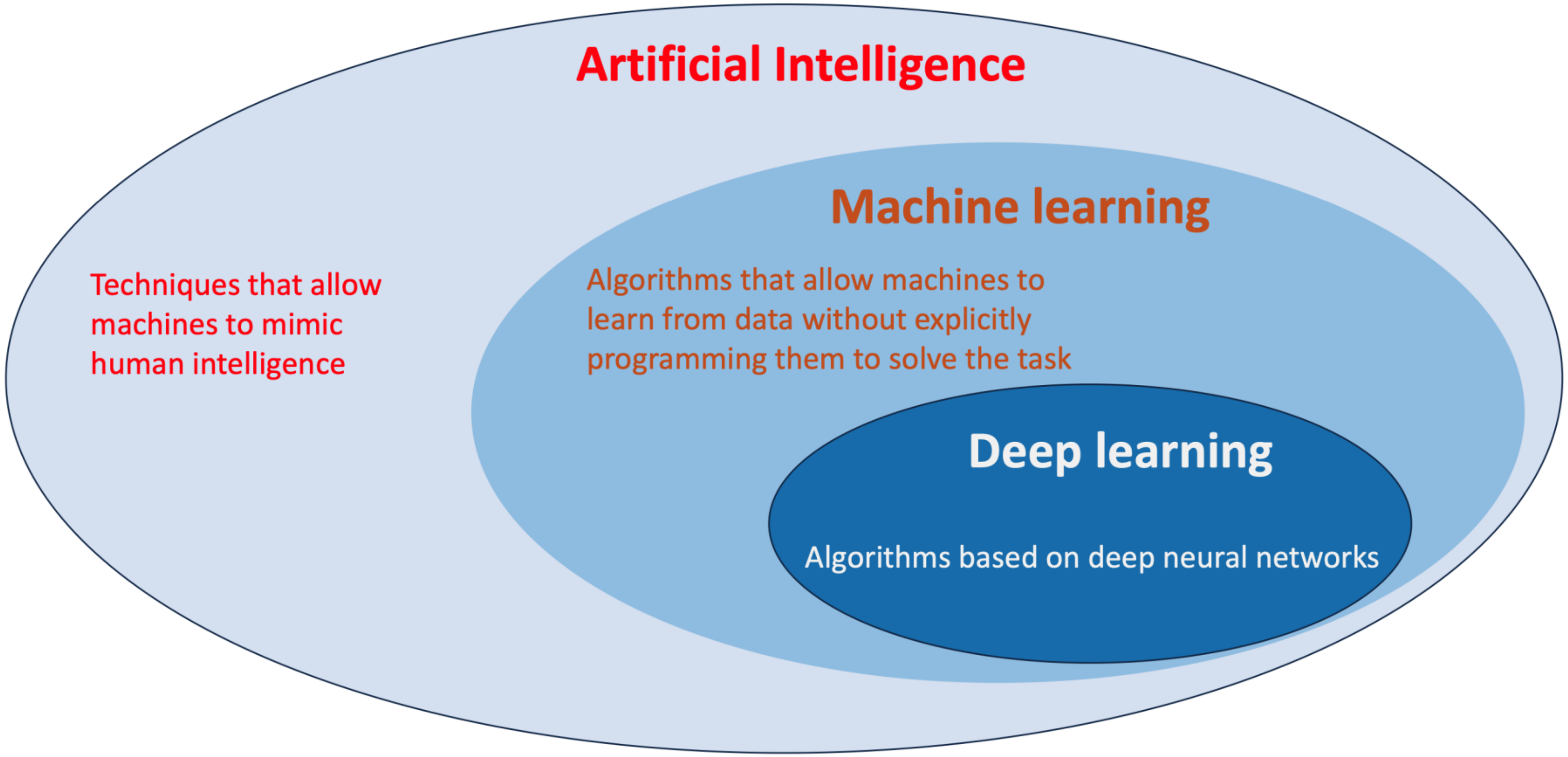
Deep Learning

Artificial Intelligence, Machine Learning, Deep Learning



Machine Learning: Regression

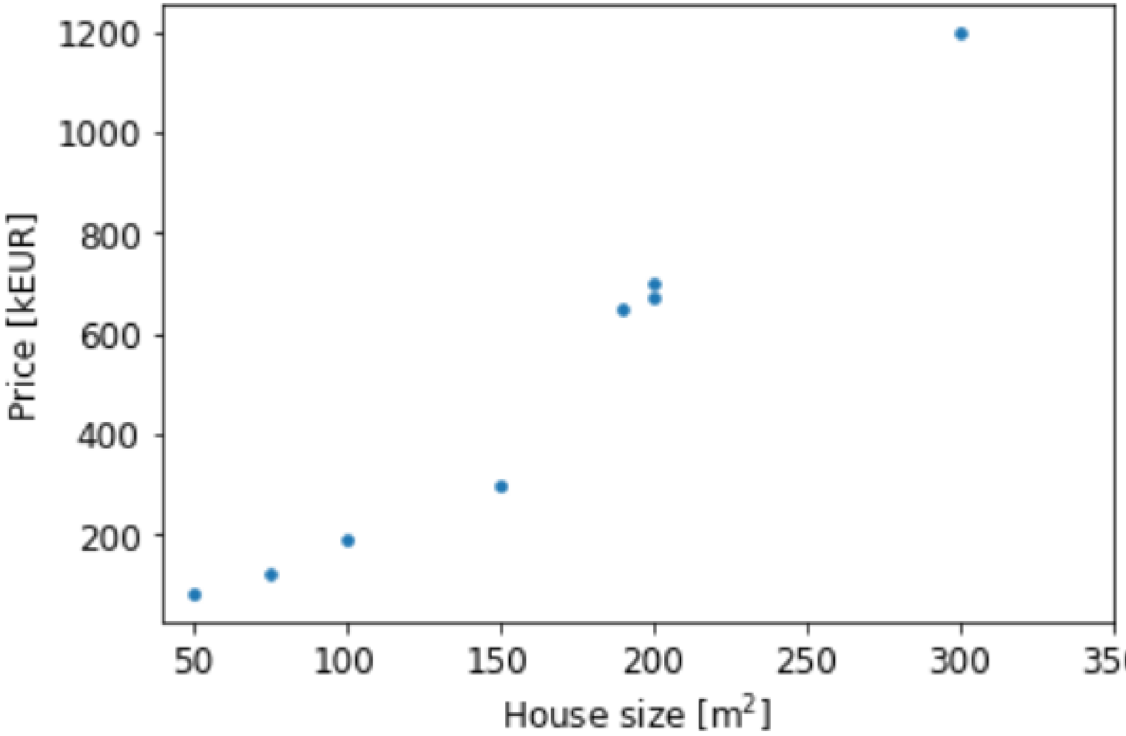
$\{(x^i, y^i)\} \quad i = 1, \dots, N$ available dataset

