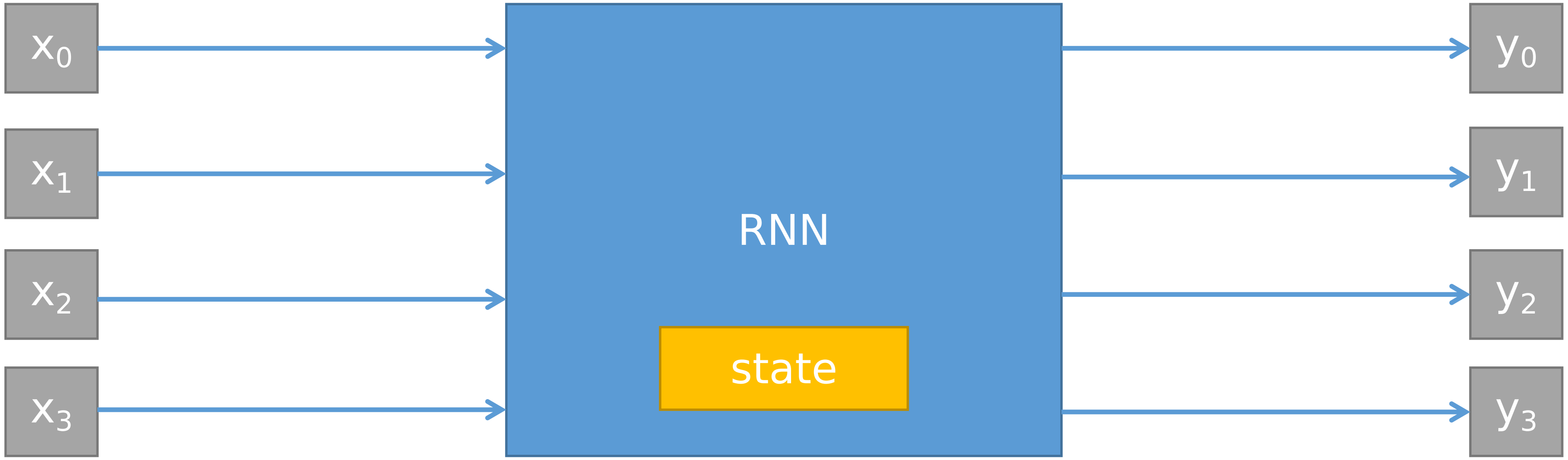
RNN

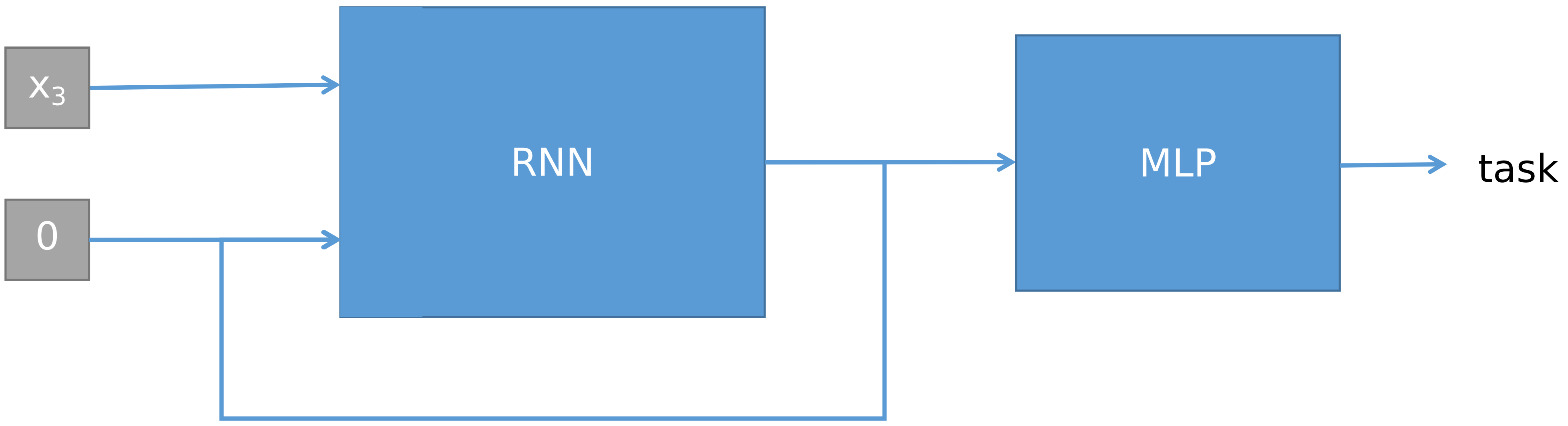
Recursive Neural Networks

A **Recursive Neural Network** is a kind of neural network **created** by **applying** the **same** set of **weights recursively** over a **time-varying input**, to **produce** a **time-varying output** over **variable-length input** structures.



