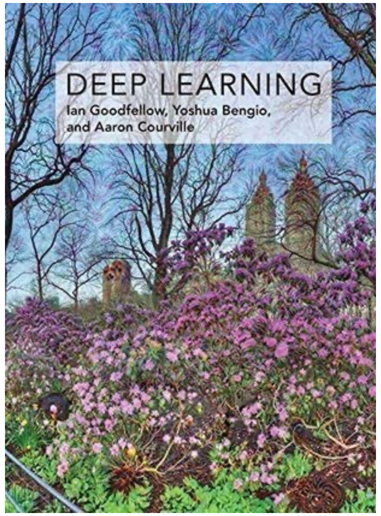
9th November 2021

ML 4 Time-series: Recurrent Neural Networks

Dr. Andrew P. Creagh | Dr. Anshul Thacker | Prof. David A. Clifton

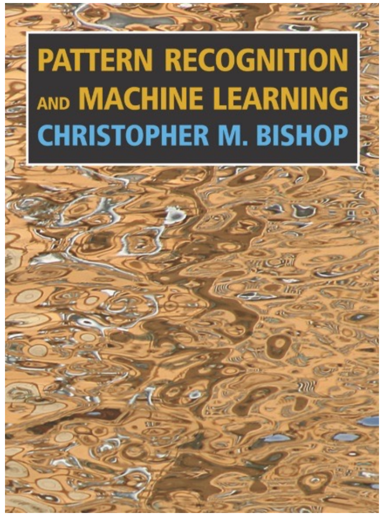
Institute of Biomedical Engineering (IBME),
Department of Engineering Science,
Old Road Campus Research Building (ORCRB),
University of Oxford

Resources



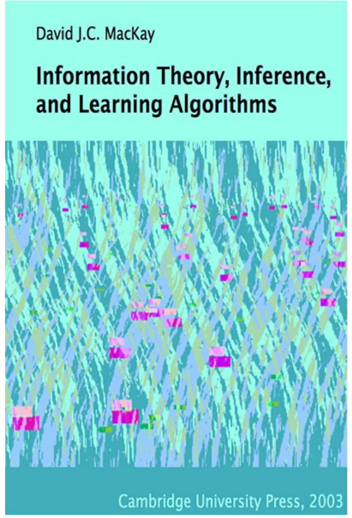
A well-written introduction to all things deep learning (DL), from leaders in the field of theoretical DL.

Very light maths



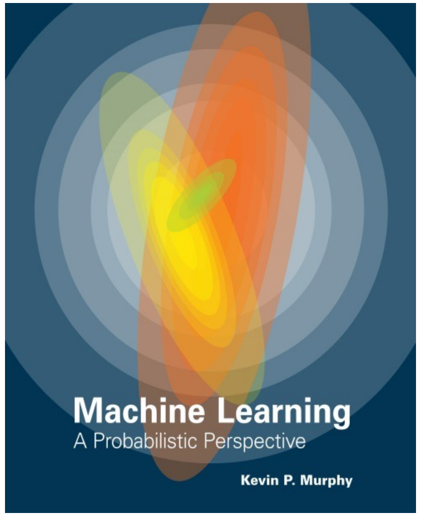
A core classic describing most non-DL algorithms. Very good for one's general understanding.

Reasonably "maths-y"



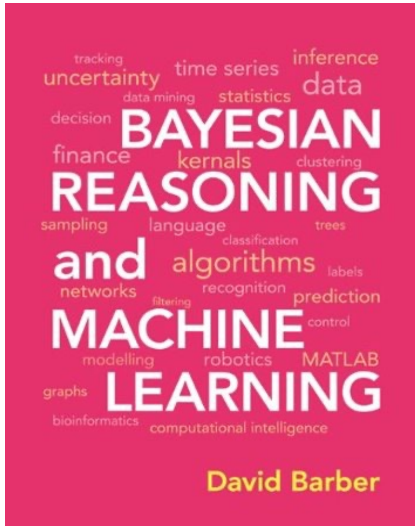
Another core classic, from one of the field's (sadly departed) foundational thinkers – good, even though it's from Cambridge

Reasonably "maths-y"



Seriously comprehensive, one of the best books for general machine learning. Excellent examples.

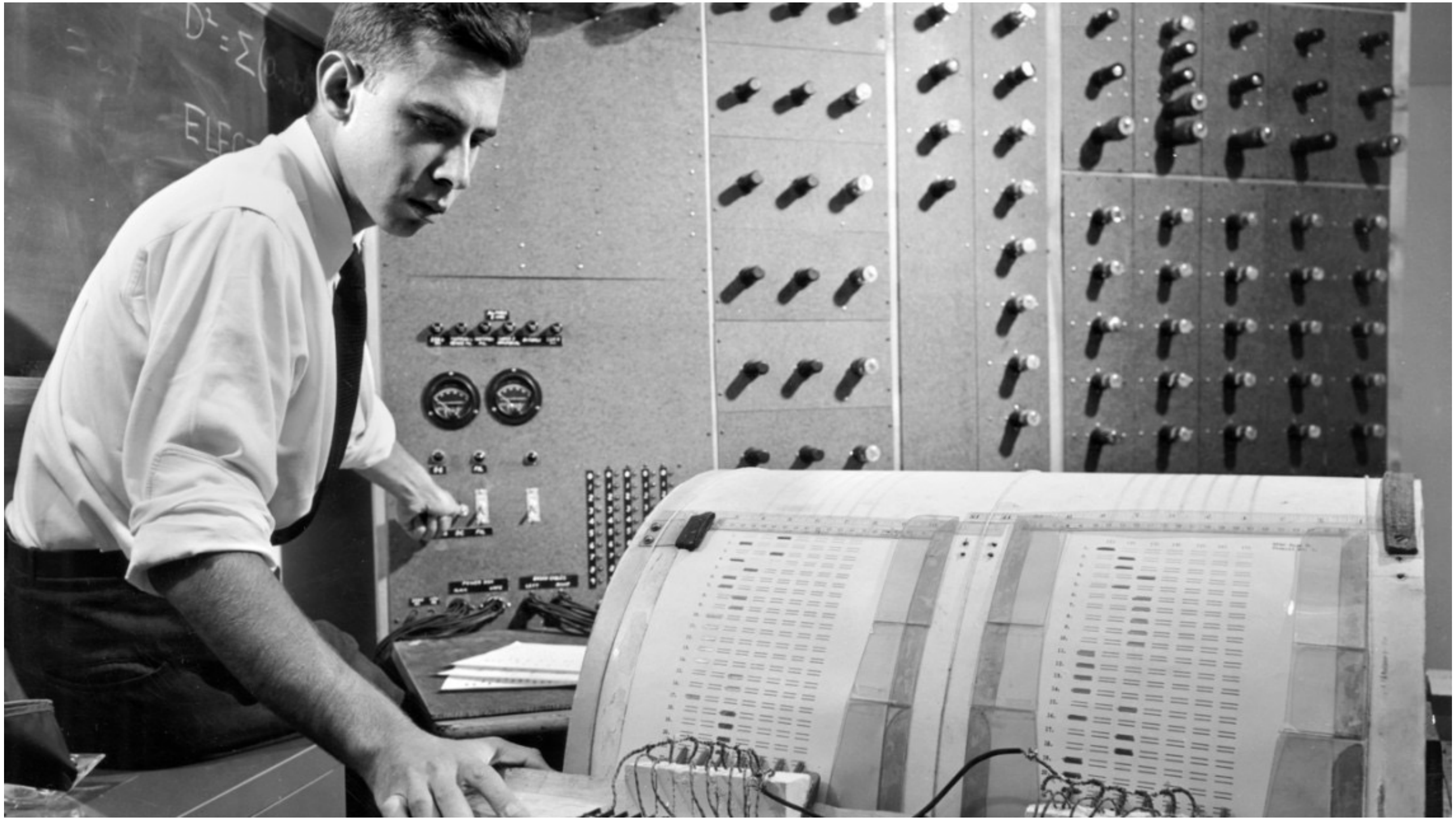
Really rather "maths-y"



Big, bad, and Bayesian. Everything you'd like to know about Bayesian machine learning. Great for time-series analysis.

Seriously "maths-y"

mild medium hot extra hot



Frank Rosenblatt 1950, Ph.D. 1956, works on the "perceptron" – what he described as the first machine "capable of having an original idea."

Source: https://news.cornell.edu/stories/2019/09/professors-perceptron-paved-way-ai-60-years-too-soon

The Perceptron

REPORT NO. 85-460-1

THE PERCEPTRON

A PERCEIVING AND RECOGNIZING AUTOMATON

(PROJECT PARA)

January, 1957

Prepared by: Frank Rosenblatt

Frank Rosenblatt,
Project Engineer

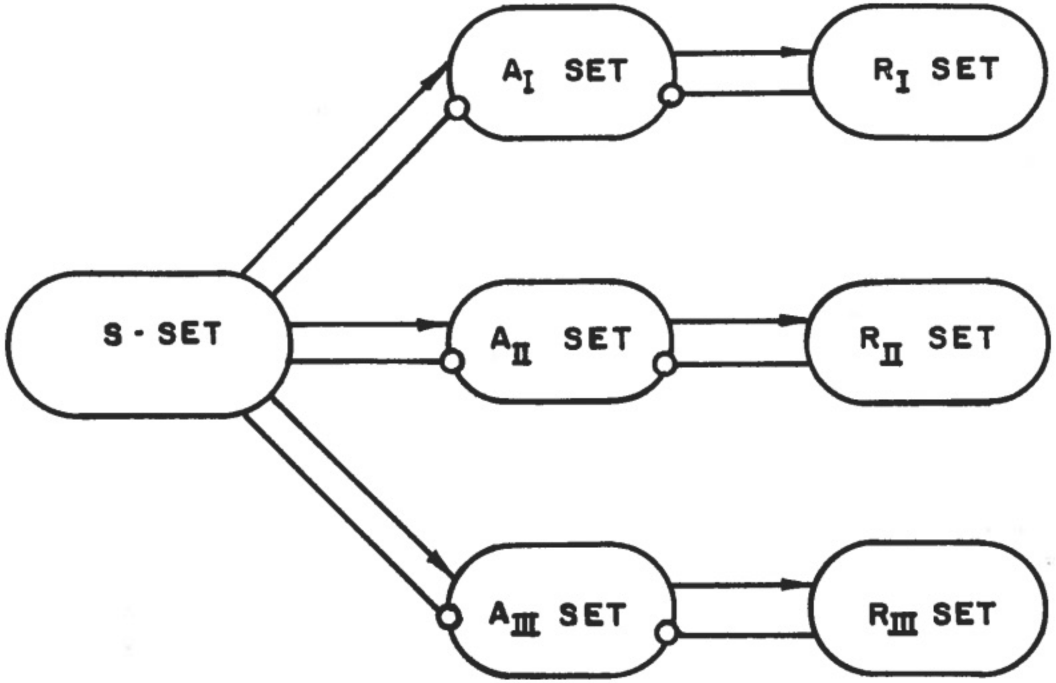
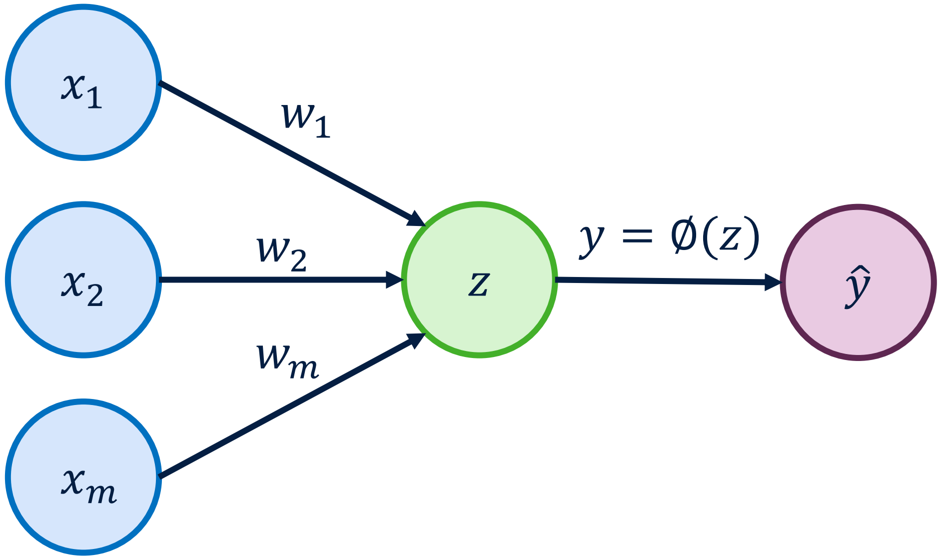


FIGURE 2

ORGANIZATION OF A PERCEPTRON WITH
THREE INDEPENDENT OUTPUT-SETS

The Perceptron

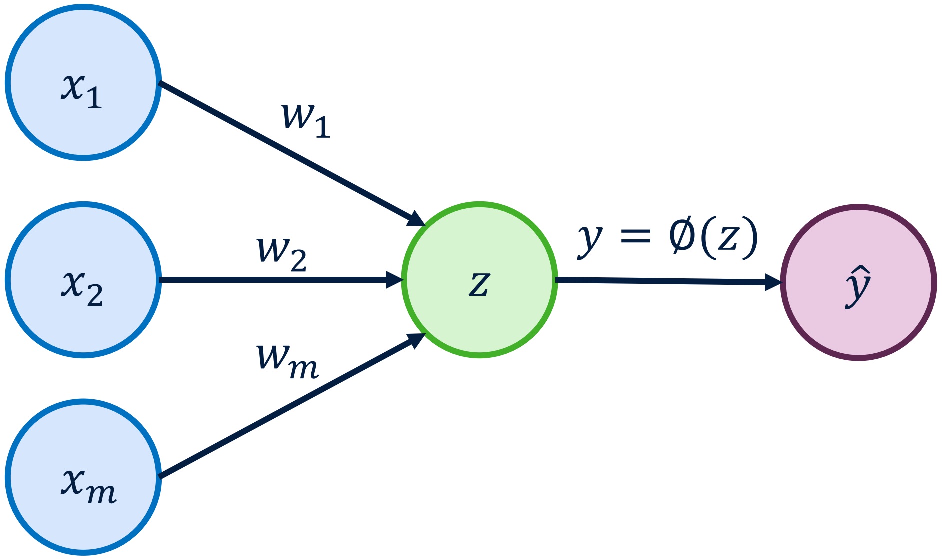
The perceptron is the building block to modern day feed-forward deep networks. A perceptron layer takes a weighted linear combination (sum) of the inputs $\mathbf{x} \in \mathbb{R}^m$ and transform this to an output through some (often non-linear) positive and monotonically increasing activation function $\emptyset(\cdot)$ to yield a prediction or output $\hat{y}$



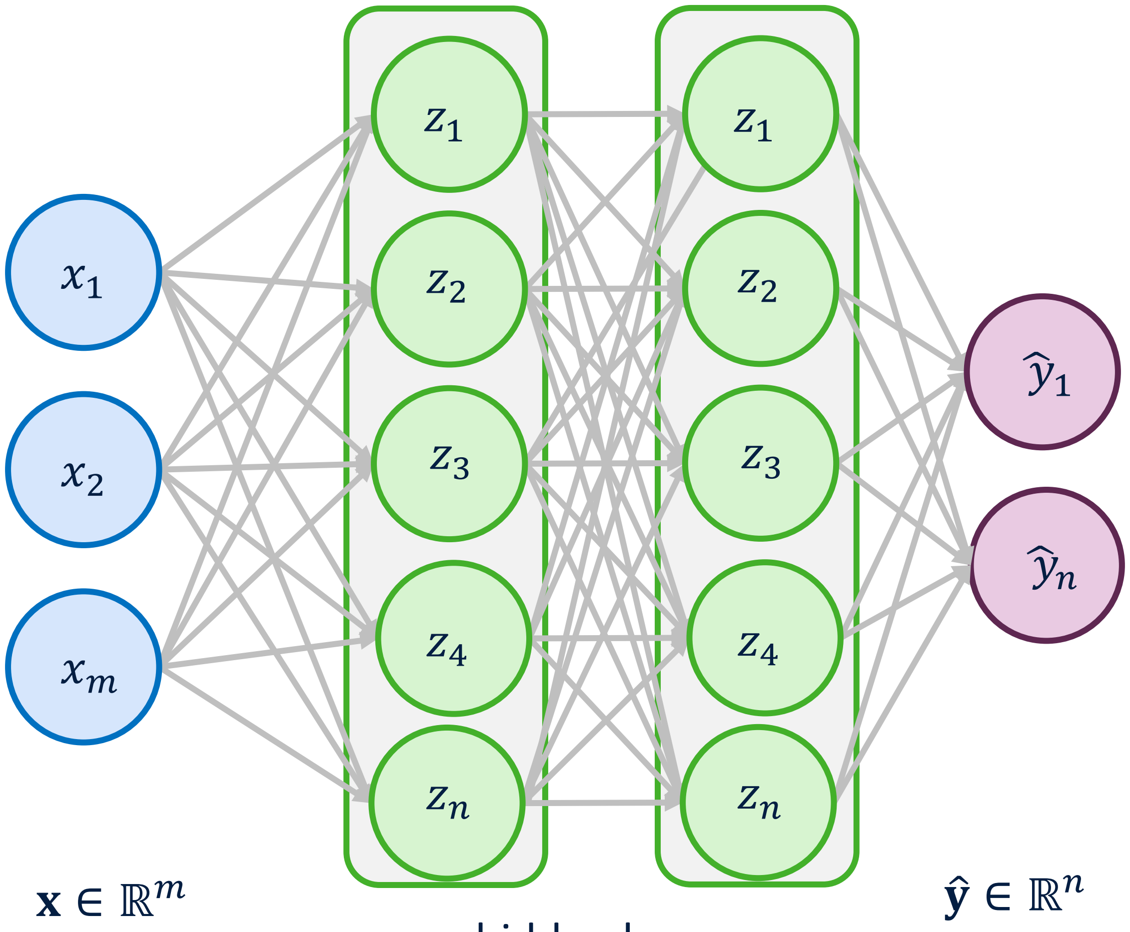
$\mathbf{x} \in \mathbb{R}^m$ $\qquad$ $z = \mathbf{w}^\intercal \mathbf{x}$ $\qquad$ $\hat{\boldsymbol{y}} = \emptyset(\mathbf{w}^\intercal \mathbf{x})$

Advancing towards deep networks

the perceptron

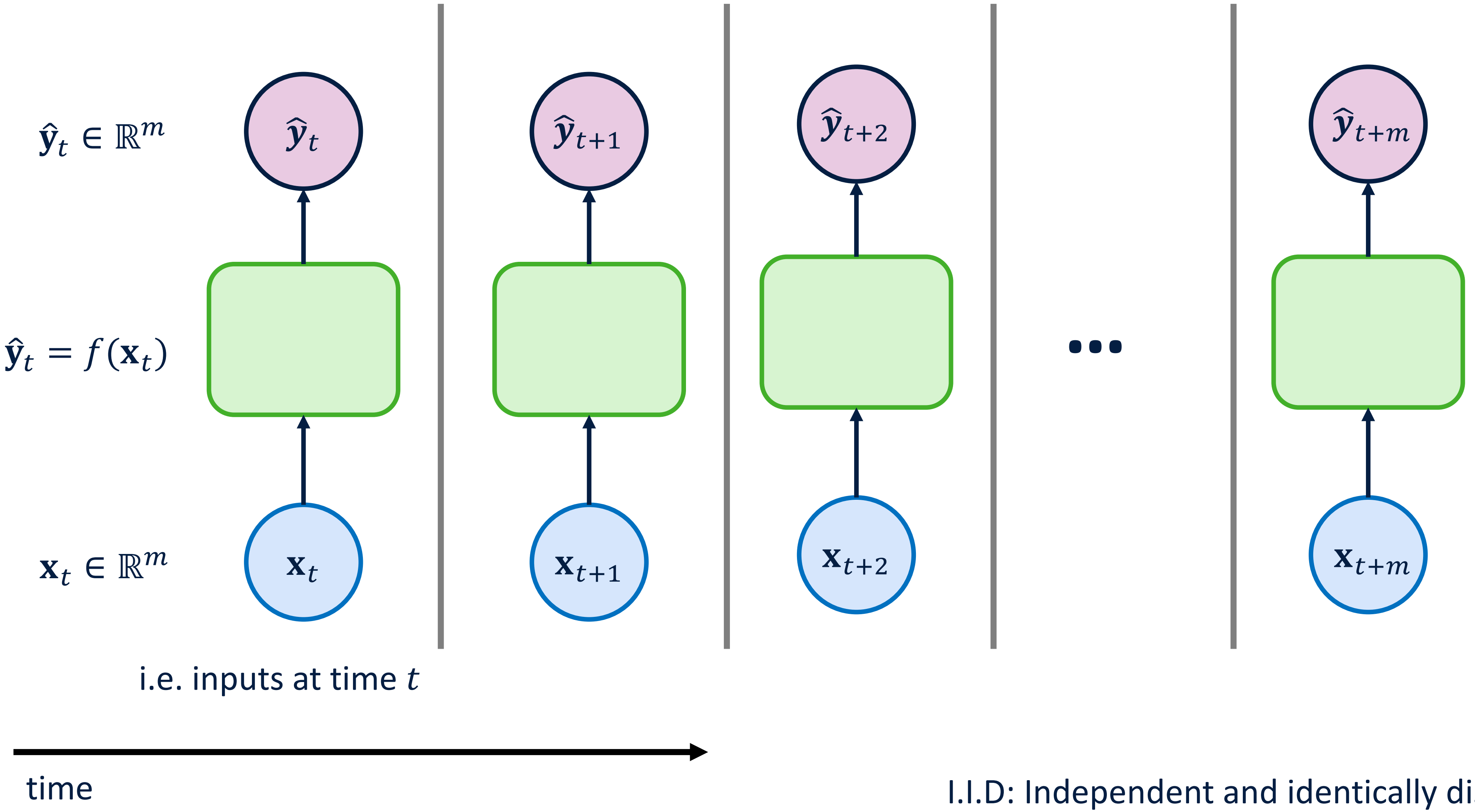


MLP, feed forward networks



Deep learning typically describes multi-layer perceptron (MLP) blocks that are stacked in cascading architectural arrangements, consisting of a number of (often non-linear) functions successively layered together, which map an input $\mathbf{x}$ to some output $\mathbf{y}$