$\hat{y}^i = M(x^i; \theta)$ parametric model

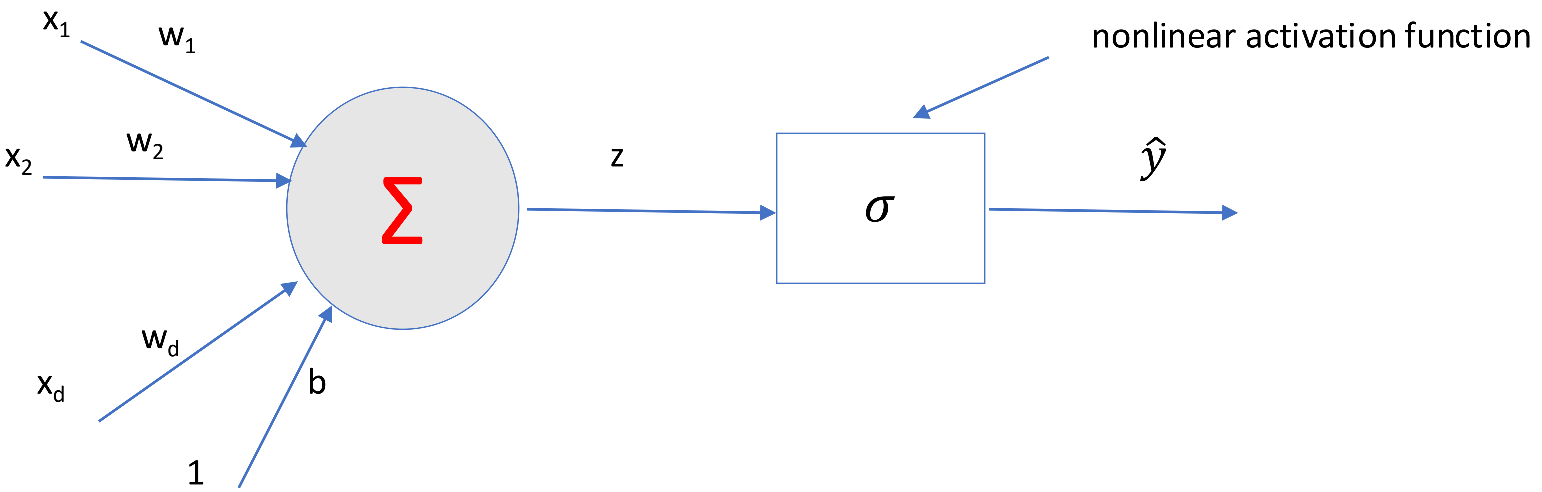
$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left(y^i - \hat{y}^i(\theta)\right)^2$ Loss (MSE)