Recursive Neural Networks: how to build a simple one

Input sequence: x_0 x_1 x_2 x_3



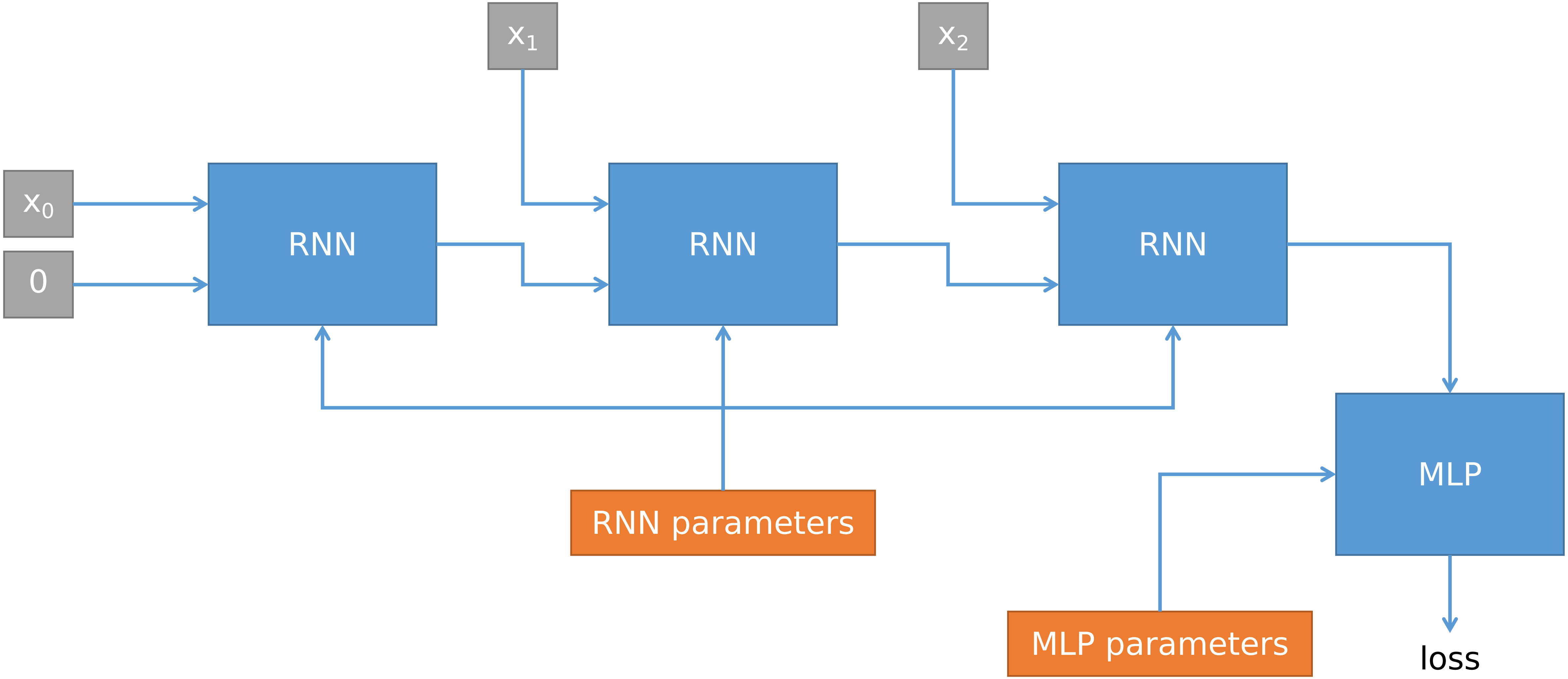
Recursive Neural Networks: Applications

Recursive Neural Networks (RNNs) find **applications** in various fields, leveraging their ability to **process hierarchical** and **time-varying** data. The key applications can be **diveded** into two **major families**:

