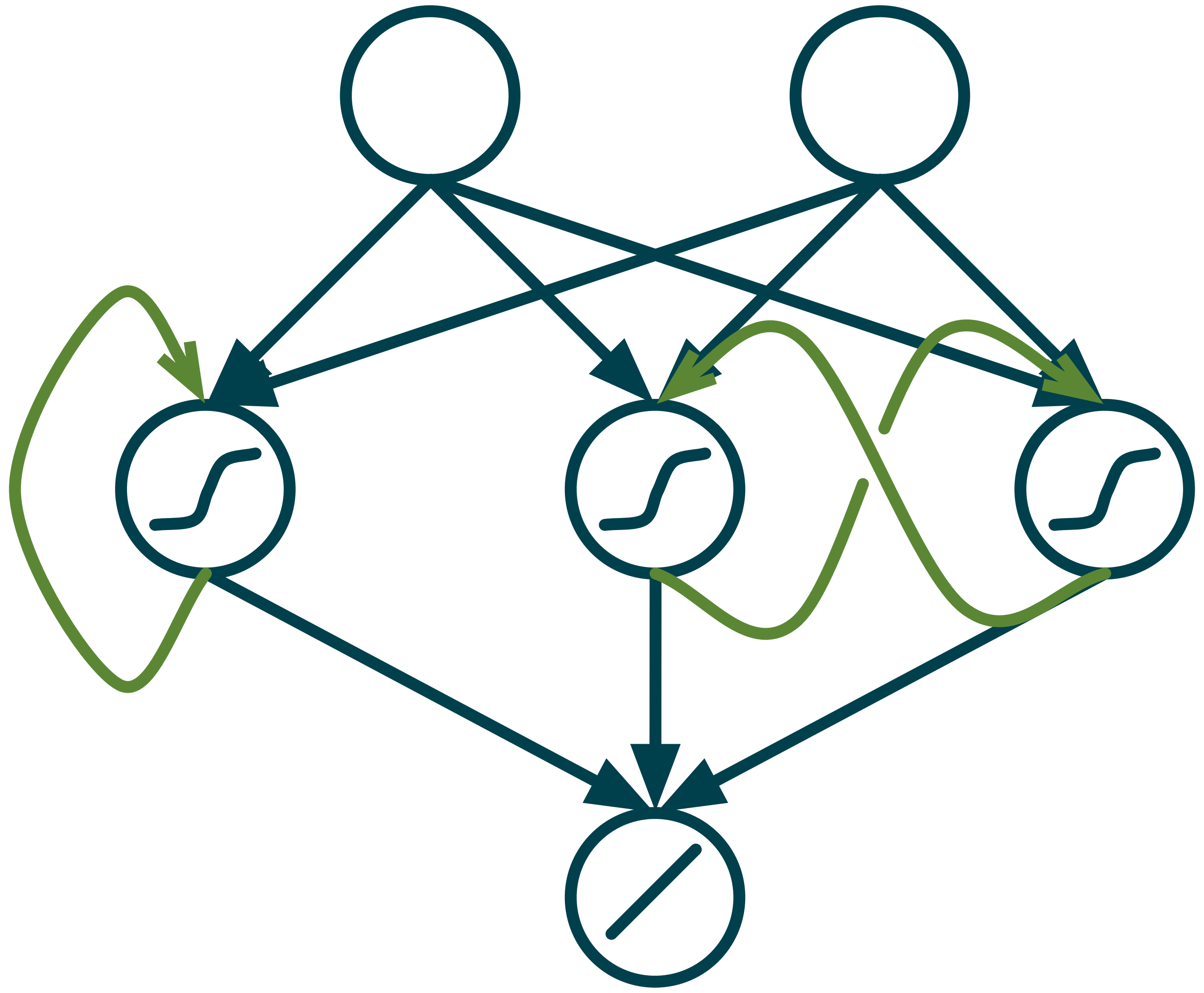
1. Create random weights $\mathbf{w}^{\text{in}}$ and $\mathbf{W}$.
2. Training data $(\mathbf{U}^{\text{train}}, \mathbf{y}^{\text{train}})$, with T steps. Run this through your network.
   a) Collect all the vectors $\mathbf{x}(t)$ in a state collection matrix $\mathbf{M}$.
   b) Collect all desired outputs in $\mathbf{d}$.
3. Compute the output weights $\mathbf{w}^{\text{out}}$:

$$w^{\text{out}} = (\mathbf{M}^{-1}\mathbf{d})^{\top}$$

Notes:

- We use the pseudo-inverse, $\mathbf{M}^{\dagger}$ because $\mathbf{M}$ is usually not square
- This is solving a *linear* regression problem, with $\mathbf{x}$ as inputs.

```
t ↓
[ 1.0548647 ,  1.61434495],
[-0.09044324, -0.85366075],
[-0.13494657, -0.17266229],
[ 0.49530346, -0.47933517],
[-0.12823182, -0.39176904],
[-1.14057797, -1.66803929],
[-0.02683053,  1.51849318],
```



$\mathbf{u}(t)$

$$\mathbf{x}(t) = \tanh(\mathbf{W}\,\mathbf{x}(t-1) + \mathbf{w}^{\text{in}}\,\mathbf{u}(t))$$

$$\hat{y}(t) = \mathbf{w}^{\text{out}}\,\mathbf{x}(t)$$

$\mathbf{M}$

```
t ↓
[ 0.46144053, -0.1853432 , -1.2101936 ],
[ 0.48214552,  1.23107819, -0.09684498],
[-0.18654759,  0.37844562,  2.21832035],
[ 1.21168797, -0.02469393, -0.50769134],
[-1.38382318, -1.44613351,  0.39553202],
[-1.00392836, -0.7489974 , -0.02276408],
[ 0.83296682, -2.43909498,  0.96621259],
```

"Echo state" network training:
(Jaeger & Haas, Science, 2004)

1. Create random weights $\mathbf{w}^{\text{in}}$ and $\mathbf{W}$.
2. Training data $(\mathbf{U}^{\text{train}}, \mathbf{y}^{\text{train}})$, with T steps. Run this through your network.
   a) Collect all the vectors $\mathbf{x}(t)$ in a state collection matrix $\mathbf{M}$.
   b) Collect all desired outputs in $\mathbf{d}$.
3. Compute the output weights $\mathbf{w}^{\text{out}}$:

$$w^{\text{out}} = (\mathbf{M}^{-1}\mathbf{d})^{\top}$$

Notes:
- We use the pseudo-inverse, $\mathbf{M}^{\dagger}$ because $\mathbf{M}$ is usually not square
- This is solving a *linear* regression problem, with $\mathbf{x}$ as inputs.

“Echo state” network training:
(Jaeger & Haas, Science, 2004)

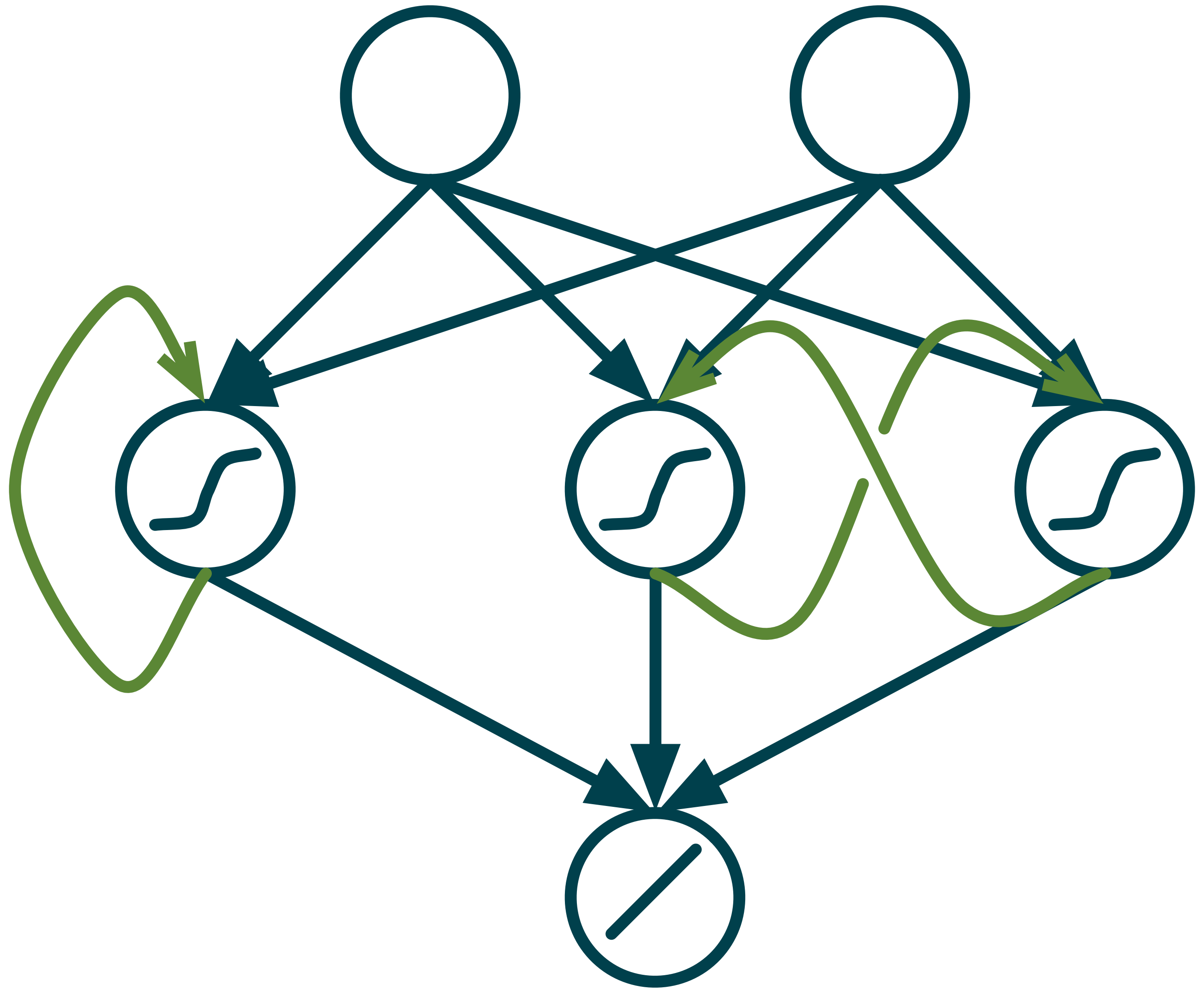
1. Create random weights $\mathbf{w}^{\text{in}}$ and $\mathbf{W}$.
2. Training data $(\mathbf{U}^{\text{train}}, \mathbf{y}^{\text{train}})$, with T steps. Run this through your network.
   a) Collect all the vectors $\mathbf{x}(t)$ in a state collection matrix $\mathbf{M}$.
   b) Collect all desired outputs in $\mathbf{d}$.
3. Compute the output weights $\mathbf{w}^{\text{out}}$:

$$w^{\text{out}} = (\mathbf{M}^{-1}\mathbf{d})^{\top}$$

Notes:
- We use the pseudo-inverse, $\mathbf{M}^{\dagger}$ because $\mathbf{M}$ is usually not square
- This is solving a *linear* regression problem, with $\mathbf{x}$ as inputs.

t

```
[ 1.0548647 ,  1.61434495],
[-0.09044324, -0.85366075],
[-0.13494657, -0.17266229],
[ 0.49530346, -0.47933517],
[-0.12823182, -0.39176904],
[-1.14057797, -1.66803929],
[-0.02683053,  1.51849318],
```



$\mathbf{u}(t)$

$$\mathbf{x}(t) = \tanh(\mathbf{W}\,\mathbf{x}(t-1) + \mathbf{w}^{\text{in}}\,\mathbf{u}(t))$$

$$\hat{y}(t) = \mathbf{w}^{\text{out}}\,\mathbf{x}(t)$$

$\mathbf{M}$

t

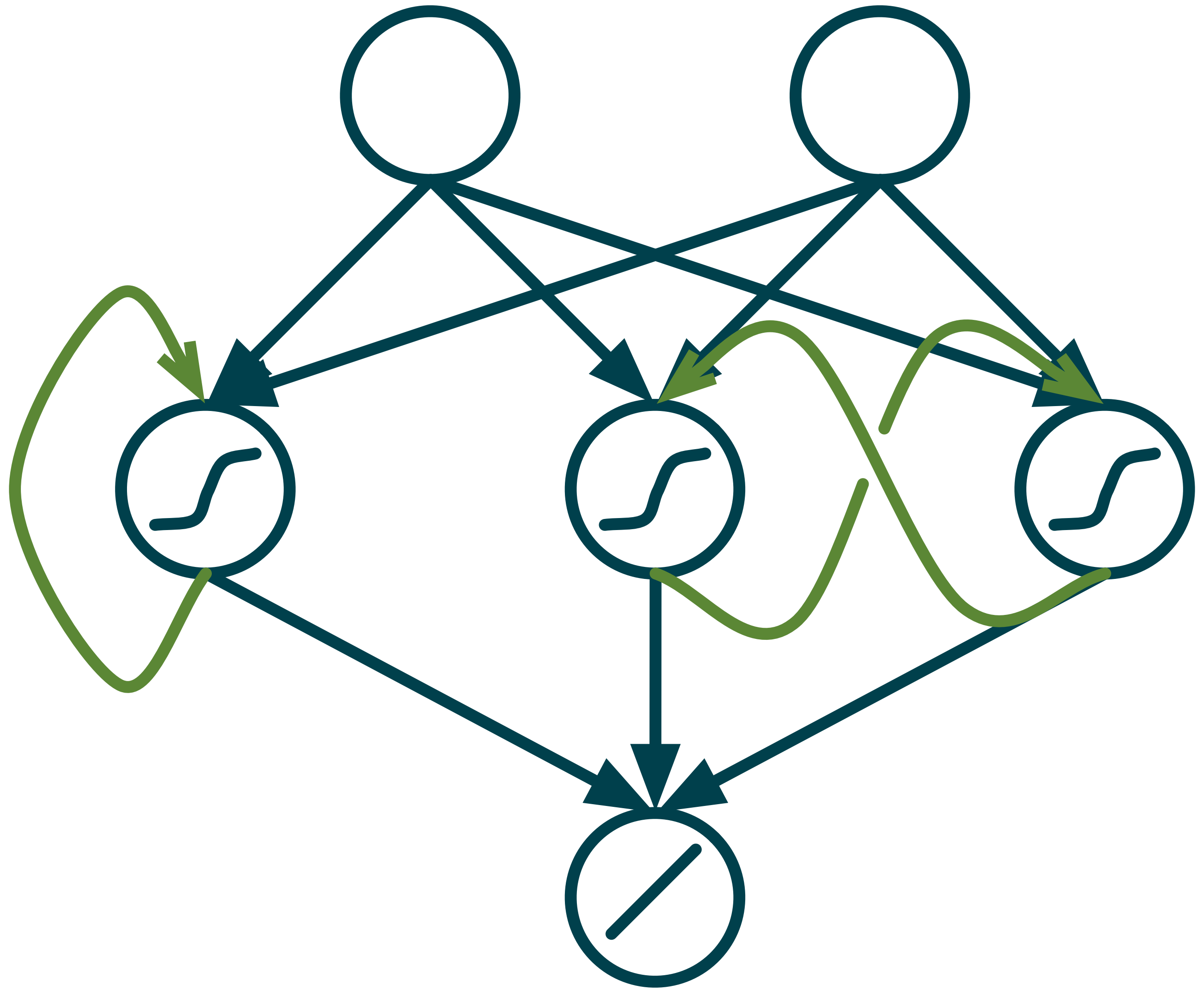
```
[ 0.46144053, -0.1853432 , -1.2101936 ],
[ 0.48214552,  1.23107819, -0.09684498],
[-0.18654759,  0.37844562,  2.21832035],
[ 1.21168797, -0.02469393, -0.50769134],
[-1.38382318, -1.44613351,  0.39553202],
[-1.00392836, -0.7489974 , -0.02276408],
[ 0.83296682, -2.43909498,  0.96621259],
```

$\mathbf{d}$

```
[ 1.4256218 ],
[-0.22524577],
[ 1.37469633],
[ 1.01146094],
[ 0.12305495],
[-0.48294611],
[-1.5294555 ],
```

t

```
[ 1.0548647 ,  1.61434495],
[-0.09044324, -0.85366075],
[-0.13494657, -0.17266229],
[ 0.49530346, -0.47933517],
[-0.12823182, -0.39176904],
[-1.14057797, -1.66803929],
[-0.02683053,  1.51849318],
[-1.15935135,  1.3009145 ]
```



$\mathbf{u}(t)$

$$\mathbf{x}(t) = \tanh(\mathbf{W}\,\mathbf{x}(t-1) + \mathbf{w}^{\text{in}}\,\mathbf{u}(t))$$

$$\hat{y}(t) = \mathbf{w}^{\text{out}}\,\mathbf{x}(t)$$

$\mathbf{M}$ $\qquad \mathbf{w}^{\text{out}} = \qquad \mathbf{d}$

t

```
[ 0.46144053, -0.1853432 , -1.2101936 ],
[ 0.48214552,  1.23107819, -0.09684498],
[-0.18654759,  0.37844562,  2.21832035],
[ 1.21168797, -0.02469393, -0.50769134],
[-1.38382318, -1.44613351,  0.39553202],
[-1.00392836, -0.7489974 , -0.02276408],
[ 0.83296682, -2.43909498,  0.96621259],
```

?

```
[ 1.4256218 ],
[-0.22524577],
[ 1.37469633],
[ 1.01146094],
[ 0.12305495],
[-0.48294611],
[-1.5294555 ],
[ 0.01315594]
```

"Echo state" network training:
(Jaeger & Haas, Science, 2004)