$$\boxed{\theta^* = arg \min_{\theta} \mathcal{L}(\theta)}$$

Real estate application

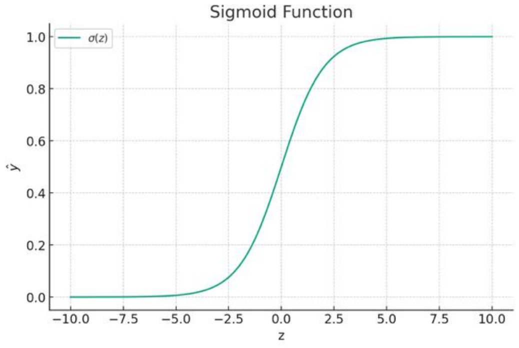


Basic units for Neural Networks

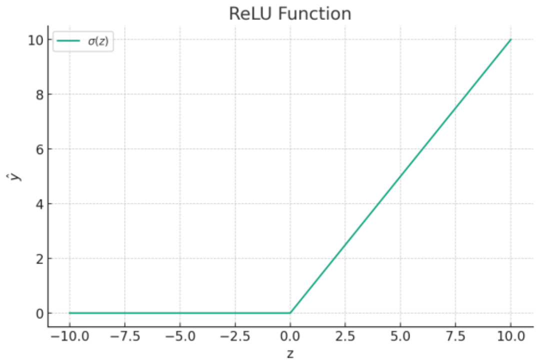


$$\hat{y} = \sigma\left(\sum_{j=1}^{n} w_j x_j + b\right)$$

$$\sigma(z) = \frac{1}{1+e^{-z}} = \frac{e^z}{1+e^z}$$



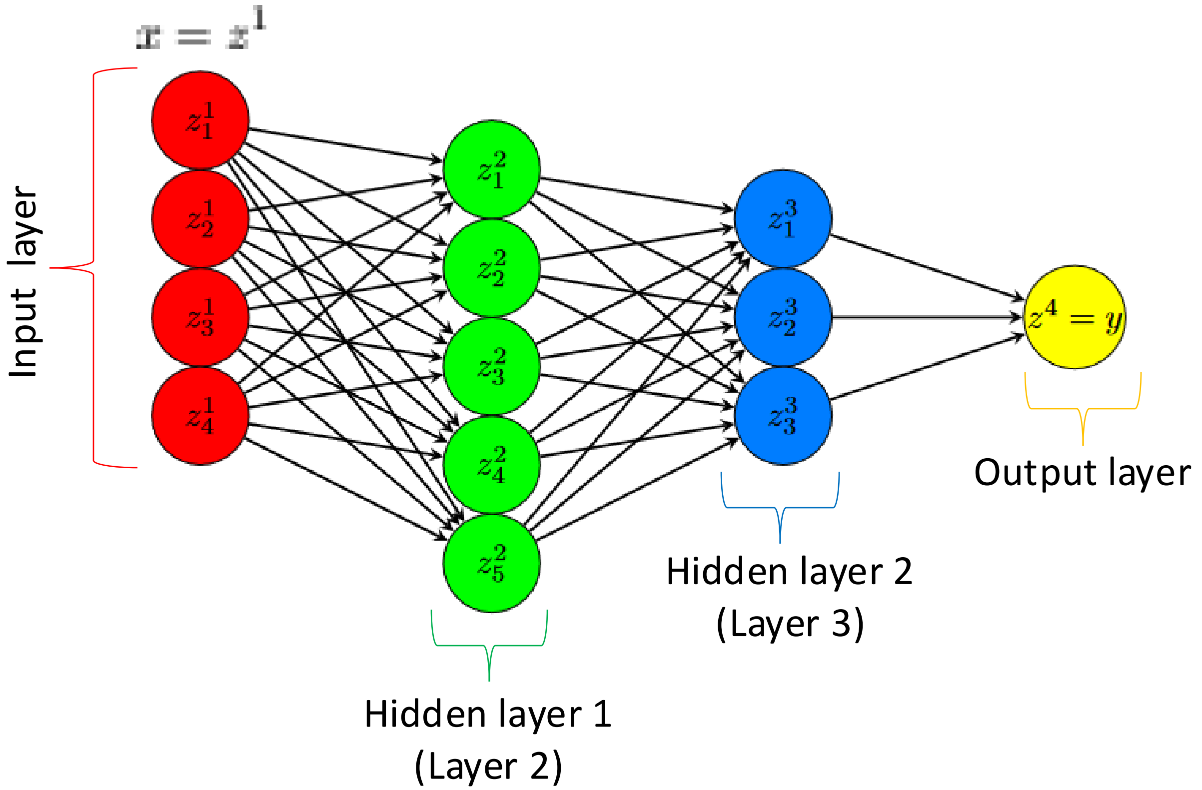
$$\sigma(z) = \begin{cases} z \ \ if\ z \geq 0 \\ 0 \ \ if\ z < 0 \end{cases}$$



We have a non-linear relation between inputs and outputs!

Fully-connected Feedforward Neural Networks

Move from one neuron to a hierarchical structure with fully connected neurons



$$z_1^2 = \sigma\left(\sum_{j=1}^{4} w_{1,j}^1 z_j^1 + b_1^1\right) = \sigma\left(W_1^1 z^1 + b_1^1\right)$$

$$z_2^3 = \sigma\left(\sum_{j=1}^{5} w_{2,j}^2 z_j^2 + b_2^2\right) = \sigma\left(W_2^2 z^2 + b_2^2\right)$$

$$y = z^4 = \sum_{j=1}^{3} w_{1,j}^3 z_j^3 + b^3 = W_1^3 z^3 + b^3$$

Overall: $y = W_3\sigma(W_2\sigma(W_1 x + b_1) + b_2) + b_3$

We can easily define a NARX structures parameterized by the weights (and biases) of the network

$$\hat{y}(k) = f\big(\underbrace{y(k-1),\ldots,y(k-na),u(k),u(k-1),\ldots,u(k-nb)}_{z^1(k)}; W, b\big)$$