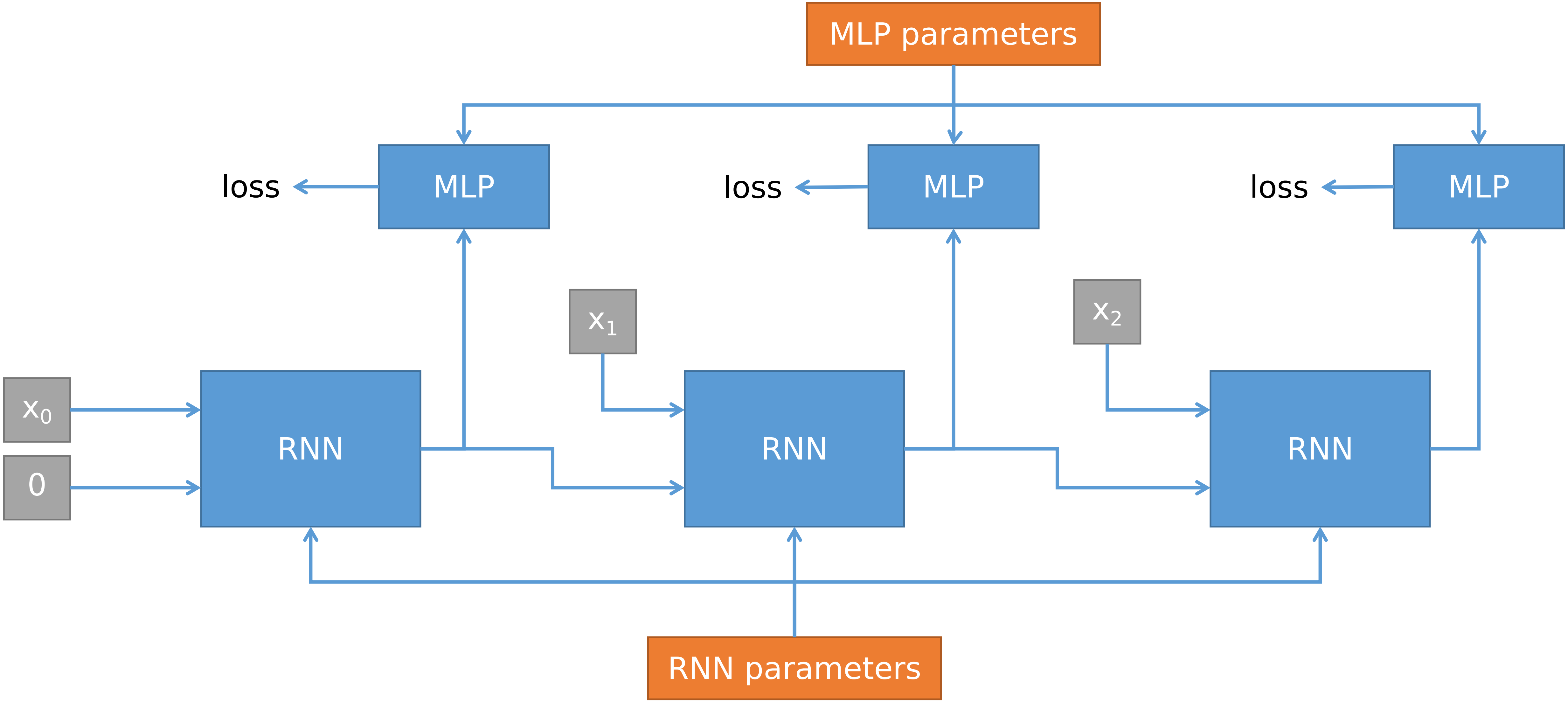
- **Sequence to Task**
  - Sequence-to-task models **process sequential input** data and **accomplish** specific **tasks**, such as **classification** or **regression**, by capturing temporal dependencies and patterns within the sequence.

- **Sequence to Sequence**
  - Sequence-to-sequence (**seq2seq**) models are designed to handle **input** and **output sequences** of **varying lengths**. Seq2seq models are applied in machine translation, text summarization, speech-to-text conversion, etc...