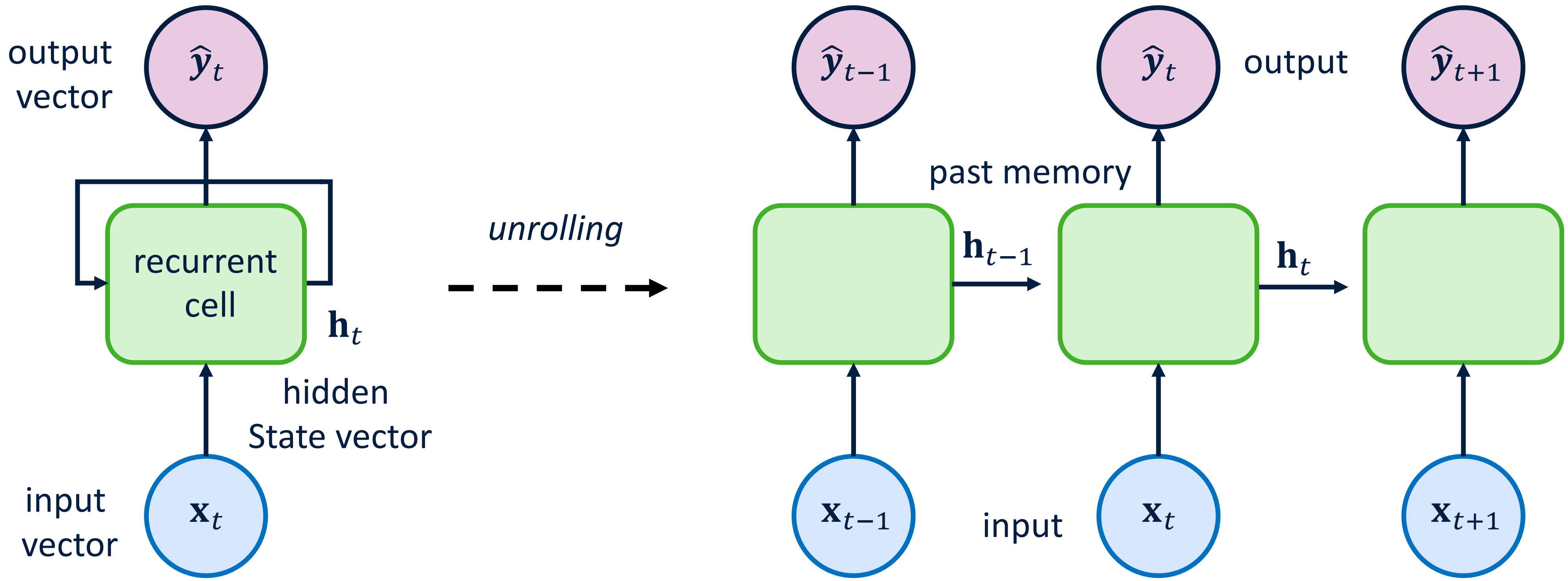
Treating individual time-steps as I.I.D.



I.I.D: Independent and identically distributed

Neurons with Recurrence

Apply a recurrence cell step at every time step to process a sequence



RNNs have a state $\mathbf{h}_t$, that is updated at each time step as a sequence is processed

$$\hat{\mathbf{y}}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1})$$

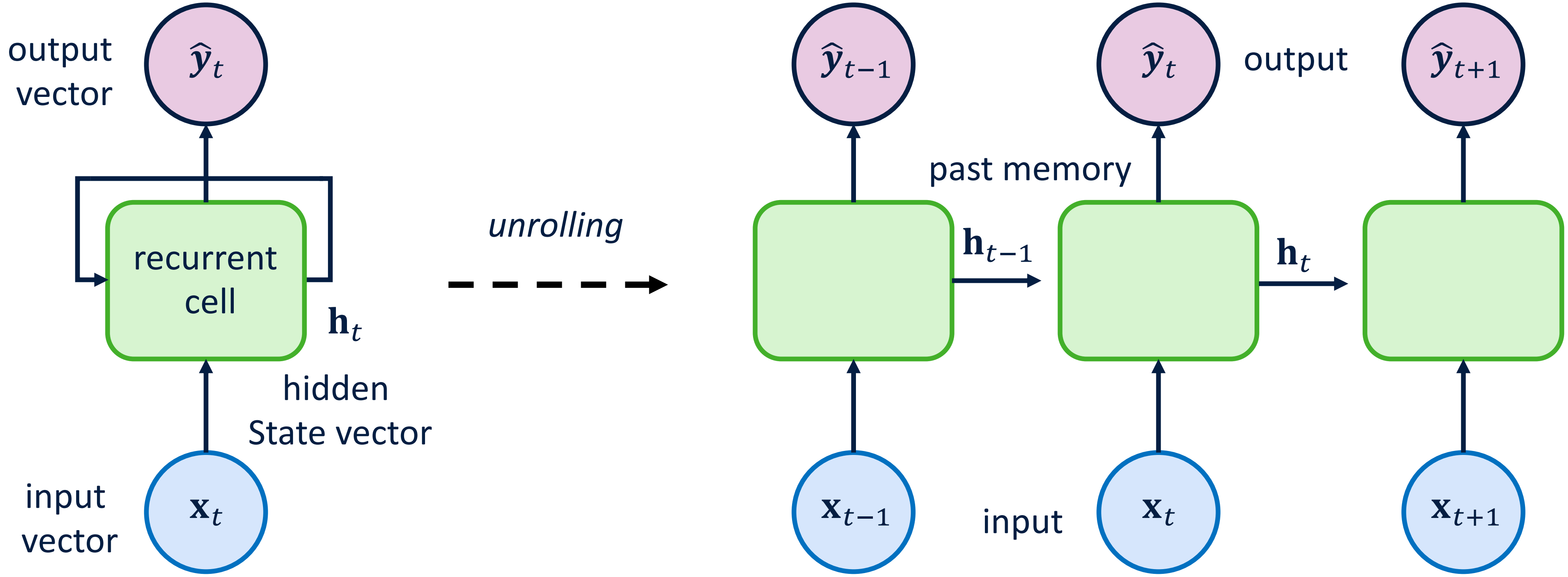
output

input

past memory

Neurons with Recurrence

Apply a recurrence cell step at every time step to process a sequence



RNNs have a state $\mathbf{h}_t$, that is updated at each time step as a sequence is processed

$$\hat{\mathbf{y}}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1})$$

output input past memory

cell state

$$\mathbf{h}_t = f_{\mathbf{w}}(\mathbf{x}_t, \mathbf{h}_{t-1})$$

update the hidden state using function with weights $\mathbf{w}$

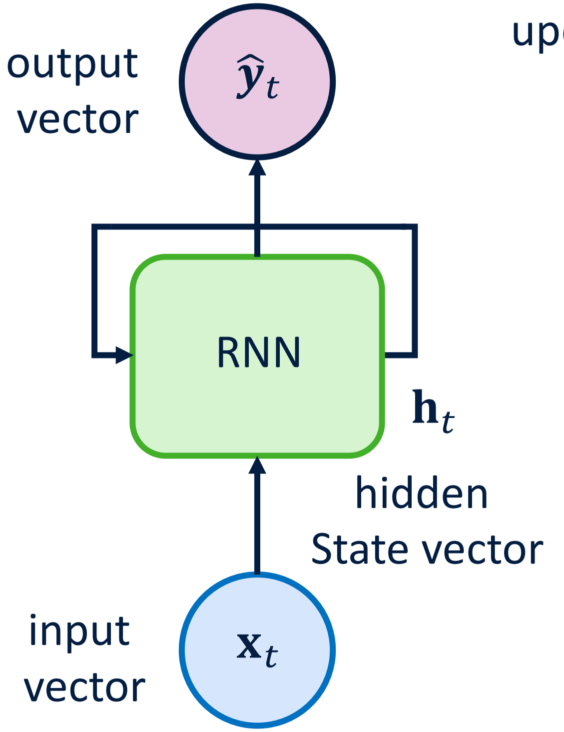
Recurrent Neural Networks (RNNs)

Apply a recurrence cell step at every time step to process a sequence

$\mathbf{W_{hh}}$: recurrent weights

$\mathbf{W_{hy}}$: hidden weights

$\mathbf{W_{xh}}$: input weights



update the hidden state using function with weights $\mathbf{w}$

$$\mathbf{h}_t = f_{\mathbf{w}}(\mathbf{x}_t, \mathbf{h}_{t-1})$$

cell state | input | previous state

$$\mathbf{h}_t = \tanh\left(\mathbf{W}_{\mathbf{hh}}^T \mathbf{h}_{t-1} + \mathbf{W}_{\mathbf{xh}}^T \mathbf{h}_t\right)$$

$$\hat{\mathbf{y}}_t = \mathbf{W}_{\mathbf{hy}}^T \mathbf{h}_t \qquad \text{(output vector)}$$

RNNs have a state $\mathbf{h}_t$, that is updated at each time step as a sequence is processed

Recurrent Neural Networks (RNNs)

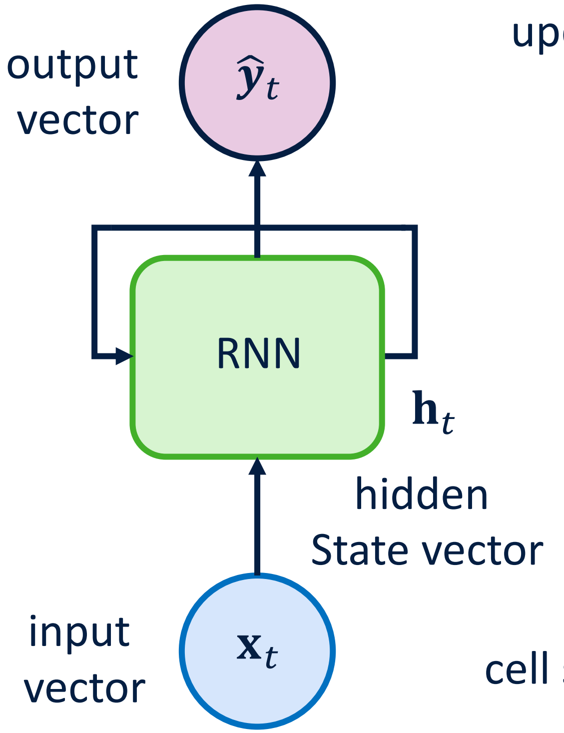
Apply a recurrence cell step at every time step to process a sequence

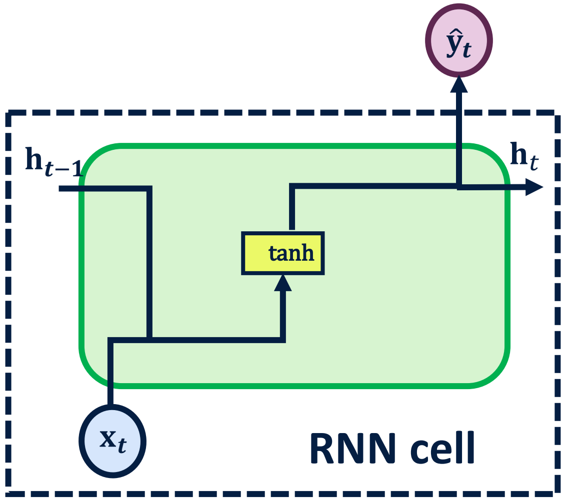
$\mathbf{W_{hh}}$: recurrent weights

$\mathbf{W_{hy}}$: hidden weights

$\mathbf{W_{xh}}$: input weights

update the hidden state using function with weights $\mathbf{w}$





cell state

$$\mathbf{h}_t = f_{\mathbf{w}}(\mathbf{x}_t, \mathbf{h}_{t-1})$$

$$\mathbf{h}_t = \tanh\left(\mathbf{W}_{\mathbf{hh}}^T \mathbf{h}_{t-1} + \mathbf{W}_{\mathbf{xh}}^T \mathbf{h}_t\right)$$

$$\hat{\mathbf{y}}_t = \mathbf{W}_{\mathbf{hy}}^T \mathbf{h}_t$$ (output vector)

RNNs have a state $\mathbf{h}_t$, that is updated at each time step as a sequence is processed

Recurrent Neural Networks (RNNs)

Apply a recurrence cell step at every time step to process a sequence

$\mathbf{W_{hh}}$: recurrent weights

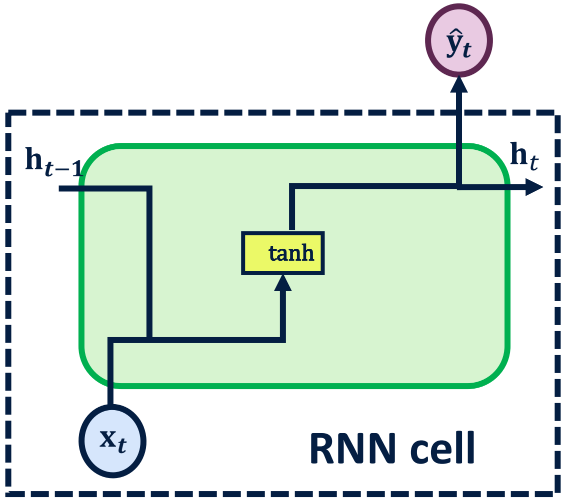
$\mathbf{W_{hy}}$: hidden weights

$\mathbf{W_{xh}}$: input weights

update the hidden state using function with weights $\mathbf{w}$

minimal code example

```
class RNN:
    # ...
    def step(self, x):
        # update the hidden state
        self.h =
        np.tanh(np.dot(self.W_hh, self.h)
        + np.dot(self.W_xh, x))
        # compute the output vector
        y = np.dot(self.W_hy, self.h)
        return y
```



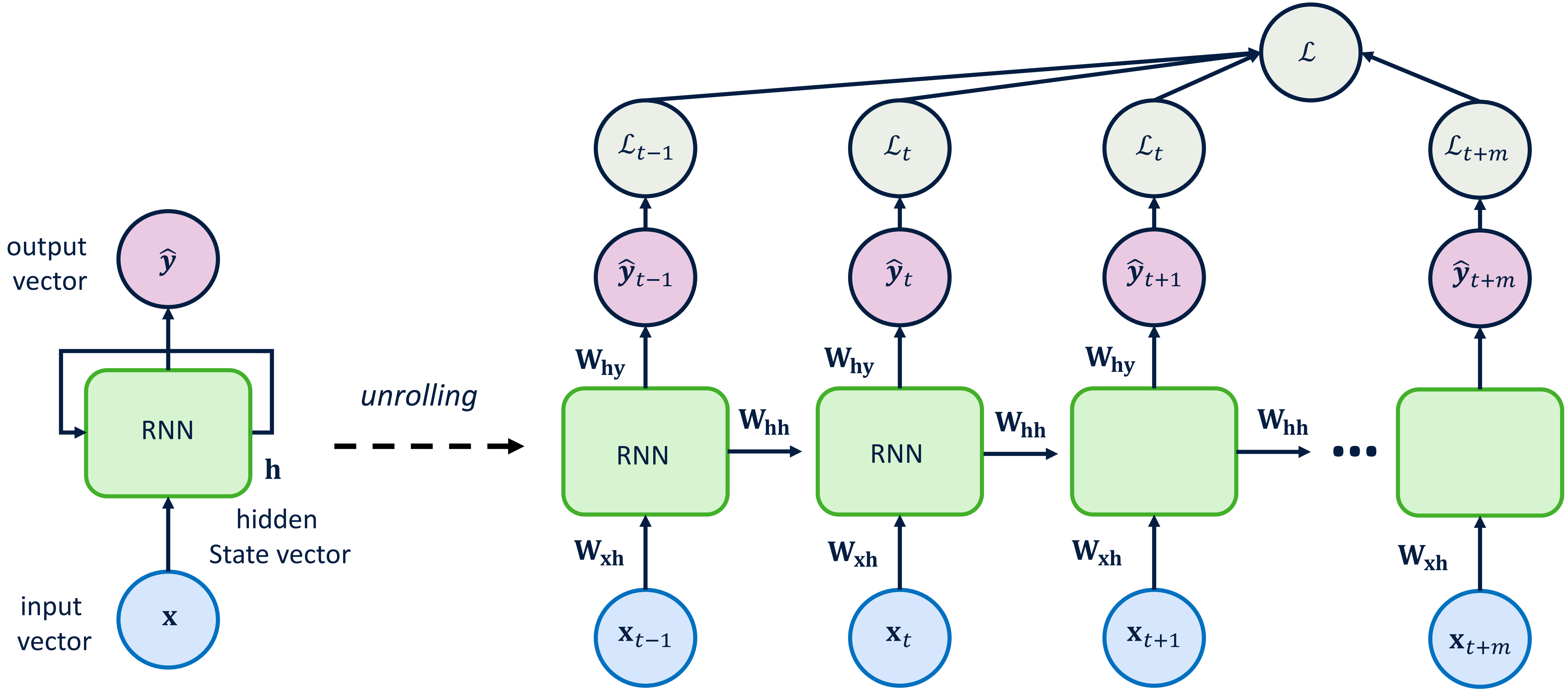
$$\mathbf{h}_t = f_{\mathbf{w}}(\mathbf{x}_t, \mathbf{h}_{t-1})$$

$$\mathbf{h}_t = \tanh\left(\mathbf{W}_{\mathbf{hh}}^T \mathbf{h}_{t-1} + \mathbf{W}_{\mathbf{xh}}^T \mathbf{h}_t\right)$$

$$\hat{\mathbf{y}}_t = \mathbf{W}_{\mathbf{hy}}^T \mathbf{h}_t \quad \text{(output vector)}$$

RNNs have a state $\mathbf{h}_t$, that is updated at each time step as a sequence is processed

Recurrent Neural Networks (RNNs)

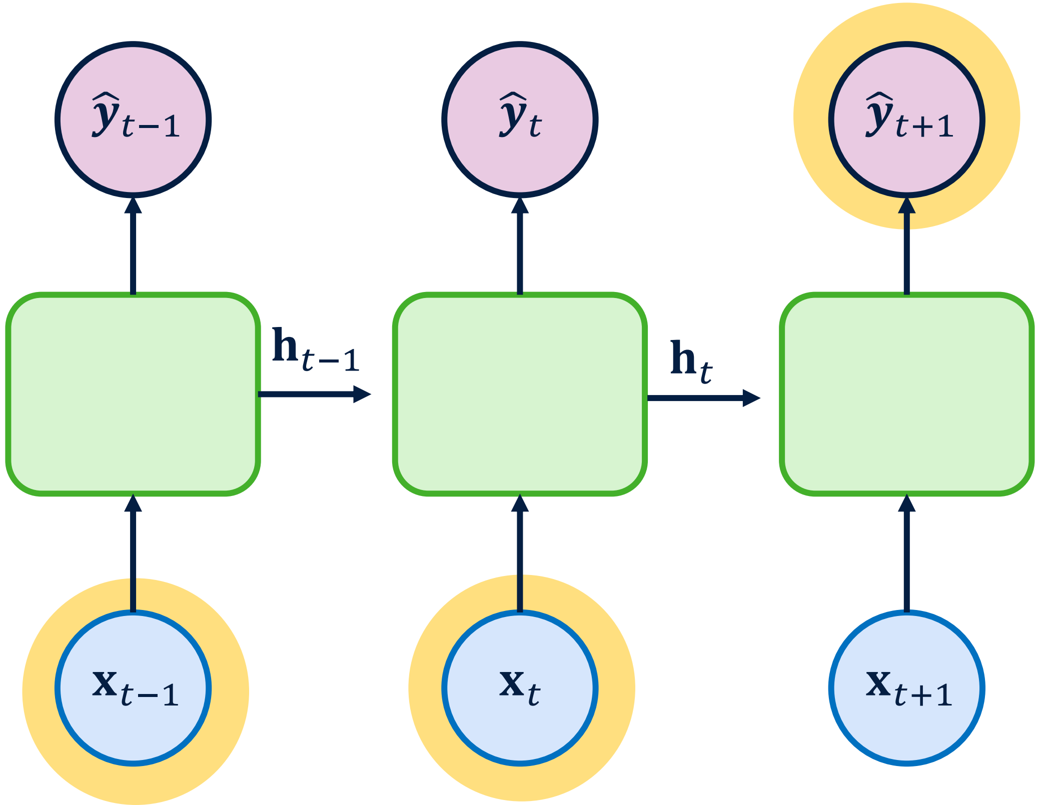


represent as a computational graph unrolled across time

$\mathbf{W_{hy}}$: hidden weights

$\mathbf{W_{hh}}$: recurrent weights

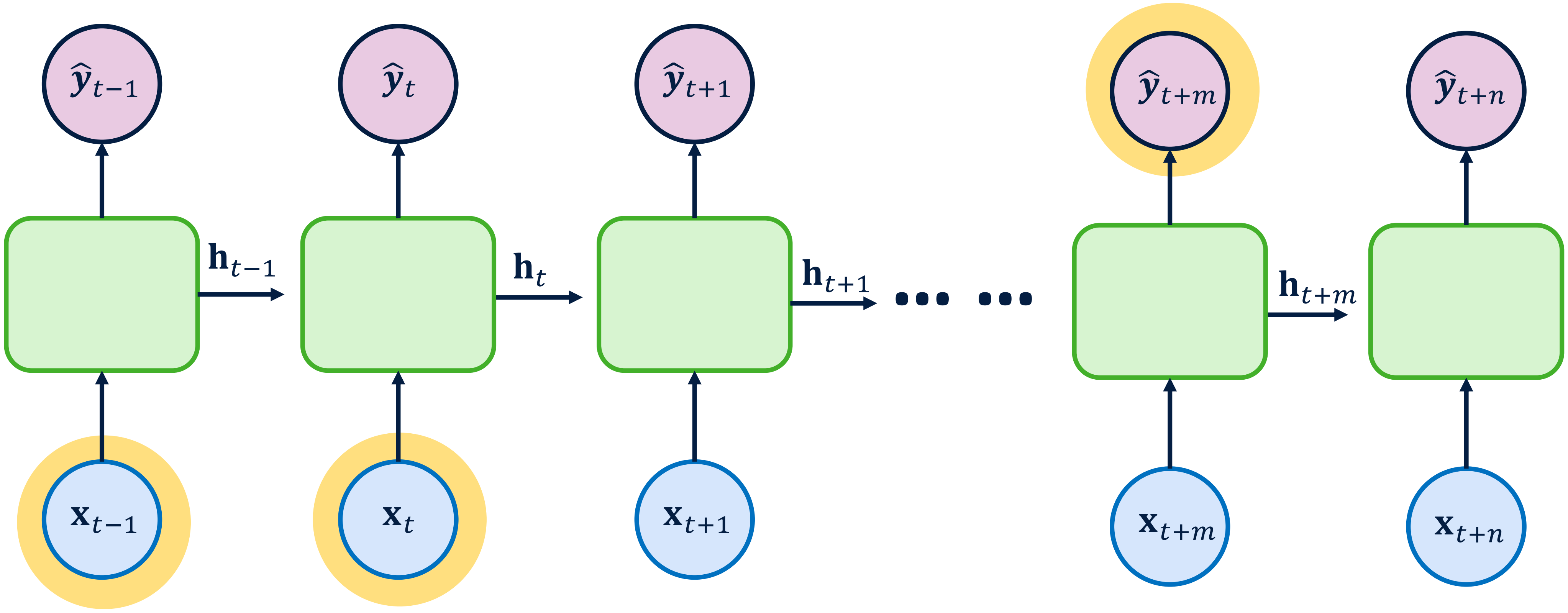
The Problem of Long-Term Dependencies



“The clouds are in the ___"

In cases certain cases, such as where the gap between the relevant information and the place that it’s needed is small, RNNs can learn to use the past information.