FFN: PyTorch

Definition of the model class

```python
import torch
import torch.nn as nn

class FeedforwardNeuralNetModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(FeedforwardNeuralNetModel, self).__init__()

        # Linear layer 1
        self.fc1 = nn.Linear(input_dim, hidden_dim[0])
        # Activation 1
        self.sigmoid1 = nn.Sigmoid()

        # Linear layer 2
        self.fc2 = nn.Linear(hidden_dim[0], hidden_dim[1])
        # Activation 2
        self.sigmoid2 = nn.Sigmoid()

        # Output layer (linear layer)
        self.output = nn.Linear(hidden_dim[1], output_dim)

    def forward(self, x):
        # Linear function  # LINEAR
        x = self.fc1(x)
        x = self.sigmoid1(x)
        x = self.fc2(x)
        x = self.sigmoid2(x)
        x = self.output(x)

        return x
```

Instantiate the model class, run and show model

```python
input_dim = 4
hidden_dim = [200, 300]
output_dim = 1
batch_dim = 10

model = FeedforwardNeuralNetModel(input_dim, hidden_dim, output_dim)

x = torch.randn((batch_dim, input_dim))
y = model(x)

from torchsummary import summary
summary(model, input_size=(input_dim,))
```

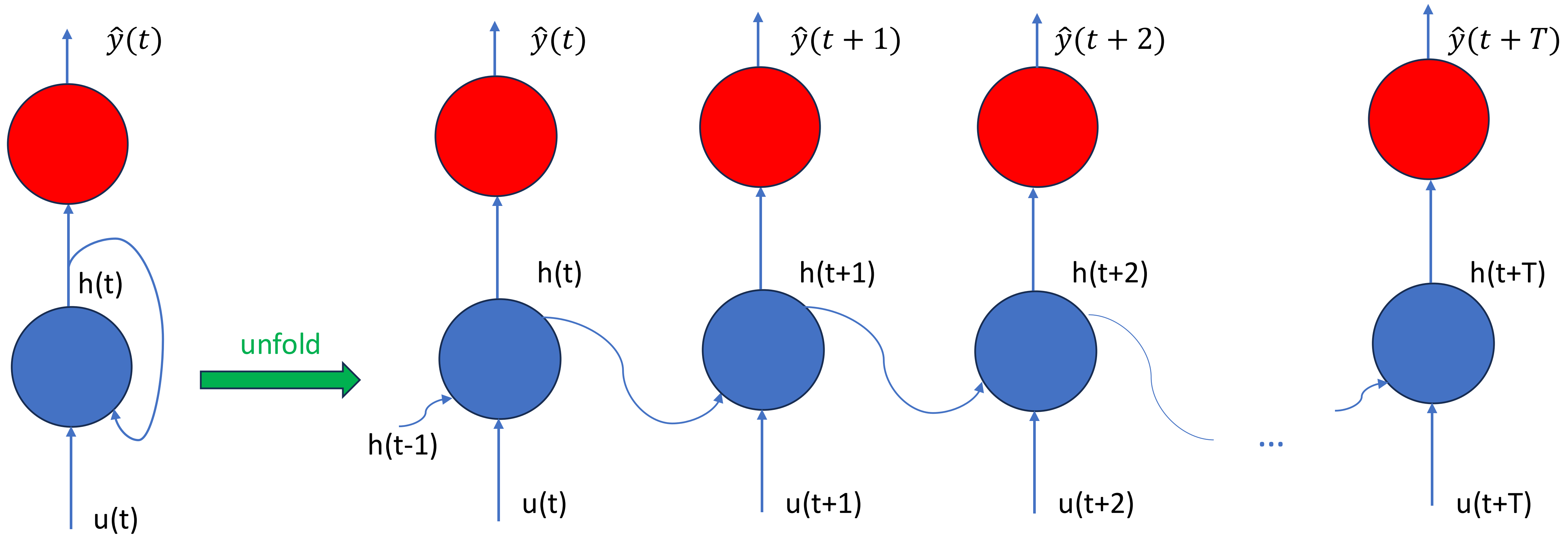
```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                 [-1, 200]           1,000
           Sigmoid-2                 [-1, 200]               0
            Linear-3                 [-1, 300]          60,300
           Sigmoid-4                 [-1, 300]               0
            Linear-5                   [-1, 1]             301
================================================================
Total params: 61,601
Trainable params: 61,601
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.01
Params size (MB): 0.23
Estimated Total Size (MB): 0.24
```

FFN: Training in PyTorch

```python
# Define the model
model = FeedforwardNeuralNetwork(input_size=4, hidden_sizes=[200, 300], output_size=1)

# Define the Optimizer
optimizer = optim.SGD(model.parameters(), lr=1e-4)

# Batch Gradient Descent
for epoch in range(max_epochs):

    optimizer.zero_grad()

    # forward pass
    y_hat = model(x)
    loss = torch.mean( (y_hat - y)**2 )

    # Backward pass and update
    loss.backward()
    optimizer.step()
```

Recurrent Neural Networks (RNNs)

- Architectures tailored to process time series data and temporal information (audio, text, video, signals, etc.)



