

# Handling sequences with PyTorch

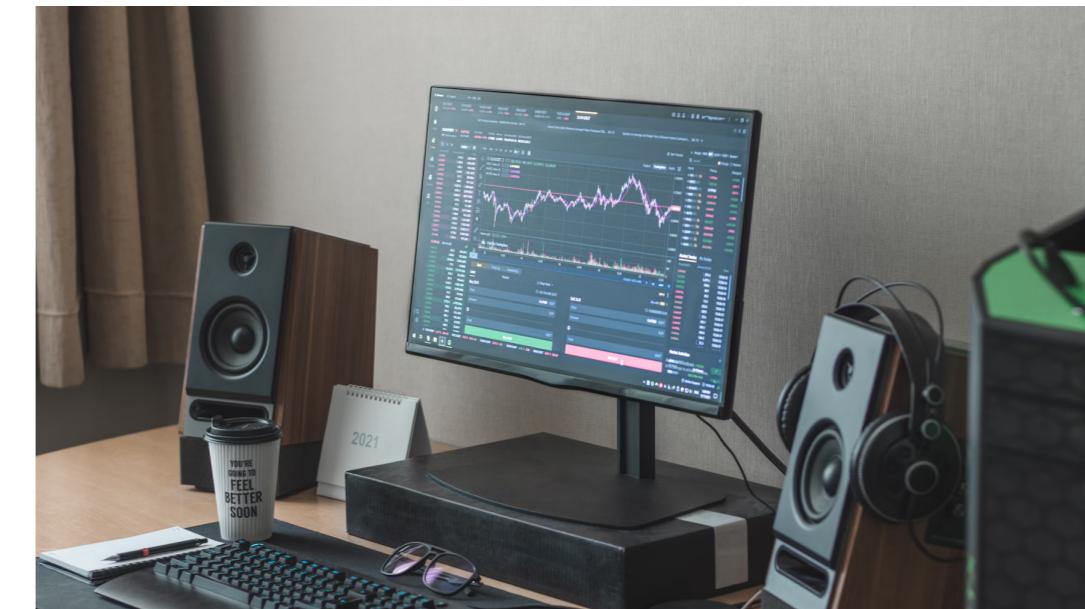
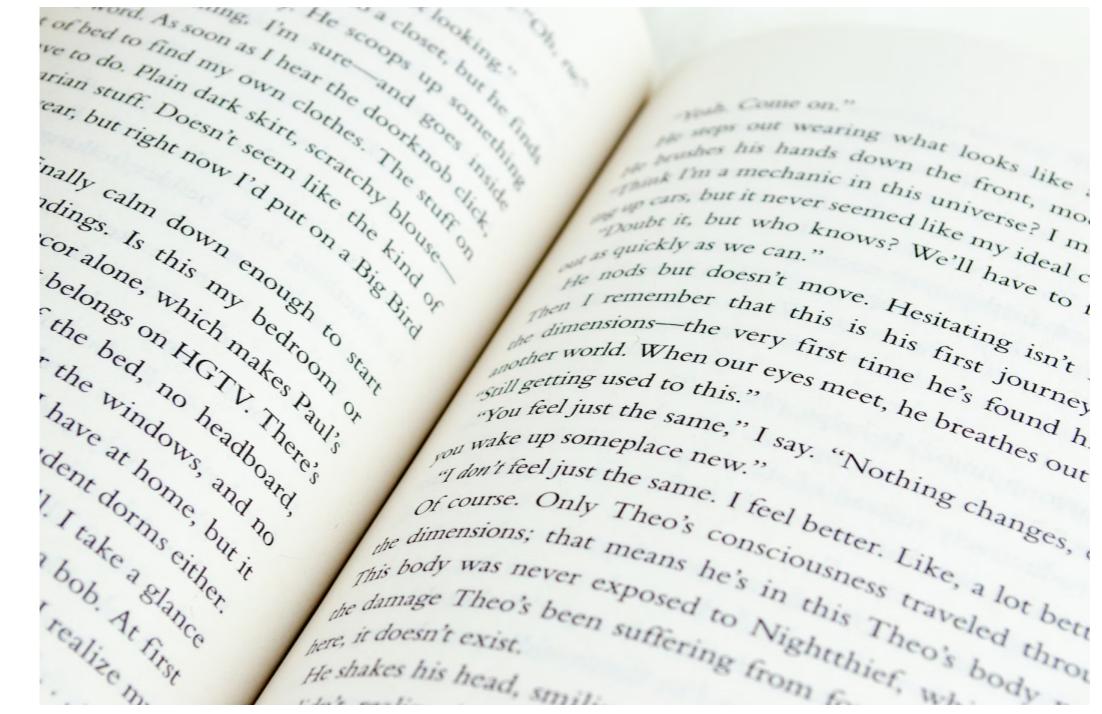
INTERMEDIATE DEEP LEARNING WITH PYTORCH



Michał Oleszak  
Machine Learning Engineer

# Sequential data

- Ordered in time or space
- Order of the data points contains dependencies between them
- Examples of sequential data:
  - Time series
  - Text
  - Audio waves



# Electricity consumption prediction

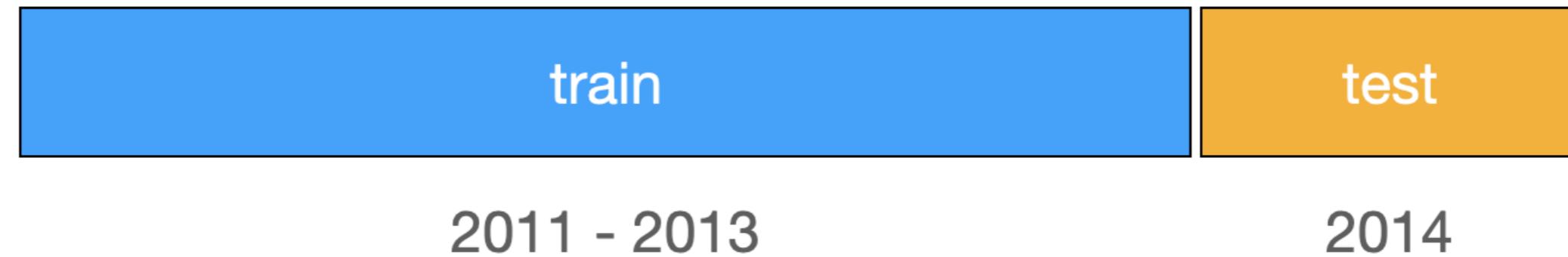
- Task: predict future electricity consumption based on past patterns
- Electricity consumption dataset:

|        | timestamp           | consumption |
|--------|---------------------|-------------|
| 0      | 2011-01-01 00:15:00 | -0.704319   |
| 1      | 2011-01-01 00:30:00 | -0.704319   |
| ...    | ...                 | ...         |
| 140254 | 2014-12-31 23:45:00 | -0.095751   |
| 140255 | 2015-01-01 00:00:00 | -0.095751   |

<sup>1</sup> Trindade, Artur. (2015). ElectricityLoadDiagrams20112014. UCI Machine Learning Repository.  
<https://doi.org/10.24432/C58C86>.