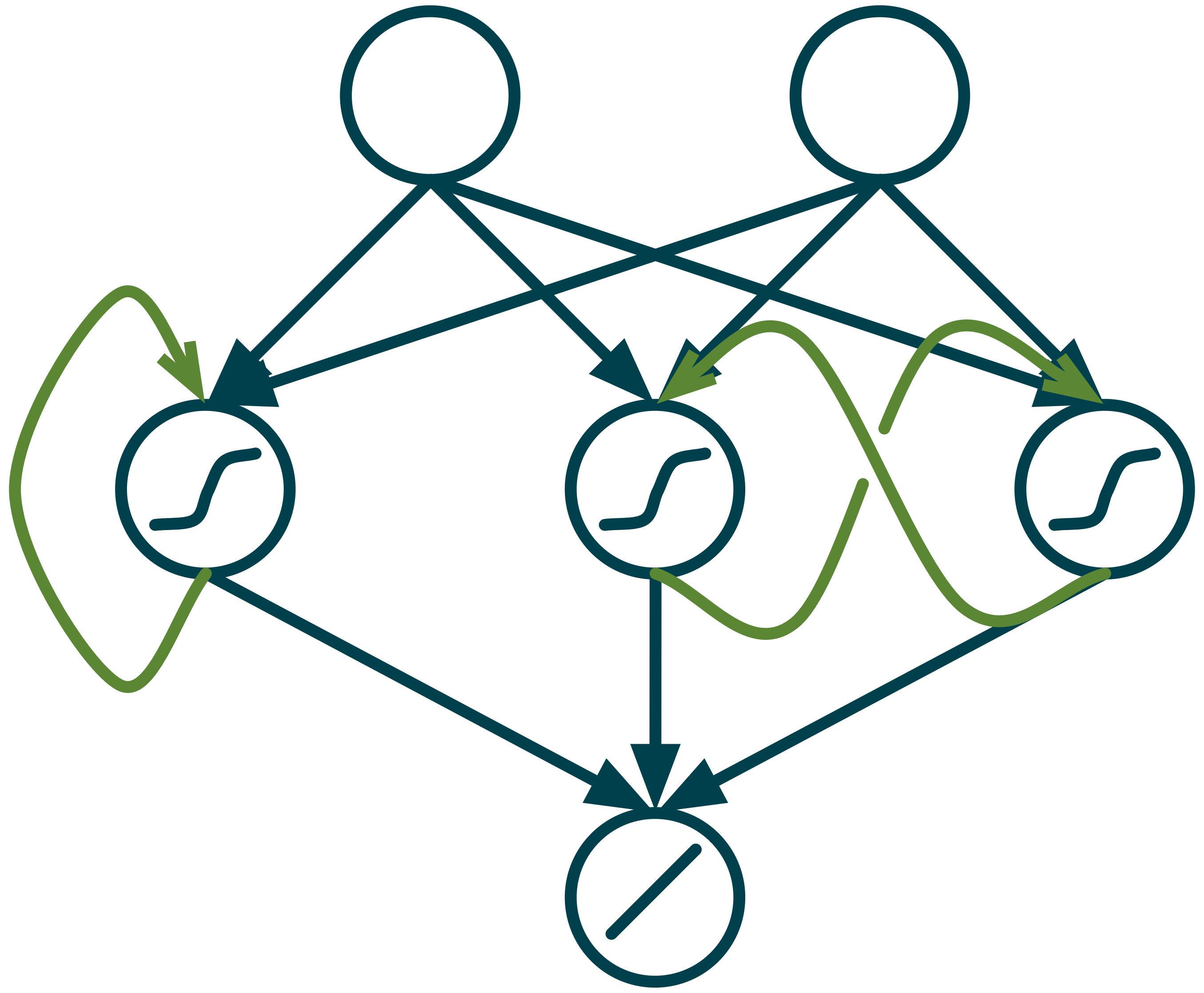
1. Create random weights $\mathbf{w}^{\text{in}}$ and $\mathbf{W}$.
2. Training data $(\mathbf{U}^{\text{train}}, \mathbf{y}^{\text{train}})$, with T steps. Run this through your network.
    a) Collect all the vectors $\mathbf{x}(t)$ in a state collection matrix $\mathbf{M}$.
    b) Collect all desired outputs in $\mathbf{d}$.
3. Compute the output weights $\mathbf{w}^{\text{out}}$:

$$w^{\text{out}} = (\mathbf{M}^{-1}\mathbf{d})^{\top}$$

Notes:
- We use the pseudo-inverse, $\mathbf{M}^{\dagger}$ because $\mathbf{M}$ is usually not square
- This is solving a *linear* regression problem, with $\mathbf{x}$ as inputs.

```
[ 1.0548647 ,  1.61434495],
[-0.09044324, -0.85366075],
[-0.13494657, -0.17266229],
[ 0.49530346, -0.47933517],
[-0.12823182, -0.39176904],
[-1.14057797, -1.66803929],
[-0.02683053,  1.51849318],
[-1.15935135,  1.3009145 ]
```

t



$\mathbf{u}(t)$

$$\mathbf{x}(t) = \tanh(\mathbf{W}\,\mathbf{x}(t-1) + \mathbf{w}^{\text{in}}\,\mathbf{u}(t))$$

$$\hat{y}(t) = \mathbf{w}^{\text{out}}\,\mathbf{x}(t)$$

$$\mathbf{w}^{\text{out}} = \Big(\mathbf{M}^{\dagger}\;\mathbf{d}\Big)^{\top}$$

```
[[0.07884279,          ( [ 0.0543568 ,  0.06158259,  0.00928494,  0.191     [ 1.4256218 ],
  0.10197564,      =     [-0.02837639,  0.07368738,  0.04573695, -0.022     [-0.22524577],
  0.02198609]]           [-0.14410255,  0.0078742 ,  0.27793245, -0.039     [ 1.37469633],
                                                                           [ 1.01146094],
                                                                           [ 0.12305495],
                                                                           [-0.48294611],
                                                                           [-1.5294555 ],
                                                                           [ 0.01315594]  )^T
```

t

"Echo state" network training:
(Jaeger & Haas, Science, 2004)

1. Create random weights $\mathbf{w}^{\text{in}}$ and $\mathbf{W}$.
2. Training data $(\mathbf{U}^{\text{train}}, \mathbf{y}^{\text{train}})$, with T steps. Run this through your network.
   a) Collect all the vectors $\mathbf{x}(t)$ in a state collection matrix $\mathbf{M}$.
   b) Collect all desired outputs in $\mathbf{d}$.
3. Compute the output weights $\mathbf{w}^{\text{out}}$:

$$w^{\text{out}} = (\mathbf{M}^{-1}\mathbf{d})^{\top}$$

Notes:
- We use the pseudo-inverse, $\mathbf{M}^{\dagger}$ because $\mathbf{M}$ is usually not square
- This is solving a *linear* regression problem, with $\mathbf{x}$ as inputs.

Echo State Networks advantages:

- easy to program
- quick training procedure
- good results for many problems

Echo State Networks advantages:

- easy to program
- quick training procedure
- good results for many problems

Some of the disadvantages:

- varying performance due to random parameters
- larger in size compared to fully trained networks, for comparable performance
- as a result, provide less insight into the problem

Echo State Networks advantages:

- easy to program
- quick training procedure
- good results for many problems

Some of the disadvantages:

- varying performance due to random parameters
- larger in size compared to fully trained networks, for comparable performance
- as a result, provide less insight into the problem

Papers / manuscripts:

- Analyzing Echo-state Networks Using Fractal Dimension, N. M. Mayer, O.Obst, IJCNN 2022. arxiv.org/abs/2205.09348
- **The Power of Linear Recurrent Neural Networks**, F. Stolzenburg et al. arxiv.org/abs/1802.03308

Linear Recurrent Neural Networks

By example

LRNN only use linear activation function, and have an initial state $\mathbf{s}$ at start time $t_0 = 0$.

Input- and output units are the same, so the network can either (a) receive input, or (b) produce output.

Linear Recurrent Neural Networks

By example

LRNN only use linear activation function, and have an initial state $\mathbf{s}$ at start time $t_0 = 0$.

Input- and output units are the same, so the network can either (a) receive input, or (b) produce output.