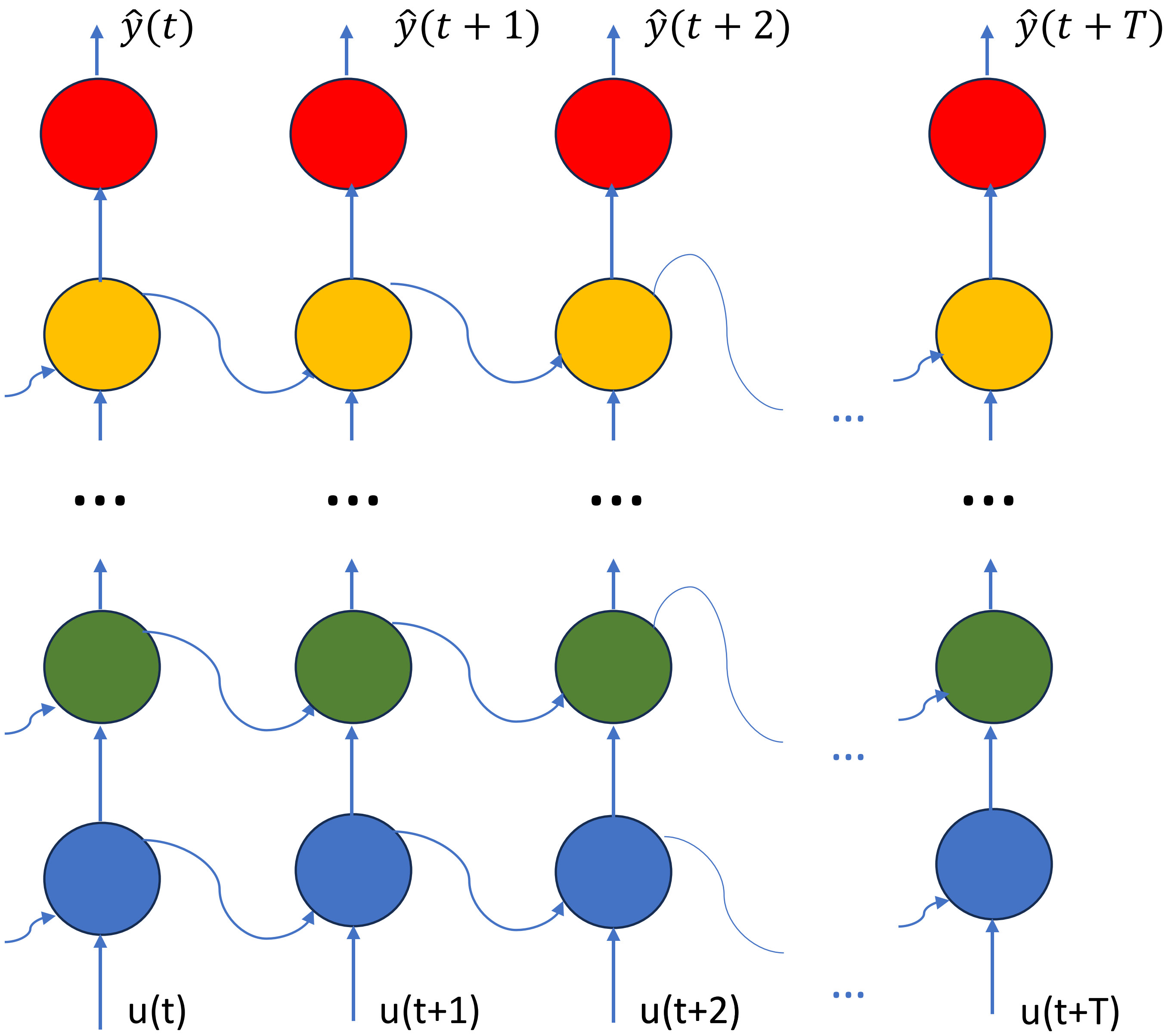
$$h(t) = f(h(t-1), u(t); W_f)$$

$$\hat{y}(t) = g(h(t); W_g)$$

$$\mathcal{L} = \sum_{i=0}^{T} \|\hat{y}(t+i) - y(t+i)\|^2$$

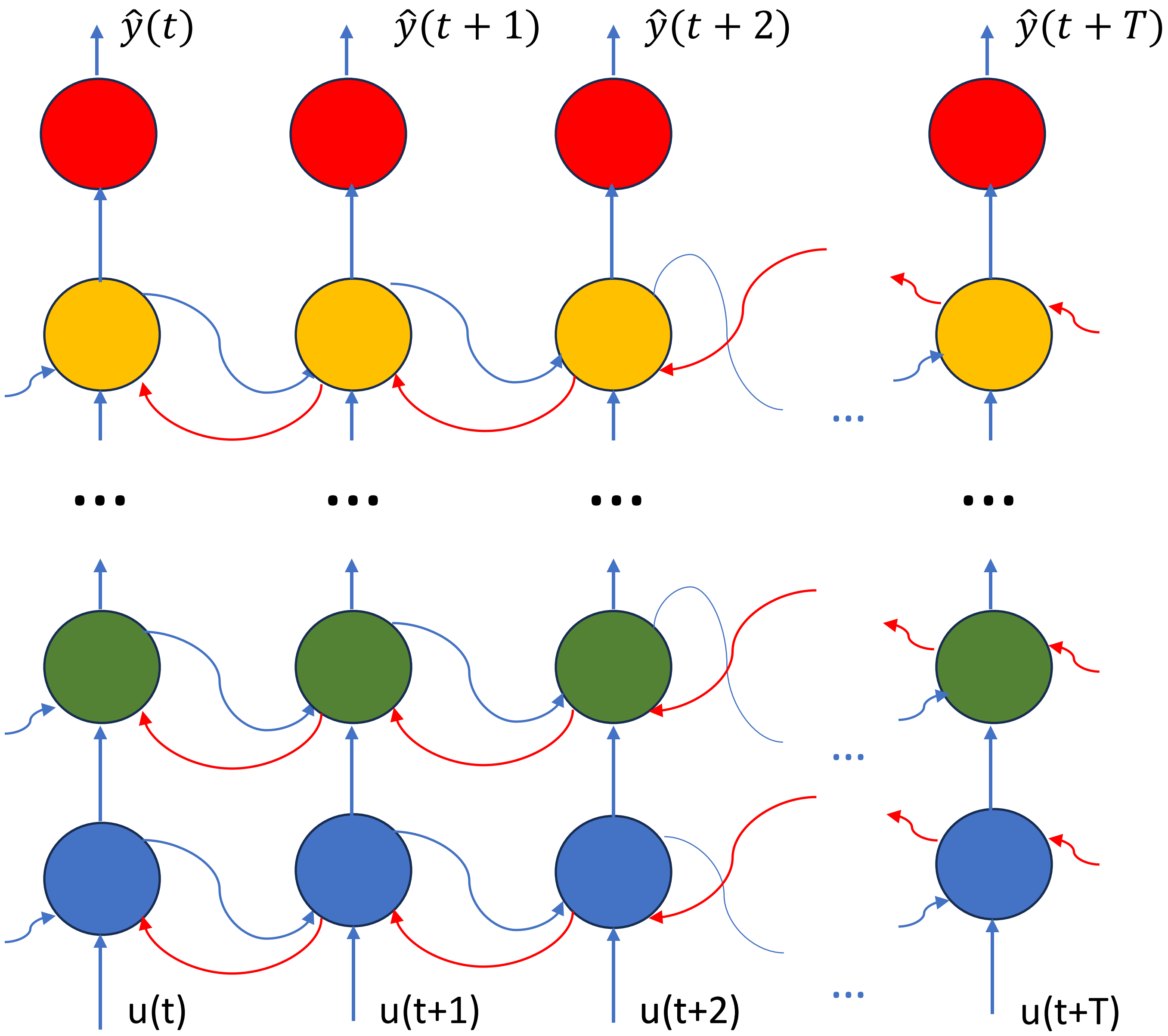
or skip some initial samples

Multi-layer RNN



Hidden states of a layer are also inputs of the next layer

Bidirectional RNN



Causality is lost. Might not be useful for prediction, but for tasks like smoothing

Creating training data

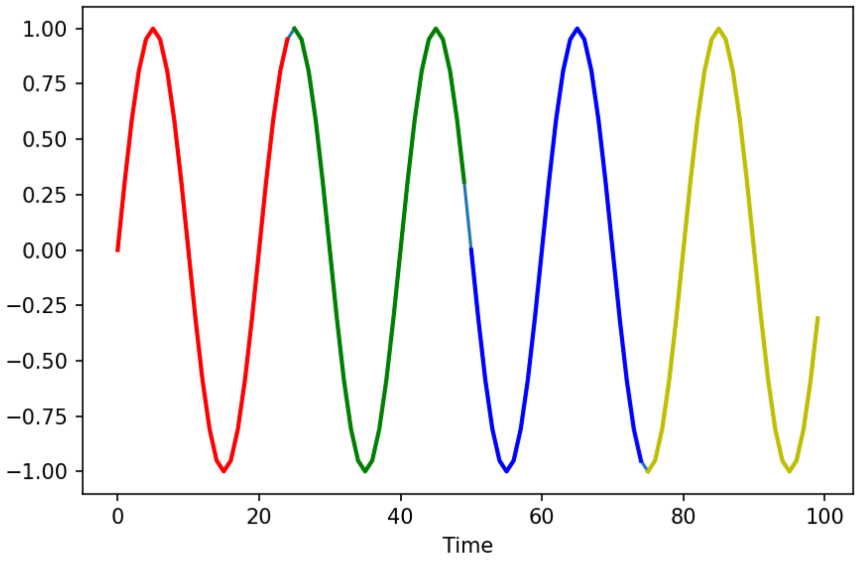
- Given an input/output sequence of length T, we can unfold the network up to T steps (you simulate T-step time ahead)

$$\mathcal{L} = \frac{1}{T}\sum_{t=0}^{T}\|\hat{y}(t) - y(t)\|^2$$