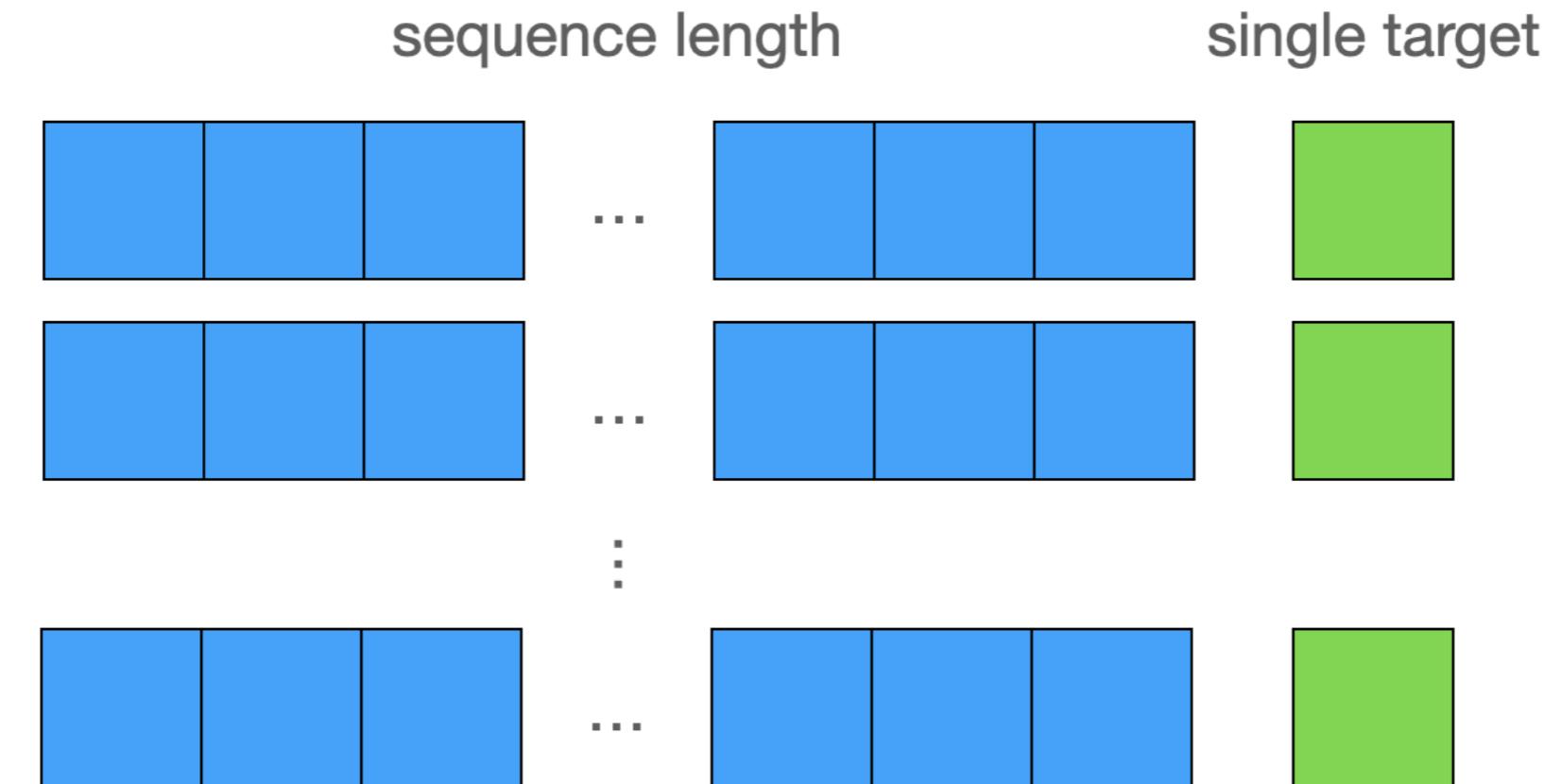
# Train-test split

- No random splitting for time series!
- **Look-ahead bias:** model has info about the future
- Solution: split by time



# Creating sequences

- Sequence length = number of data points in one training example
  - $24 \times 4 = 96$  -> consider last 24 hours
- Predict single next data point



# Creating sequences in Python

```
import numpy as np

def create_sequences(df, seq_length):
    xs, ys = [], []
    for i in range(len(df) - seq_length):
        x = df.iloc[i:(i+seq_length), 1]
        y = df.iloc[i+seq_length, 1]
        xs.append(x)
        ys.append(y)
    return np.array(xs), np.array(ys)
```

- Take data and sequence length as inputs
- Initialize inputs and targets lists
- Iterate over data points
- Define inputs and target
- Append to pre-initialized lists
- Return inputs and targets as NumPy arrays

# TensorDataset

Create training examples

```
X_train, y_train = create_sequences(train_data, seq_length)  
print(X_train.shape, y_train.shape)
```

```
(34944, 96) (34944,)
```

Convert them to a Torch Dataset

```
from torch.utils.data import TensorDataset  
  
dataset_train = TensorDataset(  
    torch.from_numpy(X_train).float(),  
    torch.from_numpy(y_train).float(),  
)
```

# Applicability to other sequential data

Same techniques are applicable to other sequences:

- Large Language Models
- Speech recognition

# **Let's practice!**

**INTERMEDIATE DEEP LEARNING WITH PYTORCH**

# Recurrent Neural Networks

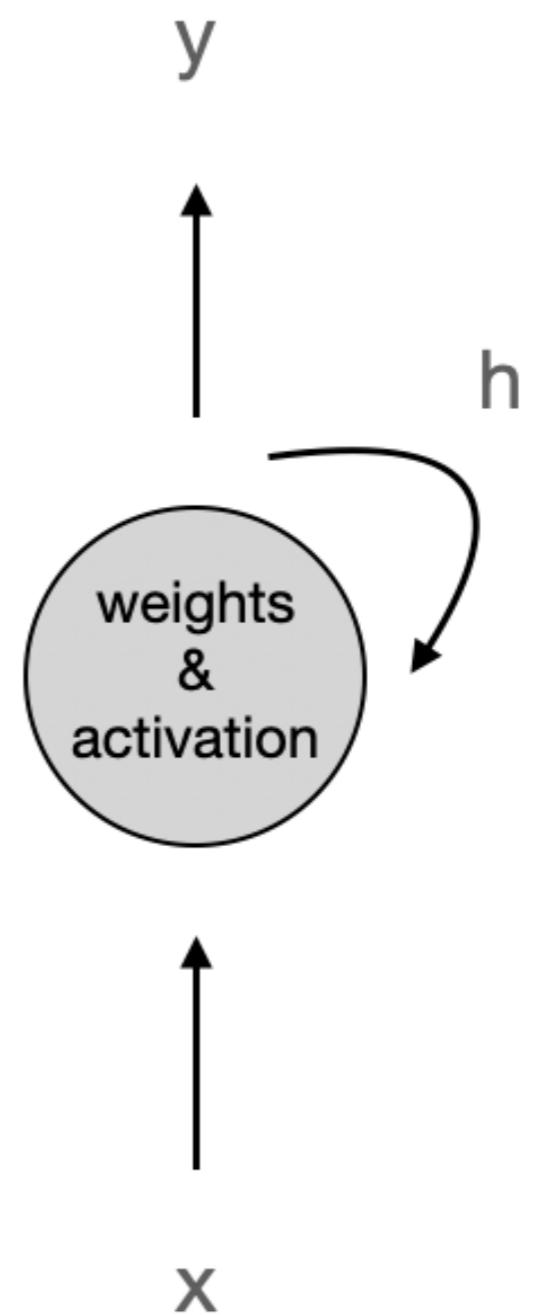
INTERMEDIATE DEEP LEARNING WITH PYTORCH



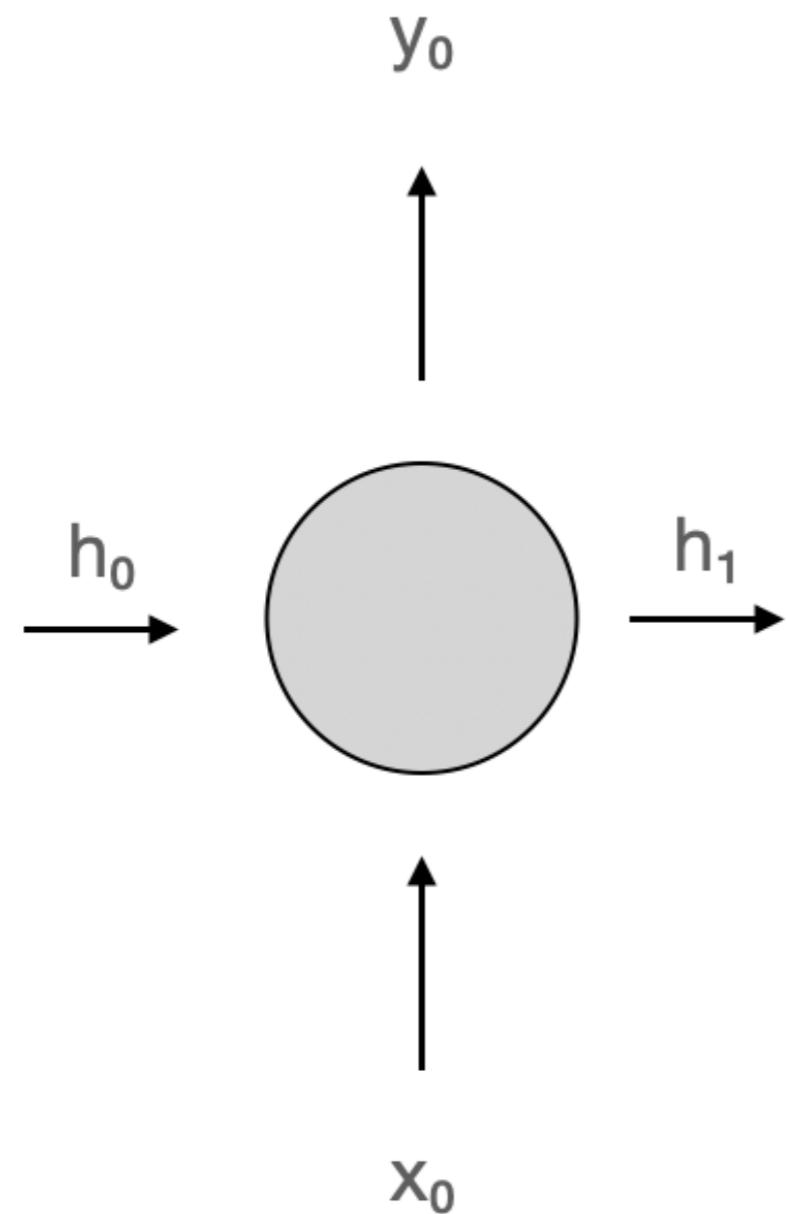
Michał Oleszak  
Machine Learning Engineer