Recursive Neural Networks: Computational Graph (end loss)



Recursive Neural Networks: Computational Graph (step loss)

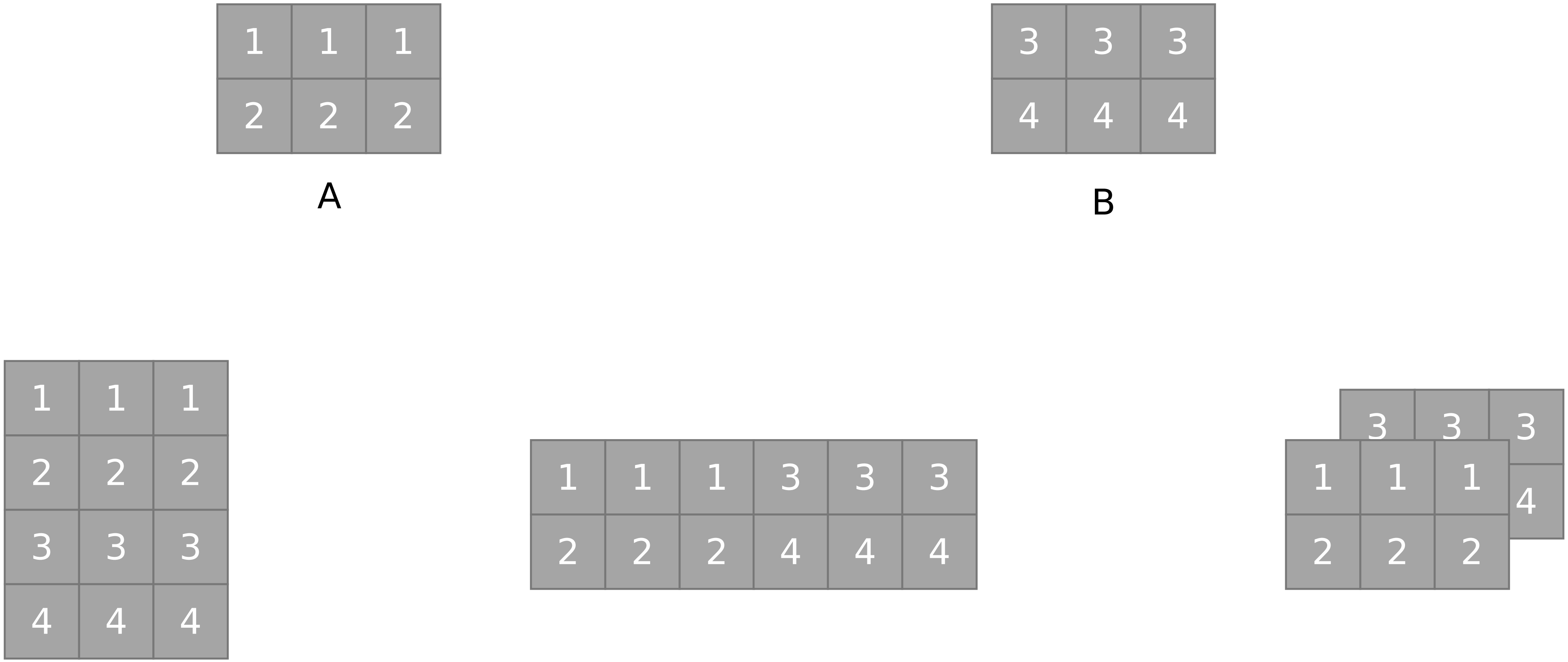


PyTorch: Concat and Stack

In PyTorch, **concat** and **stack** are **functions** used for **combining** tensors **along different dimensions**.

- **Concatenation** (**torch.concat)**:
  - This function is used to **concatenate** tensors **along** a specified **dimension**. For example, if you have two **tensors A** and **B** with the **same size** along **all dimensions except** for a **specific dimension**, you can **concatenate** them along **that** dimension using **torch.cat([A, B], dim)**.
- **Stacking (torch.stack**):
  - Stacking is **similar** to concatenation but **creates** a **new dimension** for the stacked tensors. If you have two **tensors A** and **B**, stacking them with **torch.stack([A, B], dim)** will **create** a **new dimension** along the specified dimension and stack them along that.