The Problem of Long-Term Dependencies



“I grew up in France, …… , and I speak fluent _____ ”

The Problem of Long-Term Dependencies

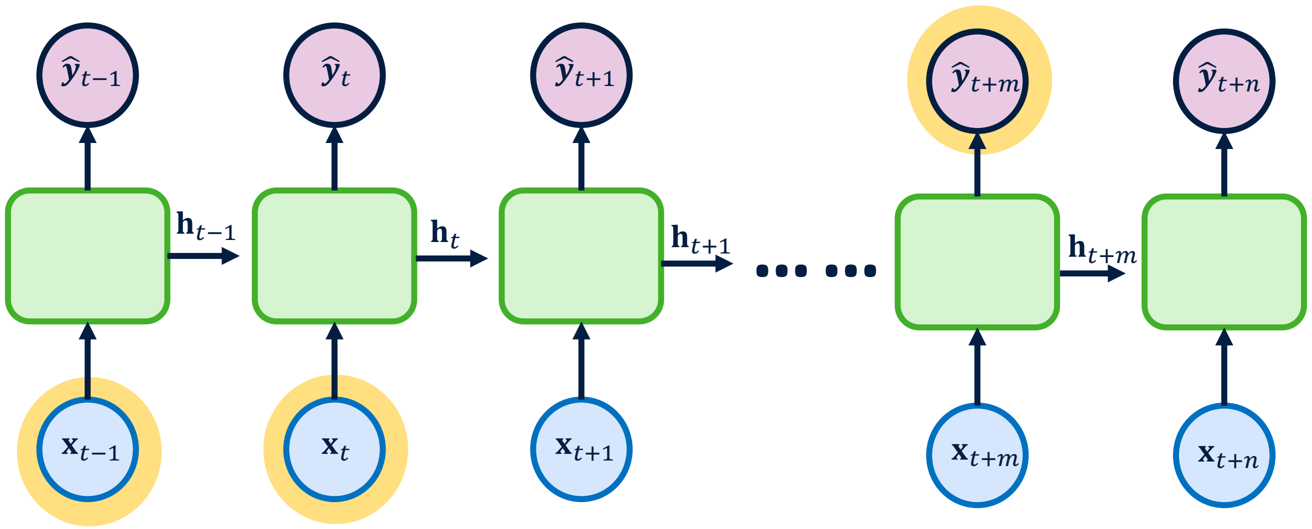
Problems of long-term information flow…

Exploding and Vanishing Gradients: Computing the gradient ∇(to train the model) w.r.t. $\mathbf{h}_t$ often requires many factors of $\mathbf{W_{hh}}$ and repeated gradient computation!

many values of:

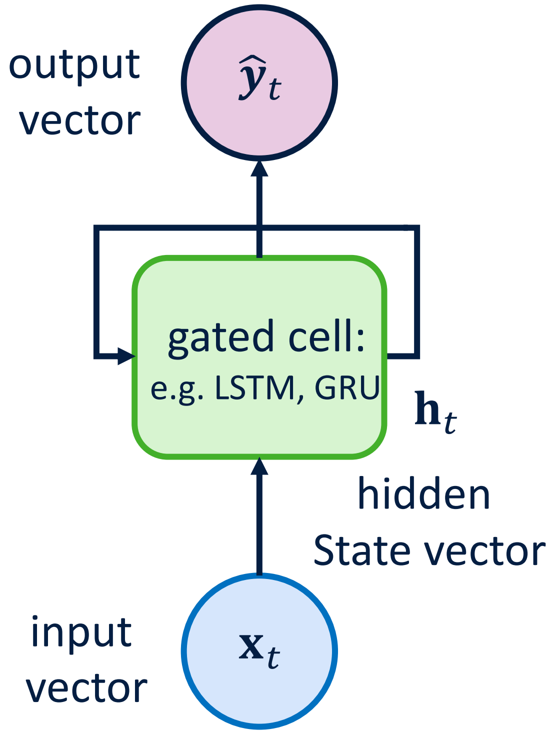
$\nabla_{\mathbf{h}_t} \gg 1$ exploding gradients

$\nabla_{\mathbf{h}_t} \ll 1$ vanishing gradients

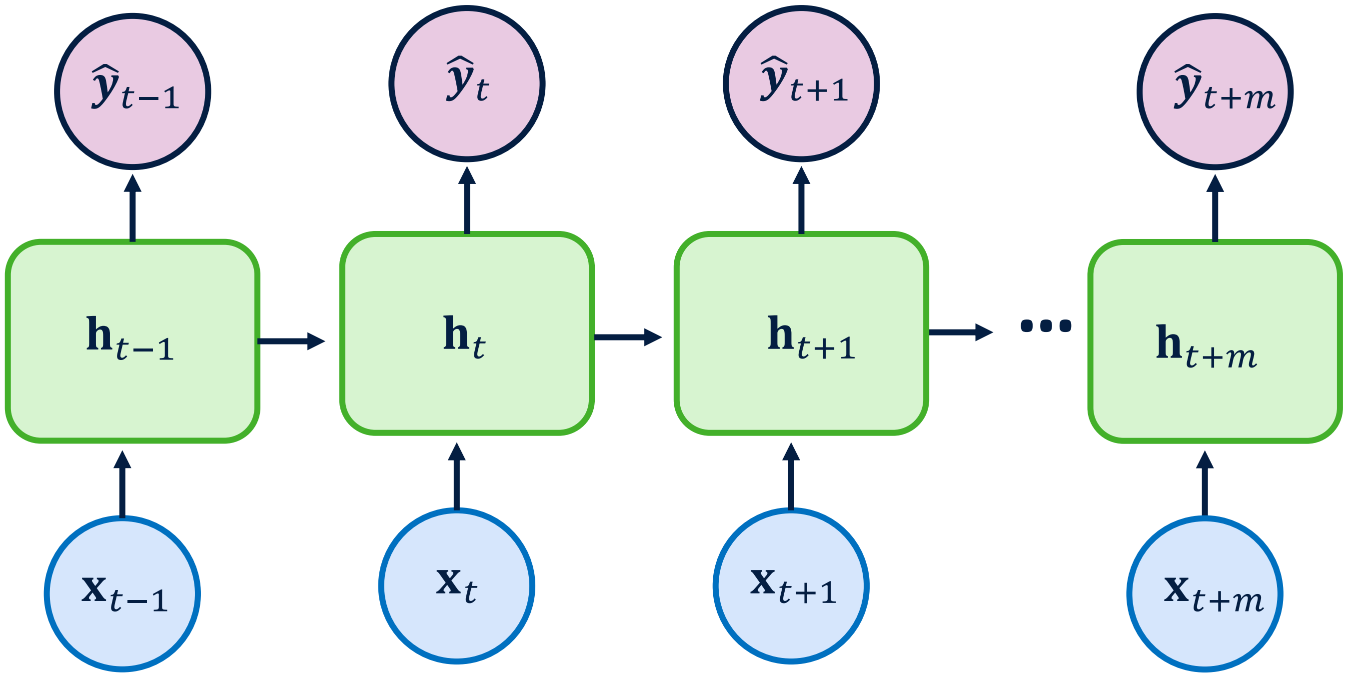


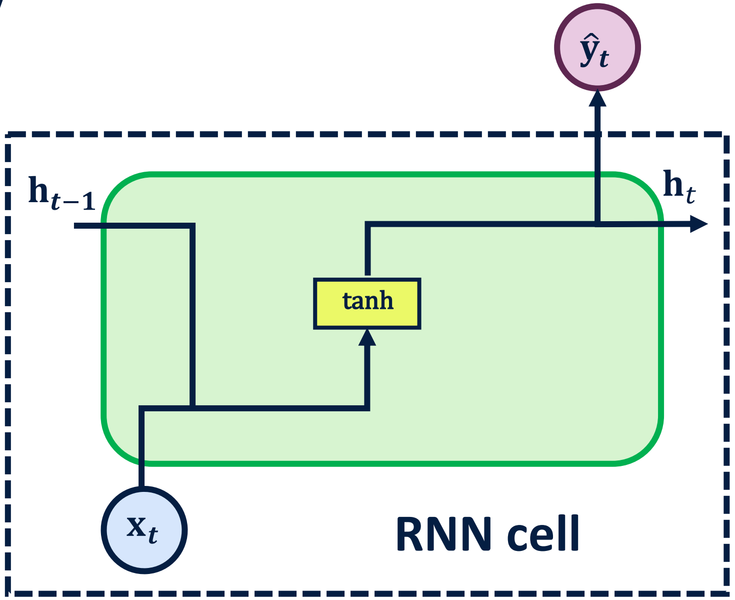
…multiply many small numbers together → errors due to further back time steps have smaller and smaller gradients → bias parameters to capture short-term dependencies

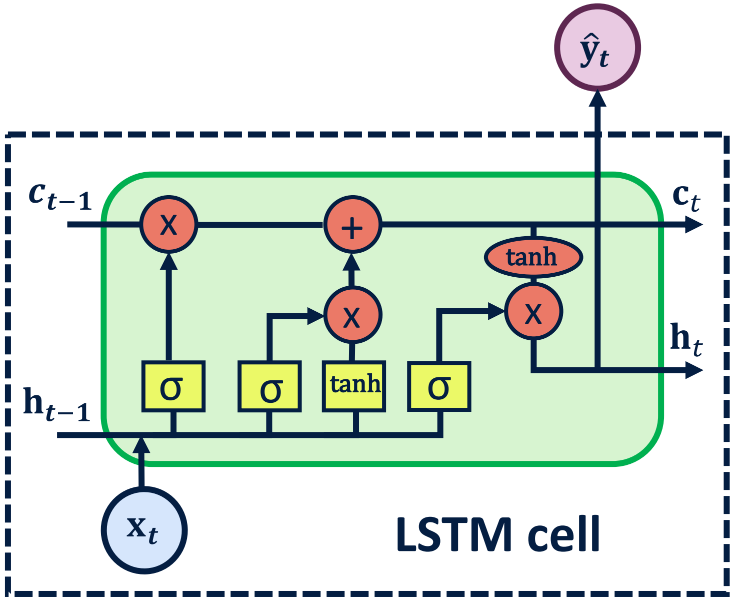
Gating to memorize long-term dependencies



Long-Short Term Memory (LSTM)

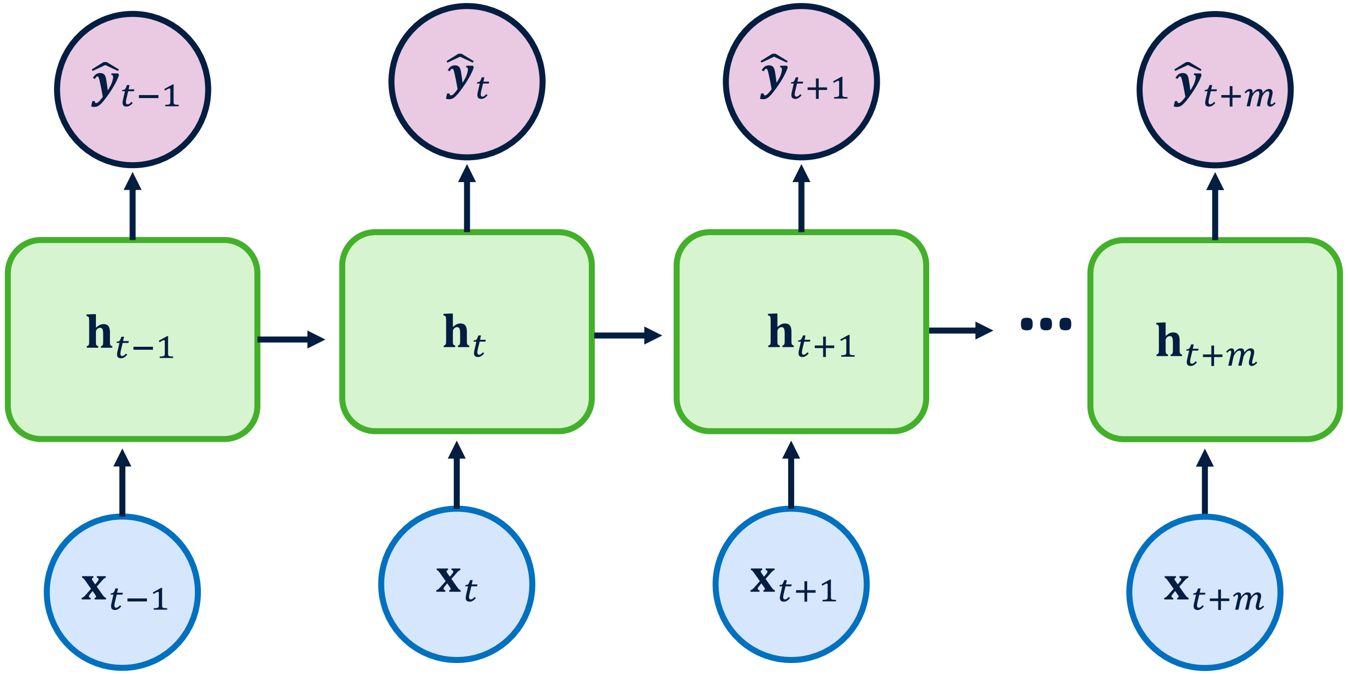


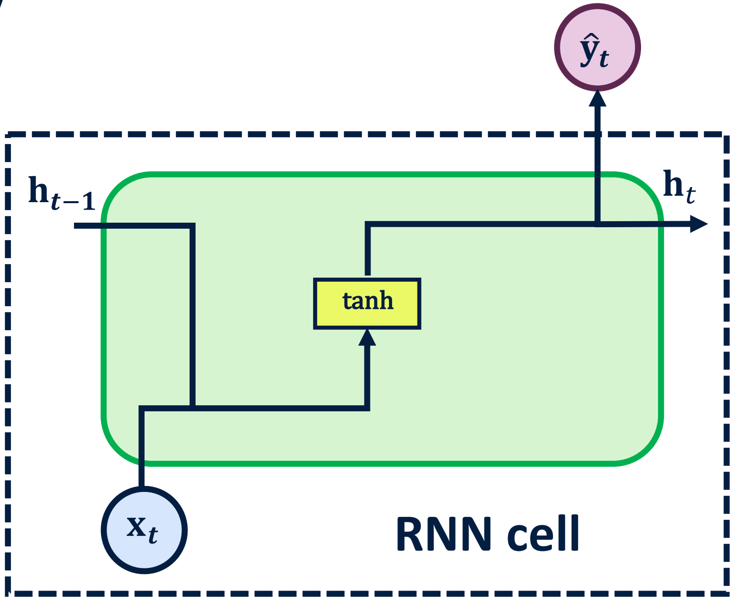


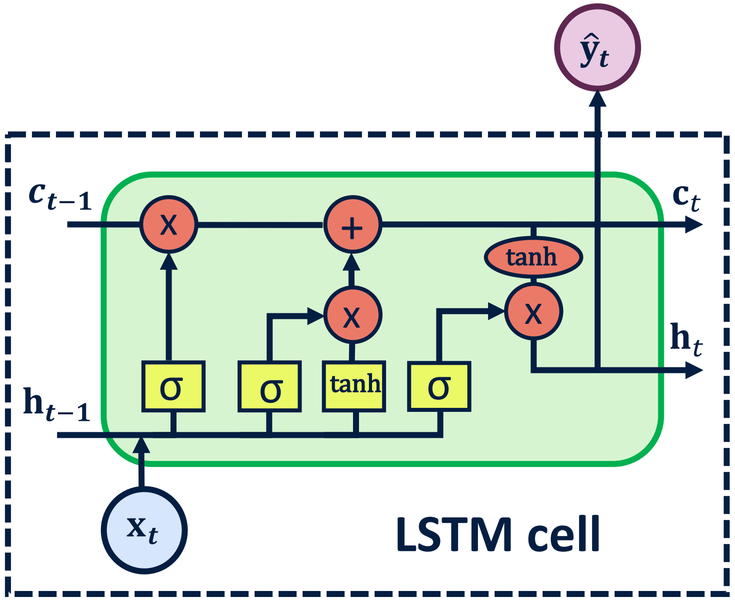


- RNN contains simple computation node
- Replace with LSTM computation block with control information flow (memory)
- LSTMS can track information over many $(t + m)$ timesteps

Long-Short Term Memory (LSTM)



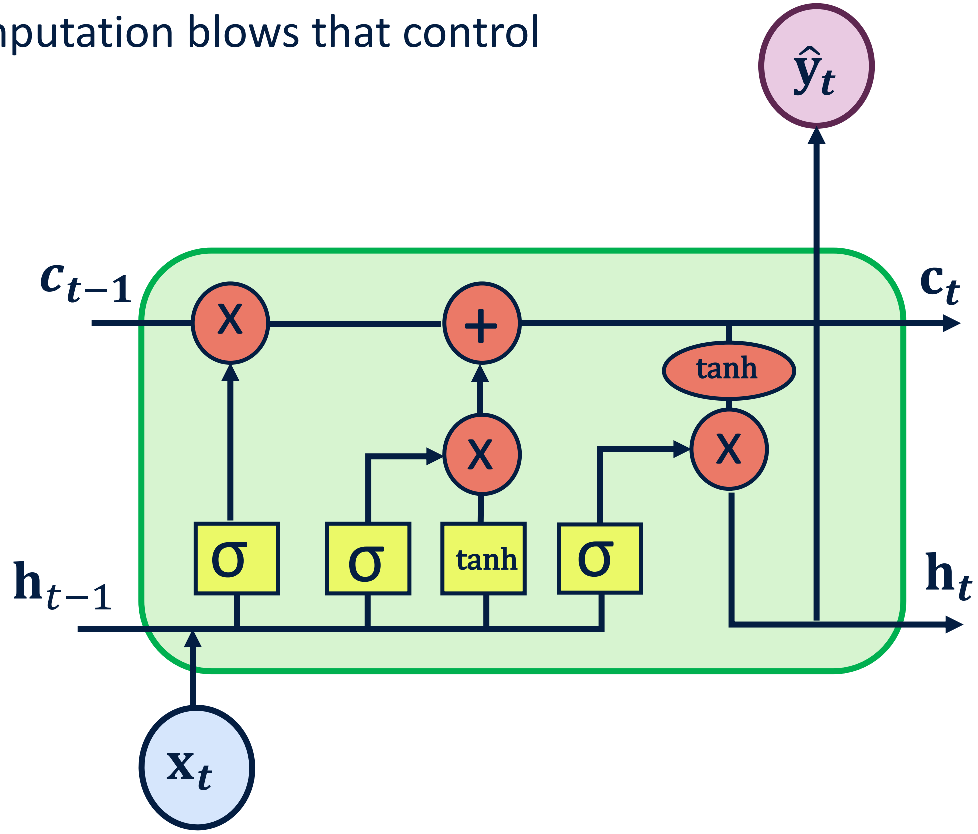




- RNN contains simple computation node
- Replace with LSTM computation block with control information flow (memory)
- LSTMS can track information over many $(t + m)$ timesteps

Long-Short Term Memory (LSTM)