# Recurrent neuron

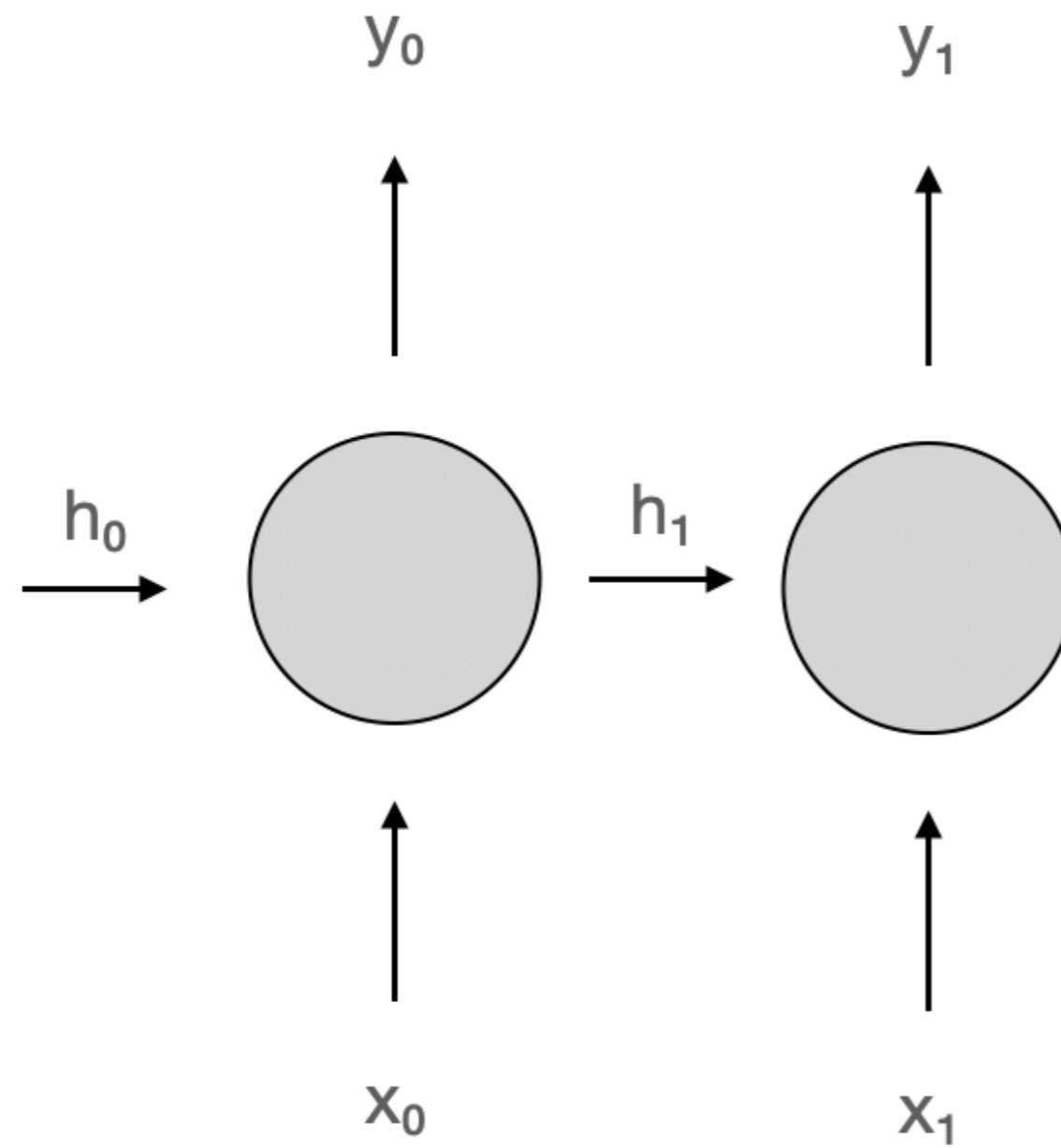
- Feed-forward networks
- RNNs: have connections pointing back
- Recurrent neuron:
  - Input  $x$
  - Output  $y$
  - Hidden state  $h$
- In PyTorch: `nn.RNN()`



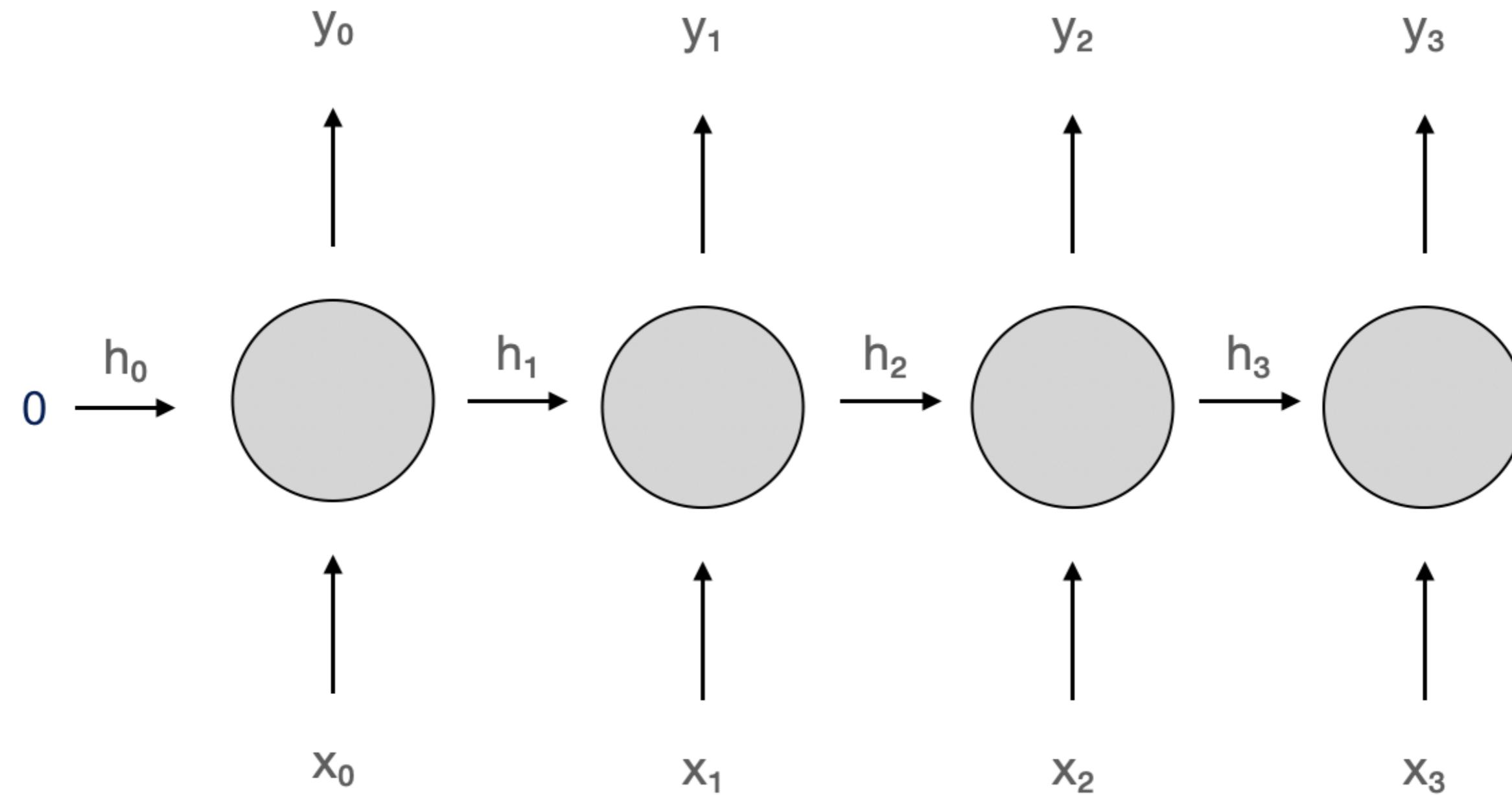
# Unrolling recurrent neuron through time