PyTorch: Concat and Stack



torch.concat((A, B), dim=0)

torch.concat((A, B), dim=1)

torch.stack((A, B), dim=0)

PyTorch: Concat and Stack

```
import torch

a = torch.tensor([[1,1,1], [2,2,2], [3,3,3],])
b = torch.tensor([[4,4,4], [5,5,5], [6,6,6]])

print(torch.concat((a,b),dim=0))
print(torch.concat((a,b),dim=1))
```

```
tensor([[1, 1, 1],
        [2, 2, 2],
        [3, 3, 3],
        [4, 4, 4],
        [5, 5, 5],
        [6, 6, 6]])
tensor([[1, 1, 1, 4, 4, 4],
        [2, 2, 2, 5, 5, 5],
        [3, 3, 3, 6, 6, 6]])
```

```
print(torch.stack((a,b),dim=0).shape)
print(torch.stack((a,b),dim=1).shape)
print(torch.stack((a,b),dim=2).shape)
```

```
torch.Size([2, 3, 3])
torch.Size([3, 2, 3])
torch.Size([3, 3, 2])
```

Simple RNN in PyTorch (end loss)

```
import torch

n_inputs = 10
n_hidden = 30
n_output = 5

rnn = torch.nn.Sequential(
    torch.nn.Linear(n_inputs + n_hidden, 100),
    torch.nn.LeakyReLU(),
    torch.nn.Linear(100, 100),
    torch.nn.LeakyReLU(),
    torch.nn.Linear(100, n_hidden)
)

mlp = torch.nn.Sequential(
    torch.nn.Linear(n_hidden, 100),
    torch.nn.LeakyReLU(),
    torch.nn.Linear(100, n_output)
)
```

```
input_sequence = torch.randn((5, 10))
target = torch.randn(5)

current_hidden = torch.zeros(30)
for t in range(0, 5):
  current_input = input_sequence[t]
  network_input = torch.concat(
    (current_input, current_hidden),
    dim=0
  )
  current_hidden = rnn(network_input)

loss = loss_fn(mlp(current_hidden), target)
```

Simple RNN in PyTorch (step loss)

```
input_sequence = torch.randn((5, 10))
targets = torch.zeros(5, 5)

current_hidden = torch.zeros(30)
losses = []
for t in range(0, 5):
  current_input = input_sequence[t]
  current_target = targets[t]
  network_input = torch.concat(
    (current_input, current_hidden),
    dim=0
  )

  current_hidden = rnn(network_input)
  current_output = mlp(current_hidden)

  losses.append(loss_fn(current_output, current_target))

losses = torch.stack(losses)
loss = losses.mean()
```

Simple batched RNN in PyTorch (out on last state)

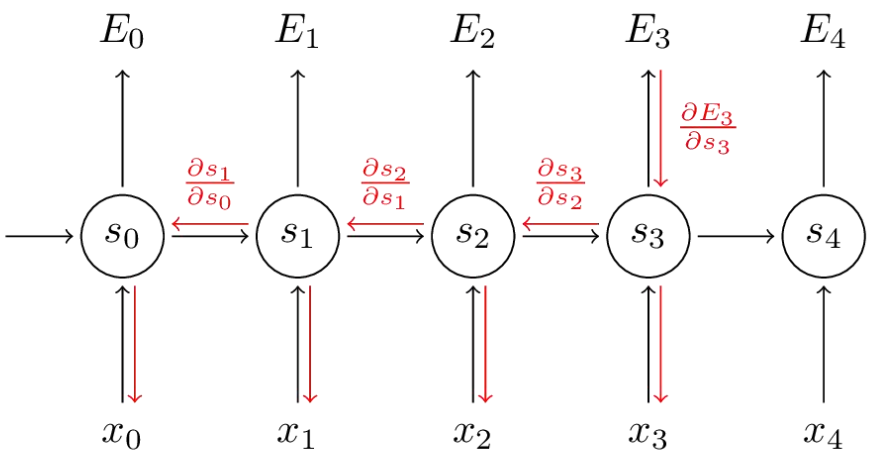
```
class RNNModel(torch.nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        self.rnn = torch.nn.Sequential(
            torch.nn.Linear(input_size + hidden_size, hidden_size),
            torch.nn.LeakyReLU(),
            torch.nn.Linear(hidden_size, hidden_size),
            torch.nn.LeakyReLU(),
            torch.nn.Linear(hidden_size, hidden_size),
        )
        self.ff = torch.nn.Linear(hidden_size, output_size)

    def forward(self, x):
        hidden_state = torch.zeros(x.shape[0], self.hidden_size, device=x.device)
        for t in range(x.shape[1]):
            input = torch.cat([x[:, t, :], hidden_state], dim=1)
            hidden_state = self.rnn(input)
        return self.ff(hidden_state)
```

Vanishing gradient problem in RNNs

RNNs are prone to the **vanishing gradient** problem, **hindering** their **ability** to **learn** from **long sequences**.