LSTM modules contain computation blows that control information flow



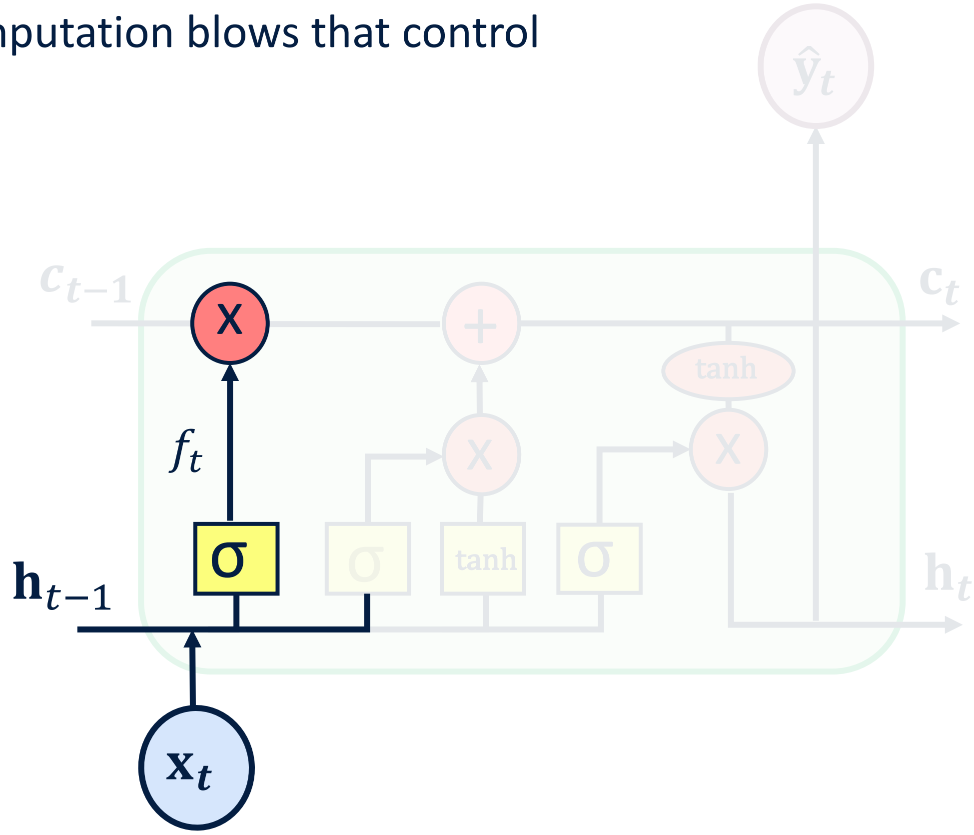
- X : pointwise multiplication
- +: pointwise addition
- σ: sigmoid function
- tanh: hyperbolic tangent function

1) Forget 2) Store 3) Update 4) Output

Long-Short Term Memory (LSTM)

LSTM modules contain computation blows that control information flow

- X : pointwise multiplication
- +: pointwise addition
- σ: sigmoid function
- tanh: hyperbolic tangent function



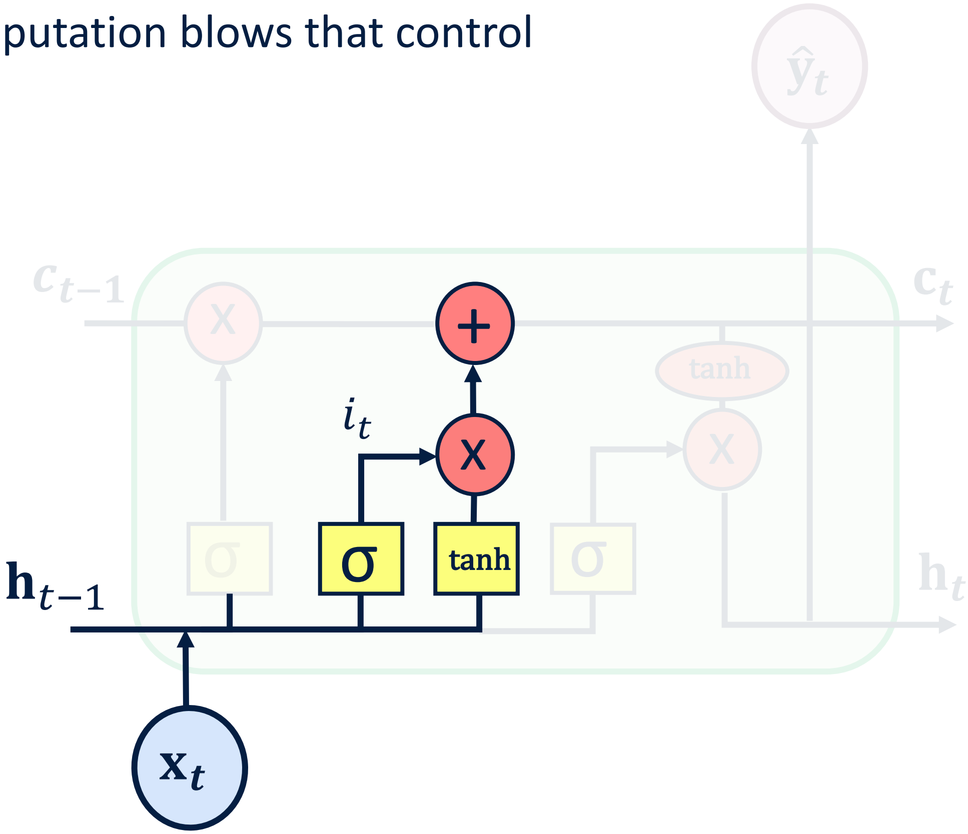
1) Forget 2) Store 3) Update 4) Output

LSTMs **forget irrelevant** parts of the previous state

Long-Short Term Memory (LSTM)

LSTM modules contain computation blows that control information flow

- X : pointwise multiplication
- +: pointwise addition
- σ: sigmoid function
- tanh: hyperbolic tangent function



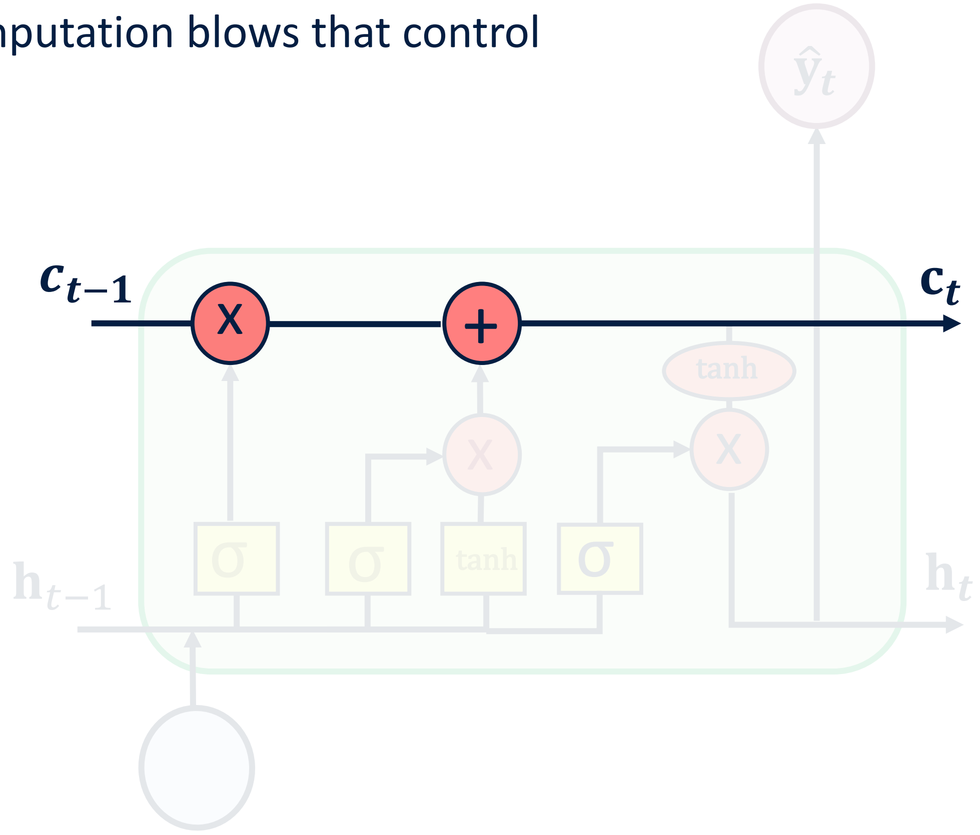
1) Forget **2) Store** 3) Update 4) Output

LSTMs **store relevant** new information into the cell state

Long-Short Term Memory (LSTM)

LSTM modules contain computation blows that control information flow

- X : pointwise multiplication
- +: pointwise addition
- σ: sigmoid function
- tanh: hyperbolic tangent function



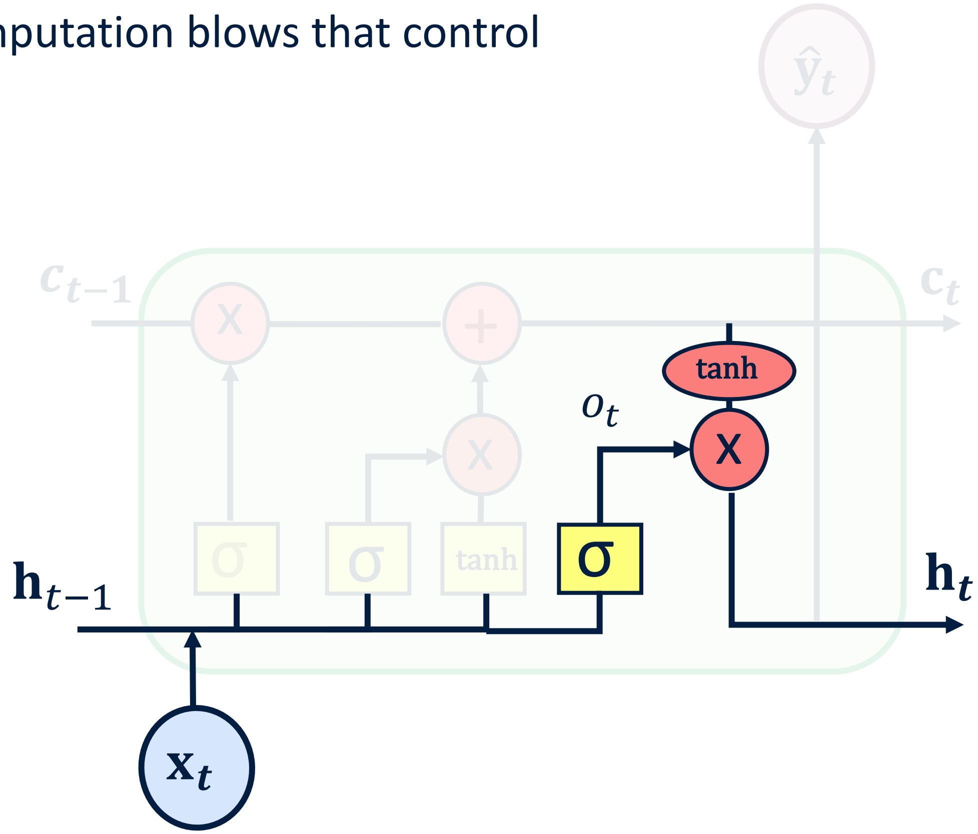
1) Forget 2) Store **3) Update** 4) Output

LSTMs **selectively update** cell state values

Long-Short Term Memory (LSTM)

LSTM modules contain computation blows that control information flow

- X : pointwise multiplication
- +: pointwise addition
- σ: sigmoid function
- tanh: hyperbolic tangent function



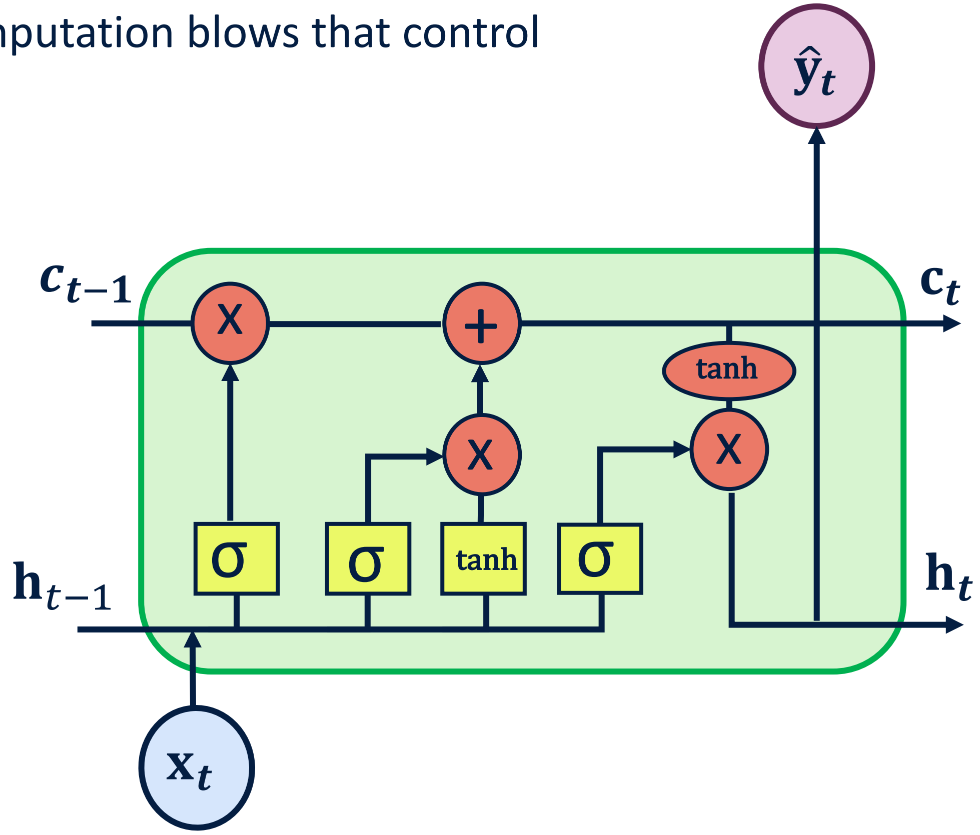
1) Forget 2) Store 3) Update **4) Output**

LSTMs **output gate** controls what information is sent to the next step, essentially returning a filtered version of the cell state

Long-Short Term Memory (LSTM)

LSTM modules contain computation blows that control information flow

- X : pointwise multiplication
- +: pointwise addition
- σ: sigmoid function
- tanh: hyperbolic tangent function

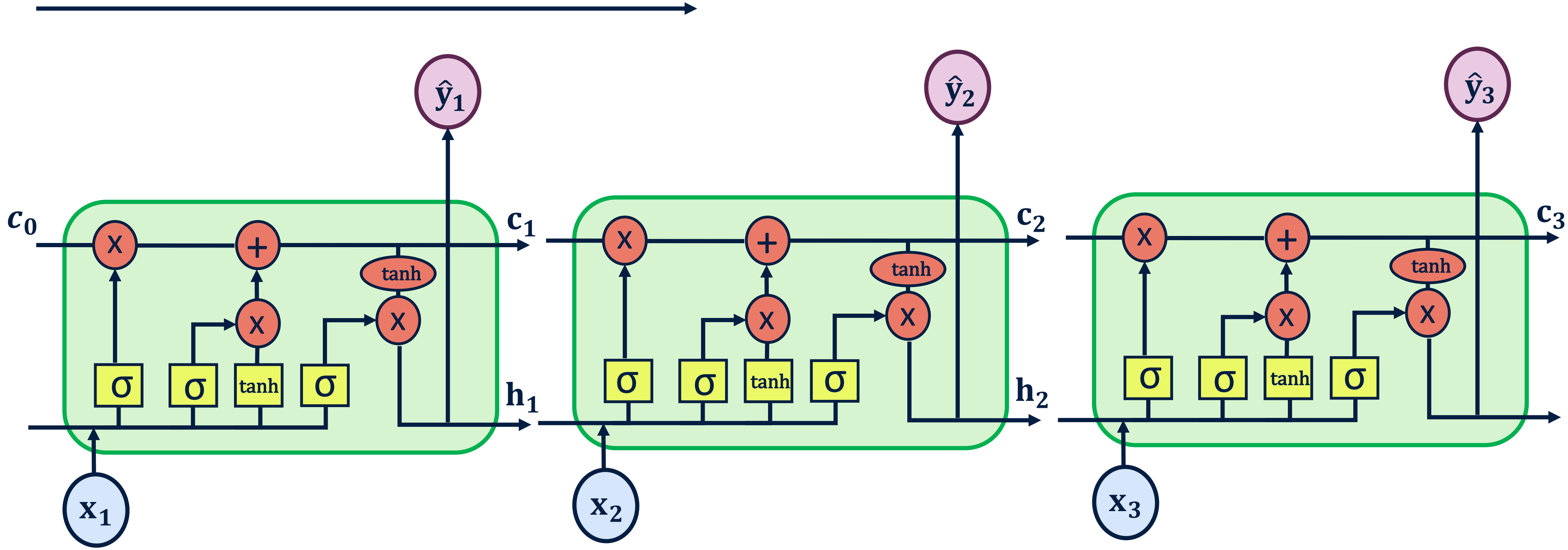


1) Forget 2) Store 3) Update 4) Output

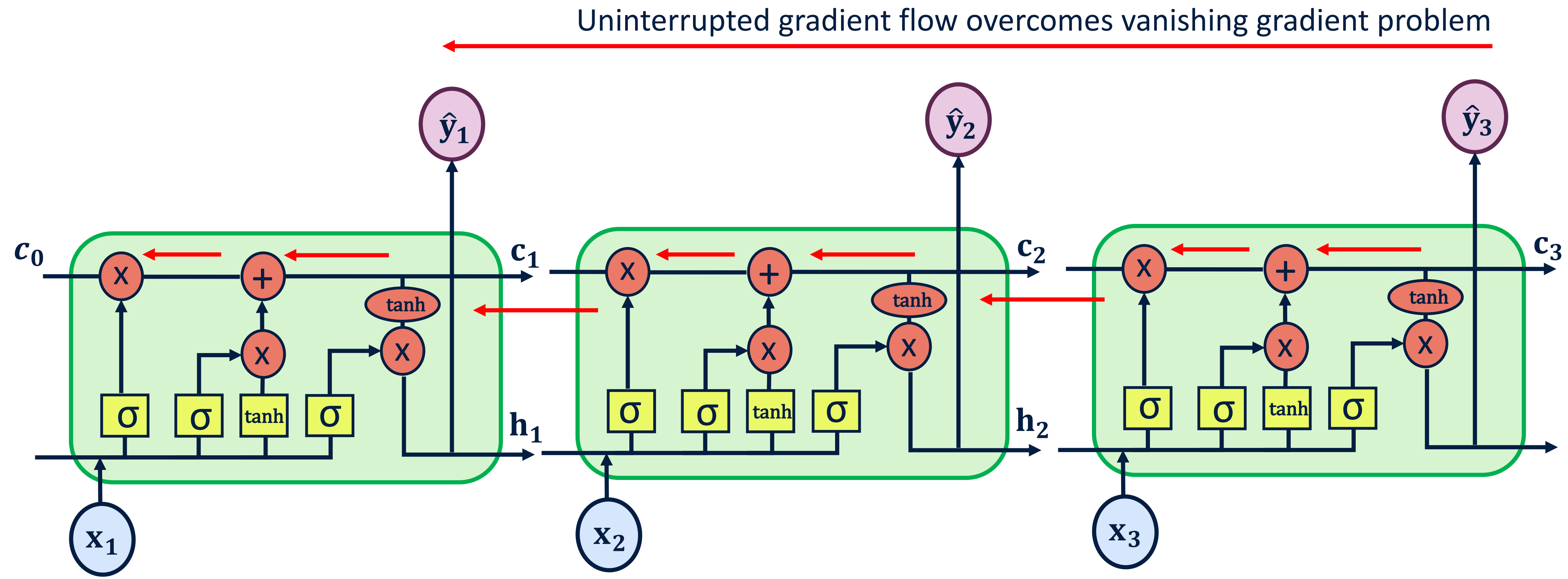
Key Concept: LSTMS maintain a separate cell state from what is outputted and uses gates to control the flow of information

Long-Short Term Memory (LSTM)

Forward pass

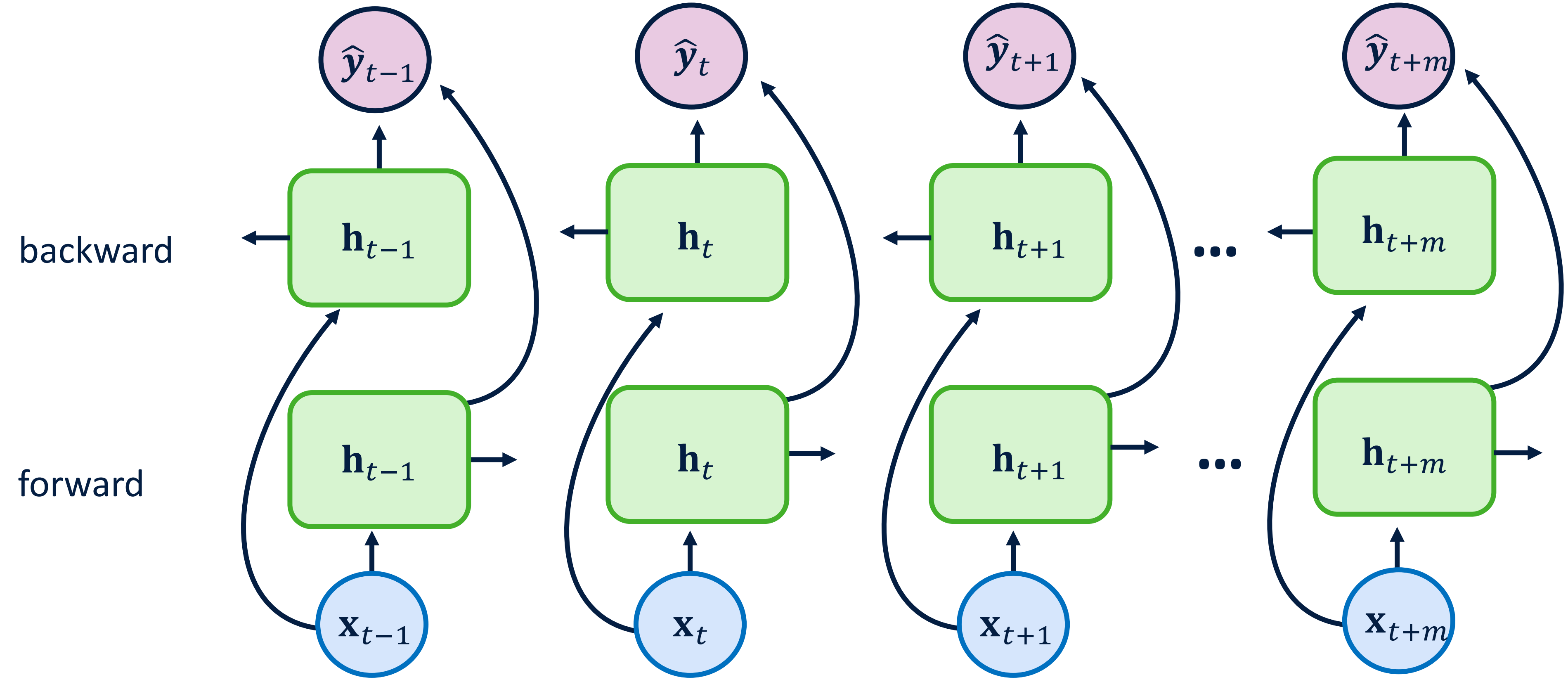


Long-Short Term Memory (LSTM)