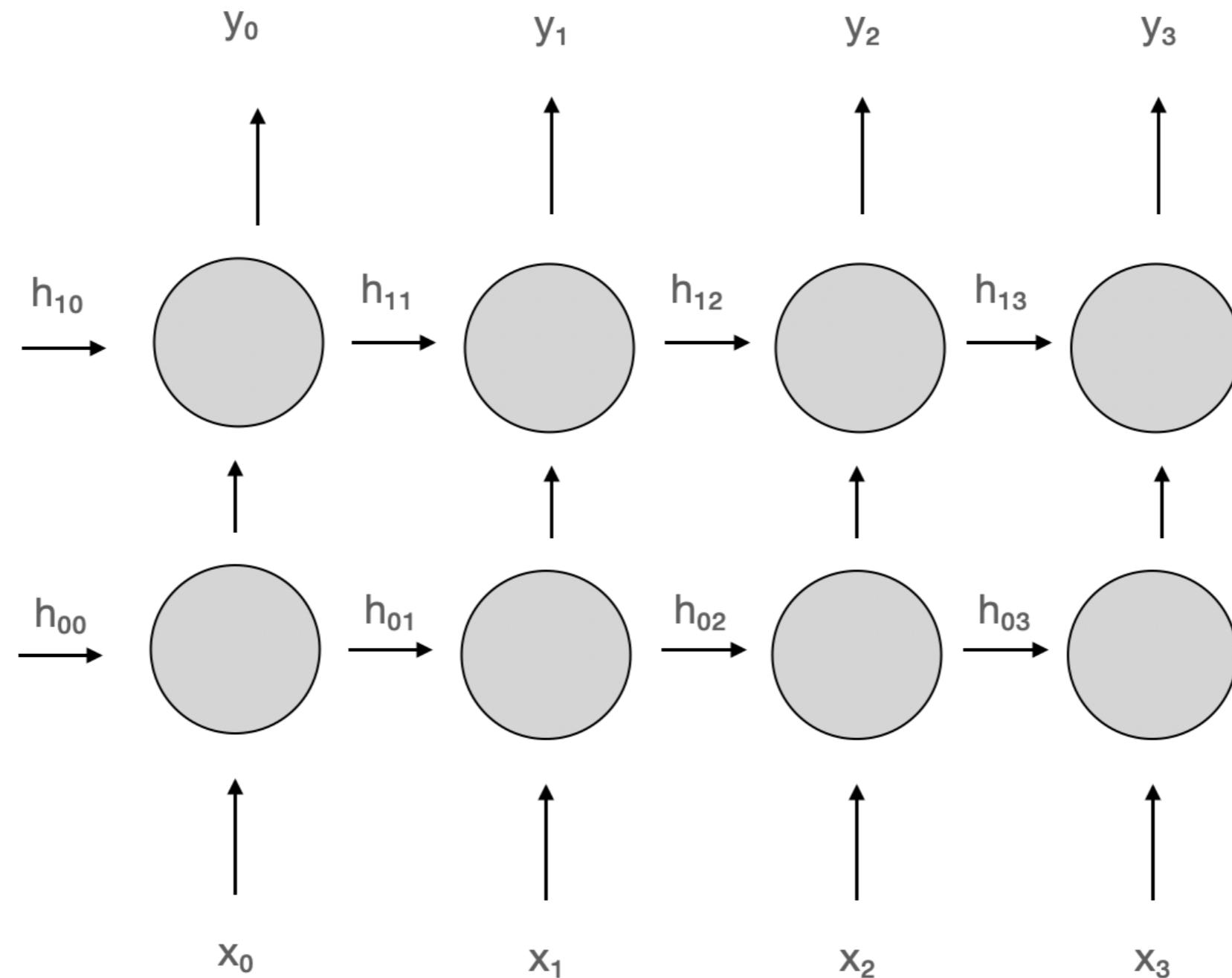
# Unrolling recurrent neuron through time