- … or we can split the sequence into shorter sub-sequences (overlapped or not) of length L<<T, and thus create batch of sub-sequences. Network is unfolded for L steps (when you train, you predict L-step time ahead)

$$\mathcal{L}^{(q)} = \frac{1}{L}\sum_{i=0}^{L}\|\hat{y}(t_q + i) - y(t_q + i)\|^2$$

$$\mathcal{L} = \frac{1}{Q}\sum_{q=1}^{Q}\mathcal{L}^{(q)}$$



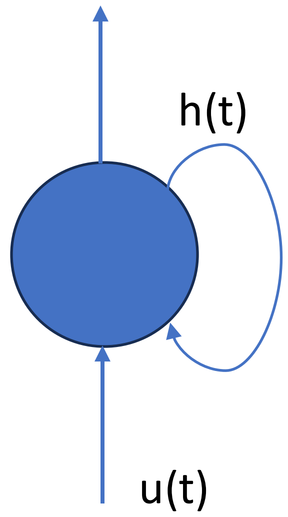
Vanilla Recurrent Neural Network

$$h(t) = \tanh(W_{hh} h(t-1) + W_{uh} u(t) + b_h)$$

CLASS torch.nn.RNN(*self*, *input_size*, *hidden_size*, *num_layers=1*, *nonlinearity='tanh'*, *bias=True*, *batch_first=False*, *dropout=0.0*, *bidirectional=False*, *device=None*, *dtype=None*) [SOURCE]