This occurs when **gradients** become **extremely small** during backpropagation through time, **especially** in **long sequences**.

Factors like repeated application of weight matrices, certain activation functions, and poor initialization contribute to this issue.



Vanishing gradient problem mitigation

The **vanishing gradient problem** in RNN can be **mitigated** but **cannot** be completely **solved.**

There are **several architectures** that tries to **mitigate** it **by selectively retain** and **forget information** during sequential processing.

In the next slides we will delve deeper into:

- Long-Short Term Memory (LSTM) networks
- Gated Recurrent Unit (GRU) networks

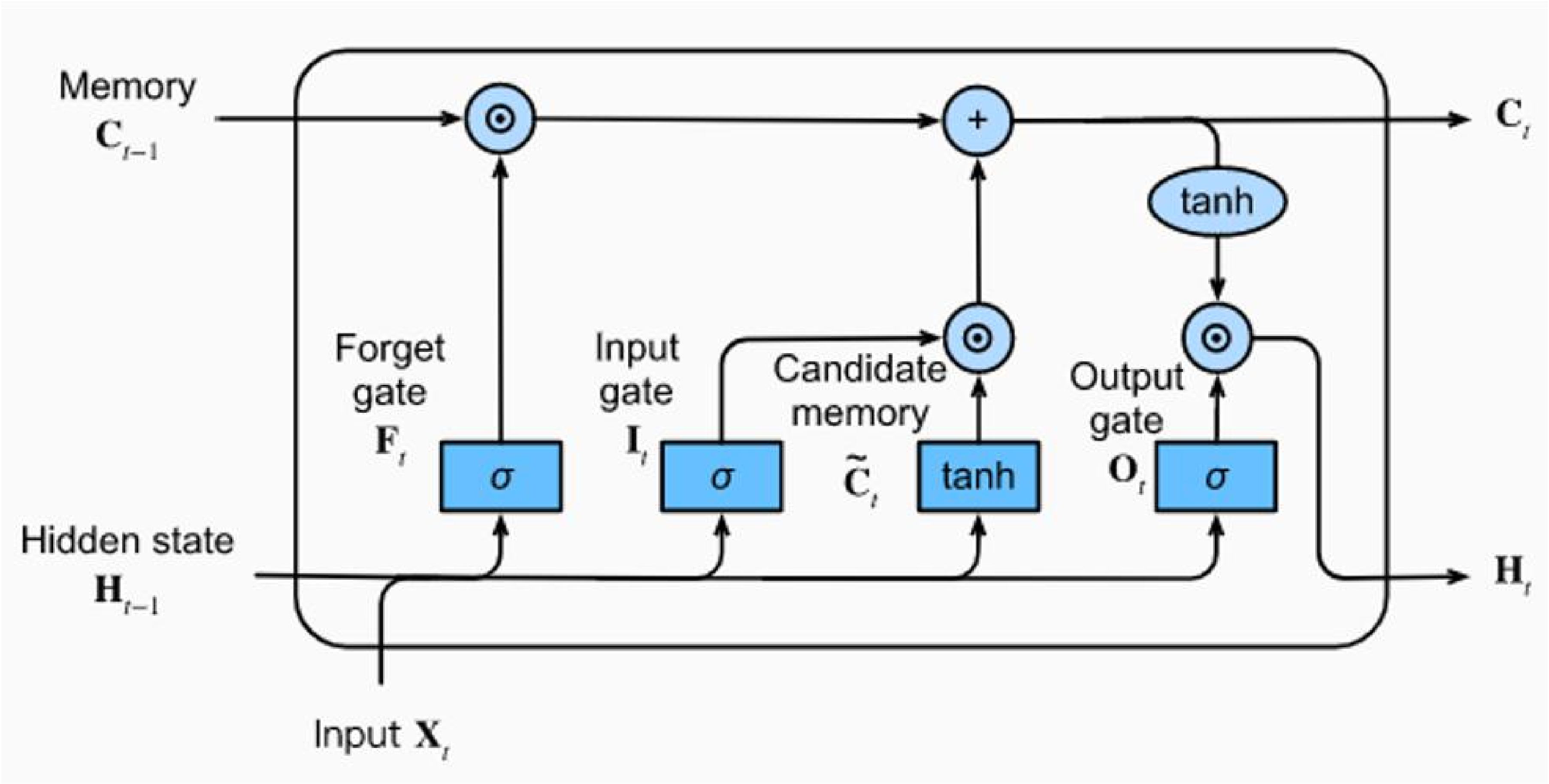
Long-Short Term Memory (LSTM)