# Unrolling recurrent neuron through time

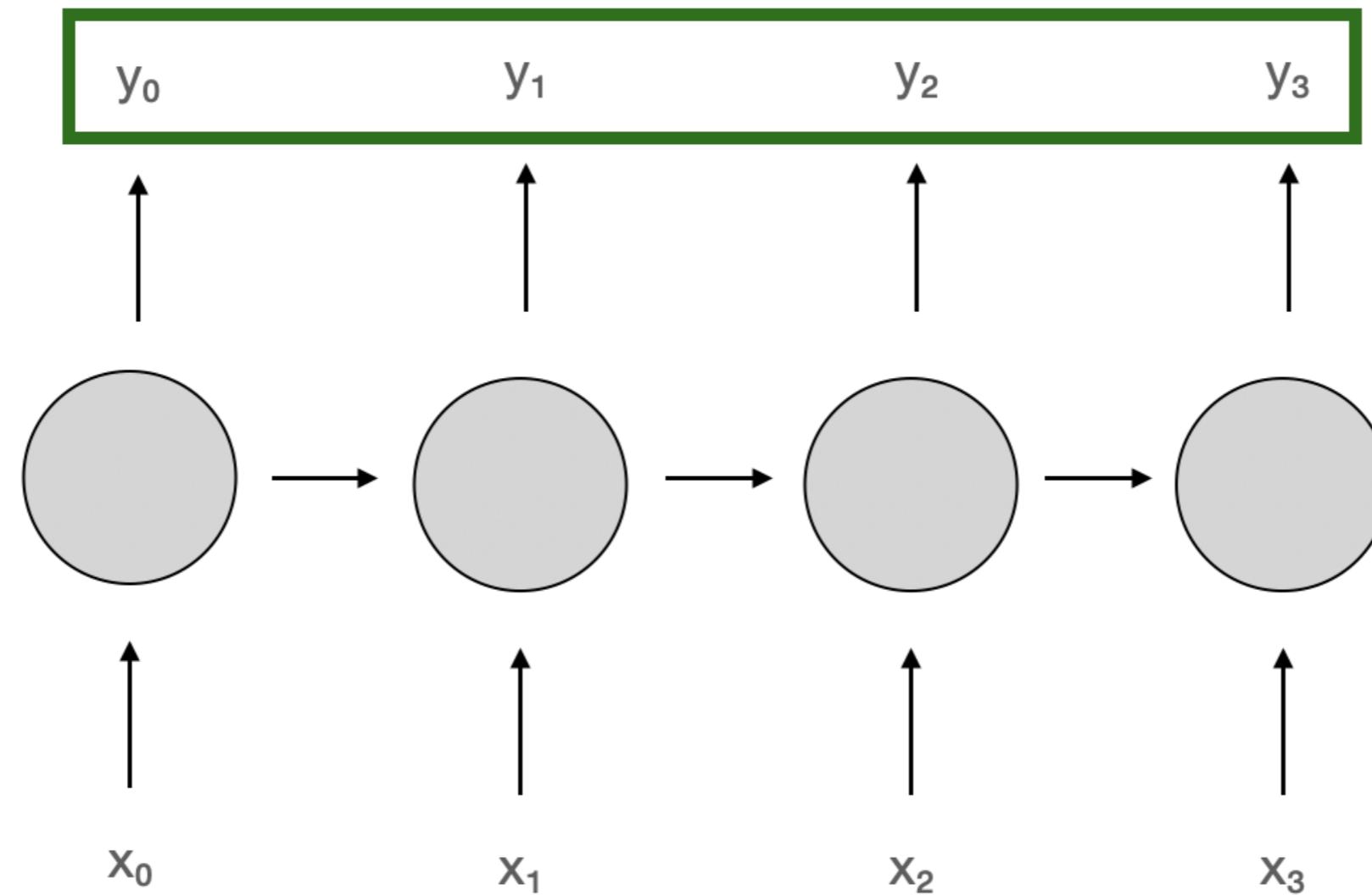


# Deep RNNs



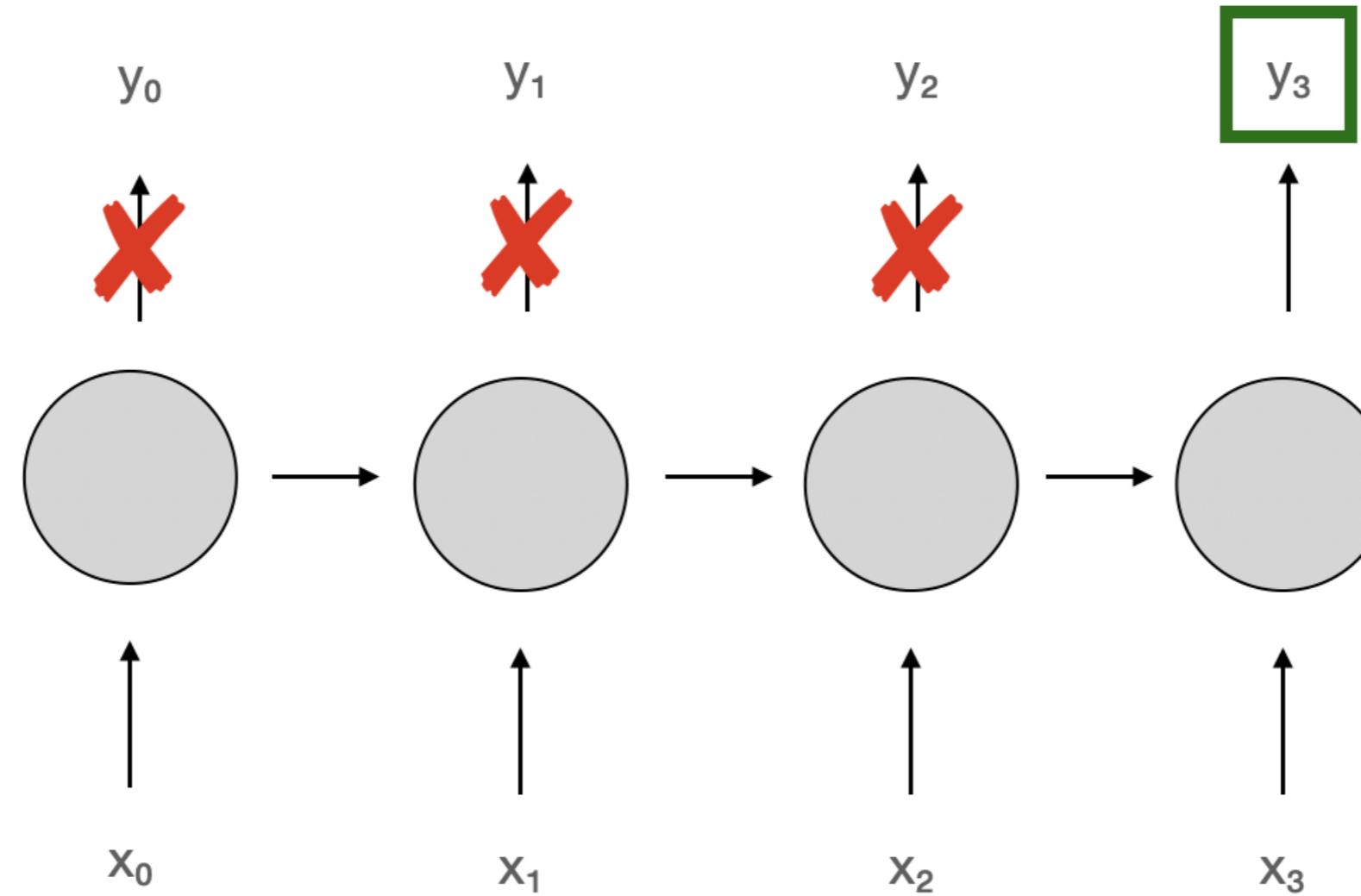
# Sequence-to-sequence architecture

- Pass sequence as input, use the entire output sequence
- **Example:** Real-time speech recognition



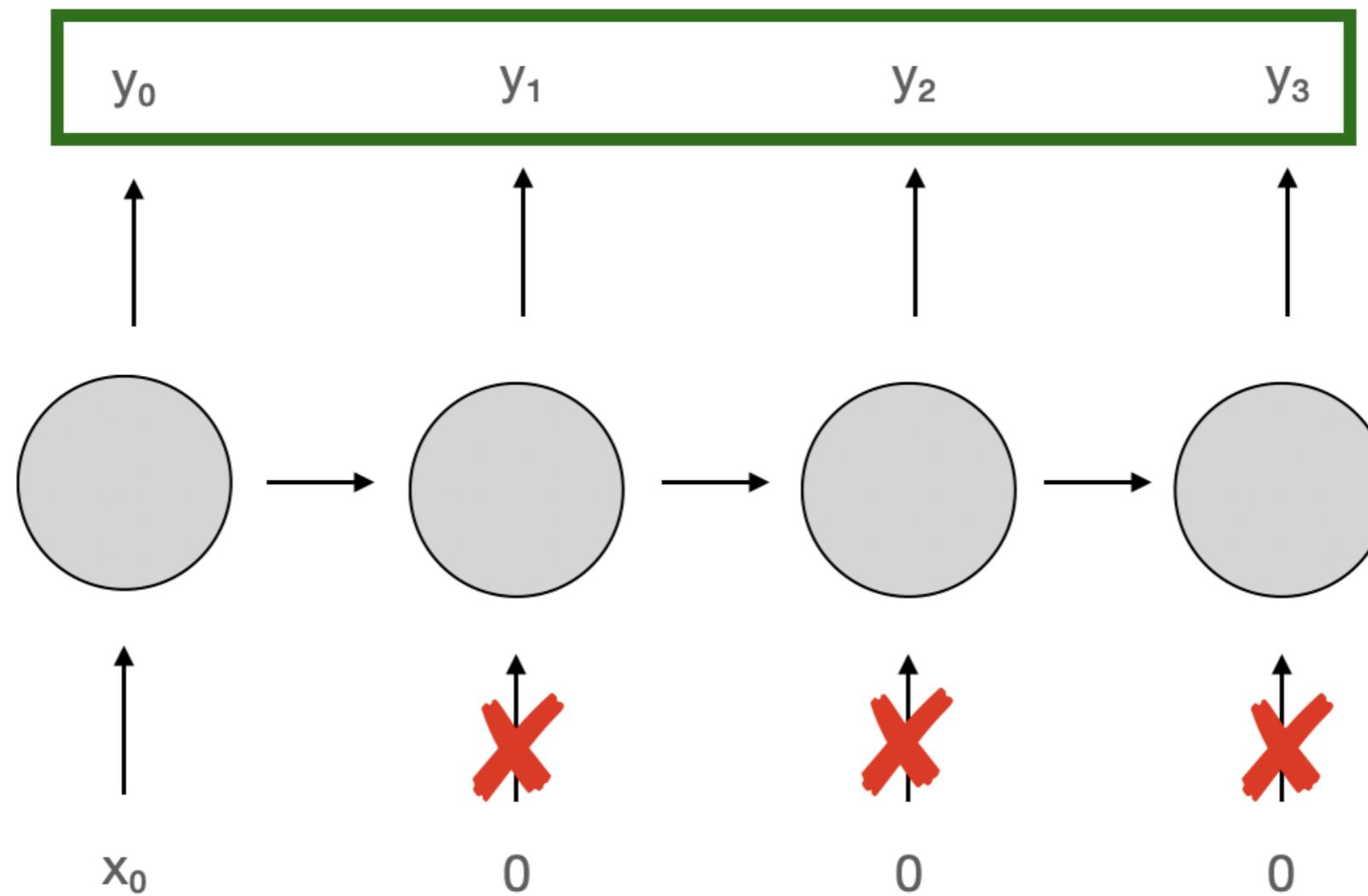
# Sequence-to-vector architecture

- Pass sequence as input, use only the last output
- **Example:** Text topic classification