$$x_i(t + \tau) = w_{i1}\, x_1(t) + \ldots + w_{iN}\, x_N(t)$$

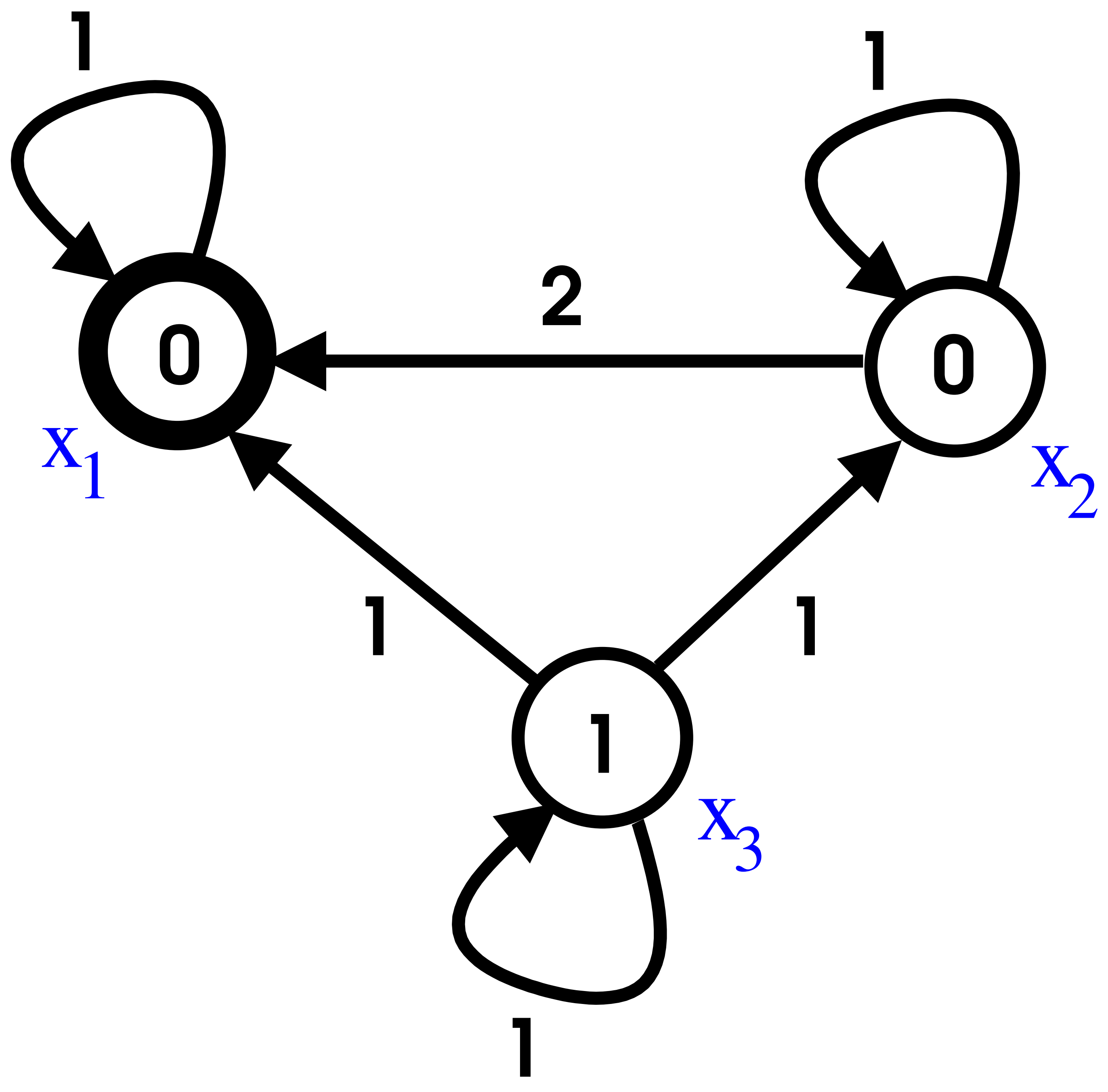
Linear Recurrent Neural Networks

By example

LRNN only use linear activation function, and have an initial state $\mathbf{s}$ at start time $t_0 = 0$.

Input- and output units are the same, so the network can either (a) receive input, or (b) produce output.

$$x_i(t + \tau) = w_{i1}\, x_1(t) + \ldots + w_{iN}\, x_N(t)$$

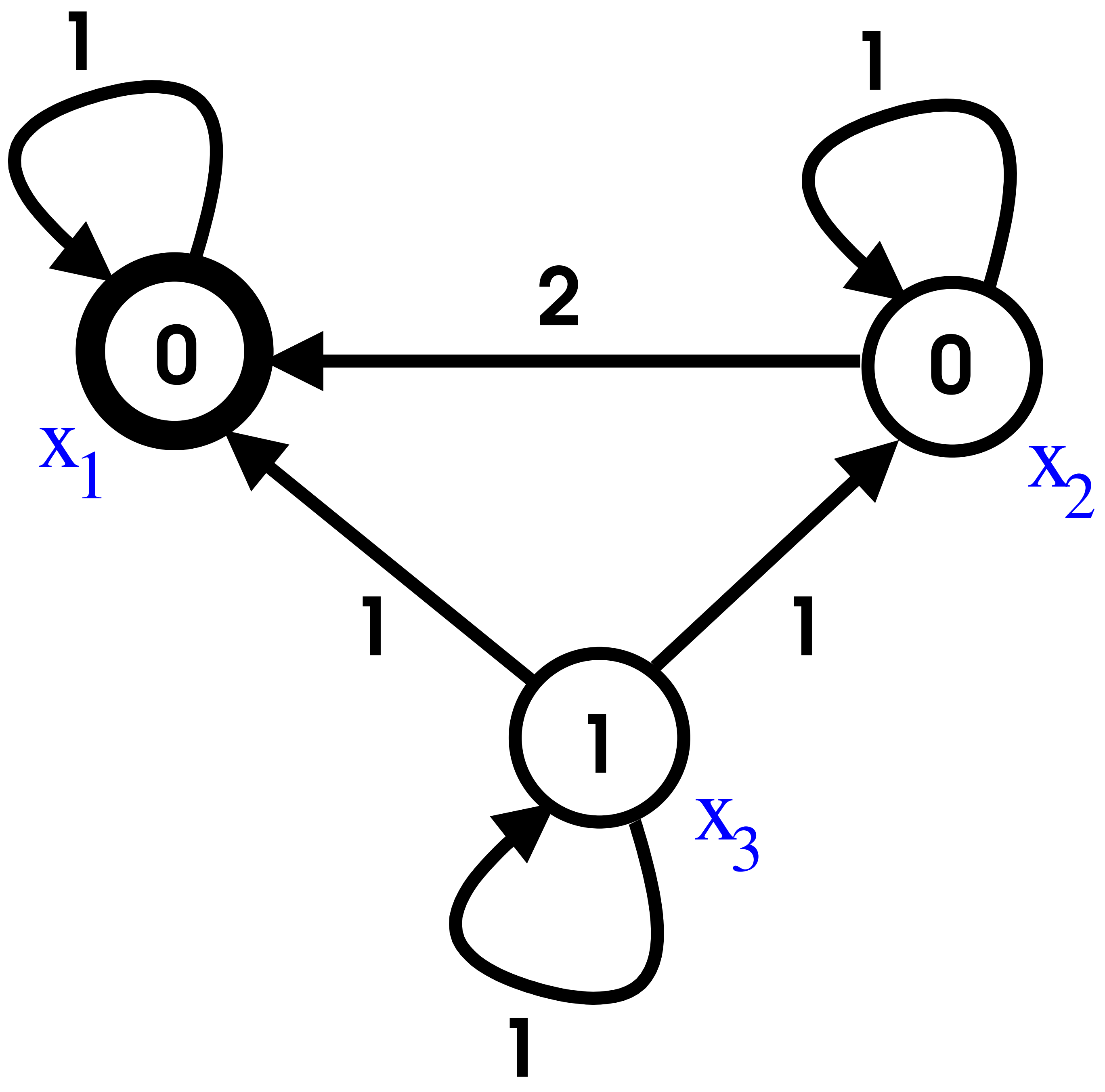


Linear Recurrent Neural Networks

By example

LRNN only use linear activation function, and have an initial state $\mathbf{s}$ at start time $t_0 = 0$.

Input- and output units are the same, so the network can either (a) receive input, or (b) produce output.



$$x_i(t+\tau) = w_{i1}\,x_1(t) + \ldots + w_{iN}\,x_N(t)$$

$\mathbf{s} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$ is an initial state (the start vector), $\tau = 1$, and

$\mathbf{W} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$ is a transition matrix.

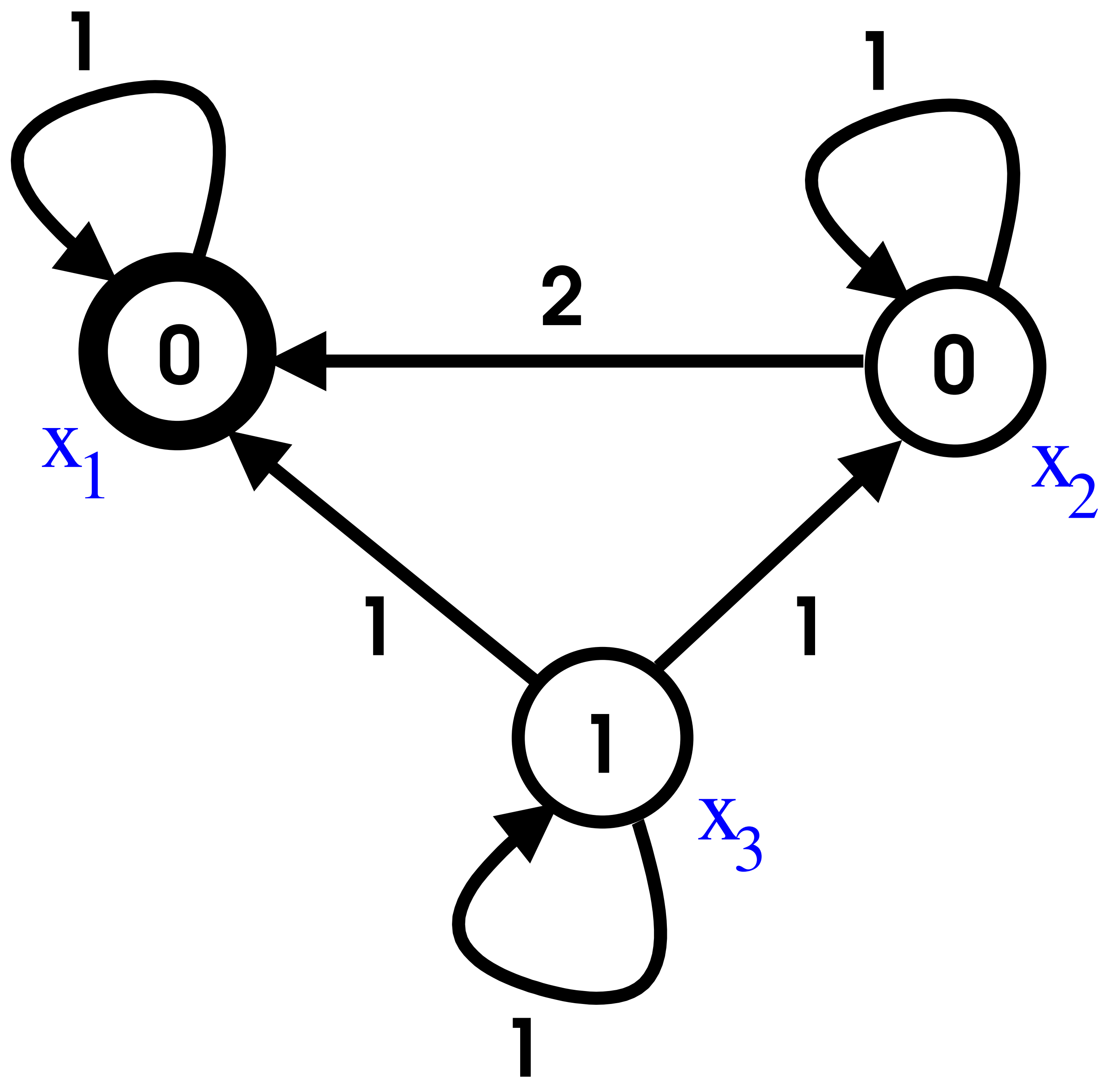
The network then computes $f(t) = t^2$.