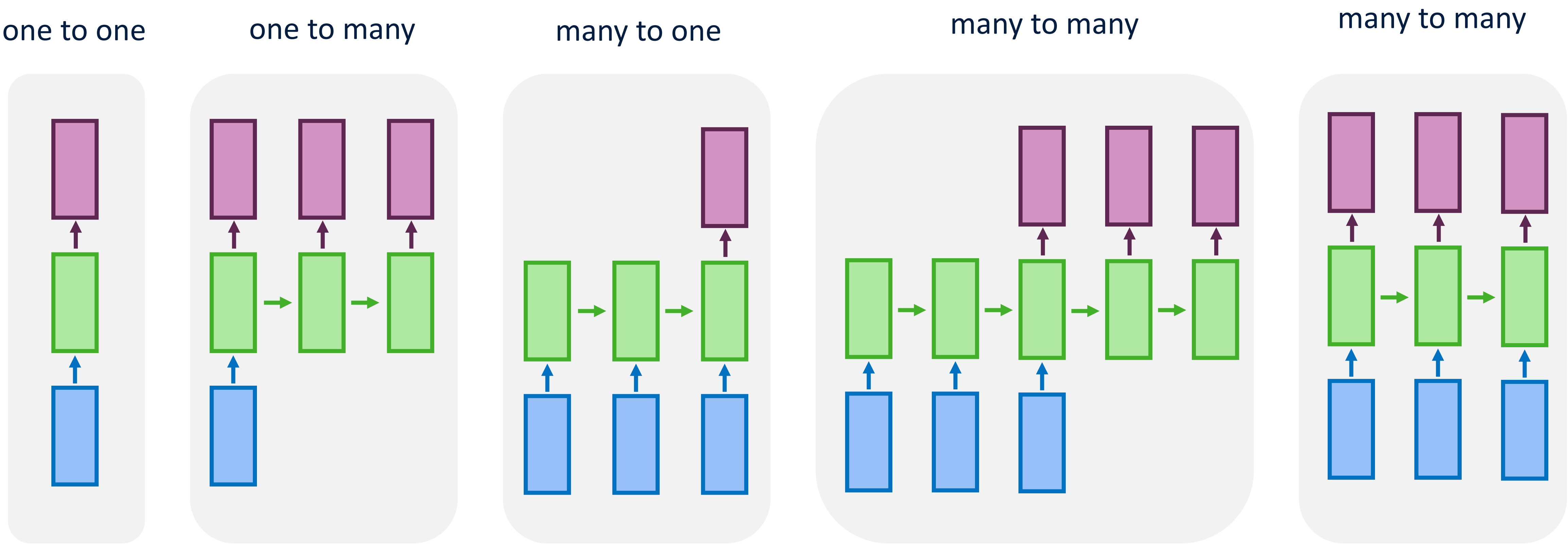
Key Concept: No multiplication with weights matrix $\mathbf{W}$ during backpropagation (don't worry about this too much!)
Multiplied by different values of forget gate → less prone to vanishing and exploding gradient problems

Bi-directional RNNs



The hidden state is the concatenated result of both forward and backward hidden states so that it can capture **past** and **future** information

Sequence Processing Options



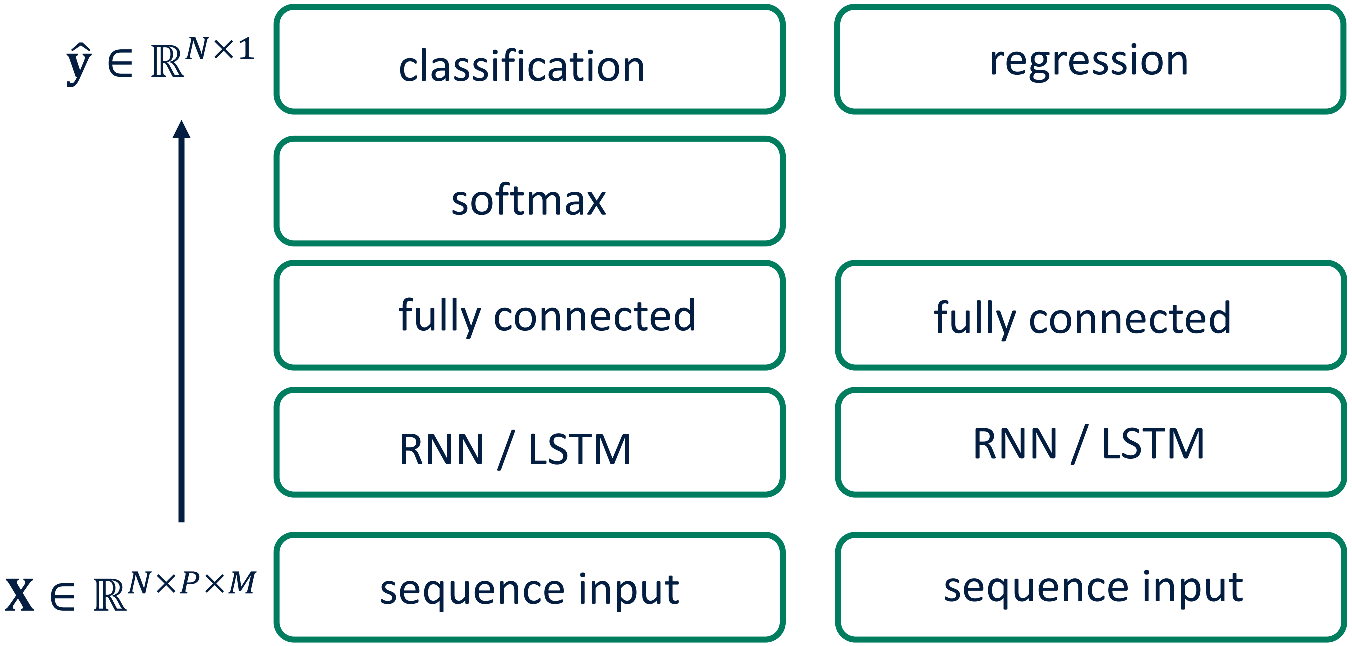
Source: https://karpathy.github.io/2015/05/21/rnn-effectiveness/

How to implement RNNs?

The practicalities:

1. Setting up your RNN
2. Picking your ML framework
3. Training your RNN
4. Avoiding overfitting
5. Your machine learning pipeline

Setting up your RNN



N: number of samples / observations.

M: number of timesteps.

P: number of features, input channels, etc.

Input data considerations:
LSTM networks support input data with varying sequence lengths.

- **Padding:** add values (usually zeros) to the start/end of sequences so that all the sequences have the same length as the longest/shortest sequence
- **Truncating:** make all sequences the same length as the shortest sequence. The remaining data in the sequences is discarded.
- **Splitting:** pads all the sequences to the nearest multiple of the specified length that is greater than the longest sequence length. Then, split each sequence into smaller sequences of the specified length

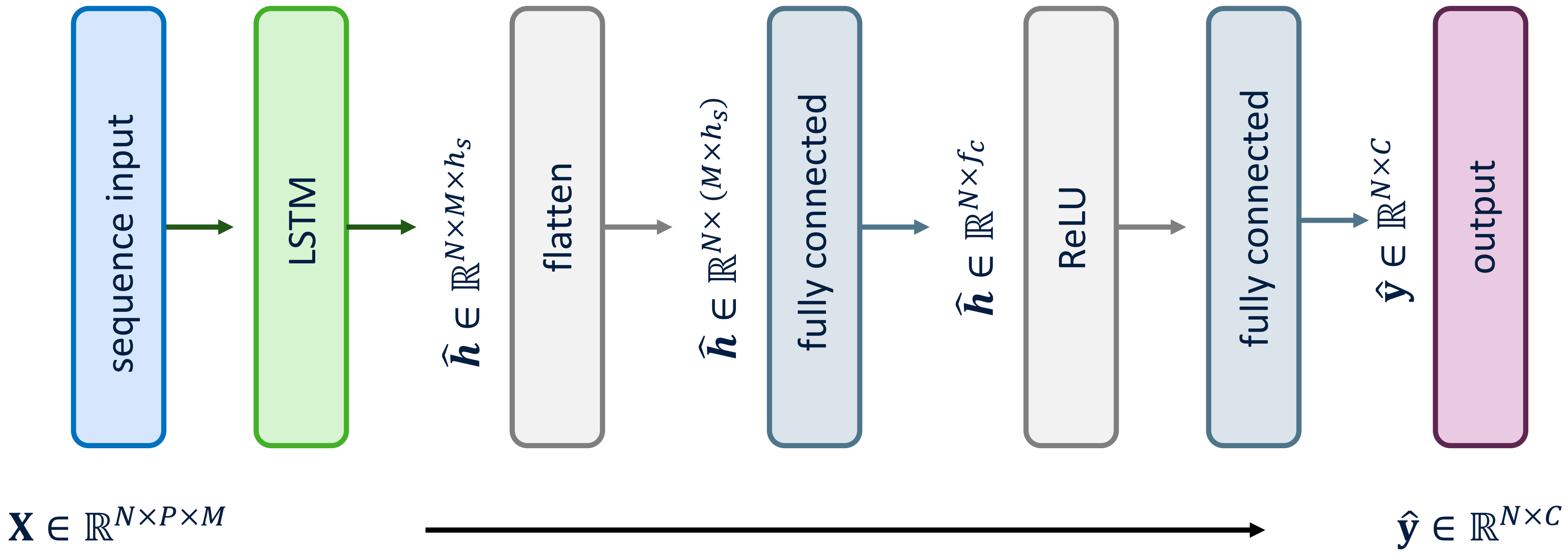
What framework to use?*

For some good comparison information:
https://www.projectpro.io/article/pytorch-vs-tensorflow-2021-a-head-to-head-comparison/416
https://medium.com/analytics-vidhya/ml03-9de2f0dbd62d

*we're using PyTorch in this workshop

LSTM example

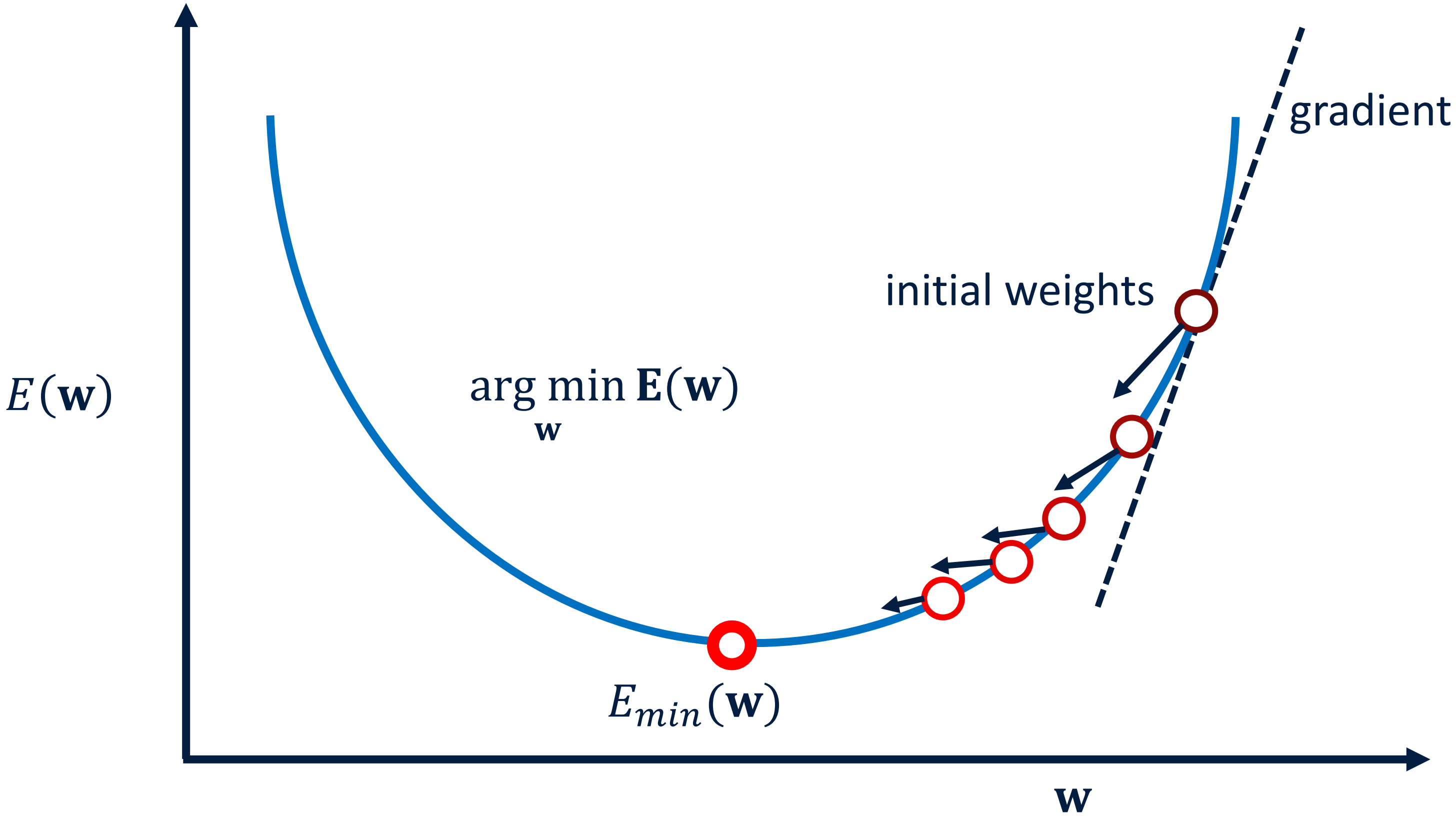
$N = 500$: number of samples / observations.
$M = 10$: number of timesteps
$P = 3$: number of features, input channels, etc.
$C = 6$: number of classes
$h_s = 128$: hidden size
$f_c = 64$: number of fully connected nodes



LSTM example code

```
import torch #pytorch
import torch.nn as nn
from torch.autograd import Variable

class Model(nn.Module):
     def __init__(self, num_classes=3, input_size=3, hidden_size=128, num_layers=1, seq_length=10):
          super(model, self).__init__()
          self.num_classes = num_classes #number of classes
          self.num_layers = num_layers #number of recurrent layers
          self.input_size = input_size #input size (# features / channels)
          self.hidden_size = hidden_size #hidden state
          self.seq_length = seq_length #sequence length

          self.lstm = nn.LSTM(input_size=input_size, hidden_size=hidden_size,
                    num_layers=num_layers, batch_first=True) #lstm
          self.fc_1 = nn.Linear(hidden_size, 100) #fully connected 1
          self.fc = nn.Linear(100, num_classes) #fully connected last layer

          self.relu = nn.ReLU()

     def forward(self,x):
          h_0 = Variable(torch.zeros(self.num_layers, x.size(0), self.hidden_size)) #hidden state
          c_0 = Variable(torch.zeros(self.num_layers, x.size(0), self.hidden_size)) #internal state
          # Propagate input through LSTM
          output, (hn, cn) = self.lstm(x, (h_0, c_0)) #lstm with input, hidden, and internal state
          hn = hn.view(-1, self.hidden_size) #reshaping the data for Dense layer next
          out = self.fc_1(hn) #first Dense
          out = self.relu(out) #relu
          out = self.fc(out) #Final Output
          return out
```

Machine Learning 101



Jacobian

gradient $\mathbf{J} = \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} = \nabla E(\mathbf{w})$

Hessian

$$\mathbf{H} = \frac{\partial^2 E(\mathbf{w})}{\partial \mathbf{w} \partial \mathbf{w}^\intercal} = \nabla^2 E(\mathbf{w})$$

We can use Newton-Raphson

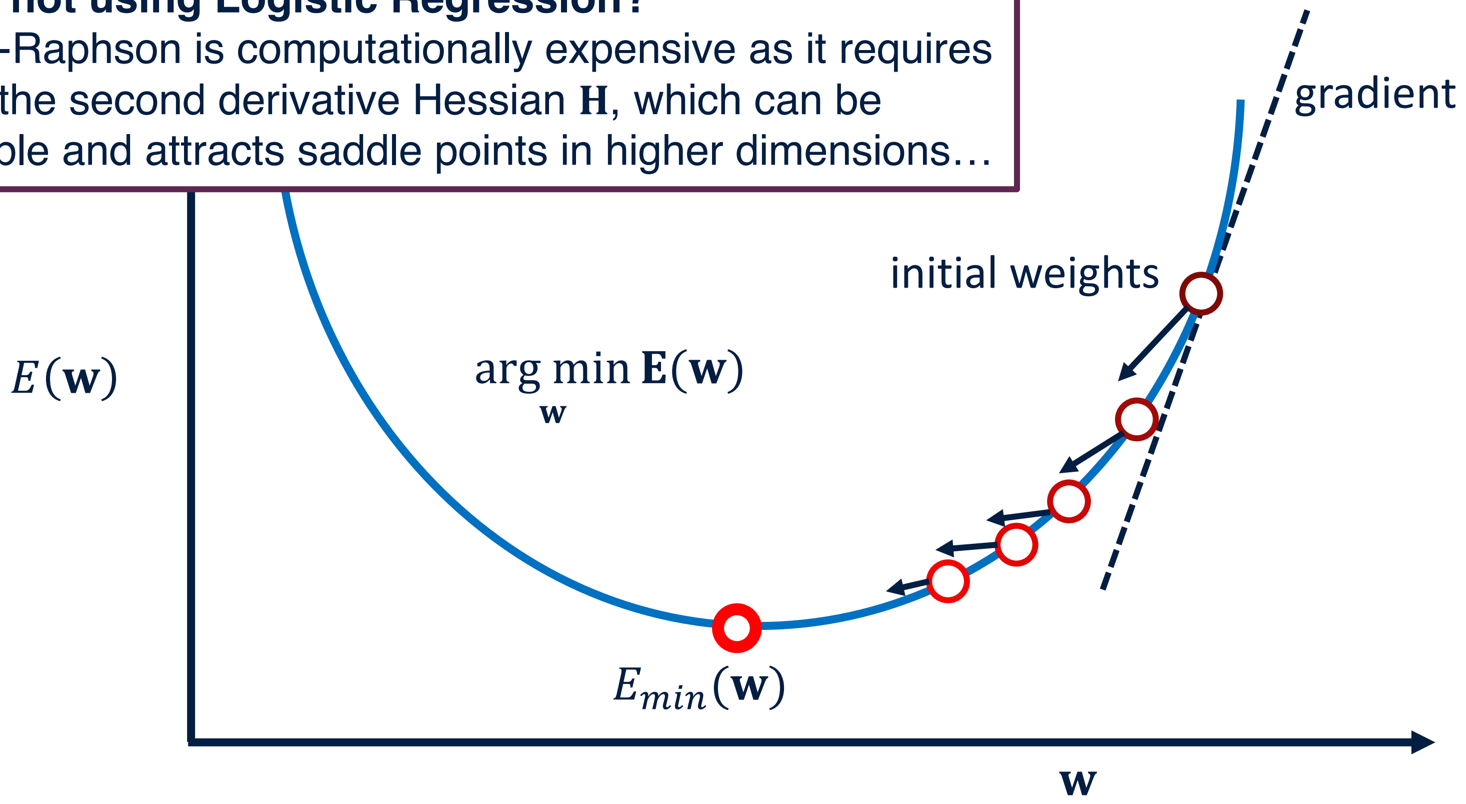
$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

Training a model, *TL;DR:*

To minimise (or maximise) an objective function, take the gradient (partial derivates) of the error function with respect to $\mathbf{w}$ and set its derivatives to zero.