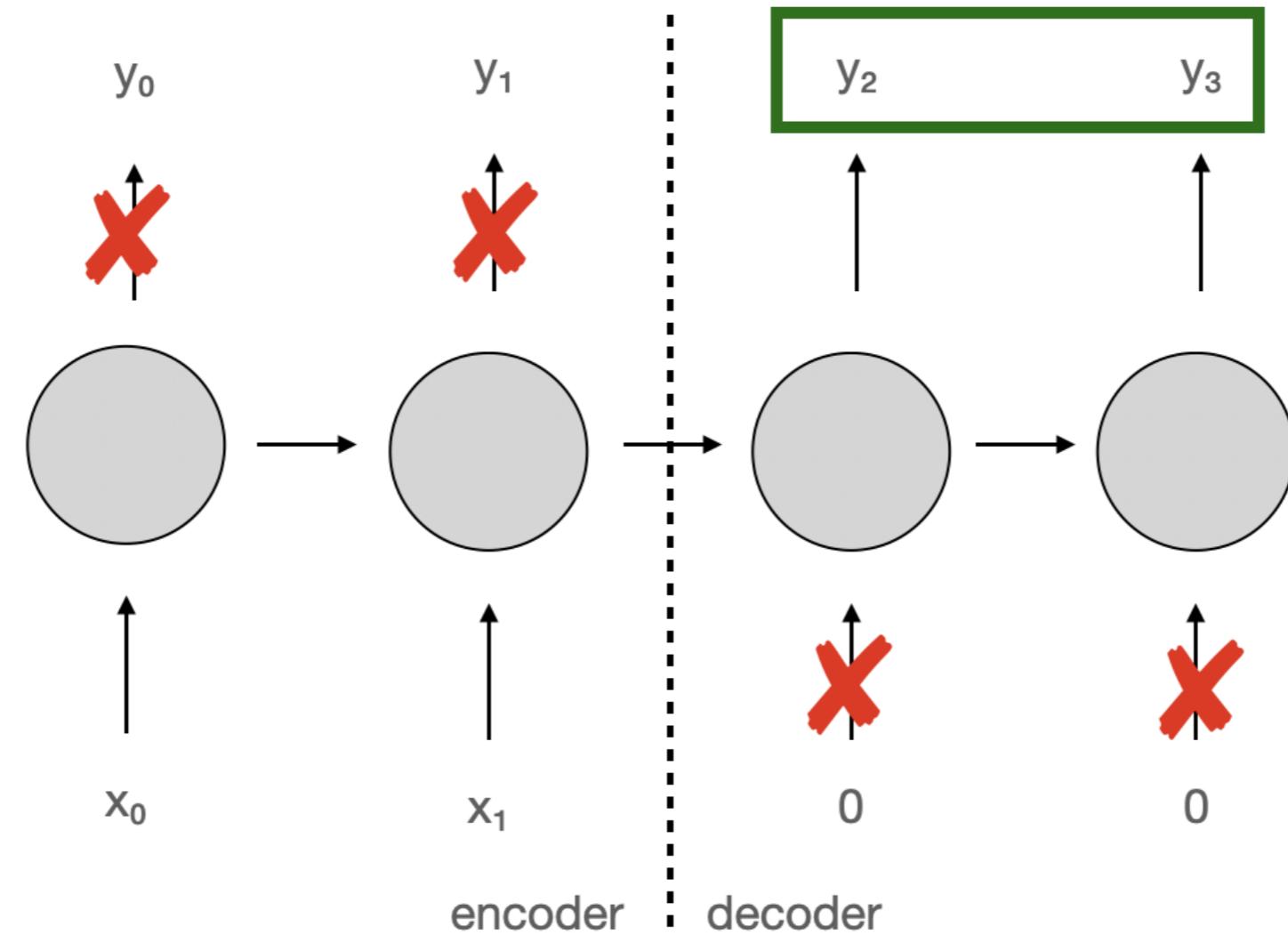
# Vector-to-sequence architecture

- Pass single input, use the entire output sequence
- **Example:** Text generation



# Encoder-decoder architecture

- Pass entire input sequence, only then start using output sequence
- **Example:** Machine translation



# RNN in PyTorch

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.rnn = nn.RNN(
            input_size=1,
            hidden_size=32,
            num_layers=2,
            batch_first=True,
        )
        self.fc = nn.Linear(32, 1)