Linear Recurrent Neural Networks

By example

LRNN only use linear activation function, and have an initial state $\mathbf{s}$ at start time $t_0 = 0$.

Input- and output units are the same, so the network can either (a) receive input, or (b) produce output.



$$x_i(t + \tau) = w_{i1}\, x_1(t) + \ldots + w_{iN}\, x_N(t)$$

$$\mathbf{s} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$ is an initial state (the start vector), $\tau = 1$, and

$$\mathbf{W} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$ is a transition matrix.

$$\mathbf{W} = \begin{bmatrix} & \mathbf{W}^{\text{out}} & \\ \mathbf{w}^{\text{in}} & & \mathbf{W}^{\text{res}} \end{bmatrix}$$

The network then computes $f(t) = t^2$.

$$f(t) = \begin{cases} \mathbf{s}, & t = 0 \\ \mathbf{W} \cdot f(t-1), & \text{otherwise} \end{cases} \quad \Rightarrow \quad f(t) = \mathbf{W}^t \cdot \mathbf{s}$$

$$f(t) = \begin{cases} \mathbf{s}, & t = 0 \\ \mathbf{W} \cdot f(t-1), & \text{otherwise} \end{cases} \qquad \Longrightarrow \qquad f(t) = \mathbf{W}^t \cdot \mathbf{s}$$

Let $\mathbf{W} = \mathbf{V} \cdot \mathbf{J} \cdot \mathbf{V}^{-1}$ be the Jordan decomposition of our transition matrix.

$$f(t) = \begin{cases} \mathbf{s}, & t = 0 \\ \mathbf{W} \cdot f(t-1), & \text{otherwise} \end{cases} \quad \Rightarrow \quad f(t) = \mathbf{W}^t \cdot \mathbf{s}$$

Let $\mathbf{W} = \mathbf{V} \cdot \mathbf{J} \cdot \mathbf{V}^{-1}$ be the Jordan decomposition of our transition matrix.

$\mathbf{J}$ is the direct sum of (one or more) Jordan blocks (ie a block diagonal matrix).

$$\mathbf{J}_m(\lambda) = \begin{bmatrix} \lambda & 1 & 0 & \dots & 0 \\ 0 & \lambda & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \lambda & 1 \\ 0 & \dots & \dots & 0 & \lambda \end{bmatrix}$$

in general, with different sizes m and different eigenvalues λ

$$f(t) = \begin{cases} \mathbf{s}, & t = 0 \\ \mathbf{W} \cdot f(t-1), & \text{otherwise} \end{cases} \quad \Rightarrow \quad f(t) = \mathbf{W}^t \cdot \mathbf{s}$$

Let $\mathbf{W} = \mathbf{V} \cdot \mathbf{J} \cdot \mathbf{V}^{-1}$ be the Jordan decomposition of our transition matrix.

$\mathbf{J}$ is the direct sum of (one or more) Jordan blocks (ie a block diagonal matrix).

$$\mathbf{J}_m(\lambda) = \begin{bmatrix} \lambda & 1 & 0 & \dots & 0 \\ 0 & \lambda & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \lambda & 1 \\ 0 & \dots & \dots & 0 & \lambda \end{bmatrix}$$

in general, with different sizes m and different eigenvalues λ

$$\mathbf{J} = \begin{bmatrix} \boxed{\begin{matrix} \lambda_1 & 1 & \\ & \lambda_1 & 1 \\ & & \lambda_1 \end{matrix}} & & & & \\ & \boxed{\begin{matrix} \lambda_2 & 1 \\ & \lambda_2 \end{matrix}} & & & \\ & & \boxed{\lambda_3} & & \\ & & & \ddots & \\ & & & & \boxed{\begin{matrix} \lambda_n & 1 \\ & \lambda_n \end{matrix}} \end{bmatrix}$$

$$f(t) = \begin{cases} \mathbf{s}, & t = 0 \\ \mathbf{W} \cdot f(t-1), & \text{otherwise} \end{cases} \quad \Rightarrow \quad f(t) = \mathbf{W}^t \cdot \mathbf{s}$$

Let $\mathbf{W} = \mathbf{V} \cdot \mathbf{J} \cdot \mathbf{V}^{-1}$ be the Jordan decomposition of our transition matrix.

$\mathbf{J}$ is the direct sum of (one or more) Jordan blocks (ie a block diagonal matrix).

$$\mathbf{J}_m(\lambda) = \begin{bmatrix} \lambda & 1 & 0 & \dots & 0 \\ 0 & \lambda & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \lambda & 1 \\ 0 & \dots & \dots & 0 & \lambda \end{bmatrix}$$

in general, with different sizes m and different eigenvalues λ

Then: $f(t) = \mathbf{W}^t \cdot \mathbf{s} = \mathbf{V} \cdot \mathbf{J}^t \cdot \mathbf{V}^{-1} \cdot \mathbf{s}$

$$\mathbf{J} = \begin{bmatrix} \boxed{\begin{matrix} \lambda_1 & 1 & \\ & \lambda_1 & 1 \\ & & \lambda_1 \end{matrix}} & & & & \\ & \boxed{\begin{matrix} \lambda_2 & 1 \\ & \lambda_2 \end{matrix}} & & & \\ & & \boxed{\lambda_3} & & \\ & & & \ddots & \\ & & & & \boxed{\begin{matrix} \lambda_n & 1 \\ & \lambda_n \end{matrix}} \end{bmatrix}$$

$$f(t) = \mathbf{W}^t \cdot \mathbf{s} = \mathbf{V} \cdot \mathbf{J}^t \cdot \mathbf{V}^{-1} \cdot \mathbf{s}$$

$$\mathbf{J} = \begin{bmatrix} \boxed{\begin{matrix} \lambda_1 & 1 & \\ & \lambda_1 & 1 \\ & & \lambda_1 \end{matrix}} & & & & \\ & \boxed{\begin{matrix} \lambda_2 & 1 \\ & \lambda_2 \end{matrix}} & & & \\ & & \boxed{\lambda_3} & & \\ & & & \ddots & \\ & & & & \boxed{\begin{matrix} \lambda_n & 1 \\ & \lambda_n \end{matrix}} \end{bmatrix}$$

$$f(t) = \mathbf{W}^t \cdot \mathbf{s} = \underline{\mathbf{V}} \cdot \mathbf{J}^t \cdot \mathbf{V}^{-1} \cdot \mathbf{s}$$

decompose into matrices
$\mathbf{v}$ of size $N \times m$

$$\mathbf{J} = \begin{bmatrix} \boxed{\begin{matrix} \lambda_1 & 1 & \\ & \lambda_1 & 1 \\ & & \lambda_1 \end{matrix}} & & & & \\ & \boxed{\begin{matrix} \lambda_2 & 1 \\ & \lambda_2 \end{matrix}} & & & \\ & & \boxed{\lambda_3} & & \\ & & & \ddots & \\ & & & & \boxed{\begin{matrix} \lambda_n & 1 \\ & \lambda_n \end{matrix}} \end{bmatrix}$$

$$f(t) = \mathbf{W}^t \cdot \mathbf{s} = \underbrace{\mathbf{V}}_{\substack{\text{decompose into matrices} \\ \mathbf{v} \text{ of size } N \times m}} \cdot \mathbf{J}^t \cdot \underbrace{\mathbf{V}^{-1} \cdot \mathbf{s}}_{\substack{\text{decompose into stack of} \\ \text{column vectors } \mathbf{w} \text{ of size } m}}$$