Long Short-Term Memory (LSTM) is a **type** of **recurrent neural network** (RNN) architecture **designed** to **address** the **vanishing** gradient **problem**.

LSTMs **introduce** a **memory cell** with a **gating** mechanism, allowing it to **selectively store**, **read**, and **erase information** over time.

input gate	$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi})$
forget gate	$f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf})$
cell gate	$g_t = \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg})$
output gate	$o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho})$
next memory	$c_t = f_t \odot c_{t-1} + i_t \odot g_t$
next hidden	$h_t = o_t \odot \tanh(c_t)$

Long-Short Term Memory (LSTM)



Gate Recurrent Unit (GRU)

The **Gated Recurrent Unit** (GRU) is also a type of **recurrent neural network** (RNN) architecture designed to **address** the **vanishing** gradient **problem**.

Similar to Long Short-Term Memory (LSTM), **GRUs** utilize **gating** mechanisms, but they have a **simpler** structure with **three gates**: the **reset** gate, the **update** gate and the **new** gate.

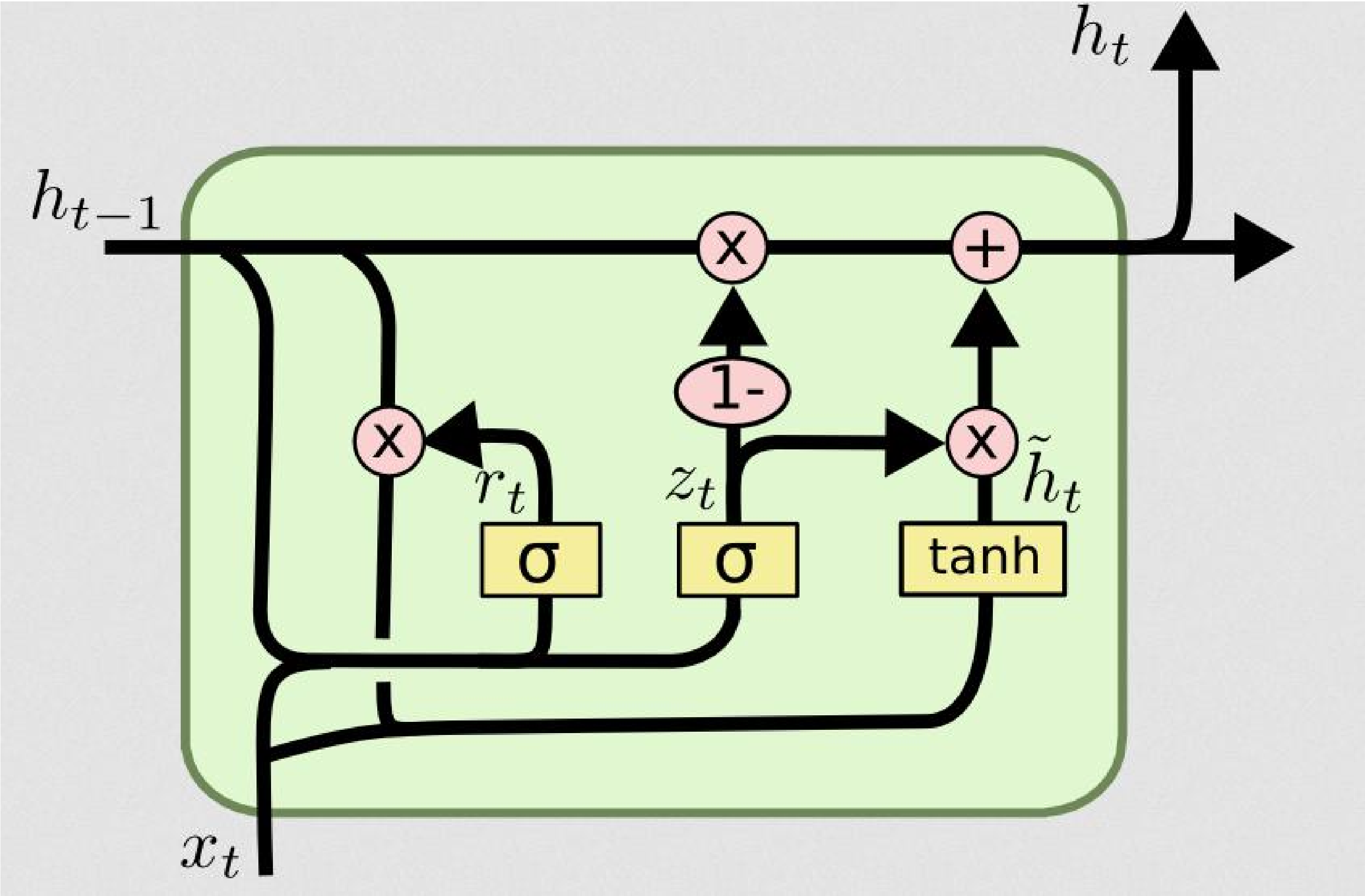
reset gate $$r_t = \sigma(W_{ir}x_t + b_{ir} + W_{hr}h_{(t-1)} + b_{hr})$$