    def forward(self, x):
        h0 = torch.zeros(2, x.size(0), 32)
        out, _ = self.rnn(x, h0)
        out = self.fc(out[:, -1, :])
        return out
```

- Define model class with `__init__` method
- Define recurrent layer, `self.rnn`
- Define linear layer, `fc`
- In `forward()`, initialize first hidden state to zeros
- Pass input and first hidden state through RNN layer
- Select last RNN's output and pass it through linear layer

# **Let's practice!**

**INTERMEDIATE DEEP LEARNING WITH PYTORCH**

# LSTM and GRU cells

INTERMEDIATE DEEP LEARNING WITH PYTORCH

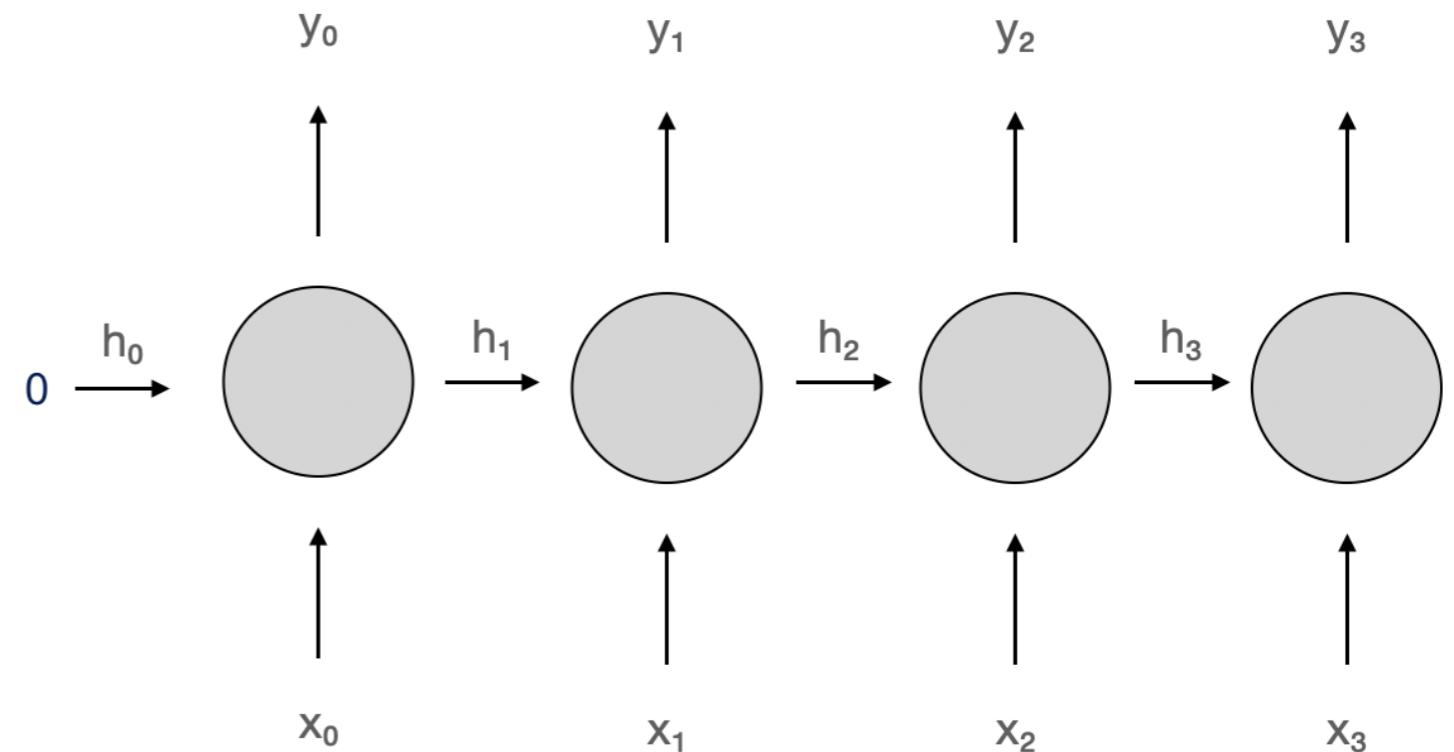


Michał Oleszak

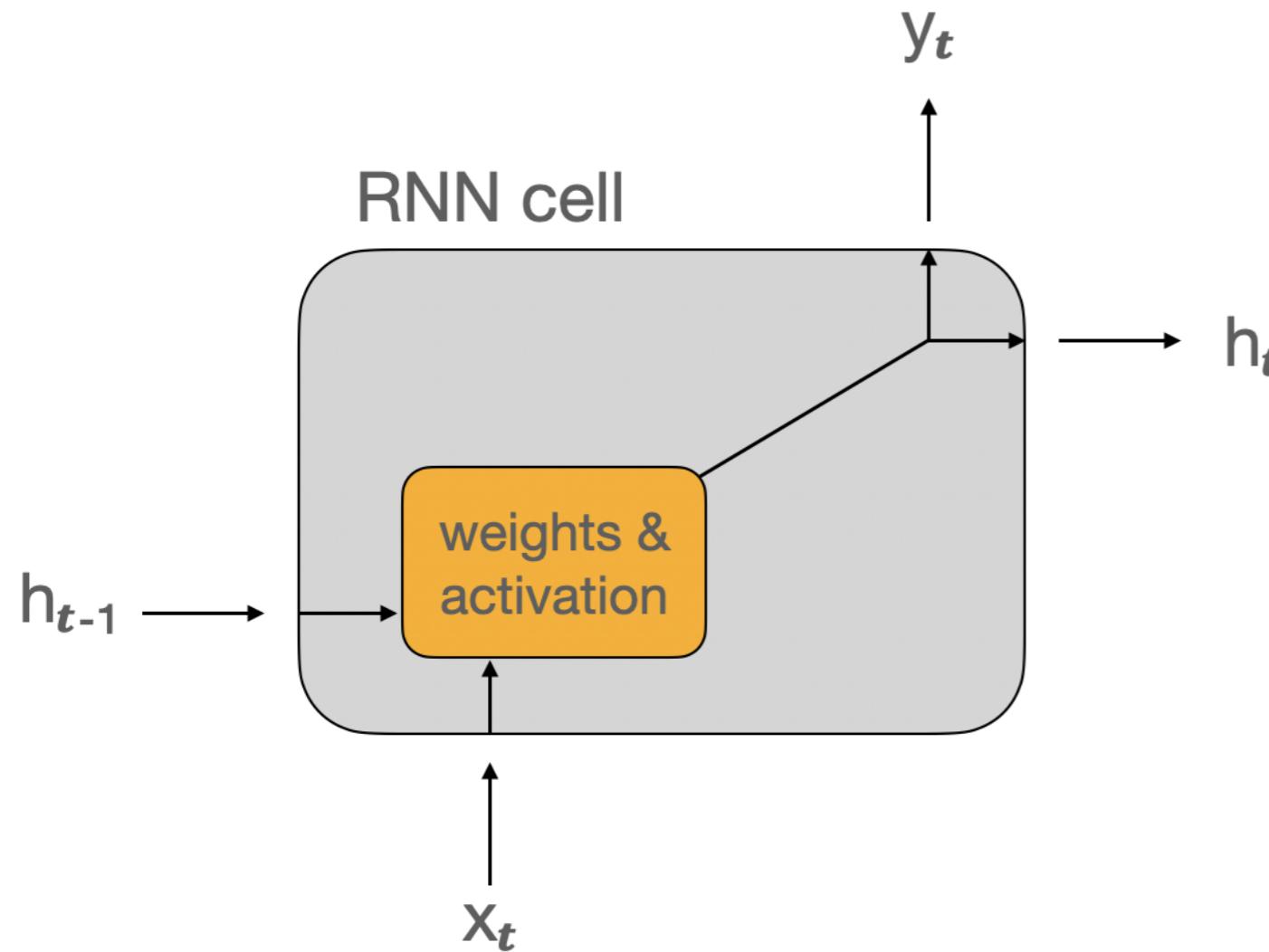
Machine Learning Engineer

# Short-term memory problem

- RNN cells maintain memory via hidden state
- This memory is very short-term
- Two more powerful cells solve the problem:
  - LSTM (Long Short-Term Memory) cell
  - GRU (Gated Recurrent Unit) cell

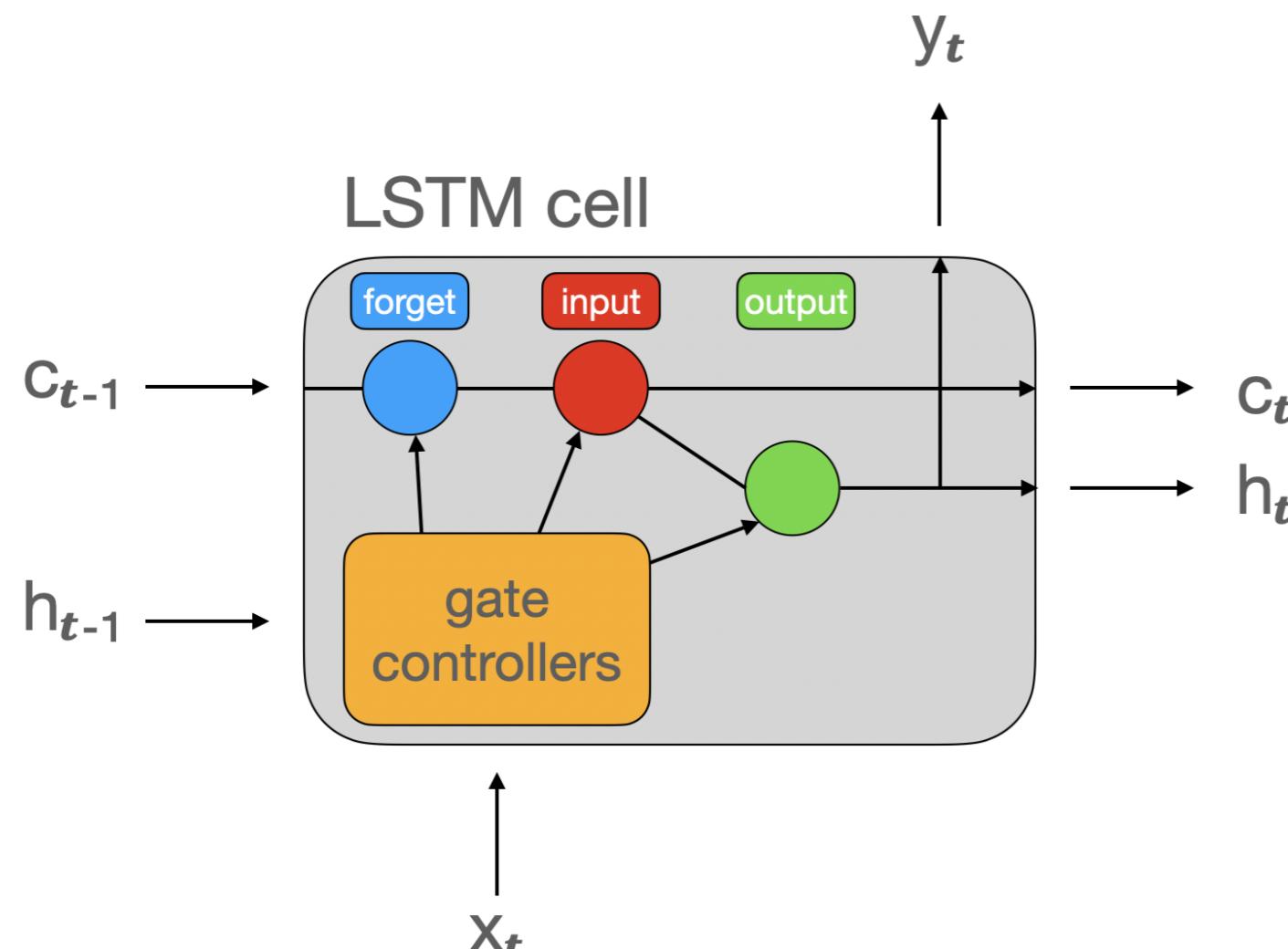


# RNN cell



- Two inputs:
  - current input data  $x$
  - previous hidden state  $h$
- Two outputs:
  - current output  $y$
  - next hidden state  $h$

# LSTM cell



- Outputs  $h$  and  $y$  are the same

- Three inputs and outputs (two hidden states):
  - $h$  : short-term state
  - $c$  : long-term state
- Three "gates":
  - **Forget gate**: what to remove from long-term memory
  - **Input gate**: what to save to long-term memory
  - **Output gate**: what to return at the current time step

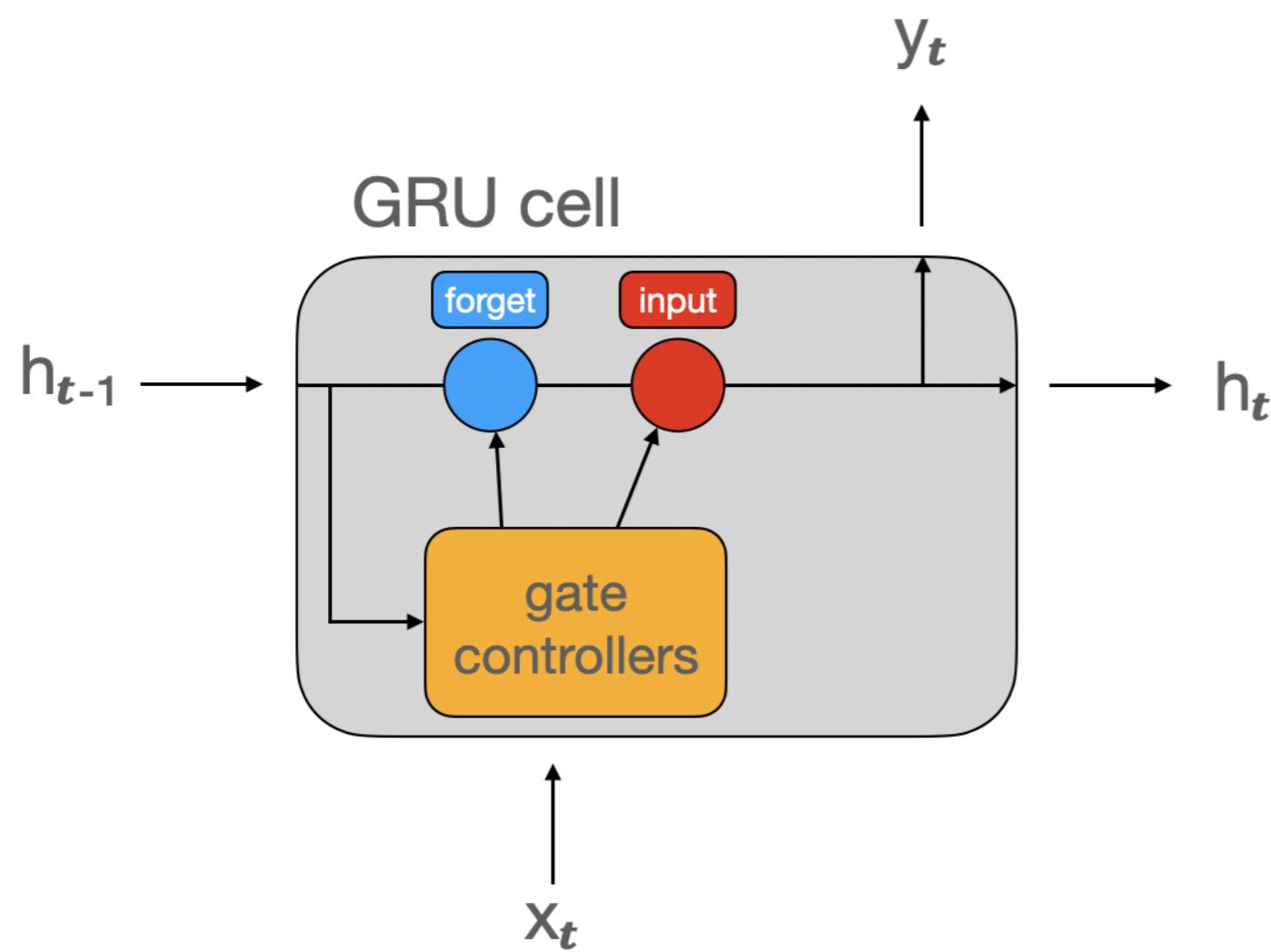
# LSTM in PyTorch