$$\mathbf{J} = \begin{bmatrix} \boxed{\begin{matrix} \lambda_1 & 1 & \\ & \lambda_1 & 1 \\ & & \lambda_1 \end{matrix}} & & & & \\ & \boxed{\begin{matrix} \lambda_2 & 1 \\ & \lambda_2 \end{matrix}} & & & \\ & & \boxed{\lambda_3} & & \\ & & & \ddots & \\ & & & & \boxed{\begin{matrix} \lambda_n & 1 \\ & \lambda_n \end{matrix}} \end{bmatrix}$$

$$f(t) = \mathbf{W}^t \cdot \mathbf{s} = \underline{\mathbf{V}} \cdot \mathbf{J}^t \cdot \underline{\mathbf{V}^{-1} \cdot \mathbf{s}}$$

decompose into matrices
$\mathbf{v}$ of size $N \times m$

decompose into stack of
column vectors $\mathbf{w}$ of size m

corresponding to the blocks in $\mathbf{J}$

$$\mathbf{J} = \begin{bmatrix} \boxed{\begin{matrix} \lambda_1 & 1 & \\ & \lambda_1 & 1 \\ & & \lambda_1 \end{matrix}} & & & & \\ & \boxed{\begin{matrix} \lambda_2 & 1 \\ & \lambda_2 \end{matrix}} & & & \\ & & \boxed{\lambda_3} & & \\ & & & \ddots & \\ & & & & \boxed{\begin{matrix} \lambda_n & 1 \\ & \lambda_n \end{matrix}} \end{bmatrix}$$

$$f(t) = \mathbf{W}^t \cdot \mathbf{s} = \underline{\mathbf{V}} \cdot \mathbf{J}^t \cdot \underline{\mathbf{V}^{-1} \cdot \mathbf{s}}$$

decompose into matrices
$\mathbf{v}$ of size $N \times m$

decompose into stack of
column vectors $\mathbf{w}$ of size m

corresponding to the blocks in $\mathbf{J}$

$$\mathbf{J} = \begin{bmatrix} \lambda_1 & 1 & & & & & & & \\ & \lambda_1 & 1 & & & & & & \\ & & \lambda_1 & & & & & & \\ & & & \lambda_2 & 1 & & & & \\ & & & & \lambda_2 & & & & \\ & & & & & \lambda_3 & & & \\ & & & & & & \ddots & & \\ & & & & & & & \lambda_n & 1 \\ & & & & & & & & \lambda_n \end{bmatrix}$$

then we can express $f(t)$ as a sum of vectors $\mathbf{u} = \mathbf{v} \cdot \mathbf{J}_m(\lambda)^t \cdot \mathbf{w}$.

$$f(t) = \mathbf{W}^t \cdot \mathbf{s} = \underline{\mathbf{V}} \cdot \mathbf{J}^t \cdot \underline{\mathbf{V}^{-1} \cdot \mathbf{s}}$$

decompose into matrices $\mathbf{v}$ of size $N \times m$

decompose into stack of column vectors $\mathbf{w}$ of size m

corresponding to the blocks in $\mathbf{J}$

$$\mathbf{J} = \begin{bmatrix} \boxed{\begin{matrix} \lambda_1 & 1 & \\ & \lambda_1 & 1 \\ & & \lambda_1 \end{matrix}} & & & & \\ & \boxed{\begin{matrix} \lambda_2 & 1 \\ & \lambda_2 \end{matrix}} & & & \\ & & \boxed{\lambda_3} & & \\ & & & \ddots & \\ & & & & \boxed{\begin{matrix} \lambda_n & 1 \\ & \lambda_n \end{matrix}} \end{bmatrix}$$

then we can express $f(t)$ as a sum of vectors $\mathbf{u} = \mathbf{v} \cdot \mathbf{J}_m(\lambda)^t \cdot \mathbf{w}$.

The Jordan block powers $\mathbf{J}_m(\lambda)^t$ are upper triangular Toeplitz matrices:

$$\left(\mathbf{J}_m(\lambda)^t\right)_{ij} = \binom{t}{j-i} \lambda^{t-(j-i)}$$

$$\begin{bmatrix} a & b & c & d & e \\ f & a & b & c & d \\ g & f & a & b & c \\ h & g & f & a & b \\ i & h & g & f & a \end{bmatrix}$$

$$f(t) = \mathbf{W}^t \cdot \mathbf{s} = \underline{\mathbf{V}} \cdot \mathbf{J}^t \cdot \underline{\mathbf{V}^{-1} \cdot \mathbf{s}}$$

decompose into matrices
$\mathbf{v}$ of size $N \times m$

decompose into stack of
column vectors $\mathbf{w}$ of size m

corresponding to the blocks in $\mathbf{J}$

$$\mathbf{J} = \begin{bmatrix} \boxed{\begin{matrix} \lambda_1 & 1 & \\ & \lambda_1 & 1 \\ & & \lambda_1 \end{matrix}} & & & & \\ & \boxed{\begin{matrix} \lambda_2 & 1 \\ & \lambda_2 \end{matrix}} & & & \\ & & \boxed{\lambda_3} & & \\ & & & \ddots & \\ & & & & \boxed{\begin{matrix} \lambda_n & 1 \\ & \lambda_n \end{matrix}} \end{bmatrix}$$

then we can express $f(t)$ as a sum of vectors $\mathbf{u} = \mathbf{v} \cdot \mathbf{J}_m(\lambda)^t \cdot \mathbf{w}$.

The Jordan block powers $\mathbf{J}_m(\lambda)^t$ are upper triangular Toeplitz matrices:

$$\left(\mathbf{J}_m(\lambda)^t\right)_{ij} = \binom{t}{j-i} \lambda^{t-(j-i)}$$

$$\begin{bmatrix} a & b & c & d & e \\ f & a & b & c & d \\ g & f & a & b & c \\ h & g & f & a & b \\ i & h & g & f & a \end{bmatrix}$$

Also: For all column vectors $\mathbf{y}$ of size N with all non-zero entries, there exists a square matrix $\mathbf{A}$ of size $N \times N$ such that for all $t \geq 0$:

$$f(t) = \mathbf{W}^t \cdot s = \mathbf{A} \cdot \mathbf{J}^t \cdot \mathbf{y}$$

Learning LRNNs

Step 1/2

$$\mathbf{W} = \begin{bmatrix} & \mathbf{W}^{\text{out}} & \\ \mathbf{W}^{\text{in}} & & \mathbf{W}^{\text{res}} \end{bmatrix}$$

“Echo state” training, for autoregressive prediction of next input value (i.e., predict $S(t+\tau)$ at time t).

Learning LRNNs

Step 1/2

$$\mathbf{W} = \begin{bmatrix} & \mathbf{w}^{\text{out}} & \\ \mathbf{w}^{\text{in}} & & \mathbf{W}^{\text{res}} \end{bmatrix}$$

"Echo state" training, for autoregressive prediction of next input value (i.e., predict $S(t+\tau)$ at time t).

1. Create random weights $\mathbf{w}^{\text{in}}$ and $\mathbf{W}^{\text{res}}$.
   Scale $\mathbf{W}^{\text{res}}$ so the largest eigenvalue is 1.0.

Learning LRNNs

Step 1/2

$$\mathbf{W} = \begin{bmatrix} & \mathbf{w}^{\text{out}} & \\ \mathbf{w}^{\text{in}} & & \mathbf{W}^{\text{res}} \end{bmatrix}$$

"Echo state" training, for autoregressive prediction of next input value (i.e., predict $S(t+\tau)$ at time t).

1. Create random weights $\mathbf{w}^{\text{in}}$ and $\mathbf{W}^{\text{res}}$.
   Scale $\mathbf{W}^{\text{res}}$ so the largest eigenvalue is 1.0.
2. Run training data through your network.

   a) Collect all the vectors $\mathbf{x}(t)$ in a state collection matrix $\mathbf{M}$.

   b) Collect all desired outputs in $\mathbf{d}$.

Learning LRNNs

Step 1/2

$$\mathbf{W} = \begin{bmatrix} & \mathbf{w}^{\text{out}} \\ \mathbf{w}^{\text{in}} & \mathbf{W}^{\text{res}} \end{bmatrix}$$

"Echo state" training, for autoregressive prediction of next input value (i.e., predict $S(t+\tau)$ at time t).

1. Create random weights $\mathbf{w}^{\text{in}}$ and $\mathbf{W}^{\text{res}}$.
   Scale $\mathbf{W}^{\text{res}}$ so the largest eigenvalue is 1.0.
2. Run training data through your network.

   a) Collect all the vectors $\mathbf{x}(t)$ in a state collection matrix $\mathbf{M}$.

   b) Collect all desired outputs in $\mathbf{d}$.
3. Compute the output weights $\mathbf{w}^{\text{out}}$:

$$w^{\text{out}} = (\mathbf{M}^{-1}\mathbf{d})^{\top}$$

Learning LRNNs

Step 1/2

$$\mathbf{W} = \begin{bmatrix} & \mathbf{w}^{\text{out}} & \\ \mathbf{w}^{\text{in}} & & \mathbf{W}^{\text{res}} \end{bmatrix}$$

"Echo state" training, for autoregressive prediction of next input value (i.e., predict $S(t+\tau)$ at time t).