Parameters

- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two RNNs together to form a *stacked RNN*, with the second RNN taking in outputs of the first RNN and computing the final results. Default: 1
- **nonlinearity** – The non-linearity to use. Can be either `'tanh'` or `'relu'`. Default: `'tanh'`
- **bias** – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **batch_first** – If `True`, then the input and output tensors are provided as (*batch, seq, feature*) instead of (*seq, batch, feature*). Note that this does not apply to hidden or cell states. See the Inputs/Outputs sections below for details. Default: `False`



$$h(t) = f(h(t-1), u(t); W_f)$$

LSTM: Long Short-Term Memory Neural Networks

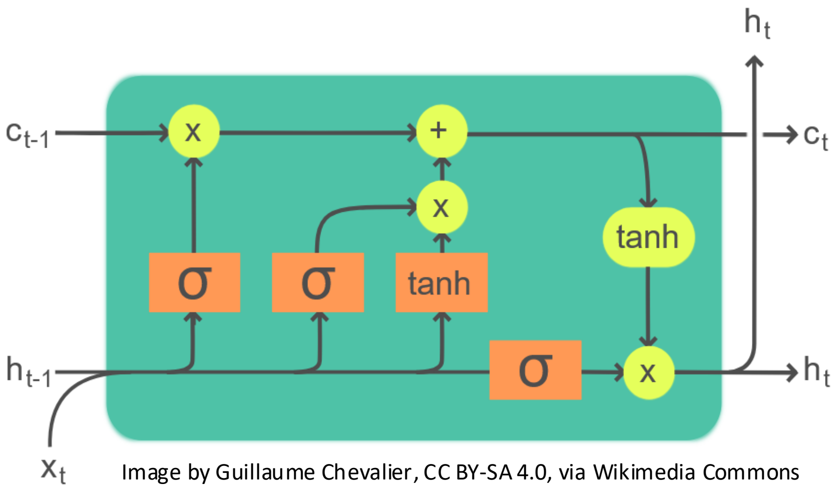
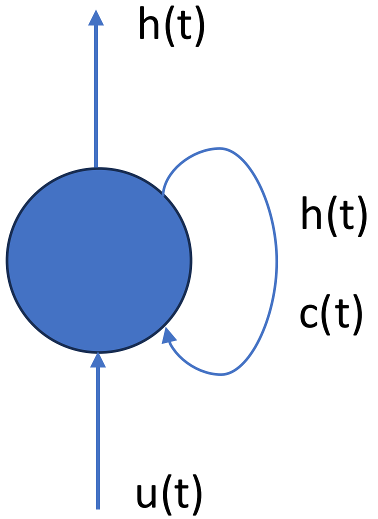


Image by Guillaume Chevalier, CC BY-SA 4.0, via Wikimedia Commons



CLASS torch.nn.LSTM(*self*, *input_size*, *hidden_size*, *num_layers=1*, *bias=True*, *batch_first=False*, *dropout=0.0*, *bidirectional=False*, *proj_size=0*, *device=None*, *dtype=None*) [SOURCE]

$$c(t) = f_c(c(t-1), h(t-1), u(t); W_c)$$

$$h(t) = f_h(c(t), h(t-1), u(t); W_h)$$

```
B, L = 3, 5
u_size, hc_size, y_size = 10, 20, 1

lstm = nn.LSTM(u_size, hc_size,  batch_first=True)
u = torch.randn(B, L, u_size)
h0 = torch.zeros(1, B, hc_size)
c0 = torch.zeros(1, B, hc_size)
h, (hn, cn) = lstm(u, (h0, c0))
L1 = nn.Linear(in_features = hc_size, out_features = y_size)
y = L1(h)
print(h.shape, y.shape)
```

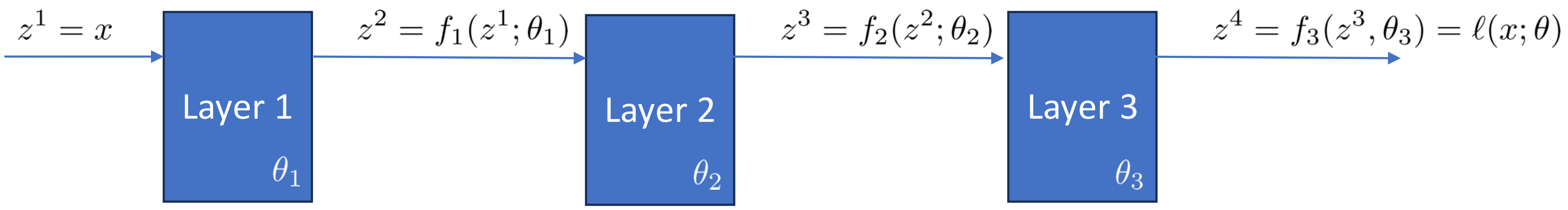
```
torch.Size([3, 5, 20]) torch.Size([3, 5, 1])
```