Machine Learning 101

But I'm not using Logistic Regression?
Newton-Raphson is computationally expensive as it requires solving the second derivative Hessian $\mathbf{H}$, which can be intractable and attracts saddle points in higher dimensions…



Jacobian

$$\mathbf{J} = \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} = \nabla E(\mathbf{w})$$

Hessian

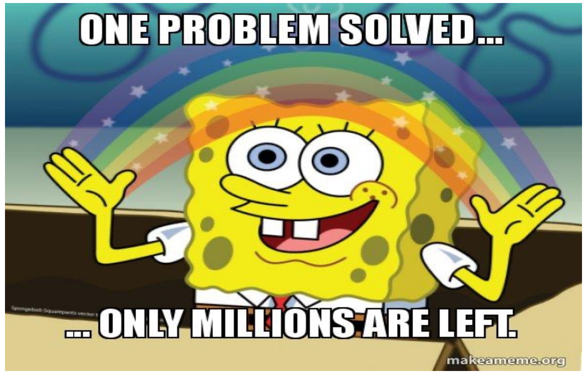
$$\mathbf{H} = \frac{\partial^2 E(\mathbf{w})}{\partial \mathbf{w} \partial \mathbf{w}^\intercal} = \nabla^2 E(\mathbf{w})$$

We can use Newton-Raphson

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \frac{\nabla_{\mathbf{w}_t} E(\mathbf{w}_t)}{\nabla^2_{\mathbf{w}_t} E(\mathbf{w})}$$

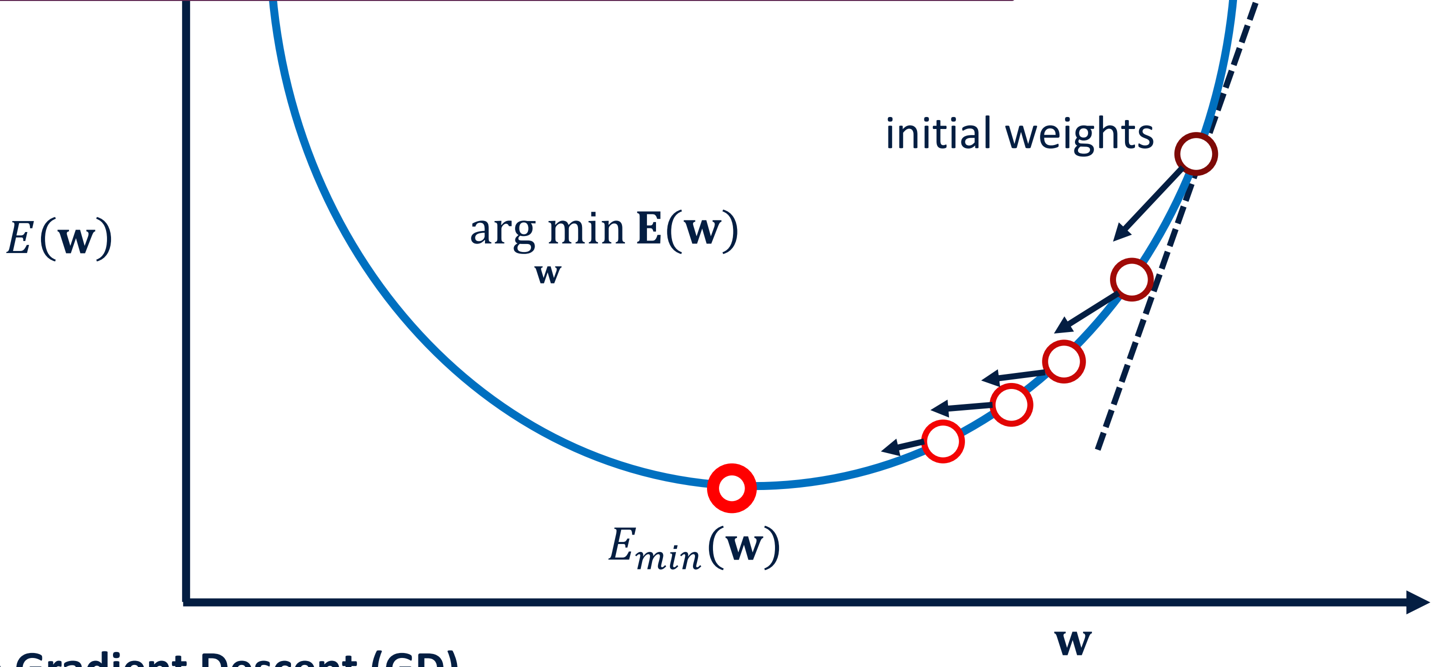
Training a model, *TL;DR:*
To minimise (or maximise) an objective function, take the gradient (partial derivates) of the error function with respect to $\mathbf{w}$ and set its derivatives to zero.

Machine Learning 101



Solution

- Take repeated small steps t in the direction of steepest decrease of $E(\mathbf{w})$ ...



Jacobian

gradient $\mathbf{J} = \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} = \nabla E(\mathbf{w})$

~~**Hessian**~~

$$\cancel{\mathbf{H} = \frac{\partial^2 E(\mathbf{w})}{\partial \mathbf{w} \partial \mathbf{w}^T}} = \nabla^2 E(\mathbf{w})$$

We can use **Gradient Descent (GD)**

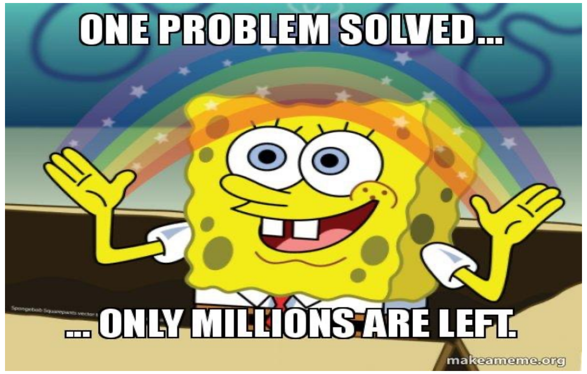
$$\boxed{\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \eta_t \nabla_{\mathbf{w}_t} E(\mathbf{w}_t)}$$

Where η is the learning rate, $\eta > 0$ determining the size of the step

Training a model, *TL;DR:*

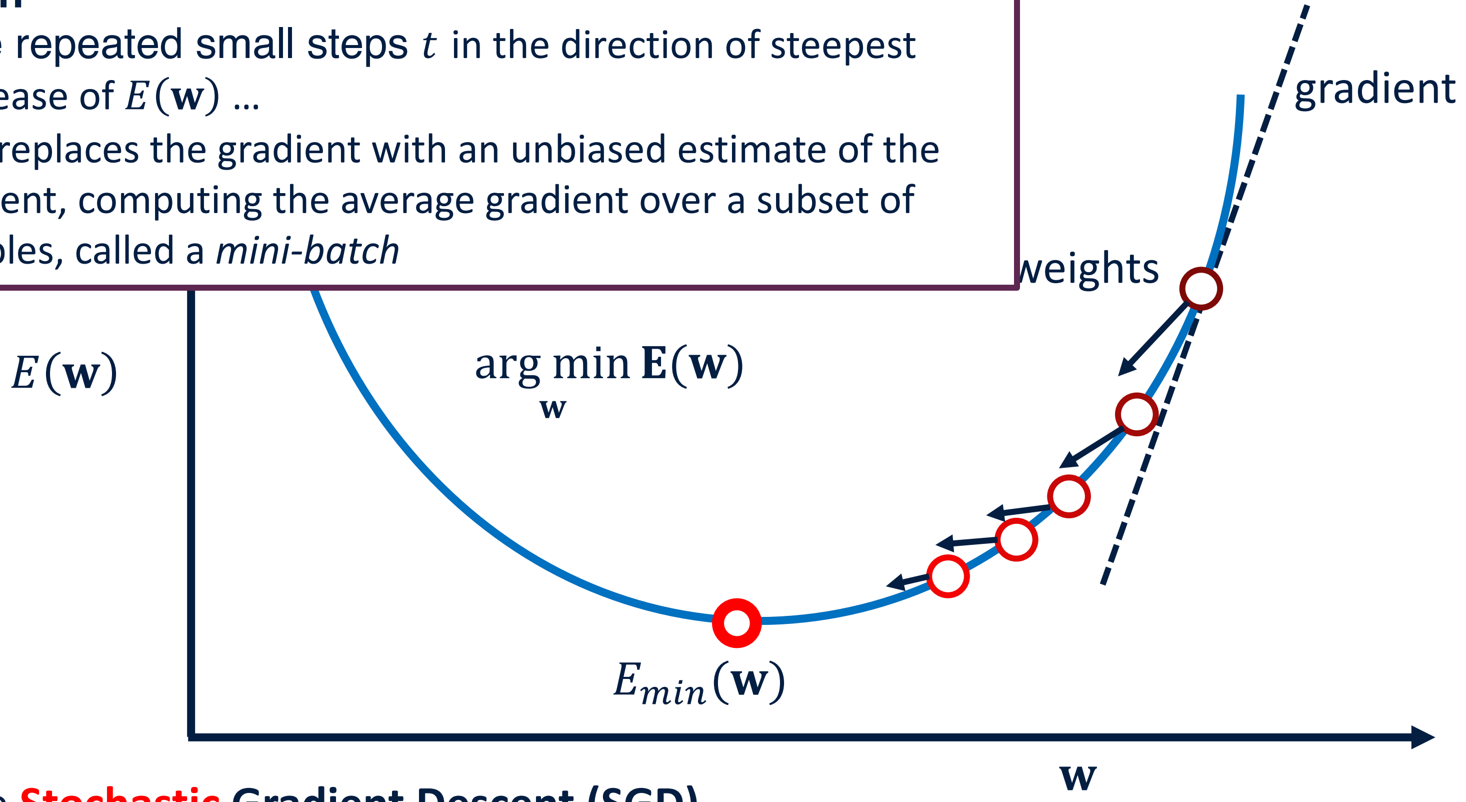
To minimise (or maximise) an objective function, take the gradient (partial derivates) of the error function with respect to $\mathbf{w}$ and set its derivatives to zero.

Machine Learning 101



Solution

- Take repeated small steps t in the direction of steepest decrease of $E(\mathbf{w})$ …
- SGD replaces the gradient with an unbiased estimate of the gradient, computing the average gradient over a subset of samples, called a *mini-batch*



Jacobian

gradient $\mathbf{J} = \dfrac{\partial E(\mathbf{w})}{\partial \mathbf{w}} = \nabla E(\mathbf{w})$

~~**Hessian**~~

$$\cancel{\mathbf{H} = \frac{\partial^2 E(\mathbf{w})}{\partial \mathbf{w} \partial \mathbf{w}^\intercal} = \nabla^2 E(\mathbf{w})}$$

We can use **Stochastic Gradient Descent (SGD)**

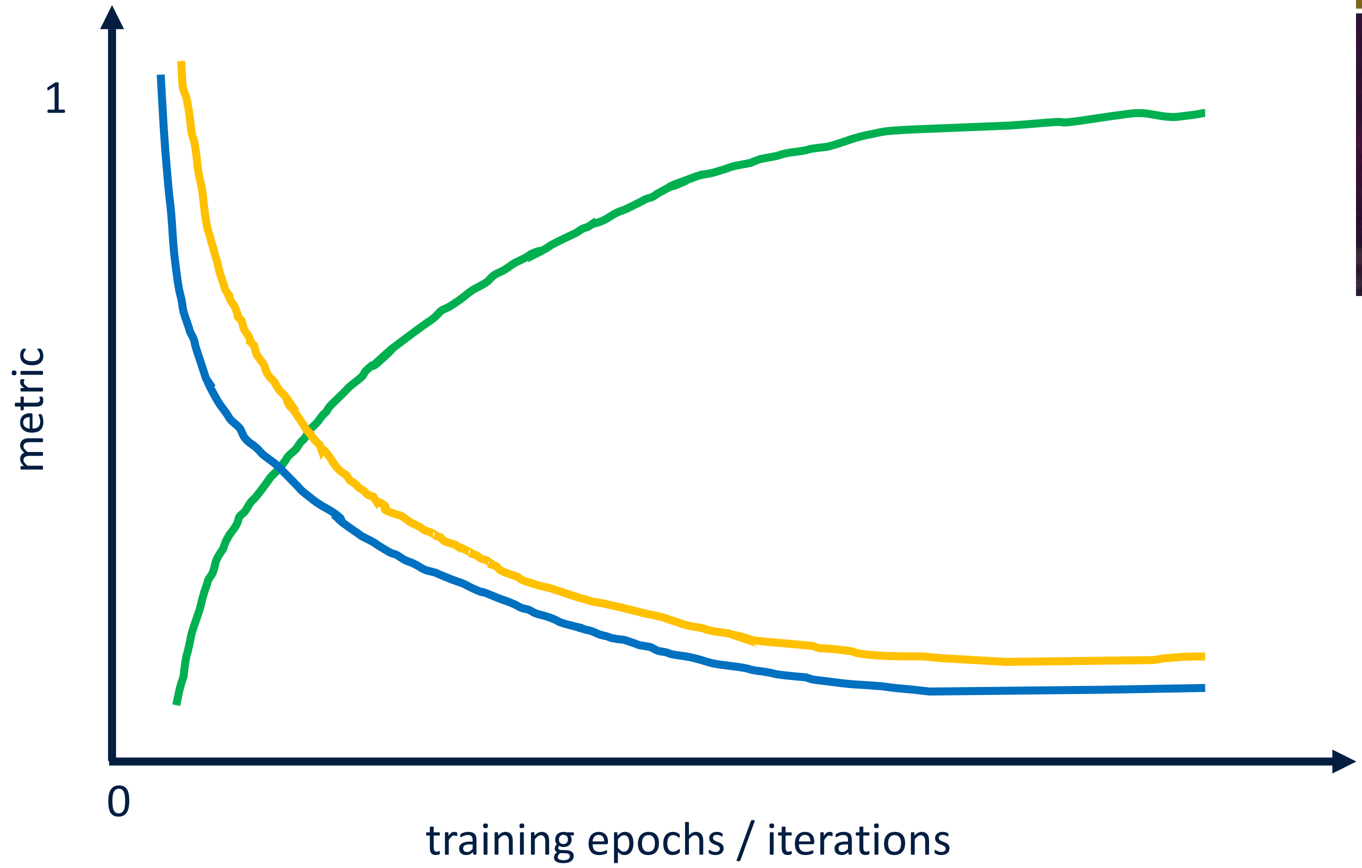
$$\boxed{\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \eta_t \frac{1}{m} \nabla_{\mathbf{w}_t} \sum_{i=1}^{m} E(\mathbf{w}_t)}$$

Where η is the learning rate, $\eta > 0$ determining the size of the step

Training a model, *TL;DR:*

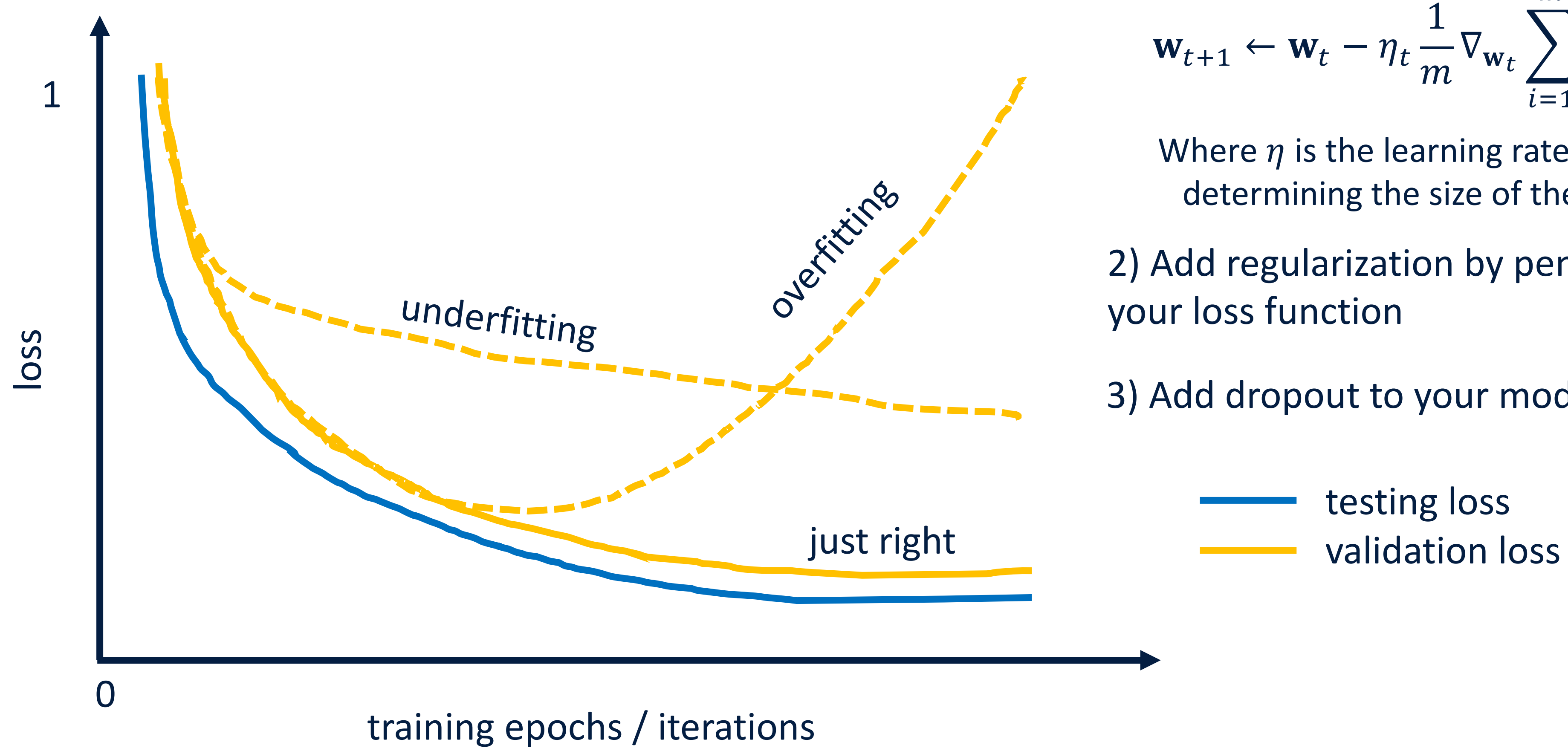
To minimise (or maximise) an objective function, take the gradient (partial derivates) of the error function with respect to $\mathbf{w}$ and set its derivatives to zero.

How to train your model



Waiting on your model to train

Avoiding Overfitting



Tips:

1) adjust your learning rate

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \eta_t \frac{1}{m} \nabla_{\mathbf{w}_t} \sum_{i=1}^{m} E(\mathbf{w}_t)$$

Where η is the learning rate, $\eta > 0$ determining the size of the step

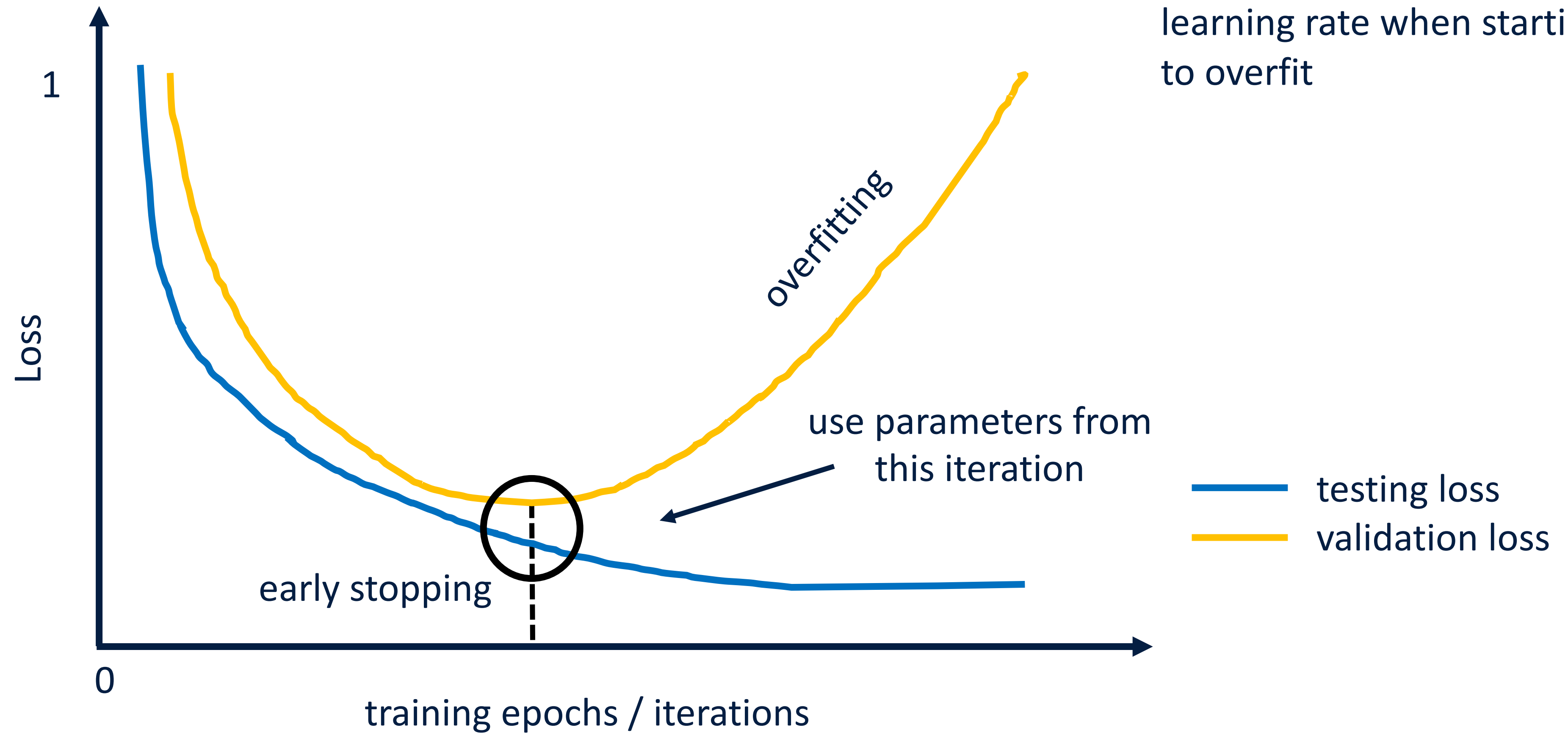
2) Add regularization by penalizing your loss function