1. Create random weights $\mathbf{w}^{\text{in}}$ and $\mathbf{W}^{\text{res}}$.
   Scale $\mathbf{W}^{\text{res}}$ so the largest eigenvalue is 1.0.
2. Run training data through your network.
   a) Collect all the vectors $\mathbf{x}(t)$ in a state collection matrix $\mathbf{M}$.
   b) Collect all desired outputs in $\mathbf{d}$.
3. Compute the output weights $\mathbf{w}^{\text{out}}$:

$$w^{\text{out}} = (\mathbf{M}^{-1}\mathbf{d})^{\mathsf{T}}$$

Because the neurons are linear:

- $\mathbf{w}^{\text{in}}$ and $\mathbf{W}^{\text{res}}$ cannot be "improved", for the training data.
- The spectral radius of $\mathbf{W}^{\text{res}}$ is set to 1.0, so values do not explode or vanish as t increases.

Note:
- the network will be fully connected (using, e.g., uniform random weights)
- the network architecture is changed in step 2

Learning LRNNs

Step 2/2

We can reduce the size, to improve generalisation and avoid overfitting.

$$f(t) = \mathbf{W}^t \cdot s = \mathbf{A} \cdot \mathbf{J}^t \cdot \mathbf{y}$$

$f(t)$ also is a sum of vectors $\mathbf{u} = \mathbf{v} \cdot \mathbf{J}_m(\lambda)^t \cdot \mathbf{w}$ (where $\mathbf{w}$ is constant)

For large t, the contribution of a Jordan component vanishes – if $\|v\| \approx 0$ and/or $|\lambda| \ll 1$.

Learning LRNNs

Step 2/2

We can reduce the size, to improve generalisation and avoid overfitting.

$$f(t) = \mathbf{W}^t \cdot s = \mathbf{A} \cdot \mathbf{J}^t \cdot \mathbf{y}$$

$f(t)$ also is a sum of vectors $\mathbf{u} = \mathbf{v} \cdot \mathbf{J}_m(\lambda)^t \cdot \mathbf{w}$ (where $\mathbf{w}$ is constant)

For large t, the contribution of a Jordan component vanishes – if $\|v\| \approx 0$ and/or $|\lambda| \ll 1$.

Find all Jordan components causing only small errors (RMSE). All others are considered *relevant*.

Derive reduced matrices $\mathbf{A}', \mathbf{J}', \mathbf{y}'$:

- From $\mathbf{A}$, take rows corresponding to input/output components,
  as well as columns corresponding to relevant network components.
- From $\mathbf{J}$, take rows and columns corresponding to relevant network components
- From $\mathbf{y}$, take rows corresponding to relevant network components

Results

(Learning LRNN)

A 2-step procedure to train LRNN,
for time series prediction tasks:

- Train output weights in a single step
- Create new network architecture in one step

% d-dimensional function, given sampled, as time series
$S = [f(0) \ldots f(n)]$

% random initialization of reservoir and input weights
$W^{\text{in}} = \text{randn}(N, d)$
$W^{\text{res}} = \text{randn}(N^{\text{res}}, N^{\text{res}})$

% learn output weights by linear regression
$X = \left[W^t \cdot s\right]_{t=0,\ldots,n}$
$Y^{\text{out}} = \left[S(1) \ \cdots \ S(n)\right]$
$W^{\text{out}} = Y^{\text{out}} / X$

% transition matrix and its decomposition
$$W = \begin{bmatrix} & W^{\text{out}} \\ W^{\text{in}} & W^{\text{res}} \end{bmatrix}$$
$J = \text{jordan\_matrix}(W)$

% network size reduction
$y = \left[1 \cdots 1\right]^{\top}$
$Y = \left[J^t \cdot y\right]_{t=0,\ldots,n}$
$A = X/Y$ with rows restricted to input/output dimensions
reduce(A, J, y) to relevant components
such that RMSE$(S,\text{Out}) < \theta$

Results

(Learning LRNN)

A 2-step procedure to train LRNN,
for time series prediction tasks:

- Train output weights in a single step
- Create new network architecture in one step

Step 2 reduces the number of neurons.

% d-dimensional function, given sampled, as time series
$S = [f(0) \dots f(n)]$

% random initialization of reservoir and input weights
$W^{\text{in}} = \text{randn}(N, d)$
$W^{\text{res}} = \text{randn}(N^{\text{res}}, N^{\text{res}})$

% learn output weights by linear regression
$X = \left[W^t \cdot s\right]_{t=0,\dots,n}$
$Y^{\text{out}} = \left[S(1) \ \cdots \ S(n)\right]$
$W^{\text{out}} = Y^{\text{out}} / X$

% transition matrix and its decomposition
$$W = \begin{bmatrix} & W^{\text{out}} \\ W^{\text{in}} & W^{\text{res}} \end{bmatrix}$$
$J = \text{jordan\_matrix}(W)$

% network size reduction
$y = \left[1 \cdots 1\right]^{\top}$
$Y = \left[J^t \cdot y\right]_{t=0,\dots,n}$
$A = X/Y$ with rows restricted to input/output dimensions
reduce(A, J, y) to relevant components
such that RMSE$(S,\text{Out}) < \theta$

Results

(Learning LRNN)

A 2-step procedure to train LRNN,
for time series prediction tasks:

- Train output weights in a single step
- Create new network architecture in one step

Step 2 reduces the number of neurons.

The reduced matrix is sparse: non-zero only in the main and adjacent diagonals, leading to a number of connections in $O(N)$ i.e., linear in the number of units.

% d-dimensional function, given sampled, as time series
$\mathrm{S} = [f(0) \dots f(n)]$

% random initialization of reservoir and input weights
$W^{\mathrm{in}} = \mathrm{randn}(N, d)$
$W^{\mathrm{res}} = \mathrm{randn}(N^{\mathrm{res}}, N^{\mathrm{res}})$

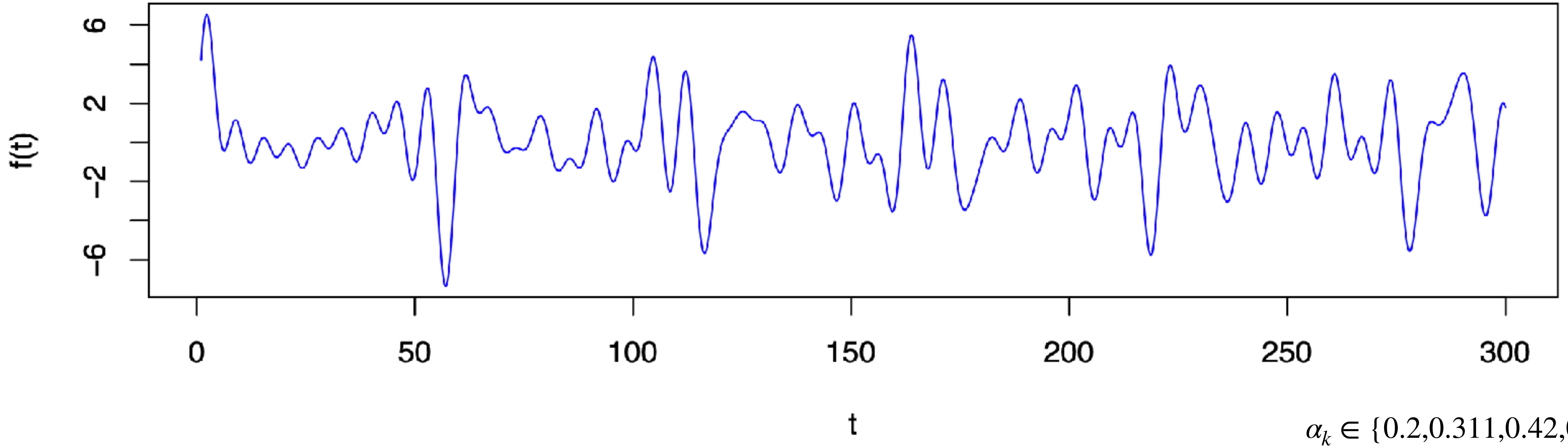
% learn output weights by linear regression
$X = \left[W^t \cdot s\right]_{t=0,\dots,n}$
$Y^{\mathrm{out}} = \left[S(1) \ \cdots \ S(n)\right]$
$W^{\mathrm{out}} = Y^{\mathrm{out}}/X$

% transition matrix and its decomposition
$$W = \begin{bmatrix} & W^{\mathrm{out}} \\ W^{\mathrm{in}} & W^{\mathrm{res}} \end{bmatrix}$$
$J = \mathrm{jordan\_matrix}(W)$

% network size reduction
$y = \left[1 \cdots 1\right]^\top$
$Y = \left[J^t \cdot y\right]_{t=0,\dots,n}$
$A = X/Y$ with rows restricted to input/output dimensions
$\mathrm{reduce}(A, J, y)$ to relevant components
such that $\mathrm{RMSE}(S, \mathrm{Out}) < \theta$

Examples

Multiple superimposed oscillators



$$S(t) = \sum_{k=1}^{K} \sin(\alpha_k t)$$

$\alpha_k \in \{0.2, 0.311, 0.42, 0.51, 0.63, 0.74, 0.85, 0.97\}$

LRNNs learn to represent the signal
with a minimal number of neurons: $N = 16$ (after step 2), better than previous SOA.