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural Computation, 9(8), 1735–1780

Backpropagation

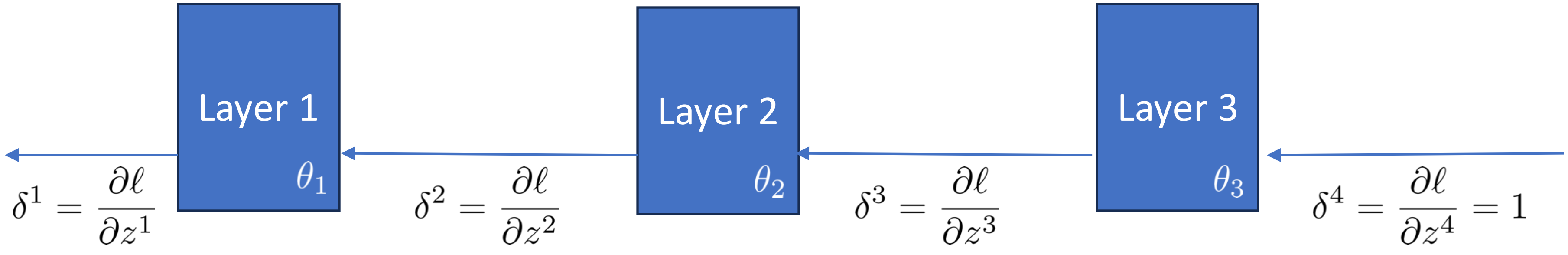
Back-propagation I

$$\ell(x;\theta) = (y_k - \hat{y}_k(x_k;\theta))^2$$



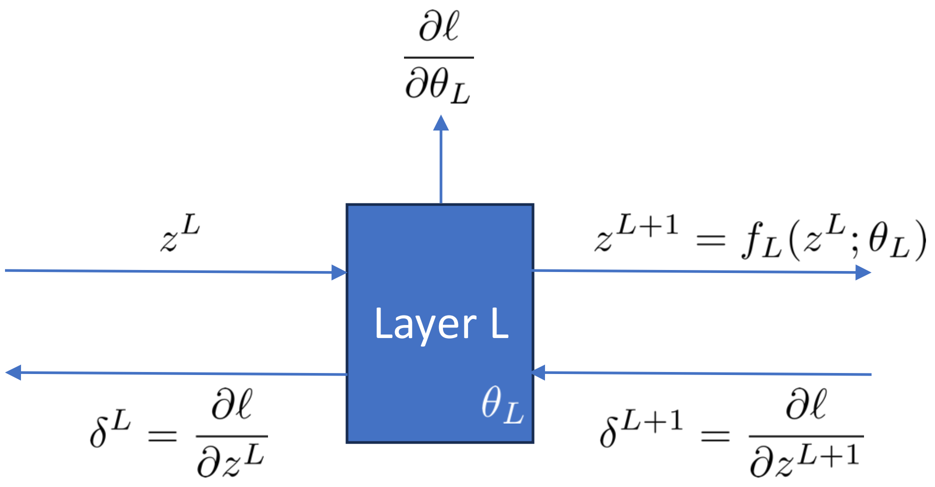
What we want: $\dfrac{\partial \ell}{\partial \theta_1}, \dfrac{\partial \ell}{\partial \theta_2}, \dfrac{\partial \ell}{\partial \theta_3}$

Idea: Back propagate $\delta^L = \dfrac{\partial \ell}{\partial z^L}, \quad L = 4, 3, 2, 1$



Back-propagation II

Focus on Layer L



$$\delta^L = \frac{\partial \ell}{\partial z^{L+1}} \frac{\partial z^{L+1}}{\partial z^L} = \delta^{L+1} \frac{\partial z^{L+1}}{\partial z^L}$$

$$\frac{\partial \ell}{\partial \theta_L} = \frac{\partial \ell}{\partial z^{L+1}} \frac{\partial z^{L+1}}{\partial \theta_L} = \delta^{L+1} \frac{\partial z^{L+1}}{\partial \theta_L}$$

We have an interface to connect blocks/layers and recursively compute the derivatives!

Technicalities

Sigmoid activation function: $\sigma(z) = \frac{1}{1+e^{-z}} = \frac{e^z}{1+e^z}$; $\sigma'(z) = \sigma(z)(1\text{-}\ \sigma(z))$

Linear layer: $z^{L+1} = W z^L$ $\quad \frac{\partial z^{L+1}}{\partial z_i^L} = W_{:,i} \rightarrow \delta^L = \delta^{L+1} W$

```
for epoch in range(max_epochs):

    optimizer.zero_grad()

    # forward pass
    y_hat = model(x)
    loss = torch.mean( (y_hat - y)**2 )

    # Backward pass and update
    loss.backward()
    optimizer.step()
```

Algorithm: reverse-mode automatic differentiation. Application to neural network training is called **back-propagation**.