3) Add dropout to your model

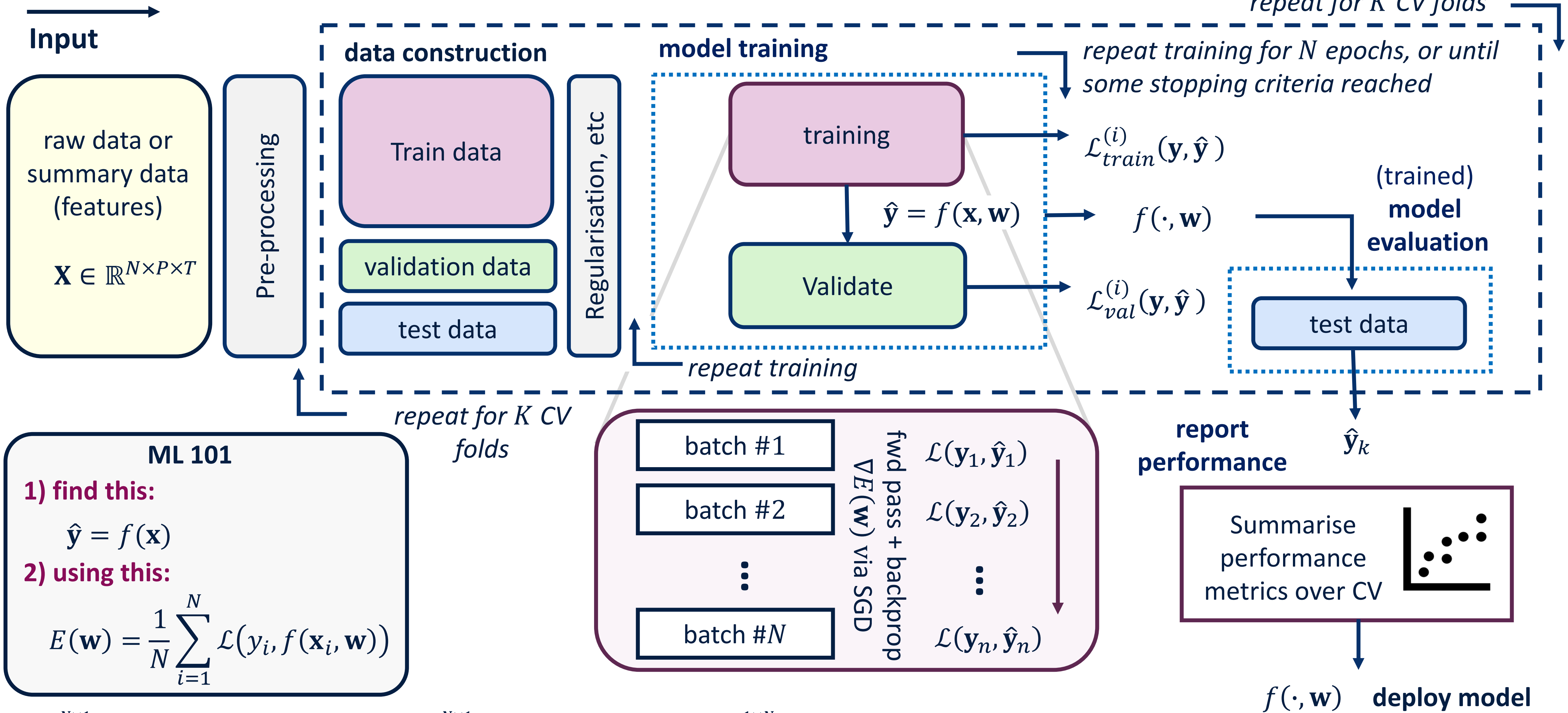
Avoiding Overfitting

Tips:

4) Add early stopping criteria
5) Dynamically lower your learning rate when starting to overfit



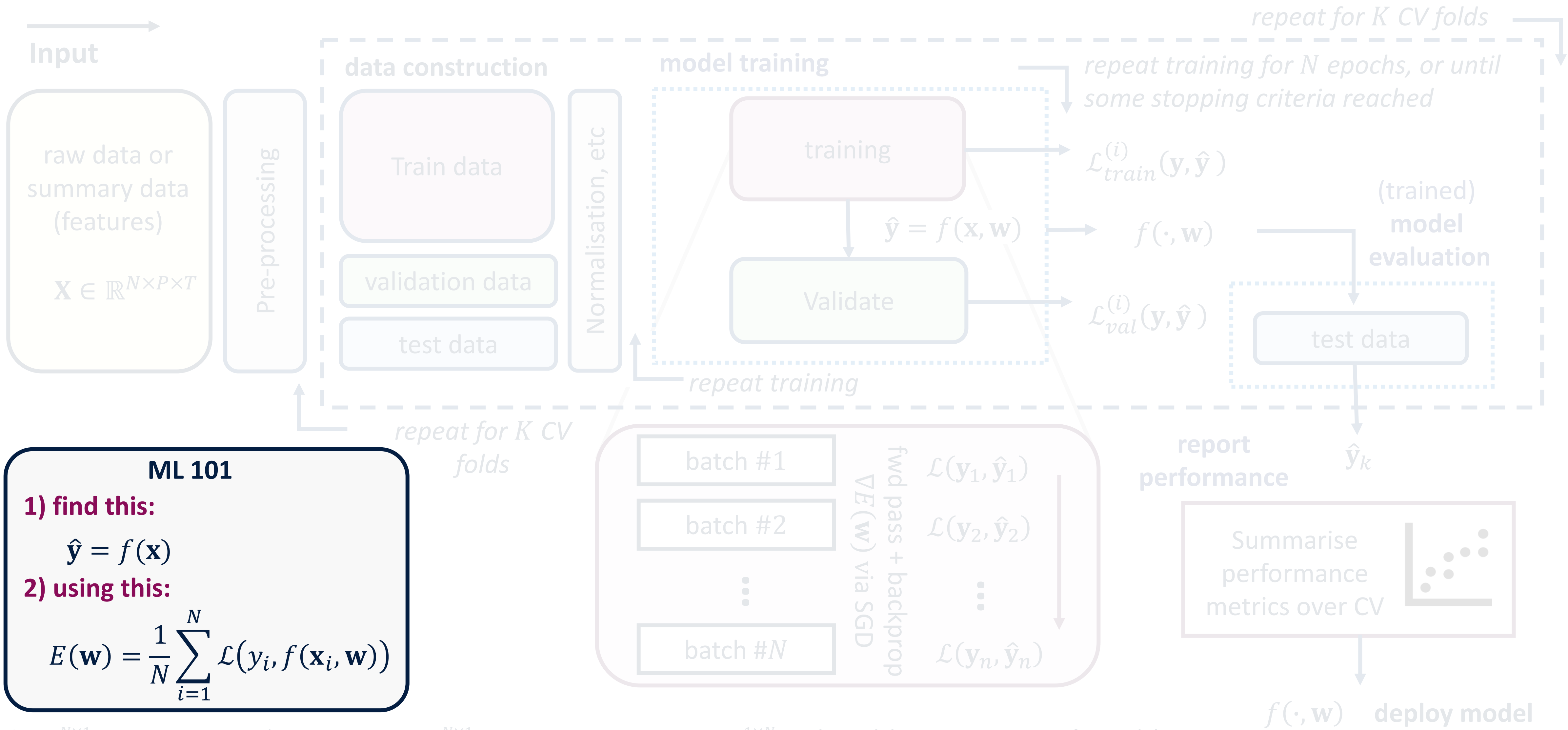
Typical Machine Learning Pipeline



$\hat{\mathbf{y}} \in \mathbb{R}^{N \times 1}$ are the predictions / estimations; $\mathbf{y} \in \mathbb{R}^{N \times 1}$ are the response labels; $\mathbf{w} \in \mathbb{R}^{1 \times N}$ are the weights or parameters of a model

N: number of samples / observations; T: number of timesteps; P: number of features, input channels, etc.

Typical Machine Learning Pipeline



ML 101

1) find this:

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

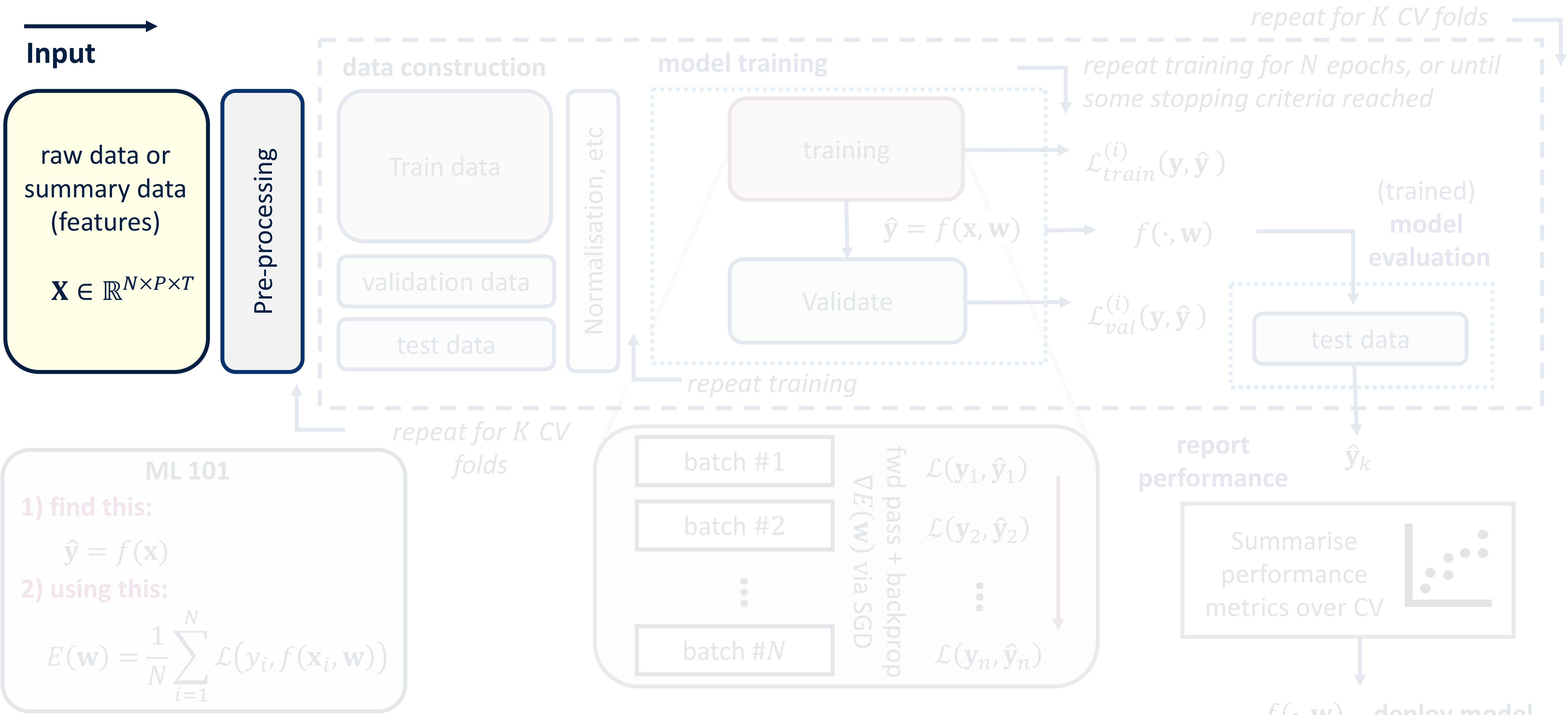
2) using this:

$$E(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(y_i, f(\mathbf{x}_i, \mathbf{w}))$$

$\hat{\mathbf{y}} \in \mathbb{R}^{N \times 1}$ are the predictions / estimations; $\mathbf{y} \in \mathbb{R}^{N \times 1}$ are the response labels; $\mathbf{w} \in \mathbb{R}^{1 \times N}$ are the weights or parameters of a model

N: number of samples / observations; M: number of timesteps; P: number of features, input channels, etc.

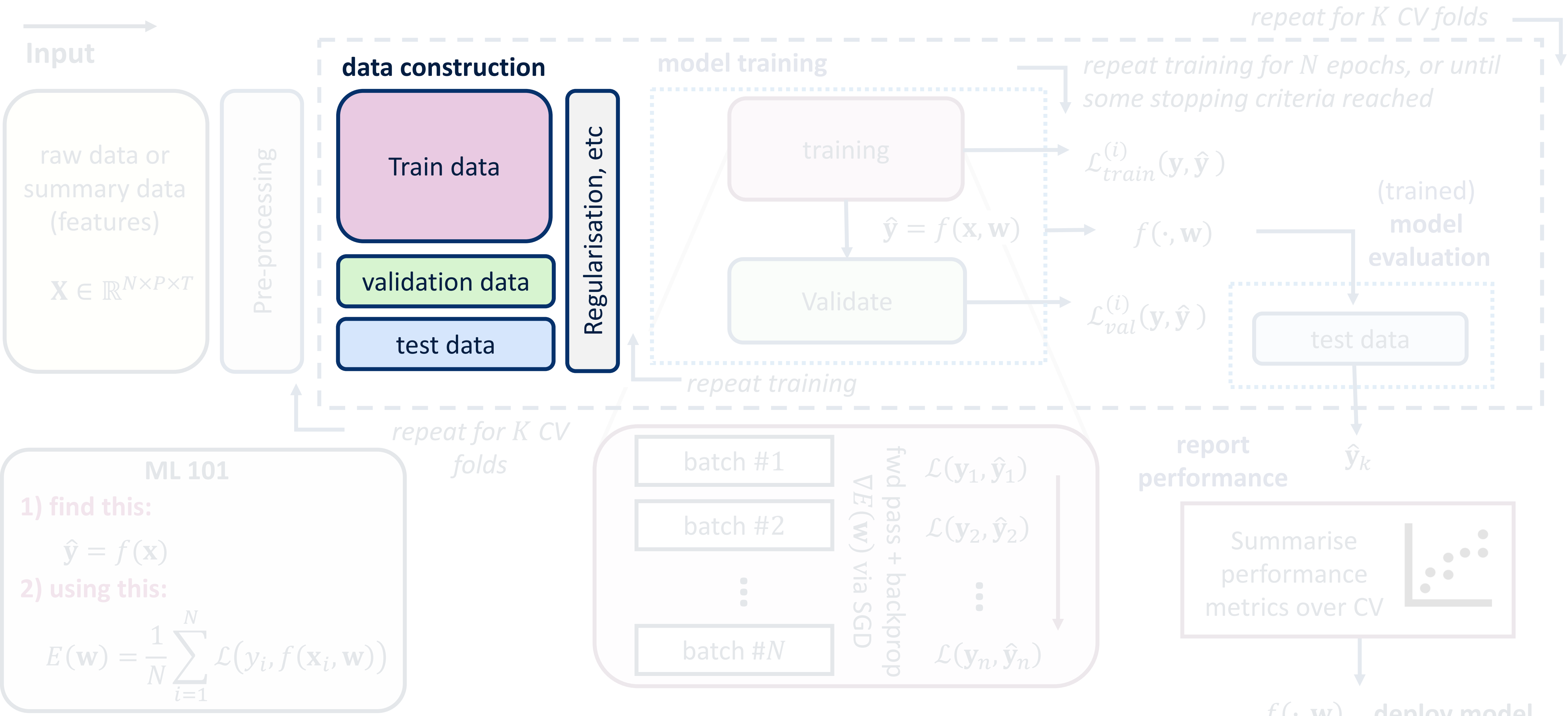
Typical Machine Learning Pipeline



$\hat{\mathbf{y}} \in \mathbb{R}^{N\times 1}$ are the predictions / estimations; $\mathbf{y} \in \mathbb{R}^{N\times 1}$ are the response labels; $\mathbf{w} \in \mathbb{R}^{1\times N}$ are the weights or parameters of a model

N: number of samples / observations; M: number of timesteps; P: number of features, input channels, etc.

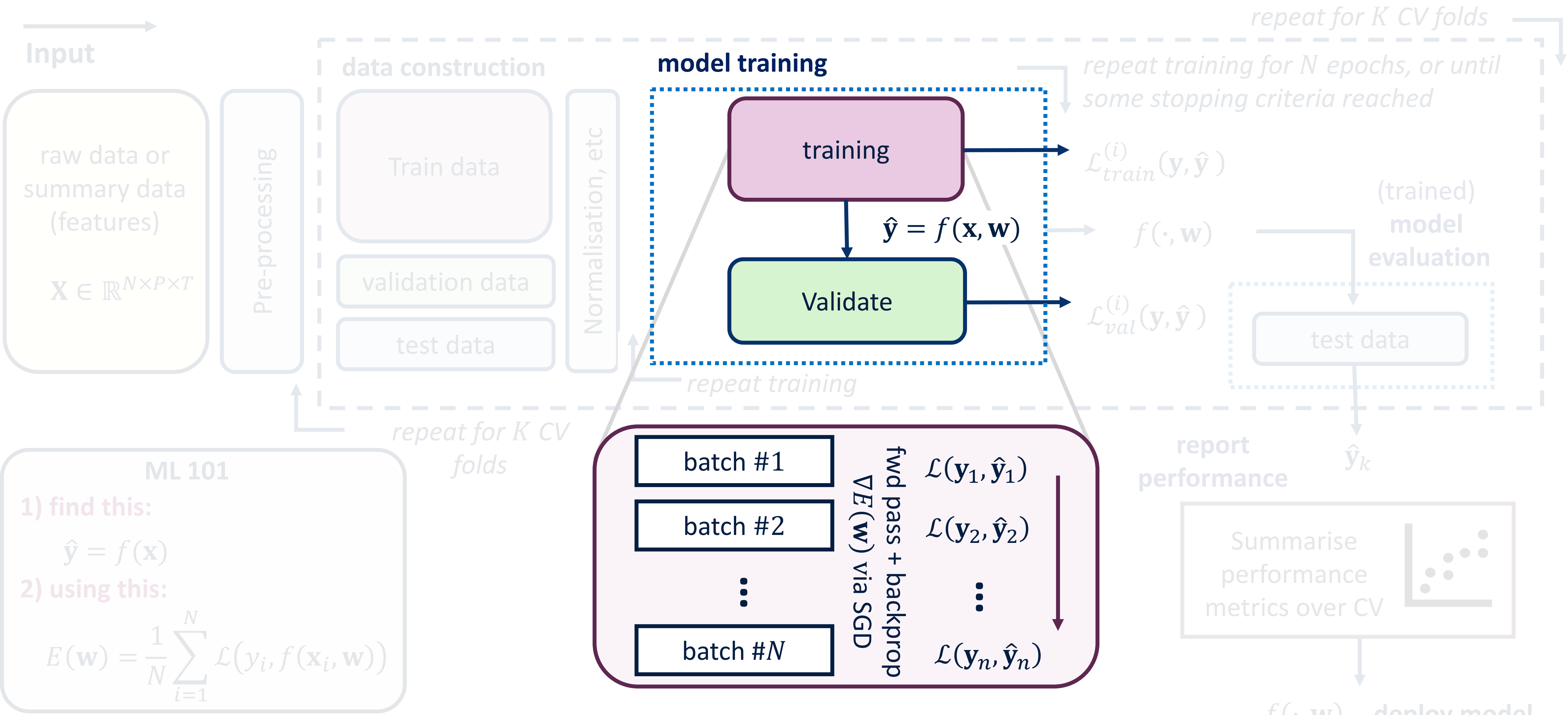
Typical Machine Learning Pipeline



$\hat{\mathbf{y}} \in \mathbb{R}^{N \times 1}$ are the predictions / estimations; $\mathbf{y} \in \mathbb{R}^{N \times 1}$ are the response labels; $\mathbf{w} \in \mathbb{R}^{1 \times N}$ are the weights or parameters of a model

N: number of samples / observations; M: number of timesteps; P: number of features, input channels, etc.

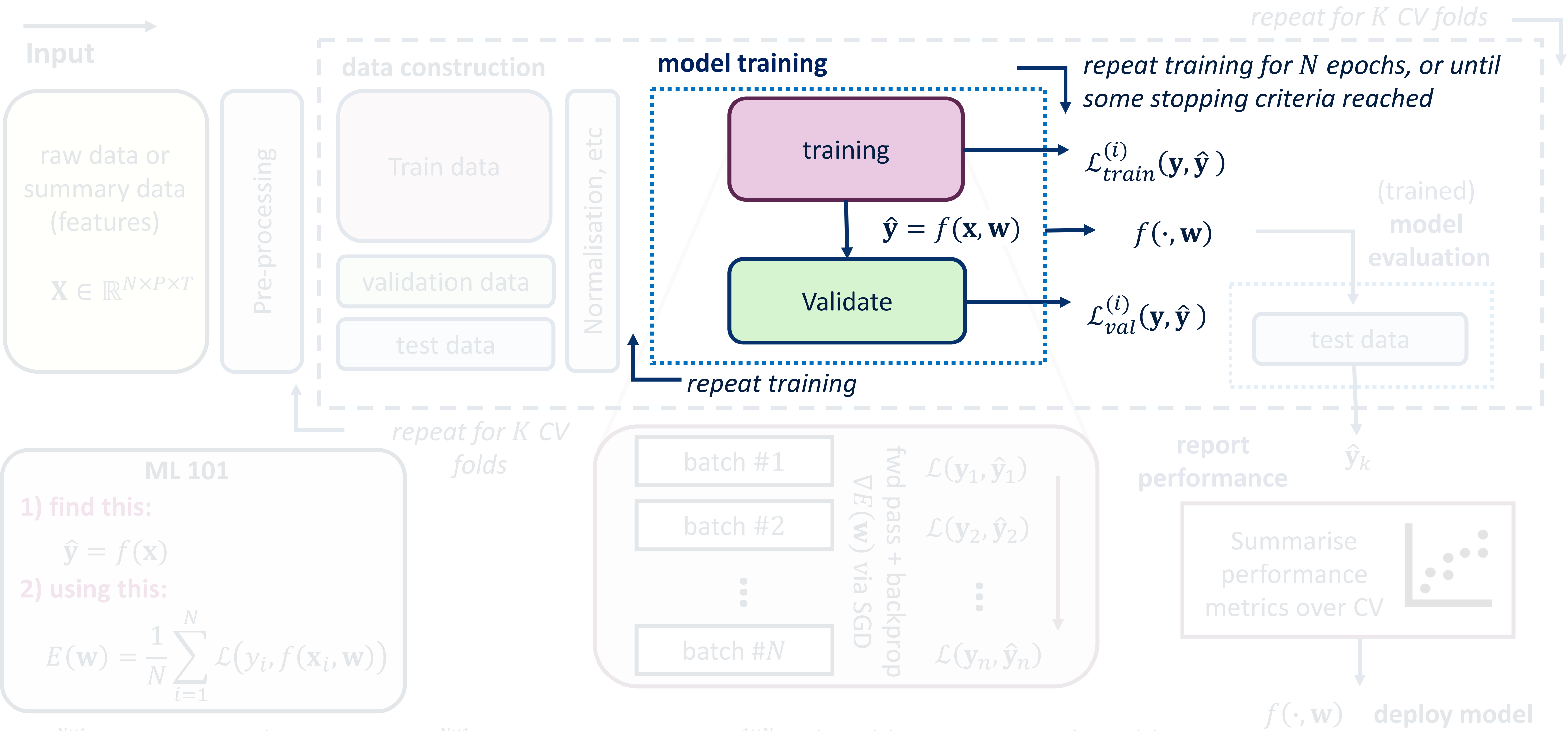
Typical Machine Learning Pipeline



$\hat{\mathbf{y}} \in \mathbb{R}^{N\times1}$ are the predictions / estimations; $\mathbf{y} \in \mathbb{R}^{N\times1}$ are the response labels; $\mathbf{w} \in \mathbb{R}^{1\times N}$ are the weights or parameters of a model

N: number of samples / observations; M: number of timesteps; P: number of features, input channels, etc.