```
class Net(nn.Module):
    def __init__(self, input_size):
        super().__init__()
        self.lstm = nn.LSTM(
            input_size=1,
            hidden_size=32,
            num_layers=2,
            batch_first=True,
        )
        self.fc = nn.Linear(32, 1)

    def forward(self, x):
        h0 = torch.zeros(2, x.size(0), 32)
        c0 = torch.zeros(2, x.size(0), 32)
        out, _ = self.lstm(x, (h0, c0))
        out = self.fc(out[:, -1, :])
        return out
```

- `__init__()` :
  - Replace `nn.RNN` with `nn.LSTM`
- `forward()` :
  - Add another hidden state `c`
  - Initialize `c` and `h` with zeros
  - Pass both hidden states to `lstm` layer

# GRU cell



- Simplified version of LSTM cell
- Just one hidden state
- No output gate

# GRU in PyTorch

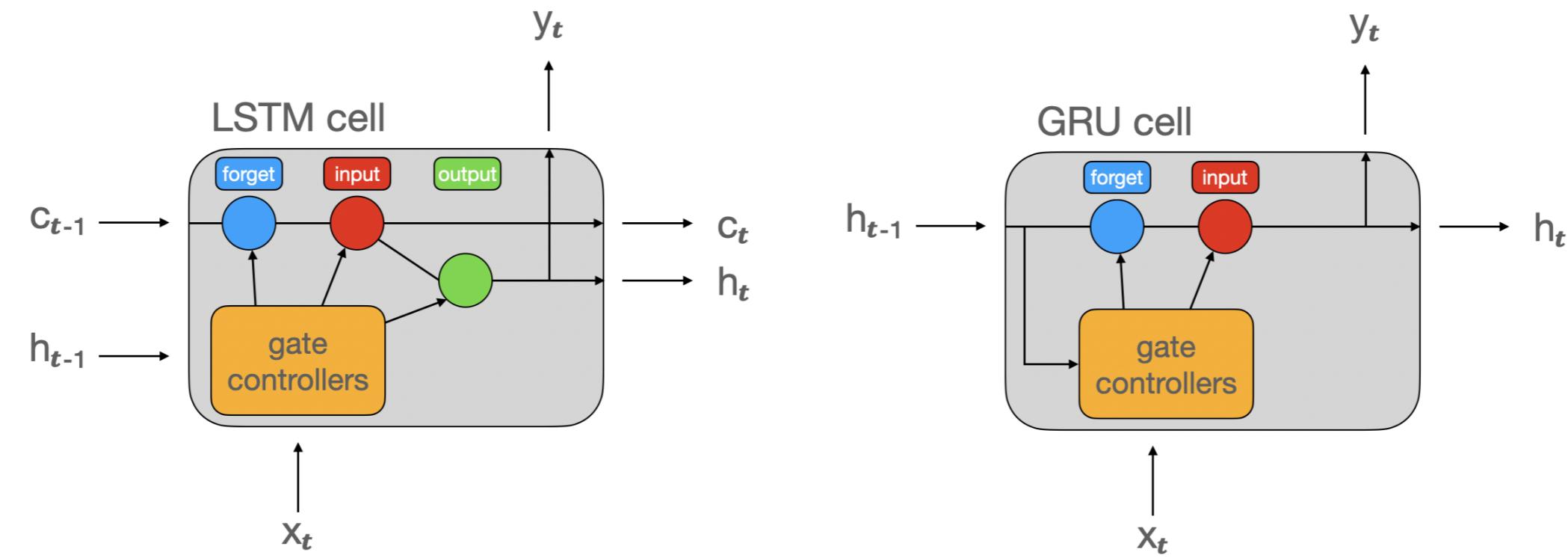
```
class Net(nn.Module):
    def __init__(self, input_size):
        super().__init__()
        self.gru = nn.GRU(
            input_size=1,
            hidden_size=32,
            num_layers=2,
            batch_first=True,
        )
        self.fc = nn.Linear(32, 1)

    def forward(self, x):
        h0 = torch.zeros(2, x.size(0), 32)
        out, _ = self.gru(x, h0)
        out = self.fc(out[:, -1, :])
        return out
```

- `__init__()` :
  - Replace `nn.RNN` with `nn.GRU`
- `forward()` :
  - Use the `gru` layer

# Should I use RNN, LSTM, or GRU?

- RNN is not used much anymore
- GRU is simpler than LSTM = less computation
- Relative performance varies per use-case
- Try both and compare



# **Let's practice!**

**INTERMEDIATE DEEP LEARNING WITH PYTORCH**

# Training and evaluating RNNs

INTERMEDIATE DEEP LEARNING WITH PYTORCH



Michał Oleszak  
Machine Learning Engineer

# Mean Squared Error Loss

- Error:

$$prediction - target$$

- Squared Error:

$$(prediction - target)^2$$

- Mean Squared Error:

$$\text{avg}[(prediction - target)^2]$$

Squaring the error:

- Ensures positive and negative errors don't cancel out
- Penalizes large errors more
- In PyTorch:

```
criterion = nn.MSELoss()
```

# Expanding tensors

- Recurrent layers expect input shape  
(batch\_size, seq\_length, num\_features)
- We got (batch\_size, seq\_length)
- We must add one dimension at the end

```
for seqs, labels in dataloader_train:  
    print(seqs.shape)
```

```
torch.Size([32, 96])
```

```
seqs = seqs.view(32, 96, 1)  
print(seqs.shape)
```

```
torch.Size([32, 96, 1])
```

# Squeezing tensors

- In evaluation loop, we need to revert the reshaping done in the training loop
- Labels are of shape (batch\_size)

```
for seqs, labels in test_loader:  
    print(labels.shape)
```

torch.Size([32])
- Model outputs are (batch\_size, 1)

```
out = net(seqs)
```

torch.Size([32, 1])
- Shapes of model outputs and labels must match for the loss function
- We can drop the last dimension from model outputs

```
out = net(seqs).squeeze()
```

torch.Size([32])

# Training loop

```
net = Net()  
criterion = nn.MSELoss()  
optimizer = optim.Adam(  
    net.parameters(), lr=0.001  
)  
  
for epoch in range(num_epochs):  
    for seqs, labels in dataloader_train:  
        seqs = seqs.view(32, 96, 1)  
        outputs = net(seqs)  
        loss = criterion(outputs, labels)  
        optimizer.zero_grad()  
        loss.backward()  
        optimizer.step()
```

- Instantiate model, define loss & optimizer
- Iterate over epochs and data batches
- Reshape input sequence
- The rest: as usual

# Evaluation loop

```
mse = torchmetrics.MeanSquaredError()

net.eval()
with torch.no_grad():
    for seqs, labels in test_loader:
        seqs = seqs.view(32, 96, 1)
        outputs = net(seqs).squeeze()
        mse(outputs, labels)

print(f"Test MSE: {mse.compute()}")
```

Test MSE: 0.13292162120342255

- Set up MSE metric
- Iterate through test data with no gradients
- Reshape model inputs
- Squeeze model outputs
- Update the metric
- Compute final metric value

# LSTM vs. GRU

- LSTM:

```
Test MSE: 0.13292162120342255
```

- GRU:

```
Test MSE: 0.12187089771032333
```

- GRU preferred: same or better results with less processing power

# **Let's practice!**

**INTERMEDIATE DEEP LEARNING WITH PYTORCH**