(2 are needed for each frequency, so 16 is the minimum for 8 superimposed signals)

Examples

Number puzzles

$$S_8 = [28, 33, 31, 36, 34, 39, 37, 42] \quad f(t) = f(t-2) + 3$$

$$S_9 = [3, 6, 12, 24, 48, 96, 192, 384] \quad f(t) = 2f(t-1)$$

$$S_{15} = [6, 9, 18, 21, 42, 45, 90,$$

$$S_{19} = [8, 12, 16, 20, 24, 28, 32,$$

Examples

Number puzzles

$$
\begin{aligned}
S_8 &= [28, 33, 31, 36, 34, 39, 37, 42] & f(t) &= f(t-2) + 3 \\
S_9 &= [3, 6, 12, 24, 48, 96, 192, 384] & f(t) &= 2f(t-1) \\
S_{15} &= [6, 9, 18, 21, 42, 45, 90, 93] & f(t) &= 2f(t-2) + 4.5 + 1.5(-1)^{t-1} \\
S_{19} &= [8, 12, 16, 20, 24, 28, 32,
\end{aligned}
$$

Examples

Number puzzles

$$S_8 = [28, 33, 31, 36, 34, 39, 37, 42] \quad f(t) = f(t-2) + 3$$

$$S_9 = [3, 6, 12, 24, 48, 96, 192, 384] \quad f(t) = 2f(t-1)$$

$$S_{15} = [6, 9, 18, 21, 42, 45, 90, 93] \quad f(t) = 2f(t-2) + 4.5 + 1.5(-1)^{t-1}$$

$$S_{19} = [8, 12, 16, 20, 24, 28, 32, 36] \quad f(t) = f(t-1) + 4$$

Examples

Number puzzles

$$S_8 = [28, 33, 31, 36, 34, 39, 37, 42] \quad f(t) = f(t-2) + 3$$
$$S_9 = [3, 6, 12, 24, 48, 96, 192, 384] \quad f(t) = 2f(t-1)$$
$$S_{15} = [6, 9, 18, 21, 42, 45, 90, 93] \quad f(t) = 2f(t-2) + 4.5 + 1.5(-1)^{t-1}$$
$$S_{19} = [8, 12, 16, 20, 24, 28, 32, 36] \quad f(t) = f(t-1) + 4$$

With 2 inputs ($S(t)$ and $S(t-1)$):

LRNN trained on the first 7 inputs can successfully predict the next value. The most frequently predicted last element is correct for 19 out of 20 series (experiments: 1000 runs / series).

Examples

Number puzzles

$$
\begin{aligned}
S_8 &= [28, 33, 31, 36, 34, 39, 37, 42] & f(t) &= f(t-2) + 3 \\
S_9 &= [3, 6, 12, 24, 48, 96, 192, 384] & f(t) &= 2f(t-1) \\
S_{15} &= [6, 9, 18, 21, 42, 45, 90, 93] & f(t) &= 2f(t-2) + 4.5 + 1.5(-1)^{t-1} \\
S_{19} &= [8, 12, 16, 20, 24, 28, 32, 36] & f(t) &= f(t-1) + 4
\end{aligned}
$$

With 2 inputs ($S(t)$ and $S(t-1)$):

LRNN trained on the first 7 inputs can successfully predict the next value. The most frequently predicted last element is correct for 19 out of 20 series (experiments: 1000 runs / series).

For 11/20, the prediction is correct in $\geq 99.7\ \%$ of the runs.

Examples

Number puzzles

$$
\begin{aligned}
S_8 &= [28, 33, 31, 36, 34, 39, 37, 42] & f(t) &= f(t-2) + 3 \\
S_9 &= [3, 6, 12, 24, 48, 96, 192, 384] & f(t) &= 2f(t-1) \\
S_{15} &= [6, 9, 18, 21, 42, 45, 90, 93] & f(t) &= 2f(t-2) + 4.5 + 1.5(-1)^{t-1} \\
S_{19} &= [8, 12, 16, 20, 24, 28, 32, 36] & f(t) &= f(t-1) + 4
\end{aligned}
$$

With 2 inputs ($S(t)$ and $S(t-1)$):

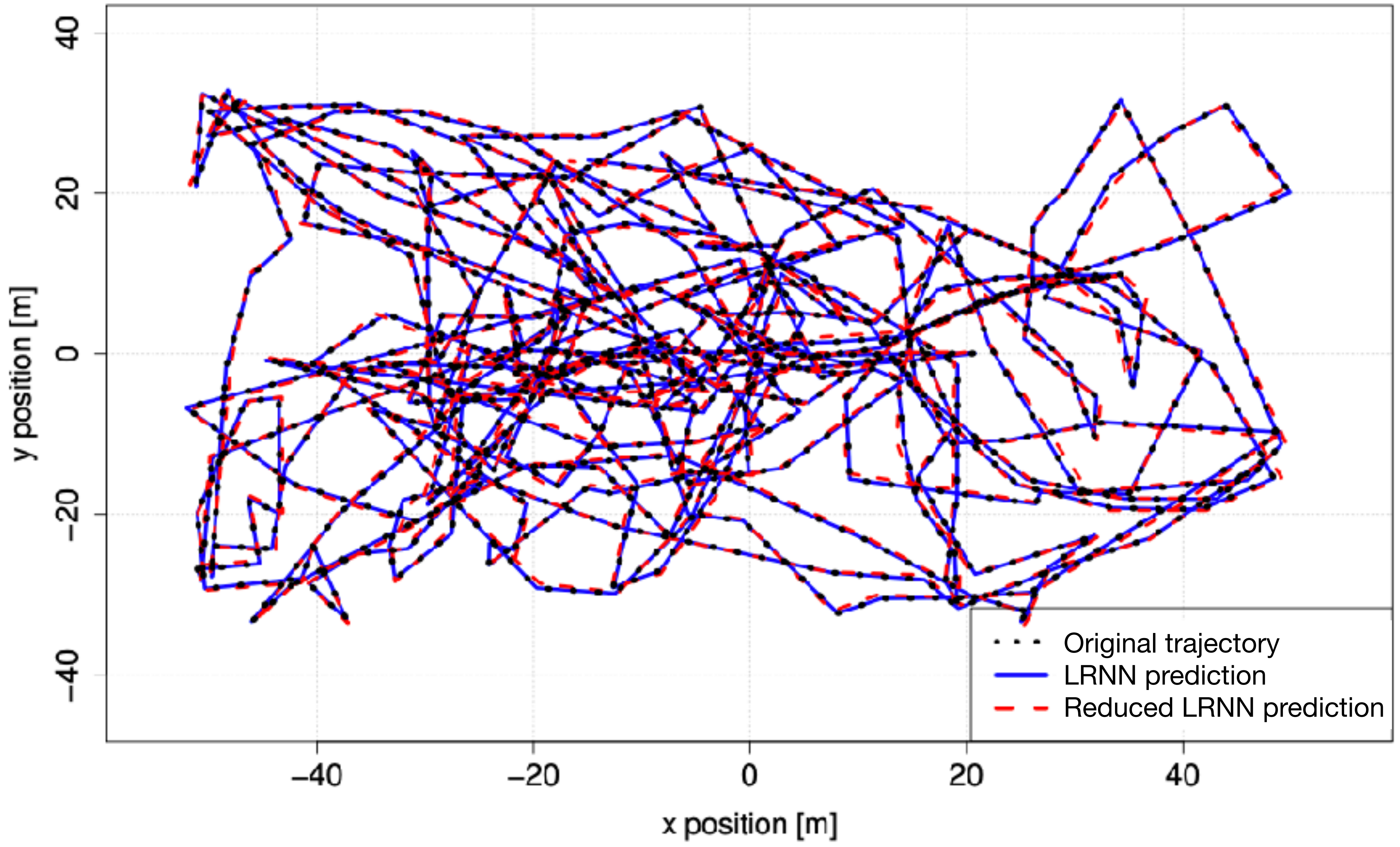
LRNN trained on the first 7 inputs can successfully predict the next value. The most frequently predicted last element is correct for 19 out of 20 series (experiments: 1000 runs / series).

For 11/20, the prediction is correct in $\geq 99.7\ \%$ of the runs.

Series like S_8 or S_{15} are some of the more difficult ones.

Example

Data compression



Using LRNN to store trajectories of a (simulated) soccer game, 6000 time steps.

- 500 reservoir units initially, almost perfect
- reduced to 340-400 units, still very small error

Training time less than 1 min, standard hardware (CPU), implementation in Octave (Matlab)

Linear Recurrent Neural Networks

- Computationally efficient approach: $O(N^3)$ (compared to NP-hard for backprop).
- Closed-form approach to network size reduction (architecture learning)
- No iterative procedure during any stage of training
- Creates small-size (down to minimal size) neural networks

Drawbacks:

- some functions cannot be realised using LRNN, e.g.,

  functions that grow faster than single exponential, like $f(t) = t\,!$ or $f(t) = 2^{2^t}$.
- Implementation becomes numerically unstable with very large networks.