update gate $$z_t = \sigma(W_{iz}x_t + b_{iz} + W_{hz}h_{(t-1)} + b_{hz})$$

new gate $$n_t = \tanh(W_{in}x_t + b_{in} + r_t \odot (W_{hn}h_{(t-1)} + b_{hn}))$$

next hidden $$h_t = (1 - z_t) \odot n_t + z_t \odot h_{(t-1)}$$

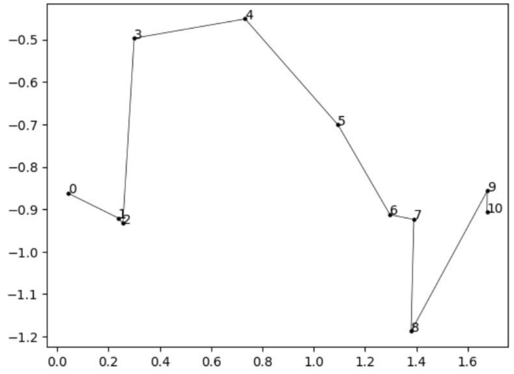
Gate Recurrent Unit (GRU)

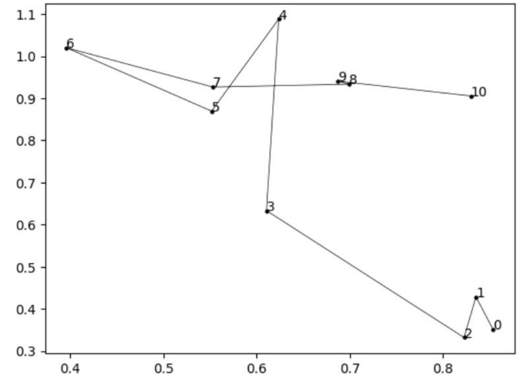


Exercise 1: RNN on custom dataset

Train this RNN to solve the sequence classification problem

```
self.rnn = torch.nn.Sequential(
    torch.nn.Linear(input_size + hidden_size, hidden_size),
    torch.nn.LeakyReLU(),
    torch.nn.Linear(hidden_size, hidden_size),
    torch.nn.LeakyReLU(),
    torch.nn.Linear(hidden_size, hidden_size),
)
self.ff = torch.nn.Linear(hidden_size, output_size)
```





Exercise 2: LSTM on custom dataset

Train a LSTM to solve the same sequence classification problem

```
self.lstm = torch.nn.LSTM(input_size, hidden_size, num_layers=3, batch_first=True)
self.ff = torch.nn.Linear(hidden_size, output_size)
```

Exercise 3: Compare RNN LSTM and GRU on custom dataset

Compare a RNN, LSTM and GRU on the same dataset

- All the recursive networks should have:
  - 300 hidden neurons
  - 3 layers
- The final classification task should be solved using a single linear layer 300 => 2
- Use same epochs, batch size and learning rate