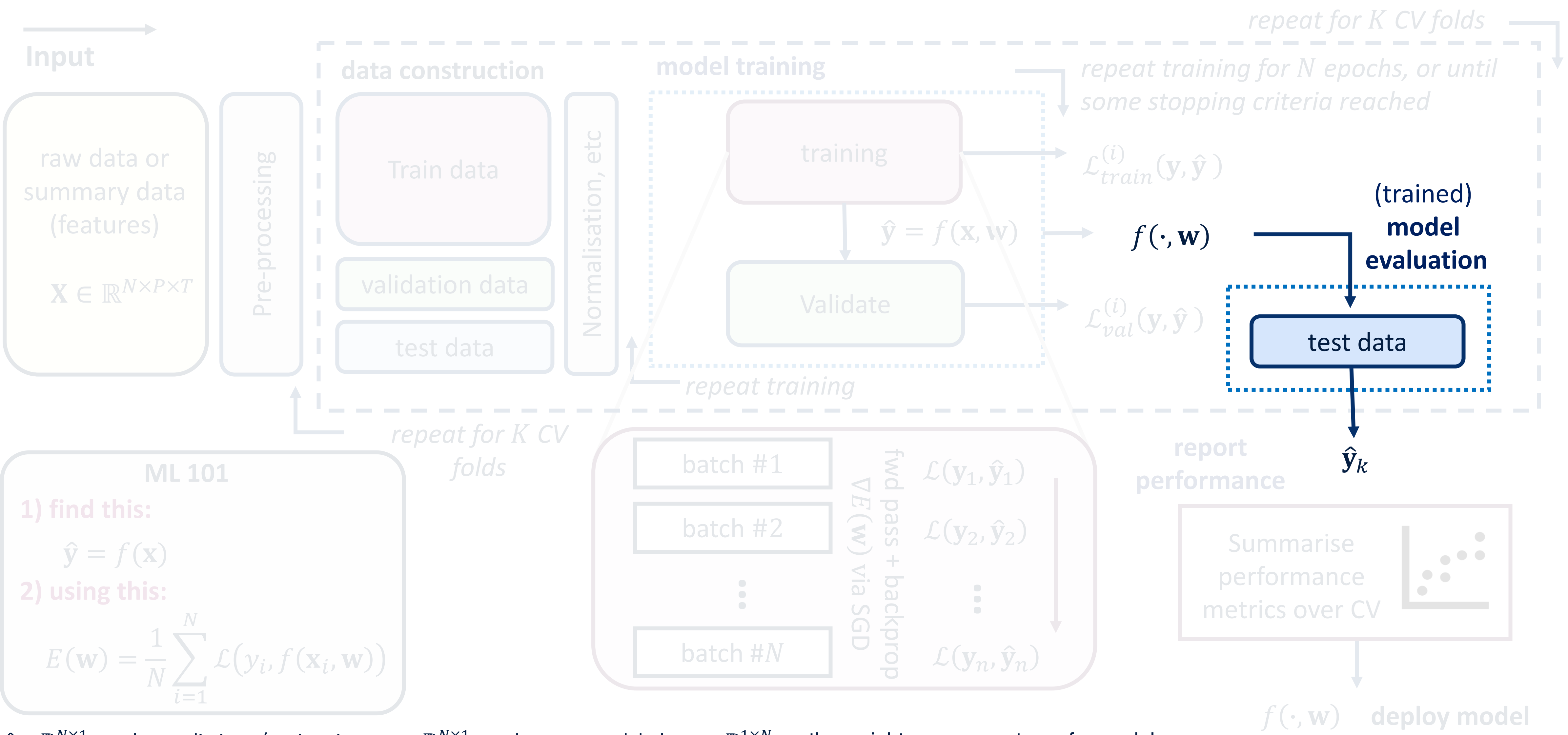
Typical Machine Learning Pipeline



$\hat{\mathbf{y}} \in \mathbb{R}^{N\times 1}$ are the predictions / estimations; $\mathbf{y} \in \mathbb{R}^{N\times 1}$ are the response labels; $\mathbf{w} \in \mathbb{R}^{1\times N}$ are the weights or parameters of a model

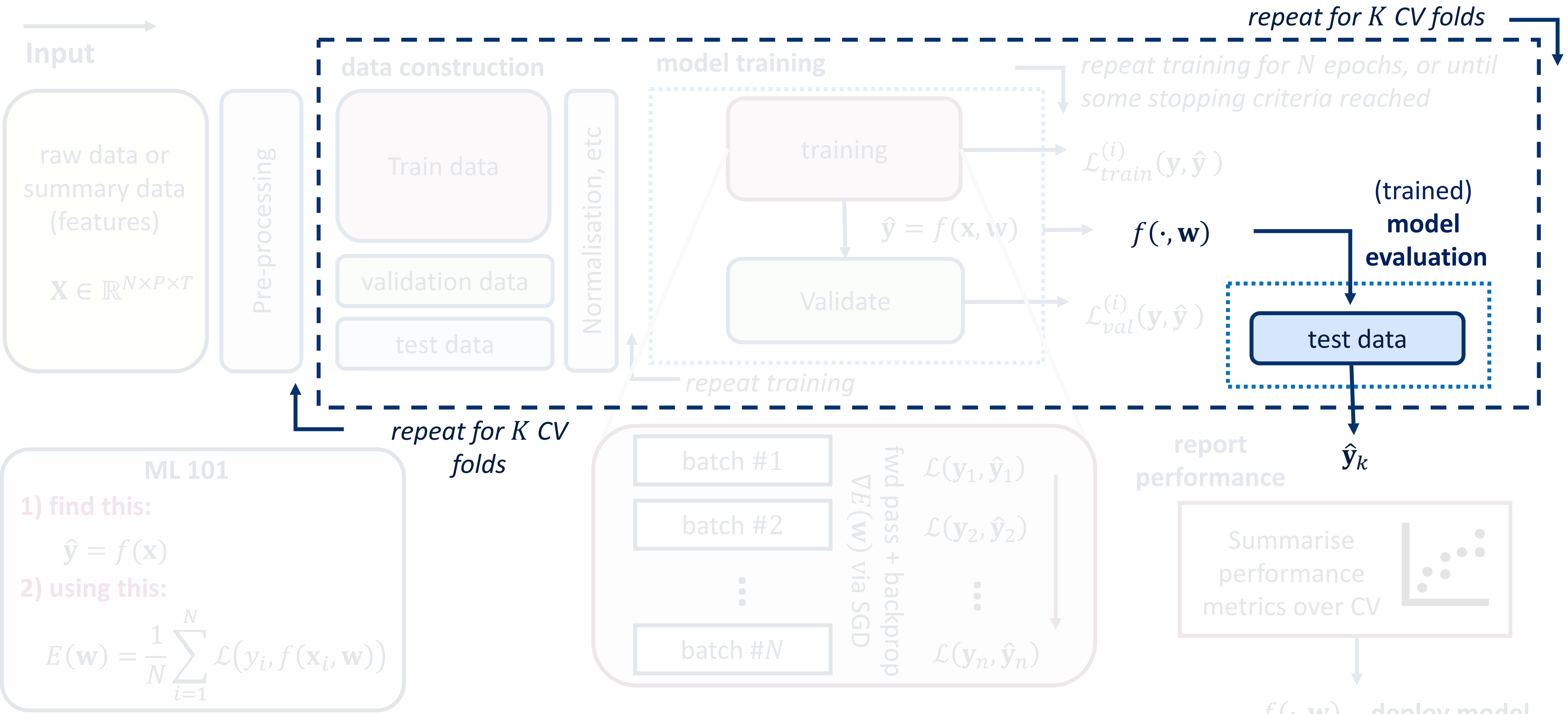
N: number of samples / observations; M: number of timesteps; P: number of features, input channels, etc.

Typical Machine Learning Pipeline



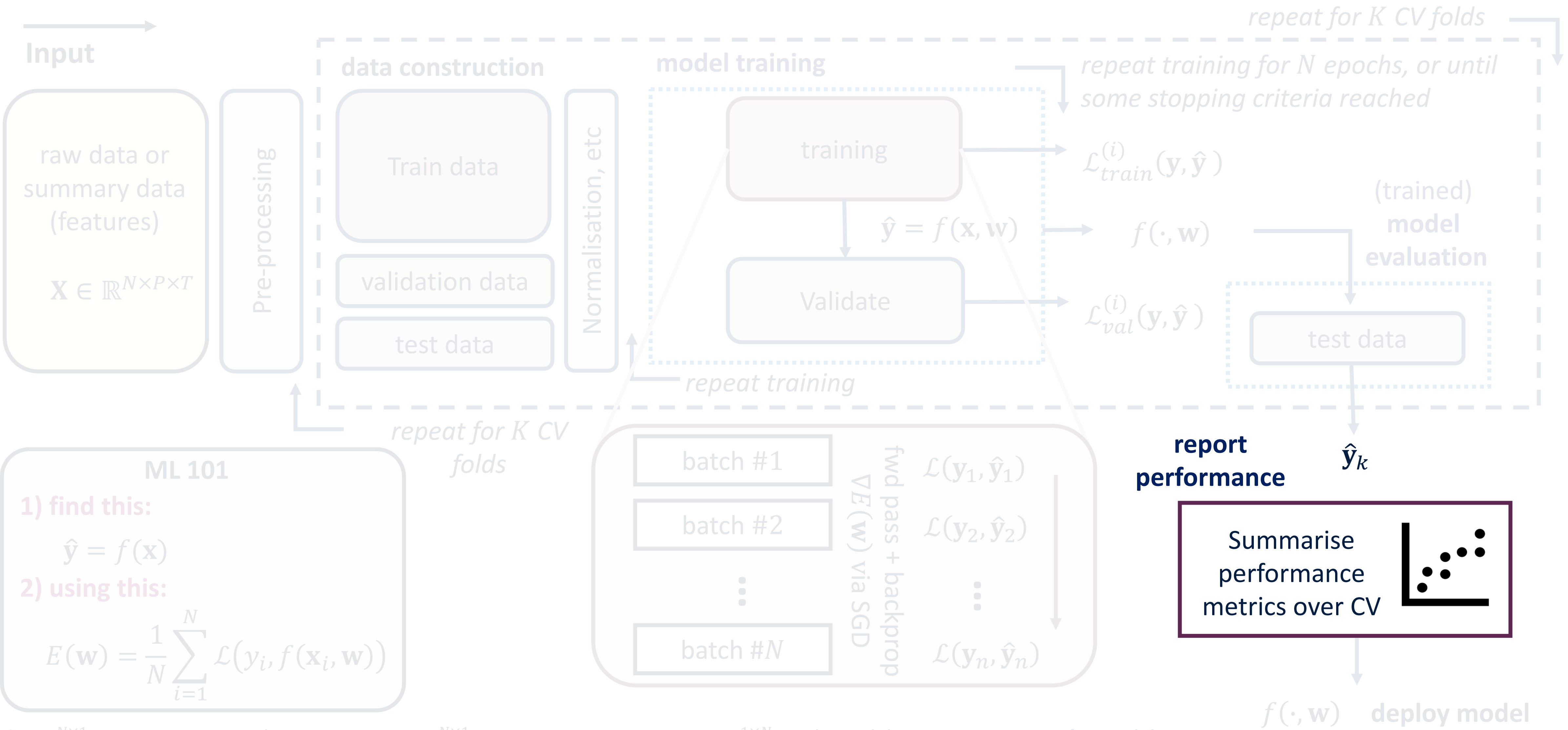
$\hat{\mathbf{y}} \in \mathbb{R}^{N \times 1}$ are the predictions / estimations; $\mathbf{y} \in \mathbb{R}^{N \times 1}$ are the response labels; $\mathbf{w} \in \mathbb{R}^{1 \times N}$ are the weights or parameters of a model
N: number of samples / observations; M: number of timesteps; P: number of features, input channels, etc.

Typical Machine Learning Pipeline



$\hat{\mathbf{y}} \in \mathbb{R}^{N \times 1}$ are the predictions / estimations; $\mathbf{y} \in \mathbb{R}^{N \times 1}$ are the response labels; $\mathbf{w} \in \mathbb{R}^{1 \times N}$ are the weights or parameters of a model
N: number of samples / observations; M: number of timesteps; P: number of features, input channels, etc.

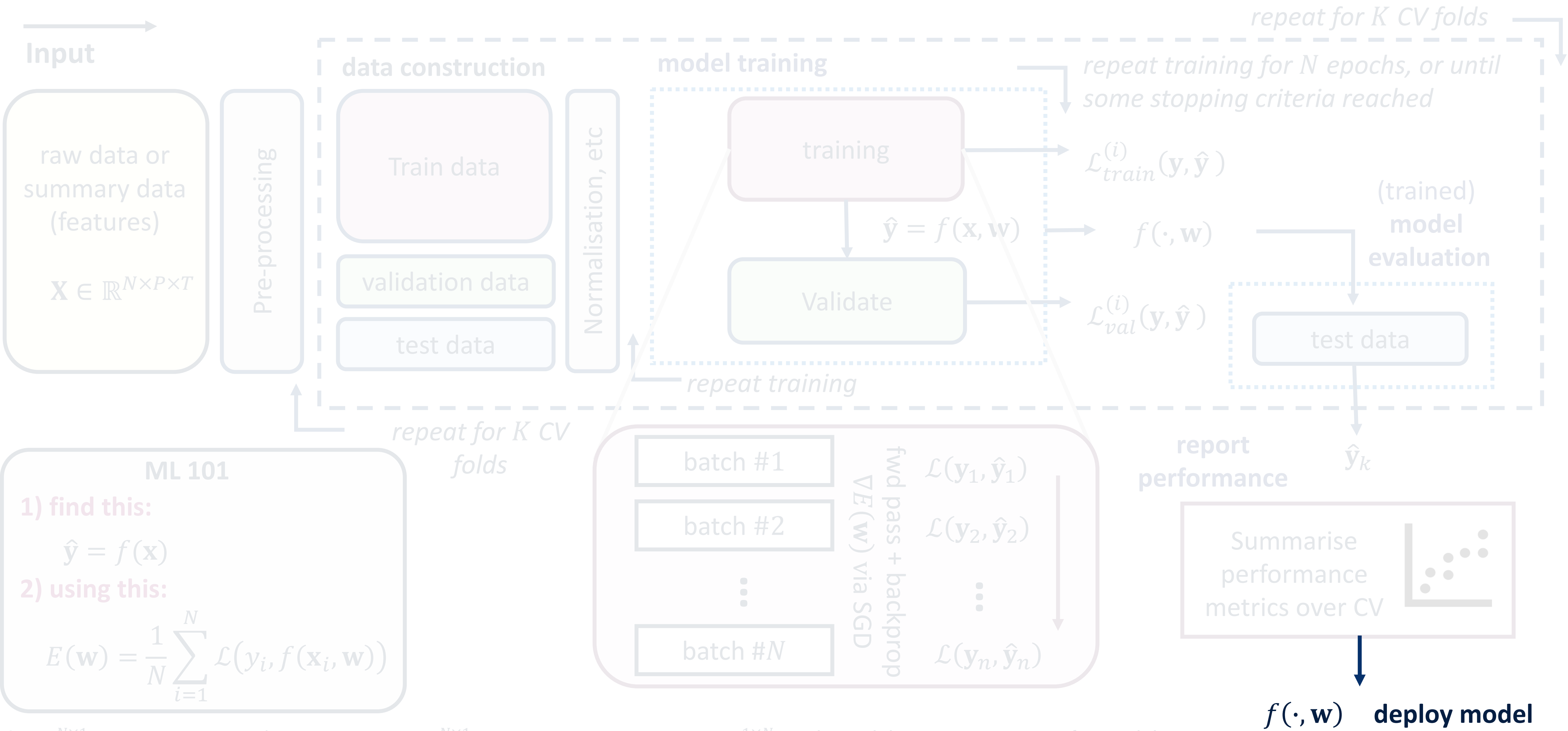
Typical Machine Learning Pipeline



$\hat{\mathbf{y}} \in \mathbb{R}^{N \times 1}$ are the predictions / estimations; $\mathbf{y} \in \mathbb{R}^{N \times 1}$ are the response labels; $\mathbf{w} \in \mathbb{R}^{1 \times N}$ are the weights or parameters of a model

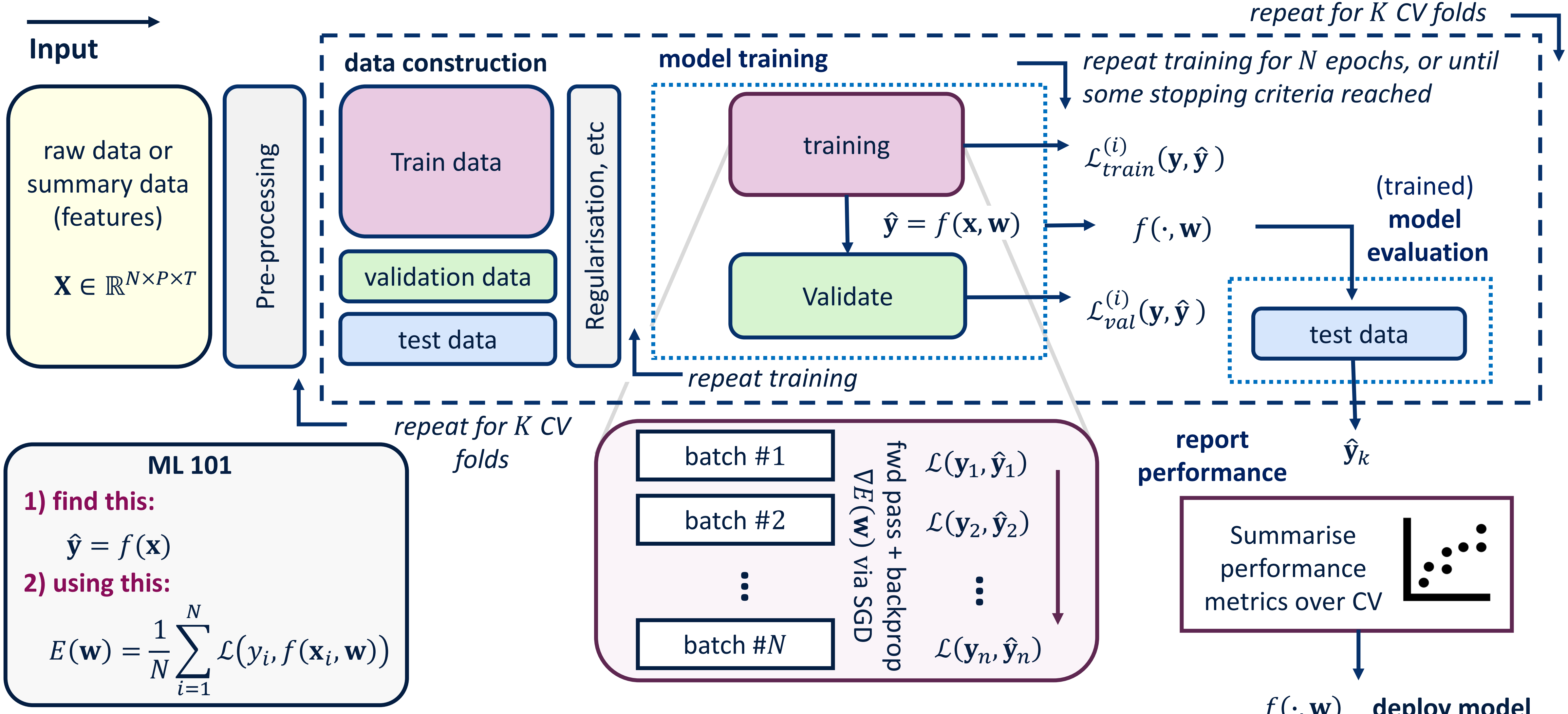
N: number of samples / observations; M: number of timesteps; P: number of features, input channels, etc.

Typical Machine Learning Pipeline



$\hat{\mathbf{y}} \in \mathbb{R}^{N \times 1}$ are the predictions / estimations; $\mathbf{y} \in \mathbb{R}^{N \times 1}$ are the response labels; $\mathbf{w} \in \mathbb{R}^{1 \times N}$ are the weights or parameters of a model
N: number of samples / observations; M: number of timesteps; P: number of features, input channels, etc.

Typical Machine Learning Pipeline



$\hat{\mathbf{y}} \in \mathbb{R}^{N\times 1}$ are the predictions / estimations; $\mathbf{y} \in \mathbb{R}^{N\times 1}$ are the response labels; $\mathbf{w} \in \mathbb{R}^{1\times N}$are the weights or parameters of a model

N: number of samples / observations; M: number of timesteps; P: number of features, input channels, etc.

ML 4 Time-series: Recurrent Neural Networks

What have we learned?

- Assuming individual time-steps as I.I.D. is a naïve approach
- Incorporating recurrence into our networks can model temporality
- RNNs are prone to *vanishing* and *exploding* gradient problems
- Gated cells, such as LSTMS overcome these difficulties, allowing us to model long-term dependencies
- How we can implement and train RNNs/LSTMS
- What a typical machine learning pipeline should look like